Oracle

Программирование на языке Java

Мартин К. Соломон
Нирва Мориссо-Леруа
Джули Басу

Издательство "ЛОРИ"

Oracle Java Component Programming with EJB, CORBA and JSP
Martin K. Solomon
Nirva Morriseau-Leroy
Julie Basu
Copyright
All rights reserved

Oracle Программирование на языке Java
Мартин К. Соломон
Нирва Мориссо-Леруа
Джули Басу

Переводчик И.Драпишников
Научный редактор А.Головко
Корректоры Т.Килимник, Н.Литвинова
Верстка Л. Федякиной

Copyright © by The McGraw-Hill Companies. All rights reserved.
Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, or stored in a database or retrieval system without the prior written permission.
ISBN 0-07-212737-6

© Издательство "ЛОРИ", 2010

Изд. N : OAI (03)
ЛР N : 070612 30.09.97 г.
ISBN 978-5-85582-276-2

Подписано в печать 01.01.2010 Формат 84x108/16
Бумага офсет N1 Гарнитура Нью-Баскервиль Печать офсетная
Печ.л. 48 Тираж 500
Цена договорная

Отпечатано в типографии ООО "Тиль-2004"
107023, Москва, ул. Электрозаводская, д. 21

Посвящаю книгу своему одиннадцатилетнему сыну Алену Мориссо-Леруа и всем членам семьи, оказавшим мне неоценимую поддержку.

Нирва Мориссо-Леруа

Памяти моих родителей — Берты и Джозефа Соломон. Также посвящаю книгу жене Эбби, прекрасным дочерям Рейчел Смит и Мишель Дейзи Соломон и своему маленькому внуку Гарретту Александеру Смиту.

Мартин К. Соломон

В память о бабушке и дедушке. Также посвящаю книгу родителям Смрити Кана и Санджибу Басу, которые всегда учили меня не забывать свою мечту.

Джули Басу

Об авторах

Нирва Морисco-Леруа, обладательница степени MSCS, является администратором баз данных Oracle8*i* и разработчиком приложений в Национальном административном университете океана и атмосферы (National Oceanic and Atmospheric Administration-University) Объединенного научного института моря и атмосферы Майами, связанном с отделом исследования ураганов (HRD – Hurricane Research Division) AOML. Она имеет более чем 16-летний опыт работы с информационными системами в качестве администратора баз данных Oracle, разработчика приложений и руководителя MIS и владеет языками Java, JDBC, SQLJ, SQL, Oracle PL/SQL и UNIX C/C++. Основные направления ее деятельности – определение потребностей пользователей, объектно-ориентированные анализ и проектирование, моделирование данных, анализ, проектирование, реализация реляционных и объектно-ориентированных БД и управление ими. Помимо этого, Морисco-Леруа в совершенстве владеет тремя языками (французским, испанским и английским) и удостоена награды "Best Java Implementation" от HPCC (High Performance Computing and Communications) NOAA, полученной на NOAATech2000 в СилверСпринг (штат Мэриленд) 21 октября 1999 г. Обратиться к автору можно по адресу: nmorisseauleroy@data-i.com.

Доктор Мартин К. Соломон является профессором кафедры разработки и изучения компьютерных технологий в Атлантическом университете Флориды. Основные аспекты его исследований: проектирование, реализация и теория систем баз данных, теория сложных вычислений и философские аспекты вычислимости. Статьи по этим темам доктор Соломон публиковал в таких престижных журналах, как "ACM Transactions on Database Systems", "Communications of the ACM", "Journal of Symbolic Logic" и "British Journal for the Philosophy of Science".

Его профессиональные интересы затрагивают все стороны работы РСУБД Oracle, он регулярно ведет аттестационные курсы Oracle и часто публикуется в бюллетене "South Florida Oracle Users Group Newsletter". Обратиться к доктору Соломону можно по адресу: marty@cse.fau.edu.

Джули Басу, доктор философии, возглавляет группу платформы Java в корпорации Oracle и в настоящее время готовит проект OracleJSP. Исследованиями и проектированием Oracle она занимается более 10 лет. Выпустила не один проект, связанный с интерфейсами программирования баз данных, финансовыми приложениями и кэшированием. Среди ее нынешних интересов web-приложения, XML и электронная коммерция.

Джули опубликовала несколько статей на конференциях и в журналах для баз данных – подробнее об этом см. домашнюю интернет-страницу автора: www.jbasu.com. Ее адрес электронной почты: julie@cs.stanford.edu.

Предисловие

Наше время не случайно называется информационным веком. Сегодня простым щелчком мышью в web-браузере можно без труда обращаться к огромным объемам информации, распределенной по всей планете. Сердце информационной революции – Интернет. Он соединяет различные сети, территориально отдаленные друг от друга, значительно расширяя границы взаимодействия отдельных людей и коммерческих организаций. За последние несколько лет электронная коммерция на основе Интернета вступила в бурный рост. Товары продаются через Web, а покупатели напрямую общаются с продавцами. Прикладные программы самообслуживания, подобные тем, что позволяют выполнять банковские операции, не выходя из дома, значительно оптимизировали бизнес. Работа подавляющего большинства приложений электронной коммерции обеспечивается базами данных Oracle, обладающими высоким уровнем масштабируемости, надежности и доступности, а также отличной производительностью с точки зрения хранения и выбора данных с помощью языка SQL.

Бум в электронной коммерции подстегнул разработку нового поколения приложений уровня предприятия, с помощью которых взаимодействие происходит по линии "организация-организация" или "организация-покупатель". Основным языком для построения подобных приложений стал Java. Причины его популярности: современная объектно-ориентированная модель программирования, переносимость среди различных платформ и безопасность операций с памятью. Кроме того, он предоставляет развитую структуру вспомогательных библиотек и инструментов. К примеру, в состав версии Java 2 Enterprise Edition входит средство Enterprise JavaBeans (EJBs), позволяющее строить защищенные и масштабируемые программные компоненты для бизнес-транзакций, а также сервлеты Java (Java Servlets) и страницы JavaServer Pages (JSPs), с помощью которых создаются динамические web-страницы. Такие ранее популярные модели распределенного программирования, как общая архитектура посредника объектных запросов (CORBA – Common Object Request Broker Architecture), также нашли отражение в Java, с помощью которого можно реализовывать компоненты CORBA. Знание основанных на Java прикладных программных интерфейсов и инструментов программирования крайне важно для разработчика Интернет-приложений.

В Oracle сразу осознали важность модели Интернет-вычислений и вложили в Java значительные средства. В состав Интернет-РСУБД Oracle8*i* входит виртуальная машина Java (Java Virtual Machine) с высоким уровнем масштабируемости под названием Oracle8*i* Jserver: она способна выполнять программы Java непосредственно в БД. JServer обеспечивает полнофункциональную поддержку Java 2 Enterprise Edition. Бизнес-компоненты Enterprise JavaBeans и CORBA выполняются в защищенном режиме в границах JServer, т.е. в непосредственной близости от бизнес-данных. Прикладной программный интерфейс (API) JDBC (Java Database Connectivity) и язык SQLJ (SQL, встроенный в Java) стандарта ANSI/ISO доступны не только на станции клиента, но и в JServer, что ускоряет обращение к SQL-данным в процессе работы. Вместе Oracle8*i* и Java образуют самую мощную серверную платформу для развертывания защищенных приложений электронной коммерции, способных обслуживать сотни клиентов одновременно.

Это именно та книга, которая позволит быстро и эффективно строить масштабируемые Интернет-приложения с использованием Java и Oracle. В первой части рассматривается стиль программирования на основе компонентов – очень эффективный метод написания модульных и удобных в обслуживании приложений, а также дается общий обзор процесса разработки приложений с помощью компонентов EJB, CORBA и JSP. В трех следующих подробно рассказано о EJB, CORBA и JSP. В начале каждой части дано краткое общее представление о том или ином компоненте, затем рассматриваются более частные вопросы программирования. Написана книга отлично,

и переходить от главы к главе очень просто, к тому же теория поясняется многочисленными примерами. Взятые из современной практики, они показывают, как с помощью компонентов Java грамотно разрабатывать приложения баз данных уровня предприятия в пошаговом режиме.

Меня поразила широта и глубина тем, раскрываемых в этой книге, а также превосходная организация материала. Могу сказать, что поистине восхищен ею. Уверен, что она послужит эффективным и по-настоящему незаменимым руководством тем, кто собирается разрабатывать приложения уровня предприятия с использованием Oracle8*i* и Java.

Дэвид А. Розенберг,
вице-президент группы платформы Java
корпорации Oracle

Благодарности

Заслуга в создании нашей книги принадлежит многим людям, которым я хочу выразить особую признательность за поддержку, советы и отклики, полученные нами во время работы над этим проектом. Прежде всего Эдварду О. Гриффину – техническому руководителю из корпорации Oracle: он помог мне в работе с новыми версиями Oracle8*i* (1.6 и 1.7). А также Хосе Альберто Фернандесу, Брейдену Н. Макдэниелу и своему бывшему соавтору Джеральду П. Момплезира – за превосходные технические замечания и предложения. Большое спасибо соавторам, Мартину К. Соломону и Джули Басу. Хочу поблагодарить доктора Марка Д. Пауэлла, эксперта в области атмосферы, и его группу H*WIND отдела исследования ураганов NOAA, доктора Джозефа Просперо из Университета Майами, Джереми Джадсона, старшего редактора комплектования, Монику Фэлтисс, Памелу Вулф, Денниса Уивера и Росса Долла из редакции Osborne/McGraw-Hill и группу серверных технологий/разработки SQLJ корпорации Oracle.

Нирва Мориссо-Леруа
Майами, Флорида
Июль 2000 г.

Весьма благодарен Арчану Дюби и Цзюки Ли за помощь в тестировании программ, а также нашему техническому редактору Хосе Альберто Фернандесу за полезные предложения и рекомендации. Отдельное спасибо Джереми Джадсону (за его неоценимую поддержку), Памеле Вулф, Россу Доллу, Монике Фэлтисс и Деннису Уиверу, членам редакции Osborne/McGraw-Hill. Выражаю признательность жене Эбби за помощь в наборе текста и соавторам Нирве Мориссо-Леруа и Джули Басу за приятное и плодотворное сотрудничество. Наконец, большое спасибо замечательным системным администраторам: Й. Серджу Джозефу, Манешу Нилаканте, Артуру Соузе, Марчело Мартинсу и Натану Картеру.

Мартин К. Соломон
Бока-Ратон, Флорида
Июль 2000 г.

Спешу поблагодарить семью и друзей за чуткость и поддержку. Особенно Индрани и Субхенду Чаудхури, деливших со мной хлеб-соль, а также сестру Джун Мухерджи и подругу Эне Сингх. Я благодарна Хосе Альберто Фернандесу, Алексу Ю и Брайану Райту из корпорации Oracle за глубокие технические рецензии и полезные рекомендации. Огромную помощь оказала мне редакция Osborne/McGraw-Hill: Джереми Джадсон, Памела Вулф, Росс Долл и Моника Фэлтисс. Было приятно работать с соавторами Нирвой Мориссо-Леруа и Мартином К. Соломоном. Хочу особо поблагодарить Нирву за приглашение принять участие в этом проекте, умение вести за собой и постоянную великодушную помощь. Наконец, все мы глубоко признательны Томасу Кьюриану и Дэвиду Розенбергу из корпорации Oracle за финансирование нашей работы.

Джули Басу
Сан-Матео, Калифорния
Июль 2000 г.

Введение

Главная тема этой книги – разработка приложений баз данных на основе компонентов Java с использованием Enterprise JavaBeans (EJBs), CORBA и JavaServer Pages (JSPs). В частности, вы научитесь разрабатывать, внедрять и настраивать многоуровневые приложения БД с помощью реализаций JDBC и SQLJ. Несмотря на то что прикладные программы, здесь представленные, были созданы для работы в базах данных Oracle версий 8.xx и выше, с небольшими изменениями их можно использовать в любой основанной на SQL, т.е. "чистой" реляционной или объектно-реляционной системе управления БД (DBMS – Database Management System).

Схемы баз данных

В центре нашего внимания будет создание приложений БД, работающих со схемами "чистой" реляционной и объектно-реляционной баз данных. "Purchase Order" (заказ на покупку) – это схема финансовой реляционной БД. Она является частью проекта базы данных, представленного в "Design of a Financial Administrative System Using the Semantic Binary Model" (проект финансовой административной системы с использованием семантической двоичной модели). "Scientific Observation" (научные наблюдения) – это схема объектно-реляционной БД. Она является частью проекта научной базы данных, представленного в "Atmospheric Observations, Analyses, and The World Wide Web Using a Semantic Database" (наблюдение за атмосферой, анализ и World Wide Web с использованием семантической БД). Обе схемы представлены в книге "Oracle8*i* SQL Programming", OMH, 1999.

Схемы Purchase Order и Observation были специально созданы для Атлантической океанографической и метеорологической лаборатории (AOML, Atlantic Oceanographic and Meteorological Laboratory) в Майами (штат Флорида) и для Лабораторий исследования окружающей среды (ERL, Environmental Research Laboratories) Национального управления океана и атмосферы (NOAA, National Oceanic and Atmospheric Administration), подразделения Министерства торговли США. В частности, схема Observation предназначалась отделу исследования ураганов (HRD – Hurricane Research Division) AOML и в настоящее время реализуется им как часть системы H*WIND, проводящей анализ ветра на поверхности воды в реальном времени для синоптиков Национального центра слежения за ураганами (NHC, National Hurricane Center) и группы связи (FEMA Hurricane Liaison Team) NHC.

SQL-сценарии создания финансовой схемы Purchase Order

При создании схемы Purchase Order в базе данных Oracle8*i* используйте такой SQL-сценарий `createposchema.sql`:

```
--
-- Имя файла: poexample.sql
--
-- Сценарием создаются все таблицы, ограничения и последовательности,
-- необходимые для примера Purchase Order.
--
-- В сценарии предполагается, что созданы пользователь БД и индекс с именем
-- INDX.
-- Убедитесь в том, что пользователю назначена временная табличная область.
--
-- Применение: sqlplus пользователь/пароль
-- SQL> @poexample.sql
```

```
--
set termout on
set echo on
spool poexample.log
-- Из-за ограничений внешних ключей сначала нужно удалить эти
-- таблицы.

DROP TABLE DEPARTMENT_LIST CASCADE CONSTRAINTS;
CREATE TABLE DEPARTMENT_LIST(
departmentno            NUMBER(5),
shortname     VARCHAR2(6),
longname      VARCHAR2(20));

DROP TABLE ACCOUNT_LIST CASCADE CONSTRAINTS;
CREATE TABLE ACCOUNT_LIST (
accountno NUMBER(5),
projectno NUMBER(5),
departmentno             NUMBER(5),
PRIMARY KEY ( accountno ));

DROP TABLE EMPLOYEE_LIST CASCADE CONSTRAINTS;
CREATE TABLE EMPLOYEE_LIST(
employeeno  NUMBER(7),
departmentno             NUMBER(5),
type          VARCHAR2(30),
lastname      VARCHAR2(30),
firstname     VARCHAR2(30),
phone         VARCHAR2(10));

DROP TABLE CREDITCARD_LIST;
CREATE TABLE CREDITCARD_LIST (
cardno           VARCHAR2(15),
employeeno NUMBER(7),
expirationdate   DATE);

DROP TABLE CHECKACCOUNT_LIST;
CREATE TABLE CHECKACCOUNT_LIST(
accountno     NUMBER(5),
employeeno    NUMBER(7));

DROP TABLE VENDOR_LIST;
CREATE TABLE VENDOR_LIST(
vendorno      NUMBER(6),
name          VARCHAR2(30),
address          VARCHAR2(20),
city          VARCHAR2(15),
state         VARCHAR2(15),
vzip          VARCHAR2(15),
country          VARCHAR2(15));

DROP TABLE PROJECT_LIST;
CREATE TABLE PROJECT_LIST (
projectno     NUMBER(5),
projectname      VARCHAR2(20),
start_date   DATE,
amt_of_funds     NUMBER,
PRIMARY KEY( projectno ));

DROP TABLE PURCHASE_LIST;
CREATE TABLE PURCHASE_LIST (
requestno     NUMBER(10),
employeeno    NUMBER(7),
vendorno      NUMBER(6),
purchasetype     VARCHAR2(20),
checkno       NUMBER(11),
whenpurchased    DATE);
```

```
DROP TABLE LINEITEM_LIST;
CREATE TABLE LINEITEM_LIST (
requestno     NUMBER(10),
lineno          NUMBER(5),
projectno     NUMBER(5),
quantity      NUMBER(5),
unit          VARCHAR2(2),
estimatedcost   NUMBER(8,2),
actualcost    NUMBER(8,2),
description   VARCHAR2(30));

--
-- Добавим ограничения к таблицам
--

ALTER TABLE DEPARTMENT_LIST
  ADD CONSTRAINT deptno_pk PRIMARY KEY(departmentno)
  USING INDEX TABLESPACE INDX;

ALTER TABLE ACCOUNT_LIST
  ADD CONSTRAINT acc_deptno_fk
  FOREIGN KEY(departmentno)
  REFERENCES DEPARTMENT_LIST(departmentno);

ALTER TABLE EMPLOYEE_LIST
  ADD CONSTRAINT employeeno_pk PRIMARY KEY(employeeno)
  USING INDEX TABLESPACE INDX;

ALTER TABLE EMPLOYEE_LIST
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY(departmentno)
  REFERENCES DEPARTMENT_LIST(departmentno);

ALTER TABLE CREDITCARD_LIST
  ADD CONSTRAINT cardno_pk PRIMARY KEY(cardno)
  USING INDEX TABLESPACE INDX;

ALTER TABLE CREDITCARD_LIST
  ADD CONSTRAINT credit_employeeno_fk FOREIGN KEY(employeeno)
  REFERENCES EMPLOYEE_LIST(employeeno);

ALTER TABLE CHECKACCOUNT_LIST
  ADD CONSTRAINT accountno_pk PRIMARY KEY(accountno)
  USING INDEX TABLESPACE INDX;

ALTER TABLE CHECKACCOUNT_LIST
  ADD CONSTRAINT check_employeeno_fk
  FOREIGN KEY(employeeno)
  REFERENCES EMPLOYEE_LIST(employeeno);

ALTER TABLE vendor_list
  ADD CONSTRAINT vendorno_pk PRIMARY KEY(vendorno)
  USING INDEX TABLESPACE INDX;

ALTER TABLE Purchase_list
  ADD CONSTRAINT requestno_pk PRIMARY KEY(requestno)
  USING INDEX TABLESPACE INDX;

ALTER TABLE LINEITEM_LIST
  ADD CONSTRAINT lineno_pk
  PRIMARY KEY(requestno,lineno,projectno)
  USING INDEX TABLESPACE INDX;

--
-- Создадим последовательности
--

DROP SEQUENCE deptno_SEQ;
CREATE SEQUENCE deptno_SEQ
```

```
  START WITH 200
  INCREMENT BY 1;

DROP SEQUENCE projectno_SEQ;
CREATE SEQUENCE projectno_SEQ
  START WITH 300
  INCREMENT BY 1;

DROP SEQUENCE employeeno_SEQ;
CREATE SEQUENCE employeeno_SEQ
  START WITH 100
  INCREMENT BY 1;

DROP SEQUENCE accountno_SEQ;
CREATE SEQUENCE accountno_SEQ
START WITH 1000
INCREMENT BY 1;

DROP SEQUENCE cardno_SEQ;
CREATE SEQUENCE cardno_SEQ
  START WITH 311200
  INCREMENT BY 1;

DROP SEQUENCE vendorno_SEQ;
CREATE SEQUENCE vendorno_SEQ
  START WITH 400
  INCREMENT BY 1;

DROP SEQUENCE requestno_SEQ;
CREATE SEQUENCE requestno_SEQ
  START WITH 500
  INCREMENT BY 1;

DROP SEQUENCE lineno_SEQ;
CREATE SEQUENCE lineno_SEQ
  START WITH 1
  INCREMENT BY 1;

spool off
```

SQL-сценарии создания научной схемы Observation

При создании схемы Observation в базе данных Oracle8*i* используйте такой SQL-сценарий createobjschema.sql:

```
--
-- Имя файла: obsexample.sql
--
-- Сценарием создаются все таблицы, ограничения и
-- последовательности, необходимые для примера Observation.

--
-- В сценарии предполагается, что созданы пользователь БД и индекс INDX.
--
-- Применение: sqlplus пользователь/пароль
-- SQL> @obsexample.sql
--
--
set termout on
set echo on
spool createobjschema.log

DROP TABLE PRODUCT_LIST;
DROP TYPE PRODUCT_TYPE;
DROP TABLE PASSED_OBSERVATION_LIST;
DROP TYPE PASSEDOBSARRAY;
DROP TYPE PASSEDOBS;
DROP TABLE OCEANIC_OBSERVATION_LIST;
```

```
DROP TYPE OCEANIC_OBSERVATION_TYPE;
DROP TYPE OCEANIC_OBSERVATION;
DROP TABLE QC_EVENT_LIST;
DROP TYPE QUALITY_CONTROL_EVENT;
DROP TABLE ATMOSEVENT_LIST;
DROP TYPE ATMOSEVENT;
DROP TABLE SCIENTIST_LIST
DROP TYPE SCIENTIST;
DROP TABLE PLATFORM_TYPE_LIST;
DROP TYPE PLATFORM_TYPE;

CREATE TYPE PLATFORM_TYPE AS    ECT(
key_id        NUMBER(8),
type          VARCHAR2(50),
description   VARCHAR2(50));

CREATE TABLE PLATFORM_TYPE_LIST OF PLATFORM_TYPE;

CREATE TYPE SCIENTIST AS OBJECT(
usr_id        NUMBER(6),
lastname      VARCHAR2(20),
firstname     VARCHAR2(20),
platform_id   NUMBER,
for_platform  REF PLATFORM_TYPE);

CREATE TABLE SCIENTIST_LIST OF SCIENTIST;

CREATE TYPE ATMOSEVENT AS OBJECT(
key_id        NUMBER(8),
when_t        DATE,
name    VARCHAR2(30),
type    VARCHAR2(20),
refkey         NUMBER(8),
transformed_to REF atmosevent);

CREATE TABLE ATMOSEVENT_LIST OF ATMOSEVENT;

CREATE TYPE OCEANIC_OBSERVATION AS OBJECT(
latitude_deg        NUMBER(10,4),
longitude_deg       NUMBER(10,4),
windspeed_mps       NUMBER(10,4),
adj_windspeed_mps   NUMBER(10,4),
wind_direction_deg  NUMBER(6),
pressure_mb NUMBER(6))

CREATE TYPE OCEANIC_OBSERVATION_TYPE AS OBJECT(
obs_id        NUMBER(8),
when_t        DATE,
at_time       CHAR(8),
station_id    NUMBER(6),
produced_id   NUMBER(8),
produced_by   REF PLATFORM_TYPE,
obsobj        OCEANIC_OBSERVATION);

-- Список всех наблюдений за океаном по дате, времени и типу
-- платформы.
CREATE TABLE OCEANIC_OBSERVATION_LIST
  OF OCEANIC_OBSERVATION_TYPE;

-- Воспользуемся qc_id_seq для обновления QUALITY_CONTROL_EVENT qc_id
CREATE TYPE QUALITY_CONTROL_EVENT AS OBJECT(
qc_id   NUMBER(8),
when_t  DATE,
at_time CHAR(8),
event_id      NUMBER(8),
for_event     REF atmosevent,
whom_id NUMBER(6),
by_whom REF scientist);
```

```
CREATE TABLE QC_EVENT_LIST OF QUALITY_CONTROL_EVENT;

CREATE OR REPLACE TYPE PASSEDOBS AS OBJECT(
obsid   NUMBER(8),
when_t  DATE);

CREATE TYPE PASSEDOBSARRAY AS TABLE OF PASSEDOBS;

CREATE TABLE PASSED_OBSERVATION_LIST(
qcid    NUMBER(8),
when_t  DATE,
at_time CHAR(8),
idobj   passedObsArray)
NESTED TABLE idobj STORE AS pobsid_list;

ALTER TABLE POBSID_LIST
  STORAGE (MINEXTENTS 1 MAXEXTENTS 20);

--
-- Теперь создадим ограничения
--

ALTER TABLE PLATFORM_TYPE_LIST
  ADD CONSTRAINT PT_KEY_ID_PK PRIMARY KEY(KEY_ID)
  USING INDEX TABLESPACE INDX;

ALTER TABLE SCIENTIST_LIST
  ADD CONSTRAINT SL_USR_ID_PK PRIMARY KEY(USR_ID)
  USING INDEX TABLESPACE INDX;

ALTER TABLE ATMOSEVENT_LIST
  ADD CONSTRAINT AL_KEY_ID_PK PRIMARY KEY(KEY_ID)
  USING INDEX TABLESPACE INDX;

ALTER TABLE OCEANIC_OBSERVATION_LIST
  ADD CONSTRAINT O_OBS_ID_PK PRIMARY KEY(OBS_ID)
  USING INDEX TABLESPACE INDX;

ALTER TABLE QC_EVENT_LIST
  ADD CONSTRAINT QC_ID_PK PRIMARY KEY(QC_ID)
  USING INDEX TABLESPACE INDX;

ALTER TABLE QC_EVENT_LIST
  ADD CONSTRAINT qc_whom_id_fk
  FOREIGN KEY(whom_id)
  REFERENCES SCIENTIST_LIST(usr_id)
  ON DELETE CASCADE;

ALTER TABLE PASSED_OBSERVATION_LIST
  ADD CONSTRAINT passed_id_pk PRIMARY KEY (passed_id)
  USING INDEX TABLESPACE INDX;

ALTER TABLE PASSED_OBSERVATION_LIST
  ADD CONSTRAINT po_qc_id_fk
  FOREIGN KEY(qcid)
  REFERENCES QC_EVENT_LIST(qc_id)
  ON DELETE CASCADE;

ALTER TABLE PASSED_OBSERVATION_LIST
  ADD CONSTRAINT passed_qcid_ukey UNIQUE(qcid)
  USING INDEX TABLESPACE INDX;

ALTER TABLE PASSED_OBSERVATION_LIST
  MODIFY (qcid NOT NULL);

ALTER TABLE PRODUCT_LIST
  ADD CONSTRAINT pr_id_pk PRIMARY KEY (id)
  USING INDEX TABLESPACE INDX;

--
-- Теперь создадим последовательности
```

```
--
-- последовательность key_id для PLATFORM_TYPE
DROP SEQUENCE PT_key_SEQ;
CREATE SEQUENCE PT_key_SEQ
START WITH 1
INCREMENT BY 1;

-- последовательность usr_id для SCIENTIST
DROP SEQUENCE USERSEQ;
CREATE SEQUENCE USERSEQ
  START WITH 1
  INCREMENT BY 1;

-- последовательность key_id для ATMOSEVENT
DROP SEQUENCE atm_key_seq;
CREATE SEQUENCE atm_key_seq
  START WITH 1
  INCREMENT BY 1;

DROP SEQUENCE OBSID_SEQ;
CREATE SEQUENCE OBSID_SEQ
  START WITH 1
  INCREMENT BY 1;

-- последовательность qc_id для QUALITY_CONTROL_EVENT
DROP SEQUENCE qc_id_seq;
CREATE SEQUENCE qc_id_seq
  START WITH 1
  INCREMENT BY 1;

-- последовательность passed_id для PASSED_OBSERVATION
DROP SEQUENCE passed_id_seq;
CREATE SEQUENCE passed_id_seq
  START WITH 1
  INCREMENT BY 1;

--
-- Теперь создадим модуль (пакет) PL/SQL
--
DROP PACKAGE OBSACTIONS;
CREATE OR REPLACE PACKAGE OBSACTIONS AS
-- Получим новый Obs_id
FUNCTION getObsId RETURN NUMBER;
PRAGMA RESTRICT_REFERENCES(getObsId,WNDS,WNPS,RNPS);

-- Добавим новое наблюдение
   PROCEDURE insertObs(p_newobsid IN NUMBER,
                       p_whent          IN VARCHAR2,
                       p_attime         IN CHAR,
                       p_ptlid          IN NUMBER,
                       p_latdeg         IN NUMBER,
                       p_londeg         IN NUMBER,
                       p_wsmps          IN NUMBER,
                       p_adjwsmps    IN NUMBER,
                       p_wddeg          IN NUMBER,
                       p_pmb         IN NUMBER,
                       p_stlid          IN NUMBER);

END OBSACTIONS;
/
CREATE OR REPLACE PACKAGE BODY OBSACTIONS AS
-- Воспользуемся частной (private) функцией FindCurrTime, чтобы
-- вычислить текущее время. Этот метод доступен только для
-- функций и процедур, указанных в разделе тела модуля OBSACTIONS
-- и аналогичен частному методу Java.
   FUNCTION FindCurrTime RETURN CHAR IS
   v_time CHAR(8);
```

```
   BEGIN
        SELECT TO_CHAR(SYSDATE,'HH24MISS') INTO v_time FROM DUAL;
        RETURN v_time;
   EXCEPTION
       WHEN NO_DATA_FOUND THEN
           RETURN v_time;
   END FindCurrTime;

-- Получим новый Obs_id
   FUNCTION getObsId RETURN NUMBER IS
     p_obsid NUMBER;
    BEGIN
        SELECT OBSID_SEQ.NEXTVAL INTO p_obsid FROM DUAL;
        RETURN p_obsid;
    EXCEPTION
        WHEN OTHERS THEN
             RETURN p_obsid;
   End getObsId;

-- Создадим объект типа Oceanic_observation

-- Это частный (private) метод.
   FUNCTION OceanicObservation (p_latdeg IN NUMBER,
                        p_londeg         IN NUMBER,
                        p_wsmps          IN NUMBER,
                        p_adjwsmps       IN NUMBER,
                        p_wddeg          IN NUMBER,
                        p_pmb            IN NUMBER)

                                  RETURN oceanic_observation IS
--       Создадим тип oceanic_observation.
         v_obs oceanic_observation :=
              oceanic_observation(p_latdeg,p_londeg,
                p_wsmps,p_adjwsmps,p_wddeg,p_pmb);
   BEGIN
        RETURN v_obs;
   END OceanicObservation;

-- Добавим новое наблюдение
   PROCEDURE insertObs(p_newobsid    IN NUMBER,
                       p_whent           IN VARCHAR2,
                       p_attime          IN CHAR,
                       p_ptlid           IN NUMBER,
                       p_latdeg          IN NUMBER,
                       p_londeg          IN NUMBER,
                       p_wsmps           IN NUMBER,
                       p_adjwsmps    IN NUMBER,
                       p_wddeg           IN NUMBER,
                       p_pmb         IN NUMBER,
                       p_stlid           IN NUMBER) IS

        v_date DATE := TO_DATE(p_whent,'DD-MON-YYYY');
        v_time CHAR(8) := p_attime;
        v_newobsid NUMBER := TO_NUMBER(p_newobsid);
        v_platformref REF PLATFORM_TYPE;

        v_obs Oceanic_Observation :=
             Oceanic_Observation(p_latdeg,p_londeg,
               p_wsmps,p_adjwsmps,p_wddeg,p_pmb);
  BEGIN
--   Проверим входную дату p_whent
        IF v_date IS NULL THEN
          SELECT SYSDATE INTO v_date FROM DUAL;
        END IF;

--      Проверим входное время
        IF v_time IS NULL THEN
```

```
            v_time := FindCurrTime;
        END IF;

--      Получим ссылку (REF) на platform_type
        SELECT REF(P) INTO v_platformref
            FROM platform_type_list P
            WHERE P.key_id = p_ptlid;

--      Вставим новый обьект
        INSERT INTO OCEANIC_OBSERVATION_LIST

            VALUES(
                v_newobsid, v_date, v_time, p_stlid, p_ptlid,
                v_platformref, v_obs
                );
        -- зафиксируем результат;
  EXCEPTION
        WHEN NO_DATA_FOUND THEN
            NULL;
        WHEN OTHERS THEN
            NULL;
  END insertObs;

END OBSACTIONS;

DROP PACKAGE QCACTIONS;
CREATE OR REPLACE PACKAGE QCACTIONS AS
--  Получим новый qc_id
    FUNCTION GETNEWQCID RETURN NUMBER;
    PRAGMA RESTRICT_REFERENCES(GETNEWQCID,WNDS,WNPS,RNPS);
--  Добавим новый QcSet
    PROCEDURE insertQcSet(p_newqcid IN NUMBER,
                          p_whent       IN VARCHAR2,
                          p_attime  IN CHAR,
                          p_evtid       IN NUMBER,
                          p_whom_id IN NUMBER);
--  Добавим новую строку в таблицу PASSED_OBSERVATION_LIST
    PROCEDURE insertQcSetObs(p_qcid IN NUMBER,
                             p_obsid IN NUMBER);
END QCACTIONS;
CREATE OR REPLACE PACKAGE BODY QCACTIONS AS
--  Выясним текущее время
    FUNCTION FindCurrTime RETURN CHAR IS
       v_time CHAR(8);
    BEGIN
            SELECT TO_CHAR(SYSDATE,'HH24MISS')
              INTO v_time FROM DUAL;
            RETURN v_time;
    EXCEPTION
            WHEN NO_DATA_FOUND THEN
              RETURN v_time;
    END FindCurrTime;

  FUNCTION GETNEWQCID RETURN NUMBER IS
       p_qcid NUMBER;
    BEGIN
       SELECT qc_id_seq.NEXTVAL INTO p_qcid FROM DUAL;
       RETURN p_qcid;
    EXCEPTION
      WHEN OTHERS THEN
        RETURN p_qcid;
    End GETNEWQCID;

  PROCEDURE insertQcSet(p_newqcid IN NUMBER,
       p_whent      IN VARCHAR2, p_attime IN CHAR,
       p_evtid      IN NUMBER, p_whom_id IN NUMBER) IS
```

```
    v_date DATE := TO_DATE(p_whent,'DD-MON-YYYY');
    v_time CHAR(8) := p_attime;
    v_newqcid NUMBER := TO_NUMBER(p_newqcid);
    v_eventref REF atmosevent;
    v_scientistref REF scientist;

  BEGIN
--    Проверим входную дату p_whent
      IF v_date IS NULL THEN
         SELECT SYSDATE INTO v_date FROM DUAL;
      END IF;
--    Проверим входное время
      IF v_time IS NULL THEN
         v_time := FindCurrTime;
      END IF;
--    Получим ссылку (REF) на atmosevent
      SELECT REF(P) INTO v_eventref
         FROM ATMOSEVENT_LIST P
         WHERE P.key_id = p_evtid;
--    Получим ссылку (REF) на scientist
      SELECT REF(P) INTO v_scientistref
         FROM SCIENTIST_LIST P
         WHERE P.usr_id = p_whom_id;
--    Вставим новую строку
      INSERT INTO QC_EVENT_LIST
        VALUES(p_newqcid,v_date,v_time,p_evtid,v_eventref,
               p_whom_id,v_scientistref);
      EXCEPTION
      WHEN OTHERS THEN
         RAISE NO_DATA_FOUND;
    END insertQcSet;
  PROCEDURE insertQcSetObs(p_qcid IN NUMBER,p_obsid IN NUMBER) IS
    v_qcdate DATE;
    v_time CHAR(8);
    v_qcid NUMBER := TO_NUMBER(p_qcid);
    v_obsdate DATE;
    v_obsid NUMBER := TO_NUMBER(p_obsid);
    v_cnt NUMBER;
    BEGIN
--     Проверим p_qcid
       IF    v_qcid IS NULL THEN
             RAISE NO_DATA_FOUND;
       END IF;
--     Проверим v_obsid
       IF    v_obsid IS NULL THEN
             RAISE NO_DATA_FOUND;
       END IF;
--     Проверим, существует ли QcSet
       SELECT Q.when_t, q.at_time INTO v_qcdate, v_time
             FROM QC_EVENT_LIST Q
             WHERE Q.qc_id = v_qcid;
--     Проверим, существует ли Obsid
       SELECT O.when_t INTO v_obsdate
             FROM OCEANIC_OBSERVATION_LIST O
             WHERE O.obs_id = v_obsid;
       v_cnt := 0;
--     Проверим, существует ли PASSED_OBSERVATION_LIST
       SELECT COUNT(1) INTO v_cnt
             FROM PASSED_OBSERVATION_LIST P
             WHERE P.qcid = v_qcid;
       IF    ( v_cnt = 0 ) THEN
             INSERT INTO PASSED_OBSERVATION_LIST
               VALUES(v_qcid,v_qcdate,v_time,PASSEDOBSARRAY());
       END IF;
```

```
        INSERT INTO TABLE
             (SELECT P.idobj FROM passed_observation_list P
               WHERE P.qcid = v_qcid)
             VALUES(PASSEDOBS(v_obsid,v_obsdate));
  EXCEPTION
     WHEN OTHERS THEN
       RAISE NO_DATA_FOUND;
  END insertQcSetObs;

END QCACTIONS;
/
```

Шрифтовые соглашения

В этой книге приняты соглашения, по которым моноширинным шрифтом выделяются:

- Классы: стандартный класс Java `Java.lang.*`
- Типы данных (прописными литерами): тип данных `REF CURSOR`
- Имена и расширения файлов (строчными литерами): файлы `.class`, расширение `.ser`
- Функции и процедуры: `insertQcSet()`, `function GetObsId()`
- Имена таблиц баз данных (прописными литерами): `PASSED_OBSERVATION`
- Ключевые слова Java в абзацах (строчными литерами полужирного начертания): `public`

Обратная связь с авторами

Авторы с радостью примут замечания и предложения читателей относительно качества и полезности этой книги. Ваши отклики для нас очень важны — высылайте их по электронной почте:

Нирва Морисео-Леруа	nmorisseauleroy@data-i.com
Мартин К. Соломон	marty@cse.fau.edu
Джули Басу	jbasu@us.oracle.com

Получение примеров по сети

Исходный программный текст и глоссарий можно найти на сайтах www.data-i.com и www.osborne.com. Программы, исходный текст которых присутствует там, указаны в файле с соответствующим именем, например `ObsImp.sqlj`.

Внимание !

Представленные здесь программы не нацелены на использование в каких бы то ни было приложениях, опасных по своей природе. Ответственность за принятие всех необходимых мер по безотказной работе подобных приложений, их дублированию, резервированию и безопасности использования полностью возложена на читателя.

О ресурсах, используемых в данной книге

На сайте издательства "Лори" (www.lory-press.ru) содержится ряд ресурсов, на которые в книге есть ссылки.

Программы

Программы, содержащиеся в книге, находятся на сайте издательства "Лори" (www.lory-press.ru) в каталоге `BookCode.zip`. Это программы из следующих глав:

Глава 1. Знакомство с распределенными вычислительными системами

Глава 4. Разработка сеансовых зерен EJB

Глава 5. Разработка сеансовых зерен: углубленное изучение

Глава 6. Знакомство с CORBA

Глава 7. Сеансы CORBA

Глава 8. Управление транзакциями в CORBA

Глава 9. Компоненты схемы Purchase Order

Глава 10. Введение в программирование с использованием страниц JSP

Приложения

Для удобства на сайте издательства "Лори" (www.lory-press.ru) размещены электронные варианты следующих приложений к книге (для сервлетов, EJB, CORBA и JSP):

▼ ВНИМАНИЕ! *Для просмотра этих файлов нужно применять Adobe Acrobat Reader версии 3.0 или выше.*

Приложение A. Краткий справочник по сервлетам и общие сведения об API сервлетов

Приложение B. Краткое описание API Enterprise JavaBeans

Приложение C. Краткий справочник по Oracle8i CORBA для Java

Приложение E. Установка и конфигурирование web-серверов для работы с Oracle JSP

Приложение F. Инструментальные средства Oracle8*i* EJB и CORBA и поддержка JSP в JDeveloper

Примеры для JSP и сервлетов

Примеры программ JSP и сервлетов находятся в каталоге `jspsamples.zip` сайта издательства "Лори" (www.lory-press.ru). Операции установки и выполнения этих программ описаны в README.txt.

Содержание

ЧАСТЬ I
Общий обзор

1 Знакомство с распределенными вычислительными системами . . . 3

Компоненты: общее представление . . . 4
Что такое компонент . . . 4
Программные компоненты: общее представление . . . 5
Преимущества компонентов и программных компонентов . . . 6
Распределенные вычислительные системы . . . 6
Распределенные объектные архитектуры . . . 7
Стандарты распределенных вычислительных систем . . . 7
Достоинства распределенных вычислительных систем . . . 9
Построение компонента в Java . . . 9
Класс ObsHelper . . . 9
Класс ObsException . . . 11
Интерфейс Observation . . . 12
Класс ObsImp: реализация с помощью JDBC . . . 12
Класс ObsImp: реализация с помощью SQLJ . . . 16
Класс Client . . . 20
Итоги . . . 22

2 Знакомство с web-приложениями . . . 23

Понятие web-сервера . . . 24
web-браузер . . . 25
Протокол HTTP . . . 27
Язык разметки HTML . . . 28
Генерация динамического содержимого на Web-страницах . . . 29
Модель сервлетов . . . 29
Знакомство с JavaServer Pages . . . 38
Архитектура web-приложений . . . 42
Упрощенная архитектура приложения . . . 42
Компонентная JSP-архитектура . . . 43
Использование библиотек тегов JSP . . . 45
Итоги . . . 47

ЧАСТЬ II
Построение компонентов Enterprise JavaBeans

3 Знакомство с Enterprise JavaBeans (EJB) . . . 51

Спецификация Enterprise JavaBeans (1.0) . . . 53
Цели проекта Enterprise JavaBeans . . . 53
Роли Enterprise JavaBeans . . . 54
Архитектура Enterprise JavaBeans . . . 55
Сервер Enterprise JavaBeans . . . 56
Контейнер Enterprise JavaBeans . . . 56
Компоненты приложения EJB . . . 57
Типы зерен EJB . . . 57
Домашний интерфейс EJB . . . 59
Удаленный интерфейс EJB . . . 62
Класс зерна уровня предприятия . . . 65
Дескриптор внедрения Enterprise JavaBeans . . . 68
Файл ejb-jar . . . 70
Клиентское приложение . . . 71
Управление транзакциями и безопасность в EJB . . . 73

Поддержка транзакций · 73
Поддержка распределения · 74
Поддержка безопасности · 74
Исключительные ситуации EJB · 74
Достоинства Enterprise JavaBeans · 75
Архитектура Oracle8*i* JServer · 76
Основы Oracle8*i* JServer · 76
Создание первого Enterprise JavaBean · 81
Итоги · 92

4 Разработка сеансовых зерен EJB · 93

Oracle8*i* JServer: поддержка транзакций · 94
Еще раз о декларативных атрибутах транзакций · 95
Управление транзакциями · 96
Управление доступом · 100
Программные ограничения · 102
Разработка приложений баз данных с использованием сеансовых зерен · 103
Обзор JDBC-драйверов Oracle · 103
Подключение с помощью JNDI · 105
Итоги · 132

5 Разработка сеансовых зерен: углубленное изучение · 133

Управление зернами в сеансах связи с базой данных · 134
Клиентский объект, начинающий новый сеанс · 134
Серверный объект, начинающий новый сеанс · 135
Два клиента, обращающиеся к одному сеансу · 136
Серверный объект, обращающийся к тому же сеансу · 137
Серверный объект, управляющий продолжительностью сеанса · 138
Серверный объект, заканчивающий сеанс · 138
Ограничение транзакций · 138
Транзакция, ограниченная на стороне клиента (автономная программа Java) 138
Транзакция, ограниченная на стороне сервера · 140
Построение транзакционного приложения · 141
Еще раз о транзакциях в EJB · 141
Требования пользователей · 142
Сеансовое зерно QcObsBean · 148
Сеансовое зерно QcSetBean · 154
Транзакционное приложение QcClient · 165
Итоги · 172

ЧАСТЬ III

Построение компонентов CORBA 173

6 Знакомство с CORBA · 175

Обзор CORBA · 176
IDL · 177
Компилятор IDL · 177
Конструкции IDL · 179
Внедрение и использование объектов CORBA · 186
Программирование модуля IDL · 186
Создание исходных файлов Java из файла IDL с помощью idl2java · 187
Программирование Java-реализации серверного объекта · 187
Компиляция файлов Java и загрузка созданных файлов классов в JServer · 187
Публикация имени серверного объекта CORBA · 188
Программирование и компиляция клиента Java · 189
Реализация серверного объекта CORBA с помощью JDBC · 191
Механизм обратных вызовов CORBA · 191
Клиентский объект обратного вызова · 192
Серверный объект · 192
Реализация клиентского объекта обратного вызова · 192
Реализация серверного объекта · 192

Клиентская программа, обращающаяся к серверному методу обратного вызова . . . 193
Механизм привязки CORBA . . . 194
Активация посредников ORB и серверных объектов . . . 194
Активация ORB . . . 194
Активация серверного объекта . . . 195
Программирование спецификаций интерфейсов на Java . . . 196
Краткие сведения о некоторых важных инструментах . . . 197
loadjava . . . 197
dropjava . . . 198
publish . . . 198
remove . . . 201
Итоги . . . 202

7 Сеансы CORBA . . . 203

Службы соединений: TTC и IIOP . . . 204
Характеристики JNDI . . . 204
JNDI-интерфейс Context . . . 205
JNDI-класс InitialContext . . . 205
CORBA-сеансы Oracle8*i* . . . 206
Прослушивающие процессы и диспетчеры базы данных . . . 206
Синтаксис URL для служб и сеансов . . . 207
Компоненты и классы URL . . . 207
Управление сеансами . . . 211
Запуск сеанса из серверного объекта . . . 211
Запуск именованного сеанса со стороны клиента . . . 212
Пример двухсеансовой программы . . . 212
Пример программы с указателями URL, содержащими имена сеансов . . . 214
Управление продолжительностью сеанса и его завершение . . . 216
Аутентификация . . . 217
Способы аутентификации клиента . . . 217
Права доступа к объектам базы данных . . . 219
Клиенты не-JNDI . . . 219
Итоги . . . 220

8 Управление транзакциями в CORBA . . . 221

Ограниченные возможности JTS . . . 223
Ограничения транзакций и контекст транзакций . . . 223
Интерфейсы службы транзакций . . . 224
Методы службы транзакций Java . . . 225
public static synchronized TransactionService getTS() . . . 225
public void begin() . . . 225
public Control suspend() . . . 226
public void resume(Control control) . . . 226
public void commit(boolean report_heuristics) . . . 226
public void rollback() . . . 226
public void rollback_only() . . . 227
public void set_timeout(int seconds) . . . 227
public Status get_status() . . . 227
public String get_transaction_name() . . . 227
Использование методов JTS в клиентских программных конструкциях CORBA . . . 227
Пример ограничения на стороне клиента . . . 228
Использование методов JTS в серверных программных конструкциях CORBA . . . 230
Пример ограничения на стороне сервера . . . 230
Компонент серверного объекта CORBA . . . 233
accountlist.idl . . . 234
AccountListImpl.sqlj . . . 235
Клиент CORBA для компонента AccountList . . . 237
Пример с приостановкой и возобновлением транзакции . . . 238
Клиент, обращающийся к компоненту EJB . . . 239
Клиент CORBA, обращающийся к компоненту EJB и компоненту серверного объекта CORBA . . . 241
JTA . . . 243

Структура клиента с ограничением JTA на стороне клиента 243
Пример ограничения JTA на стороне клиента 244
Структура реализации серверного объекта
с ограничением JTA на стороне сервера 245
Итоги 245

9 Компоненты схемы Purchase Order 247

CORBA-компонент DepartmentList 248
Требования пользователей DepartmentList 249
Общий вид DepartmentList 249
IDL-файл для DepartmentList 249
Реализация серверного объекта для DepartmentList 250
Клиент для DepartmentList 251
CORBA-компонент EmployeeList 252
Требования пользователей EmployeeList 253
Общий вид EmployeeList 253
IDL-файл для EmployeeList 254
Реализация серверного объекта для EmployeeList 254
Клиент для EmployeeList 257
CORBA-компонент VendorList 258
Требования пользователей VendorList 258
Общий вид VendorList 259
IDL-файл для VendorList 259
Реализация серверного объекта для VendorList 259
Клиент для VendorList 261
CORBA-компонент ProjectList 262
Требования пользователей ProjectList 262
Общий вид ProjectList 263
IDL-файл для ProjectList 263
Реализация серверного объекта для ProjectList 263
Клиент для ProjectList 265
CORBA-компоненты PurchaseList и LineItemList 266
Требования пользователей PurchaseList и LineItemList 267
Общий вид PurchaseList и LineItemList 268
IDL-файл для PurchaseList и LineItemList 269
Реализация серверного объекта для PurchaseList 271
Реализация серверного объекта для LineItemList 274
Клиент для PurchaseList 278
Итоги 280

ЧАСТЬ IV

Построение web-приложений с помощью страниц JSP

10 Введение в программирование с использованием страниц JSP . . . 283

Основы JSP-программирования 285
Элементы сценариев 285
Неявные объекты JSP 287
Написание Hello.jsp 287
Выполнение Hello.jsp 289
Заглянем внутрь 291
Директивы JSP 294
Директива page 295
Директива include 296
Вызов других страниц с JSP 297
Тэг <jsp:include> 298
Тэг <jsp:forward> 300
Использование JSP-страниц вместе с сервлетами 301
Вызов сервлета с JSP-страницы 302
Вызов JSP из сервлета 303
Особенности сред Servlet 2.0 305
Обработка ошибок JSP 307
JSP-реализация Oracle 308

Использование SQLJ в JSP · 309
Использование транслятора командной строки ojspc · 311
Итоги · 314

11 Использование JSP вместе с зернами JavaBean · 315

Разработка JSP-приложений с помощью компонентов JavaBean · 316
Знакомство с JavaBean · 316
Стандартные JSP-тэги для зерен JavaBean · 318
Создание HelloBean.jsp с помощью UserNameBean · 321
Использование зерен JavaBean с разными областями действия · 322
Разработка JSP-приложения с контролем за сеансами · 328
Выполнение операций над базой данных с помощью зерен JavaBean · 333
Ввод информации в базу данных · 333
Исполнение статических SQL-запросов · 338
Исполнение динамических SQL-запросов · 345
Зерна доступа к базам данных в Oracle JSP 1.1 · 349
Библиотека зерен Oracle JSP · 349
Генерация XML-страниц с помощью JSP-страниц и зерен JavaBean · 350
Оптимизация SQL-операций · 353
Использование готовых JDBC-операторов · 354
Пакетное обновление · 355
Кэширование описателей операторов · 355
Пулинг соединений · 355
Кэширование результатов запроса · 357
Итоги · 358

12 Использование JSP вместе с EJB, CORBA и библиотеками тэгов · 359

Разработка web-приложений с помощью EJB-компонентов · 360
Сеансовое EJB для заказов · 360
Вызов OrderEJB из web-приложения · 365
Разработка web-приложений с помощью компонентов CORBA · 372
Объект CORBA для заказов · 372
Вызов объекта CORBA из web-приложения · 374
Использование библиотек тэгов JSP · 378
Директива taglib · 379
SQL-тэги · 379
Тэги JML · 384
Реализация специального тэга · 385
Итоги · 395

ЧАСТЬ V
Приложения

A Краткий справочник по сервлетам и общие сведения об API сервлетов · 399

Краткий справочник по сервлетам · 400
Написание сервлетов · 400
Компиляция сервлетов · 404
Внедрение сервлетов в Tomcat · 405
Вызов сервлетов · 405
Servlet API 2.2 · 406
Иерархическая структура классов · 407
Иерархическая структура интерфейсов · 407
Классы и интерфейсы · 408

B Краткое описание API Enterprise JavaBeans · 417

Краткий справочник по Enterprise JavaBeans · 418
Создание зерна уровня предприятия (сеансового зерна) · 418
Enterprise JavaBeans API · 424
Иерархическая структура пакета javax.ejb · 424
Иерархическая структура интерфейсов · 424
Пакет javax.ejb · 424

C Краткий справочник по Oracle8i CORBA для Java 431
Программирование модуля IDL 432
Генерация исходных файлов Java из файла IDL с помощью idl2java 432
Программирование Java-реализации серверного объекта 433
Компиляция файлов Java и загрузка сгенерированных файлов классов в JServer 433
Публикация имени серверного объекта CORBA 434
Программирование и компиляция клиента Java 434
Выполнение клиента 435

D Краткий справочник по JavaServer Pages 437
Справочник по синтаксису JSP 438
Директива page 438
Директива taglib 438
Тэги include 438
Тэги JavaBean 438
Комментарии 439
Объявление переменных и функций 439
Скриптлеты 439
Выражения 440
Тэг jsp:plugin 440
Тэг jsp:forward 440
Справочник по JSP API 440
Пакет javax.servlet.jsp 441
Пакет javax.servlet.jsp.tagext 443

E Установка и конфигурирование web-серверов для работы с Oracle JSP 447
Установка Oracle JSP в Tomcat 448
Этапы установки 448
Этапы конфигурирования 449
Запуск сервера Tomcat 452
Выполнение JSP-программ в Tomcat 452
Установка и конфигурирование Java Web Server 453
Этапы установки 453
Этапы конфигурирования 454
Выполнение JSP-программ в JWS 455
Установка и конфигурирование Apache и JServ 455
Этапы конфигурирования 456
Выполнение JSP-программ в Apache/JServ 456
Параметры конфигурации Oracle JSP 457

F Инструментальные средства Oracle8i EJB и CORBA и поддержка JSP в JDeveloper 461
Инструментальные средства Oracle8*i* EJB и CORBA 462
Инструментальные средства Oracle8*i*, используемые для EJB 462
Инструментальные средства Oracle8*i*, используемые как для EJB, так и для CORBA 464
Инструментальные средства Oracle8i, используемые для CORBA 469
Поддержка JSP в JDeveloper 472
Создание нового приложения JavaServer Pages 473

Библиография 481

ЧАСТЬ I

Общий обзор

ГЛАВА 1

Знакомство с распределенными вычислительными системами

Основная тема книги – разработка клиентских и серверных компонентов Java с помощью реализаций JDBC и SQLJ. Первая из них позволяет встраивать динамический SQL в программы Java, а вторая – включать в программы Java статические SQL-конструкции. В статическом встраиваемом SQL все SQL-операторы, включаемые в программу, становятся известными во время компиляции, в то время как в динамическом встраиваемом, по крайней мере, часть их совершенно неизвестна до момента выполнения программы.

Прочитав книгу, вы научитесь разрабатывать, внедрять и настраивать такие многоуровневые программные компоненты для БД, которые будут манипулировать информацией, находящейся на сервере данных Oracle8*i* версий 8.1.5, 8.1.6 и новой 8.1.7. Они будут использоваться как простые бизнес-объекты, но, что особенно важно, внедряются с помощью таких компонентных моделей, как Enterprise JavaBeans (EJBs), CORBA, сервлеты Java, JavaBeans или JavaServer Pages (JSPs). Последние позволяют компаниям устанавливать одно и то же приложение на самые разные рабочие серверы и распределять таковое среди самых разных клиентов, в том числе автономных программ Java, клиентов EJB, CORBA, JSP, сервлетов и webбраузеров.

В части I (главы 1 и 2) представлены новейшие тенденции разработки программных компонентов, основные принципы распределенных вычислительных систем и разработки web-приложений, в том числе сервлетов Java.

В части II (главы 3-5) раскрываются основные понятия архитектуры Enterprise JavaBeans (EJB 1.0 и 1.1). Здесь строятся и развертываются в базе данных Oracle8*i* простые и составные компоненты EJB для БД.

В части III (главы 6-9) разрабатываются и внедряются Java-компоненты CORBA. Кроме того, создаются Java-клиенты CORBA, использующие компоненты EJB, которые будут построены в части II.

В части IV (главы 10-12) описана разработка компонентов JavaServer Pages и JavaBean. В главе 12 с помощью расширяемого языка разметки (XML – Extensible Markup Language), JSP и сервлетов строятся клиентские приложения, которые будут пользоваться компонентами EJB и CORBA, построенными в частях II и III.

Итак, темы настоящей главы:

- Базовые понятия для компонентов
- Базовые понятия для программных компонентов
- Распределенные вычислительные системы
- Основные строительные блоки компонента, использующего Java (в этом разделе с помощью реализаций JDBC и SQLJ строится простой компонент Java).

Предлагаем вам общий обзор самых современных тенденций разработки программных компонентов, а также основных принципов распределенных вычислительных систем.

Компоненты: общее представление

Время крупных, монолитных систем уже уходит. Скорость этого процесса весьма велика, и циклы разработки сокращаются от нескольких месяцев до пары недель. Сегодня распространены укороченные процессы разработки, когда крупные и сложные приложения строятся из нескольких небольших "частей" – *компонентов*, которые разрабатываются за более короткое время.

Что такое компонент

Компонент (component) – это независимая программная единица уровня приложения, разрабатываемая не для конкретного приложения, а с определенной целью. К программированию компонентов можно привлечь группу разработчиков, наделив каждого своими обязанностями, что, несомненно,

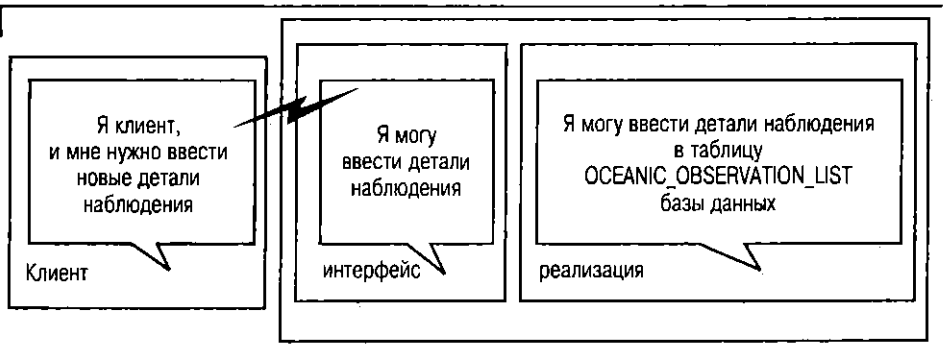
сэкономит время на разработку. Проектирование особенно крупных приложений нередко поручают даже нескольким организациям.

Компонентный подход к программированию основывается на базовых понятиях теории ориентации на объекты. Хотя нередко термины "компонент" и "объект" используются как синонимы, компонент – это не объект. Объект (object) – это экземпляр класса, создаваемый во время выполнения программы. Компонент может быть классом, но обычно представляет собой совокупность классов и интерфейсов. На этапе выполнения программы компонент "оживает" при конкретизации его классов. Таким образом, на данном этапе компонент становится сетью объектов.

Как правило, компоненты – это многократно используемые бизнес-объекты с заранее определенным режимом работы. Детали их реализации скрываются в интерфейсах, обособляющих и формирующих (инкапсулирующих) набор функциональных возможностей и соединяющих компоненты. Интерфейс (interface) – это набор именованных операций, которые могут вызываться клиентами. Более того, грамотно описанные интерфейсы определяют точки входа компонента, и доступ к нему осуществляется только через его интерфейс. При использовании компонентного метода поставщики услуг и клиенты взаимодействуют друг с другом через спецификацию интерфейса, которая становится посредником между двумя сторонами, позволяя им сотрудничать и работать совместно. В разделе "Построение компонента в Java" этой главы с помощью Java строится компонент `Observation`. В главе 4 функциональные возможности компонента `Observation` расширяются, и он развертывается в качестве зерна (Bean) уровня предприятия.

На рис. 1.1 представлен общий вид компонента.

Рис. 1.1. Состав компонента



При программировании с ориентацией на компоненты обращаются к различным их аспектам. Сам компонент можно построить при помощи любого языка программирования, конечно при условии, что последним поддерживаются интерфейсные соглашения, принятые для стандарта конкретного компонента. Во многих языках программирования, например COBOL, Object COBOL, C, C++, Pascal, Object Pascal, Modula 2, Eiffel или Smalltalk, нет достаточной поддержки инкапсуляции, полиморфизма, безопасности типов или комбинации этих характеристик. В настоящее время самым известным языком программирования с использованием компонентов является Java.

С появлением программного продукта Java 2 Enterprise Edition (J2EE) от Sun Microsystems комбинация EJB, сервлетов и JSP формирует мощную платформу для построения приложений на основе компонентов.

Программные компоненты: общее представление

Компоненты применяются в основном по двум причинам: из-за возможности многократного использования большим числом приложений и благодаря интерфейсам. Правильное применение интерфейсов может принести большую пользу, позволяя, например, строить самодостаточные единицы или одновременно и параллельно разрабатывать приложения. Компоненты лучше всего применять в процессе сборки приложений. Компонентный подход должен

быть не революционным, а эволюционным. Собирая, т.е. объединяя заранее заготовленные компоненты, разработчики могут создавать новые конструкции.

Сборные системы, состоящие из компонентов, называются *программными компонентами* (software components). "Программный компонент – это структурная единица с заранее оговоренными интерфейсами и исключительно явными контекстными зависимостями. Он может быть установлен автономно и включен в конструкцию посторонними" [European Conference on Object-Oriented Programming (ECOOP), 1996].

Изначально программные компоненты рассматривались по аналогии с аппаратными компонентами и микросхемами. Они совершенно самостоятельны, т.е. независимы от среды и приложения и при построении работоспособной системы взаимодействуют друг с другом посредством методов (операций), заявленных в их интерфейсах. Подробнее о программных компонентах мы поговорим в главе 5 при построении компонентного приложения, состоящего из трех зерен Enterprise JavaBeans.

Преимущества компонентов и программных компонентов

- **Независимость** Компонент – обобщенная единица, не зависящая от приложения.
- **Многократное использование** Это многократно используемые единицы. По сравнению с конкретными решениями конкретных проблем компоненты более общие, что допускает неоднократное их использование в самых различных контекстах.
- **Настройка** Отдельные компоненты можно делать на заказ, удовлетворяя определенные потребности, а готовый – настроить так, чтобы он им соответствовал.
- **Компоновка** Собрав несколько компонентов, можно сформировать работоспособную систему.
- **Простота модернизации и обслуживания** Модернизация отдельных компонентов устраняет необходимость в объемной модернизации, обязательной в монолитных системах.
- **Прозрачность местонахождения** Компоненты могут находиться в любом месте сети, на наиболее удобных для их функционирования компьютерах; это определяется функциями компонента.
- **Распределение** С помощью таких стандартов распределенных вычислительных систем, как Enterprise JavaBeans, CORBA или Distributed Component Object Model/Component Object Model (DCOM/COM) корпорации Microsoft компоненты и программные компоненты можно распределить по всей сети предприятия.

Распределенные вычислительные системы

Общей характеристикой современных корпоративных и правительственных сетей является их неоднородность (гетерогенность). Неоднородные системы представляют собой комбинацию множества ОС, разнесенных среди нескольких аппаратных (мэйнфреймы, рабочие станции, персональные компьютеры и др.) и программных компонентов. Распределение объектных вычислений – это метод построения программной инфраструктуры, объединяющей компоненты сети и позволяющей им взаимодействовать друг с другом.

Рассмотрим, каковы основные принципы распределенных вычислительных систем и как последние связаны со специалистами по информационным технологиям, отвечающими за построение информационных систем, и системами управления БД, в частности с объектно-реляционной БД Oracle8*i* (версий 8.1.5, 8.1.6 и 8.1.7), реализующей архитектуры компонентных моделей на сервере баз данных.

В основе всех архитектур распределенных вычислений лежит взаимодействие компьютеров. Новейшей разработкой в системах распределенных вычислений является распределение объектов, когда объекты (бизнес-логика и данные) разносятся по неоднородной сети: независимо от того, находятся ли они в различных адресных пространствах или на разных компьютерах, объекты кажутся частями единого целого.

Термином "распределенные объектные вычисления" (distributed object computing) обозначаются те программы и приложения, которые удаленно вызывают другие программы, находящиеся в других адресных пространствах, а возможно, и на других компьютерах и/или в других сетях. Распределенные объектные вычисления – это основа вычислений, ставшая результатом постепенного сближения объектно-ориентированной технологии и технологии клиент/сервер. Более того, она обеспечивает взаимодействие и возможность многократного использования распределенных объектов, что позволяет разработчикам строить системы, собирая компоненты от разных поставщиков.

Распределенные объектные архитектуры

Распределенные объектные системы служат фундаментом трехуровневой архитектуры, в которой логические схемы представления, или первый уровень, находятся на станции клиента, бизнес-логика – на среднем уровне, а база данных – на третьем. Распределенная объектная технология расширяет средний уровень, позволяя обращаться не только к одному прикладному объекту, но и к нескольким. В результате рождается новая архитектура, называемая *N-уровневой* (N-tier), или *многоуровневой* (multi-tier). В ней возможно сосуществование множества прикладных объектов (т.е. серверов баз данных, объектов Java RMI, EJB, CORBA, DCOM и др.), причем клиентские и серверные объекты взаимодействуют посредством специального протокола удаленного вызова методов (RMI, remote method invocation). Протокол RMI используется для удаленного вызова коммуникационных методов. Например, у каждой из моделей CORBA, Java RMI и Microsoft DCOM он свой. У любого прикладного объекта есть определенный интерфейс объектной оболочки, где заявляются услуги, предоставляемые объектом и особенно важно, что связь осуществляется только через этот интерфейс.

В основе всех распределенных объектных протоколов лежит одна и та же базовая архитектура. Распределенные объектные архитектуры основаны на сетевом коммуникационном слое (уровне), состоящем из трех частей: *объектного сервера* (object server), *скелета* (skeleton) и *изолятора* (stub). Первый и второй располагаются, как правило, на среднем уровне, но в Oracle8*i* находятся на третьем (т.е. на сервере баз данных). Изолятор размещается на машине клиента и обеспечивает межпроцессную связь клиентских и серверных объектов. Для клиента он выступает в роли посредника и несет ответственность за коммуникационные запросы первого, передаваемые объектному серверу через скелет. Изолятор и скелет отвечают за то, чтобы объектный сервер (который может находиться на среднем или третьем уровне) выглядел так, будто он работает в определенном месте.

Пересылают данные из одного адресного пространства в другое изолятор и скелет с помощью двух процессов – *упорядочения* (marshaling) и *обратного упорядочения* (unmarshaling). "Во время упорядочения параметры вызова метода (в пространстве клиента) или возвращаемые значения (в пространстве сервера) упаковываются в стандартный формат для передачи" (См. книгу "Oracle8*i* SQLJ Programming", Osborne/McGraw-Hill (далее OMcGH), 1999).

Стандарты распределенных вычислительных систем

Для распределенных объектных вычислений существует несколько стандартов:

- Группа управления объектами (OMG, Object Management Group) в начале 1990 г. разработала общую архитектуру посредника объектных запросов (CORBA, Common Object Request Broker Architecture),

в основе которой лежит абстрактная модель, называемая архитектурой управления объектами (OMA, Object Management Architecture), где для организации взаимодействия между клиентскими и серверными объектами и управления ими применяется посредник объектных запросов (ORB, Object Request Broker). Удаленное обращение осуществляется через протокол Internet Inter-ORB (IIOP). Он дает возможность писать распределенные программы взаимодействия через Интернет на разных языках программирования. Объекты и интерфейсы CORBA определяются при помощи языка описания интерфейсов OMG (OMG IDL, OMG Interface Definition Language). Он позволяет взаимодействовать клиентским и серверным объектам, написанным на разных языках программирования. В число соответствий OMG IDL для языков программирования (Java, C, C++, Ada и COBOL) входят описания специфичных типов данных и интерфейсов для обращения к объектам CORBA. Объекты CORBA можно распределять на многие аппаратные платформы (рабочие станции UNIX, Windows NT и др.). CORBA описывается в главах 6-9.

- Распределенная компонентная объектная модель (DCOM, Distributed Component Object Model), разработанная в Microsoft,– это компонентная технология распределения приложений в архитектуре Windows. Она основана на компонентной объектной модели (COM, Component Object Model), которая позволяет клиентам вызывать службы, предоставляемые согласующимися с COM компонентами (объектами COM). Объекты и интерфейсы COM определяются при помощи языка описания интерфейсов IDL (Microsoft Interface Definition Language), расширения стандарта DCE Interface Definition Language.
- Удаленный вызов методов Java (RMI, Remote Method Invocation) от Sun JavaSoft позволяет объекту Java, функционирующему в одной виртуальной машине Java (JVM, Java Virtual Machine), вызывать методы другого объекта Java, функционирующего в другой JVM. В RMI с этой целью используется протокол JRMP (Java Remote Method Protocol). Кстати, способ, изначально задуманный для работы только в среде Java. В июне 1999 г. Sun выпустила спецификацию RMI over IIOP (RMI-IIOP): будучи разработанной совместно Sun и IBM, она позволяет объектам Java взаимодействовать с объектами CORBA. Спецификацией RMI-IIOP поддерживаются и платформы JDK начиная с 1.1.6, и Java 2.
- Компонентная архитектура JavaBeans от Sun JavaSoft позволяет разработчикам создавать клиентские компоненты, которые можно собирать при помощи визуальных построителей приложений (например, Oracle JDeveloper или Visual Café) и невизуальных средств. Подробнее о разработке компонентов JavaBean см. в главе 11.
- Enterprise JavaBean (EJB) – это компонентная модель, позволяющая разработчикам распределять компоненты на сервере (на прикладных серверах и серверах баз данных). В приложениях EJB удаленный вызов соответствует спецификации RMI, но производители не ограничены транспортным протоколом RMI. Например, на сервере EJB в Oracle8*i* в качестве транспортного протокола применяется RMI over IIOP. Серверные компоненты используются на прикладных серверах промежуточного программного обеспечения, где компоненты обслуживаются во время выполнения программы и доступны для удаленных клиентов. С появлением РСУБД Oracle8*i* разработчики получили возможность сохранять объекты EJB и CORBA внутри базы данных. С помощью EJB разрабатываются и внедряются N-уровневые, распределенные и объектно-ориентированные приложения Java. Подробнее о ней см. главы 3-5.

Достоинства распределенных вычислительных систем

- **Разбиение сложных прикладных программ на программные компоненты** Следовательно, различные задачи можно распределять сразу среди нескольких разработчиков, получая несколько оперативных и независимых решений.
- **Упрощение модернизации и обслуживания** Нередко обновление и обслуживание монолитных систем оказывается довольно дорогим и долгим. Программные приложения, моделирующие бизнес-объекты, более гибки, расширяемы, и применять их можно неоднократно.
- **Распределение программных компонентов среди компьютеров, наиболее подходящих для выполнения задачи** Кроме того, программные компоненты могут использоваться несколькими приложениями.
- **Использование *объектных оболочек* при обращении к старым системам** Старые (унаследованные) системы — неотъемлемая часть нынешних. Объектные оболочки — объектно-ориентированные интерфейсы, окружающие старые системы,— позволяют последним в полной мере участвовать в работе информационных систем нового поколения, делая их доступными и разрешая связь с ними. Например, web-браузеры с CORBA или клиенты CORBA могут напрямую вызывать объектные оболочки, если, конечно, оболочки созданы с помощью OMG IDL.

Построение компонента в Java

Компонент в Java строится с помощью базовых блоков. Более того, компонент `Observation` — простой бизнес-объект — разрабатывается с применением реализаций JDBC и SQLJ. `Observation` позволяет клиенту обращаться к базе данных Oracle8*i* и вводить массив наблюдений за атмосферой океана в таблицу `OCEANIC_OBSERVATION_LIST`. Эта таблица является частью научной схемы наблюдений, представленной во введении. В главе 4 функциональные возможности этого компонента расширяются, и он внедряется в базу данных Oracle8*i* в качестве компонента EJB.

Компонент состоит как минимум из интерфейса и класса реализации, которые, работая в совокупности, создают некий функциональный набор. В частности, `Observation` состоит из:

- Интерфейса `Observation`
- Класса реализации `ObsImp`
- Класса `ObsHelper`
- Класса `ObsException`
- Класса `Client`

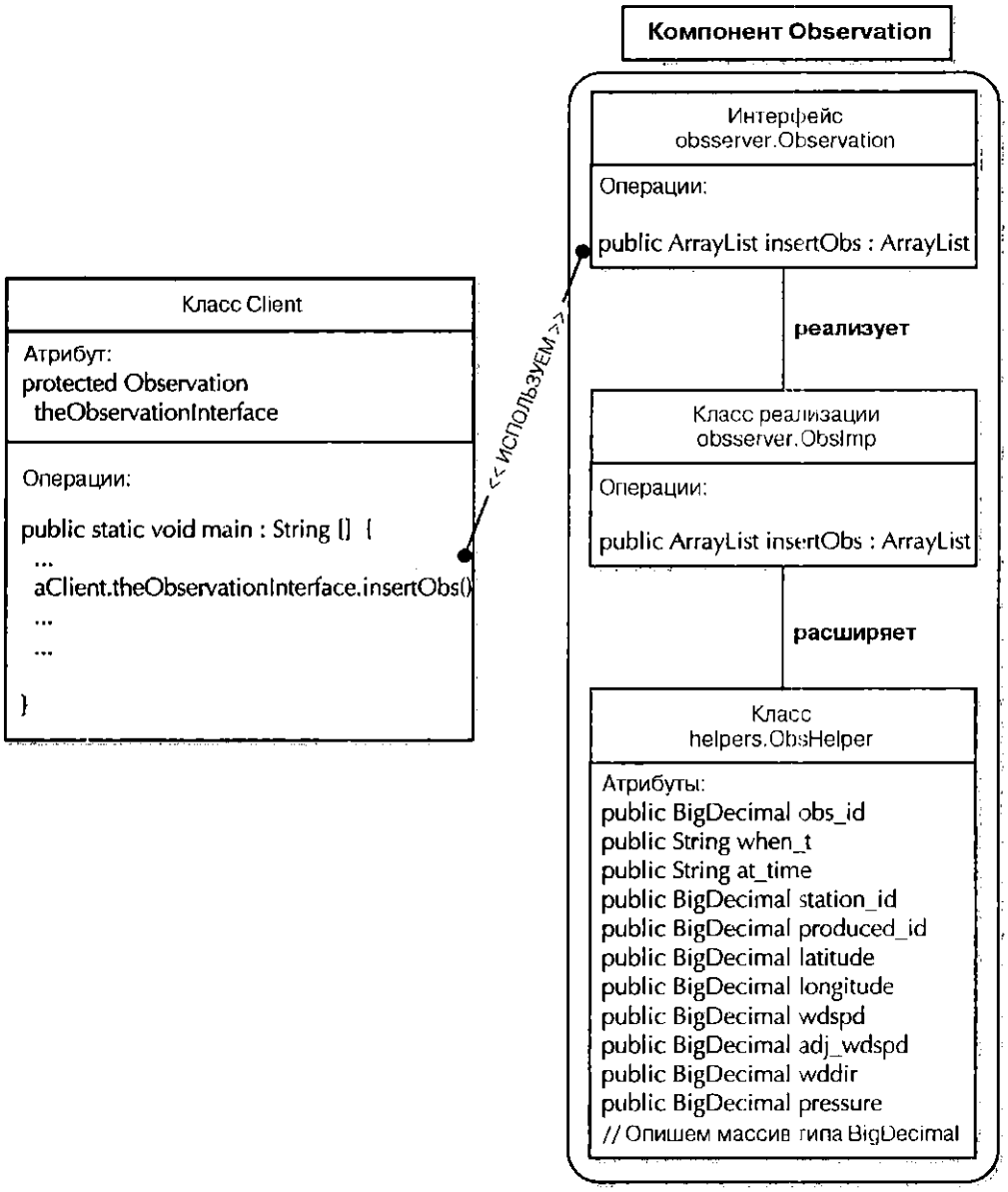
На рис. 1.2 представлены основные строительные блоки компонента в Java. Прежде всего — описания классов `ObsHelper` и `ObsException`.

Класс `ObsHelper`

Указанным классом реализуется интерфейс `java.io.Serializable`. В `ObsHelper` содержится несколько конструкторов, позволяющих создать несколько объектов `ObsHelper` в соответствии с конкретными потребностями. Экземпляры класса используются для прямого и обратного упорядочения данных, пересылаемых между приложением `Client` и классом `ObsImp`, который будет создан в последующих разделах этой главы. Вот описание `ObsHelper`:

```
/*    Имя программы:    ObsHelper.java
**
**    Назначение:       Серийный Helper-класс Java, используемый
**                      для передачи объектов параметров
**                      (OCEANIC_OBSERVATION_TYPE)
**                      между клиентами, компонентами EJB и CORBA.
```

Рис. 1.2.
Общий вид компонента в Java




```
*/
package helpers;

import java.io.Serializable;
import java.math.BigDecimal;

public class ObsHelper
    implements java.io.Serializable {

     // Переменные для элементов
        public BigDecimal obs_id = null;
        public String when_t = null;
        public String at_time = null;
        public BigDecimal station_id = null;
        public BigDecimal produced_id = null;
        public BigDecimal latitude = null;
        public BigDecimal longitude = null;
          public BigDecimal wdspd = null;
        public BigDecimal adj_wdspd = null;
```

```
        public BigDecimal wddir = null;
        public BigDecimal pressure = null;

        // Опишем массив типа BigDecimal
        public BigDecimal[] idArray = null;

        // Конструктор по умолчанию
         public ObsHelper() {
    }
        // Конструктор 1 с параметрами
        public ObsHelper (
             BigDecimal obs_id, String when_t, String at_time,
             BigDecimal station_id, BigDecimal produced_id,
             BigDecimal latitude, BigDecimal longitude,
             BigDecimal wdspd, BigDecimal adj_wdspd,
             BigDecimal wddir, BigDecimal pressure) {
                this.obs_id = obs_id;
                this.when_t = when_t;
                this.at_time = at_time;
                this.station_id = station_id;
                this.produced_id = produced_id;
                this.latitude = latitude;
                this.longitude = longitude;
                this.wdspd = wdspd;
                this.adj_wdspd = adj_wdspd;
                this.wddir = wddir;
                this.pressure = pressure;
    }
}
```

Класс ObsException

В Java существует класс Throwable, характеризующий все, что можно установить в качестве исключительной ситуации, или исключения (exception). Базовым типом, устанавливаемым из любого стандартного метода библиотечного класса Java и пользовательских методов и подпрограмм этапа выполнения, является класс java.lang.Exception. Скорее всего, понадобится создать собственные исключительные ситуации, чтобы обозначить особые ошибки, которые могут возникнуть во время работы программы. Организовать собственный класс исключений можно с помощью существующего типа исключительных ситуаций, унаследовав его. ObsException — это пользовательский класс исключений, расширяющий класс java.lang.Exception. В главе 5 ObsException будет переопределен так, чтобы он унаследовал свойства java.rmi.RemoteException. Последний будет применяться в компонентах EJB для переноса исключительной ситуации из серверных объектов на станцию клиента. Вот описание класса ObsException:

```
/*    Имя программы:     ObsException.java
**
**    Назначение:        Класс исключений, который должен передаваться
**                       из серверных объектов клиенту.
**
**
*/

package helpers;

public class ObsException
      extends Exception {

    // Конструктор по умолчанию
    public ObsException() {
    } // Конец конструктора

    // Конструктор с параметрами
```

```
    public ObsException (String msg ) {
      super(msg);
    } // Конец конструктора с параметрами

} // Конец класса ObsException
```

Интерфейс Observation

Интерфейс (interface) определяется как совокупность именованных операций, у каждой из которых есть определенная сигнатура и, возможно, возвращаемый тип. Интерфейсом Java определяется набор методов или объявлений констант без реализации тел методов. Все объявления методов интерфейса Java автоматически являются абстрактными (abstract) и общими (public). В интерфейсе Observation заявлен один метод — insertObs(). При вызове приложением Client метод insertObs() возвращает вызывающему объект ArrayList, состоящий из объектов ObsHelper.

Приведем описание интерфейса Observation:

```
// Имя программы:    Observation.java
// Название:         Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Описание:         Интерфейс Java, определяющий
//                   вызываемый метод insertObs() клиента.
package obsserver;

// импортируем helper-классы
import helpers.ObsHelper;
import helpers.ObsException;

import java.util.ArrayList;
import java.sql.SQLException;

public interface Observation {

    // Вставим объект ArrayList Наблюдений
    // в таблицу OCEANIC_OBSERVATION_LIST
    // и возвратим объект ArrayList Идентификаторов
    // Наблюдений
    public ArrayList insertObs()
          throws SQLException, ObsException;

} // Конец интерфейса Observation
```

Класс ObsImp: реализация с помощью JDBC

В Java для реализации объявленного интерфейса в новом классе применяется конструкция implements. Поэтому при реализации интерфейса Observation воспользуемся ею в классе ObsImp. Когда мы создаем новый класс, реализующий интерфейс Java, первый должен реализовывать каждый метод второго. Интерфейсом Observation определяется лишь один метод, insertObs(), и его необходимо реализовать в классе ObsImp, для чего применим JDBC. В главах 4 и 5 эта реализация будет пересмотрена и расширена.

В классе ObsImp описывается и реализуется несколько частных (private) методов Java, т.е. тех, что не могут вызываться клиентами, и один общий (public), который клиенты могут вызывать:

- Частный метод getNewObs() не может вызываться клиентами: он обращается к переопределяемому методу getNewObs (Connection conn), который, в свою очередь, генерирует новый идентификатор наблюдения. Данный метод возвращает вызывающему объект ObsHelper с идентификатором наблюдения.

- Частный метод getNewObs (Connection conn) вызывает PL/SQL-функцию ObsActions.getObsId(), генерирующую новый идентификатор наблюдения. При вызове методом getNewObs() он возвращает вызывающему объект ObsHelper. Полное описание PL/SQL-модуля ObsActions дано во введении.
- Частный метод insertObs (ObsHelper obs) вызывает PL/SQL-метод ObsActions.insertObs (...), который вводит OCEANIC_OBSERVATION_TYPE в таблицу OCEANIC_OBSERVATION_LIST. Он возвращает вызывающему объект ObsHelper.
- Общий метод ArrayList insertObs (ArrayList inObs) — единственный, вызываемый клиентами. Напомним, что он и единственный, описанный в интерфейсе Observation. Метод принимает объект ArrayList из объектов ObsHelper в качестве входных данных и возвращает объект ArrayList из идентификаторов наблюдений. Для этого он последовательно вызывает метод insertObs (ObsHelper obs).
- Частный статический (static) метод connectDb() подключается к базе данных и возвращает вызывающему объект java.sql.Connection.

Вот описание класса ObsImp:

```
// Имя программы:    ObsImp.java
// Название:         Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Описание:         Класс реализации Java, реализующий
//                   вызываемый метод insertObs() клиента,
//                   заявленный в интерфейсе Observation.

package obsserver;

// импортируем helper-классы
import helpers.ObsHelper;
import helpers.ObsException;

// импортируем специфичное для приложения исключение
import java.sql.SQLException;

import java.sql.*;
import oracle.sql.*;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.util.ArrayList;

public class ObsImp implements Observation {

    // Конструктор по умолчанию
    public ObsImp() {
    } // Конец конструктора

    // Этот метод НЕ вызывается клиентами.
    // Поэтому он не указан в интерфейсе
    // Observation.
    private ObsHelper getNewObs(Connection conn)
        throws SQLException, ObsException {

    // Создадим объект CallableStatement
    CallableStatement cstmt = null;

    try {
            // Подготовим строку символов (String) для вызова
            String sqlId = "{? = call OBSACTIONS.GETOBSID}";
```

```
                // Подготовим вызов с помощью объекта conn
                cstmt = conn.prepareCall(sqlId);

                // Объявим, что ? - это возвращаемое значение типа Integer
                cstmt.registerOutParameter (1, Types.INTEGER);

                // Получим новый obsid, выполнив запрос
                cstmt.execute();

                // Сохраним результат запроса
                // в переменной anObsId.
                BigDecimal anObsId = new BigDecimal(cstmt.getInt (1));

                // Возвратим объект ObsHelper, используя новый идентификатор
                return new ObsHelper(anObsId);

        } // Конец try
        catch (SQLException e) {
              throw e;
        } // Конец catch

         // Выполним очистку. Этот метод гарантирует, что независимо
         // от происходящего в нем объект CallableStatement
         // будет всегда закрываться после обработки
         //
         finally {
                        if ( cstmt != null ) cstmt.close();
         } // Конец finally()

    } // Конец getNewObs(conn)

    private ObsHelper insertObs(ObsHelper obs, Connection conn)
        throws SQLException, ObsException {

        // Создадим объект CallableStatement
        CallableStatement cstmt = null;

        try {
                // Подготовим строку символов для вызова
                // PL/SQL-процедуры OBSACTIONS.INSERTOBS()
                String sql =
                   "{call OBSACTIONS.INSERTOBS(?,?,?,?,?,?,?,?,?,?,?)}";

                cstmt = conn.prepareCall(sql);

                // Установим входные параметры
                // для PL/SQL-процедуры OBSACTIONS.INSERTOBS()
                cstmt.setInt(1, obs.obs_id.intValue());
                cstmt.setString(2, obs.when_t);
                cstmt.setString(3, obs.at_time);
                cstmt.setInt(4, obs.produced_id.intValue());
                cstmt.setInt(5, obs.latitude.intValue());
                cstmt.setInt(6, obs.longitude.intValue());
                cstmt.setInt(7, obs.wdspd.intValue());
                cstmt.setInt(8, obs.adj_wdspd.intValue());
                cstmt.setInt(9, obs.wddir.intValue());
                cstmt.setInt(10, obs.pressure.intValue());
                cstmt.setInt(11, obs.station_id.intValue());

                // Необходимо вызвать метод execute() объекта
                // CallableStatement. Если забудем его включить, возникнут
                // достаточно странные ошибки.
                cstmt.execute();

                // Возвратим объект ObsHelper
                return new ObsHelper(obs.obs_id);
```

```
        } // Конец try
        catch (SQLException e) {
             throw e;
        } // Конец catch

        // Выполним очистку
        finally {
                 if ( cstmt != null ) cstmt.close();
    } // Конец finally()

} // Конец insertObs(obs,conn)

// Введем ArrayList из OCEANIC_OBSERVATION_TYPE
public ArrayList insertObs(ArrayList inObs)
     throws SQLException, ObsException {

    int loopVar = 0;

    // Объявим массив объектов ObsHelper
    ArrayList returnArrayList = inObs;

    int arrayListSize = inObs.size();

    Connection conn = null;

    try {
             // С помощью тонкого JDBC-драйвера Oracle
             // подключимся к базе данных
             conn = connectDb();

             // Повторим действия и сохраним ArrayList
             // идентификаторов obs_id в returnArrayList
             while ( loopVar < arrayListSize ) {

                // Получим новый obs_id на каждом шаге
                ObsHelper anObsid = getNewObs(conn);
                // Перенесем объект ObsHelper в obs
                ObsHelper obs = (ObsHelper)inObs.get(loopVar);

                // Заменим obs.obs_id новым anObsid.obs_id
                obs.obs_id = anObsid.obs_id;

                // Вызовем insertObs(..), добавив новую строку
                // в базу данных, и возвратим новый
                // ObsHelper с новым obs_id.
                returnArrayList.add(insertObs ( obs, conn ));

                loopVar++; // Увеличим счетчик
                } // Конец while

             // Уменьшим объект ArrayList до его текущего размера
             returnArrayList.trimToSize();

             // Возвратим ArrayList идентификаторов
             return returnArrayList;
        } // Конец try
        catch (SQLException e) {
             throw e;
        } // Конец catch
        // Выполним очистку
        finally {
                  if ( conn != null ) conn.close();
        } // Конец finally()
        } // Конец insertObs(ArrayList inObs)

    // Этот метод вызывается всеми методами, которым нужно
    // подключиться к базе данных с помощью тонкого
```

```
// JDBC-драйвера Oracle
private static Connection connectDb()
    throws SQLException {

    // Создадим переменную java.sql.Connection
    Connection conn = null;
        DriverManager.registerDriver
          (new oracle.jdbc.driver.OracleDriver());

        // Подключимся к серверу базы данных data-i.com
        conn = DriverManager.getConnection(
          "jdbc:oracle:thin:@data-i.com:1521:ORCL",
           "scott","tiger");
    // Возвратим объект java.sql.Connection
    return conn;
} // Конец connectDb()

} // Конец класса ObsImp
```

Теперь, после построения компонента Observation, нужно скомпилировать исходные файлы Java. Обратите внимание на то, что для некоторых методов компонента в качестве входного параметра и возвращаемого типа использовался объект ArrayList. Тип ArrayList для Java описан в 1.2.x, поэтому для компиляции исходного программного текста следует пользоваться компилятором, соответствующим JDK 1.2.x. Если применять JDK 1.2.x нежелательно, замените тип ArrayList на любой массив Java или тип Vector.

Для компиляции программ воспользуемся командой javac:

```
// Естественно, установим собственный CLASSPATH.
Javac -g helpers\*.java obsserver\*.java
```

Теперь опишем класс ObsImp с помощью SQLJ.

▼ ВНИМАНИЕ! *Следите за тем, чтобы исходный программный текст для реализаций JDBC и SQLJ не присутствовал в одном каталоге. При компиляции программы SQLJ компилятор генерирует исходный файл .java. Если реализация JDBC находится в том же каталоге, она будет перезаписана. Во избежание подобной ситуации исходный текст всех программ, представленных в книге, хранится в каталогах JdbcImplementation и SqljImplementation. Указанные файлы можно скопировать на сайтах www.data-i.com или www.osborne.com.*

Класс ObsImp: реализация с помощью SQLJ

Как и реализация JDBC, эта реализация будет пересмотрена и расширена в главах 4 и 5. Она очень похожа на ту, что строится через JDBC, однако имеет ряд незначительных отличий.

- В SQLJ подключение к базе данных идет с помощью объектов соединения двух типов: интерфейс sqlj.runtime.ConnectionContext и класс sqlj.runtime.ref.DefaultContext. Экземпляр любого из них указывает на местонахождение сервера данных, на котором будет выполняться SQL-операция. С помощью следующего программного фрагмента подключаемся к БД:

```
// Объявим переменную типа DefaultContext
protected static DefaultContext localHostCtx;

// Подключимся к базе данных. Метод Oracle.getConnection()
// возвращает объект DefaultContext.
localHostCtx =
        Oracle.getConnection
            ("jdbc:oracle:thin:@data-i.com:1521:ORCL",
                "scott","tiger",true);
```

- Для реализаций JDBC, где вызываются хранимые функции или процедуры, в программе нужно устанавливать специальную среду. Если не указать любой из установочных операторов или попытаться выполнить их не в том порядке, что описан заранее, будут выданы сообщения об ошибках Java. В SQLJ функция просто вызывается. Например:

```
#sql [localHostCtx] returnId =
        {VALUES ObsActions.getObsId};
```

Обратите внимание на простоту оператора и особенно на то, что среда здесь не устанавливается. SQLJ-объект `localHostCtx DefaultContext` связывается с оператором при выполнении SQL-команды в базе данных. Более подробно о соединительных объектах SQLJ рассказано в главе 5 книги "Oracle8*i* SQLJ Programming", OMcGH, 1999.

- Вызывать процедуры в SQLJ так же просто, как и функции. Например:

```
#sql [localHostCtx]
        { call ObsActions.insertObs(
        :IN obs_id, :IN when_t, :IN at_time, :IN produced_id,
        :IN latitude, :IN longitude, :IN wdspd, :IN adj_wdspd,
        :IN wddir, :IN pressure, :IN station_id)
};
```

Приведем описание класса `ObsImp` с помощью реализации SQLJ:

```
// Имя программы:    ObsImp.sqlj
// Имя программы:    ObsImp.java
// Название:     Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Описание:     Класс Java, который использует SQLJ для
//                   реализации вызываемого клиентами метода
//                   insertObs(), заявленного в интерфейсе
//                   Observation.

package obsserver;

// импортируем helper-классы
import helpers.ObsHelper;
import helpers.ObsException;

// импортируем специфичное для приложения исключение
import java.sql.SQLException;

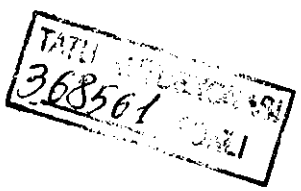
import java.sql.*;
import oracle.sql.*;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.math.*;
import java.util.ArrayList;

// Импортируем модули SQLJ
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.Oracle;
import oracle.jdbc.driver.*;

public class ObsImp implements Observation {
    // Создадим глобальную переменную типа
    // sqlj.runtime.ref.DefaultContext.
    protected static DefaultContext localHostCtx;

    // Конструктор
```



```
public ObsImp() throws SQLException {
    // Подключимся к базе данных
    // с помощью SQLJ-объекта DefaultContext
    connectDb();
    System.out.println("In ObsImp(), I got a connection");
} // Конец конструктора

// Этот метод, который не могут вызывать клиенты, подключается
// к БД и вызывает переопределяемый getNewObs(), получая
// новый obs_id.
private ObsHelper getNewObs()
     throws SQLException, ObsException {

    // Объявим переменную.
    int returnId;

    // Вызовем getObsId() из PL/SQL-модуля ObsActions, чтобы
    // получить новый obs_id.
    try {
         // Для вызова PL/SQL-функции воспользуемся объектом
         // localHostCtx DefaultContext
         #sql [localHostCtx] returnId =
                {VALUES ObsActions.getObsId};

         BigDecimal anObsId = new BigDecimal(returnId);

         // Возвратим объект ObsHelper с помощью нового идентификатора
         return new ObsHelper(anObsId);
    } // Конец try
    catch (SQLException e) {
         throw e;

    } // Конец catch
    catch (java.lang.NullPointerException e) {
         throw e;
    } // Конец catch
} // Конец getNewObs

// Этот метод получает новый obs_id и добавляет новый
// OCEANIC_OBSERVATION_TYPE в таблицу OCEANIC_OBSERVATION_LIST.
private ObsHelper insertObs(ObsHelper obs)
     throws SQLException, ObsException {
    try {
         BigDecimal obs_id = obs.obs_id;
         String when_t = obs.when_t;
         String at_time = obs.at_time;
         BigDecimal produced_id = obs.produced_id;
         BigDecimal latitude = obs.latitude;
         BigDecimal longitude = obs.longitude;
         BigDecimal wdspd = obs.wdspd;
         BigDecimal adj_wdspd = obs.adj_wdspd;
         BigDecimal wddir = obs.wddir;
         BigDecimal pressure = obs.pressure;
         BigDecimal station_id = obs.station_id;

         // Вызовем PL/SQL-процедуру OBSACTIONS.INSERTOBS() с помощью
         // объекта localHostCtx DefaultContext
         #sql [localHostCtx]
            { call ObsActions.insertObs(
              :IN obs_id, :IN when_t, :IN at_time, :IN produced_id,
              :IN latitude, :IN longitude, :IN wdspd, :IN adj_wdspd,
              :IN wddir, :IN pressure, :IN station_id)
            };
         // Возвратим объект ObsHelper
         return new ObsHelper(obs.obs_id);
```

```
        } // Конец try
        catch (SQLException e) {
          throw e;
        } // Конец catch

    } // Конец insertObs(obs)

    public ArrayList insertObs(ArrayList inObs)
         throws SQLException, ObsException {

        int loopVar = 0;
        // Объявим массив объектов ObsHelper
        ArrayList returnArrayList = inObs;

        int arrayListSize = inObs.size();
        try {
             // Повторим действия и сохраним ArrayList
             // из идентификаторов obs_id в returnArrayList
             while ( loopVar < arrayListSize ) {
               // узнаем новый obs_id на каждом шаге
               ObsHelper anObsid = getNewObs();

               // Перенесем объект ObsHelper в obs
               ObsHelper obs = (ObsHelper)inObs.get(loopVar);

               // Заменим obs.obs_id новым anObsid.obs_id
               obs.obs_id = anObsid.obs_id;

               // Вызовем insertObs(..), добавив новую строку в БД,
               // и возвратим новый
               // ObsHelper с новым obs_id.
               returnArrayList.add(insertObs(obs));

               loopVar++; // Увеличим счетчик
             } // Конец while
             returnArrayList.trimToSize();

             // Возвратим ArrayList идентификаторов
             return returnArrayList;
        } // Конец try
        catch (SQLException e) {
          throw e;
        } // Конец catch
    } // Конец insertObs(ArrayList inObs)

    // Этот метод создает экземпляр SQLJ-объекта DefaultContext
    // и сохраняет объект в переменной localHostCtx.
    // Этот метод вызывается, когда клиент создает экземпляр
    // класса ObsImp.
    private static void connectDb()
        throws SQLException {

        // Создадим Default Context для определенного
        // хоста и подключимся к БД.
        localHostCtx =
          Oracle.getConnection
             ("jdbc:oracle:thin:@data-i.com:1521:ORCL",
               "scott","tiger",true);

    } // Конец connectDb()
} // Конец класса ObsImp
```

Теперь нужно скомпилировать исходный программный текст реализации SQLJ. Более подробно о компиляции и выполнении программ SQLJ говорится в приложении F и в книге "Oracle8*i* SQLJ Programming", OMcGH, 1999.

Для компиляции класса ObsImp воспользуемся такой командой:

```
// Установим свой CLASSPATH
sqlj-status ObsImp.sqlj
```

Класс Client

Данный класс пользуется компонентом Observation, описанным в предыдущих разделах и выполняет следующие задачи:

- Объявление глобальной переменной theObservationInterface типа интерфейса Observation.
- Описание конструктора Client(). При конкретизации класса Client его конструктор создает экземпляр класса ObsImp и сохраняет объект в переменной theObservationInterface.
- Создание объекта ArrayList, состоящего из объектов ObsHelper.
- Вызов метода insertObs(), заявленного в интерфейсе Observation и реализованного в классе ObsImp. Метод возвращает приложению Client объект ArrayList, состоящий из идентификаторов.
- Вызов локального метода printObsIds (obsids) для вывода всех новых идентификаторов наблюдений, создаваемых компонентом.

Приведем описание класса Client:

```
/* Имя программы:    Client.java
** Версия:
** Описание:         Клиентское приложение: реализация JDBC
**                   Автономное приложение Java, использующее
**                   компонент Observation.
*/

// Импортируем классы JDBC
import java.sql.*;

// Импортируем helper-классы
import helpers.ObsHelper;
import helpers.ObsException;

// Сделаем интерфейс obsserver.Observation
// видимым для приложения Client
import obsserver.*;

import java.sql.*;
import oracle.sql.*;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.util.ArrayList;

public class Client {

    //Объявим переменную типа Observation
    protected Observation theObservationInterface = null;

    // Конструктор
    public Client() {
        // Получим экземпляр класса ObsImp и сохраним его
        // в переменной theObservationInterface
        theObservationInterface = new ObsImp();
    } // Конец конструктора

    public static void main (String args[])
        throws Exception {
```

```
        // Получим экземпляр класса ObsImp
        Client aClientApp = new Client();
        System.out.println("I got an ObsImp instance");

        // Создадим объект ArrayList
        ArrayList obs = new ArrayList(2);
        System.out.println("I created an ArrayList object");

        // Сохраним набор объектов ObsHelper в объекте obs ArrayList
        while (loopVar++ < 2) {
             BigDecimal obs_id = null;
             String when_t = "16-AUG-2000";
             String at_time = "085500";
             BigDecimal station_id = new BigDecimal(0);
             BigDecimal produced_id = new BigDecimal(1);
             BigDecimal latitude = new BigDecimal(0);
             BigDecimal longitude = new BigDecimal(0);
             BigDecimal wdspd = new BigDecimal(0);
             BigDecimal adj_wdspd = new BigDecimal(0);
             BigDecimal wddir = new BigDecimal(0);
             BigDecimal pressure = new BigDecimal(0);

             // Создадим объект ObsHelper
             ObsHelper obs1 =
                new ObsHelper(obs_id,when_t,at_time,station_id,
                              produced_id,latitude,longitude,
                              wdspd,adj_wdspd,wddir,pressure);

             // Введем обьект ObsHelper в ArrayList
             obs.add(obs1);

        } // Конец while

        System.out.println("I going to call the insertObs() method");
        // Вызовем метод insertObs() класса ObsImp и получим
        // объект ArrayList, состоящий из объектов ObsHelper, которые
        // представляют собой набор идентификаторов наблюдений.
        ArrayList obsids =
             aClientApp.theObservationInterface.insertObs(obs);
        System.out.println("I called the insertObs() success!");

        System.out.println("I going to call the local "
                           +" printObsIds() method");
        // Отобразим наблюдения
        printObsIds(obsids);

        System.out.println("I called printObsIds");

    } // Конец main

    public static void printObsIds(ArrayList obs)
          throws Exception {

             int loopVar = 0;
             int arraySize = obs.size();

             System.out.println("I have Obsids, see size:"
                                +arraySize);

             while ( loopVar < arraySize ) {
                // Считаем объект ObsHelper в объекте ArrayList
                ObsHelper anObsid = (ObsHelper)obs.get(loopVar);
                System.out.println("id: " +anObsid.obs_id);

                loopVar++;
            } // Конец while
```

```
    } // // Конец printObsIds
} // Конец класса Client
```

Теперь скомпилируем и выполним приложение `Client`:

```
// Установим свой CLASSPATH
// Во-первых, скомпилируем программу
javac -g Client.Java

// Во-вторых, выполним программу
java Client
```

Итоги

Итак, мы изучили основы компонентов, программных компонентов и распределенных объектных вычислительных систем. Были представлены главные распределенные объектные парадигмы: CORBA, DCOM, Java RMI и EJB. Мы создали простой компонент для приложения баз данных, обращающийся к БД Oracle8*i* и вводящий массив типа `OCEANIC_OBSERVATION_TYPE` в таблицу `OCEANIC_OBSERVATION_LIST`. А в заключение написали приложение `Client` — автономную программу Java на основе компонента.

В главе 2 речь пойдет о web-приложениях, которые генерируют динамическое содержимое, настраиваемое согласно индивидуальным потребностям. В частности, будут описаны технологии Java Server Pages (JSPs) и сервлетов, генерирующих динамическое содержимое, легко переносимое с платформы на платформу.

ГЛАВА 2

Знакомство с web-приложениями

В предыдущей главе говорилось о различных моделях программирования для распределенных вычислений. В настоящей мы опишем составляющие и архитектуру web-приложений. Интернет породил целое поколение новых прикладных программ, позволяющих общаться через World Wide Web. web-программирование, начавшись с создания развлекательных приложений, быстро "посерьезнело" — и вот уже появились бизнес-приложения уровня предприятия, обладающие широкими функциональными возможностями, надежные и масштабируемые. Объем операций, выполняемых через Интернет (их называют обычно *электронным бизнесом* (e-business), или *электронной коммерцией* (e-commerce)), за последние три года чрезвычайно вырос. Продукты рекламируются и сбываются непосредственно через так называемые web-витрины (store fronts), партнеры обмениваются бизнес-данными в электронном виде, а интенсивность торгового оборота растет благодаря приложениям самообслуживания, например программам банковских расчетов на дому. В этой области произошла настоящая революция, продолжающаяся и по сей день. Рассмотрим отдельные технологии электронного бизнеса.

Новая модель электронного бизнеса была воплощена благодаря совершенствованию web-приложений, вызываемых с удаленного компьютера по сети одним щелчком по клавише мыши. Такие приложения поддерживают интерактивный ввод и представление данных через любой стандартный web-браузер. За удобными для пользователей экранами скрываются транзакционные бизнес-компоненты, взаимодействующие с базами данных, например Oracle8*i*. В этой главе мы раскроем основные понятия и принципы работы web-приложений, а именно:

- Понятие web-сервера, его работу с протоколом HTTP
- Модели сервлетов для генерации динамических web-страниц
- JavaServer Pages, алгоритм обработки web-страниц и преимущества их над сервлетами.
- Компонентную архитектуру построения элегантных и мощных web-приложений.

Понятие web-сервера

На абстрактном уровне web-сервер — это программа, принимающая запросы по сети, на основании параметров запроса выполняющая определенные логические схемы и возвращающая результаты клиентскому приложению. Запросы передаются, как правило, пользователем, который щелкает мышью на *гиперссылках*, или *гиперсвязях* (hyperlinks), web-страницы, отображаемой web-браузером. Ответ, генерируемый web-сервером, обрабатывается и отображается браузером. Общая схема взаимодействия web-сервера и web-браузера по сети представлена на рис. 2.1.

web-сервер *прослушивает* (listens) сеть в порту на той машине, где функционирует. Стандартным механизмом передачи клиентских запросов и ответов сервера по Интернету или через частные внутренние сети (интранет) является протокол HTTP. Входящие HTTP-запросы, направляемые определенной машине или в определенный порт, обрабатываются конкретным web-сервером. На одной машине могут работать несколько web-серверов: пока они прослушивают сеть в разных портах, никакого конфликта не будет.

При обработке HTTP-запроса web-серверу порой приходится выполнять самые разнообразные функции: считывать содержимое файлов, выполнять программы, считывающие данные в базе или записывающие их туда, и др. В частности, он может выполнить *сервлет* (servlet) Java, используя встроенный *двигатель*, или *систему поддержки, сервлетов* (servlet engine) и виртуальную машину Java (Java Virtual Machine), чтобы сгенерировать динамическое содержимое web-страниц. Например, HTTP-запросом может вызываться сервлет, выдающий приветствие и текущее время и динамически обрабатываемый на основании

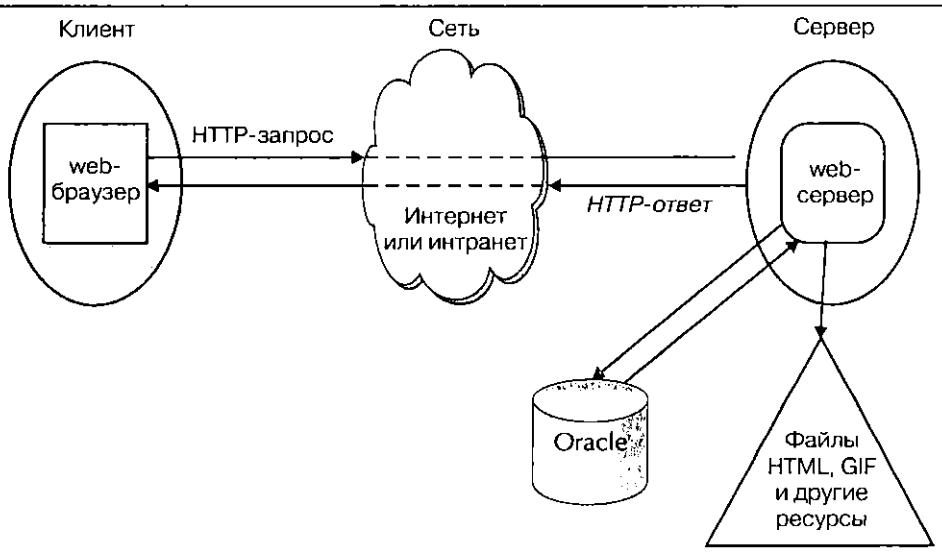


Рис. 2.1. Общая схема web-сервера

реального времени вызова. Пример такого сервлета приведен ниже. Программы, выполняемые web-сервером, называются *серверными приложениями* (server-side).

Самым популярным web-сервером на сегодня считается web-сервер Apache. Это программа с бесплатно распространяемым доступным исходным текстом, коллективно разрабатываемая множеством программистов по всему миру. Его можно бесплатно получить на сайте www.apache.org. Она быстро завоевала популярность благодаря простоте применения, гибкости, доступности для нескольких платформ (в том числе операционной системы Linux) и возможности обращения к исходному тексту в процессе отладки. web-сервер Apache написан на языке C и имеет расширяемую модульную архитектуру, позволяющую вводить в его состав другие функциональные модули.

Вторым по значимости web-сервером считается информационный Интернет-сервер (IIS, Internet Information Server) корпорации Microsoft, обладающий набором мощных функциональных средств. Однако работает он в основном в системах Windows NT, и поэтому область его применения ограничена. Для генерации динамических web-страниц в IIS поддерживаются активные серверные страницы (ASPs, Active Server Pages). Аналогом ASPs на основе Java является технология JavaServer Pages компании Sun Microsystem. В отличие от базирующихся на Java технологий сервлетов и JSP языком создания сценариев для ASPs является JScript или VBScript, а компонентной моделью – COM и DCOM. Доступ к БД в ASPs облегчают активные объекты данных (ADOs, Active Data Objects) от компании Microsoft, в то время как в JSP и сервлетах для той же цели служат API Java Database Connectivity (JDBC) или SQLJ (SQL, встроенный в Java).

web-браузер

web-сервер работает совместно с web-браузером, предоставляющим интуитивно понятный пользовательский интерфейс для web-взаимодействия. Главной функцией последнего является отображение web-страниц, написанных на одном из языков разметки, например на HTML (Hypertext Markup Language – гипертекстовый язык разметки) или XML (Extensible Markup Language – расширяемый язык разметки). На большинстве web-страниц присутствуют гиперссылки, которые связаны с универсальными указателями ресурсов (URLs, Universal Resource Locators), идентифицирующими другие web-ресурсы. Если щелкнуть на гиперссылке мышью, браузер создаст запрос для базового URL. Общий формат URL таков:

- *протокол имя_сервера*[*:номер_порта*] *имя_ресурса*

Так, URL может иметь вид http://www.data-i.com, https://www.mybank.com:8080/accounts, mailto:user@earth.net или ftp://www.myserver.com/aFile.txt. Первой частью URL является спецификация протокола, например http, https (secure HTTP – защищенный HTTP), mailto (протокол электронной почты) или ftp (file transfer protocol – протокол передачи файлов). Имя машины сервера может быть обозначено:

- Числовым Интернет-адресом (IP-адресом), например 144.23.245.123, представляющим четыре байта 32-разрядного адреса машины. Адрес 127.0.0.1 представляет локальную машину, где функционирует браузер.

Либо:

- Символическим именем, например www.data-i.com. Специальное символическое имя localhost указывает обычно на локальную машину, где функционирует браузер.

При желании после имени сервера можно указать номер порта, например 8080. Остальная часть строки URL идентифицирует ресурс на web-сервере на основе иерархической структуры каталогов, отделяя их символом /. Такое иерархическое имя интерпретируется web-браузером.

Алгоритм доставки и место назначения запроса зависят от протокола, имени машины сервера и номера порта, указанных в URL. Например, URL следующего запроса идентифицирует страницу с именем `examples/jsp/chapter2/WelcomeUser.jsp`, которая должна быть доставлена через HTTP с локальной машины и из порта номер 8080:

❒ `http://localhost:8080/examples/jsp/chapter2/WelcomeUser.jsp`

Браузер направляет HTTP-запрос (как правило, метод GET) по частной интранет или Интернет на web-сервер, который, как предполагается, функционирует на хосте и в порту, указанных в URL. web-сервер интерпретирует входящий запрос и посылает обратно страницу в соответствии с URL запроса. HTTP-ответ обрабатывается и отображается браузером. Помимо текста, на возвращаемой странице часто содержатся указатели URL на другие ресурсы, например фото, схемы или рисунки. Браузер выявляет такие указатели на web-странице, а затем запрашивает их содержимое так же, как и в главном запросе. Кроме того, с web-сервера на машину клиента, где работает браузер, можно загрузить и более объемные аудио- и видеоданные с помощью соответствующих дополнительных программных конструкций.

Кроме отображения текстовых и графических данных, большинство web-браузеров способны выполнять прикладные программы. Например, web-браузер Netscape Communicator имеет встроенную виртуальную машину Java (Java Virtual Machine) и запускает небольшие программы Java, апплеты (applets), загружаемые с web-сервера. В отличие от апплетов, которыми занимается клиентский браузер, сервлеты (см. ниже) – это программы Java, выполняемые виртуальной машиной Java на web-сервере. Кроме того, в браузерах, как правило, организуется "объектная модель документов", представляющая отображаемые страницы. Она позволяет работать с документом локально, при помощи клиентских языков создания сценария, подобных JavaScript или VBScript.

Сегодня ведущими web-браузерами являются Netscape Communicator и Microsoft Internet Explorer. Они служат воротами в World Wide Web и в большинство интранет. Различные в деталях, все они используют стандартный протокол HTTP и язык разметки HTML. Если браузер соответствует общепринятым стандартам, взаимодействие с web-сервером не зависит от того, какой браузер применяется. Чаще всего при вызове сервлетов и страниц JSP как из Netscape Communicator, так и из Internet генерируется одна и та же web-страница.

Протокол HTTP

В основе web-общения лежит HTTP – *Hypertext Transfer Protocol* (протокол передачи гипертекста). Это главный протокол Интернета, вытесняющий более старые протоколы, такие как ftp и gopher. Простой и универсальный, он строится на связи запрос/ответ: запросом клиента HTTP генерируется ответ сервера, передаваемый обратно клиенту HTTP. Пара сообщений запрос/ответ иногда называется *транзакцией* (transaction) HTTP. Сообщения передаются посредством TCP/IP (*Transmission Control Protocol/Internet Protocol* – протокол управления передачей данных/протокол Интернета) – базового механизма пересылки информации для Интернета и большинства интранет-сетей.

Сообщение HTTP состоит из заголовков и необязательного тела. Заголовки сообщения HTTP содержат различную информацию о данных, передаваемых в теле. Например, в заголовке Content-type указывается MIME-тип данных (скажем, HTML или XML), а в заголовке Content-length – число байтов в теле сообщения. Заголовки меняются в зависимости от того, является ли сообщение HTTP-запросом или HTTP-ответом.

HTTP-запрос

Обычно HTTP-запрос передается набором URL запроса в окне браузера или щелчком мышью на гиперссылке на web-странице, отображаемой в тот момент браузером. Запрос можно сгенерировать и программным путем. В запросе указывается определенный метод, который подлежит выполнению web-сервером, например GET, POST, HEAD, PUT или DELETE. Они описываются в HTTP-спецификации. Наиболее часто используются два метода: GET и POST. С помощью первого, как правило, считываются страницы на web-сервере. В нем можно задавать параметры запроса в виде пар *имя_параметра=значение_параметра* в *строке символов запроса* (query string), добавляемой к URL запроса. Например, введя следующий URL, вы создадите HTTP-запрос GET с параметром запроса с именем user и значением Tara:

- `http://www.data-i.com:8080/examples/jsp/WelcomeUser.jsp?user=Tara`

Метод POST, напротив, посылает параметры в теле HTTP-запроса, и в URL запроса они не фигурируют. Он более удобен, чем GET, при загрузке больших объемов данных, например содержимого файла, или передаче конфиденциальной информации – номера или пароля банковского счета и т.п. POST обычно используется тогда, когда запрос модифицирует данные на сервере. Метод HEAD применяется реже, чем GET или POST. Он не имеет тела сообщения и предпочтителен, как правило, при передаче информации о состоянии, например даты изменения страницы (актуально для кэширования).

HTTP-ответ

HTTP-ответ генерируется web-сервером в результате обработки HTTP-запроса. Он состоит из необязательных HTTP-заголовков и тела. Заголовки используются web-браузером для отображения содержимого тела. Так, если в заголовке Content-type указан text/html (наиболее часто применяемый MIME-тип для web-страниц), web-браузер обрабатывает HTML-теги (метки) в теле сообщения, форматируя содержимое страницы. В ответе могут присутствовать данные и других типов, например простой текст или двоичный код. Алгоритм отображения, который используется web-браузером, определяется данными, получаемыми в заголовке Content-type.

В заголовке HTTP-ответа всегда содержится код состояния HTTP. Для разных результатов обработки запроса в HTTP описаны стандартные коды состояния. Например, код состояния 200 свидетельствует об успешном обслуживании запроса, а 500 – о внутренней ошибке сервера. Другим часто встречающимся кодом состояния HTTP является код 404, сообщающий о том, что web-сервер не может найти запрашиваемую страницу.

Язык разметки HTML

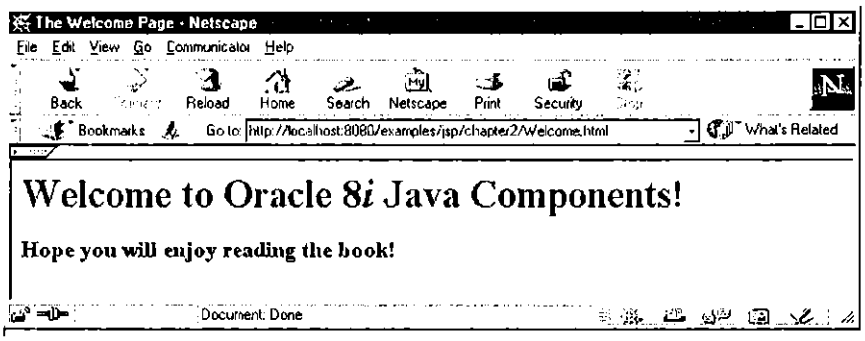
Протокол HTML работает в тесной взаимосвязи с языком разметки HTML. Отображение содержимого, подлежащего отображению на web-странице, в HTML основано на применении специальных тегов разметки со спецификациями HTML. Существуют теги разного предназначения, например для указания способа форматирования текста, для обработки данных, вводимых пользователем, для перехода от одной страницы к другой и др. Используются они, как правило, попарно – каждому начальному соответствует конечный, а текст, заключенный между ними, зависит от их действия. Например, тот, что находится между тегами <B> и </B>, отображается жирным шрифтом. Для тегов характерно вложение в иерархической структуре. В качестве завершенного примера рассмотрим HTML-страницу, представленную в листинге 2.1.

Листинг 2.1. HTML-страница

```
<HTML>
  <HEAD> <TITLE> The Welcome Page </TITLE> </HEAD>
  <BODY BGCOLOR="white">
   <H1> Welcome to Oracle 8<I>i</I> Java Components! </H1>
   <P><B><BIG> Hope you will enjoy reading the book! </BIG></B></P>
  </BODY>
</HTML>
```

В этом HTML-фрагменте теги <HEAD> и <TITLE> показывают, что в окне браузера нужно вывести название The Welcome Page. Затем тег <BODY> отмечает начало реального содержимого страницы. Его атрибут BGCOLOR означает, что фоновым цветом страницы должен быть белый. Блок текста в границах тегов <H1> и </H1> отображается как заглавие первого уровня, причем "i" в строке символов Oracle8*i* будет выделен курсивом, для чего служат теги <I> и </I>. Тег <P> указывает на начало нового абзаца, а <B> – на представление текста жирным шрифтом, пока не встретится соответствующий конечный текст. Тег <BIG> указывает размер шрифта. Начальные и конечные теги выделяют соответствующие разделы текста. web-страница, отображаемая браузером для файла Welcome.html, показана на рис. 2.2.

Рис. 2.2. Отображение браузером страницы Welcome.html



Существуют теги, указывающие нумерованные и маркированные списки, цвета и шрифты текста, а также отображающие таблицы, рамки и формы. HTML-форма – важное средство, часто применяемое при построении интерактивных web-приложений. Это механизм получения данных от пользователя, для чего служат "входные окна" на web-странице, отображаемой браузером, куда пользователь вводит нужную информацию. Кроме того, в форме могут присутствовать кнопки, активизирующие выполнение определенных операций после ввода данных. Щелкнув мышью на одной из них, вы создадите новый HTTP-запрос к серверу. Запрос может вызывать серверную программу, например сервлет или JSP. О применении HTML-форм говорится в части IV этой книги.

Помимо ввода и представления данных, в HTML поддерживается еще и перемещение со страницы на страницу. Одним из наиболее полезных тегов

HTML является <A>, описывающий гиперссылку на URL для некоторых web-ресурсов. Этим простым, но удобным тегом на web-страницах реализуется мощное средство "щелчка и просмотра". Когда для атрибута href задана HTTP-ссылка (связь), <A> указывает браузеру сгенерировать HTTP-запрос, обращенный к указанному URL, когда пользователь щелкает мышью на соответствующей ссылке. Страница, возвращаемая как HTTP-ответ, заменяет ту, что сначала отображалась в окне браузера. Это позволяет перемещаться по WWW (заниматься "web-серфингом") со страницы на страницу, причем они могут находиться на разных машинах, нередко в абсолютно разных частях света. Тесно взаимодействуя, HTTP и HTML произвели настоящую революцию в использовании Интернета.

Генерация динамического содержимого на web-страницах

Сначала web-серверы работали с web-страницами, содержащими лишь статичные данные, подобные HTML-файлам и изображениям. Однако вскоре этого оказалось недостаточно. Чтобы web-приложение нашло практическое применение, оно должно генерировать данные, специфичные для пользователя. Например, приложение по банковским расчетам на дому должно показывать состояние счета для конкретного номера, вводимого в HTTP-запросе. Всего лишь несколько лет назад единственным механизмом динамической обработки содержимого web-страниц был общий шлюзовый интерфейс (CGI, *Common Gateway Interface*). Сегодня доступны значительно усовершенствованные модели серверного программирования, но CGI по-прежнему поддерживается большинством web-серверов. С его помощью web-сервер может выполнять серверные прикладные программы. Однако недостатком CGI является то, что для выполнения каждой запрошенной программы создается новый процесс, а это ухудшает масштабируемость. Вслед за CGI появились более эффективные, но специфичные для платформ технологии, подобные ISAPI от Microsoft или NSAPI от Netscape. В Active Server Pages от Microsoft реализован другой подход, но применение этого средства ограничено платформой Windows. В Netscape также поддерживается серверное средство сценариев – SSJS, использующее для генерации динамических web-страниц программные фрагменты JavaScript.

На сегодня самыми популярными технологиями генерации динамического содержимого являются сервлеты Java и JavaServer Pages (JSPs). Основанные на Java, программы для сервлетов и JSP могут в полной мере использовать мощь, переносимость, безопасность, модульность и расширяемость объектно-ориентированной платформы Java. Об этих методах программирования, а также об их удобстве с точки зрения разработки серверных web-приложений говорится ниже. Архитектура JSP основана на сервлетах, и поэтому понимание принципов действия сервлетов важно при изучении JSP.

Модель сервлетов

Опишем вкратце модели сервлетов.

Сервлет (servlet) Java – это серверная программа, вызываемая web-сервером для обслуживания HTTP-запросов. Он работает в виртуальной машине Java (Java Virtual Machine) на web-сервере и обычно выполняет какие-то вычисления, генерируя содержимое HTTP-ответа. По сравнению с CGI и другими методами серверного программирования сервлеты лучше, потому что:

- В отличие от программы CGI, при каждом вызове сервлета не создается новый процесс. Сервлеты вызываются, как правило, посредством потоков, обслуживающих отдельные HTTP-запросы. Число одновременно порождаемых потоков можно ограничить, что помогает избежать перегрузки сервера.
- Это обычные программы Java, транслируемые в не зависящий от платформы байтовый код, и поэтому их можно автоматически переносить

на любую машину, где применяется Java. Работают они в области, защищенной границами виртуальной машины Java, и не могут быть причиной нарушения правил доступа к памяти и вызывать аварии сервера.

- Серверы имеют полный доступ к мощным прикладным программным интерфейсам Java, таким как Enterprise Java Beans, Java Mail, JNDI и RMI.

Технология сервлетов достаточно молода – спецификация Servlet 2.0 была окончательно одобрена в 1998 году. Тем не менее модель сервлетов и ее реализации быстро укрепились, в первую очередь благодаря их популярности среди web-разработчиков и энергичной поддержке со стороны производителей. Спецификация API Servlet 2.2 была опубликована в декабре 1999 г. Следующая версия API (2.3) создается во время написания данной книги так называемым Java-сообществом (JCP, Java Community Process), основанным компанией Sun Microsystems.

Написание простого сервлета

Далее приведен пример простого сервлета HelloServlet, выдающего приветствие пользователю вместе с текущим временем. Этим примером иллюстрируется базовая структура сервлета и основные этапы обработки HTTP-запроса. Приведены и пояснения для разных этапов.

```
/** Имя программы:   HelloServlet.java
** Назначение:       Распечатка приветствия и текущего времени
**/
package chapter2

import javax.servlet.*;          // (см. пояснение 1)
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;

public class HelloServlet extends HttpServlet {      // (см. пояснение 2)
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)    // (см. пояснение 3)
                throws ServletException, IOException {
        PrintWriter out= response.getWriter();       // (см. пояснение 4)
     response.setContentType("text/html");           // (см. пояснение 5)
        out.println("<HTML>");   // (см. пояснение 6)
     out.println("<HEAD><TITLE>The Hello Servlet</TITLE></HEAD>");
     out.println("<BODY BGCOLOR=\"white\">");
     out.println("<H2>Hello " + request.getParameter("user") +
                 ", how are you?</H2>"); //   (см. пояснение 7)
     out.println("<P><B>The current time is " + new java.util.Date());
                                          // (см. пояснение 8)
     out.println("<P>Hope you have a nice day! </B>");
     out.println("</BODY>");
     out.println("</HTML>");
     out.close();                    // (см. пояснение 9)
    }
}
```

Пояснения к HelloServlet.Java:

1. Первые два оператора `import` подключают библиотеки из спецификации сервлета, которые являются частью пакетов `javax.servlet` и `javax.servlet.http`, в первую очередь предназначенного для работы с основанными на HTTP сервлетами. Обратите внимание на префикс пакетов `javax`: он говорит о том, что данный пакет является частью расширенной структуры Java и не входит в состав базовых библиотек Java (префикс которых – `java`). Набор классов и интерфейсов пакетов `javax.servlet` и `javax.servlet.http` для спецификации Servlet 2.2 приведен в приложении А.

2. Этот оператор объявляет класс HelloServlet. Как правило, сервлет расширяет абстрактный класс javax.servlet.http.HttpServlet, который описывает такие методы, как doGet(), doPost() и service(). Методы doGet(), doPost() и подобные им do...() класса HttpServlet предоставляют стандартные реализации соответствующих методов HTTP-запроса. Один из них или несколько чаще всего переопределяется конкретным классом сервлетов, например HelloServlet. Метод service() выступает в роли диспетчера для соответствующих обработчиков запроса, например doGet() и doPost(). Он довольно редко переопределяется классами сервлетов.
3. Этот оператор описывает метод doGet(). У данного метода два аргумента: HTTP-объект request и HTTP-объект response типа HttpServletRequest и HttpServletResponse соответственно. Оба они описаны в пакете javax.servlet.http. При выполнении тела метода могут устанавливаться исключительные ситуации типа javax.servlet.ServletException или java.io.IOException.
4. Этот оператор извлекает объект PrintWriter – часть HTTP-объекта response. Объект PrintWriter используется для записи символьных данных, которые должны быть возвращены в качестве результата.
5. Этот оператор устанавливает MIME-тип генерируемой страницы. Типичные значения – text/html или text/xml, указывающие на HTML- или XML-содержимое. В значении можно задавать необязательную кодировку набора символов результата, например text/html; charset=koi8-r. Стандартной кодировкой набора символов является ISO-8859-1, называемая также Latin-1.
6. Этот оператор – тег <HTML>, указывающий на начало генерируемой страницы.
7. Этот оператор извлекает значение параметра с именем из HTTP-объекта request, вызывая метод getParameter(), и выводит приветственное сообщение.
8. Этот оператор, создавая новый экземпляр класса java.util.Date(), определяет текущее время и выводит его в ответе.
9. Этот оператор закрывает объект PrintWriter, указывая на окончание потока данных ответа.

Выполнение HelloServlet.java Установите программу на своем web-сервере, следуя инструкциям, предлагаемым в приложении А. Процесс состоит из компиляции программного текста сервлета и размещения скомпилированного класса Java в нужном месте, доступном web-серверу, где и будет выполняться сервлет. Например, для сервера Tomcat, функционирующего на машине UNIX, сервлет можно установить в приложении examples – достаточно дать команду оболочки:

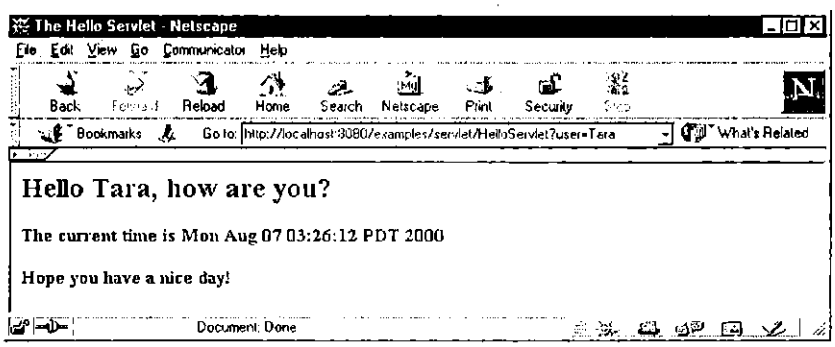
```
# Сценарий UNIX для компиляции HelloServlet на сервере Tomcat
javac -d ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes \
      -g -classpath ${TOMCAT_HOME}/lib/servlet.jar \
      HelloServlet.java
```

Здесь предполагается, что переменная среды TOMCAT_HOME установлена как корневой каталог системы Tomcat. На рис. 2.3 показаны выходные данные, генерируемые при вызове HelloServlet вводом соответствующего URL, например:

```
http://localhost:8080/examples/servlet/chapter2.HelloServlet?user=Tara
```

Обратите внимание на то, что при описании класса сервлета в URL используется полное уточненное имя chapter2.HelloServlet. Параметру user строкой запроса URL присваивается значение Tara.

Рис. 2.3.
Выходные данные HelloServlet



Модель выполнения сервлета

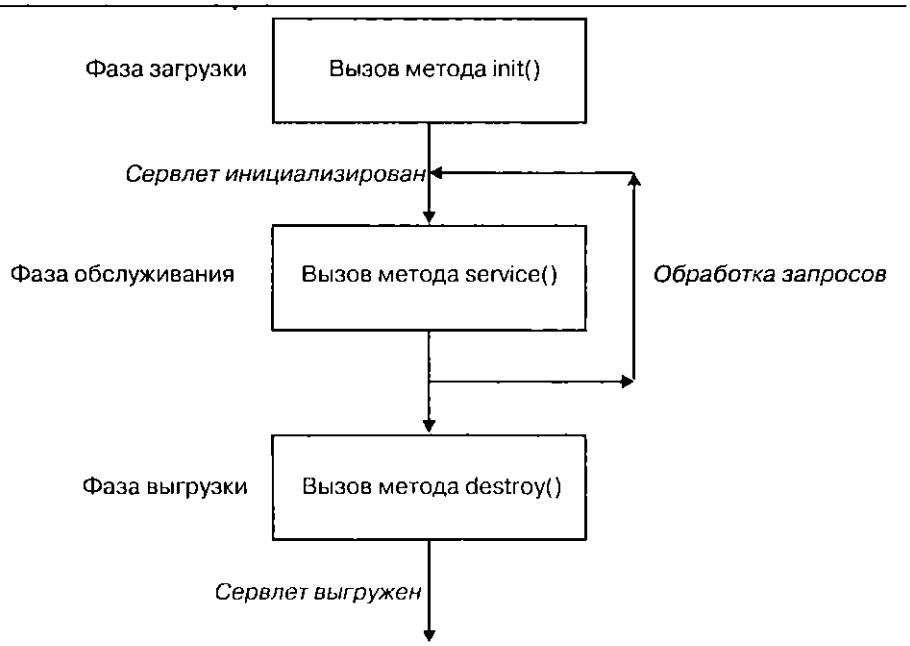
Теперь, после написания сервлета, нас интересует, каким образом он выполняется? Напомним, что сервлет – это обычный класс Java, т.е. его можно скомпилировать в виде байтового кода при помощи стандартного компилятора Java. Этап компиляции выполняется, как правило, вручную: другими словами, разработчик сервлета компилирует класс за отдельный шаг. Для страниц JSP (это будет показано ниже) этап компиляции автоматизирован. Сгенерированный класс нужно сделать доступным для виртуальной машины Java. Однако прежде чем сервлет заработает, необходимо выполнить ряд дополнительных операций. Например, нужно создать экземпляр класса сервлета. Возникает ряд важных вопросов. Будет ли обслуживать все HTTP-запросы один экземпляр класса сервлета или потребуется несколько экземпляров? Когда уничтожается экземпляр сервлета? Попробуем ответить на них.

Двигатель сервлетов *Двигатель* (engine) сервлетов, называемый также *контейнером* (container) или *исполнителем* (runner) сервлетов,– это программа, выполняющая установленные сервлеты. Двигатель сервлетов отвечает за:

- Создание и инициализацию экземпляров сервлетов
- Вызов сервлетов, основанных на HTTP-запросах, и формирование HTTP-ответов из генерируемых выходных данных
- Уничтожение экземпляров сервлетов и результатов их работы

Процесс загрузки, создания, выполнения и, наконец, уничтожения экземпляра сервлета называется его *жизненным циклом* (life cycle). Различные стадии жизненного цикла сервлета представлены на рис. 2.4.

Рис. 2.4.
Жизненный цикл сервлета



Как правило, двигатель сервлетов создает один экземпляр класса сервлета, хотя порой используется пул экземпляров сервлетов. Точный момент создания экземпляра определяется конфигурацией web-сервера. Например, это возможно при запуске web-сервера или первом вызове сервлета через HTTP-запрос. Независимо от времени создания, двигателем сервлетов всегда вызывается метод `init()`, и только потом сервлет начинает обрабатывать запрос. Метод `init()` выполняет все логические процедуры инициализации, необходимые для обслуживания сервлетом клиентских запросов. Так, сервлет может установить в своем методе `init()` соединение с базой данных и пользоваться этим соединением для выполнения последующих SQL-запросов. Аналогичным образом метод `destroy()` сервлета всегда вызывается перед удалением экземпляра и служит для выполнения всех необходимых операций очистки, например завершения ранее установленного соединения с БД.

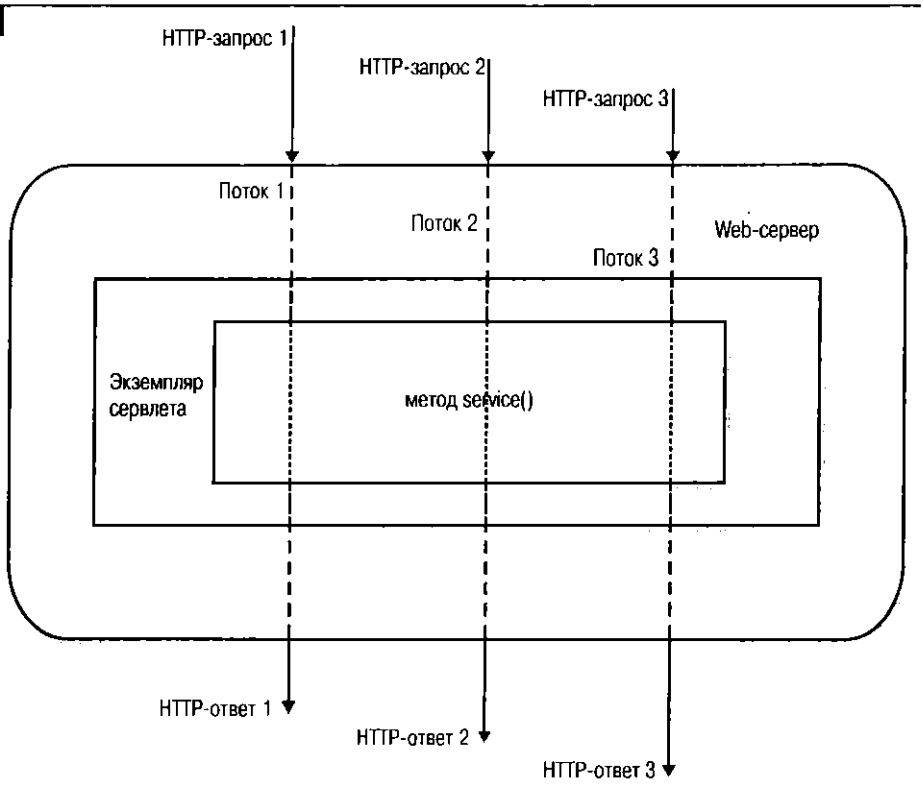
После создания и инициализации экземпляр сервлета готов к обслуживанию запросов. При поступлении запроса для данного сервлета двигатель обычно порождает новый поток для обработки запроса. Поток вызывает метод `service()` класса сервлета. Чаще всего сервлеты расширяют класс `javax.servlet.http.HttpServlet` и реализация метода `service()` наследуется из этого суперкласса. Сигнатура метода `service()` такова:

```
protected void service(HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException
```

Этот метод выступает в роли "диспетчера" – он определяет тип HTTP-запроса (например, `GET` или `POST`) и просто вызывает подходящий метод (`doGet()` или `doPost()`) для сервлета. Сигнатуры этих методов те же, что и у метода `service()` (см. пример метода `doGet()` для `HelloServlet` выше).

Теперь посмотрим, что произойдет, если одновременно поступит несколько HTTP-запросов или если новый запрос придет во время обработки предыдущего. В этом случае двигатель сервлетов обычно использует другой поток выполнения, вызывая метод `service()` того же экземпляра сервлета, даже во время обработки первого запроса (рис. 2.5). В этой схеме подразумевается, что код обработки запросов в сервлете способен

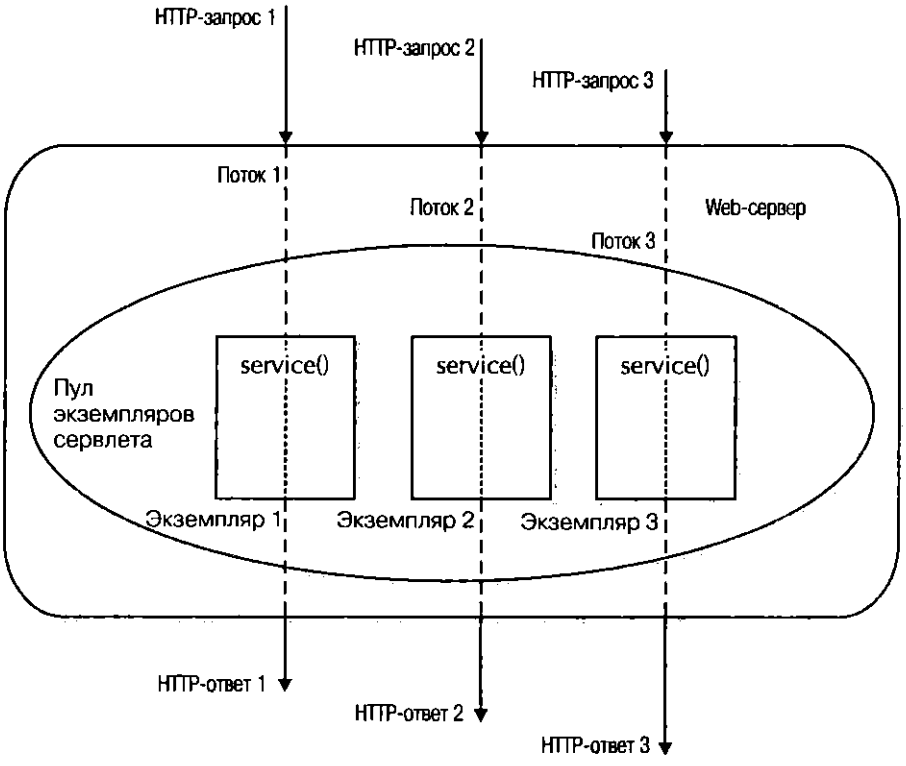
Рис. 2.5. Многопоточный доступ к одному экземпляру сервлета



обслуживать одновременное обращение со стороны нескольких потоков (исключением из этого правила является ситуация, когда классом сервлета реализуется интерфейс `SingleThreadModel`; см. следующий абзац). Следовательно, при написании сервлета *нужно обязательно следить за тем, чтобы программный текст (код) был безопасен* с точки зрения потоков. В Java для надлежащей защиты программ от конфликтов при многопоточном выполнении следует использовать оператор `synchronized`. В частности, нужно синхронизировать доступ к переменным экземпляров класса сервлета, например к переменной, представляющей соединение с базой данных. Нередко необходимо группировать некоторые операции, выполняемые над базой, делая их частью атомарной транзакции, соответствующей одиночному HTTP-запросу. Таким образом, доступ к соединению с базой данных нужно координировать среди нескольких потоков, обслуживающих разные HTTP-запросы.

Запрет многопоточного доступа в некоторых случаях снижает производительность системы и эффективность обработки запросов. Так, при обслуживании отдельных транзакций для каждого запроса к базе данных весь код в методе `doGet()` или `doPost()`, возможно, придется синхронизировать, что приводит к последовательной обработке одновременно поступающих запросов. В подобных ситуациях для сервлета приемлем другой режим выполнения программ, называемый *однопоточной* (single-thread) моделью. В такой схеме (рис. 2.6) два запроса никогда одновременно не передаются одному и тому же экземпляру сервлета, и проблем с синхронизацией не возникает. Вместо этого двигатель создает для данного сервлета пул экземпляров и, обслуживая новый запрос, использует свободный экземпляр из этого пула. Подобная схема для сервлета допустима при реализации интерфейса `javax.servlet.SingleThreadModel` – пустого интерфейса без методов, выступающего в роли простого указателя на альтернативный алгоритм выполнения программ. Сервлет, где реализован такой интерфейс, можно, без сомнения, считать безопасным с точки зрения потоков.

Рис. 2.6. Выполнение сервлета в однопоточной модели



Использование сеансов сервлетов

По своей природе HTTP – универсальный протокол, не сохраняющий состояние. Другими словами, HTTP-запрос никак не связан с любым другим, даже если оба поступают от одного и того же клиента. Однако при взаимодействии web-приложений состояние часто должно *сохраняться*. Например, в web-приложении для банковских расчетов на дому нередко предусматривается регистрационная страница, где пользователь вводит номер и пароль своего счета, а эта информация необходима для следующего HTTP-запроса, когда пользователь запрашивает состояние счета. Другим известным примером взаимодействия с сохранением состояния являются сетевые магазины, когда пользователь собирает товары в виртуальную "корзину", заключающую в себе несколько запросов (см. главу 11). Все это требует ввода понятия сеанса (session), в котором, по существу, группируются серии запросов от одного клиента.

Посмотрим, как организуется работа с сеансами. Каждый сеанс идентифицируется на сервере уникальным ключом. Ключ сеанса может содержаться в сообщении HTTP-ответа, сохраняться клиентом и вводиться в последующие запросы, обращенные к данному серверу, указывая текущий сеанс. Распространены три способа контроля над сеансами в сервлетах:

- Через домашние данные
- Путем перезаписи URL
- Посредством встроенного в сервлеты API для контроля над сеансами

Мы кратко опишем их ниже. Сеансы можно реализовывать и менее известными способами, которые выходят за рамки нашей книги. Заметим, что сеанс может быть завершен явно сервлетом или неявно по истечении времени сервером. По истечении срока действия сеанса его ключ становится недостоверным, даже если HTTP-запрос с данным ключом получен позже.

Домашние данные (cookie) Это просто идентификационные данные сеанса, посылаемые клиенту в HTTP-заголовке `Set-Cookie` как часть сообщения HTTP-ответа. Сервлет может направить домашние данные явно, вызвав метод `addCookie()` объекта `HttpServletResponse`. Клиент вправе сохранить их и посылать во всех последующих запросах назад серверу для идентификации сеанса. Однако порой web-браузер не имеет поддержки домашних данных, либо пользователь явно запрещает их в браузере. В таких случаях применяется метод перезаписи (переадресации) URL.

Перезапись URL При использовании этого метода все указатели URL, включаемые в ответ сервера, видоизменяются так, чтобы нести информацию о ключе сеанса. Например, идентификатор пользователя добавляется к URL в виде строки запроса:

❒ `http://localhost:8080/examples/servlet/shopOnline?userid=Pam`

Когда впоследствии пользователь щелкает мышью на гиперссылке HTTP на web-странице, отображаемой браузером, для ссылки применяется этот расширенный URL. Сервлет на другой стороне извлекает идентификатор сеанса из строки запроса URL. Этот метод неудобен тем, что в ответе приходится модифицировать все указатели URL, вводя ключ сеанса, а это дополнительная нагрузка на сервлет. Еще хуже то, что теряется конфиденциальность, поскольку идентификатор сеанса явно или неявно присутствует на генерируемой странице.

Сеансовый API сервлетов В прикладном программном интерфейсе (API) сервлетов Java предусмотрена встроенная поддержка сеансов. С этим API должен работать любой web-браузер, поддерживающий сервлеты. С помощью данного API программист, создающий сервлеты, изолируется от всех деталей процесса слежения за сервлетами, реализуемого web-сервером; отправка домашних данных или перезапись URL могут происходить в фоновом режиме.

Посмотрим, как программист использует сеансовый API. Хранить и считывать данные, относящиеся к сеансу работы, призван объект HTTPSession – экземпляр типа javax.servlet.http.HttpSession, который можно извлечь из HTTP-объекта request, вызвав его метод getSession(boolean create). Если аргумент boolean метода getSession() истинен (true), для данного запроса запускается новый сеанс (при условии, конечно, что он уже не открыт). Если же вызвать getSession(false), возвращается значение null (при условии, что сеанс не существует). Сервлет, создающий сеанс, называется сервлетом с *сохранением состояния* (stateful). Сервлеты, *не сохраняющие состояния* (stateless), сеансами не пользуются.

После извлечения объекта сеанса вызовом getSession() сервлет использует его для записи данных сеанса, например о соединении с БД или о корзине магазина. Затем к ним смогут обращаться все сервлеты, вызываемые в одном HTTP-сеансе. Данные хранятся и считываются в объекте session с помощью методов setAttribute() и getAttribute(). Например, в следующем программном фрагменте сервлета сначала получается объект session, инициализируется объект соединения с базой данных conn, а затем с объектом conn связывается атрибут сеанса с именем dbConnection:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
            throws ServletException, IOException {
... ...
// Сначала получим объект session
javax.servlet.http.HttpSession session request.getSession(true);
// Инициализируем соединение с БД
java.sql.Connection conn = ... ...;
// Сохраним сведения о соединении с БД в объекте session
session.setAttribute("dbConnection", conn);
... ...
}
```

Впоследствии описатель соединения с базой данных можно считать в объекте session с помощью ключа dbConnection метода getAttribute():

```
java.sql.Connection conn = (java.sql.Connection)
                           session.getAttribute("dbConnection");
```

Обратите внимание: считанный объект необходимо приводить к соответствующему типу. Существует также метод removeAttribute(String name) типа HttpSession, применяемый для удаления хранимого значения.

Контекст сервлетов

В некоторых случаях web-приложению приходится сохранять не состояние сеансов, а состояние, связанное со всем приложением. Например, пул открытых соединений с БД порой нужно делать доступным всем сеансам web-приложения. Так из чего же именно состоит web-приложение? Это совокупность программ и ресурсов, выполняющих определенную задачу, например сообщение о расходах. web-приложение может состоять из статичных HTTP-страниц, логотипов и изображений, сервлетов и страниц JSP и таких программных компонентов, как JavaBeans и Enterprise JavaBeans, вспомогательных служебных классов и прочих ресурсов. Как правило, у каждого web-приложения на web-сервере свой маршрут. Так, создающее отчеты о расходах может находиться на сайте http://www.data-i.com/webexpense. Все указатели URL с этим префиксом ссылаются на программы и ресурсы в границах данного приложения. Такой маршрут называется *корнем приложения* (application root) и обычно уникален для того или иного web-приложения.

В реализациях Servlet 2.2 каждому из них соответствует собственный экземпляр ServletContext. Объект ServletContext имеет тип javax.servlet.ServletContext. Описатель этого объекта могут считывать все сервлеты и страницы JSP приложения вызовом метода getServletContext(). В объекте ServletContext существует множество полезных методов для различных целей, в том числе призванных:

- Получать сведения о web-сервере. Например, методы `getServerInfo()` и `getMajorVersion()` возвращают сведения о двигателе сервлетов и поддерживаемой версии.
- Получать сведения о web-приложении. Например, `getInitParameter()` и `getInitParameterNames()` возвращают сведения о параметрах инициализации, связанных с объектом `ServletContext`. Метод `getRealPath()` возвращает сведения о реальном маршруте для данного виртуального маршрута, например `/welcome.html`.
- Записывать сообщения в файл журнала методом `log()`.
- Распределять данные по всему приложению. Данные с общим состоянием можно записывать в объект `ServletContext`, считывать в нем и удалять из него с помощью методов `setAttribute()`, `getAttribute()` и `removeAttribute()`. Они аналогичны соответствующим методам объекта `HttpSession`, о котором говорилось выше.
- Получать объект `RequestDispatcher` для вызова других сервлетов и страниц JSP в web-приложении (см. главу 10).

Полный набор методов объекта `ServletContext` приведен в приложении А. В следующем программном фрагменте показан пример вывода сведений о сервере, сохранения переменной с именем `connPool` в атрибуте контекста с именем `dbConnectionPool` и занесения информации в журнал:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
            throws ServletException, IOException {


ServletContext scontext = getServletContext();
PrintWriter out = response.getWriter();
out.println("Here is my server information:" +
             scontext.getServerInfo());

... ... ...
scontext.setAttribute("dbConnectionPool", connPool);
scontext.log("Database connection pool created.");
...
}
```

Реализации сервлетов

web-сервером Apache активно используется двигатель сервлетов Apache JServ. web-сервер Apache и двигатель сервлетов JServ применяются в прикладном Интернет-сервере Oracle (iAS, Oracle Internet Application Server) версии 1.0. Однако JServ, где реализован лишь интерфейс Servlet 2.0, вытесняется такими двигателями, как Tomcat, который поддерживает API Servlet 2.2. Тем не менее иметь представление о JServ весьма полезно, поскольку он активно используется во многих средах. Apache JServ состоит из двух отдельных компонентов, один из них написан на языке C, а другой — на Java. C-компонент называется `mod_jserv` и соединяет web-сервер Apache с Java-компонентом, который и есть настоящий двигатель сервлетов.

Стандартом для API Servlet 2.1 является web-сервер Java (Java Web Server) компании Sun Microsystems, который появился в составе инструментария разработки JavaServer Web Development Kit (JSWDK). Эта реализация вытесняется проектом Tomcat — совместной разработкой Apache Software Foundation и Sun Microsystems. Tomcat — это стандартная система для спецификации Servlet 2.2. Существуют и другие двигатели сервлетов от других производителей, например JRun от Allaire. В Oracle8*i* редакции 8.1.7 двигатель сервлетов Oracle Servlet Engine функционирует в границах БД. В нем реализован API Servlet 2.2 и работает виртуальная машина Java, встроенная в Oracle8*i*. Достоинство функционирования программ сервлетов и JSP внутри БД — их выполнение в непосредственной близости от SQL-данных (с использованием серверного Oracle-драйвера JDBC) в защищенной среде базы.

Знакомство с JavaServer Pages

Технология JavaServer Pages (JSP) встроена в описанную выше модель сервлетов и основана на смешивании программных конструкций Java со статичными шаблонами HTML или XML. Логические схемы Java призваны генерировать динамическое содержимое страницы, а язык разметки — структурировать и представлять данные. Главная цель проекта JSP — строго отделить текст HTML или XML от программ Java специально для web-дизайнеров, которые незнакомы с программированием на Java. С такой целью в спецификации JSP предусмотрены встроенные средства компонентного программирования при помощи JavaBeans и библиотек тегов. Краткий обзор средств дан в следующем разделе, а более подробный рассказ см. в части IV.

Технология JSP достаточно нова, но уже получила широкое распространение в среде прикладных web-дизайнеров, в промышленности и быстро развивается. Последняя версия спецификации JSP 1.1 появилась в ноябре 1999 г., а следующая (1.2) разрабатывается во время написания этой книги.

В настоящее время JSP поддерживается web-продуктами нескольких производителей, в том числе Oracle. В основе web-сервера Oracle Internet Application Server (iAS), увидевшего свет в июне 2000 г., лежат web-сервер Apache и двигатель сервлетов Jserv. В состав iAS входит собственная реализации JSP компании Oracle. С Oracle*8i* версии 8.1.7 интегрирован двигатель Servlet 2.2 и JSP 1.1, выполняющий программы JSP и сервлетов непосредственно в БД (с использованием внутренней виртуальной машины Java). Стандартные реализации для спецификаций JSP 1.1 и Servlet 2.2 предоставляются сервером Tomcat, который разрабатывается Apache Software Foundation (www.apache.org) в тесном сотрудничестве с Sun Microsystems и другими промышленными партнерами. Ранее стандартная реализация JSP 1.0 была выпущена в Sun Microsystems как часть Java Web Server (JWS) и JavaServer Web Development Kit (JSWDK). Инструкции по установке JSP-двигателя Oracle с двигателями сервлетов JServ, JWS и Tomcat приведены в приложении E.

Базовая программа JSP

Пример простого сервлета `HelloServlet` был приведен в предыдущем разделе. В нем считывалось имя пользователя, передаваемое в качестве параметра HTTP-запроса, и выводились приветствие и текущая дата. Изменение сервлета на JSP поможет понять, в чем разница между двумя моделями программирования. Соответствующая JSP-программа выглядит так:

```
<!-- Имя программы: WelcomeUser.jsp
  -- Назначение: Поприветствовать пользователя и вывести
  -- текущее время.
<HTML>
    <HEAD><TITLE>The WelcomeUser JSP </TITLE><HEAD>
    <BODY>
        <H2> Hello <%= request.getParameter("user") %>, how are you?</H2>
        <P><B> The current time is <%= new java.util.Date() %>.
        <P>    Hope you have a nice day! </B></P>
    </BODY>
</HTML>
```

Обратите внимание: программный текст Java (выделен жирным шрифтом) на странице JSP встроен в программный текст HTML с помощью специальной конструкции `<% ... %>`, которая, в частности, применяется для встраивания выражений Java. Выражения обрабатываются на этапе выполнения программы, их значения преобразуются в строки символов и выводятся как часть ответа. Встраивать можно и операторы Java, называемые *скриптлетами* (scriptlets), выполняя с их помощью условные и другие динамические логические схемы страницы JSP.

В примере `WelcomeUser.jsp` в результате вызова метода `request.getParameter("user")` параметр с именем user считывается из HTTP-объекта `request` точно так же, как и в примере `HelloServlet.java`. Однако объект `request` не нужно описывать явным образом — в JSP он доступен и без этого (см. главу 10).

Очевидно, что JSP-программа гораздо компактнее и проще, чем HelloServlet. В WelcomeUser.jsp статический текст и динамические выражения не нужно выводить, вызывая метод println() явным образом, как в сервлете. В нашем примере для JSP отсутствуют и объявления методов, а исключительные ситуации не устанавливаются. Ясно, что выполняется большой объем работы, но он нам не виден. Разберем, как следует компилировать и выполнять JSP.

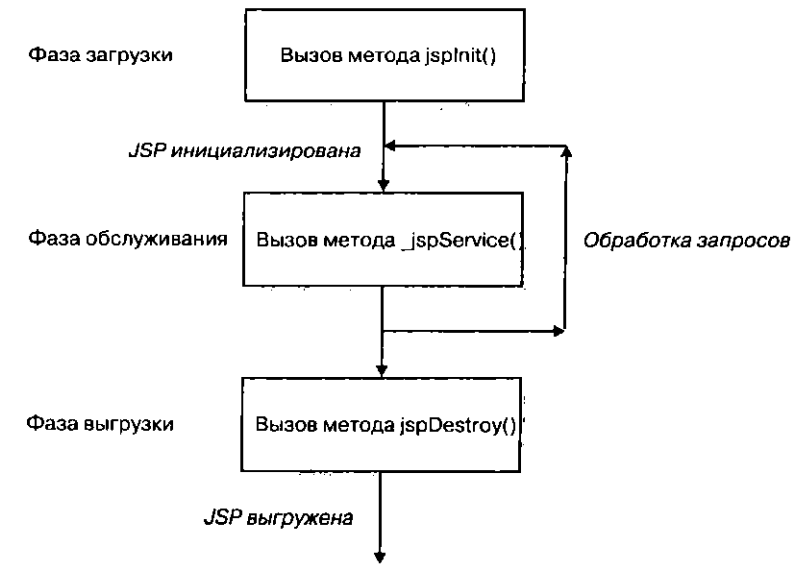
Модель выполнения JSP

Все JSP-программы выполняются двигателем JSP, работающим в границах web-сервера. Обычно двигатель JSP функционирует в качестве сервлета, вызываемого тогда, когда URL запроса заканчивается расширением .jsp. Такая схема вызова называется *отображением расширений* (extension mapping).

После вызова двигатель JSP сначала находит JSP-файл, на который ссылается URL. Если файл не был преобразован заранее, двигатель JSP превращает "смешанный" исходный текст JSP в код сервлетов Java. Генерируемый код соответствует правилам, установленным спецификацией JSP, и в нем используются классы JSP этапа выполнения (см. главу 10). Характеристики кода, обычно не слишком интересные программисту JSP, помогают понять принципы алгоритма выполнения JSP-конструкций. Короче говоря, генерируемый класс сервлета реализует стандартный интерфейс javax.servlet.jsp.HttpJspPage. Этим интерфейсом расширяется супертип javax.servlet.jsp.JspPage, имеющий два метода: jspInit() и jspDestroy(). Они вызываются двигателем JSP для инициализации и уничтожения генерируемого сервлета. В интерфейсе javax.servlet.jsp.HttpJspPage содержится еще один метод – _jspService(), генерирующий выходную страницу и такой же полезный в многопоточной обработке HTTP-запросов, как метод service() в сервлетах. В _jspService() содержатся динамические конструкции, например JSP-выражения и скриптлеты, встроенные в страницу JSP. Генерируемый сервлет компилируется в байтовый код Java С помощью стандартного компилятора Java. Наконец, сгенерированный класс сервлета загружается и выполняется двигателем сервлетов, а результаты возвращаются клиенту.

Последующие или одновременно поступающие запросы обращаются к тому же методу _jspService(), используя новый поток, порождаемый двигателем сервлетов. Так что нужно следить за безопасностью потоков в программном тексте JSP – если, например, присутствуют объявления переменных уровня класса. В качестве альтернативы в JSP можно установить однопоточный режим выполнения программ, описанный выше для сервлетов. Нужный режим выбирается на странице при помощи JSP-директивы (*directive*, см. главу 10). После окончательной выгрузки JSP-класса (например, при остановке web-сервера) двигателем JSP вызывается метод jspDestroy(). Очевидно, что жизненный цикл JSP очень похож на жизненный цикл сервлета, описанный выше (рис. 2.7).

Рис. 2.7. Жизненный цикл JSP



Рассмотрим внимательнее процесс вызова JSP путем указания URL в web-браузере. Возьмем, к примеру, URL:

- `http://localhost:8080/examples/jsp/chapter2/WelcomeUser.jsp?user=Tara`

После передачи этого запроса внутри web-сервера обычно происходит следующее:

1. Вызывается двигатель JSP, так как URL запроса (без строки запроса `user=Tara`) заканчивается расширением `.jsp`.
2. Двигатель JSP ищет на web-сервере файл `WelcomeUser.jsp` по иерархическому маршруту `examples/jsp/chapter2`. Если файл не находится, в ответе возвращается код состояния 404.
3. После нахождения исходного JSP-файла двигатель JSP решает, нужно ли тот преобразовывать, — например, когда отсутствует файл класса или источник изменился со времени последнего преобразования. Если в преобразовании нет необходимости, управление выполнением программы передается на п. 5.
4. Двигатель JSP преобразует JSP-файл. В результате описание класса сервлета генерируется в исходном файле Java, имя которого формируется по имени JSP-файла (точное имя класса зависит от реализации).
5. Генерируемый текст Java компилируется, и для сервлета создается байтовый код.
6. Если генерируемый класс не был загружен ранее, он загружается в данный момент, и для инициализации класса вызывается его метод `jspInit()`.
7. Вызывается метод `_jspService()` генерируемого класса — в результате перед вами экран, изображенный на рис. 2.3.

Описанные шаги подытожены в блок-схеме на рис. 2.8.

Сравнение JSP и сервлета

Модель программирования JSP во многих отношениях удобнее сервлетов, потому что в ней:

- Предусмотрен сокращенный синтаксис для смешивания статических программных конструкций с динамической логикой, и программы JSP, как правило, гораздо компактнее и проще для понимания, чем сервлеты.
- Программный текст Java четко выделяется в таких компонентах, как JavaBeans и библиотеки тегов JSP (см. ниже). Это позволяет создавать страницы JSP в первую очередь тем web-дизайнерам, которые могут сосредоточить все свое внимание на представлении данных и просто подключать компоненты, разработанные программистами Java. Сервлеты же, напротив,— чистые классы Java: они пишутся обычно разработчиками Java, не являющимися специалистами в HTML. Эти факторы сильно осложняют процесс написания и сопровождение программ сервлетов, для которых необходим большой объем логических схем представления информации.
- Поддерживается модель "быстрой разработки приложений" (RAD, rapid application development), где исходный JSP-файл автоматически преобразуется двигателем JSP. Еще одним достоинством JSP является синтаксис, совместимый с XML, что позволяет авторизовать страницы JSP с помощью инструментов, основанных на XML.

Итак, когда и для чего используются сервлеты? Как всегда, у каждого подхода есть свои за и против. Выбор зависит от поставленной задачи. Там, где программа должна динамически генерировать на выходной странице двоичные данные, например какое-то изображение, или загружать файл с машины клиента, более удобен сервлет. JSP полезно использовать в первую очередь для написания символьных данных, подобных статичному тексту HTML.

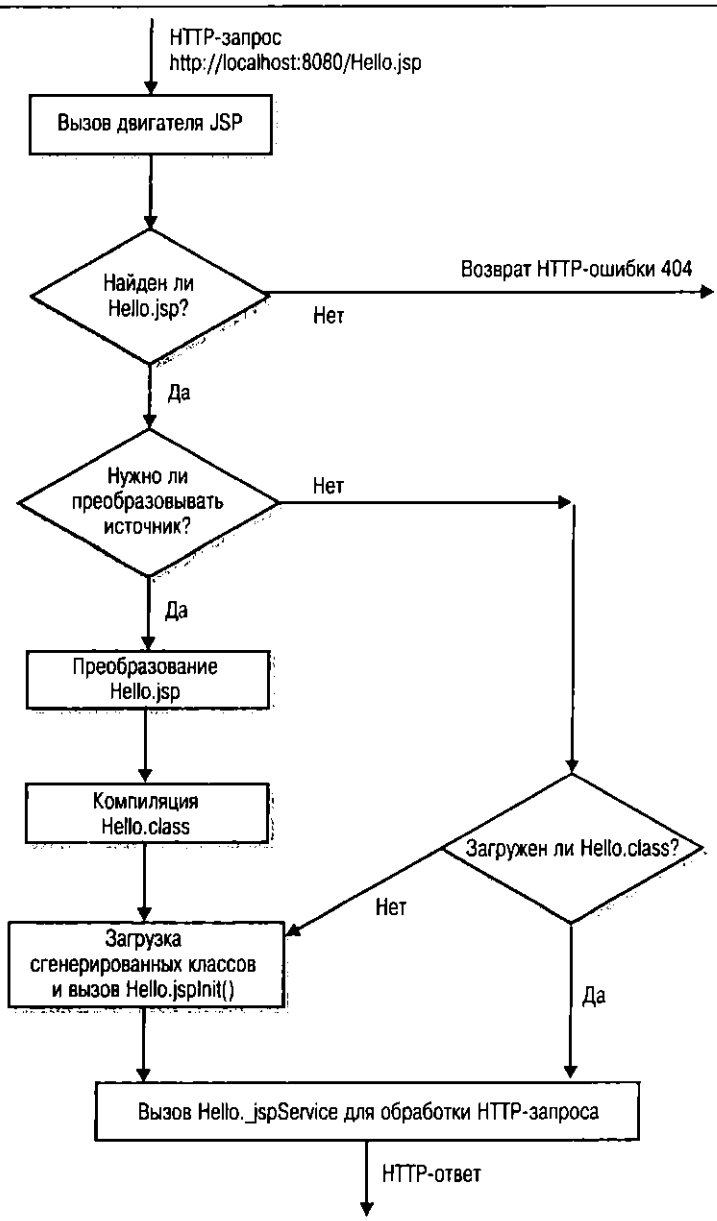


Рис. 2.8. Этапы компиляции и выполнения JSP

Основой встроенного алгоритма вывода в JSP является объект `javax.servlet.jsp.JspWriter`, обладающий разнообразными методами вывода информации скалярных типов, но не содержащий методов записи необработанных байтов. Объект учитывает кодировку символов генерируемого результата, которая не применима по отношению к некодированному потоку байтов. Напротив, сервлет предоставляет два механизма записи данных в HTTP-объект `response`:

- Метод `response.getWriter()` возвращает объект `java.io.PrintWriter` для записи символьных данных. В них обычно содержатся теги и текст HTML на местном языке, которые нужным образом интерпретируются браузером с помощью кодировки символов, указанной в заголовке HTTP-ответа.
- Метод `response.getOutputStream()` возвращает объект `javax.servlet.ServletOutputStream` для записи двоичных данных.

Важно помнить, что и модель JSP, и модель сервлетов основаны на Java и поэтому обе наследуют всю мощь и гибкость этой платформы. Различаются

они исключительно уровнем абстрагирования и удобством программирования. Так, сервлет можно вызвать из JSP и наоборот. Примеры подобного взаимодействия будут приведены в главе 10.

Архитектура web-приложений

Главным критерием качества web-приложения является простота его применения. Однако за простыми экранами и привлекательной графикой скрываются сложные и мощные вычислительные модули. Например, как средство электронной покупки экраны очень удобны, ведь потребитель имеет возможность разглядывать товары, чтобы не ошибиться в выборе. На этапе расчета в "кассе" бизнес-транзакция обычно производит оплату кредитной карточкой, сохраняет заказ в БД и предоставляет покупателю дополнительную и сопроводительную информацию. Решать все эти задачи должны защищенные, надежные и масштабируемые серверные прикладные программы. Далее мы представим архитектуры, обеспечивающие модульную разработку и простоту обслуживания подобных web-приложений. Там эффективно используются программные *компоненты*.

О них мы уже говорили в предыдущей главе. Повторим, что программный компонент – это программный модуль с грамотно описанными интерфейсом и семантикой вызова. Для web-программирования при помощи сервлетов и JSP такие компоненты могут быть полезны при генерации динамического содержимого, которое затем аккуратно объединяется со статичным текстом и тегами представления HTML. Главное преимущество компонентов заключается в том, что их могут автономно разрабатывать программисты Java, в то время как логика представления пишется специалистами в области HTML. Таким образом, основанная на компонентах модель JSP-программирования четко распределяет цели и обязанности разработчиков и web-дизайнеров, благодаря чему их роли и функции чаще всего не пересекаются.

Теперь исследуем различные схемы проектирования web-приложений со страницами JSP, начиная с очень простой архитектуры и постепенно переходя к схемам посложнее. Каждую архитектуру следует выбирать исходя из потребностей приложения. Но стоит учитывать, что большинство их с течением времени растет и лучше всего начинать с архитектуры, чьи функции можно в будущем расширить, используя различные модули, хотя сначала это покажется лишним.

В Java2 Enterprise Edition поддерживаются компоненты самых разных типов, в том числе JavaBeans и Enterprise JavaBeans. Кроме того, в Java можно реализовывать и компоненты CORBA. В спецификации JSP 1.1 представлены еще и *библиотеки тегов* (tag libraries) – упакованные утилиты, которые легко вызываются из JSP при помощи специальных тегов. Ниже показано, как различные компоненты использовать вместе со страницами JSP, создавая модульные web-приложения.

Упрощенная архитектура приложения

Сначала разберем, что произойдет, если не применять компонентное программирование. Подобная упрощенная архитектура представлена на рис. 2.9. В качестве примера можно привести JSP, открывающую и обслуживающую соединение с БД как средство электронных покупок непосредственно на странице. Другой пример – JSP, напрямую вызывающая компонент EJB или CORBA. Подобную схему достаточно легко запрограммировать, но годится она лишь для небольших приложений. Основной ее недостаток: JSP-источник очень скоро может стать громоздким и перегруженным излишней логикой программирования, например вызовами JDBC и поиском EJB через подпрограммы JNDI. В конечном счете такой подход, скорее всего, вызовет сложности в обслуживании приложения. К тому же простая страничная архитектура может пострадать от проблем с масштабированием при обработке запросов, одновременно поступающих от массы клиентов из-за неверного распределения таких критически важных ресурсов, как соединения с БД.

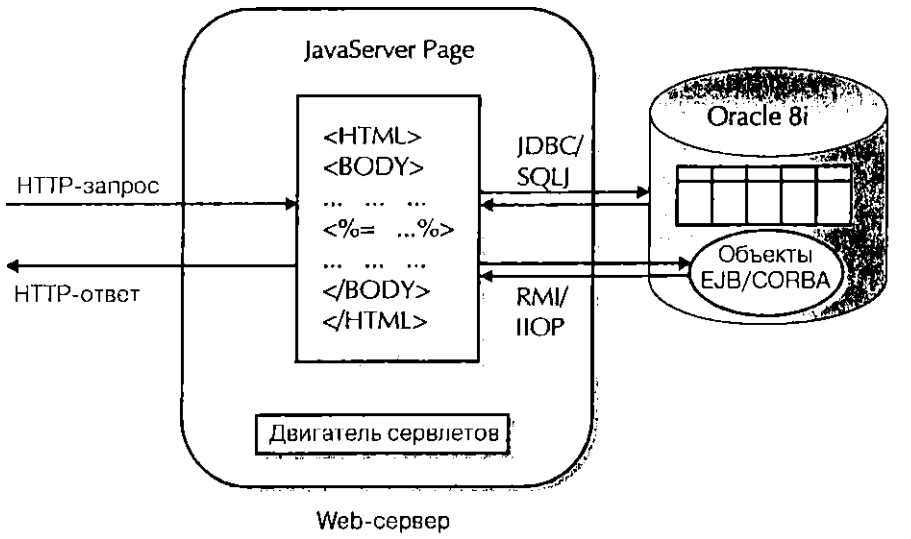


Рис. 2.9. Упрощенная архитектура web-приложения

Компонентная JSP-архитектура

Одним из способов совершенствования простой архитектуры, представленной на рис. 2.9, является применение JavaBeans ("зерен" Java). Технология JavaServer Pages была создана специально под JavaBeans. Существует несколько стандартных тегов JSP, подобных `<jsp:useBean>` и `<jsp: setProperty>`, упрощающих создание компонентов JavaBean и работу с ними.

Так что же такое JavaBean? В сущности, это стандартный класс Java, соответствующий определенным правилам проектирования. Он должен быть общим (`public`) классом с общим конструктором, не имеющим аргументов. Кроме того, типичной характеристикой JavaBean служит его набор атрибутов, или *свойств* (properties), а также общих методов доступа для чтения и записи значений свойств. В спецификации JavaBean (являющейся частью структуры Java2) описаны стандартные соглашения по именованию методов, позволяющие автоматически определять их имена на основе имен свойств. Например, по умолчанию для методов доступа свойства `X` используются имена `getX()` и `setX()`. Простой класс JavaBean с именем `SimpleBean.java` показан в листинге 2.2.

Листинг 2.2. Простой класс JavaBean

```
public class SimpleBean {
  public SimpleBean.() {};
  private int X; // Свойство зерна
  public int getX() {   // Метод получения для свойства X
    return this.X;
  }
  public void setX(int newX) {   // Метод установки для свойства X
    this.X - newX;
  }
```

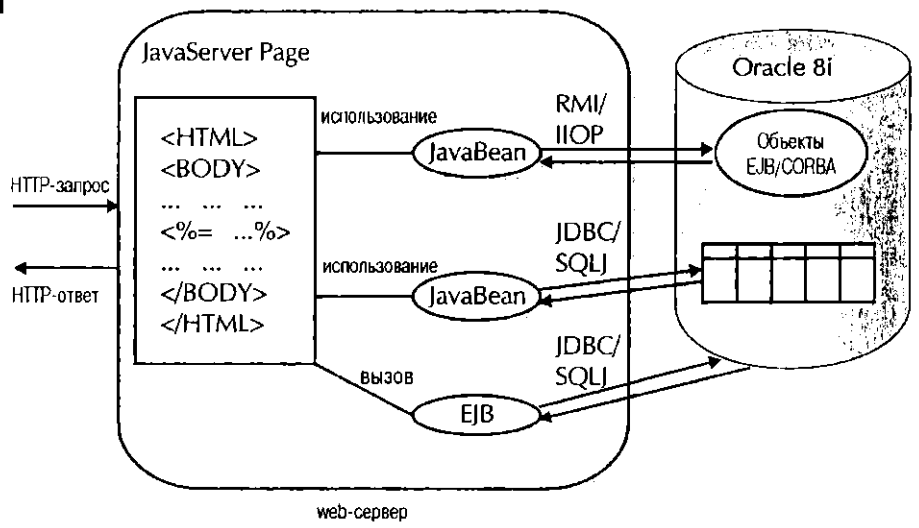
Для методов доступа зерен можно использовать и другие имена, но в этом случае они должны явно предоставляться вспомогательным классом-дескриптором (описателем) (типа `java.beans.BeanInfo`). Правила проектирования, которым следует JavaBean, позволяют динамически исследовать его свойства и манипулировать ими в той среде, в которую встроено зерно.

Здесь уместно сделать важное замечание: не путайте JavaBean с Enterprise JavaBean! Названия их схожи, но задачи совершенно разные. Изначально зерна JavaBean применялись только в графических пользовательских интерфейсах, но оказалось, что они удобны в роли компонентов JSP. Обычно JSP создает экземпляр класса JavaBean посредством тега `<jsp:useBean>`, устанавливает его свойства по параметрам HTTP-запроса посредством тега `<jsp:setProperty>`, а затем вызывает методы зерна для генерации динамического содержимого.

Зерна же Enterprise JavaBean – гораздо более тяжеловесная структура, с управляемыми по типу контейнера транзакциями и защитой. Они функционируют в первую очередь в качестве масштабируемых и распределяемых бизнес-компонентов (см. часть II этой книги).

На рис. 2.10 представлена прикладная архитектура, где вместе со страницами JSP используются зерна JavaBean, компоненты EJB и объекты CORBA. Для содержания динамической логики, например при выполнении SQL-операций в БД или вызове EJB (который сам обращается к базе при обработке транзакций), на странице JSP может присутствовать одно или несколько зерен JavaBean. Подобное разделение позволяет четко разграничивать программный текст Java и HTML и облегчать функции сопровождения. Заметим, что на странице JSP можно напрямую вызывать компоненты EJB и CORBA, не привлекая JavaBean. Однако такому подходу присущ уже упомянутый недостаток – загромождение JSP программными конструкциями Java для логики поиска JNDI и обработки исключительных ситуаций. Именно поэтому часто в качестве "оболочки" объекта EJB или CORBA удобнее применять JavaBean или сервлет (см. главу 12). Понятно, что оболочка в виде JavaBean или сервлета должна отслеживать все изменения, вносимые в методы компонента EJB или CORBA.

Рис. 2.10. Компонентная архитектура JSP



Архитектура "модель-представление-контроллер"

Показанная на рис. 2.10 архитектура, несомненно, совершеннее той, что изображена на рис. 2.9. Тем не менее порой она не лучшее решение более сложных задач. Для примера рассмотрим приложение, где пользователь сначала должен быть проверен на подлинность (аутентифицирован). Это очень распространенное требование для web-приложений. Часто необходимо еще и подтверждать правильность параметров запроса. В таких случаях рекомендуется выделять "фронтальную", внешнюю логику приложения в самостоятельный сервлет или JSP без представления информации. Программа перехватывает запросы клиентов и, как правило, инициализирует компоненты JavaBean, EJB и др. перед передачей их на другие JSP-страницы, где осуществляется представление вычисленных данных. Компоненты EJB и JavaBean – это "модель", презентационные JSP-страницы – "представление", а внешняя JSP-страница или сервлет – "контроллер" для обработки запроса. Именно поэтому подобную архитектуру часто обозначают как "модель-представление-контроллер" (MVC – "model-view-controller"), а также Model 2 – по терминологии, применявшейся в ранних версиях спецификации JSP. Ее иллюстрирует рис. 2.11.

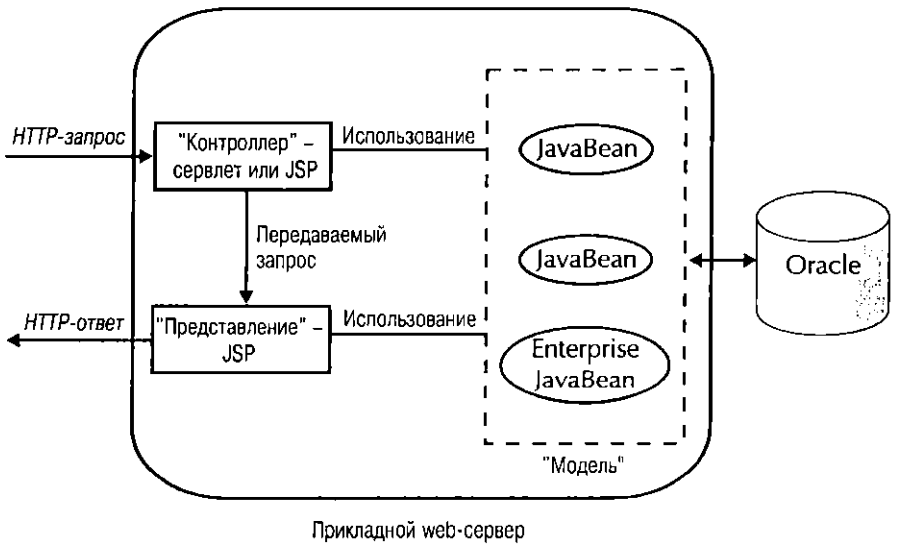


Рис. 2.11. Архитектура "модель-представление-контроллер"

К достоинствам архитектуры MVC относится очень четкое разделение динамического содержимого и представления. На JSP или в сервлете контроллера предусматривается единая точка входа в web-приложение, где удобно централизуются логика аутентификации пользователей, проверка подлинности параметров, поиск EJB и выполнение других аналогичных задач. Программа контроллера может также динамически выбирать разные JSP-страницы представления исходя из параметров запроса или возможностей клиента, тем самым обеспечивая различное представление одних и тех же данных. Презентационные JSP-страницы просто считывают те компоненты JavaBean, EJB или CORBA, которые были ранее инициализированы программой контроллера, и вызывают нужные методы для четкого извлечения и отображения динамического содержимого.

Использование библиотек тегов JSP

Как говорилось выше, технология JSP разрабатывалась для совместного использования с компонентами JavaBean. В дополнение к таким стандартным функциональным тегам, как `<jsp:include>` или `<jsp:forward>`, в JSP существует ряд встроенных тегов, облегчающих использование зерен JavaBean. Но если вдруг понадобятся теги для встраивания компонентов других типов, более общих, чем JavaBean? По определению в JavaBean должно присутствовать описание конструктора без аргументов, конкретизируемого без параметров из внешней среды. Это требование ограничивает возможности, если при инициализации компонента нужно передавать какие-либо параметры. Кроме того, по умолчанию зерна JavaBeans, используемые в JSP, не имеют доступа к контексту JSP, в котором вызываются, за исключением случаев, когда такие логические схемы не программируются явным образом. Короче говоря, иногда необходима гибкость больше той, что предоставляется набором встроенных тегов.

Вместо описания дополнительных тегов в спецификации JSP 1.1 для JSP-страниц создан гибкий механизм расширения тегов. Данная схема базируется на *библиотеках тегов* (tag libraries) – предварительно упакованных утилитах, которые можно вызывать при помощи специальных функциональных тегов на JSP-странице. Опишем вкратце действие этой схемы. Специальный функциональный тег имеет определенное имя, ни одного, один или несколько атрибутов и необязательное тело. Каждому специальному тегу соответствует класс *обработчика тега* (tag handler). Набор специальных тегов, доступных в библиотеке тегов, описывается в файле дескриптора (описателя) *библиотеки тегов* (tag library descriptor), создаваемом с помощью XML. Существует особая директива `taglib`, применяемая для указания библиотек тегов, используемых

на JSP-странице. С помощью этой директивы можно указать файл дескриптора, ассоциируемый с библиотекой, и префикс, который будет применяться для специальных тегов библиотеки.

Например, на JSP-странице, представленной в листинге 2.3, используются SQL-теги библиотеки с именем sqltaglib.tld, находящейся в каталоге WEB-INF ниже корня web-приложения. У этих тегов (локальный) префикс ora. Теги <ora:dbOpen>, <ora:dbQuery> и <ora:dbClose> применяются для открытия БД, выполнения SQL-запроса и отображения результатов в виде HTTP-таблицы, а также для закрытия базы соответственно. Подробнее о них см. главу 12.

Листинг 2.3. JSP-страница со специальными тегами для SQL-операций

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="ora" %>
    <HTML>
          <HEAD><TITLE> Example of a Tag Library </TITLE></HEAD>
          <BODY BGCOLOR="white">
                <ora:dbOpen URL="jdbc:oracle:oci8:@"
                            user="jspuser" password="jsp" connId="conn">
                            </ora:dbOpen>
                <ora:dbQuery connId="conn">
                select * from EMPLOYEE_LIST
                </ora:dbQuery>
                <ora:dbClose connId="conn" />
             <HR>
          </BODY>
    </HTML>
```

В ходе выполнения программы двигатель JSP вызывает соответствующий класс обработчика тега для каждого специального тега, встречающегося на странице. Вызов происходит при помощи классов расширения тегов и интерфейсов, описанных в спецификации JSP 1.1. Свойства класса обработчика тега (реализуемого как класс JavaBean!), в число которых входит объект PageContext для JSP-страницы, устанавливаются двигателем JSP во время обработки запроса. Таким образом, схема библиотек тегов — это, в сущности, компонентная модель с удобным синтаксисом.

Механизм расширения тегов очень эффективен, поскольку делает архитектуру тегов JSP полностью модульной и расширяемой, не снижая ее переносимости. Хотя это средство довольно ново, ряд производителей уже предлагают библиотеки специальных тегов (указатели на ресурсы см. на jsptags.com). Можно без труда создать свою собственную библиотеку со специальными тегами для конкретного компонента и выполнять ее с помощью разных двигателей JSP (см. главу 12).

Другие варианты проектирования

Помимо рассмотренных архитектур, существуют и другие варианты проектирования модульных web-приложений, определяемые конкретными требованиями среды. Например, если шаблонный HTTP-текст не статичен, а должен определяться во время работы в соответствии с параметрами запроса, с помощью функциональных тегов <jsp:include> и <jsp:forward> можно динамически выбирать соответствующий шаблонный текст (см. главу 10). Помимо динамического выбора статичных HTTP-файлов, существует возможность переносить обработку запросов на общие web-ресурсы, в частности на другие JSP-страницы и в другие сервлеты, создавая модель "вложенного" рабочего процесса. Подробное описание различных архитектурных вариантов для JSP-приложений и их сравнительная характеристика даны в книге "Professional JSP" Карла Эйвдела и др. издательства Wrox Press Inc.

Итоги

Данная глава познакомила вас с серверным web-программированием на основе сервлетов и JSP. Сначала мы определили базовое понятие web-сервера. Интернет очень быстро расширяется, чему способствует простота обращения из браузеров к файлам и программам, которые находятся на web-серверах, функционирующих на удаленных машинах. За последние два года бизнес по Интернету пережил настоящий бум, и такая тенденция, скорее всего, сохранится. Протокол HTTP и язык разметки HTML в тесной взаимосвязи друг с другом предлагают простые, но эффективные схемы транспортировки и представления данных в сетях. Они являются основой web-взаимодействия. Сервлеты и страницы JSP – относительно новые технологии, которые, будучи основанными на языке Java, в полной мере пользуются мощью, гибкостью и переносимостью объектно-ориентированной платформы Java.

В этой главе мы рассказали о таких важных свойствах API сервлетов, как контроль за сеансами для приложений, сохраняющих состояние, а также привели пример простого сервлета. Были описаны этапы выполнения программ сервлетов в контейнере и разные стадии жизненного цикла сервлета. Понимание принципов работы сервлетов является основой понимания модели JSP, поскольку та базируется на архитектуре сервлетов. JSP быстро завоевала популярность как среди web-дизайнеров, так и среди разработчиков программ Java, что объясняется удобством ее тегов, подобных сценариям, и интегрированной поддержкой компонентного программирования. С помощью специальных символов <%...%> на JSP-странице программные конструкции Java можно встраивать в текст HTML или XML, генерируя динамическое содержимое. Именно поэтому создавать JSP-страницы по сравнению с сервлетами быстрее и проще. Перед выполнением JSP-источники преобразуются в сервлеты Java, затем запускаются при помощи двигателя сервлетов. Двигатель JSP расширяет функциональные возможности двигателя сервлетов такими высокопроизводительными свойствами, как автоматическое обратное преобразование JSP-файлов. Это свойство помогает быстро разрабатывать web-приложения. Помимо прочего, нами были описаны отличия моделей сервлетов и JSP, а также условия, при которых один подход предпочтительнее другого.

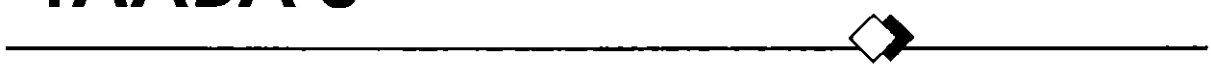
В заключение рассматривалась важная тема проектирования модульных и масштабируемых web-приложений. Архитектура JSP обеспечивает четкое разделение статического содержимого и динамических данных посредством встроенной поддержки компонентов JavaBean. Кроме того, в спецификации JSP 1.1 описана эффективная схема расширения тегов для использования библиотек тегов других фирм и построения собственных тегов. В программах JSP и сервлетов можно вызывать и объекты Enterprise JavaBeans и CORBA: эти компоненты направлены, как правило, на работу с транзакционной бизнес-логикой. Мы рассказали об использовании подобных модульных компонентов в построении совершенных web-приложений, достаточно простых в разработке, изучении и обслуживании.

О JSP-программировании речь пойдет в части IV этой книги (главах 10-12). Основные способы программирования для JSP-страниц описаны в главе 10. Глава 11 сфокусирует ваше внимание на компонентной разработке с помощью JavaBeans и JSP, причем особый акцент будет сделан на приложениях для баз данных. В главе 12 рассказывается о построении web-приложений с помощью JSP и компонентов других типов, например объектов Enterprise JavaBeans и CORBA, внедренных в Oracle8i. Кроме того, мы обсудим использование библиотек тегов JSP. Вы убедитесь в том, что JSP и сервлеты Java – это действительно мощные инструменты построения совершенных web-приложений, умело скрывающих свою сложность благодаря грамотно созданным модульным компонентам.

ЧАСТЬ II

Построение компонентов Enterprise JavaBeans

ГЛАВА 3



Знакомство с Enterprise JavaBeans (EJB)

Enterprise JavaBeans ("зерна" Java уровня предприятия) – это архитектура распределенных вычислений, основанных на применении компонентов, *Enterprise JavaBeans Specification Version 1.0*, Sun Microsystems, Inc. Она является серверной компонентной моделью, решающей такие задачи, как управление распределенными бизнес-объектами в многоуровневой архитектуре. Компоненты EJB – это компоненты Java, функционирующие на сервере (приложений или баз данных). Они могут работать и выполняться в любой среде, имеющей интерпретатор Java (виртуальную машину Java (JVM, Java Virtual Machine)) и контейнер EJB. Enterprise JavaBeans позволяет приложениям связываться в многоуровневых клиентских и серверных средах, а также в Интернет- и интранет-структурах. Чтобы грамотно ее использовать нужно освоить технологию EJB. Этой теме посвящены три главы нашей книги. В данной главе рассмотрена общая архитектура EJB и ее главные компоненты, а в главах 4 и 5 – основанные на полученных знаниях приемы построения компонентов EJB, реализующих бизнес-правила. Хотя материал носит в основном теоретический характер, хорошее его усвоение обязательно для любого разработчика приложений, который хочет научиться создавать приложения EJB.

Итак, в этой главе обсуждаются:

- Элементы Enterprise JavaBeans Specification 1.0 (17 декабря 1999 г. появилась спецификация EJB 1.1, но главные структуры EJB не изменились: они общие для обоих вариантов; поэтому главные элементы спецификации EJB 1.1 будут описаны при изучении версии EJB 1.0)
- Компоненты архитектуры EJB
- Компоненты приложения EJB
- Типы зерен Enterprise JavaBean: сеансовые и сущностные
- Дескриптор внедрения EJB
- Файл `ejb-jar`
- Управление транзакциями и безопасность в EJB
- Исключительные ситуации EJB
- Достоинства Enterprise JavaBeans
- Архитектура Oracle8*i* JServer: JServer, содержащий сервер EJB и контейнер EJB
- Создание первого Enterprise JavaBean

Понятие компонента было введено в главе 1. Напомним, что *компонентами* (components) называются заранее созданные фрагменты прикладного программного текста, которые можно собрать в работоспособные прикладные системы. С их помощью реализуются надежные и многократно используемые логические бизнес-схемы. Компонент может находиться в любом месте сети, в том же адресном пространстве (на том же компьютере), что и клиент, сервер приложений или сервер баз данных.

Спецификацией EJB (1.0 или 1.1) описывается базовая архитектура компонента EJB, с указанием структуры его интерфейсов и механизмов взаимодействия со своим контейнером и другими компонентами. Кроме того, в ней предлагаются рекомендации по созданию и реализации компонентов EJB, которые в своей совокупности образуют более крупное приложение. Построители приложений могут объединять компоненты EJB разных разработчиков или разных поставщиков, создавая единое приложение.

Об инициативе создания Enterprise JavaBeans было заявлено на конференции JavaOne-97, а начальный проект появился в декабре 1997 г. Официально спецификация EJB 1.0 увидела свет 21 марта 1998 г. О последней редакции спецификации EJB – Enterprise JavaBeans 1.1 – было объявлено 17 декабря 1999 г. В нее внесено множество изменений, ухудшающих обратную совместимость с зернами, первоначально разработанными для контейнеров EJB 1.0.

Порой переход от зерен уровня предприятия, совместимых с версией 1.0, к версии 1.1 – весьма болезненная процедура: все зависит от реализации, предлагаемой производителем. Компоненты EJB, созданные с помощью 1.0, работать с версией 1.1 не будут.

В Oracle8*i* версий 8.1.5 и 8.1.6 реализована спецификация EJB версии 1.0, т.е. действуют сервер и контейнер для зерен 1.0. В обеих версиях РСУБД поддерживаются только сеансовые зерна, как того требует спецификация. В версии 8.1.7 (чье появление планируется в год написания этой книги) будут поддерживаться и сущностные зерна. Oracle8*i* версии 8.1.7 будет совместима с версией 1.1. Отметим, что текст сеансовых зерен, выполняемых в EJB-контейнере в версиях 8.1.5 и 8.1.6, будет действителен и в версии 8.1.7. Большинство зерен уровня предприятия, разрабатываемых в этой книге, было внедрено ("как есть", без изменения программного текста) в базах данных Oracle8*i* версий 8.1.5, 8.1.6 и 8.1.7 (бэта-версии). Тем не менее стоит заметить, что зерна, внедренные в одной редакции, перед развертыванием в другой необходимо перекомпилировать. Например, если зерна EJB, внедренные в базе данных Oracle8*i* версии 8.1.5, нужно развернуть в БД редакции 8.1.6 или будущей 8.1.7, все компоненты, подлежащие внедрению, необходимо перекомпилировать. Кроме того, для версии 8.1.7 придется изменить программный текст при ограничении транзакций на стороне клиента или использовании службы транзакций Java (JTS, Java Transaction Service) напрямую. Более подробно о программах ограничения транзакций на стороне клиента (о клиентских программах EJB и CORBA) и использовании JTS говорится соответственно в главах 4 и 9.

В настоящей главе описаны главные элементы EJB 1.0. За более детальной информацией о различиях между спецификациями EJB обращайтесь к документации на *Enterprise JavaBeans 1.1* и документу "Create ForwardCompatible Beans in EJB, Part 1", java.sun.com/products/ejb/docs.html и www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj ejb1.html.

Спецификация Enterprise JavaBeans (1.0)

Спецификацию Enterprise JavaBeans (от Sun Microsystems) считают основой таких серверных служб Java, как обработка транзакций, безопасность и именование. Основа – это набор классов, вводящий абстрактный проект для решения определенных задач в общий круг взаимосвязанных проблем. *Транзакция* (transaction) – это неделимая (атомарная) единица работы, состоящая из одной или нескольких операций, когда все изменения вносятся ими как единое целое и либо полностью реализуются, либо полностью отменяются. Реализация операций как единого целого называется завершением, или *фиксацией* (commit), а отмена операций – *откатом* (rollback).

В спецификации EJB 1.0 описаны главные структуры основы EJB, интерфейсы и обычный режим работы его компонентов, в том числе два типа зерен уровня предприятия – сеансовые (содержащие модели бизнес-процессов) и сущностные (выступающие в роли долговременных сетевых хранилищ информации). Кроме того, здесь показано, каким образом между компонентами EJB распределяются роли, обязанности и службы, которые те должны предоставлять.

Производители, желающие создавать серверные реализации EJB, должны предоставлять те службы, которые заявлены в спецификации EJB (1.0 или 1.1). Сегодня под поддержку спецификации EJB адаптированы продукты множества производителей, в том числе BEA, GemStone, IBM, Persistence Software, Netscape/Sun, Oracle, Borland, Tandem, Symantec, Sybase и Visigenic.

Цели проекта Enterprise JavaBeans

Согласно спецификации, EJB направлен на:

- **Независимость от операционной системы** Он может работать на любой платформе: UNIX, Microsoft Windows, Hewlett-Packard и Mac OS.

- **Независимость от промежуточного программного обеспечения** Компоненты CORBA могут работать с любым промежуточным ПО, где реализована спецификация EJB.
- **Возможность взаимодействия зерен уровня предприятия с другими прикладными программными интерфейсами языка Java, с приложениями, созданными на языках, отличных от Java, и с CORBA** Клиенты CORBA и языков, отличных от Java, вправе пользоваться зернами уровня предприятия и наоборот. Однако в большинстве реализаций это не соблюдается. В настоящее время взаимодействие возможно лишь с помощью основных (примитивных) типов и ссылок на удаленные объекты.
- **Простоту разработки и внедрения распределенных приложений**
- **Возможность многократного использования и переносимость компонентов** Разработчики могут многократно использовать Enterprise JavaBeans и переносить зерна из системы одного производителя EJB в систему другого.

Роли Enterprise JavaBeans

В спецификации EJB определены шесть ролей для процесса прикладной разработки и внедрения компонентов EJB:

- *Поставщик зерен* (Bean provider) отвечает за соответствие спецификации написанных им необходимых интерфейсов и классов. Разработчик Java может без труда стать поставщиком зерен, т.е. разрабатывать свои собственные компоненты EJB. Напомним, что компоненты EJB – это компоненты Java. Enterprise JavaBeans – это простые, переносимые, многократно используемые и легко внедряемые компоненты, предназначенные для реализации бизнес-логики. Можно также приобрести существующие зерна EJB и ввести их в состав своего приложения или объединить свои собственные с зернами других производителей, построив нужное приложение. Интересный аспект работы поставщика зерен: ему больше не надо уделять внимание таким сложным вопросам, как многопоточная, транзакционная и распределенная обработка информации. Следовательно, он может сосредоточиться на написании приложений, имеющих дело исключительно с бизнес-логикой, и не учитывать вопросы взаимодействия и связи по сети.
- *Сборщик приложений* (application assembler) компонует одно или несколько зерен, разработанных поставщиками, создавая более крупные прикладные единицы (новые зерна уровня предприятия или приложения не-EJB). Эта роль описана в обеих версиях EJB: 1.0 и 1.1. Помимо этого, в версии 1.1 спецификации EJB учитывается возможность указания внешних ссылок, которые зерно делает во время выполнения. Внешние ссылки пригодятся, когда одному зерну нужно вызвать другое. В такой ситуации сборщик при компоновке прикладной единицы должен обеспечить возможность связи зерен между собой.
- *Специалист по внедрению* (deployer) в конкретной операционной среде отвечает за правильную установку классов и интерфейсов EJB на сервере EJB. Например, если поставщик зерен разрабатывает Enterprise JavaBeans и хочет внедрить (развернуть) их в базе данных Oracle8*i*, это будет делать администратор БД (DBA, database administrator). В нашей книге процесс внедрения зерен описан достаточно подробно, но все же в большинстве случаев эту роль выполняет DBA. Во время развертывания специалист по внедрению создает объект домашнего интерфейса EJB, доступный клиентам, использующим зерна уровня предприятия. Чтобы сделать зерно доступным, производители серверов EJB должны предоставлять клиентам, желающим с ним работать, пространство имен и домашнее эталонное имя зерна. *Пространство имен* (namespace) – это иерархическая совокупность объектов, аналогичная

системным файлам и каталогам UNIX. Oracle и другие производители EJB-серверов для хранения ссылки на домашний объект предоставляют понятное для JNDI пространство имен. Sun Microsystems поставляет JNDI в пакете `javax.naming`. *Прикладной программный интерфейс именования и каталогов Java* (JNDI, Java Naming and Directory Interface) от Sun Microsystems, Inc.– это API, предоставляющий приложениям Java функциональные средства работы с каталогами и именования. Он определяется как независимый от какой бы то ни было конкретной реализации службы каталогов. JNDI позволяет приложениям Java единообразно обращаться к самым различным каталогам. Более подробно об интерфейсе JNDI рассказано в главе 5 этой книги и в документе "Java Naming and Directory Interface Application Programming (JNDI API)", java.sun.com/products/jndi/.

- *Поставщик сервера* (server provider) – производитель операционной системы (ОС), промежуточного ПО или БД – предоставляет EJB-сервер, реализующий контейнеры для сеансовых и сущностных зерен. EJB-сервер – это прикладная система, в которой работают EJB-контейнеры. Поставщик может предоставлять контейнеры и инструменты внедрения для конкретного сервера и публиковать свои низкоуровневые интерфейсы, чтобы другие производители могли создавать контейнеры и инструменты внедрения, взаимодействующие с поставляемыми.
- *Поставщик контейнера* (container provider) создает программные инструменты установки классов и интерфейсов EJB на EJB-сервере; кроме того, он предоставляет средства контроля за контейнером и управления им, а тот, в свою очередь, контролирует зерна и управляет ими на этапе выполнения. Например, Oracle предоставляет программный инструмент `deployejb`, который позволяет специалисту по внедрению генерировать классы Java, используемые клиентами для обращения к домашним и удаленным объектам EJB (см. последующие разделы главы). Помимо этого, поставщики контейнеров устанавливают ссылки в понятном JNDI пространстве имен и обеспечивают поддержку разных версий для установленных компонентов EJB. Некоторые сведения об интеграции и доступности служб сервера, применяющего JNDI, приводятся в спецификации EJB 1.1, а более подробную информацию см. в спецификации Java 2 Enterprise Edition (J2EE).
- *Системный администратор* (system administrator) наблюдает за системой при помощи инструментов динамического контроля и управления. Такие средства предоставляют, как правило, поставщики серверов и контейнеров EJB. Заметим, что, хотя в своей среде сервер и контейнер EJB -- два самостоятельных компонента, они нередко предоставляются одним и тем же производителем. Например, корпорация Oracle объединила EJB-сервер и EJB-контейнер в границах СУБД Oracle8*i* (версий 1.5 и 1.6).

Архитектура Enterprise JavaBeans

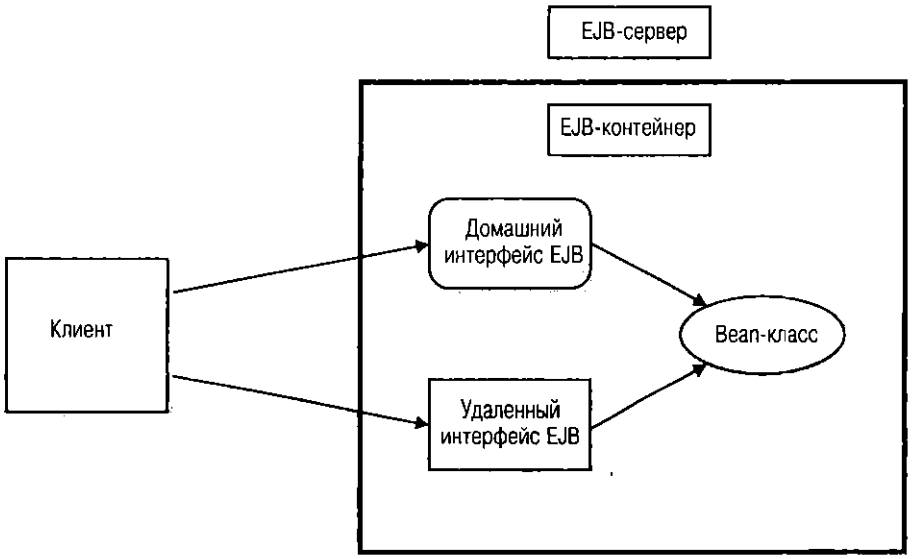
Enterprise JavaBeans – это архитектура для серверных компонентов, в частности компонентов приложений с распределенными транзакциями. Чтобы понять, как именно работает приложение EJB, нужно иметь представление об основных составляющих системы EJB, а именно:

- EJB-сервере
- EJB-контейнере
- EJB-приложении
- Клиентском приложении

Сервер Enterprise JavaBeans

EJB-сервер (рис. 3.1) – это приложение высокого уровня, управляющее EJB-*контейнерами* и организующее доступ к системным службам. Обычно он работает со всеми зернами уровня предприятия в границах файла `ejb-jar` и предоставляет такие служебные объекты, как службы транзакций и безопасности. Зерна выполняются в EJB-контейнере, который, в свою очередь, работает внутри EJB-сервера. Последним может быть любой сервер, способный содержать EJB-контейнер и обеспечивать его необходимыми службами. Примерами EJB-серверов являются серверы баз данных, серверы приложений и серверы промежуточного ПО.

Рис. 3.1. Архитектура Enterprise JavaBeans



EJB-сервер реализует системные службы вне зерен, содержит EJB-контейнер и обеспечивает контейнер такими низкоуровневыми службами, как средства сетевого взаимодействия. Контейнер, в свою очередь, предоставляет службы зерну. Спецификация EJB 1.1 требует, чтобы EJB-сервер позволял использовать объект и обращаться к нему через JNDI и службу транзакций.

Контейнер Enterprise JavaBeans

Enterprise JavaBeans – это компоненты, находящиеся и функционирующие в EJB-контейнере. В вакууме работать они не могут. Предоставить необходимые инструменты установки зерен в EJB-контейнере обязаны производители контейнеров.

EJB-контейнер – это абстракция, отделяющая бизнес-логику от базовой системы. Между зернами и сервером выполняется масса различных операций. Контейнер предоставляет зерну службы, необходимые тому для обращения к своей среде. Организуется единообразный интерфейс, управляющий взаимодействием между зернами и сервером. Контейнер в силах управлять множеством экземпляров и компонентов EJB, состоящих из нескольких классов. Он располагается на EJB-сервере, который может находится в любом месте сети, например на сервере приложений, БД или промежуточного ПО. Следовательно, контейнер тоже может размещаться в любом месте, там, где находится его сервер. Контейнер предоставляет своему зерну набор таких услуг, как управление транзакциями и ресурсами (безопасность, управление кэшем, управление посылкой сообщений), работа с разными версиями (многовариантность), масштабируемость, мобильность, устойчивость и средства подключения к базам данных. Поскольку все эти функции выполняет EJB-контейнер, разработчик компонента EJB может сосредоточить все свои усилия на реализации бизнес-правил, тем самым возлагая на него выполнение служебных задач.

Производитель контейнера предоставляет набор инструментов или классов, функционирующих в границах EJB-сервера. Так, Oracle предоставляет инструменты `deployejb` и `session_sh`: первое позволяет внедрять компоненты EJB в Oracle8*i* версий 1.5 и выше, а второе – манипулировать пространством имен, в котором находятся имена EJB-объектов (см. раздел "Архитектура Oracle8*i* JServer" этой главы и приложение D, "SQLJ Quick Reference Guide" книги "Oracle8*i*: SQLJ Programming").

Две основные функции контейнера – сделать EJB-объекты доступными для клиентов и отвечать на операции поиска клиентов. С этой целью при необходимости создаются и активируются зерна EJB. Для выполнения первой функции "имена" внутренних зерен EJB публикуются при помощи какой-либо службы именования (обычно JNDI). Термин "опубликовать" (publish) означает "сохранить ссылку на домашний и другие объекты EJB в пространстве имен". Напомним, что пространство имен аналогично каталогу UNIX. Клиенты, использующие EJB, находят домашний эталонный объект при помощи метода `lookup()` интерфейса `javax.naming.Context`. Ссылка на домашний объект позволяет клиентам создавать новые экземпляры зерен или пользоваться существующими (см. главы 4 и 5).

Пользователь EJB никогда не взаимодействует с EJB-контейнером напрямую. Среду зерна EJB составляют его контекст (`javax.ejb.SessionContext` или `javax.ejb.EntityContext`, в зависимости от типа EJB), набор свойств и все службы, необходимые зерну для реализации нужной бизнес-логики. Интерфейс `javax.ejb.SessionContext` детально обсуждается в пояснении 1 к листингу 3.6 данной главы.

EJB-контейнер не реализует и не расширяет никакого конкретного интерфейса или класса из спецификации EJB. Но у него есть определенные обязанности ("контракт контейнера" (container contract), как определено в EJB 1.0 и 1.1), которые он должен строго выполнять. Напомним, что первая его обязанность – опубликование "имен" содержащихся в нем зерен EJB посредством службы именования, а также ответ на операции поиска JNDI созданием и/или активацией зерен при их запросе. Кроме того, контейнер должен предоставлять зерну объект `javax.naming.Context`, через который можно считывать все другие объекты среды. Понятие контекста дается в интерфейсе `javax.naming.Context`, который представляет собой контекст именования, состоящий из набора связей "имя-объект" и содержащий методы для исследования и обновления этих связей.

Компоненты приложения EJB

В компонентной модели описывается набор интерфейсов и классов (например, в виде пакетов Java), используемых строго определенным образом. В компоненте изолируется и формируется набор функциональных средств. Компонент разрабатывается не для конкретного приложения, а для конкретной цели и после описания становится независимой программной единицей, которую можно распространить на другие приложения и использовать там.

Enterprise JavaBeans – это компоненты Java, создаваемые на языке Java или SQLJ. SQLJ является расширением Java, позволяющим встраивать в программы Java статические SQL-операторы. Заметим, что в статическом встраиваемом SQL все SQL-операторы, включаемые в программу, становятся известными во время компиляции, тогда как в динамическом (встраиваемом) SQL, по крайней мере, некоторые SQL-операторы совершенно неизвестны до момента выполнения программы. JDBC предоставляет разработчикам Java возможность встраивать динамический SQL в свои программы, а с помощью SQLJ подключать статические SQL-конструкции (см. книгу "Oracle8*i* SQLJ Programming").

Типы зерен EJB

Зерно уровня предприятия – это строительный блок, используемый в одиночку или вместе с другими зернами для построения более крупного

приложения. Существуют два типа зерен уровня предприятия: сеансовые и сущностные. Сеансовые зерна – это расширение клиентского приложения, а сущностные – долговременные объекты (например, строки таблицы базы данных). Замечательное свойство контейнеров EJB (контейнеров сеансовых или сущностных зерен): они управляют средами транзакций и безопасности зерна вместо его поставщика. Обратите внимание: сеансовые зерна находятся в контейнере сеансовых зерен, а сущностные – в контейнере сущностных.

Сеансовые зерна

Сеансовое зерно (session Bean) – это объект, используемый только одним клиентом. В спецификации EJB 1.0 сеансовое зерно определяется как "логическое расширение клиентской программы, функционирующей на сервере". Такие зерна характерны тем, что они:

- Предназначены для кратковременного диалога с одним клиентом и выполняются от имени этого клиента. Отметим, что клиент инициирует диалог с сеансовым зерном, когда вызывает метод класса зерна (Bean-класса).
- Относительно недолговечны. Продолжительность их существования зависит от продолжительности существования использующего их клиента. Например, клиентское приложение баз данных устанавливает сеанс связи с базой при подключении к ней клиента. В Oracle8*i* клиент может использовать сеансовые зерна, хранящиеся в базе данных. Однако при отключении клиента или сбое в его работе EJB-контейнер Oracle8*i* удаляет все сеансовые зерна, им запрашиваемые.
- Могут быть транзакционными, т.е. применяться для манипулирования данными в базе. Например, EJB-сервер Oracle8*i* позволяет создавать сеансовые зерна, считывающие, вводящие, обновляющие и удаляющие данные реляционных таблиц, а также таблиц объектов. Кроме того, классы зерен, где реализуется бизнес-логика, могут вызывать хранимые процедуры Java, SQLJ, процедуры и функции PL/SQL, а также выполнять встроенные динамические (JDBC) и статические (SQLJ) SQL-операторы.

Типы сеансовых зерен: STATELESS и STATEFUL

Во время внедрения для сеансового зерна устанавливается режим управления состоянием: STATELESS (без сохранения состояния) или STATEFUL (с сохранением состояния). Сеансовое зерно имеет тип STATELESS, когда им может пользоваться любой клиент и оно не сохраняет состояния диалога (более того, его состояние не сохраняется для методов и транзакций), а тип STATEFUL – когда состояние диалога сохраняется. Отметим, что для EJB-сервера Oracle8*i* различие между сеансовыми зернами STATELESS и STATEFUL несущественно. Дело в том, что ORB и виртуальная машина Java в Oracle8*i* действуют под управлением многопоточного сервера (MTS – multithreaded server).

Сущностные зерна

Сущностное зерно (entity Bean) – это долговременный объект, к которому можно обращаться из сеанса в сеанс и который могут разделять несколько клиентов. Такие зерна характерны тем, что они:

- Представляют данные устройств длительного хранения информации, таких как БД. Поскольку с данными в базах могут одновременно работать несколько пользователей, сущностные зерна обеспечивают возможность совместного доступа к информации.
- Транзакционные: Сущностные зерна позволяют разработчикам создавать приложения, которые автоматически обновляют информацию баз данных. Замечательное свойство Enterprise JavaBeans (сеансовых или сущностных зерен) – управление транзакциями больше не входит в обязанности разработчика. Отметим, что Enterprise JavaBeans находятся в EJB-контейнере, а контейнер функционирует на EJB-сервере. Транзакции управляются EJB-сервером и EJB-контейнером.

- Устойчивы и поэтому способны пережить аварии EJB-сервера и не уничтожаются при уничтожении клиента или отключении его от БД. Пока данные сохраняются в базе, сущностное зерно действует. Эту модель можно применять для реляционных, объектно-реляционных и объектно-ориентированных БД.

Цель этой книги – научить вас созданию сеансовых зерен для базы данных Oracle8*i*. В них (версии 1.5 и 1.6) реализуются только сеансовые зерна, как того требует спецификация EJB 1.0. Новая версия Oracle8*i* 8.1.7 будет совместимой с EJB 1.1, и поэтому в ней будут реализовываться и сеансовые, и сущностные. Заметим, что по сравнению со спецификацией EJB 1.0 описание компонентов зерна уровня предприятия в спецификации EJB 1.1 не изменилось. Следовательно, создание сеансовых зерен типа 1.0 похоже на создание сеансовых зерен типа 1.1. Однако не стоит пренебрегать дополнительными сведениями о сущностных зернах (см. библиографический справочник в конце книги).

Компонентами зерна (сеансового или сущностного) уровня предприятия являются:

- Домашний интерфейс
- Удаленный интерфейс
- Bean-класс (класс зерна) (обратите внимание: для приложения EJB можно создать один или несколько Bean-классов)
- Дескриптор внедрения

Домашний интерфейс EJB

Клиент работает с зерном уровня предприятия через свой домашний интерфейс. EJB-сервер с помощью определенного сервера имен регистрирует объект `EJBHome`, чтобы клиент мог найти ссылку на него. У сеансовых и сущностных зерен должны быть свои собственные домашние интерфейсы. Поставщик зерна описывает домашний интерфейс, а EJB-контейнер создает (во время внедрения) класс, реализующий его. В домашнем интерфейсе перечисляются методы создания новых зерен, удаления зерен и нахождения зерен. Так, в состав домашнего интерфейса входит один или несколько методов `create()`, используемых клиентами для создания экземпляров нового зерна. В нашей книге под экземпляром имеется в виду экземпляр Java, определяемый как экземпляр класса Java. Он создается на этапе выполнения программы конкретизацией данного класса. Следовательно, экземпляр зерна создается на этапе выполнения конкретизацией Bean-класса или зерна уровня предприятия. Более подробно о Bean-классах говорится в последующих разделах этой главы и в главе 4.

Домашний интерфейс зерна уровня предприятия описывает, каким образом клиентская программа или другое зерно уровня предприятия создает, находит зерно и удаляет его из контейнера. Домашний интерфейс EJB является интерфейсом Java RMI и поэтому должен соответствовать правилам, принятым для интерфейсов Java RMI (см. Remote Method Invocation Specification, java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html).

Правила Java RMI таковы:

- В конструкции `throws` каждого метода, помимо исключительных ситуаций, специфичных для приложения, должна объявляться `java.rmi.RemoteException`.
- Удаленный объект любого *сериализуемого* (serializable) типа Java может быть аргументом или возвращаемым значением. К таким типам относятся основные (примитивные) типы Java, удаленные объекты Java и неудаленные объекты Java, реализующие интерфейс `java.io.Serializable`. Напомним, что он очень важен для упорядочения (marshaling) и обратного упорядочения (unmarshaling) объектов Java. В книге "Oracle8*i* SQLJ Programming" отмечается: "Во время упорядочения параметры вызова метода (в пространстве клиента) или возвращаемые значения

(в пространстве сервера) упаковываются в стандартный формат для передачи. Во время обратного упорядочения они распаковываются, и данные представляются нужным образом в адресном пространстве принимающего процесса". Содержимое сериализуемых объектов записывается в потоки Java (т.е. последовательности байтов) вместе с информацией, достаточной для того, чтобы процесс-приемник смог восстановить объект в соответствии с версией его класса.

- Удаленный объект, передаваемый в качестве аргумента или возвращаемого значения (или непосредственно, или встроенным в локальный объект), необходимо объявлять как удаленный интерфейс.

Домашний интерфейс определяется как расширение интерфейса javax.ejb.EJBHome, который, в свою очередь, расширяет интерфейс java.rmi.Remote. Описание интерфейса javax.ejb.EJBHome дано в листинге 3.1.

Листинг 3.1. Интерфейс javax.ejb.EJBHome

```
(см. пояснение 1)
public interface javax.ejb.EJBHome
extends java.rmi.Remote{
  public abstract EJBMetaData getEJBMetaData()
    throws RemoteException; // (см. пояснение 2)
public HomeHandle getHomeHandle()
    throws RemoteException; // (см. пояснение 3)
  public abstract void remove(Handle handle)
    throws RemoteException,RemoteException; // (см. пояснение 4)
public abstract void remove(Object primaryKey)
    throws RemoteException,RemoteException; // (см. пояснение 5)
}
```

Пояснения к листингу 3.1:

1. Этот оператор описывает интерфейс javax.ejb.EJBHome, расширяющий интерфейс java.rmi.Remote.
2. Интерфейс EJBMetaData позволяет клиенту получать информацию о зерне уровня предприятия. Информация, получаемая через интерфейс EJBMetaData, используется инструментами производителя и клиентами, которым нужно вызывать зерна динамически. В конструкциях throws всех методов Java класса Java, расширяющих интерфейс java.rmi.Remote, должна использоваться RemoteException. Эта исключительная ситуация устанавливается при неудачном выполнении метода в результате сбоя системного уровня.
3. Этот метод возвращает объект HomeHandle. При неудачном выполнении этого метода в результате сбоя системного уровня устанавливается исключительная ситуация RemoteException.
4. Этот метод удаляет EJB-объект, идентифицируемый по его описателю (handle). Первая исключительная ситуация RemoteException устанавливается, когда зерно или контейнер не позволяет клиенту удалить объект, а вторая – когда метод не может быть выполнен в результате сбоя системного уровня.
5. Этот метод удаляет EJB-объект, идентифицируемый по его первичному ключу (primary key). Последовательность установления RemoteException та же, что и в пояснении 3, за исключением случая, когда объект является первичным ключом в таблице, хранимой в БД. Метод используется только для сущностных зерен.

Enterprise JavaBean считывается при помощи объекта, являющегося наследником объекта EJBHome. Существует взаимно-однозначное отношение между классом Enterprise JavaBean и объектом EJBHome в конкретном пространстве имен. EJBHome используется в качестве источника своих объектов Enterprise JavaBean. Процесс обнаружения клиентом домашнего объекта

называется *поиском* (look up). Во время поиска на сервере возникает конкретизация домашнего объекта: клиенты сначала получают ссылку на подкласс EJBHome и с помощью метода create(), указанного в домашнем интерфейсе, запрашивают ссылку на удаленный объект, а затем экземпляр Enterprise JavaBean. При конкретизации домашнего интерфейса домашний серверный объект EJB также создает удаленный интерфейс и экземпляры зерна уровня предприятия.

Ссылки на домашние объекты EJB размещаются, как правило, в службе именования, и для обращения к ним клиенты пользуются JNDI. Напомним, что обычно EJB-сервер предоставляет пространство имен для хранения таких объектных ссылок. Например, по умолчанию EJB-сервер Oracle8*i* предоставляет пространство имен test, в котором разработчик может хранить объекты EJB. Помимо этого, в Oracle существует инструмент session_sh, позволяющий создавать пространство имен по собственному усмотрению.

Все домашние интерфейсы EJB должны расширять интерфейс javax.ejb.EJBHome. В разделе "Создание первого Enterprise JavaBean" данной главы домашний интерфейс зерна тоже расширяет его. Заметим также, что домашний объект, его реализующий, должен поставляться производителем контейнера, поскольку только тот может разработать программу-источник создания зерен уровня предприятия.

Требования к домашнему интерфейсу EJB таковы:

- Интерфейс должен расширять интерфейс javax.ejb.EJBHome.
- Аргументы и возвращаемые значения методов, описанных в этом интерфейсе, должны быть правильными типами Java, и в их спецификации исключительных ситуаций throws должна присутствовать исключительная ситуация java.rmi.RemoteException.
- В домашнем интерфейсе сеансового зерна описывается один или несколько методов create(...).
- Каждый метод create() должен называться "create" и соответствовать одному из методов ejbCreate(), описанных в классе данного зерна. Число и типы аргументов соответствующего метода ejbCreate() должны быть такими же.
- Возвращаемый тип метода create должен быть типом удаленного интерфейса зерна.
- Все исключительные ситуации, описанные в конструкции throws метода ejbCreate() класса зерна, должны быть описаны в конструкции throws соответствующего метода create() удаленного интерфейса.
- В конструкции throws метода create() должна присутствовать исключительная ситуация javax.ejb.CreateException.

Домашние интерфейсы EJB, описываемые для зерен, должны расширять интерфейс javax.ejb.EJBHome. Пример домашнего интерфейса EJB приведен в листинге 3.2.

Листинг 3.2. Домашний интерфейс My8iEJBHome

```
// (см. пояснение 1)
public interface My8iEJBHome extends javax.ejb.EJBHome {
    // (см. пояснение 2)
    My8iEJB create() throws CreateException, RemoteException;
}
```

Пояснения к листингу 3.2:

1. Этот оператор объявляет интерфейс My8iEJBHome, который расширяет интерфейс javax.ejb.EJBHome. Напомним, что интерфейсы EJB всех разработчиков должны расширять интерфейс EJBHome.
2. Этот оператор объявляет метод create(), с помощью которого клиенты создают новые экземпляры зерна и обращаются к существующим. В частности, данным методом будет создан экземпляр удаленного интерфейса My8iEJB.

Методы домашнего интерфейса зерна EJB

В домашнем интерфейсе зерна будет содержаться один или несколько методов `create()`. Создавая класс зерна (Bean-класс), необходимо описать один или несколько методов `ejbCreate()`. Нужно так объявлять соответствующие методы `create()`, чтобы их сигнатуры совпадали с характеристиками методов `ejbCreate()`, объявленных в Bean-классе. Методы `create()` позволяют создавать экземпляры зерен. В дополнение к методам `create()` у сущностных зерен могут быть еще и методы обнаружения (`finder`). Пример с интерфейсом `My8iEJBHome` является законченным, иллюстрирующим процесс создания домашнего интерфейса для зерна уровня предприятия.

Помните, что EJB-контейнер реализует домашний интерфейс каждого установленного в контейнере зерна уровня предприятия и делает домашние интерфейсы доступными клиентам через JNDI. Таким образом, EJB-контейнер Oracle8*i* будет реализовывать домашние интерфейсы всех зерен, которые нужно установить в базе данных Oracle8*i*.

Удаленный интерфейс EJB

В удаленном интерфейсе EJB объявляются методы, которые могут вызываться клиентами. Подобно интерфейсу Java удаленный интерфейс EJB описывает поведение компонента EJB. Поведение – это так называемый контракт, открыто предлагаемый в интерфейсе объекта. Клиент, желающий воспользоваться объектом EJB, должен делать это через удаленный интерфейс EJB, в котором описаны точки доступа. Разработчик зерна предоставляет описание удаленного интерфейса, а во время внедрения EJB-контейнер создает класс, реализующий его.

Требования к удаленному интерфейсу EJB таковы:

- Интерфейс должен расширять интерфейс `javax.ejb.EJBObject`.
- Методы, описанные в данном интерфейсе, должны соответствовать правилам, установленным для интерфейсов Java RMI.
- В конструкции `throws` должна присутствовать исключительная ситуация `java.rmi.RemoteException`.
- Для каждого метода, описанного в удаленном интерфейсе, должен существовать соответствующий метод Bean-класса с теми же:
  - Именем
  - Числом, типами аргументов и возвращаемым типом
  - Исключительными ситуациями, описанными в конструкции `throws`

Удаленный интерфейс EJB должен расширять интерфейс `javax.ejb.EJBObject`. Методы зерна вызываются неявно, через объект `EJBObject`, выступающий в роли посредника. Каждый `EJBObject` функционирует в домашней области и обладает в границах этой области уникальными идентификационными характеристиками. Имея дело с сеансовым зерном, EJB-контейнер генерирует уникальный идентификационный номер для каждого вновь создаваемого объекта сеанса. Сеансовый `EJBObject` поддерживает бизнес-логику объекта сеансового зерна. Во время выполнения программы объект `EJBObject` передает вызов метода зерна экземпляру сеансового зерна. Разработчик сущностного зерна должен предоставлять первичный ключ для вновь создаваемого или созданного ранее объекта сущности, а контейнер с помощью первичного ключа идентифицирует объект сущности в границах домашней области. Обратите внимание: все вызовы клиентов проходят через `EJBObject`, и именно он, а не домашний объект реализует удаленный интерфейс EJB.

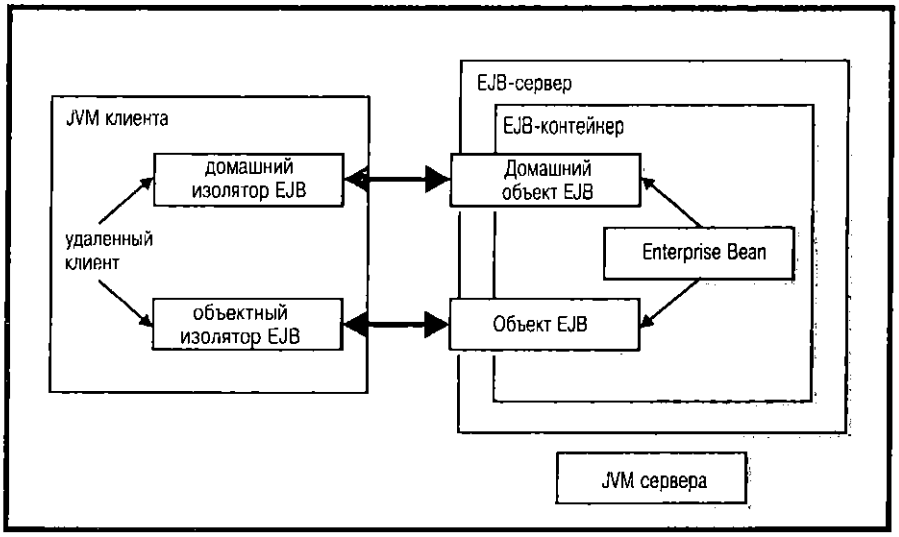
Интерфейс `javax.ejb.EJBObject`

В спецификации EJB говорится о службах, предоставляемых либо EJB-контейнером, либо объектом `EJBObject`. Отметим, что производитель контейнера предоставляет как EJB-контейнер, так и `EJBObject`, у которых разные точки входа в зерно. Кроме того, стоит отметить, что объект `EJBObject` создается на

этапе внедрения. Производитель контейнера должен поставить для этого соответствующий инструмент. На шаге 9 раздела "Создание первого Enterprise JavaBean" данной главы сеансовое зерно внедряется в базу данных Oracle8*i* с помощью специального средства `deployejb`.

При внедрении зерна уровня предприятия инструмент производителя самостоятельно формирует *stub-классы* (классы изолятора) клиента и сервера (рис. 3.2). С помощью клиентского stub-класса EJB-клиенты издалека вызывают объекты EJB. Клиентский stub-класс выступает посредником изолятора сервера, а изолятор сервера – посредником метода реального объекта. Клиент сначала ищет ссылку на домашний серверный объект, получает ссылку на удаленный объект, а затем вызывает методы объекта зерна так, как если бы тот находился в адресном пространстве клиента. Во время поиска посредник клиента упаковывает параметры вызова в сообщение запроса. Распаковывает сообщение посредник сервера (серверный stub-класс).

Рис. 3.2. Связь с клиентом посредством клиентского и серверного stub-классов



Удаленный интерфейс зерна расширяет интерфейс `EJBObject`, а класс `EJBObject` реализует удаленный интерфейс, делая класс `EJBObject` специфичным для Bean-класса. У каждого Bean-класса появится специальный класс `EJBObject`. По определению, интерфейс `EJBObject` – это интерфейс Java RMI и, следовательно, должен соответствовать правилам RMI в спецификации Java RMI. Описание интерфейса `EJBObject` представлено в листинге 3.3.

Листинг 3.3. Интерфейс `javax.ejb.EJBObject`

```
// (см. пояснение 1)
Public interface javax.ejb.EJBObject
  extends java.rmi.Remote {
    // (см. пояснение 2)
    public EJBHome getEJBHome() throws RemoteException;
    // (см. пояснение 3)
    public Object getHandle() throws RemoteException;
    // (см. пояснение 4)
    public void remove()() throws RemoteException;
}
```

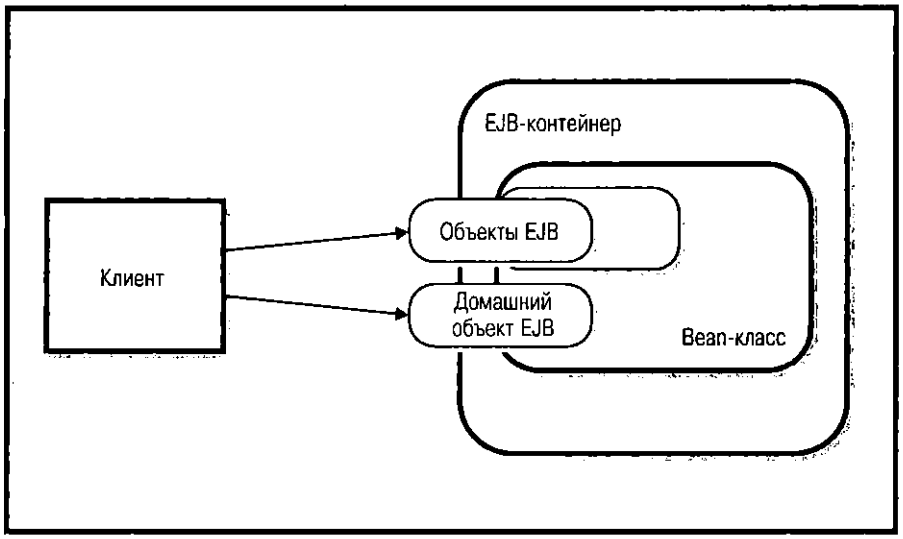
Пояснения к листингу 3.3:

1. Этот оператор описывает интерфейс `javax.ejb.EJBObject` как расширение интерфейса `java.rmi.Remote`. Это очень важный интерфейс в EJB API. Напомним, что `EJBObject` и контейнер работают в паре. К счастью, оба класса (и для контейнера, и для `EJBObject`) предоставляет производитель контейнера. Во время внедрения в Oracle инструмент `deployejb` автоматически формирует классы контейнера и `EJBObject`.

2. Этот оператор описывает метод `getEJBHome()`, который возвращает объект `EJBHome` вызывающему. В разделе "Домашний интерфейс EJB" этой главы уже отмечалось, что клиенты, желающие пользоваться зернами EJB, создают их через соответствующий домашний интерфейс. Но еще важнее то, что ссылка на домашний объект должна находиться в службе именования, которая доступна клиентам, использующим JNDI. При внедрении зерна в базу данных Oracle8*i* специальный инструмент Oracle сохраняет ссылку на домашний объект зерна в пространстве имен, которое указано в файле дескриптора внедрения, создаваемом для зерна. В этой книге используется пространство имен по умолчанию `test`, самостоятельно создаваемое Oracle при установке базы данных Oracle8*i*.
3. Этот оператор описывает метод `getHandle()` и возвращает Java-объект `Object` вызывающему. `Handle` (описатель) — это объект, идентифицирующий объект EJB. Клиенты, имеющие ссылку на объект EJB, могут узнать описатель объекта, вызвав метод *`getHandle()`* по ссылке.
4. Этот оператор описывает метод `remove()`, используемый клиентами и EJB-контейнером для удаления объекта EJB.

Объекты контейнера и `EJBObject` (рис. 3.3) совместно реализуют службы, необходимые контейнеру. Например, одной из функций контейнера является управление транзакциями, и он знает, какие атрибуты транзакций применить по отношению к методам зерна, но вызываются эти методы именно через `EJBObject`. В свою очередь, `EJBObject` должен взаимодействовать с контейнером, чтобы определить контекст транзакций для вызова бизнес-метода.

Рис. 3.3. Совместная работа EJB-контейнера и EJBObject



Клиентские программы выполняют методы удаленных зерен EJB через объект EJB. Объект EJB реализует удаленный интерфейс Bean-класса на сервере. В удаленном интерфейсе объявляются бизнес-методы Bean-класса. Через удаленный интерфейс зерна EJB-контейнер формирует клиентский и серверный stub-классы, причем второй передает вызовы клиентов объекту EJB, а первый возвращает полученную в результате информацию клиенту.

Интерфейс `EJBObject` должен расширяться удаленными интерфейсами всех зерен EJB. Следовательно, разрабатываемый удаленный интерфейс также расширяет интерфейс `EJBObject`. Пример удаленного интерфейса EJB приведен в листинге 3.4.

Листинг 3.4. Интерфейс `My8iEJB`

```
// (см. пояснение 1)
public interface My8iEJB extends javax.ejb.EJBObject {
```

```
    // (см. пояснение 2)
    public String getFirst8iEJB()
        throws RemoteException;
}
```

Пояснения к листингу 3.4:

1. Этот оператор объявляет общий (public) интерфейс My8iEJB, который расширяет интерфейс javax.ejb.EJBObject.
2. Этот оператор объявляет метод, который возвращает Java-объект типа String вызывающему. Метод должен в точности соответствовать методу Bean-класса.

Методы удаленного интерфейса зерна EJB

Сигнатуры методов, реализации которых находятся в Bean-классе, должны указываться в удаленном интерфейсе EJB. Данные методы в точности соответствуют тем, что реализованы в Bean-классе. Клиенты, использующие Enterprise JavaBeans, не могут обращаться непосредственно к объекту зерна. Другими словами, методы зерна никогда не вызываются клиентом напрямую.

Класс зерна уровня предприятия

Bean-класс – это программа, создаваемая на Java или SQLJ. Это программное тело с полями и методами, реализующими определенную производственную бизнес-логику. Bean-класс существует в границах EJB-контейнера и выполняется на сервере (приложений или баз данных). Это класс Java, который должен:

- Описываться как общий (public)
- Реализовывать интерфейс или javax.ejb.SessionBean, или javax.ejb.EntityBean (сеансовые зерна реализуют интерфейс javax.ejb.SessionBean, а сущностные – интерфейс javax.ejb.EntityBean. У каждого интерфейса свои собственные методы и семантика)
- При необходимости реализовывать удаленный интерфейс EJB
- Реализовывать один или несколько бизнес-методов и методы ejbCreate(), а при необходимости – интерфейс javax.ejb.SessionSynchronization

Приведем описание интерфейса javax.ejb.SessionBean. Описание интерфейса javax.ejb.EntitySession см. далее в приложении B и главе 18 спецификации EJB 1.0.

Описание интерфейса javax.ejb.SessionBean представлено листингом 3.5.

Листинг 3.5. Интерфейс javax.ejb.SessionBean

```
// (см. пояснение 1)
public interface javax.ejb.SessionBean
    extends javax.ejb.EnterpriseBean {
        public abstract void ejbActivate(); // (см. пояснение 2)
        public abstract void ejbPassivate(); // (см. пояснение 3)
        public abstract void ejbRemove(); // (см. пояснение 4)
        // (см. пояснение 5)
        public abstract void setSessionContext(SessionContext ctx);
}
```

Пояснения к листингу 3.5:

1. Этот оператор объявляет общий интерфейс javax.ejb.SessionBean, который расширяет интерфейс javax.ejb.EnterpriseBean. Каждый Bean-класс должен реализовывать интерфейс EnterpriseBean, расширяющий интерфейс java.io.Serializable – важный интерфейс для транспортировки (упорядочения и обратного упорядочения) объектов Java по сетям.
2. Этот оператор объявляет метод ejbActivate(), который вызывается при активации экземпляра после пребывания его в пассивном состоянии.

Временная передача состояния простаивающего сеансового зерна в какое-либо вторичное устройство хранения называется *пассивацией* (passivation), а обратная передача – *активацией* (activation). Во время активации экземпляр EJB получает все ресурсы, которые он освободил ранее в результате выполнения метода ejbPassivate(). EJB-сервер Oracle8*i* не использует ни метод ejbActivate(), ни метод ejbPassivate().

3. Этот оператор объявляет метод ejbPassivate(), вызываемый прежде, чем экземпляр переходит в пассивное состояние. Освобождаются все ресурсы, которые позднее экземпляр может вновь получить методом ejbActivate().
4. Этот оператор объявляет метод ejbRemove(). Контейнер вызывает метод перед тем, как прекратить деятельность сеансового объекта, освобождая ресурсы системы и уничтожая экземпляр зерна по запросу клиента.
5. Этот оператор объявляет метод setSessionContext(SessionContext ctx), с помощью которого контейнер использует сохраняет ссылку на объект контекста в переменной. Метод вызывается при создании зерна.

Описание интерфейса javax.ejb.SessionContext приведено в листинге 3.6.

Листинг 3.6. Интерфейс javax.ejb.SessionContext

```
public interface javax.ejb.SessionContext
    extends javax.ejb.EJBContext {
        public abstract EJBObject getEJBObject();
}
```

Пояснения к листингу 3.6:

EJB-контейнер предоставляет экземплярам сеансового зерна класс интерфейса SessionContext. Класс состоит из нескольких методов, которые позволяют клиенту манипулировать контекстом экземпляров, обслуживаемым контейнером. Интерфейс SessionContext обеспечивает доступ к сеансовому контексту этапа выполнения, который поддерживается контейнером для сеансового зерна. Сеансовый контекст предоставляет экземпляру сеансового зерна ссылки на свой удаленный интерфейс, а также сведения о службах транзакций и безопасности. Во время выполнения программы, после создания экземпляра, EJB-контейнер передает объект EJBContext данному экземпляру, и объект контекста остается связанным с экземпляром, пока тот существует. Аналогичным образом, сущностным зернам интерфейс javax.ejb.EntityContext обеспечивает доступ к сущностному контексту этапа выполнения, который поддерживается EJB-контейнером (контейнером сущностных зерен) для экземпляра сущностного зерна. Оба интерфейса (SessionContext и EntityContext) расширяют интерфейс EJBContext и предоставляют дополнительные методы, специфичные для типа зерна уровня предприятия. Описание интерфейса EJBContext приведено в листинге 3.7.

Листинг 3.7. Интерфейс javax.ejb.EJBContext

```
public interface javax.ejb.EJBContext {

  // Получение java.security.Identity вызывающего
  public abstract Identity getCallerIdentity();

  // Получение домашнего интерфейса зерна
  public abstract EJBHome getEJBHome();

  // Получение свойств среды зерна
  public abstract Properties getEnvironment();

  // Проверка, не помечена ли транзакция только для отката
  public abstract boolean getRollbackOnly();
```

```
    // Получение интерфейса ограничения транзакций
    public abstract UserTransaction getUserTransaction();

    // Проверка, есть ли у вызывающего данная роль
    public abstract boolean isCallerInRole(Identity role);

    // Пометка текущей транзакции для отката
    public abstract void setRollbackOnly();
}
```

Все сеансовые зерна должны реализовывать интерфейс SessionBean. Обратите внимание: для связи экземпляра сеансового зерна с его контекстом контейнером зерна вызывается метод setSessionContext(). Контейнер обслуживает контекст сеансового зерна на протяжении всего времени существования зерна. Пример класса сеансового зерна приведен в листинге 3.8.

Листинг 3.8. Класс My8iEJBClass

```
// (см. пояснение 1)
public class My8iEJBClass
    implements javax.ejb.SessionBean {
    // (см. пояснение 2)
    public String getFirst8iEJB()
        throws RemoteException {
        String aString =
               "First EJB application stored in the "
               + " Oracle8i database on data-i.com server "
               + " from the Oracle8i Java Component Programming "
               + " with EJB, CORBA, and JSP book.";
        return aString;
    }
    public My8iEJBClass() {
    }
    // (см. пояснение 3)
    public void ejbCreate()
        throws RemoteException, CreateException {
    }
    public void ejbActivate()
        throws RemoteException {
    }
    public void ejbPassivate()
        throws RemoteException {
    }
    public void ejbRemove()
        throws RemoteException {
    }
    // (см. пояснение 4)
    public void setSessionContext(SessionContext ctx)
        throws RemoteException {
    }
}
```

Пояснения к листингу 3.8:

1. Этот оператор объявляет общий класс My8iEJBClass, расширяющий интерфейс javax.ejb.SessionBean. Интерфейс SessionBean реализуется каждым классом сеансового зерна. Контейнер использует методы SessionBean для сообщения экземплярам зерна о событиях жизненного цикла экземпляра.
2. Этот оператор объявляет метод, который возвращает String вызывающему. Метод соответствует тому, что указан в удаленном интерфейсе. Напомним, что последний должен в точности соответствовать методу, реализуемому в его Bean-классе (см. листинг 3.4).

3. Этот оператор объявляет метод ejbCreate(). Напомним, что каждому методу create(), указанному в домашнем интерфейсе, должен соответствовать метод ejbCreate() в классе его зерна. Все исключительные ситуации, описанные в конструкции throws метода ejbCreate(), должны быть описаны в конструкции throws соответствующего метода create(). Число и типы аргументов обоих методов должны совпадать.
4. Этот оператор устанавливает контекст сеанса. Данный метод вызывается после создания экземпляра зерна. Напомним, что интерфейс SessionContext позволяет обращаться к сеансовому контексту этапа выполнения, который остается связанным с экземпляром в течение всего его существования.

Методы Bean-класса уровня предприятия

В Bean-классе должен быть описан один или несколько методов, реализующих бизнес-логику, причем сигнатуры должны подчиняться следующим правилам:

- Бизнес-метод должен объявляться общим (public).
- Типы аргументов и возвращаемых значений методов должны быть разрешенными типами Java RMI.
- В конструкции throws можно описывать произвольные исключительные ситуации, специфичные для приложения.

Кроме того, класс должен реализовывать один или несколько методов ejbCreate(), причем все они должны подчиняться определенным правилам спецификации EJB 1.0, а именно:

- Метод должен называться ejbCreate.
- Метод должен объявляться общим.
- Возвращаемым типом должен быть void.
- Аргументы методов должны быть разрешенными типами интерфейсов Java RMI.
- В конструкции throws можно описывать произвольные исключительные ситуации, специфичные для приложения.
- В конструкции throws допустима java.rmi.RemoteException.
- В конструкции throws допустима javax.ejb.CreateException.

Дескриптор внедрения Enterprise JavaBeans

В разделе "Роли Enterprise JavaBeans" данной главы раскрывалась роль специалиста по внедрению (deployer). Напомним, что он отвечает за установку на EJB-сервере интерфейсов, классов EJB и их вспомогательных классов. Этот специалист решает такие вопросы, как предоставление связующей информации этапа выполнения (например, сведений о пространстве имен EJB-сервера и имени сервера баз данных), с помощью которой клиенты обращаются к БД. Вполне возможно, что внедрять создаваемые зерна придется разработчику. Из нашей книги вы узнаете, как это можно сделать: главное — иметь ясное представление об Enterprise JavaBeans и характеристиках среды выполнения.

Дескрипторы внедрения (deployment descriptors) — это сериализуемые классы, функции которых сходны с функциями файлов свойств. Они позволяют описывать и настраивать атрибуты этапа выполнения — поведение серверных компонентов на этапе выполнения (например, безопасность, транзакционный контекст и т.д.) — без обязательного изменения Bean-класса или его интерфейсов. Класс DeploymentDescriptor — это общий базовый класс классов дескрипторов внедрения SessionDescriptor (только для сеансовых зерен) и EntityDescriptor (только для сущностных зерен). Дескриптор внедрения описывает одно или несколько зерен уровня предприятия. В Oracle8*i* версий 8.1.5 и 8.1.6 его файл имеет текстовый формат Oracle, а в версии 8.1.7 — и формат XML, и текстовый.

Сформировав для зерна интерфейсные классы и Bean-класс, нужно создать для него файл дескриптора внедрения. Дескриптор выступает в роли таблицы свойств (файла), в которой перечисляются требования этапа выполнения и безопасности, предъявляемые к зерну, а также характеристики управления транзакциями, связанные с последним. После описания дескриптора внедрения и сохранения его в файле зерно можно упаковывать в файл jar (Java archive – архив Java) для внедрения.

При внедрении формируются дополнительные классы, участвующие во внутренней работе EJB-контейнера. Для внедрения Enterprise JavaBeans производитель контейнера предоставляет нужные инструментальные средства. Напомним, что в Oracle8*i* это инструмент deployejb. Дополнительные классы позволяют EJB-компоненту управлять зернами на этапе их выполнения. Из раздела "Управление транзакциями и безопасность в EJB" этой главы вы узнаете, как EJB-контейнер управляет транзакциями с помощью их атрибутов, указанных в файле дескриптора внедрения. Процесс внедрения (развертывания) зерна состоит из установки одного или нескольких jar-файлов и назначения свойств, параметров этапа выполнения и ограничений безопасности для зерен. Описание класса javax.ejb.deployment.Deployment-Descriptor для EJB 1.0 приведено в приложении B.

На этапе выполнения EJB-контейнер управляет зернами EJB с помощью класса javax.ejb.deployment.Deployment-Descriptor. Напомним, что контейнер отвечает за управление транзакциями и обеспечение безопасности зерна. При создании для зерен файла дескриптора внедрения в этот файл заносятся значения атрибутов транзакций и безопасности, указывающие контейнеру, каким образом следует обращаться со средой выполнения зерна. EJB-контейнер считывает файл, определяя атрибуты транзакций, которые необходимо применить по отношению ко всему зерну или его конкретным методам.

Пример простого файла дескриптора внедрения приведен в листинге 3.9. Для создания этого файла можно воспользоваться любым редактором ASCII.

Листинг 3.9. Файл дескриптора внедрения EJB

```
// (см. пояснение 1)
SessionBean server.My8iEJBClass {
   // (см. пояснение 2)
   BeanHomeName = "test/My8iEJB";
   // (см. пояснение 3)
   HomeInterfaceClassName = server.My8iEJBHome;

   RemoteInterfaceClassName = server.My8iEJB;
   // (см. пояснение 4)
   AllowedIdentities = {PUBLIC};
   // (см. пояснение 5)
   SessionTimeout = 0;
   // (см. пояснение 6)
   StateManagementType = STATEFUL_SESSION;
   // (см. пояснение 7)
   RunAsMode = CLIENT_IDENTITY;
   // (см. пояснение 8)
   EnvironmentProperties {
   }
}
```

Пояснения к листингу 3.9:

1. Этот оператор объявляет зерно server.My8iEJBClass как сеансовое. Создавая файл дескриптора внедрения, не забудьте указать полный маршрут к тем интерфейсам и классам EJB, которые туда нужно включить.
2. Этот оператор сохраняет имя объекта EJB в том пространстве имен (службе именования), где объект находится. Каталог test – это пространство имен, создаваемое по умолчанию при установке Oracle8*i*. При внедрении зерна My8iEJB Oracle сделает службу именования доступной для клиентов, которые желают работать с зерном.

3. Этот и следующий операторы сохраняют ссылки на домашнюю и удаленную области зерна соответственно.
4. Этот оператор описывает атрибут безопасности, который будет связан со средой выполнения методов зерна. Спецификация EJB позволяет присваивать конкретному пользователю идентификационное значение – *роль* (role – роль БД) для всего зерна или его конкретных методов. Например, в базе данных Oracle можно создавать роли базы. Процесс создания роли выходит за рамки нашей книги – за необходимой информацией обратитесь к справочнику "Oracle8*i* SQL Reference", том 2. Напомним, что в дескрипторе внедрения присутствуют элементы управления доступом, позволяющие контейнеру обеспечивать безопасность на этапе выполнения от имени зерна. В этом примере указано, что пользователи, которым в Oracle предоставлена системная роль PUBLIC, могут работать со всем зерном, т.е. вправе обращаться ко всем методам зерна.
5. Этот оператор устанавливает значение тайм-аута сеанса в секундах. Нулевое значение говорит о том, что контейнером будет использоваться собственное значение, заданное по умолчанию.
6. Этот оператор устанавливает тип управления сеансовым зерном, указывая, что тип сеанса STATEFUL_SESSION. В базе данных Oracle8*i* зерна STATELESS и STATEFUL ничем не отличаются.
7. Этот оператор устанавливает атрибут RunAsMode, указывая контейнеру идентификационные данные безопасности, которые нужно связать с выполнением метода зерна. Так, если в дескрипторе внедрения идентифицировать безопасность конкретного метода, эти данные связываются с вызовом во время его выполнения. Атрибут RunAsMode применяется для идентификации безопасности как безопасность клиента (CLIENT_IDENTITY), привилегированной системной учетной записи (SYSTEM_IDENTITY) или учетной записи конкретного пользователя (SPECIFIED_IDENTITY). Обратите внимание: значение, указанное на уровне зерна, применяется по отношению ко всем методам, безопасность которых не идентифицирована конкретно. Поскольку для разных методов зерен STATEFUL идентификационные данные безопасности RunAs, связанные с экземпляром, определяются во время создания экземпляра, уровни методов одного зерна могут быть различны. В отношении зерен STATELESS указанные идентификационные данные RunAs должны быть одинаковы у всех методов, выполняемых в одной транзакции.
8. Здесь разработчик зерна будет описывать значения свойств среды выполнения зерна. Поставщик зерна должен описать свойства среды выполнения, т.е. пары *ключ:значение*, необходимые экземплярам зерна во время выполнения. Значения редактируются, как правило, во время разработки с помощью инструментов поставщика контейнера. В последующих главах книги показано, как перечислять эти свойства среды в файле дескриптора внедрения.

Далее, при разработке других компонентов EJB, мы расскажем, как описывать требования к транзакциям и безопасности зерна в файле дескриптора внедрения. Производитель может предоставить свой инструмент создания такого файла или воспользоваться средством другого производителя. Так, например, с помощью средства Oracle JDeveloper можно создавать Enterprise JavaBeans, используя мастер EJB Wizard. Этим инструментом для создания компонента EJB файл дескриптора внедрения будет сгенерирован автоматически.

Файл ejb-jar

Результатом действий разработчика зерна станет jar-файл, так называемый ejb-jar, состоящий из интерфейсов (домашнего и удаленного), классов, подчиненных им классов, свойств среды и дескриптора внедрения EJB. Формат

файла описан в спецификации EJB и является стандартным, используемым инструментом EJB для "упаковки" Enterprise JavaBeans. Файлы EJB упаковываются для внедрения. Упакованное зерно, передаваемое специалисту по внедрению, должно состоять из двух файлов:

- jar-файла (файла ejb-jar), сформированного из всех файлов интерфейсов и классов Java для зерен
- Файла-описателя (manifest), элементы которого описывают содержимое файла ejb-jar. Этот текстовый файл должен называться META-INF/MANIFEST.MF. Разработчикам приложений, пишущим Enterprise JavaBeans, создавать файл manifest не нужно. Команда jar формирует его автоматически и сохраняет в файле ejb-jar зерна. Пример файла-описателя приведен в листинге 3.10.

Листинг 3.10. Файл-описатель EJB

```
META-INF/MANIFEST.MF
Server/My8iEJB.class
Server/My8iEJBHome.class
Server/My8iEJBClass.class
Server/My8iEJB.ejb
```

Клиентское приложение

В спецификации EJB описывается так называемый контракт (соглашение), заключаемый между клиентом и контейнером. Он предусматривает унифицированную модель для клиентов, которые используют зерна уровня предприятия, находящиеся не только в адресном пространстве клиента, но и в других. Чтобы клиенты могли работать с зернами, поставщики зерен и сервера должны предоставить:

- Идентификационные данные объекта EJB
- Методы EJB
- Домашний интерфейс EJB

Клиентские приложения Java и не-Java работают с Enterprise JavaBeans, ссылаясь на домашний объект зерна. Oracle и другие производители, реализующие спецификацию EJB, позволяют клиентам находить ссылку на домашний интерфейс зерна с помощью JNDI. Во время поиска клиенты должны знать, где находится пространство имен, а также имя ссылки на домашний объект зерна. Такую информацию клиентам обязан предоставить разработчик зерна.

Теперь рассмотрим, как находить домашний интерфейс зерна в базе данных Oracle8*i*.

Обнаружение домашнего интерфейса зерна EJB

По отношению к EJB-серверу Oracle8*i* интерфейс JNDI выступает в роли интерфейса связи (SPI-драйвера) с OMG-классом CosNaming, предоставляющим службу именования для "опубликованных" объектов EJB и CORBA. Сейчас, чтобы разрабатывать Enterprise JavaBeans и внедрять зерна в БД Oracle, нет необходимости в изучении деталей OMG-класса CosNaming или JNDI. Тем не менее, чтобы пользоваться JNDI, нужно знать:

- Как пользоваться методами JNDI, позволяющими обращаться к объекту домашнего интерфейса зерна. Объекты домашних интерфейсов постоянно хранятся в пространствах имен, которые делаются доступными для JNDI. Так, домашние интерфейсы EJB публикуются в базе данных Oracle8*i*. При внедрении туда зерна сервер сохраняет ссылку на домашний объект в пространстве имен, описанном в файле дескриптора внедрения зерна. Указания на имя пространства имен, где находится зерно, и имя самого зерна следует в файле дескриптора внедрения. Например:

```
// Имя пространства имен - test.
```

```
// Имя зерна - My8iEJB.
BeanHomeName = "/test/My8iEJB";
```

- Как устанавливать среду для JNDI-объекта Context. Напомним, что именно он связывает имя зерна с объектом зерна. Более того, в Context содержатся методы для исследования и обновления этих связей.

Считывание объекта Context

Перед считыванием JNDI-объекта Context клиенту необходимо установить свойства среды безопасности зерна. Напомним, что с их помощью EJB-контейнер обеспечивает безопасность зерен. В javax.naming.Context передаются шесть свойств (см. приложение В и главу 5). Передает свойства объекту Context Java-объект Hashtable (хэш-таблица).

При считывании первого с помощью JNDI в среде баз данных Oracle8*i* первый получаемый объект Context связывается с контекстом именования корня публикующего контекста Oracle8*i*. Контекст именования корня Oracle можно получить через JNDI-объект InitialContext:

```
// Во-первых, создадим хэш-таблицу для хранения свойств среды
// безопасности, необходимых JNDI.
Hastable env = new Hashtable();

// Во-вторых, сохраним свойства в объекте env.
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");

..
// В-третьих, создадим объект начального контекста, вызвав
// конструктор InitialContext().
Context anInitialContext = new InitialContext(env);
```

При наличии объекта Context можно вызвать его метод lookup() и получить ссылку на домашний объект. Для этого нужно знать полный маршрут к опубликованному объекту, базовую систему, в которой находится объект, номер порта IIOP прослушивающего процесса системы и системный идентификатор БД (SID — system identifier). В листинге 3.11 переменная objectName содержит имя пространства имен, связанное с именем класса зерна, а переменная serviceURL — характеристики соединения IIOP для подключения к базе данных Oracle8*i*.

Листинг 3.11. Объявление переменных

```
// Объявим Java-переменную, содержащую имя пространства имен и имя
// опубликованного объекта.
String objectName = "/test/My8iEJB";
// Воспользуемся хост-именем области хранения БД Oracle8i
// sess_iiop://localHost:2481:orcl. Например:
String serviceURL = sess_iiop://data-i.com:2481:orcl;
```

В листинге 3.12 видим, как с помощью метода lookup() объекта InitialContext узнается ссылка на домашний объект.

Листинг 3.12. Получение ссылки на домашний интерфейс EJB

```
// Создадим объект домашнего интерфейса для хранения ссылки
// на домашний объект.
My8iEJBHome my8iEJBHome =
    (My8iEJBHome)
        anInitialContext.lookup(serviceURL + objectName);
```

Создание экземпляра удаленного интерфейса зерна

При наличии домашнего интерфейса зерна можно вызвать один из методов create() объекта домашнего интерфейса зерна и в результате получить экземпляр удаленного интерфейса зерна. Например, чтобы получить удаленный экземпляр my8iEJB:

```
// Создадим объект удаленного интерфейса.
My8iEJB my8iEJBBean = null;

// С помощью метода create() домашнего объекта создадим экземпляр
// удаленного интерфейса зерна.
my8iEJBBean = my8iEJBHome.create();
```

Вызов метода зерна

Теперь, имея в своем распоряжении экземпляр удаленного интерфейса зерна, воспользуемся им для вызова метода зерна, который описан в Bean-классе:

```
...
// Вызовем метод зерна.
OutputFromBean = my8iEJBBean.getFirst8iEJB();
```

Управление транзакциями и безопасность в EJB

Одним из замечательных свойств архитектуры EJB является то, что управление транзакциями больше не входит в число обязанностей разработчика. Эту функцию поставщики зерен передают серверу и контейнеру EJB. Поэтому производители последних (реализующие спецификацию EJB) предоставляют средства управления транзакциями. Когда зерна уровня предприятия разрабатываются для базы данных Oracle8*i*, управлять транзакциями будут EJB-сервер и EJB-контейнер Oracle8*i*.

Поддержка транзакций

В Enterprise JavaBeans поставщики зерен могут разрабатывать приложения, обращающиеся к данным и манипулирующие ими в одной БД данных и распределенных базах (нескольких, находящихся в любом месте сети). Более того, на сайте можно использовать серверы и контейнеры EJB от различных производителей, реализующих одну спецификацию EJB. Однако компоненты Enterprise JavaBeans и CORBA, находящиеся в БД Oracle8*i*, могут пользоваться только теми объектами EJB и CORBA, которые находятся на том же сервере данных Oracle8*i*. Другими словами, эти компоненты не могут обращаться к объектам EJB и CORBA, размещенным в распределенных базах данных.

Контейнер управляет транзакциями через декларативную службу управления транзакциями, которая, в свою очередь, управляется *атрибутами транзакций* (transaction attributes). Создавая дескриптор внедрения для зерна, можно указать значения атрибутов транзакций, чтобы контейнер знал, как управлять транзакциями зерна. В спецификации EJB (версий 1.0 и 1.1) описаны следующие значения атрибутов транзакций:

- TX_NOT_SUPPORTED
- TX_BEAN_MANAGED
- TX_REQUIRED
- TX_SUPPORTS
- TX_RQUIRES_NEW
- TX_MANDATORY

Семантика и характеристики атрибутов транзакций EJB, а также их значения будут описаны в главе 4, при разработке других приложений. Пока мы лишь отметим, что процессом выполнения зерна можно управлять, указывая значения атрибутов транзакций в его дескрипторе внедрения. При этом управление транзакциями автоматически передается EJB-контейнеру.

Поставщики EJB-контейнеров должны реализовывать интерфейс `javax.jts.UserTransaction`, определяемый как часть API службы транзакций Java (JTS, Java Transaction Service). JTS описывает все интерфейсы языка программирования Java, относящиеся к управлению транзакциями на платформе Java. Документы на пакет `javax.jts` содержатся в приложении B к спецификации

EJB 1.0. См. также "Understanding Enterprise Java APIs Why corporate Java developers need to know about JNDI, JTS, and JMS", www.componentmag.com/html/from_pages/feature.shtml и "Java Transaction Service (JTS)", java.sun.com/products/jts.

Все классы и интерфейсы, входящие в пакет `javax.jts` и относящиеся к Enterprise JavaBeans, перечислены в листинге 3.13.

Листинг 3.13. Классы JTS

```
interface UserTransaction
class HeuristicCommitException
class HeuristicException
class HeuristicMixedException
class HeuristicRollbackException
class TransactionRequiredException
class TransactionRolledbackException
class InvalidTransactionException
```

Обратите внимание: хотя фундаментом архитектуры Enterprise JavaBeans являются API Java и язык программирования Java, она позволяет взаимодействовать и с клиентами, и с серверами не-Java.

Поддержка распределения

В спецификации 1.0 архитектура Enterprise JavaBeans отображается на CORBA, что позволяет клиентам последней обращаться к любому объекту зерна уровня предприятия на сервере, где она разрешена, используя для этого промышленный стандартный протокол Internet Inter-ORB Protocol (IIOP). Отметим, что в Oracle8*i* версий 8.1.5 и 8.1.6 распределение поддерживается только в границах одной и той же базы данных. Следовательно, клиенты EJB и CORBA, существующие на сервере данных Oracle8*i*, могут применять объекты EJB и CORBA, располагающиеся в той же самой базе. Однако клиенты EJB и CORBA, находящиеся в любой области сети, имеют возможность обращаться к зернам уровня предприятия и к объектам CORBA, расположенным на любом сервере данных Oracle8*i*, посредством протоколов RMI over IIOP и IIOP соответственно.

Кроме того, в Enterprise JavaBeans реализация системы безопасности передается разработчиками серверу и контейнеру EJB.

Поддержка безопасности

В архитектуре EJB используется пакет `java.security`, предоставляющий защитные интерфейсы для языка программирования Java. Так, безопасность зернам обеспечивают классы `java.security.Identity`, которые выступают в качестве API для описания идентификационных данных клиента, определяя конкретного пользователя или конкретную роль. EJB-контейнер отображает экземпляры класса `java.security.Identity` на учетные записи пользователей и роли EJB-сервера.

В интерфейсе `javax.ejb.EJBContext` содержатся два специальных метода, которые позволяют экземпляру зерна идентифицировать клиента, вызывающего метод, а именно:

- `GetCallerIdentity()`
- `IsCallerInRole (Identity clientIdentity)`

Оба метода используются EJB-контейнером для идентификации клиентов, обращающихся к зернам.

Исключительные ситуации EJB

После запросов клиентов EJB-контейнер обрабатывает все исключительные ситуации, устанавливаемые методами зерна. При сбоях системного уровня или в линиях связи методами зерна устанавливается, как правило, `java.rmi.RemoteException`.

Удаленный и домашний интерфейсы EJB – это интерфейсы Java RMI. В сигнатурах методов RMI должна обязательно присутствовать исключительная ситуация `java.rmi.RemoteException`. Следовательно, в конструкциях `throws` методов удаленного и домашнего интерфейсов EJB она тоже должна фигурировать.

Напомним, что клиенты обращаются к зернам EJB через их удаленный и домашний интерфейсы. Когда клиент EJB делает запрос, исключительная ситуация `java.rmi.RemoteException` устанавливается для клиента при сбое системного уровня или в коммуникационной подсистеме между клиентом и компонентами EJB. Коммуникационный или системный сбой может произойти между клиентом и такими компонентами, как:

- **EJB-контейнер** Контейнер устанавливает исключительную ситуацию `java.rmi.RemoteException` для клиента, когда не может выполнить его запрос. Напомним, что клиент инициирует запрос вызовом метода класса зерна.
- **Класс зерна уровня предприятия** Зерно устанавливает исключительную ситуацию `java.rmi.RemoteException` для клиента при возникновении ошибки системного уровня, например ошибки БД.

Помимо обязательной исключительной ситуации `java.rmi.RemoteException`, в конструкции `throws` методов интерфейсов и классов EJB их может быть сколько угодно, в том числе специфичных для приложения. Например, в классах EJB, обращающихся к реляционным и объектно-реляционным базам данных, может присутствовать исключительная ситуация `SQLException` из пакета `java.sql`. В приложениях EJB, разрабатываемых в последующих главах книги, методы, перечисляемые в интерфейсах и классах, обращаются к базе данных Oracle8*i*. Поэтому, кроме обязательной `java.rmi.RemoteException`, в этих методах будет содержаться `SQLException`. Отметим, что при установлении зернами исключительных ситуаций, специфичных для приложения, те без изменений передаются контейнером клиентам EJB; контейнер должен преобразовать исключительную ситуацию уровня приложения в `java.rmi.RemoteException`.

В дополнение к исключительным ситуациям `java.rmi.RemoteException` и специфичным для приложения в конструкциях `throws` методов `create()`, `remove()` и `finder()` (указываемых в домашнем интерфейсе EJB) должны присутствовать `javax.ejb.CreateException`, `javax.ejb.RemoveException` и `javax.ejb.FindException`. Обратите внимание: `javax.ejb.FindException` включается в состав метода `finder()` – единственного, применимого к сущностным зернам.

Исключительные ситуации, отличные от `java.rmi.RemoteException`, могут устанавливаться, когда клиенты выполняют транзакции. Известно, что одной из ключевых характеристик Enterprise JavaBeans является поддержка распределенных транзакций. Клиенты, начинающие транзакции, могут получать исключительные ситуации, связанные с транзакциями. Например, `javax.jts.TransactionRolledbackException` устанавливается, когда транзакция была помечена для отката. Откат производится, когда транзакция не может завершиться. Напомним, что транзакция – это единица работы, выполняемая при выдаче оператора `commit` или прерываемая оператором `rollback`.

Достоинства Enterprise JavaBeans

В документе "Enterprise JavaBeans Technology Server Component Model for the Java" (java.sun.com/products/ejb/white_paper.html#ejbcompmod) перечислены достоинства EJB:

- **Переносимость, возможность настройки и неоднократного использования компонентов** Компоненты EJB можно переносить с согласующегося с EJB сервера приложений одного производителя на сервер другого и выполнять на любой платформе. Архитектурой Enterprise JavaBeans описывается стандартная модель поддержки серверами приложений Java переносимости типа WORA ("write once, run anywhere" – "записать однажды, выполнять где угодно"). Напомним,

что поведение и параметры выполнения приложения излагаются в файле дескриптора внедрения зерна. Компоненты EJB можно настраивать на выполнение в среде предприятия простым изменением набора атрибутов, указанных в файле дескриптора внедрения. Следовательно, обращаться к исходному тексту программы совсем не нужно. Однажды описанные и упакованные компоненты можно использовать неоднократно.

- **Надежность и независимость от архитектуры** В спецификациях EJB (1.0 и 1.1) описывается надежная серверная компонентная модель – и приложения, разработанные для одной платформы, можно повторно внедрить на другой.
- **Ускорение процесса разработки приложений** Решение таких сложных задач, как связь компонентов, управление транзакциями и управление потоками больше не входит в обязанности разработчика. Благодаря этому поставщик зерен может сосредоточить все усилия на разработке классов EJB, реализующих бизнес-логику, а не реализовывать в своих программах сложные инфраструктурные службы.
- **Совместимость компонентов** Модель Enterprise JavaBeans совместима с такими популярными распределенными моделями, как OMG CORBA и Microsoft DCOM.
- **Универсальность и масштабируемость** Модель EJB поддерживает web-(Интернет-/интранет-) приложения. Архитектура EJB походит и для небольших, и для крупномасштабных приложений.

Архитектура Oracle8*i* JServer

Разработчики Oracle создают серверные приложения уже давно. Приложения пишутся на PL/SQL – процедурном языке РСУБД Oracle, который является расширением SQL и тесно интегрирован с SQL-данными. Этот язык обладает такими свойствами, как обработка исключительных ситуаций, переопределение и ограниченный объем скрытой информации (в результате объявления переменных и типов не в спецификации модуля, а в его теле). Кроме того, подпрограммы (процедуры и функции) и триггеры PL/SQL можно хранить на сервере баз данных Oracle. Подпрограмма состоит из набора операторов PL/SQL, группируемых в единое целое для выполнения набора взаимосвязанных задач. Подпрограммы создаются и сохраняются в БД в скомпилированном виде. Помимо этого, программисты приложений могут создавать модули PL/SQL. Модуль (пакет) PL/SQL состоит из набора (обычно взаимосвязанных) функций и процедур PL/SQL. Вместо создания множества отдельных функций и процедур PL/SQL со взаимосвязанными функциональными средствами разработчики выстраивают модуль, содержащий все эти подпрограммы.

Oracle8*i* версии 1.5 и выше позволяет разработчикам Java создавать серверные приложения на Java или SQLJ в виде хранимых процедур Java или SQLJ, компонентов Enterprise JavaBeans или компонентов CORBA.

До появления РСУБД Oracle8*i* сервер данных состоял из:

- Двигателя (системы поддержки) SQL
- Двигателя (системы поддержки) PL/SQL

Начиная с Oracle8*i* в состав базы данных Oracle помимо двигателей SQL и PL/SQL входит Oracle8*i* JServer – серверная платформа Java. Сегодня пользователи Oracle8*i*, имевшие дело с предыдущими версиями Oracle, могут обращаться к SQL-данным с помощью двух различных прикладных программных интерфейсов (API) – JDBC и SQLJ. Более подробно об SQLJ можно узнать из книги "Oracle8*i* SQLJ Programming" (Osborne/McGraw-Hill, ISBN 0072121602).

Основы Oracle8*i* JServer

Среда JServer (рис. 3.4) состоит из:

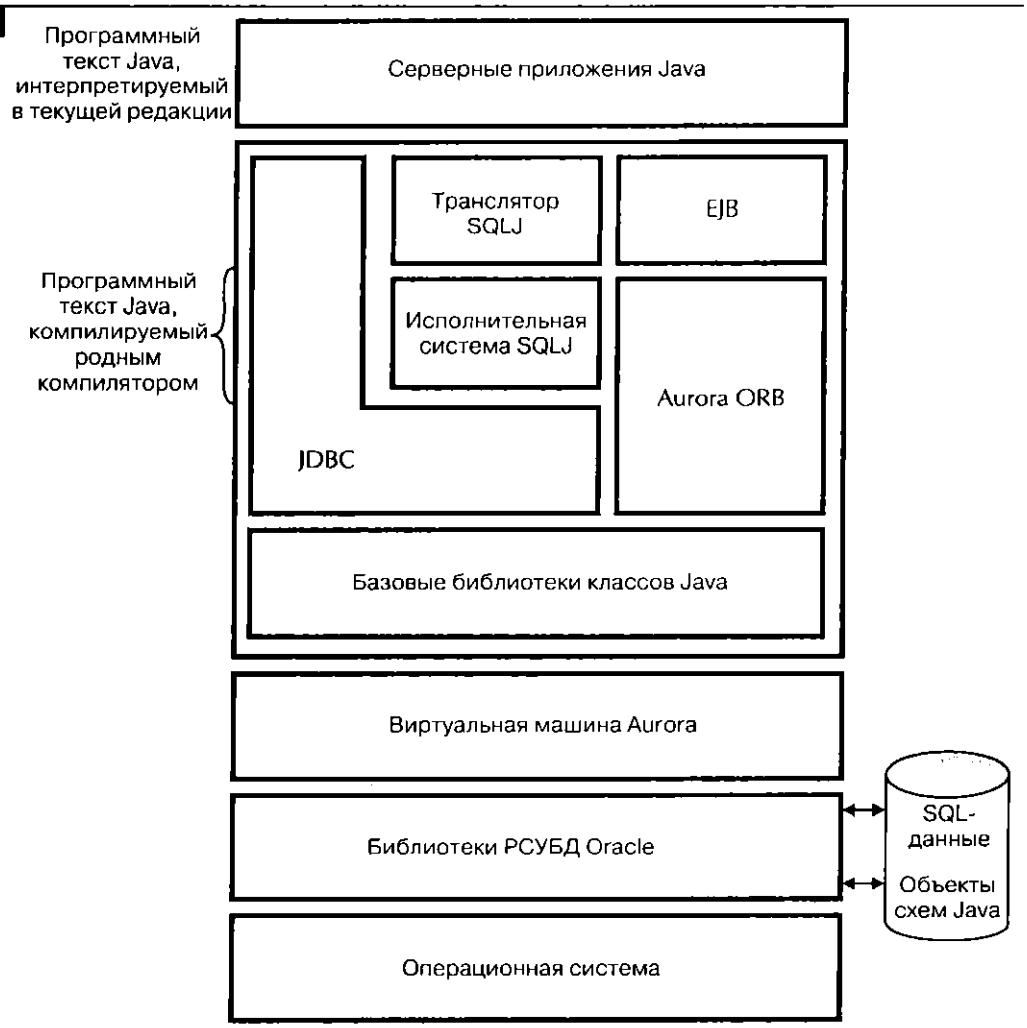


Рис. 3.4. Архитектура Oracle8*i* JServer ("Oracle8*i* Java Developer's Guide")

- Виртуальной машины Java (JVM, Java Virtual Machine) Oracle Aurora и библиотек классов Java
- Посредника объектных запросов (ORB, Object Request Broker) Aurora/ORB, а также EJB-сервера и EJB-контейнера Oracle8*i*.

Оба компонента: и JVM, и Aurora/ORB – тесно интегрированы с PL/SQL и функциональными средствами реляционных систем управления базами данных, РСУБД (RDBMS, relational database management systems) Oracle.

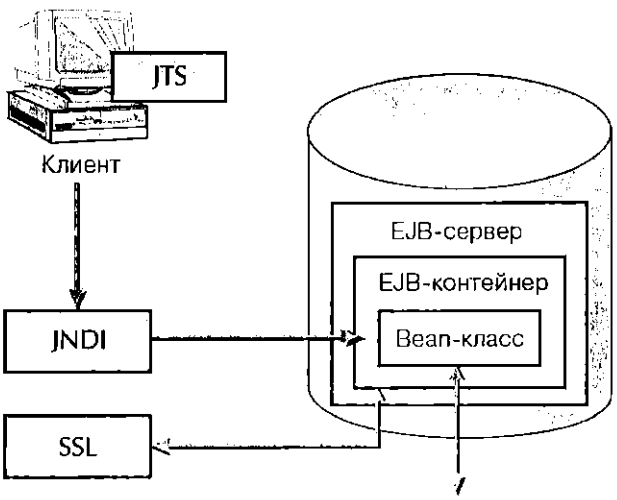
Виртуальная машина JServer – это совместимая с Java 1.1 (Oracle8*i* версии 8.1.5) и 1.2.1 (Oracle8*i* версии 8.1.6) среда выполнения Java.

EJB-сервер и EJB-контейнер Oracle8*i*

Oracle8*i* JServer соответствует спецификации EJB 1.0. Реализация EJB в Oracle (рис. 3.5) предлагает такие средства, как:

- Интерфейс JNDI для связи со служебным OMG-классом `CosNaming`, обнаруживающий и активирующий зерна. Клиенты EJB и CORBA работают с приложениями EJB и CORBA через этот интерфейс.
- Соединения уровня защиты сокетов (SSL, Secure Socket Layer), повышающие безопасность системы.
- Реализация службы транзакций Java (JTS, Java Transaction Service) для ограничения транзакций на стороне клиента. Транзакцию называют ограниченной, когда клиент вызывает методы `begin()` и `commit()`

Рис. 3.5.
Архитектура EJB в Oracle8*i*



пакета JTS. Производители EJB, включая Oracle, предоставляют свои собственные реализации интерфейсов. Так, Oracle предлагает интерфейсы `oracle.aurora.jts.client.AuroraTransactionService` и `oracle.aurora.jts.util.TS`, которые необходимо импортировать в клиентские программы, чтобы клиенты могли в определенной степени управлять транзакцией зерна (см. главы 4 и 5).

- Реализация архитектуры транзакций Java (JTA, Java Transaction Architecture) в Oracle8*i* 8.1.7 (бета-версия) или Java Transaction Service (JTS) для ограничения транзакций на стороне клиента.
- Интерфейс `UserTransaction` для транзакций, управляемых зернами.

JServer поставляется со встроенной системой CORBA 2.0 (Inprise Visibroker for Java 3.2 под Oracle8*i* 1.5 и Visibroker 3.4 под Oracle8*i* 1.6) и поддержкой Enterprise JavaBeans (только спецификация EJB 1.0). Система Visibroker 3.4 совместима и с JDK 1.1, и с Java 2. Разработчики Oracle и Java могут использовать любые языки, взаимодействуя непосредственно с сервером данных Oracle8*i* посредством промышленного стандартного протокола IIOP.

Oracle поставляет JServer со множеством инструментов и утилит, каждая из которых написана на Java, что обеспечивает простоту разработки и внедрения компонентов Java и SQLJ. В настоящее время пользователи могут работать с SQL-конструкциями, находящимися в базе данных Oracle8*i*, при помощи `SQLPLUS` (инструмента Oracle) или клиента Java. При пользовании Java выполняются программные конструкции Java, вызывая хранимые процедуры Java/SQLJ, методы сеансовых зерен или методы объектов CORBA, находящиеся на сервере данных Oracle8*i*.

Независимо от того, применяются ли для обращения к SQL-данным Oracle-инструмент `SQLPLUS` или клиентская программа, на сервере необходимо организовать сеанс связи с БД. Для этого клиенты Oracle используют Net8, JDBC или IIOP (рис. 3.6). Сеанс связи с JServer клиенты Java устанавливают через протокол IIOP. Сеансы связи с JServer аналогичны сеансам связи с базой данных, за исключением того, что область их действия ограничена объектной памятью Java. Такой сеанс JServer охватывает время действия всех объектов, на которые ссылаются статические переменные Java. С точки зрения клиента, у каждого сеанса JServer, как и у сеанса БД, своя собственная частная объектная память.

Концептуально разработка CORBA и EJB в границах JServer весьма схожа. Так, доступ к компонентам обоих типов одинаков. Объекты публикуются в базе данных Oracle8*i* при помощи служебного OMG-класса `CosNaming`, и к ним можно обращаться через Oracle-интерфейс JNDI для связи с `CosNaming` или просто через JNDI (рис. 3.7) (см. часть III (главы 6-9) книги).

Рис. 3.6.
Обращение к базе данных Oracle8*i*

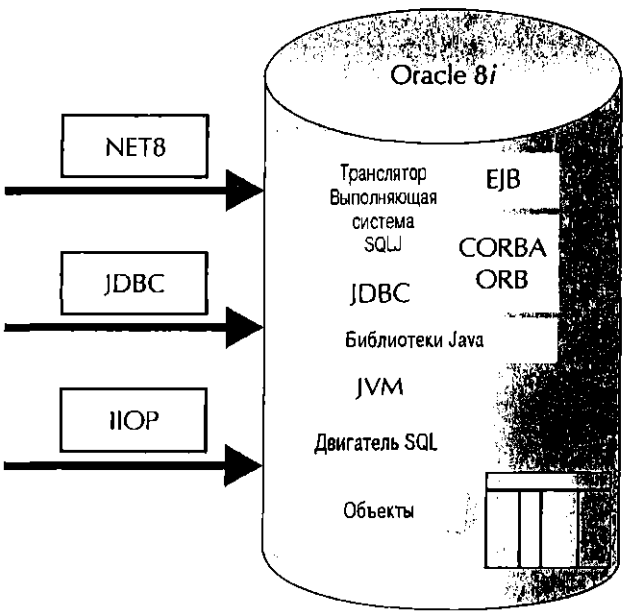
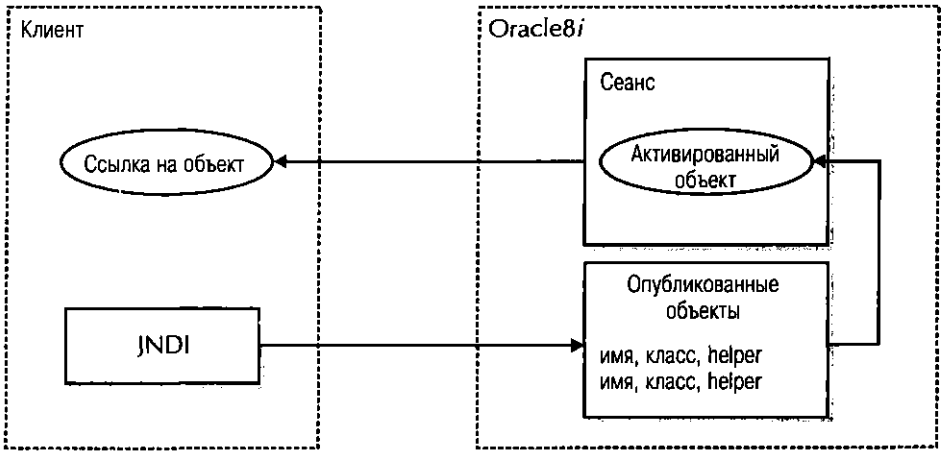


Рис. 3.7.
Публикация объектов Oracle8*i*



Oracle8*i* JServer поставляется со множеством средств разработки приложений EJB и CORBA. Это инструменты командной строки, выполняемые из оболочки UNIX или приглашения Windows. Из них имеют непосредственное отношение к EJB:

- `deployejb`
- `ejbdescriptor`
- `dropjava`
- `sess-sh`
- `remove`

С их помощью можно компилировать OMG IDL, загружать файлы классов и исходных программ Java в базу данных, публиковать объекты в пространстве имен сеанса, а также отображать имена опубликованных объектов, размещающихся в пространстве имен, и манипулировать ими (см. приложение F). Напомним: чтобы объект EJB или CORBA был доступен клиентам, его необходимо опубликовать. Для публикации объектов CORBA применяется инструмент `publish`, а объектов EJB — `deployejb`.

Oracle-инструмент deployejb

Средство deployejb готовит компонент к использованию его клиентами. Оно работает с дескриптором внедрения зерна и jar-файлом, содержащим интерфейсы, классы и подчиненные им классы EJB. Инструмент превращает создаваемый для зерна файл дескриптора внедрения в сериализуемый объект Java, а также формирует и компилирует классы, связывающие клиентов с опубликованными объектами. Напомним, что для публикации объектов в базе данных нужно иметь специальные системные привилегии Oracle. Скорее всего, эту задачу будет выполнять администратор БД Oracle. Отличный пример использования данного инструмента — шаг 9 раздела "Создание первого Enterprise JavaBean" настоящей главы. Синтаксис deployejb представлен в листинге 3.14.

Листинг 3.14. Oracle-инструмент deployejb

```
deployejb -user <username> -password <password>
-service <serviceURL> -descriptor <file>
-temp <dir> <beanjar>
  [-addclasspath <dirlist>]
  [-describe]
  [-generated <clientjar>]
  [-help]
  [-iiop]
  [-keep]
  [-republish]
  [-role <role>]
  [-ssl]
  [-useServiceName]
  [-verbose]
  [-version]
```

Вот пример использования deployejb:

```
deployejb -temp temp -u scott -p tiger
-s %ORACLE_SERVICE%
-descriptor My8iEJB.ejb -generated My8iEJBClient.jar My8iEJB.jar
```

Oracle-инструмент ejbdescriptor

Средство ejbdescriptor позволяет разработчику преобразовывать читаемый формат ASCII (.txt) файла дескриптора внедрения в нечитаемый (.ejb). Синтаксис инструмента таков:

```
ejbdescriptor {-parse | -dump} <infile> <outfile>
```

Параметр -parse используется для создания сериализуемого выходного файла <outfile> дескриптора внедрения из входного <infile>:

```
Ejbdescriptor -parse My8iEJB.txt My8iEJB.ejb
```

Параметр -dump используется для создания текстового файла <outfile> из сериализуемого <infile> дескриптора внедрения:

```
Ejbdescriptor -dump My8iEJB.ejb My8iEJB.txt
```

Oracle-инструмент dropjava

Средство dropjava превращает имена файлов командной строки и распакованное содержимое файла jar или zip в имена объектов схем, а затем удаляет объекты схем и соответствующие им строки таблиц. В командной строке можно вводить имена файлов .java, .class, .sqlj, .zip, .jar и ресурсов в любом порядке. Синтаксис таков:

```
dropjava {-u | -user} <user>/<password>[@<database>]
[options] {<file>.java | <file>.class | file.sqlj |
<file>.jar | <file.zip> | <resourcefile>} ...
```

```
[{-o | -oci8}]
[{-S | -schema} <schema>] [-stdout ]
[-s | synonym] [{-t | -thin}]
[{-v | -verbose}]
```

Вот пример использования dropjava:

```
dropjava -u scott/tiger -schema SCOTT My8iEBJ.jar
```

Oracle-инструмент sess_sh

Средство sess_sh (session shell — оболочка сеансов) — это интерактивный интерфейсный инструмент, позволяющий манипулировать пространством имен сеансов экземпляра базы данных. Аргументы соединения с базой указываются при запуске sess_sh. С помощью этого инструмента пространства имен создаются, просматриваются и удаляются. В среде БД Oracle привилегию на манипулирование пространствами имен имеет только пользователь sys. Поэтому, если нужен доступ к пространству имен сеансов Oracle, рекомендуем обращаться к администратору базы данных. Синтаксис таков:

```
sess_sh [options] -user <user>
-password <password> -service <serviceURL>
[-d | -describe]
[-h | -help]
[-iiop] [-role <rolename>]
[-ssl] [-useServiceName] [-version]
```

Вот пример использования sess_sh:

```
sess_sh -user scott -password tiger
 -service sess_iiop://data-i.com:2481:orcl
```

Oracle-инструмент remove

Данное средство удаляет PublishedObject или PublishingContext из пространства имен сеансов, но не удаляет объект схемы классов Java, связанный с PublishedObject. Синтаксис инструмента таков:

```
remove <name> -user <username>
-password <password> -service <serviceURL>
[options]
  [{-d | -describe}] [{-h | -help}]
  [-iiop] [{-r | -recurse}]
  [-role role] [-ssl]
  [-useServiceName] [-version]
```

Вот пример использования remove:

```
remove /test/My8iEJB -user scott -password tiger
-service sess_iiop://data-i.com:2481:orcl
```

В главах 6 и 7 рассказано об инструментах Oracle для работы с компонентами CORBA, а в главах 10, 11 и 12 — о других, имеющих отношение к Java Server Pages (JSP) и языку Extended Markup Language (XML).

Создание первого Enterprise JavaBean

Итак, имея представление об основах Enterprise JavaBeans, перейдем к созданию приложений EJB на языках Java и SQLJ. Воспользуемся полученными знаниями и разработаем на Java простое сеансовое зерно, внедрим его в базу данных Oracle8*i* и создадим клиент EJB, работающий с зерном. Заметим, что все программы, разработанные в этой книге, были внедрены в Oracle8*i* версий 8.1.5, 8.1.6 и 8.1.7 (бета-версии).

Процесс создания Enterprise JavaBean достаточно прост. В спецификациях EJB (1.0 и 1.1) предлагается единая стандартная модель построения зерен

EJB (сеансовых и сущностных). Следуя ей, можно использовать службы, предоставляемые EJB-сервером и EJB-контейнером. Напомним, что при построении зерен уровня предприятия больше не нужно учитывать такие сложные моменты, как многопоточная, транзакционная и распределенная обработка; эти службы для зерен предоставят EJB-сервер и EJB-контейнер.

Итак, предположим, что вы выполняете роль поставщика зерна и поэтому предоставляете:

- Удаленный интерфейс EJB.
- Домашний интерфейс EJB.
- Bean-класс или классы.
- Свойства среды для зерна (пары *ключ/значение*), необходимые зерну на этапе выполнения. Свойства среды определяются как стандартный объект `java.util.Properties`. Java-класс `Properties` — это подкласс Java-класса `Hashtable`, представляющего собой набор свойств и их значений. За подробностями о классах `Properties` и `Hashtable` обращайтесь к книге "Teach Yourself Java 2 in 21 Days" (Wrox Press, апрель 1999 г.) или к любому другому справочнику по Java. Свойства среды перечисляются в файле дескриптора внедрения.
- Дескриптор внедрения.
- Файл `ejb-jar`. Обратите внимание: файл `ejb-jar` можно создать для каждого разрабатываемого зерна EJB или всех зерен сразу. В этом файле содержатся все классы EJB и зависящие от них, дескриптор внедрения, удаленный и домашний интерфейсы EJB, свойства среды, а также файл `Manifest`, идентифицирующий дескриптор внедрения для одного или нескольких зерен.

Для начала создадим сеансовое зерно, иллюстрирующее процесс построения зерна уровня предприятия. Здесь вы должны углубить свои знания о модели Enterprise JavaBeans: мы не зря предлагаем конкретные пошаговые инструкции по построению и использованию приложений сеансовых зерен. Напомним, что EJB-сервер Oracle8*i* реализует только сеансовые зерна. Отсюда следует, что материал, изложенный в главах 3, 4 и 5, будет полезен при построении сеансовых зерен и их внедрении в базу данных Oracle8*i*, а также при разработке EJB-клиентов, которые будут их использовать.

В отношении сеансового зерна от вас требуется не так много:

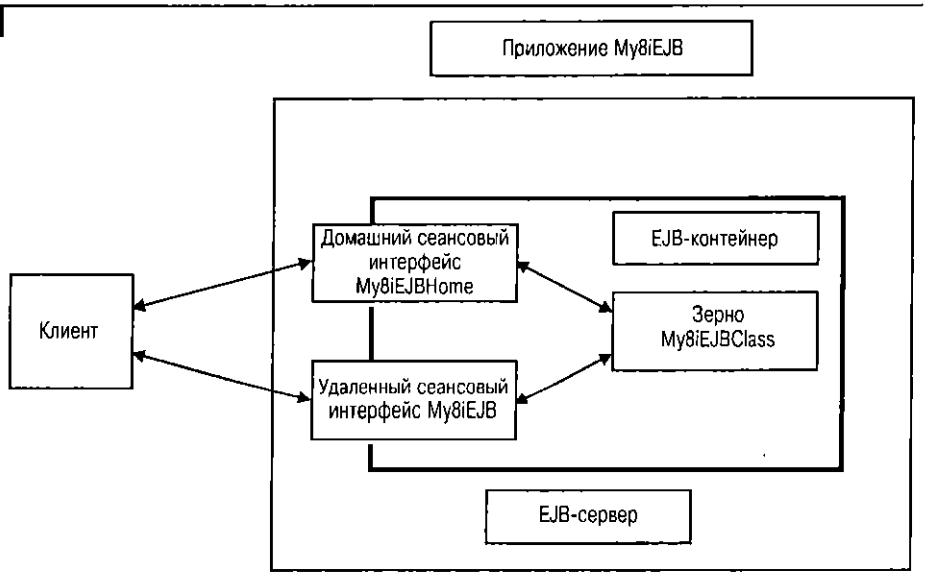
- Построить сеансовое зерно, методы которого возвращают строку символов (`String`) Java следующего содержания: "`First EJB application stored in the Oracle8i database on data-i.com server from the Oracle8i Java Component Programming with EJB, CORBA, and JSP book.`" ("Первое приложение EJB, сохраняемое в базе данных Oracle8i на сервере data-i.com, из книги "Oracle8i: Java-компонентное программирование при помощи EJB, CORBA и JSP").
- Построить клиент, который вызывает методы зерна, сохраняет значение, возвращаемое методом зерна, в его локальной переменной, а затем отображает это значение.

Общий вид EJB-приложения представлен на рис. 3.8. Клиентское приложение — это программа Java, которая вызывает метод и взаимодействует с сеансовым зерном через его домашний и удаленный интерфейсы. EJB-клиентами могут быть любые программы Java/SQLJ: апплеты, сервлеты или приложения. В приложении сеансового зерна `My8iEJB` клиент вызывает метод, который опубликован в удаленном интерфейсе для доступа к бизнес-логике, реализуемой Bean-классом. Напомним, что удаленный интерфейс зерна описывает, *что* именно оно делает, а класс зерна — *как* оно выполняет свою работу.

Процесс создания зерна имеет несколько этапов:

1. Установка системных переменных среды `CLASSPATH` и `PATH`.

Рис. 3.8. Общий вид приложения сеансового зерна My8iEJB



2. Создание одного или несколько каталогов для хранения исходного программного текста Java, который будет разработан. Одновременно подумайте о названиях пакетов для приложений EJB. На этом этапе мы создадим два каталога: `client` и `server`. В пакете Java с именем `client` будут храниться исходные программы клиентского приложения Java и файл классов Java, а в пакете с именем `server` — исходные программы Java и файлы классов Java для всех интерфейсов и классов EJB, имеющих отношение к сеансовому зерну `My8iEJB`.
3. Создание удаленного интерфейса.
4. Создание домашнего интерфейса.
5. Создание класса реализации зерна.
6. Компиляция домашнего и удаленного интерфейсов и класса зерна.
7. Описание дескриптора внедрения зерна.
8. Создание файла `ejb-jar`.
9. Внедрение Enterprise JavaBean.
10. Создание клиента.
11. Компиляция клиента.
12. Выполнение клиента.

Шаг 1. Установка системных переменных среды `CLASSPATH` и `PATH`. За сведениями об установке среды на платформах UNIX и Windows обратитесь к файлам `Makefile` и `SetupEnvironment` соответственно. Эти файлы входят в состав оперативных программных файлов, которые можно скачать или с www.data-i.com, или с www.osborne.com.

Шаг 2. Создание каталогов `client` и `server`, имена которых соответствуют именам пакетов Java.

Шаг 3. Создание удаленного интерфейса EJB. Удаленный интерфейс — обязательный компонент всех сеансовых зерен. Напомним, что в удаленном интерфейсе зерна определяется точка входа в класс зерна и объявляются методы, которые могут вызываться клиентами. Удаленный интерфейс `My8iEJB` должен расширять интерфейс `javax.ejb.EJBObject`, а в конструкции `throws` его бизнес-метода `getFirst8iEJB()` должна присутствовать исключительная ситуация `java.rmi.RemoteException`.

Опишем удаленный интерфейс My8iEJB:

```
// Имя программы: My8iEJB.java
// Версия:
// Автор:          Ваше имя
// Описание:       Удаленный интерфейс EJB, обьявляющий метод
// Bean-класса     My8iEJBClass.
//

package server; // (см. пояснение 1)

// Обязательные пакеты Java и ejb для удаленных интерфейсов EJB.
import java.rmi.*; // (см. пояснение 2)
import javax.ejb.*;

// Опишем общий интерфейс, который расширяет интерфейс
// javax.ejb.EJBObject.
puolic interface My8iEJB extends EJBObject {

 // Опишем общий метод, который в точности соответствует методу,
 // описанному в Bean-классе.
 public String getFirst8iEJB()
    throws RemoteException;
}
```

Пояснения к интерфейсу My8iEJB:

1. Этот оператор описывает интерфейс как часть пакета server.
2. Этот и последующий операторы импортируют пакеты java.rmi.* и javax.ejb.* соответственно. Напомним, что удаленный интерфейс EJB — это интерфейс Java RMI, и поэтому он должен иметь доступ к исключительной ситуации RemoteException.

Шаг 4. Создание домашнего интерфейса EJB. Клиенты находят зерно по ссылке на его домашний интерфейс. Домашний интерфейс EJB — обязательный компонент для построения EJB-приложений. Следовательно, интерфейс My8iEJBHome должен расширять интерфейс javax.ejb.EJBHome. Необходимо описать, по крайней мере, один метод create(), и в конструкции throws этого метода должны присутствовать исключительные ситуации java.rmi.RemoteException и CreateException. Можно описать сколь угодно большое число методов create(), однако каждому из них должен в точности соответствовать метод ejbCreate(), описанный в классе зерна. Но важнее то, что метод create() отвечает за создание новых экземпляров удаленных интерфейсов EJB или многократное использование клиентами существующих удаленных экземпляров. Поэтому возвращаемый тип метода create() должен быть типом удаленного интерфейса EJB. Воспользуемся удаленным интерфейсом My8iEJB, описанным на шаге 3, в качестве возвращаемого типа метода create(), указанного в домашнем интерфейсе. Опишем домашний интерфейс My8iEJBHome:

```
// Имя программы: My8iEJBHome.java
// Версия:
// Автор:          Ваше имя
// Описание:       Домашний интерфейс EJB, используемый для создания
//                 экземпляров удаленного интерфейса EJB
//                 My8iEJB.
//

package server;

import java.rmi.*;
import javax.ejb.*;

// Этот интерфейс должен быть общим и расширять
// интерфейс javax.ejb.EJBHome.
public interface My8iEJBHome extends EJBHome {
```

```
// В описании домашнего интерфейса разработчика должны
// присутствовать один или несколько методов create().
// Каждому указанному здесь методу create() должен
// соответствовать метод ejbCreate(), указанный в Bean-классе.

   My8iEJB create() throws CreateException, RemoteException;
}
```

Шаг 5. Создание Bean-класса. Обычно Bean-класс реализует определенную бизнес-логику. Простое зерно содержит один бизнес-метод, который при вызове возвращает клиенту Java-объект String. Все сеансовые зерна расширяют интерфейс javax.ejb.SessionBean. В конструкции throws всех описываемых методов должна присутствовать исключительная ситуация RemoteException, а также, если нужно, любые исключительные ситуации, специфичные для приложения. Но не забывайте, что приведенные здесь исключительные ситуации уровня приложения следует указывать и в конструкции throws удаленного интерфейса. Для каждого метода create() из домашнего интерфейса в Bean-классе должен существовать метод ejbCreate(), причем сигнатуры методов должны совпадать. Кроме того, в Bean-классе надлежит описать методы ejbActivate(), ejbPassivate() и ejbRemove(). Несмотря на то что в базе данных Oracle8*i* (1.5 или 1.6) они и не используются, указать их в Bean-классе все же надо. Опишем класс My8iEJBClass:

```
// Имя программы: My8iEJBClass.java
// Версия:
// Автор:         Ваше имя
// Описание:      Bean-класс EJB, который состоит из метода,
//                возвращающего при вызове клиентом Java-объект
//                типа String.
//

package server;

import java.rmi.RemoteException;
import javax.ejb.*;

// Bean-класс должен расширять интерфейс javax.ejb.SessionBean.
public class My8iEJBClass implements SessionBean {
    // (см. пояснение 1)
    public String getFirst8iEJB()
        throws RemoteException {
        String aString =
               "First EJB application stored in the "
               + " Oracle8i database on data-i.com server "
               + " from the Oracle8i Java Component Programming "
               + " with EJB, CORBA, and JSP book.";
        return aString;

    // Обратите внимание на метод ejbCreate(), который соответствует
    // методу create() из описания домашнего интерфейса.
    public void ejbCreate()
             throws RemoteException, CreateException {
    }
    // (см. пояснение 2)
    public void ejbActivate()
        throws RemoteException {
    }
    public void ejbPassivate()
        throws RemoteException {
    }
    public void ejbRemove()
        throws RemoteException {
    }
    // (см. пояснение 3)
```

```
    public void setSessionContext(SessionContext ctx)
        throws RemoteException {
    }
}
```

Пояснения к классу My8iEJBClass:

1. Этот оператор объявляет простой бизнес-метод реализации, в точности соответствующий бизнес-методу, опубликованному в удаленном интерфейсе.
2. Этот и два последующих оператора описывают методы ejbActivate(), ejbPassivate() и ejbRemove(). Их необходимо объявлять в классе зерна.
3. Этот оператор описывает метод setSessionContext(), используемый контейнером для связи объекта java.naming.Context с объектом зерна. Объект SessionContext остается вместе с зерном в течение всего его жизненного цикла, пока клиент не будет отключен или уничтожен.

Шаг 6. Компиляция домашнего и удаленного интерфейсов и класса зерна. Установите одну или несколько переменных для CLASSPATH и скомпилируйте удаленный и домашний интерфейсы и Bean-класс:

```
// Предположим, что создана переменная с именем
// YOUR_CLASSPATH_VARIABLE, в которой сохранено значение
// CLASSPATH.
Javac -classpath %YOUR_CLASSPATH_VARIABLE% -g
My8iEJB.java My8iEJBHome.java My8iEJBClass.java
```

Шаг 7. Описание дескриптора внедрения зерна. Если на шаге 6 никаких ошибок не возникло, можно продолжить работу (в противном случае исправьте ошибки). Напомним, что в этом файле указываются значения атрибутов транзакций и безопасности, инструктирующие контейнер, чтобы он управлял средой транзакций и безопасности зерна во время его выполнения.

```
// Имя программы: My8iEJB.ejb
// Версия:
// Автор:          Ваше имя
// Описание:       Дескриптор внедрения для сеансового зерна My8iEJB.
//
// Опишем server.My8iEJBClass как сеансовое зерно.
SessionBean server.My8iEJBClass {

    // Укажем маршрут к пространству имен и имя, которое будет
    // использоваться для хранения ссылки на домашний интерфейс
    // сеансового зерна.
    BeanHomeName = "test/My8iEJB";

    // Сохраним имя домашнего интерфейса.
    HomeInterfaceClassName = server.My8iEJBHome;

    // Сохраним имя удаленного интерфейса.
    RemoteInterfaceClassName = server.My8iEJB;

    // Воспользуемся Oracle-ролью PUBLIC, инструктирующей EJB-контейнер,
    // чтобы он разрешил применять сеансовое зерно My8iEJB
    // всем пользователям, которым предоставлена эта системная роль.
    AllowedIdentities = {PUBLIC};

    // Для этой переменной можно установить значение в секундах.
    // Однако значение 0 говорит о том, что будет использоваться
    // значение, заданное для контейнера по умолчанию.
    SessionTimeout = 0;

    // Необходимо установить переменную управления состоянием как
    // STATEFUL_SESSION или STATELESS_SESSION; для Oracle8i это
```

```
    // безразлично. Все сеансовые зерна, сохраняемые в БД,
    // считаются STATELESS.
    StateManagementType = STATEFUL_SESSION;

    // Проинструктируем контейнер, чтобы он при проверке
    // идентификационных характеристик безопасности клиента
    // использовал атрибут CLIENT_IDENTITY.
    RunAsMode = CLIENT_IDENTITY;
}
```

Шаг 8. Создание файла `ejb-jar`. Процесс внедрения зерен упрощается благодаря использованию `.jar`-файла, состоящего из взаимосвязанных интерфейсов и подчиненных им классов. Применим команду `jar` с нулевым (0) параметром, чтобы он сгенерировал несжатый `.jar`-файл, как того требует Oracle8*i* версии 1.5. В новой редакции Oracle8*i* 1.6 поддерживаются как сжатые, так и несжатые файлы JAR. Заметим, что у других производителей требования могут быть иными. В приглашении UNIX или DOS введем:

```
// Если мы в родительском каталоге, где находится каталог server,
// то воспользуемся командой:
jar cvf0 server/My8iEJB.jar server/*.class

// Если мы непосредственно в каталоге server, применим команду.
jar cvf0 My8iEJB.jar *.class
```

Шаг 9. Внедрение Enterprise JavaBean. Зерна уровня предприятия можно применять только после их внедрения (развертывания). Заметим, что клиентские и серверные stub-классы, а также те, что используются сервером и контейнером EJB, формируются только во время внедрения. Oracle-инструмент `deployejb` выполняет две задачи: загружает интерфейсы и классы, содержащиеся в файле `ejb-jar`, в БД Oracle8*i* и генерирует на сервере базы классы `EJBHome` и `EJBObject`, а также все те классы, которые необходимы серверу и контейнеру EJB и клиенту.

При использовании инструмента `deployejb` он генерирует `.jar`-файл, состоящий из клиентских stub-классов, необходимых клиенту для работы с зерном. Имя для этого файла можно придумать или воспользоваться тем, которое инструмент создает по умолчанию. При внедрении сеанса `My8iEJB` назовем `.jar`-файл `My8iEJBClient.jar`, указав инструменту `deployejb`, где сохранять клиентские stub-классы.

Установим `CLASSPATH` и внедрим сеансовое зерно `My8iEJB`. Можно описать переменную `ORACLE_SERVICE` для хранения URL службы Oracle8*i*, но, в принципе, ее можно и не описывать. Приведем пример объявления переменной `ORACLE_SERVICE` (из приглашения DOS), после чего следует команда `deployejb`:

```
// Сначала установим CLASSPATH.
// Опишем переменную: среда Windows.
SET ORACLE_SERVICE=sess_iiop://data-i.com:2481:orcl

// Среда UNIX.
setenv ORACLE_SERVICE sess_iiop://data-i.com:2481:orcl

deployejb -republish
-temp temp -u scott -p tiger
-s %ORACLE_SERVICE% -verbose
-descriptor My8iEJB.ejb -generated My8iEJBClient.jar My8iEJB.jar
```

В листинге 3.15 приведен пример выходных данных, генерируемых инструментом `deployejb`.

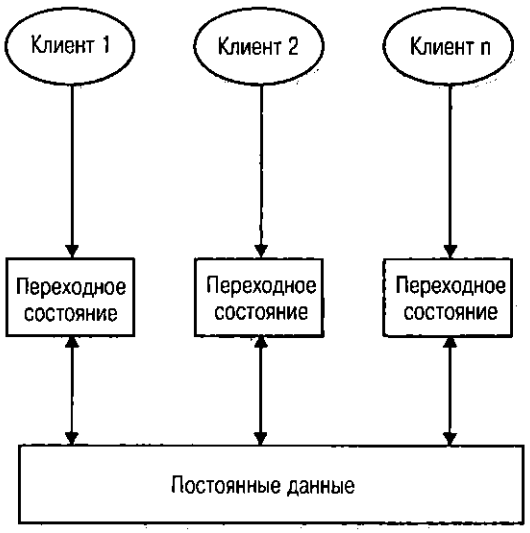
Листинг 3.15. Пример выходных данных инструмента `deployejb`

```
...
server/My8iEJBHelper already resolved
```

```
resolving: server/My8iEJBHolder
server/_st_My8iEJBHome already resolved
oraclex/java/lang/ObjectHelper already resolved
oraclex/java/lang/ObjectHolder already resolved
server/My8iEJBHomeOperations already resolved
done
Generating EJBHome and EJBObject on the server...done
Publishing EJBHome...done
```

Шаг 10. Создание клиента. EJB-клиенты — т.е. апплеты Java/SQLJ, сервлеты, приложения, а также клиенты Java Server Page (JSP), CORBA и клиенты не-Java — могут пользоваться зернами EJB. Первое, что должен сделать клиент `MyFirstEJBClient` (рис. 3.9) — это найти ссылку на домашний интерфейс EJB, описанный на шаге 4 этого раздела. Ссылка на домашний объект зерна находится посредством JNDI. Создадим объект `env Hashtable`, сохраним свойства среды зерна в объекте `env` и передадим их объекту `InitialContext ic`.

Рис. 3.9. Клиенты обращаются к сеансовым зернам



Теперь, обладая контекстным объектом `ic`, с помощью метода `lookup()` найдем ссылку на домашний интерфейс зерна и уже с ее помощью создадим экземпляр удаленного интерфейса зерна (`my8iEJBBean`), через который дистанционно вызовем метод `getFirst8iEJB()` зерна.

В этом разделе клиент `MyFirstEJBClient` — это приложение Java, исходный текст которого находится в пакете с именем `client`. В части IV этой книги показано, как создавать клиентов сервлетов и JSP, использующих EJB. Клиент `MyFirstEJBClient` описан в следующем листинге:

```
/*  Имя программы:   MyFirstEJBClient.java
**  Назначение:      Клиент Java, который использует
**                   сеансовое зерно My8iEJB, хранящееся на
**                   сервере данных Oracle8i.
**  Задачи клиента:
**  1. Найти домашний интерфейс зерна.
**  2. Удостовериться в подлинности клиента на сервере.
**  3. Активировать экземпляр Bean-класса My8iEJBClass.
**  4. Вызвать метод getFirst8iEJB() зерна.
*/

package client; // (см. пояснение 1)

// Импортируем удаленный интерфейс зерна (см. пояснение 2).
import server.My8iEJB;
```

```
// Импортируем домашний интерфейс зерна.
import server.My8iEJBHome;

// Импортируем класс ServiceCtx из пакета Oracle.
// (см. пояснение 3)
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// (см. пояснение 4)
// Импортируем классы Java, обязательные для использования JNDI.
import javax.naming.Context;
import javax.naming.InitialContext;

// (см. пояснение 5)
// Импортируем классы, специфичные для приложения.
import java.util.Hashtable;
import java.sql.*;

public class MyFirtsEJBClient {

    // (см. пояснение 6)
    public static void main(String[] args)
        throws Exception {
    // Проверим входной параметр.
    if (args.length != 4) {
        System.out.println("usage: Client "
             +"serviceURL objectName user password");
        System.exit(1);
    } // Конец if.

    // Создадим String-объекты для хранения входных параметров,
    // содержащих:
    // args [0] = sess_sh://data-i.com:2481:orcl;
    // args [1] = /test/My8iEJB;
    // args [2] = scott;
    // args [3] = tiger;
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];

    // Создадим объект Hashtable для хранения переменных среды.
    Hashtable env = new Hashtable();
    // Обязательная установка свойств JNOI-среды безопасности,
    // подлежащих передаче объекту Context.

    // (см. пояснение 7)
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, password);
    env.put(Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN);

     // Создадим экземпляр класса Context.
     Context ic = new InitialContext (env);

     // Создадим экземпляр домашнего интерфейса объекта EJB.
     My8iEJBHome myEJBHome = null;

     // Создадим удаленный объект.
     My8iEJB my8iEJBBean = null;

    System.out.println("Variable Created Properly");
  System.out.println("serviceURL : " + serviceURL);
  System.out.println("objectName : " + objectName);
```

```
        try {
             // Воспользуемся домашним интерфейсом для нахождения
             // объекта EJB (см. пояснение 8).
             myEJBHome =
                (My8iEJBHome)ic.lookup (serviceURL + objectName);
             System.out.println("Home Lookup Properly");

             // Создадим экземпляр зерна (см. пояснение 9).
             my8iEJBBean = myEJBHome.create();

             System.out.println("Bean Created Properly");

        } // Конец try.
        catch ( Exception ex ) {
             System.out.println("Cannot locate"
             +" or create My8iEJB object");
             System.err.println("error : " + ex);
             System.exit(1);
        } // Конец catch.

        // Создадим Java-объект String для хранения результата
        // вызова метода зерна.
        String outputFromBean = null;
        // Вызовем удаленный метод EJB: getFirst8iEJB().
        // Вызовем метод зерна с помощью удаленного экземпляра
        // my8iEJBBean и сохраним результат в Java-переменной
        // outputFromBean.

        try {
             // (см. пояснение 10)
             outputFromBean = My8iEJBBean.getFirst8iEJB();
             System.out.println("Method Invocation went well: "
                       + outputFromBean);
        } // Конец try
        catch (Exception ex) {
               System.err.println(" Unable to invoke "
               + " the getFirst8iEJB() method : " + ex);
               System.exit(1);
        } // Конец catch
    }
}
```

Пояснения к приложению MyFirstEJBClient:

1. Этот оператор объявляет класс MyFirstEJBClient и указывает, что файл класса Java находится в пакете с именем client.
2. Этот и последующий операторы импортируют соответственно удаленный server.My8iEJB и домашний server.My8iEJBHome интерфейсы. Клиенты, использующие зерна уровня предприятия, обязаны импортировать эти интерфейсы в свои программы. Обратите внимание на то, что класс server.My8iEJBClass импортировать в клиентскую программу не нужно. Почему? Вспомните, что клиент не может напрямую обращаться к данному классу. Единственными обязательными классами, необходимыми клиентам для работы с методами зерна, являются объекты его домашнего и удаленного интерфейсов.
3. Этот оператор импортирует Oracle-класс ServiceCtx, расширяющий JNDI-класс Context. Клиенты, которые используют зерна EJB, размещенные в базе данных Oracle8*i*, должны устанавливать тип безопасности для соединения с БД. Существуют три возможных значения (см. приложение А):
   - **NON_SSL_LOGIN** Подлинность клиента проверяется во время регистрации именем пользователя и паролем.

- **SSL_CREDENTIAL** Клиент подтверждает свою подлинность перед сервером, предоставляя имя пользователя и пароль, зашифрованные по соединению Secure Socket Layer (SSL). Сервер подтверждает свою подлинность перед клиентом, предоставляя свои характеристики.
- **SSL_LOGIN** Клиент подтверждает свою подлинность перед сервером с помощью имени пользователя и пароля в границах протокола регистрации по соединению SSL. Сервер не подтверждает свою подлинность перед клиентом.

4. Этот и следующий операторы импортируют соответственно классы `Context` и `InitialContext`. Объект `InitialContext` нужен клиенту для поиска ссылки на домашний интерфейс зерна.
5. Этот и следующий операторы импортируют `Hashtable` и все пакеты Java, специфичные для приложения. Напомним, что объект `Hashtable` нужно создавать для хранения свойств среды безопасности зерна.
6. Этот оператор описывает метод `main()`. Данный клиент является приложением Java, поэтому метод `main()` нужно описать в классе.
7. Этот и три последующих оператора устанавливают свойства среды безопасности, которые необходимы JNDI-объекту `Context` для аутентификации клиентов, обращающихся к БД. Сервер данных Oracle — это защищенный сервер, поэтому перед обращением к серверу клиентское приложение должно удостоверять себя. Клиент удостоверяет себя, когда объект CORBA или EJB начинает новый сеанс работы.
8. Этот оператор получает ссылку на объект домашнего интерфейса зерна посредством метода `InitialContext lookup()`.
9. Этот оператор использует метод `create()` объекта домашнего интерфейса, чтобы создать экземпляр удаленного интерфейса зерна.
10. Этот оператор использует удаленный экземпляр зерна для вызова метода класса зерна.

Шаг 11. Компиляция клиента. Обратите внимание на то, что в `CLASSPATH` нужно включить `ejb-jar`-файл `My8iEJB.jar`, созданный на шаге 8 этого раздела и `jar`-файл `My8iEJBClient.jar`, сгенерированный инструментом `deployejb` на шаге 9.

```
// Установим CLASSPATH и скомпилируем клиента.
javac -classpath %YOUR_CLASSPATH_VARIABLE% MyFirstEJBClient.java
```

Шаг 12. Выполнение клиента.

```
// Если CLASSPATH не установлен, нужно сделать это сейчас.
// Выполним клиент.
// При желании можно создать переменную для хранения URL службы
// Oracle8i. Так, содержимым переменной ORACLE_SERVICE может быть:
// sess_sh://data-i.com:2481:orcl
// Выполним клиент.

java -classpath %YOUR_CLASSPATH_VARIABLE%
client.MyFirstEJBClient %ORACLE_SERVICE% /test/My8iEJB scott tiger
```

В листинге 3.16 представлены выходные данные, какие следует ожидать после выполнения клиента.

Листинг 3.16. Выходные данные клиента

```
Variable Created Properly
serviceURL : sess_iiop://data-i.com:2481:ORCL
objectName : /test/My8iEJB
Home Lookup Properly
Bean Created Properly
Method Invocation went well:
```

```
First EJB application stored in the Oracle8i database
on data-i.com server from the Oracle8i Java Component
Programming with EJB, CORBA, and JSP book.
```

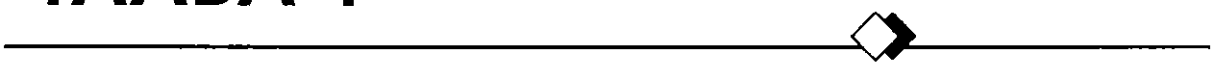
Итоги

В этой главе говорилось о базовых характеристиках технологии Enterprise JavaBeans и архитектуре Oracle8*i* JServer. В частности, было рассказано о:

- Двух различных типах зерен: сеансовых и сущностных
- Способах управления транзакциями в границах Enterprise JavaBeans
- Построении сеансового зерна и описании домашнего и удаленного интерфейсов EJB и Bean-класса
- Упаковке зерен в файлах `ejb-jar` для внедрения
- Файле дескриптора внедрения
- Внедрении сеансового зерна в базу данных Oracle8*i*
- Создании клиентов EJB, которые используют сеансовые зерна, находящиеся в базе данных Oracle8*i*

В главе 4 мы расскажем о создании компонентов EJB (на Java и SQLJ), которые работают с данными, хранящимися в базе данных Oracle8*i*.

ГЛАВА 4



Разработка сеансовых зерен EJB

В предыдущей главе обсуждались основные концепции технологии Enterprise JavaBeans (EJB); было разработано и внедрено `My8iEJB` – простое сеансовое зерно, метод которого, будучи вызванным клиентом, возвращает вызывающему Java-объект типа `String`. В этой главе вы примените полученные знания и разработаете сеансовые зерна, которые будут манипулировать данными, хранящимися в базе данных Oracle8*i*. А самое главное – научитесь создавать Bean-классы при помощи JDBC и SQLJ.

В частности вы узнаете о:

- Поддержке транзакций сервером Oracle8*i* JServer для компонентов EJB и CORBA
- Использовании атрибутов транзакций EJB при описании политики транзакций Enterprise JavaBeans
- Разработке приложений БД, где используются сеансовые зерна

Oracle8*i* JServer: поддержка транзакций

В моделях компонентов EJB и CORBA реализуется распространение контекста транзакций и распределенные транзакции. Однако выбор нужного способа остается за производителем. Например, в Oracle8*i* JServer (версии 8.1.5 и 8.1.6) распространение контекста транзакций реализуется, но распределенные транзакции не поддерживаются. В версии 8.1.7 они будут поддерживаться только для клиентов, не использующих зерна.

Транзакция характеризуется так называемыми свойствами ACID, согласно которым она

- **Atomic** (атомарна, т.е. неделима). Единица работы, в процессе которой все внесенные во время транзакции изменения либо полностью реализуются, либо целиком отменяются.
- **Consistent** (согласована). Изменения, вносимые во время транзакции, должны всегда оставлять базу данных в согласованном состоянии.
- **Isolated** (изолирована). Изменения, вносимые во время транзакции, вне нее невидимы.
- **Durable** (долговременна). Изменения, вносимые во время транзакции, фиксируются или откатываются назад.

В Oracle8*i* JServer поддерживаются два транзакционных интерфейса API, используемые в приложениях CORBA и EJB:

- *API службы транзакций Java* (JTS, Java Transaction Service) – менеджер транзакций, который поддерживает отображение спецификации 1.0 JTA (Java Transaction API), реализующей отображение спецификации 1.1 службы объектных транзакций (OTS – Object Transaction Service) OMG. В СУБД Oracle8*i* (версий 8.1.5, 8.1.6 и новой 8.1.7) JTS включен в состав Oracle8*i* JServer. В Oracle8*i* версии 8.1.7 JTS поддерживается только для обратной совместимости; Oracle 8.1.7 соответствует спецификации EJB 1.1, требующей использовать JTA для унификации поддержки автономных клиентских и серверных объектов Java. Разработчики CORBA обязаны привлекать JTS к клиентским ограниченным транзакциям и управлению объектными транзакциями на сервере. Транзакцию называют ограниченной (demarcated), когда у нее есть четко определенные точки начала и окончания. Разработчики вправе явно ограничить транзакцию, выполнив метод `begin()`, и явно закончить транзакцию, выполнив метод `commit()` или `rollback()`. Отметим, что EJB-клиенты, не использующие зерна, также могут применять JTS для управления транзакциями, в то время как клиенты, использующие зерна, не могут. В Oracle8*i* клиент, не использующий зерна размещается вне базы данных Oracle, а использующий – только в БД. О клиентских и серверных объектах CORBA, применяющих JTS, говорится в главе 8.

- Интерфейс javax.jts.UserTransaction описывает методы, которые позволяют приложению явным образом определять границы транзакций. Архитектура Enterprise JavaBeans обязывает EJB-контейнеры поддерживать ограничение транзакций, предоставляя реализацию интерфейса UserTransaction. Последний применяется клиентами, использующими зерна TX_BEAN_MANAGED и не использующими их. Так, Oracle предоставляет oracle.aurora.jts.client.AuroraTransactionService — реализацию интерфейса UserTransaction, которая позволяет разработчикам EJB явным образом ограничивать транзакции в своих программах. Транзакции ограничиваются явно выполнением соответствующего метода begin() или commit() интерфейса UserTransaction. Например, клиент начинает транзакцию явным вызовом UserTransaction.begin(), а завершает явным вызовом либо UserTransaction.commit(), либо UserTransaction.rollback(). Для серверных транзакций выбирают неявное ограничение, обязывая управлять транзакциями EJB-контейнер. UserTransaction используется в зернах уровня предприятия, когда зерно функционирует с атрибутом транзакций TX_BEAN_MANAGED.

Отметим, что в Oracle8*i* JServer (Oracle8*i* версий 8.1.5, 8.1.6 и 8.1.7) не поддерживается вложение транзакций. Транзакции называются *вложенными* (nested), когда программа пытается начать новую транзакцию до завершения или отката существующей. Попытки их использования заставляют службу транзакций устанавливать исключительную ситуацию SubTransactionUnavailable. Отметим также, что методы JTS для таймаута (задержки) транзакций, например setTimeout(), и службы транзакций, поставляемые в Oracle8*i* (версий 8.1.5 и 8.1.6), не способны взаимодействовать с другими реализациями OTS и управлять распределенными транзакциями. В Oracle8*i* (версии 8.1.7) управление транзакциями, распределенное среди нескольких серверов баз данных, доступно автономным клиентам Java, но не серверным объектам. Кроме того, заметим, что, будучи частью EJB 1.1, UserTransaction был перенесен из javax.jts.UserTransaction в javax.jta.UserTransaction.

Еще раз о декларативных атрибутах транзакций

Перейдем к описанию семантики атрибутов транзакций EJB и способам их использования, чтобы EJB-контейнер управлял средой транзакций зерна надлежащим образом. В последующих разделах этой главы будет разработано EJB-приложение, использующее атрибуты транзакций, — и тогда станет ясно, как и когда их нужно указывать в файле дескриптора внедрения зерна.

EJB-серверы — транзакционные серверы. Поставщики зерен могут поручить управление всеми транзакциями EJB-контейнеру. Для этого служат атрибуты транзакций EJB. Транзакционная политика Enterprise JavaBeans определяется атрибутами транзакций, указываемыми в файле дескриптора внедрения EJB. Возможность определять степень участия зерен уровня предприятия в транзакциях путем программирования атрибутов является одним из самых важных свойств компонентной модели EJB.

Атрибут транзакций сообщает EJB-контейнеру, какова область действия транзакции до и после выполнения методов зерна. В EJB область включает в себя зерна (сеансовые или сущностные), участвующие в конкретной транзакции. Она начинается тогда, когда клиент вызывает метод зерна. Напомним, что транзакция состоит из одной или нескольких задач, и для успешного полного ее выполнения должны быть успешно выполнены все задачи, составляющие единицу работы.

В тех случаях, когда клиентская программа во время конкретной транзакции может использовать несколько зерен, транзакция, начав действовать, распространяется на остальные вновь создаваемые зерна.

На EJB-сервере реализуется принцип двухфазного завершения, или двухфазной фиксации (two-phase commit): распространение контекста транзакций и распределение транзакций. EJB-сервером Oracle8*i* (Oracle8*i* версий 1.5 и 1.6) распространяется только контекст транзакций. Таким образом,

контексты транзакций, генерируемые на EJB-сервере Oracle8*i*, никогда не распространяются вне базы данных Oracle8*i*. Отметим, что в одной службе контексты транзакций не могут охватывать несколько серверов или несколько сеансов БД.

Приложение `QualityControl`, которое будет создано в главе 5, иллюстрирует принцип распространения области действия транзакций одного зерна на другое. В клиентском приложении `QcClient` область действия транзакции охватывает зерна `QcBean` и `ObservationBean`. Она начинается, когда `QcClient` создает экземпляр `observationRemote` и вызывает метод `getAllObs()` класса `ObsImp`. Затем распространяется на вновь создаваемый экземпляр `QcRemote` (см. главу 5).

Управление транзакциями

Способ управления средой транзакций зерна указывает EJB-контейнеру переменная `TransactionAttribute` файла дескриптора внедрения EJB. Транзакциями можно управлять на двух уровнях:

- **На уровне зерна** Здесь атрибут применяется по отношению ко всем методам зерна. Пример управления транзакциями на уровне зерна приведен в листинге 4.1.

  Листинг 4.1. Управление транзакциями на уровне зерна

```
// Имя файла:    Obs.ejb
// Назначение:   Файл дескриптора внедрения для сеансового зерна
//               Observation.

SessionBean obsserver.ObsImp {

  BeanHomeName = "test/ObservationBean";
  HomeInterfaceClassName = obsserver.ObsHome;
  RemoteInterfaceClassName = obsserver.ObsRemote;

  SessionTimeout = 0;
  AllowedIdentities = { SYSDBA };
  //
  TransactionAttribute = TX_REQUIRED;
  RunAsMode =LIENT_IDENTITY;
}
```

- **На уровне метода** На этом уровне транзакционный атрибут применяется только к методу и переопределяет значение, указанное на уровне зерна. Это касается значений всех атрибутов транзакций, кроме `TX_BEAN_MANAGED`. Напомним, что его нельзя смешивать с другими атрибутами и переопределять на данном уровне.

  Пример управления транзакциями на уровне метода приведен в листинге 4.2.

  Листинг 4.2. Управление транзакциями на уровне метода

```
// Имя файла:    Obs2.ejb
// Назначение:   Файл дескриптора внедрения для сеансового зерна
//               Observation.

SessionBean obsserver.ObsImp {

  BeanHomeName = "test/ObservationBean";
  HomeInterfaceClassName = obsserver.ObsHome;
  RemoteInterfaceClassName = obsserver.ObsRemote;

  SessionTimeout = 0;
  AllowedIdentities = { SYSDBA };
  // Управление транзакциями на уровне зерна.
  TransactionAttribute = TX_NOT_SUPPORTED;
  RunAsMode = CLIENT_IDENTITY;
```

```
    public ObsHelper insertObs(ObsHelper obs){
        // Управление транзакциями на уровне метода.
        TransactionAttribute = TX_REQUIRED;
    }
}
```

В спецификациях EJB (1.0 и 1.1) описаны шесть возможных значений для атрибута транзакций:

- TX_NOT_SUPPORTED
- TX_REQUIRED
- TX_SUPPORTS
- TX_REQUIRES_NEW
- TX_MANDATORY
- TX_BEAN_MANAGED

TX_NOT_SUPPORTED Атрибут можно объявлять для всего зерна или его конкретных методов. В первом случае TX_NOT_SUPPORTED запрещает EJB-контейнеру Oracle8*i* вызывать поддержку транзакций для методов зерна. Значение атрибута транзакций, объявленное на уровне метода, будет явным образом переопределять значение атрибута транзакций, указанное на уровне зерна. Это касается всех атрибутов транзакций EJB, за исключением значения атрибута TX_BEAN_MANAGED. Заметим, что зерно, работающее в режиме TX_NOT_SUPPORTED, не выполняет никаких SQL-операций. При попытке сделать это EJB-сервер Oracle8*i* устанавливает исключительную ситуацию. Хороший пример применения значения атрибута TX_NOT_SUPPORTED — приложение My8iEJB, разработанное в разделе "Создание первого Enterprise JavaBean" главы 3. Обратите внимание: это EJB-приложение не выполняет никаких SQL-операций. Можно модифицировать файл дескриптора внедрения My8iEJB.ejb, указав значение атрибута транзакций TX_NOT_SUPPORTED:

```
// Имя файла: My8iEJB.ejb


    // Укажем контейнеру, чтобы он при проверке
    // идентификационных характеристик безопасности клиента
    // использовал атрибут CLIENT IDENTITY.
    RunAsMode = CLIENT_IDENTITY;

    // Дополнительная конструкция для указания значения атрибута
    // транзакций.
    TransactionAttribute = TX_NOT_SUPPORTED;
}
```

Заметим, что при изменении файла дескриптора внедрения зерна последнее нужно внедрять повторно. Напомним, что EJB-контейнер просматривает файл дескриптора внедрения только в процессе развертывания, но транзакционную политику зерна реализует во время работы.

TX_REQUIRED В базе данных Oracle8*i* в транзакции может участвовать большое число клиентских и серверных объектов, и порой она заключает в себе множество методов этих объектов. Значение TX_REQUIRED сообщает контейнеру о том, что метод зерна должен вызываться в области действия транзакции. Область определяется по ее контексту (псевдообъекту), переносящему состояние транзакции с одного объекта на другой. Например, когда клиент вызывает метод *begin транзакции*, служба транзакций клиента неявно создает контекст транзакции клиента. После создания контекста методы вызываются в его границах.

При объявлении в дескрипторе внедрения значения атрибута транзакций TX_REQUIRED возможны два сценария:

- **Клиент начинает транзакцию явным образом** При управлении средой транзакций клиентские приложения, не использующие зерна, применяют JTS или JTA-интерфейсы oracle.aurora.jts.client.AuroraTransactionService и oracle.aurora.jts.util.TS, а использующие зерна — интерфейс javax.jts.UserTransaction (8.1.5 и 8.1.6, но javax.jta.UserTransaction в 8.1.7). В обоих наборах интерфейсов описываются методы begin() и commit(), позволяющие клиентам управлять средой транзакций зерна. Например, клиент начинает транзакцию, вызывая транзакционный метод begin(). Тогда его служба транзакций неявно создает объект контекста транзакции, а затем распространяет контекст на каждый объект, вызываемый клиентом. Кажется, что объект транзакции применяется службой транзакций клиента, хотя на самом деле им пользуется EJB-контейнер, передавая вызовы методов зерна. В листинге 4.3 показано, как клиент, не применяющий зерна, может явно начать транзакцию, вызвав метод begin() из интерфейса oracle.aurora.jts.util.TS. Листинг 4.4 является примером использования зерном экземпляра SessionContext для вызова методов begin() и commit() из интерфейса UserTransaction.

Листинг 4.3. Транзакция, ограниченная клиентом без зерен

```
// Транзакциями управляет клиент, не применяющий зерна.
// Импортируем интерфейсы.
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.jts.*;
import oracle.aurora.jts.util.TS;

..
public class ObsClientApp {


// Служба транзакций должна быть инициализирована до того, как
// клиент начнет транзакцию.
AuroraTransactionService.initialize (ic, serviceURL);

// Клиент берет управление на себя, начиная
// транзакцию явным образом.
TS.getTS().getCurrent().begin();

// Вызовем один или несколько серверных методов.

// Клиент явно заканчивает транзакцию фиксацией.
TS.getTS().getCurrent().commit (false);

// или
TS.getTS().getCurrent().commit (true);
..
}
```

Листинг 4.4. Транзакция, ограниченная клиентом зерна

```
// Транзакциями управляет клиент, применяющий зерна.
// Импортируем интерфейс UserTransaction.
import javax.jts.UserTransaction;

..
public class PassedObsImp implements SessionBean {
      SessionContext ctx;

..
public void setSessionContext (SessionContext ctx) {
      this.ctx = ctx;
}
```

```
..
// Начнем транзакцию.
ctx.getUserTransaction().begin();

..
// Вызовем методы.

// Закончим транзакцию.
ctx.getUserTransaction().commit();
}
```

- **Клиент не начинает транзакцию** Напомним, что поставщик зерен не обязан явно управлять транзакциями через клиента. Беря за основу транзакционные атрибуты, указанные для зерна в файле дескриптора внедрения, EJB-сервер будет сам неявно управлять транзакциями. В том случае, когда клиент не начинает транзакцию, контейнер начинает новую транзакцию с каждого вызова метода, и транзакция пытается завершиться после окончания каждого из них. Протокол завершения (фиксации) заполняется до отправки результатов зерна клиенту. Если в текущей транзакции принимают участие другие зерна (т.е. началась область действия транзакции), EJB-сервер Oracle8*i* посылает контекст транзакции этим серверным объектам, а также другим запрашиваемым ресурсам.

TX_SUPPORTS Это значение атрибута указывает контейнеру на необходимость включить в область действия транзакции все зерно или конкретный метод. Если клиент установил контекст транзакции, контейнер зерна использует этот контекст. В обратном случае EJB-методы вызываются без транзакционной поддержки. Более того, при вызове метода, являющегося частью области действия транзакции, другим методом транзакция распространяется с первого на второй.

TX_REQUIRES_NEW Этот значение атрибута говорит контейнеру о том, чтобы он всегда вызывал методы зерна с новой транзакцией. Контейнер зерна начинает транзакцию перед каждым вызовом метода зерна и пытается завершить ее после его выполнения. Отметим, что перед созданием новой транзакции любая существующая приостанавливается и возобновляется после окончания новой.

TX_MANDATORY Указанное на уровне метода, это значение атрибута обязывает EJB-контейнер использовать для управления транзакциями контекст транзакции клиента. Если транзакционный объект клиентом не был создан, контейнер устанавливает для клиента исключительную ситуацию `TransactionRequired`. Заметим, что в этом сценарии контекст транзакции клиента передается ресурсам или другим объектам зерен, вызываемым из объекта зерна.

TX_BEAN_MANAGED Как видно из названия, значение `TX_BEAN_MANAGED` обязывает контейнер передать управление транзакциями самому зерну. Сеансовое зерно обращается к службе транзакций через объект `SessionContext` — параметр метода `setSessionContext()`. Контейнер должен предоставить зерну объект `Context`, с помощью которого можно будет считывать все другие объекты среды.

Во время инициализации (т.е. при конкретизации зерна) зерно вызывает метод `setSessionContext()`, чтобы получить свой объект `SessionContext`. Его передает методу EJB-контейнер.

Указывая в файле дескриптора внедрения значение атрибута `TX_BEAN_MANAGED`, Bean-класс должен использовать метод интерфейса `javax.jts.UserTransaction`, чтобы управлять транзакциями самостоятельно. Запомните, что `TX_BEAN_MANAGED` нельзя смешивать с другими значениями атрибутов транзакций и переопределять объявлениями атрибутов на уровне методов. При использовании транзакций, управляемых зернами, их границы охватывают методы зерна; можно начать транзакцию в одном методе, а откат или завершение транзакции произвести в отдельном методе, вызываемом впоследствии.

Контейнер делает интерфейс javax.jts.UserTransaction доступным для зерна при помощи метода EJBContext.getUserTransaction(), что показано на следующем примере:

```
import javax.jts.UserTransaction;
... EJBContext ic = ...;
...
UserTransaction tx = ic.getUserTransaction();
tx.begin() ;
... // операции
tx.commit();
```

Контейнер должен управлять транзакциями зерна в режиме TX_BEAN_MANAGED следующим образом:

- Когда клиент вызывает сеанс типа STATEFULTX_BEAN_MANAGED, контейнер приостанавливает выполнение всех поступающих транзакций. В такой ситуации контейнер позволяет экземпляру сеанса инициировать транзакцию с помощью интерфейса javax.jts.UserTransaction. Экземпляр связывается с транзакцией и остается связанным с ней до ее окончания.
- Когда инициируемая зерном транзакция связывается с экземпляром, методы экземпляра выполняются в ее границах.

Бизнес-метод, начинающий транзакцию, может быть выполнен без завершения или отката транзакции. Однако контейнер должен сохранять связь между транзакцией и экземпляром для множества клиентских вызовов, пока транзакция не будет закончена. Например, в ситуациях, когда клиент, использующий зерно, начинает внутреннюю транзакцию (вызывая метод зерна) после запуска службы транзакций, внешняя будет приостановлена и возобновлена после того, как управляемая контейнером (внутренняя) начнется, выполнится, закончится или приостановится.

Рекомендации по использованию атрибутов транзакций

- По умолчанию атрибуту транзакций следует присваивать значение TX_REQUIRED. Зерна EJB с этим атрибутом можно достаточно легко компоновать, выполняя всю работу в области действия одной транзакции с помощью конкретного транзакционного API.
- Атрибут транзакций TX_REQUIRED_NEW полезен тогда, когда методы зерна нужно завершать без каких бы то ни было условий, независимо от того, выполняется ли какая-либо транзакция или нет. Примером является метод зерна, выполняющий протоколирование (запись в журнал). Пример такого зерна – сеансовое зерно serversideLogging (приложение A в "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide").

Управление доступом

В файле дескриптора внедрения можно описать для зерен политику управления доступом. Достаточно атрибуту транзакций AllowedIdentities присвоить определенное значение. Это может быть имя пользователя, роль Oracle или их комбинация. Когда атрибуту AllowedIdentities присваивается значение (имя пользователя, роль или комбинация), применять зерно могут только те пользователи, которые идентифицируются этим значением (скажем, пользователи, имеющие данную роль). Как и транзакциями, доступом можно управлять на уровне зерна или метода.

Управление доступом на уровне зерна

В файле дескриптора внедрения Obs.ejb, который будет создан ниже, в разделе "Разработка приложений баз данных с использованием сеансовых зерен", с помощью атрибута AllowedIdentities организуется управление доступом для

всего зерна. В этом примере атрибуту присваивается Oracle-роль SYSDBA, чтобы работать с зерном могли те пользователи Oracle, которым она предоставлена. Напомним, что режим управления доступом, указанный для зерна, позволяет обращаться ко всем методам, описанным в нем. Например:

```
// Все пользователи, кому предоставлена Oracle-роль SYSDBA, могут
// применять все методы зерна.
AllowedIdentities = {SYSDBA};

// Или разрешается применять зерно только пользователю SCOTT.
AllowedIdentities = {SCOTT};
```

Управление доступом на уровне метода

Режим управления доступом для зерна можно переопределить, если задать его на уровне метода. Это можно сделать в описаниях методов в файле дескриптора внедрения. Например:

```
// ejb-файл дескриптора внедрения.
...
...
// Установим режим на уровне зерна.
AllowedIdentities = {SYSDBA};

// Переопределим режим уровня зерна
// режимом уровня метода.
public void insertObs(ObsHelper obs)
    throws SQLException,RemoteException {
    TransactionAttribute = TX_REQUIRED;
    RunAsMode = CLIENT_IDENTITY;

    // Этот метод может вызывать только Oracle-пользователь SCOTT.
    AllowedIdentities = {SCOTT};
}
```

Заметим, что когда вы указываете режим управления доступом на уровне метода, последний должен быть или общим бизнес-методом, или методом ejbCreate() зерна. Здесь режим управления на уровне зерна переопределяется методом ejbCreate():

```
public void ejbCreate()
    throws RemoteException, CreateException {
    AllowedIdentities = {SCOTT};
}
```

Синхронизация сеансов

EJB-разработчики, которым нужно получать от EJB-контейнера уведомления о транзакционном состоянии их зерна, при желании могут реализовать в своем Bean-классе интерфейс SessionSynchronization. При этом контейнер с помощью последнего сохраняет состояние зерна в БД и использует его для уведомления зерна о границах транзакций. Для этого EJB-контейнер пользуется методами, указанными в интерфейсе SessionSynchronization. Описание интерфейса javax.ejb.SessionSynchronization представлено в листинге 4.5.

Листинг 4.5. Интерфейс javax.ejb.SessionSynchronization

```
public interface javax.ejb.SessionSynchronization {
    // (см. пояснение 1)
    public abstract void afterBegin()
        throws RemoteException;
    // (см. пояснение 2)
    public abstract void afterCompletion(boolean committed)
        throws RemoteException;
    // (см. пояснение 3)
    public abstract void beforeCompletion()
```

```
        throws RemoteException;
}
```

Пояснения к интерфейсу `javax.ejb.SessionSynchronization`**:**

1. Метод `afterBegin()` уведомляет экземпляр сеансового зерна о начале новой транзакции. После начала транзакция распространяется на последующие методы зерна, которые будут вызываться в ее контексте. Например, его можно использовать для чтения данных в БД и их кэширования в полях экземпляра.
2. Метод `afterCompletion()`, как видно из его названия, сообщает экземпляру о выполнении (фиксации) или невыполнении (откате) транзакции. В отличие от метода `afterBegin()`, выполняется без контекста транзакций. Параметр `committed` устанавливается истинным (`true`), если транзакция завершена, и ложным (`false`), если произведен ее откат.
3. Метод `beforeCompletion()` уведомляет сеансовое зерно о том, что транзакция близится к завершению. Например, экземпляр может использовать этот метод для записи всех кэшируемых данных в БД. Как и `afterBegin()`, выполняется в подходящем транзакционном контексте.

▼ ВНИМАНИЕ! *При использовании в зерне вызовов JDBC или SQLJ для обновления базы данных не применяйте их для работы с транзакциями. При помощи JDBC или SQLJ нельзя напрямую указывать в программах SQL-операции завершения и отката. Позвольте зерну самому управлять транзакциями, указав атрибут `TX_BEAN_MANAGED` в дескрипторе внедрения или через интерфейс `javax.jts.UserTransaction`.*

Например, не следует явно использовать метод `commit()` интерфейса `Connection` пакета `java.sql` для фиксации изменений в БД. Применение метода в контексте с JDBC показано в листинге 4.6:

Листинг 4.6. Использование метода `commit()`

```
Connection aConnection = ...;

...
// При использовании метода commit() транзакция завершается
// явным образом.
// НЕ ВЫПОЛНЯЙТЕ ОПЕРАЦИЮ ЗАВЕРШЕНИЯ В ЗЕРНЕ ЯВНЫМ ОБРАЗОМ.
aConnection.commit();
```

Несмотря на то что компоненты EJB очень гибки, на бизнес-методы, реализуемые Bean-классом, налагается ряд ограничений. Разработчики зерен, реализуя методы Bean-класса, должны их учитывать.

Программные ограничения

В спецификации EJB 1.0 установлены следующие программные ограничения:

- EJB не может ни запускать новые потоки, ни прерывать выполнение функционирующего. В Oracle8*i* (версий 8.1.5 и 8.1.6) при запуске в EJB нового потока исключительная ситуация не устанавливается, но поведение приложения становится непредсказуемым, виной чему взаимодействие с локальными объектами потоков в ORB.
- В EJB запрещено использовать примитивы синхронизации потоков.
- В EJB запрещено напрямую вызывать базовый менеджер транзакций. Единственное исключение – применение зерен с атрибутом транзакций `TX_BEAN_MANAGED`. Такие зерна для ограничения транзакций могут пользоваться интерфейсом `javax.jts.UserTransaction`.

- В EJB запрещено изменять `java.security.Identity`. При попытке сделать это устанавливается исключительная ситуация `java.security.SecurityException`.
- В EJB запрещено использовать JDBC- или SQLJ-методы `commit` и `rollback` либо выдавать прямые SQL-команды `commit` или `rollback` в методах Bean-класса.

В Oracle8*i* версии 8.1.7 базы данных будут соответствовать спецификации EJB 1.1, поэтому там сохранятся программные ограничения, устанавливаемые в этой спецификации.

Разработка приложений баз данных с использованием сеансовых зерен

По определению сеансовые зерна выполняют конкретные бизнес-задачи — компоненты Java, содержащие модели бизнеса-процессов. Напомним, что компонент — это независимая программная (т.е. не зависящая от приложения) единица, выполняющая конкретную задачу. В начале этой главы мы создали сеансовое зерно `ObservationBean`, которое работает с данными наблюдений за атмосферой, хранящимися на сервере данных Oracle8*i* отдела исследования ураганов (HRD, Hurricane Research Division) Национального управления океана и атмосферы (NOAA, National Oceanic and Atmospheric Administration). Точнее `ObservationBean` представляет собой набор бизнес-методов, позволяющих клиентам (ученым, наблюдающим за атмосферой) работать с таблицей `OCEANIC_OBSERVATION_LIST`. Таблица входит в состав *научной схемы наблюдений* (Observation), представленной во введении в начале этой книги, где содержится полное описание схем `PURCHASE_ORDER` и `OBSERVATION`.

Теперь все компоненты зерна `ObservationBean` мы создадим с помощью Java. Enterprise JavaBean будет загружено в базу данных Oracle8*i* (версий 8.1.5 и 8.1.6) и опубликовано там. Напомним, что EJB-сервер Oracle8*i* предоставляет доступное для JNDI пространство имен, в котором он во время внедрения сохраняет имя публикуемого зерна и ссылку на домашний объект зерна. Обращаться к БД Oracle можно разными способами:

- **Java Database Connectivity (JDBC)** Даст возможность встраивать динамические SQL-операторы в программы Java, позволяя манипулировать данными, хранящимися в БД Oracle8 или Oracle8*i* (версий 1.5 и 1.6). Используйте JDBC в клиентской программе Java (находящейся в любом месте сети) для работы с SQL-конструкциями, размещенными в базе. Кроме того, JDBC удобно применять в исходном программном тексте Java, находящемся внутри базы данных Oracle8*i* (версий 1.5 и 1.6), в виде хранимых процедур Java или SQLJ, зерен EJB и приложений CORBA (только Java).
- **SQLJ** Позволяет встраивать в программы Java статические SQL-операторы.
- **Internet Inter-ORB Protocol (IIOP)** позволяет обращаться к компонентам EJB и CORBA, хранящимся в Oracle8*i* версии 1.5 и выше.

Обзор JDBC-драйверов Oracle

Для клиентов Java и SQLJ, обращающихся к базе данных Oracle, корпорация Oracle предлагает JDBC-драйверы четырех типов:

- **Тонкий (`thin`) драйвер** 100-процентный драйвер Java для использования на стороне клиента, особенно в случаях, когда использовать JNDI нельзя. Драйвер не требует установки Oracle в адресном пространстве клиента. Его применяют для таких клиентов web-браузеров, как апплеты, сервлеты и Java Server Pages (JSP). В следующем программном фрагменте тонкий драйвер Oracle используется для соединения с БД:

```
// Создадим Java-объект String для хранения хост-имени:
String myhost = "data-i.com";
// Самый распространенный синтаксис соединения таков:
```

```
String connStr1 = "jdbc:oracle:thin:myhost:1521:orcl";

// ИЛИ
String connStr2 =
"jdbc:oracle:thin:@(description=(address=(host=myhost)
    (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))";

// Откроем соединение для тонкого драйвера JDBC с помощью connStr1.
Connection conn =
      DriverManager.getConnection(connStr1, "scott", tiger");

// ИЛИ откроем соединение для тонкого драйвера JDBC с помощью
// connStr2.
Connection conn =
DriverManager.getConnection(connStr2, "scott", "tiger");
```

- **OCI-драйверы (OCI8 и OCI7)** Предназначены для использования на стороне клиента с установкой клиента Oracle. Драйвер используется, если клиенты находятся на том же сервере, что и сервер баз данных, или если на машине клиента было установлено клиентское программное обеспечение Oracle. OCI7- или OCI8-драйвер Oracle применяется для открытия соединения с БД:

```
// Откроем соединение для JDBC-драйвера OCI.
String myHostString = "data-i.com";
Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@myHostString",
                                   "scott", "tiger");
```

- **Серверный тонкий (thin) драйвер** В функциональном плане это то же самое, что и клиентский тонкий драйвер, только используется для программ, работающих внутри сервера Oracle. Применяется для хранимых процедур Java и SQLJ, зерен EJB и приложений CORBA, находящихся в БД Oracle8*i*, или обращения к удаленному серверу Oracle с сервера Oracle, выполняющего функции промежуточного ПО. Синтаксис такой же, что и у тонкого драйвера для автономных клиентских программ Java (см. листинг для *тонкого драйвера* выше).
- **Серверный внутренний драйвер** Предназначен для программ, функционирующих внутри целевого сервера (т.е. сервера Oracle, к которому необходимо обращаться). Используется так же, как и серверный тонкий драйвер, описанный выше.

Базовые функции серверного и клиентского JDBC-драйверов Oracle одинаковы. Они поддерживают следующие стандарты и средства (см. "Oracle8*i* JDBC Developer's Guide and Reference"):

- JDK 1.2.x/JDBC 2.0 или JDK 1.1.x/JDBC 1.22 (с Oracle-расширениями для функций JDBC 2.0). Отметим, что обе реализации используют разные наборы файлов классов.
- Одинаковый синтаксис и интерфейсы API
- Одинаковые расширения Oracle
- Полную поддержку многопоточных приложений

В листинге 4.7 показан процесс создания программы Java для обращения к серверу данных Oracle.

Листинг 4.7. Подключение посредством JDBC-драйверов Oracle

```
// Импортируем нужные пакеты Java.
// Следующие пакеты необходимы для всех клиентских программ
// Java/SQLJ, обращающихся к БД Oracle.
import java.sql.*;
import oracle.jdbc.driver.*;

// Необходимо зарегистрировать OracleDriver для программ Java,
// обращающихся к БД.
  DriverManager.registerDriver
        (new oracle.jdbc.driver.OracleDriver());
String myhost = "data-i.com";
// Откроем соединение для тонкого драйвера JDBC.
Connection conn = DriverManager.getConnection
  ("jdbc:oracle:thin:@(description=(address=(host=myhost)
        (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))",
    "scott", "tiger");

// Открытие соединения для JDBC-драйвера OCI.
String myHostString = "data-i.com";
Connection conn =
  DriverManager.getConnection ("jdbc:oracle:oci8:@myHostString",
        "scott", "tiger");

// Серверный внутренний драйвер.
Connection conn =
  new oracle.jdbc.driver.OracleDriver().defaultConnection();

// Подключение методом getConnection() класса OracleDriver
// с использованием внутреннего серверного драйвера.
Connection conn =
  DriverManager.getConnection("jdbc:default:connection:");
```

Подключение с помощью JNDI

Для создания нового объекта ServiceCtx задействуем метод lookup() JNDI-объекта InitialContext, указав URL в идентификаторе службы:

```
// Создадим объект Hashtable.
Hashtable environment = new Hashtable();

// Установим свойства среды с помощью
// свойств контекста URL_PKG_PREFIXES
// javax.naming.Context.SECURITY_PRINCIPAL
// javax.naming.Context.SECURITY_CREDENTIALS
// javax.naming.Context.SECURITY_ROLE
// javax.naming.Context.SECURITY_AUTHENTICATION

ServiceCtx service = (ServiceCtx)
initContext.lookup("sess_iiop://localhost:2481:ORCL");
```

Теперь все готово к построению сеансового зерна ObservationBean. Первый шаг разработки проекта – определение потребностей пользователей. Предположим, что после консультаций пользователи предъявили к сеансовому зерну ObservationBean следующие требования.

Нужно создать зерно, позволяющее пользователям манипулировать объектами OCEANIC_OBSERVATION_TYPE, хранящимися в таблице OCEANIC_OBSERVATION_LIST. Если конкретно, то клиенты хотят иметь возможность:

- Вводить новый объект OCEANIC_OBSERVATION_TYPE
- Вводить массив объектов OCEANIC_OBSERVATION_TYPE
- Получать перечень всех объектов OCEANIC_OBSERVATION_TYPE

- Просматривать всю информацию, имеющую отношение к объекту OCEANIC_OBSERVATION_TYPE, по конкретному идентификатору наблюдения (obs_id)

Таблица OCEANIC_OBSERVATION_LIST является частью научной схемы Observation (см. введение). В листинге 4.8 представлены типы Oracle8*i* и таблица, используемые зерном ObservationBean:

Листинг 4.8. Типы OCEANIC_OBSERVATION и OCEANIC_OBSERVATION_TYPE и таблица OCEANIC_OBSERVATION_LIST

```
CREATE TYPE OCEANIC_OBSERVATION AS OBJECT(
    latitude_deg        NUMBER(10,4),
    longitude_deg       NUMBER(10,4),
    windspeed_mps       NUMBER(10,4),
    adj_windspeed_mps   NUMBER(10,4),
    wind_direction_deg  NUMBER(6),
    pressure_mb         NUMBER(6))/
CREATE TYPE OCEANIC_OBSERVATION_TYPE AS OBJECT(
    obs_id          NUMBER(8),
    when_t          DATE,
    at_time         CHAR(8),
    station_id      NUMBER(6),
    produced_id     NUMBER(8),
    produced_by     REF PLATFORM_TYPE,
    obsobj          OCEANIC_OBSERVATION)
/
CREATE TABLE OCEANIC_OBSERVATION_LIST
       OF OCEANIC_OBSERVATION_TYPE;
/
```

ObservationBean использует также PL/SQL-функцию OBSACTIONS.GETOBSID() и PL/SQL-процедуру OBSACTIONS.INSERTOBS(). Первая генерирует новый идентификатор наблюдений, а вторая добавляет новую строку типа OCEANIC_OBSERVATION_TYPE в таблицу OCEANIC_OBSERVATION_LIST. Как GETOBSID(), так и INSERTOBS() являются частью PL/SQL-модуля OBSACTIONS (см. листинг 4.9). Разработчик Java может и не создавать функции и процедуры PL/SQL. Что использовать: только Java или SQLJ либо комбинацию Java/SQL и PL/SQL – зависит от разрабатываемого приложения. Корпорация Oracle рекомендует использовать Java или PL/SQL в случаях, когда:

"PL/SQL является расширением SQL и типы данных не преобразуются. В ситуациях, когда объем операций записи/чтения БД велик, а объем вычислений мал, программа, написанная на PL/SQL, всегда будет работать значительно быстрее написанной на Java. При большом же объеме вычислений и малом объеме операций чтения/записи БД применяйте Java, так как программа, написанная на этом языке, будет всегда работать значительно быстрее написанной на PL/SQL" (подробнее см. сайт technet.oracle.com/products/oracle8i/htdocs/jserver_faq/ 815faq0005.html).

В приложении ObservationBean решение об использовании функций и процедур PL/SQL нужно принимать с учетом двух условий:

- **Повышение производительности** ObservationBean – вполне законченное приложение базы данных. В данном случае с точки зрения производительности программа на PL/SQL быстрее, чем на Java.
- **Использование процедур/функций PL/SQL** Разработчику Java или поставщику зерен, который создает приложения Java, обращающиеся к БД, крайне важно знать, как вызывать процедуры и функции PL/SQL. Возможно, писать программные конструкции на PL/SQL и не понадобится, но не исключено, что их придется использовать в собственных программах Java. Ниже мы покажем, как это делается.

Листинг 4.9. PL/SQL-модуль OBSACTIONS

```
CREATE OR REPLACE PACKAGE OBSACTIONS AS
-- Получим новый Obs_id
    FUNCTION getObsId RETURN NUMBER;
    PRAGMA RESTRICT_REFERENCES(getObsId,WNDS,WNPS,RNPS);

-- Добавим новое наблюдение
    PROCEDURE insertObs(p_newobsid IN NUMBER,
                        p_whent       IN VARCHAR2,
                        p_attime      IN CHAR,
                        p_ptlid       IN NUMBER,
                        p_latdeg      IN NUMBER,
                        p_londeg      IN NUMBER,
                        p_wsmps       IN NUMBER,
                        p_adjwsmps    IN NUMBER,
                        p_wddeg       IN NUMBER,
                        p_pmb         IN NUMBER,
                        p_stlid       IN NUMBER);

END OBSACTIONS;
/
CREATE OR REPLACE PACKAGE BODY OBSACTIONS AS
-- Воспользуемся частной (private) функцией FindCurrTime для
-- вычисления текущего времени. Этот метод доступен только
-- функциям и процедурам, указанным в разделе тела модуля OBSACTIONS,
-- и аналогичен частному методу Java.
    FUNCTION FindCurrTime RETURN CHAR IS
    v_time CHAR(8);
    BEGIN
        SELECT TO_CHAR(SYSDATE,'HH24MISS') INTO v_time FROM DUAL;
        RETURN v_time;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
           RETURN v_time;
    END FindCurrTime;

-- Получим новый Obs_id
    FUNCTION getObsId RETURN NUMBER IS
      p_obsid NUMBER;
    BEGIN
        SELECT OBSID_SEQ.NEXTVAL INTO p_obsid FROM DUAL;
        RETURN p_obsid;
    EXCEPTION
        WHEN OTHERS THEN
             RETURN p_obsid;
    End getObsId;

-- Создадим объект типа Oceanic_observation
-- Это частная (private) функция.
FUNCTION OceanicObservation (p_latdeg IN NUMBER,
                    p_londeg     IN NUMBER,
                    p_wsmps      IN NUMBER,
                    p_adjwsmps   IN NUMBER,
                    p_wddeg      IN NUMBER,
                    p_pmb        IN NUMBER)
                        RETURN oceanic_observation IS

--      Создадим тип oceanic_observation.
        v_obs oceanic_observation :=
             oceanic_observation(p_latdeg,p_londeg,
                p_wsmps,p_adjwsmps,p_wddeg,p_pmb);
    BEGIN
        RETURN v_obs;
    END OceanicObservation;
```

```
-- Добавим новое наблюдение
    PROCEDURE insertObs(p_newobsid IN NUMBER,
                        p_whent       IN VARCHAR2,
                        p_attime      IN CHAR,
                        p_ptlid       IN NUMBER,
                        p_latdeg      IN NUMBER,
                        p_londeg      IN NUMBER,
                        p_wsmps       IN NUMBER,
                        p_adjwsmps    IN NUMBER,
                        p_wddeg       IN NUMBER,
                        p_pmb         IN NUMBER,
                        p_stlid       IN NUMBER) IS

        v_date DATE := TO_DATE(p_whent,'DD-MON-YYYY');
        v_time CHAR(8) := p_attime;
        v_newobsid NUMBER := TO_NUMBER(p_newobsid);
        v_platformref REF PLATFORM_TYPE;

        v_obs Oceanic_Observation :=
         Oceanic_Observation(p_latdeg,p_londeg,
           p_wsmps,p_adjwsmps,p_wddeg,p_pmb);
    BEGIN
--      Проверим входную дату p_whent
        IF    v_date IS NULL THEN
              SELECT SYSDATE INTO v_date FROM DUAL;
        END IF;

--      Проверим входное время
        IF    v_time IS NULL THEN
              v_time := FindCurrTime;
        END IF;

--      Получим ссылку (REF) на platform_type
    SELECT REF(P) INTO v_platformref
              FROM platform_type_list P
              WHERE P.key_id = p_ptlid;

--      Вставим новый объект
        INSERT INTO OCEANIC_OBSERVATION_LIST
              VALUES(
                    v_newobsid, v_date, v_time, p_stlid, p_ptlid,
                    v_platformref, v_obs
                   );
        -- зафиксируем результат; (см. пояснение 1)
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
              NULL;
        WHEN OTHERS THEN
              NULL;
    END insertObs;

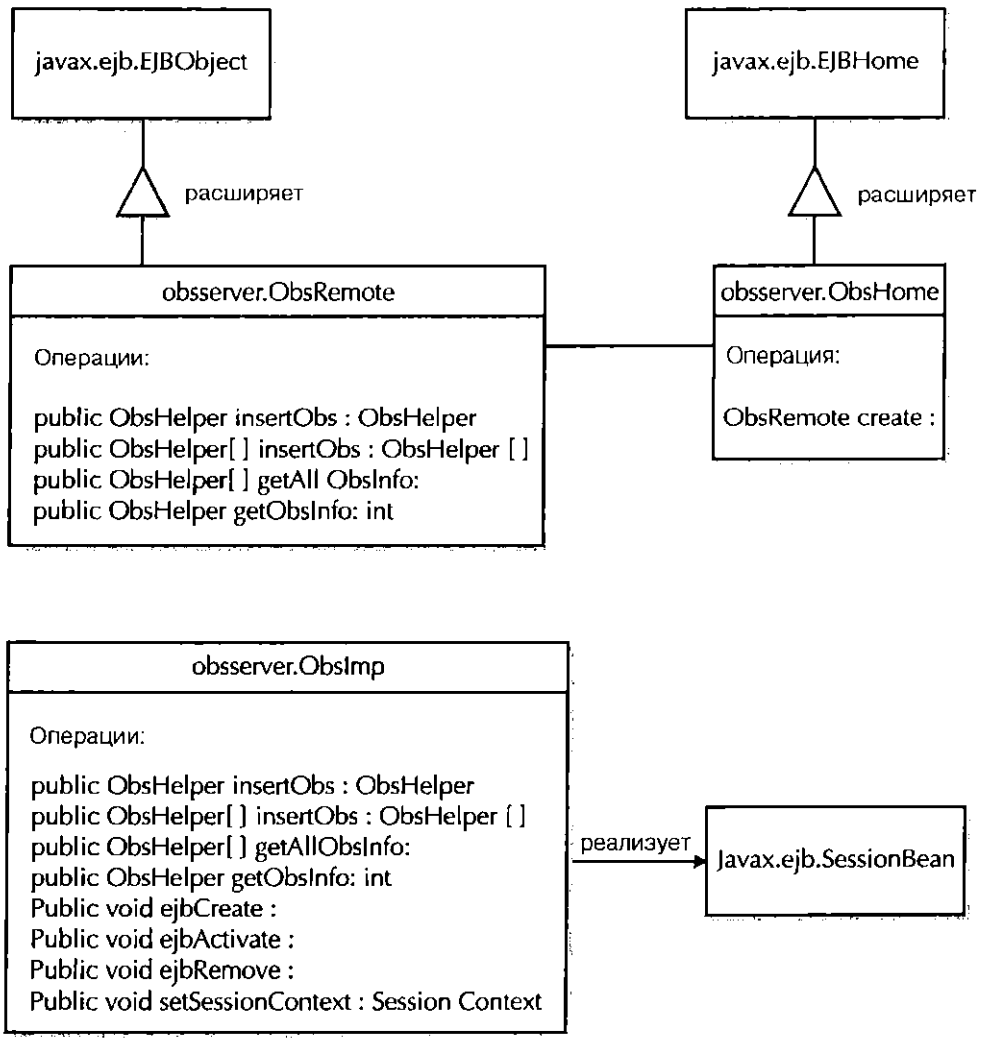
END OBSACTIONS;
/
```

Пояснение к PL/SQL-модулю OBSACTIONS:

1. Обратите внимание: операция явного завершения в модуле PL/SQL не нужна. EJB-контейнер сделает это сам в соответствии со значением атрибута транзакций, указанным в файле дескриптора внедрения для ObservationBean. В разделе "Файл дескриптора внедрения obs.ejb" данной главы используется значение атрибута транзакций TX_REQUIRED, обязывающее EJB-контейнер начинать новую транзакцию для каждого вызова метода. Напомним, что транзакция попытается выполнить операцию завершения перед окончанием вызова.

Второй шаг в разработке приложения – его анализ. На этом этапе создается общий вид приложения. Здесь для написания исходной программы используется Java. Следовательно, общий вид состоит из интерфейсов Java и Bean-класса, а также вспомогательных классов, включая их взаимоотношения (если есть) друг с другом. Общий вид сеансового зерна `ObservationBean` представлен на рис. 4.1.

Рис. 4.1. Общий вид сеансового зерна ObservationBean



Компонентами зерна `ObservationBean` являются:

- Класс `ObsHelper` – сериализуемый (serializable) Java, используемый зерном `ObservationBean` для прямого и обратного упорядочения объектов `ObsHelper` среди клиентских и серверных объектов.
- **Интерфейс** `ObsRemote`, **расширяющий интерфейс** `javax.ejb.EJBObject`.
- **Интерфейс** `ObsHome`, **расширяющий интерфейс** `javax.ejb.EJBHome`.
- Класс `ObsImp`, **реализующий** `javax.ejb.SessionBean`.
- Файл дескриптора внедрения `obs.ejb`.

На рис. 4.1 не показаны файл дескриптора внедрения и клиентское приложение, которое будет работать с зерном `ObservationBean`. Обычно в общий вид (высший уровень) компонента EJB эти компоненты не включаются. Напомним, однако, что файл дескриптора внедрения EJB – важный компонент модели EJB, и поэтому нужно создать его, а также клиентское приложение, работающее с `ObservationBean`. Исходный текст программ, представленных в

этой главе, содержится в файле Ch4ReaderCode.zip. Здесь присутствуют два варианта интерфейса ObsRemote, классов ObsImp и Chapter4Client. Первый вариант идентичен тому, что приведен в этой главе, а во втором вместо массива Java применяется Java-объект ArrayList. Второй вариант будет использоваться и в приложении, разрабатываемом в главе 5.

Как видим, сеансовое зерно ObservationBean состоит из двух пакетов Java: helpers и obsserver. Класс ObsHelper находится в пакете helpers, а удаленный интерфейс ObsRemote, домашний интерфейс ObsHome и Bean-класс ObsImp — в пакете obsserver. Удаленный интерфейс ObsRemote и Bean-класс ObsImp используют класс ObsHelper. Точнее класс ObsHelper используется в качестве входного параметра и/или возвращаемого типа метода, объявленного в удаленном интерфейсе и реализованного в Bean-классе.

Класс ObsHelper

Данный класс реализуется интерфейсом java.io.Serializable. Экземпляры класса будут использоваться в прямом и обратном упорядочении данных, которые необходимы Bean-классу, работающему с информацией, хранящейся в реляционных и объектно-реляционных таблицах Oracle. Опишем класс ObsHelper:

```
/*  Имя программы:   ObsHelper.java
**
**  Назначение:      Серийный Helper-класс Java для
**                   передачи объектов параметров
**                   (OCEANIC_OBSERVATION_TYPE)
**                   между клиентами, компонентами EJB и CORBA.
*/

package helpers;

import java.io.Serializable;
import java.math.BigDecimal;

public class ObsHelper
   implements java.io.Serializable {

    // Переменные для элементов
     public BigDecimal obs_id = null;
     public String when_t = null;
     public String at_time = null;
     public BigDecimal station_id = null;
     public BigDecimal produced_id = null;
     public BigDecimal latitude = null;
     public BigDecimal longitude = null;
     public BigDecimal wdspd = null;
     public BigDecimal adj_wdspd = null;
     public BigDecimal wddir = null;
     public BigDecimal pressure = null;
     // Опишем массив типа BigDecimal
     public BigDecimal[] idArray = null;

    // (см. пояснение 1)
       // Конструктор по умолчанию
       public ObsHelper() {

    }
    // Конструктор 1 с параметрами
      public ObsHelper (
        BigDecimal obs_id, String when_t, String at_time,
        BigDecimal station_id, BigDecimal produced_id,
        BigDecimal latitude, BigDecimal longitude,
    BigDecimal wdspd,    BigDecimal adj_wdspd,
        BigDecimal wddir,    BigDecimal pressure) {
```

```
            this.obs_id = obs_id;
            this.when_t = when_t;
            this.at_time = at_time;
            this.station_id = station_id;
            this.produced_id = produced_id;
            this.latitude = latitude;
            this.longitude = longitude;
            this.wdspd = wdspd;
            this.adj_wdspd = adj_wdspd;
            this.wddir = wddir;
            this.pressure = pressure;
        }
    // Конструктор 2 с параметрами
    // Используем его только для инициализации идентификаторов obs_id
       public ObsHelper (BigDecimal obs_id) {
         this.obs_id = obs_id;
        }
    // Конструктор 3 с параметрами
    // Используем его только для инициализации idArray
       public ObsHelper (BigDecimal[] arrayOfId) {

           if   ( arrayOfId.length > 0 ) {
                int loopVar = 0;
                idArray = new BigDecimal[arrayOfId.length];

                // Повторим
                while ( loopVar < arrayOfId.length ) {
                    idArray[loopVar++] = arrayOfId[loopVar++];
           }
        }
}
```

Пояснение к классу `ObsHelper`:

1. В этом классе содержится несколько конструкторов, позволяющих создать несколько объектов `ObsHelper` в соответствии с конкретными потребностями.

Удаленный интерфейс `ObsRemote`

Данный EJB-интерфейс `ObsRemote` описывает методы, которые могут вызываться клиентами, работающими с сеансовым зерном `ObservationBean`. В соответствии со спецификацией EJB удаленные интерфейсы EJB должны расширять интерфейс `java.ejb.EJBObject` и конструкции `throws` сигнатур всех методов должны содержать `java.rmi.RemoteException`. Помимо нее, в конструкциях `throws` описаний методов может присутствовать любая исключительная ситуация, специфичная для приложения. Обратите внимание: все бизнес-методы, объявленные в интерфейсе `ObsRemote`, обращаются к базе данных Oracle8*i*. Напомним, что в конструкциях `throws` всех методов Java, обращающихся к БД, должна содержаться `java.sql.SQLException`. Следовательно, в конструкциях `throws` методов `getNewObs()`, `insertObs()`, `getAllObs()` и `getObsInfo()` будут присутствовать как `RemoteException`, так и `SQLException`.

▼ ВНИМАНИЕ! *Бизнес-методы, определяемые в удаленном интерфейсе, должны в точности соответствовать своим телам, описываемым в Bean-классе.*

Отметим, что в представленном здесь варианте интерфейса `ObsRemote` в качестве входных параметров и возвращаемых типов используется Java-массив типов `ObsHelper`. В главе 5 вы увидите другой вариант, где задействован `ArrayList` типов `ObsHelper`. Вариант *ObservationBean* из главы 4 работает в базе данных Oracle8*i* версий 8.1.5, 8.1.6 и бета-версии 8.1.7, а вариант зерна из главы 5 — только в 8.1.6 и 8.1.7. Все дело в том, что последний вариант содержит Java-объект `ArrayList`, соответствующий JDK1.2x. Опишем интерфейс `ObsRemote`:

```
// Имя программы:    ObsRemote.java
// Название:         Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Описание:         Удаленный интерфейс EJB для зерна
//                   наблюдений.

package obsserver;

// импортируем helper-классы
import helpers.ObsHelper;
import java.util.Vector;

// Обязательные пакеты Java для всех зерен EJB.
import java.rmi.*;
import javax.ejb.*;

// Специфичная для приложения исключительная ситуация или
// SQLException для приложений баз данных.
import java.sql.SQLException;

public interface ObsRemote extends EJBObject {

   // Этот метод получает новый obs_id из БД.
   public ObsHelper getNewObs( )
         throws SQLException, RemoteException;

   // Воспользуемся объектом ObsHelper в качестве передающего параметра
   // для транспортировки данных, которые нужно ввести в БД.
   // Интерфейсы ObsRemote и ObsHome, а также Bean-класс находятся
   // в базе данных Oracle8i.
   public ObsHelper insertObs(ObsHelper obs)
         throws SQLException, RemoteException;

   // Переопределим метод insertObs для передачи массива объектов
   // ObsHelper в Bean-класс.
   public ObsHelper[] insertObs(ObsHelper[] obs)
         throws SQLException, RemoteException;

   // Получим сведения обо всех наблюдениях и возвратим информацию
   // в массив из объектов ObsHelper.
   public ObsHelper[] getAllObs()
         throws SQLException, RemoteException;

      // Получим сведения о наблюдении и возвратим объект ObsHelper
   // по идентификатору наблюдения.
   public ObsHelper getObsInfo(int obsid)
          throws SQLException, RemoteException;
}
```

Домашний интерфейс ObsHome

Домашний интерфейс EJB должен расширять интерфейс `javax.ejb.EJBHome`. Следовательно, интерфейс `ObsHome` также расширяет интерфейс `EJBHome`. В домашнем интерфейсе необходимо указать один или несколько методов `create()`. Напомним, что метод `create()` позволяет создавать новые экземпляры зерна или повторно использовать существующие, что зависит от типа описываемого зерна. Возвращаемым типом метода `create()` должен быть удаленный интерфейс EJB (в данном случае `ObsRemote`), и в его конструкции `throws` обязательны обе исключительные ситуации: и `CreateException`, и `RemoteException`. Опишем интерфейс `ObsHome`:

```
// Имя программы:    ObsHome.java
// Версия:
```

```
// Авторское право:  Copyright (c) 2000
// Описание:         Домашний интерфейс EJB для сеансового зерна
//                   наблюдений.
//

package obsserver;

import java.rmi.*;
import javax.ejb.*;

public interface ObsHome extends EJBHome {

// Создадим экземпляр интерфейса ObsRemote.
public ObsRemote create()
        throws CreateException, RemoteException;
}
```

Bean-класс ObsImp: реализация с помощью JDBC

Bean-класс ObsImp — это JDBC-версия реализации бизнес-методов. SQLJ-реализация данного класса приведена в конце этой главы. Bean-класс ObsImp определяет тела методов, в точности соответствующих тем, что были описаны в интерфейсе ObsRemote. Bean-класс, как того требует спецификация EJB, должен расширять интерфейс SessionBean, и класс ObsImp поступает так же. Последний иллюстрирует JDBC-реализацию и, устанавливая соединение, явным образом применяет серверный внутренний драйвер. Опишем класс ObsImp:

```
/* Имя программы:    ObsImp.java
** Версия:
** Описание:         Bean-класс EJB, позволяющий клиенту вводить
**                   объект ObsHelper, массив объектов ObsHelper и
**                   считывать массив объектов ObsHelper.
**                   Реализация, использующая JDBC.
*/

package obsserver;

// Импортируем helper-классы
import helpers.ObsHelper;

import java.rmi.RemoteException;
import javax.ejb.*;

// Импортируем специфичную для приложения исключительную ситуацию
import java.sql.SQLException;

import java.sql.*;
import oracle.sql.*;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.util.Vector;

// Этот оператор описывает Bean-класс, реализующий интерфейс
// javax.ejb.SessionBean, как указано в спецификациях EJB.
public class ObsImp implements SessionBean {

  // Этот метод, который могут вызывать клиенты, подключается к
  // БД и вызывает переопределяемый getNewObs(conn)
  // для получения нового obs_id.
  // Во-первых, он вызывает метод connectDb(),
  // устанавливая соединение с базой данных Oracle8i.
  // Во-вторых - вызывает переопределяемый метод
  // getNewObs(conn), используя объект conn (типа
  // java.sql.Connection) в качестве входного параметра метода.
  // При вызове getNewObs(conn) создает новый идентификационный
```

```
// номер наблюдения (obs_id).
public ObsHelper getNewObs()
     throws SQLException, RemoteException {

     Connection conn = null;
     try {
          // С помощью внутреннего драйвера JDBC подключимся
          // к базе данных
     conn = connectDb();
             // получим новый obs_id
          ObsHelper obs = getNewObs(conn);
             // Возвратим объект ObsHelper
          return obs;

     } // Конец try
     catch(SQLException e) {
             // установим новую SQLException
                throw e;
          } // Конец catch()
} // Конец getNewObs()

// Этот метод НЕ вызывается клиентами и поэтому не указан в
// интерфейсе ObsRemote. Он вызывает PL/SQL-функцию
// OBSACTIONS.GETOBSID(), чтобы создать новый obs_id.
// Для этого ему нужны методы объекта java.sql.Connection
// (см. пояснение 1)
public ObsHelper getNewObs(Connection conn)
     throws SQLException, RemoteException {

 // Создадим объект CallableStatement
    CallableStatement cstmt = null;
 try {
      // Подготовим String для вызова
        String sqlId =
         "{? = call OBSACTIONS.GETOBSID}";

     // Подготовим вызов, используя параметр conn
     cstmt = conn.prepareCall(sqlId);

     // Объявим, что ? - возвращаемое значение типа Integer
  cstmt.registerOutParameter (1, Types.INTEGER);

  // Получим новый obsid, выполнив запрос
  cstmt.execute();

  // Сохраним результат запроса в переменной anObsId
 BigDecimal anObsId = new BigDecimal(cstmt.getInt (1));

  // Возвратим объект ObsHelper, используя новый идентификатор
  return new ObsHelper(anObsId);

 } // Конец try
 catch (SQLException e) {
     throw e;
 } // Конец catch

// Выполним очистку. Наш оператор использует метод finally()
// для очистки и корректного выхода из него. Обратите
// внимание: после применения объект
// CallableStatement всегда необходимо закрывать. Этот метод
// гарантирует, что независимо от происходящего в нем объект
// CallableStatement будет всегда закрываться после обработки
     finally {
          if ( cstmt != null ) cstmt.close();
  } // Конец finally()
```

```
} // Конец getNewObs(conn)

 // Данный метод, который могут вызывать клиенты, получает номер
 // нового obs_id и добавляет новый OCEANIC_OBSERVATION_TYPE в

 // таблицу OCEANIC_OBSERVATION_LIST. Для этого он вызывает
 // переопределяемый метод insertObs(obs,conn). Напомним, что
 // описание последнего содержится в интерфейсе ObsRemote.

public ObsHelper insertObs(ObsHelper inObs)
     throws SQLException, RemoteException {

 ObsHelper obs = inObs;
 Connection conn = null;

     try {
          // Воспользуемся внутренним драйвером JDBC для подключения к БД
          conn = connectDb();

          // Получим новый obs_id в БД
          ObsHelper anObsid = getNewObs(conn);

          // Обновим входной obs_id
          obs.obs_id = anObsid.obs_id;

          // Вызовем метод insertObs(obs,conn), чтобы ввести новый
          // OCEANIC_OBSERVATION_TYPE в OCEANIC_OBSERVATION_LIST
          anObsid = insertObs(obs,conn);

          // Возвратим объект ObsHelper
         return new ObsHelper(anObsid.obs_id);

   } // Конец try
   catch (SQLException e) {
       throw e;
   } // Конец catch
 } // Конец insertObs(obs)

// Этот оператор описывает вспомогательный метод, который не могут
// вызывать клиенты и который вводит Oracle-тип
// OCEANIC_OBSERVATION_TYPE в таблицу OCEANIC_OBSERVATION_LIST. Для
// этого он вызывает PL/SQL-процедуру OBSACTIONS.INSERTOBS(...).
public ObsHelper insertObs(ObsHelper obs, Connection conn)
     throws SQLException, RemoteException {

     // Создадим объект CallableStatement
        CallableStatement cstmt = null;

     try {
          // Подготовим String для вызова PL/SQL-процедуры
           // OBSACTIONS.INSERTOBS() (см. пояснение 2)
             String sql =
              "{call OBSACTIONS.INSERTOBS(?,?,?,?,?,?,?,?,?,?,?)}";

             cstmt = conn.prepareCall(sql);

             // Установим входные параметры для PL/SQL-процедуры
             // OBSACTIONS.INSERTOBS().
             // Этот и десять последующих операторов используют
             // комбинацию методов setInt() и setString() объекта
             // CallableStatement для установки параметров процедуры
             // OBSACTIONS.INSERTOBS().
             cstmt.setInt(1, obs.obs_id.intValue());
             cstmt.setString(2, obs.when_t);
             cstmt.setString(3, obs.at_time);
             cstmt.setInt(4, obs.produced_id.intValue());
```

```
                cstmt.setInt(5, obs.latitude.intValue());
                cstmt.setInt(6, obs.longitude.intValue());
                cstmt.setInt(7, obs.wdspd.intValue());
                cstmt.setInt(8, obs.adj_wdspd.intValue());
                cstmt.setInt(9, obs.wddir.intValue());
                cstmt.setInt(10, obs.pressure.intValue());
                cstmt.setInt(11, obs.station_id.intValue());

                cstmt.execute();

                // Возвратим объект ObsHelper
                return new ObsHelper(obs.obs_id);
        } // Конец try
        catch (SQLException e) {
         throw e;
        } // Конец catch
        // Выполним очистку
        finally {
                if ( cstmt != null ) cstmt.close();
                // НЕ ЗАКРЫВАЙТЕ СОЕДИНЕНИЕ ВНУТРИ БАЗЫ ДАННЫХ
                // if ( conn != null ) conn.close();
        } // Конец finally()
    } // Конец insertObs(obs,conn)

// Введем массив из OCEANIC_OBSERVATION_TYPE

public ObsHelper[] insertObs(ObsHelper[] inObs)
        throws SQLException, RemoteException {

    int loopVar = 0;
    // Объявим массив из объектов ObsHelper
    ObsHelper[] returnArray = new ObsHelper[inObs.length];

    Connection conn = null;
    try {
        // Воспользуемся внутренним драйвером JDBC для подключения к БД
        conn = connectDb();

        // Повторим и сохраним массив идентификаторов obs_id в
        // returnArray[]
        // Этот оператор описывает while-цикл, в котором по очереди
        // вызываются getNewObs(conn) и insertObs(obs,conn)
        while  ( loopVar < inObs.length ) {
             // Получим новый obs_id на каждой итерации
             ObsHelper anObsid = getNewObs(conn);

             // Перенесем объект ObsHelper в obs
             ObsHelper obs = inObs[loopVar];

             // Заменим obs.obs_id новым anObsid.obs_id
             obs.obs_id = anObsid.obs_id;

             // Вызовем insertObs(..), чтобы добавить новую строку
             // в БД, и возвратим новый ObsHelper с новым obs_id.
             returnArray[loopVar] = insertObs ( obs, conn );

             loopVar++; // Увеличим счетчик
          } // Конец while

          // Возвратим массив идентификаторов
          return returnArray;
    } // Конец try
    catch (SQLException e) {
        throw e;
    } // Конец catch
```

```
} // Конец insertObs(ObsHelper[] inObs)

// Этот метод использует динамический SQL для получения всех данных
// из таблицы OCEANIC_OBSERVATION_LIST.
// Результаты запроса сохраняются в массиве Java.
// В главе 5 массив Java будет заменен Java-объектом ArrayList.
public ObsHelper[] getAllObs()
        throws SQLException, RemoteException {

       Connection conn = null;

     // Для сохранения строк таблицы необходим вектор (Vector) Java,
     // так как неизвестно, сколько строк считывается.
     Vector obsList = null;

     // Объявим Java-объект ResultSet object
     ResultSet rset = null;

     // Объявим переменную PreparedStatement. Обратите особое
     // внимание: для считывания списка всех наблюдений, хранящихся в
     // таблице OCEANIC_OBSERVATION_LIST, используется JDBC
     // (динамический SQL в программе Java).
     PreparedStatement pstmt = null;

     // Подготовим SQL-оператор SELECT
     String sql =
          "SELECT P.obs_id, TO_CHAR(P.when_C,'DD-MM-YYYY'),"
          + "P.at_time, P.station_id, P.produced_id, "
          + "P.obsobj.latitude_deg, P.obsobj.longitude_deg, "
          + "P.obsobj.windspeed_mps, P.obsobj.adj_windspeed_mps, "
          + "P.obsobj.wind_direction_deg, P.obsobj.pressure_mb "
          + "FROM OCEANIC_OBSERVATION_LIST P";

     try {
          // Воспользуемся внутренним драйвером JDBC для подключения к БД
          conn = connectDb();

          pstmt = conn.prepareStatement(sql);

          // Выполним запрос и сохраним результат
          // в Java-объекте ResultSet.
          rset = pstmt.executeQuery();

          // Создадим экземпляр вектора obsList
        obsList = new Vector();

          while (rset.next()) {
            ObsHelper oneObs = populateData(rset);
            // Введем объект oneObs в Java-вектор obsList
            obsList.addElement(oneObs);
     }  // Конец while

     // Получим размер вектора obsList
     int arraySize = obsList.size();

     // Создадим массив ObsHelper, используя
     // размер вектора obsList
     ObsHelper anArrayOfObsHelper[] = new ObsHelper[arraySize];

     // Скопируем всю информацию из вектора
     // в массив anArrayOfObsHelper
     obsList.copyInto(anArrayOfObsHelper);

     // Возвратим массив объектов ObsHelper вызывающему
     return anArrayOfObsHelper;
} // Конец try
```

```
 catch (SQLException e) {
     throw e;
 } // Конец catch
 catch (java.lang.Exception e) {
     throw new SQLException("ArrayList error in Select "
     + " Observations: " + e.getMessage());} // Конец catch

 // Выполним очистку
 finally {
   if ( rset != null ) rset.close();
    if ( pstmt != null ) pstmt.close();
 }
} // Конец getAllObs()

// Получим сведения о наблюдении для конкретного obs_id.
// Этот оператор описывает метод, который считывает информацию
// конкретной строки таблицы с указанным идентификатором наблюдения.
public ObsHelper getObsInfo(int obsid)
          throws SQLException, RemoteException {

     Connection conn = null;

     // Объявим Java-объект ResulSet
     ResultSet rset = null;

     // Объявим переменную PreparedStatement
     PreparedStatement pstmt = null;

     // Подготовим SQL-оператор SELECT.
     // Обратите внимание на использование JDBC (динамического SQL
     // в программе Java) для выполнения запроса.
     String sql =
          "SELECT P.obs_id, TO_CHAR(P.when_t,'DD-MM-YYYY'),"
          + "P.at_time, P.station_id, P.produced_id, "
          + "P.obsobj.latitude_deg, P.obsobj.longitude_deg, "
          + "P.obsobj.windspeed_mps, P.obsobj.adj_windspeed_mps, "
          + "P.obsobj.wind_direction_deg, P.obsobj.pressure_mb "
          + "FROM OCEANIC_OBSERVATION_LIST P "
          + "WHERE P.obs_id = ?";

     try {
          conn = connectDb();

          pstmt = conn.prepareStatement(sql);

          pstmt.setInt(1, obsid);
          rset = pstmt.executeQuery();

          ObsHelper oneObs = null;
          while (rset.next()) {
             oneObs = populateData (rset);
          }
          // Возвратим объект ObsHelper
             return oneObs;

     } // Конец try
     catch (SQLException e) {
          throw e;
 }   // Конец catch

     // Выполним очистку
     finally {
          if ( rset != null ) rset.close();
          if ( pstmt != null ) pstmt.close();
     } // Конец finally
```

```
} // Конец getObsInfo()

// Этот метод заполняет объект ObsHelper данными
// результирующего набора.
public ObsHelper populateData (ResultSet rset)
 throws SQLException {

 try {
     // Объявим объект ObsHelper с помощью конструктора по умолчанию.
     ObsHelper oneObs = new ObsHelper();

     // Установим переменные для элементов объекта ObsHelper.
     // Заметим, что можно было бы воспользоваться конструктором 1
     // с параметрами, описанным в классе ObsHelper.

     oneobs.obs_id    = new BigDecimal(rset.getInt(1));
     oneobs.when_t    = rset.getString(2);
     oneobs.at_time   = rset.getString(3);

     rset.getInt(4);
     if  (rset.wasNull())
          oneObs.station_id = null;
     else
          oneObs.station_id = new BigDecimal(rset.getInt(4));
     rset.getInt(5);
     if  (rset.wasNull())
          oneObs.produced_id = null;
     else
          oneObs,produced_id = new BigDecimal(rset.getInt(5));
     rset.getInt(6);
     if  (rset.wasNull())
          oneObs.latitude = null;
     else
          oneObs.latitude = new BigDecimal(rset.getInt(6));
     rset.getInt(7);
     if  (rset.wasNull())
          oneObs.longitude = null;
     else
          oneObs.longitude = new BigDecimal(rset.getInt(7));
     rset.getInt(8);
     if  (rset.wasNull())
          oneObs.wdspd = null;
     else
          oneObs.wdspd = new BigDecimal(rset.getInt(8));
     rset.getInt(9);
     if  (rset.wasNull())
          oneObs.adj_wdspd = null;
     else
          oneObs.adj_wdspd = new BigDecimal(rset.getInt(9));
     rset.getInt(10);
     If  (rset.wasNull())
          oneObs.wddir = null;
     else
          oneObs.wddir = new BigDecimal(rset.getInt(10));
     rset.getInt(11);
     if  (rset.wasNull())
          oneObs.pressure = null;
     else
          oneObs.pressure = new BigDecimal(rset.getInt(11));
     return oneObs;
} // Конец try
catch (SQLException e) {
     throw e;
}   // Конец catch
```

```
    catch (java.lang.NullPointerException e) { -
        throw new java.lang.NullPointerException
          (e.getMessage());
    } // Конец catch
  } // Конец populateData()

  // Этот метод вызывается всеми методами, которым нужно подключаться
  // к БД при помощи внутреннего драйвера JDBC.
  public static Connection connectDb()
     throws SQLException, RemoteException {

     // Обратите внимание: этот метод доступен только методам
     // Bean-класса, но НЕ клиентам.
     // Создадим переменную java.sql.Connection.
     Connection conn = null;

     // Создадим ее экземпляр с помощью внутреннего JDBC-драйвера Oracle8i.
     conn =
       DriverManager.getConnection("jdbc:default:connection:");

     // Возвратим объект java.sql.Connection.
     return conn;

  } // Конец connectDb()

  // Этот оператор и пять последующих методов описывают обязательные
  // методы зерна уровня предприятия.
  // Обратите внимание: метод ejbCreate() можно создать таким, чтобы
  // он самостоятельно получал объект java.sql.Connection.
  // Напомним, что метод ejbCreate() автоматически вызывается при
  // вызове клиентом метода create() домашнего интерфейса EJB. Если
  // подключить логику установления соединения с БД, это
  // соединение будет существовать в течение времени действия зерна.
  // Отметим также, что здесь можно выполнить еще и подготовительные
  // операторы; при этом операторы компилируются один раз, а не при
  // каждом вызове методов.
  public void ejbCreate()
     throws RemoteException, CreateException {
  } // Конец ejbCreate()

  public void ejbActivate()
    throws RemoteException {
  } // Конец ejbActivate()

  public void ejbPassivate()
    throws RemoteException {
  } // Конец ejbPassivate()
  public void ejbRemove()
     throws RemoteException {
  } // Конец ejbRemove()

  public void setSessionContext(SessionContext ctx)
     throws RemoteException {
  } // Конец setSessionContext()

} // Конец класса ObsImp
```

Пояснения к классу `ObsImp`:

1. **Этот оператор описывает бизнес-метод, который не могут вызывать клиенты. Заметим, что последний можно объявить как частный (private) или защищенный (protected). Перед вызовом процедуры или функции PL/SQL в программе Java необходимо установить JDBC (см. "Oracle8*i* JDBC Developer's Guide, Release 8.1.5 and higher"). Короче говоря, нужно:**

- Подключиться к базе данных и получить объект java.sql.Connection. Отметим, что этот метод не соединяется с БД, а использует параметр conn, передаваемый ему методом вызывающего.
- Создать переменную CallableStatement.
- Воспользоваться методом prepareCall() объекта Connection.
- С помощью методов setXXXX объекта CallableStatement установить входные и/или выходные параметры для функции или процедуры PL/SQL (если нужно).
- С помощью метода execute() объекта CallableStatement выполнить запрос.
- С помощью методов getXXXX объекта CallableStatement считать результаты запроса.

2. Этот оператор вызывает PL/SQL-процедуру OBSACTIONS.INSERTOBS(), чтобы добавить новую строку в таблицу OCEANIC_OBSERVATION_LIST. OBSACTIONS.INSERTOBS() – это процедура, реализующая логику ввода информации в БД. Отметим, что установка JDBC для вызова процедуры PL/SQL аналогична установке для вызова функции PL/SQL.

Файл дескриптора внедрения obs.ejb

Теперь на примере двух файлов дескрипторов внедрения покажем, как организовывать управление доступом и транзакциями на уровнях зерна и метода. Для использования обоих файлов необходимо внедрить зерно ObservationBean дважды. Заметим, что отдельные компоненты сеансового зерна ObservationBean не нужно ни модифицировать, ни перекомпилировать – их нужно просто повторно внедрить. Чтобы изменить политику транзакций или доступа зерен, необходимо лишь изменить или обновить файл дескриптора внедрения и повторно развернуть зерна.

▼ ВНИМАНИЕ!

Формат, используемый для упаковки зерен EJB, описан в спецификации EJB. Обратите внимание: в EJB 1.1 способ написания файлов дескрипторов внедрения изменился по сравнению с EJB 1.0. Дескрипторы внедрения для серверов EJB 1.1 необходимо описывать на расширяемом языке разметки (XML, Extensible Markup Language). В новой версии Oracle8i (8.1.7) файл дескриптора внедрения будет описываться на XML, как того требует спецификация EJB 1.1. В этой книге используется собственное представление дескрипторов Oracle, поскольку оно поддерживает оба контейнера – и EJB 1.0, и EJB 1.1.

Опишем файл дескриптора внедрения Obs.ejb:

```
// Имя файла :   Obs.ejb
// Назначение:   Файл дескриптора внедрения для сеансового зерна
//               наблюдений.

SessionBean obsserver.ObsImp {

    // (см. пояснение 1)
    BeanHomeName = "test/ObservationBean";
    HomeInterfaceClassName = obsserver.ObsHome;
    RemoteInterfaceClassName = obsserver.ObsRemote;

    SessionTimeout = 0;
    // (см. пояснение 2)
    AllowedIdentities = { SYSDBA };
    TransactionAttribute = TX_REQUIRED;
    RunAsMode = CLIENT_IDENTITY;
```

Пояснения к файлу obs.ejb:

1. Этот и два последующих оператора описывают test/ObservationBean (пространство имен и имя опубликованного объекта), имена домашнего и удаленного интерфейсов.
2. Этот и следующий операторы объявляют управление соответственно доступом и транзакциями на уровне зерна. Напомним, что при указании на уровне зерна атрибуты идентификации и транзакций применяются по отношению ко всем его методам.

Файл obs.jar

Теперь, когда сеансовое зерно ObservationBean создано, надо с помощью файла SetupAndRun скомпилировать все компоненты и создать файл obs.jar. В состав файла obs.jar должны входить файлы классов:

- helpers/ObsHelper.class
- obsserver/ObsRemote.class
- obsserver/ObsHome.class
- obsserver/ObsImp.class

Внедрение сеансового зерна ObservationBean

В приглашении установим CLASSPATH и внедрим зерно уровня предприятия:

```
// приглашение NT
deployejb -republish -keep -temp temp -u scott -p tiger
    -s %ORACLE_SERVICE% -verbose
    -descriptor Obs1.ejb -generated ObsClient.jar Obs.jar

// приглашение UNIX
deployejb -republish -keep -temp temp -u scott -p tiger \
    -s $ORACLE_SERVICE -verbose \
    -descriptor Obs1.ejb -generated ObsClient.jar Obs.jar
```

Создание клиентского приложения

Наше простое клиентское приложение будет:

- Устанавливать переменные среды JNDI
- Находить ссылку на интерфейс ObsHome
- Создавать экземпляр ObsRemote
- С помощью экземпляра ObsRemote вызывать по порядку все методы, описанные в интерфейсе ObsRemote и реализованные в Beап-классе ObsImp

В главах 8 и 12 будет показано, как создавать клиенты CORBA и Java Server Pages (JSP), использующих сеансовые зерна. Опишем приложение Chapter4Client:

```
/* Имя программы: Chapter4Client.java
** Версия:
** Описание:       EJB-клиент: реализация JDBC
**
*/
// Импортируем классы JDBC
import java.sql.*;

// Импортируем helper-классы
import helpers.ObsHelper;

import obsserver.ObsRemote;
import obsserver.ObsHome;

// Импортируем класс ServiceCtx из пакета Oracle
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
```

```
// Импортируем Java-классы, необходимые для использования JNDI
import javax.naming.Context;
import javax.naming.InitialContext;

// Импортируем Java-классы, специфичные для приложения
import java.util.Hashtable;

import java.util.*;
import java.math.*;

public class Chapter4Client {    public static void main (String args
[])
    throws Exception {

    // Проверим входной параметр
    if (args.length != 4) {
        System.out.println("usage: Client "
                +"serviceURL objectName user password");
        System.exit(1);
    }   // Конец if

    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];
    Hashtable env = new Hashtable();
    // Установим среду JNDI, чтобы клиент мог найти
    // объект EJB
    env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, password);
    env.put(Context.SECURITY_AUTHENTICATION,
                    ServiceCtx.NON_SSL_LOGIN);

    // Создадим экземпляр класса Context
    Context ic = new InitialContext (env);


    ObsHome myObsHome = null;


    ObsRemote myObservationBean = null;

    try {
        // Найдем объект EJB с помощью
        // домашнего интерфейса
        myObsHome =
             (ObsHome)ic.lookup (serviceURL + objectName);
        System.out.println("Home Lookup Properly");

        // Создадим экземпляр зерна
        myObservationBean = myObsHome.create();
        System.out.println("Bean Created Properly");

        getAllObsInfo (myObservationBean);
        System.out.println("getAllObsInfo done");

        getOneObsInfo (myObservationBean);
        System.out.println("getOneObsInfo done");

        insertAnArrayOfObs (myObservationBean);
        System.out.println("insertAnArrayOfObs done");
    } // Конец try
    catch ( Exception ex ) {
        System.out.println("Cannot locate"
                +" or create object");
```

```
            System.err.println("error : " + ex);
            System.exit(1);
        } // Конец catch
    } // Конец main

    public void getAllObsInfo (ObsRemote myObservationBean)
        throws Exception {

       try {
            // Вызовем метод getAllObs(), реализованный в классе ObsImp
            ObsHelper[] arrayOfObs = myObservationBean.getAllObs();

            int loopVar = 0;
            while ( loopVar < arrayOfObs.length ) {
                    ObsHelper obs = arrayOfObs[loopVar];

                    System.out.println ("Inside JdbcProcClient "
                     + "obs.obs_id: " +obs.obs_id);

                    loopVar++;

            }
        } // Конец try
        catch (SQLException e) {
              System.err.println( e.getMessage() );
              System.exit(1);
        } // Конец catch
    } // Конец getAllObsInfo()

    /* Тестирование getObsInfo() */
    public static void getOneObsInfo (ObsRemote myObservationBean)
        throws Exception {
       try {
            int obs_id = 2;
            // Вызовем метод Bean-класса
            ObsHelper obs = myObservationBean.getObsInfo(obs_id);

            System.out.println ("Inside JdbcProcClient obs.obs_id: "
                            +obs.obs_id);
            System.out.println("when_t: " +obs.when_t);
        }   // Конец try
        catch (SQLException e) {
            System.err.println( e.getMessage() );
            System.exit(1);
        } // Конец catch
    } // Конец getOneObsInfo()

    /* Тестирование insertObs(ObsHerper[] obs) */
    public static void insertAnArrayOfObs (ObsRemote myObservationBean)
        throws Exception {
        try {
            int loopVar = 0;

            while (loopVar++ < 2) {
                 BigDecimal obs_id = null;
                 String when_t = "04-JUL-2000";
                 String at_time = "085500";
                 BigDecimal station_id = new BigDecimal(0);
                 BigDecimal produced_id = new BigDecimal(1);
                 BigDecimal latitude = new BigDecimal(0);
                 BigDecimal longitude = new BigDecimal(0);
                 BigDecimal wdspd = new BigDecimal(0);
                 BigDecimal adj_wdspd = new BigDecimal(0);
                 BigDecimal wddir = new BigDecimal(0);
                 BigDecimal pressure = new BigDecimal(0);
```

```
                ObsHelper obs1 =
                   new ObsHelper(obs_id,when_t,at_time,
                       station_id,produced_id,latitude,
                       longitude,wdspd,adj_wdspd,wddir,pressure);

                ObsHelper obs2 = myObservationBean.insertObs(obs1);
                System.out.println ("Inside JdbcProcClient " +
                       "obs2.obs_id: " +obs2.obs_id);
              }
        } // Конец try
        catch (SQLException e) {
              System.err.println( e.getMessage() );
              System.exit(1);
         } // Конец catch
       } // Конец insertAnArrayOfObs()
} // Конец класса Chapter4Client
```

Bean-класс ObsImp: реализация с помощью SQLJ

Обратите внимание: из Bean-классов можно использовать или ObsImp.java, или ObsImp.sqlj, но не оба сразу. Главное, чтобы до внедрения реализации одного из них с помощью Oracle-инструмента dropjava было удалено внедренное ранее зерно с тем же именем. Параметр -republish Oracle-инструмента deployejb не удаляет ненужные зависимые классы.

Реализация при помощи SQLJ очень похожа на реализацию посредством JDBC. В первом случае вводится совсем немного новых понятий, требующих детального обсуждения за исключением, пожалуй, итератора SQLJ. Приведем здесь краткое описание итератора SQLJ; за более подробными сведениями обращайтесь к "Oracle8*i* SQLJ Programming".

Итератор SQLJ

Итератор (iterator) SQLJ – это строго определенный вариант встраиваемого курсора SQL, используемый для приема выходных данных оператора SELECT. Итераторы бывают двух категорий: именованные и позиционные.

Объявление класса итератора Первый этап обработки данных с помощью итератора – описание класса итератора в объявлении SQLJ. В объявлении класса итератора указывается класс Java, который SQLJ строит самостоятельно. Транслятор SQLJ заменяет объявление итератора SQLJ на Java-объявление класса с тем же именем, что и итератор. Например, в SQLJ-программе ObsImp присутствует объявление именованного итератора:

```
#sql iterator ObsIter
            (java.lang.Integer obs_id,String whent,
            String attime,java.lang.Integer stat_id,
            java.lang.Integer prod_id,java.lang.Float lat_deg,
            java.lang.Float lon_deg,java.lang.Float wdspd_s,
            java.lang.Float adjwdspd,java.lang.Float wddir_s,
            java.lang.Integer pres);
```

Экземпляр итератора ObsIter можно заполнить любым оператором SELECT, если его набор элементов списка выбора (т.е. выражений, следующих за ключевым словом SELECT) является подмножеством множества атрибутов ObsIter. Другими словами, экземпляр ObsIter можно заполнить любым оператором SELECT, список выбора которого состоит только из столбцов obs_id, whent, attime, stat_id, prod_id, lat_deg, lon_deg, wdspd_s, adjwdspd, wddir_s и pres.

Транслятор SQLJ превращает объявление SQLJ-итератора ObsIter в Java-объявление Java-класса ObsIter, который будет содержать:

- Метод next(), считывающий данные из итератора построчно
- Методы доступа obs_id(), whent(), attime(), stat_id(), prod_id(), lat_deg(), lon_deg(), wdspd_s(), adjwdspd(), wddir_s() и pres(), которые возвращают значения столбцов таблицы OCEANIC_OBSERVATION_LIST в обрабатываемой в тот момент строке

- Метод close(), дезактивирующий экземпляр итератора

Обработка с помощью именованного итератора

Подытожим этапы обработки информации при помощи именованного итератора:

- С помощью объявления SQLJ опишем класс итератора.
- Объявим экземпляр класса итератора.
- Заполним экземпляр итератора выходными данными оператора SELECT, совместимого с выполняемым оператором SQLJ.
- С помощью метода next() класса итератора считаем следующую строку экземпляра итератора.
- Извлечем значения столбцов из текущей строки итератора, применив методы доступа класса итератора.
- Деактивируем экземпляр итератора, вызвав метод close() класса итератора.

Вот описание SQLJ-программы ObsImp:

```
/* Имя программы:    ObsImp.sqlj
// Версия:
// Автор:            Ваше имя
** Описание :        Bean-класс EJB, позволяющий клиенту вводить
**                   объект ObsHelper, массив объектов ObsHelper
**                   и считывать массив объектов ObsHelper.
**                   Реализация, использующая SQLJ.
**
*/
package obsserver;

// Импортируем helper-классы
import helpers.ObsHelper;

import java.rmi.RemoteException;
import javax.ejb.*;

// Импортируем специфичную для приложения исключительную ситуацию
import java.sql.SQLException;

import java.sql.*;
import oracle.sql.*;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.math.*;
import java.util.Vector;

public class ObsImp implements SessionBean {

// Опишем итератор SQLJ
#sql iterator ObsIter
              (java.lang.Integer obs_id,String whent,
               String attime,java.lang.Integer stat_id,
               java.lang.Integer prod_id,java.lang.Float lat_deg,
               java.lang.Float lon_deg,java.lang.Float wdspd_s,
               java.lang.Float adjwdspd,java.lang.Float wddir_s,
               java.lang.Integer pres);

    // Метод, который могут вызывать клиенты, неявно
    // подключается к БД и вызывает getObsId()
    // из PL/SQL-модуля ObsActions, чтобы получить новый obs_id.
```

```
public ObsHelper getNewObs()
    throws SQLException, RemoteException {

    int returnId;

    // Вызовем getObsId() из PL/SQL-модуля ObsActions,
    // чтобы узнать новый obs_id
    try {

    // Этот оператор вызывает метод ObsActions.getObsId.
    // Обратите внимание на простоту объявления SQLJ
    // и на отсутствие явных объектов java.sql.Connection
    // и CallableStatement и необходимых для программ JDBC
    // вызывающих процедур или функций PL/SQL.
    // Вызовем функцию PL/SQL.
    #sql returnId = {VALUES ObsActions.getObsId};

    BigDecimal anObsId = new BigDecimal(returnId);

    // Возвратим объект ObsHelper, используя новый идентификатор.
    return new ObsHelper(anObsId);

    }  // Конец try
    catch (SQLException e) {
      throw e;
    } // Конец catch
    catch (java.lang.NullPointerException e) {
       throw new SQLException("Cannot generate new "
                   +"Observation Id " + e.getMessage());

    }
} // Конец getNewObs()

// Этот оператор объявляет методы insertObs(). Обратите
// внимание на отсутствие объекта Connection в описании метода.
// Этот метод получает новый obs_id и добавляет новый
// OCEANIC_OBSERVATION_TYPE в таблицу OCEANIC_OBSERVATION_LIST.
// Этот метод вызывают клиенты.

public ObsHelper insertObs (ObsHelper obs)
      throws SQLException, RemoteException {
    try {
         // Узнаем новый obs_id в базе данных
         ObsHelper anObsId = getNewObs();
         BigDecimal obs_id = anObsid.obs_id;
         String when_t = obs.when_t;
         String at_time = obs.at_time;
         BigDecimal produced_id = obs.produced_id;
         BigDecimal latitude = obs.latitude;
         BigDecimal longitude = obs.longitude;
         BigDecimal wdspd = obs.wdspd;
         BigDecimal adj_wdspd = obs.adj_wdspd;
         BigDecimal wddir = obs.wddir;
         BigDecimal pressure = obs.pressure;
         BigDecimal station_id = obs.station_id;

         // Этот оператор использует параметры IN для базовых
         // переменных Java в объявлении SQLJ. Базовые переменные
         // Java могут иметь и параметры OUT, сохраняющие данные
         // из функций и процедур PL/SQL в объявлении SQLJ. Этот оператор
         // вызывает PL/SQL-процедуру OBSACTIONS.INSERTOBS().
         #sql { call ObsActions.insertObs(
                :IN obs_id, :IN when_t, :IN at_time, :IN produced_id,
                :IN latitude, :IN longitude, :IN wdspd, :IN adj_
                 wdspd,
```

```
                :IN wddir, :IN pressure, :IN station_id )
              };

        // Возвратим объект ObsHelper
     return new ObsHelper(anObsid.obs_id);
    } // Конец try
    catch (SQLException e) {
       throw e;
    } // Конец catch
} // Конец insertObs(obs)

// Введем массив из OCEANIC_OBSERVATION_TYPE
public ObsHelper[] insertObs(ObsHelper[] inObs)
    throws SQLException, RemoteException {
  int loopVar = 0;

  // Объявим массив из объектов ObsHelper
  int arraySize = inObs.length;

  ObsHelper[] returnArray = new ObsHelper[arraySize];

 try {
     // Повторим и сохраним массив идентификаторов obs_id в returnArray[]
     while ( loopVar < arraySize ) {

          ObsHelper obs = inObs[loopVar];

        // Вызовем insertObs(..), чтобы добавить новую строку в БД

        returnArray[loopVar] = insertObs ( obs );
        loopVar++;
   } // Конец while

   // Возвратим массив идентификаторов
   return returnArray;

  } // Конец try
  catch (SQLException e) {
    throw e ;
  } // Конец catch

} // Конец insertObs(ObsHelper[] inObs)

// Этот метод использует динамический SQL для возврата всех данных
// из таблицы OCEANIC_OBSERVATION_LIST.
public ObsHelper[] getAllObs()
    throws SQLException, RemoteException {

    // Для сохранения строк таблицы необходим вектор (Vector) Java,
    // так как неизвестно, сколько строк считывается.
    Vector obsList = null;

    // Объявим переменную итератора SQLJ
    ObsIter anSqljRs = null;

    try {
         // Этот оператор вводит результаты запроса
         // в экземпляр итератора SQLJ
         #sql anSqljRs =
            { SELECT P.obs_id AS obs_id,
              TO_CHAR(P.when_C, 'DD-MM-YYYY') As whent,
              P.at_time AS attime, P.station_id AS stat_id,
              P.produced_id AS prod_id,
              P.obsobj.latitude_deg AS lat_deg,
              P.obsobj.longitude_deg AS lon_deg,
              P.obsobj.windspeed_mps AS wdspd_s,
```

```
                P.obsobj.adj_windspeed_mps AS adjwdspd,
                P.obsobj.wind_direction_deg AS wddir_s,
                P.obsobj.pressure_mb AS pres
                      FROM OCEANIC_OBSERVATION_LIST P
              };

              while (anSqljRs.next()) {
                // Объявим объект ObsHelper

            ObsHelper oneObs = populateData(anSqljRs);

            // Введем объект oneObs в Java-вектор obsList
            obsList.addElement(oneObs);

        } // Конец while

        // Получим размер obsList
        int arraySize = obsList.size();

        // Создадим массив ObsHelper, используя вектор obsList
        ObsHelper[] anArrayOfObsHelper[] = new ObsHelper[arraySize];

        // Скопируем всю информацию из вектора
        // в массив anArrayOfObsHelper
        obsList.copyInto(anArrayOfObsHelper);

        // Возвратим массив вызывающему
        return anArrayOfObsHelper;
      } // Конец try
      catch (SQLException e) {
          throw e;
     } // Конец catch
     catch (java.lang.Exception e) {
        throw new SQLException("ArrayList error in Select "
            + " Observations: " + e.getMessage());
     }  // Конец catch

     // Выполним очистку
     finally {
              if ( anSqljRs != null ) anSqljRs.close();
     } // Конец finally()

  }  // Конец getAllObs()

  // Получим сведения о наблюдении для конкретного obs_id.
  public ObsHelper getObsInfo(int obsid)
       throws SQLException, RemoteException {

      int obs_id = obsid;

      ObsIter anSqljRs = null;

      try {
           #sql anSqljRs =
              { SELECT P.obs_id AS obs_id,
                TO_CHAR(P.when_C, 'DD-MM-YYYY') As whent,
                P.at_time AS attime, P.station_id AS stat_id,
                P.produced_id AS prod_id,
                P.obsobj.latitude_deg AS lat_deg,
                P.obsobj.longitude_deg AS lon_deg,
                P.obsobj.windspeed_mps AS wdspd_s,
                P.obsobj.adj_windspeed_mps AS adjwdspd,
                P.obsobj.wind_direction_deg AS wddir_s,
                P.obsobj.pressure_mb AS pres
              FROM OCEANIC_OBSERVATION_LIST P
              WHERE P.obs_id = :obsid
```

```
    };
    return populateData (anSqljRs);

} // Конец try
catch (SQLException e) {
    throw e;
} // Конец catch
catch (java.lang.NullPointerException e) {
    throw new SQLException("Observation does not exist "
         +"for obs_id: " +obsid +" " + e.getMessage());
}

 // Выполним очистку:
 // Этот оператор закрывает экземпляр SQLJ-итератора anSqljRs.
 // Как и в случае с Java-экземпляром ResultSet, необходимо
 // закрыть экземпляр SQLJ-итератора anSqljRs.
 finally {
         if ( anSqljRs != null ) anSqljRs.close();
 } // Конец finally()

} // Конец getObsInfo()

// Этот метод заполняет объект ObsHelper данными
// результирующего набора.
Public ObsHelper populateData (ObsIter anSqljRs)
    throws SQLException {

    try {
         // Объявим объект ObsHelper с помощью конструктора по умолчанию.
         ObsHelper oneObs = new ObsHelper();

         // Воспользуемся методами доступа из экземпляра
         // SQLJ-итератора anSqljRs.
         // Установим переменные для элементов объекта ObsHelper.
         // Этот метод использует методы доступа для установки элементов
         // данных объекта ObsHelper.
         oneObs.obs_id =
                   new BigDecimal(anSqljRs.obs_id().intValue());
         oneObs.when_t = anSqljRs.whent();
         oneObs.at_time = anSqljRs.attime();

         if (anSqljRs.stat_id() == null)
             oneObs.station_id = null;
         else
             oneObs.station_id =
                new BigDecimal(anSqljRs.stat_id().intValue());
         if (anSqljRs.prod_id() == null)
             oneObs.produced_id = null;
         else
             oneObs.produced_id =
                new BigDecimal(anSqljRs.prod_id().intValue());
         if (anSqljRs.lat_deg() == null)
             oneObs.latitude = null;
         else
             oneObs.latitude =
                new BigDecimal(anSqljRs.lat_deg().intValue());
         if (anSqljRs.lon_deg() == null)
             oneObs.longitude = null;
         else
            oneObs.longitude =
                new BigDecimal(anSqljRs.lon_deg().intValue());
         if (anSqljRs.wdspd_s() == null)
             oneObs.wdspd = null;
         else
```

```
                    oneObs.wdspd =
                        new BigDecimal(anSqljRs.wdspd_s().intValue());
                if (anSqljRs.adjwdspd() == null)
                    oneObs.adj_wdspd = null;
                else
                    oneObs.adj_wdspd =
                        new BigDecimal(anSqljRs.adjwdspd().intValue());
                if (anSqljRs.wddir_s() == null)
                    oneObs.wddir = null;
                else
                   oneObs.wddir =
                        new BigDecimal(anSqljRs.wddir_s().intValue());
                if (anSqljRs.pres() == null)
                    oneObs.pressure = null;
                else
                    oneObs.pressure =
                        new BigDecimal(anSqljRs.pres().intValue());
                return oneObs;
            } // Конец try
            catch (SQLException e) {
                 throw e;
            } // Конец catch
            catch (java.lang.NullPointerException e) {
                 throw new java.lang.NullPointerException
                   (e.getMessage());
            }

        } // Конец populateData()

        public void ejbCreate()
            throws RemoteException, CreateException {
        } // Конец ejbCreate()

        public void ejbActivate()
            throws RemoteException {
        } // Конец ejbActivate()

        public void ejbPassivate()
            throws RemoteException {
        } // Конец ejbPassivate()

        public void ejbRemove()
            throws RemoteException {
        } // Конец ejbRemove()

        public void setSessionContext(SessionContext ctx)
            throws RemoteException {
        } // Конец setSessionContext()

    } // Конец класса ObsImp
```

Пояснение к классу `ObsImp` (реализация с помощью SQLJ):

Обратите внимание на простоту и элегантность реализации с помощью SQLJ по сравнению с JDBC. Заметьте также, что при использовании SQLJ на сервере не нужно подключаться к базе данных явным образом; т.е. не нужно явно инициализировать соединение, как это было в реализации с помощью JDBC. В серверных программах SQLJ соединение неявно. За дополнительной информацией об SQLJ обращайтесь к "Oracle8*i* SQLJ Programming".

Итоги

В данной главе говорилось о том, как описывать политику доступа и транзакций зерен уровня предприятия. Было разработано сеансовое зерно `ObservationBean` – компонентное приложение базы данных, обладающее набором бизнес-методов для работы с таблицей `OCEANIC_OBSERVATION_LIST`. Помимо этого, мы создали Bean-класс `ObsImp` с помощью JDBC, и SQLJ.

В главе 5 мы рассмотрим более сложные концепции EJB, с одновременной разработкой сеансового зерна, которое моделирует процесс наблюдения за атмосферой, подвергаемый контролю за качеством. Ученые Отдела исследования ураганов (HRD, Hurricane Research Division) Национального управления океана и атмосферы (NOAA, National Oceanic and Atmospheric Administration) применяют эти наблюдения на практике, изучая ветер в тропических циклонах на поверхности океана. Их изыскания помогают синоптикам Национального центра слежения за ураганами (NHC, National Hurricane Center) определять текущую силу шторма и ветра, который может нанести ущерб.

ГЛАВА 5

Разработка сеансовых зерен: углубленное изучение

В главе 4 говорилось об описании транзакций с точки зрения свойств ACID (атомарность, согласованность, изолированность и долговременность). Транзакции, соответствующие этим критериям, считаются надежными. Из этой главы вы узнаете о:

- Управлении зернами EJB в сеансах связи с БД
- Ограничении транзакций в программах явным образом
- Построении надежных транзакционных приложений с использованием клиентского ограничения транзакций

Управление зернами в сеансах связи с базой данных

Зерна EJB, функционирующие в EJB-контейнере Oracle8*i*, выполняются в рамках сеанса связи с БД. Клиент начинает новый серверный сеанс неявно при активации серверного объекта CORBA или EJB. Теперь нас интересует управление зернами уровня предприятия в сеансах связи с базой данных Oracle. О работе во время сеансов с объектами CORBA более подробно рассказано в главе 7.

Oracle8*i* JServer открывает сеанс БД, когда клиент вызывает JNDI-метод `lookup()`. Напомним, что он вызывает его для считывания ссылки на домашний интерфейс EJB. База данных дает возможность явно управлять запуском сеанса, либо из программы Java (клиентского объекта, находящегося вне БД), либо из серверного объекта EJB или CORBA. Помимо этого, она позволяет EJB-объектам определять длительность сеанса связи с ней, а также завершать сеанс.

Клиентский объект, начинающий новый сеанс

В клиентских программах, находящихся в некоей области сети, новый сеанс БД начинается при вызове клиентом JNDI-метода `lookup()`. Метод `lookup()` требует, чтобы входной параметр представлял собой комбинацию URL и имени опубликованного объекта. Чтобы начать новый сеанс связи с базой данных Oracle8*i*, необходимо указать URL `sess_iiop://localhost:port#:SID`, а также опубликованное имя нужного объекта. Обратите внимание: сеанс устанавливается во время нахождения ссылки на домашний объект зерна. Однако его можно начать и без обращения к конкретному объекту. Для этого нужно просто воспользоваться URL `sess_iiop://localhost:port#:SID`. Например, Oracle-инструменты `deployejb` и `sess_sh` начинают новый сеанс базы данных, не обращаясь ни к какому объекту (см. приложение F).

Напомним, что инструмент `deployejb` применяется для внедрения зерен на сервере данных Oracle8*i*, а `sess_sh` – для создания, просмотра и удаления пространств имен. Приведем синтаксис обоих:

```
// Oracle-инструмент deployejb начинает новый сеанс связи с базой
// данных Oracle8i, находящейся на data-i.com.
depioyejb-user scott/tiger -service sess_iiop://data-i.com:2481:ORCL

// Oracle-инструмент sess_sh начинает новый сеанс связи с БД.
sess_sh -user sys -password syspassword
   -service sess_iiop://data-i.com:2481:ORCL
```

В листинге 5.1 приведен программный фрагмент, начинающий новый сеанс базы данных и считывающий ссылку на домашний интерфейс `ObservationBean`:

Листинг 5.1. Автономный клиент, начинающий новый сеанс связи с базой данных

```
// Установим переменные среды
Hashtable env = new Hashtable();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, user);
env.put (Context.SECURITY_CREDENTIALS, password);
env.put (Context.SECURITY_AUTHENTICATION,
                              ServiceCtx.NON_SSL_LOGIN);

// Получим описатель для InitialContext
Context ic = new InitialContext (env);

// Начнем новый сеанс базы данных, одновременно считывая
// ссылку на домашний интерфейс ObservationBean
ObsHome observationHome =
    (ObsHome)ic.lookup ("sess_iiop://data-i.com:2481:ORCL"
                        + "/test/ObservationBean");
```

Серверный объект, начинающий новый сеанс

При вызове JNDI-метода `lookup()` в зернах уровня предприятия, размещенных в базе данных Oracle8*i*, тоже открывается новый сеанс. Для начала нового сеанса внутри БД нужно указать URL `sess_iiop://thisServer` вместе с опубликованным именем объекта. Например, в классе реализации `QcSetImp` сеансового зерна `QcSetBean` программный фрагмент, представленный в листинге 5.2, используется для начала нового сеанса базы с одновременным считыванием ссылки на домашний интерфейс `QcObsBean`:

Листинг 5.2. Зерно EJB, начинающее новый сеанс связи с БД

```
// Установим среду (см. пояснение 1)
Hashtable env = new Hashtable();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
// Создадим InitialContext
Context ic = new InitialContext(env);
// Сохраним опубликованное имя объекта
String objectName = new String("/test/QcObsBean");
// Создадим новый сеанс БД, одновременно считывая
// ссылку на домашний объект
// (см. пояснение 2)
QcObsHome anQcObsHome =
    (QcObsHome)ic.lookup("sess_iiop://this Server"
                         +objectName);
```

Пояснения к листингу 5.2:

1. Обратите внимание: для серверных объектов, начинающих новый сеанс в границах открытого ранее, необходимо устанавливать только переменную среды `URL_PKG_PREFIXES`, так как другие переменные среды (`SECURITY_PRINCIPAL`, `SECURITY_CREDENTIALS` и `SECURITY_AUTHENTICATION`) уже были установлены. Сеансовые зерна `QcSetBean` и `QcObsBean` будут разработаны в разделе "Построение транзакционного приложения" этой главы.
2. Этот программный фрагмент нужно применять только при установлении нового сеанса связи с БД. Зерна EJB, выполняющиеся в базе данных Oracle8*i*, могут вызывать другие зерна в том же сеансе. Для повышения производительности системы избегайте создания новых сеансов базы данных. Более подробно об активации серверных объектов в одном сеансе рассказано в разделе "Серверный объект, обращающийся к тому же сеансу" этой главы.

После открытия сеанса базы данных сервером Oracle8*i* JServer к нему могут обращаться несколько отдельных клиентов, находящихся в сети, или существующий объект EJB. Работая с сеансом, существующее зерно EJB может вызывать методы другого зерна в границах того же сеанса.

Два клиента, обращающиеся к одному сеансу

Чтобы к одному и тому же сеансу базы данных Oracle8*i* могли обращаться два клиента, нужно:

1. В первом клиенте сохранить описатель (Handle) зерна.
2. Во втором клиенте считать описатель и воспользоваться им для обращения к экземпляру зерна.

Описатель зерна

В спецификации EJB описан интерфейс javax.ejb.Handle (см. листинг 5.3), позволяющий нескольким клиентам получать ссылку на объект EJB посредством объекта Handle. Handle (описатель) — это объект, который идентифицирует объект EJB. Клиентская программа может начать новый сеанс базы данных, сохранить Handle для объекта EJB в файле, а позже воспользоваться им, чтобы еще раз получить ссылку на объект EJB. Другие клиенты, работающие в том же сеансе, могут применить тот же описатель для обращения к объекту EJB. Класс реализации для интерфейса Handle предоставляется, как правило, EJB-контейнером и создается во время внедрения. Важно, чтобы класс описателя реализовывал интерфейс java.io.Serializable. Вот описание интерфейса Handle:

Листинг 5.3. Интерфейс javax.ejb.Handle

```
public interface javax.ejb.Handle{
  public abstract EJBObject getEJBObject();
}
```

Интерфейс Handle описывает метод getEJBObject(), позволяющий клиенту считывать ссылку на EJBObject (см. главу 3). Использование метода getEJBObject() для считывания ссылки на удаленный интерфейс зерна демонстрируется в листинге 5.6.

Напомним, что удаленные интерфейсы EJB должны расширять интерфейс javax.ejb.EJBObject. Более того, все вызовы, организуемые клиентами, проходят через объект EJBObject. Реализация методов, описанных в интерфейсе javax.ejb.EJBObject, также предоставляется EJB-контейнером и также генерируется на этапе внедрения. В качестве напоминания в листинге 5.4 еще раз приведено описание интерфейса javax.ejb.EJBObject.

Листинг 5.4. Интерфейс javax.ejb.EJBObject

```
Public interface javax.ejb.EJBObject
  extends java.rmi.Remote {
    public abstract EJBHome getEJBHome() throws RemoteException;
    public abstract Handle getHandle() throws RemoteException;
    public abstract Object getPrimaryKey() throws RemoteException;
    public abstract boolean isIdentical(EJBObject obj)
            throws RemoteException;
    public void remove()() throws RemoteException;
}
```

Метод getHandle() (см. листинг 5.4) возвращает Handle для объекта EJB. Следовательно, клиент, обладающий ссылкой на объект EJB, может получить Handle объекта, вызвав метод getHandle() по ссылке. Использование метода getHandle() иллюстрируется листингом 5.6. Обратите внимание на то, что описатели всех объектов EJB реализуют интерфейс Handle.

Сохранение описателя зерна

Для сохранения Handle зерна в файле воспользуемся программным фрагментом, приведенным в листинге 5.5.

Листинг 5.5. Сохранение описателя зерна

```
// Получим ссылку на интерфейс ObsHome
ObsHome anObsHome =
   (ObsHome) ic.lookup("sess_iiop://data-i.com"+
              ":2481:ORCL/test/ObservationBean");
// Получим ссылку на интерфейс ObsRemote
ObsRemote anObsRemote = anObsHome.create();

// Предположим, что имя файла сохранено в переменной handlefile.
// Создадим выходной поток объектов из файлового потока
FileOutputStream fostream = new FileOutputStream(handlefile);
ObjectOutputStream ostream = new ObjectOutputStream(fostream);

// Получим описатель зерна с помощью getHandle()
// и сохраним его, записав в файл
ostream.writeObject(anObsRemote.getHandle());

// Необходимо очистить потоки. Для этого
// воспользуемся операторами:
ostream.flush ();
fostream.close ();
```

Считывание описателя зерна

В клиентских программах, выполняющихся в одном сеансе базы данных, можно считать описатель зерна, предварительно сохраненный в файле другим клиентом, воспользовавшись для этого программным фрагментом, приведенным в листинге 5.6. Для получения ссылки на удаленный интерфейс зерна применяйте метод getEJBObject() интерфейса javax.ejb.Handle:

Листинг 5.6. Считывание описателя и получение ссылки на удаленный интерфейс

```
// Предположим, что второму клиенту передается имя файла, где
// сохранен описатель зерна. Узнаем описатель, считав файл,
// имя которого хранится в переменной handlefile.
FileInputStream finstream = new FileInputStream (handlefile);
ObjectInputStream istream = new ObjectInputStream (finstream);
finstream.close ();

// С помощью метода readObject() класса istream обратимся к интерфейсу
// ObsRemote и создадим объект javax.ejb.Handle.
javax.ejb.Handle handle = (javax.ejb.Handle)istream.readObject();

// С помощью метода getEJBObject() класса Handle еще раз
// узнаем ссылку на интерфейс ObsRemote.
ObsRemote anObsRemote = (ObsRemote)handle.getEJBObject();

...
// С помощью удаленного интерфейса вызовем методы зерна.
anObsRemote.insertObs(anObservation);
```

Серверный объект, обращающийся к тому же сеансу

Зерна EJB могут вести поиск других серверных объектов в том же сеансе, где выполняются. Чтобы обнаружить и активировать другой опубликованный объект в том же сеансе, где функционирует серверный объект, нужно указать URL в сочетании с опубликованным именем объекта: sess_iiop://thisServer/:thisSession. Например:

```
// Создадим InitialContext
Hashtable env = new Hashtable();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext(env);
SomeObjectHome anObjectHome =
  (SomeObjectHome)ic.lookup("sess_iiop://thisServer/:thisSession"
                            +"/test/SomeObject");
```

Серверный объект, управляющий продолжительностью сеанса

Автономные клиентские программы Java не могут явным образом управлять продолжительностью сеанса базы данных Oracle, в отличие от серверных объектов. Для этого применяется метод `oracle.aurora.net.Presentation.sessionTimeout(int x)`. Метод `sessionTimeout()` принимает параметр типа `int` и является частью файла `aurora.zip`. Чтобы использовать первый, нужно набрать `(ORACLE_HOME)/javavm/lib/aurora.zip` в CLASSPATH. Отметим, что таймаут сеанса устанавливается в секундах. Следующий программный фрагмент устанавливает продолжительность сеанса:

```
// Создадим переменную
int aTimeValue = 30;
...
// С ее помощью установим продолжительность сеанса
oracle.aurora.net.Presentation.sessionTimeout(aTimeValue);
```

Серверный объект, заканчивающий сеанс

Для завершения сеанса связи с БД явным образом автономные клиентские программы Java пользуются методом `System.exit(int x)`, входной параметр которого с типом `int` содержит завершающее значение. Однако в серверных программах, находящихся в базе данных Oracle8*i*, выполнение метода `System.exit()` сеанс не закроет. Чтобы явно завершить сеанс БД, серверные программы должны пользоваться методом `oracle.aurora.vm.OracleRuntime.exitSession(int x)`. Как и метод `System.exit()`, метод `exitSession()` в качестве завершающего значения принимает параметр типа `int`. Например:

```
// Закончим сеанс базы данных внутри базы данных Oracle
oracle.aurora.vm.OracleRuntime.exitSession(1);
```

Ограничение транзакций

Поставщики зерен могут обязать EJB-контейнер неявно управлять всеми транзакциями, как это предусмотрено транзакционной политикой, описанной в файле дескриптора внедрения. Однако при желании можно предоставить клиентским программам Java (не использующим зерна) или зернам EJB возможность явно определять границы транзакций программным путем, через службу транзакций Java (JTS, Java Transaction Service). JTS позволяет приложениям более точно определять свойства транзакций, чем при использовании декларативных атрибутов транзакций.

Транзакции ограничиваются на *стороне клиента* (client-side demarcated) или на *стороне сервера* (server-side demarcated). Напомним, что ограниченной называют транзакцию с четко определенными точками начала и окончания.

Транзакция, ограниченная на стороне клиента (автономная программа Java)

В Oracle8*i* версий 8.1.5 и 8.1.6 для ограничения транзакций программным путем клиенты EJB могут пользоваться API службы транзакций Java (JTS, Java Transaction Service). В Oracle8*i* версии 8.1.7 клиенты EJB, ограничивающие транзакции, обязаны применять API транзакций Java (JTA, Java Transaction

API). В любом случае разработчики могут явно ограничивать транзакции в своих программах соответствующими методами begin() или commit(), которые предоставляются API транзакций, реализуемым базой данных Oracle8*i*. Для клиентских ограниченных транзакций с использованием JTS (Oracle8*i* версий 8.1.5 и 8.1.6) нужно включить в исходный программный текст пакет oracle.aurora.jts.client.AuroraTransactionService с помощью операторов import:

```
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.client.TS;
```

Например, в программе QcClient, разрабатываемой позже в этой главе, транзакции явно ограничиваются с помощью JTS:

```
// Ограничим и начнем транзакцию
// (Oracle8i версий 8.1.5 и 8.1.6)
TS.getTS ().getCurrent ().begin ();
...
// Найдем и создадим экземпляр(ы) одного или нескольких зерен.
// Вызовем методы зерна.

// Закончим транзакцию одним из следующих операторов:
TS.getTS ().getCurrent ().commit (true/false);
TS.getTS ().getCurrent ().rollback ();
```

Заметим, что создаваемая версия СУБД Oracle8*i* (8.1.7) будет согласована с EJB 1.1. Его спецификация требует, чтобы EJB-контейнер поддерживал ограничение транзакций на уровне приложения реализацией интерфейса javax.transaction.UserTransaction. Этот интерфейс станет заменой интерфейсу javax.jts.UserTransaction, который применяется EJB-контейнерами, согласующимися с EJB 1.0. API JTA унифицирует процесс определения границ транзакций программным путем для зерен EJB (зерен TX_BEAN_MANAGED) и автономных клиентских программ Java. За дополнительной информацией о JTA обращайтесь к разделу "Java Transaction API (JTA)" сайта java.sun.com/products/jta/index.html.

В следующем программном фрагменте показано, как применять JTA в клиентской программе EJB:

```
// Ограничим и начнем транзакцию
// (Oracle8i бета-версии 8.1.7)
// Сначала импортируем в программу пакет javax.transaction.*.
Import javax.transaction.*;

...
public class Client {
  public static void main ( String[] arg ) throws Exception {
    // 1. Выполним аутентификацию в базе данных, как это обычно
    // делается для любого клиента, и установим переменные среды
    Hashtable env = new Hashtable();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION,
                          ServiceCtx.NON_SSL_LOGIN);
    // Получим описатель для InitialContext
    Context ic = new InitialContext (env);

    // 2. Считаем объект UserTransaction из пространства имен JNDI
    UserTransaction ut;
    ut = (UserTransaction)ic.lookup(...);

    // 3. С помощью метода begin() объекта UserTransaction начнем транзакцию
    ut.begin();
```

```
    // Вызовем методы зерна
    ...
    // 4. С помощью метода commit() объекта UserTransaction
    // закончим транзакцию
    ut.commit() ;
    ...
  } // Конец main()

} // Конец класса Client
```

Заметим также, что в Oracle8*i* версии 8.1.7 будет поддерживаться JTS, но только для обеспечения обратной совместимости.

Транзакция, ограниченная на стороне сервера

Для клиентских и серверных объектов-источников транзакции ограничиваются по-разному. Поставщики зерен EJB могут позволить EJB-контейнеру самому неявно ограничить транзакцию либо с помощью интерфейса javax.jts.UserTransaction (Oracle8*i* версий 8.1.5 и 8.1.6) ограничить ее явным образом. Предпочтительнее ограничивать транзакции неявно, поскольку при этом вся ответственность возлагается на EJB-контейнер. Кроме того, неявные декларативные транзакции не нужно ни начинать, ни останавливать. Чтобы ограничить транзакции явным образом, зерна применяют интерфейс UserTransaction. Необходимое условие — чтобы значением атрибута транзакций было TX_BEAN_MANAGED.

В Oracle предлагается класс oracle.aurora.jts.client.AuroraTransactionService — реализация интерфейса UserTransaction, позволяющая разработчикам EJB явно ограничивать транзакции в своих программах. Для зерен TX_BEAN_MANAGED (Oracle8*i* версий 8.1.5 и 8.1.6) воспользуемся следующим программным фрагментом, явно ограничив транзакции в серверных объектах EJB:

```
// Включим оператор импорта
// (Oracle8.i версий 8.1.5 и 8.1.6):
import javax.jts.UserTransaction;

// С помощью объекта SessionContext начнем транзакцию
ctx.getUserTransaction ().begin ();

..
// Вызовем методы

// С помощью объекта SessionContext закончим транзакцию
ctx.getUserTransaction ().commit ();
```

В Oracle8*i* версии 8.1.7 транзакции явно ограничивались бы в зернах EJB такой же программной конструкцией. Для зерен TX_BEAN_MANAGED (8.1.7) воспользуемся следующим программным фрагментом, определив границы транзакций:

```
// Включим оператор импорта
// (Oracle8.1 версии 8.1.7):
import javax.transaction.UserTransaction;

...
// Считаем UserTransaction из контекста сеанса
UserTransaction ut;
ut = (UserTransaction)ic.lookup (...);

// С помощью метода begin() объекта UserTransaction
// начнем транзакцию
ut.begin();
...
// Вызовем методы зерна.
```

```
// С помощью метода commit() объекта UserTransaction
// закончим транзакцию
ut.commit();
```

Построение транзакционного приложения

Теперь мы разработаем транзакционное приложение, реализующее процесс наблюдения за атмосферой с проверкой качества наблюдений. Оно даст возможность нескольким компонентам EJB выполнять работу, являющуюся частью одной атомарной транзакции. В нем используется служба JTS, которая позволяет автономным клиентским программам Java и зернам EJB управлять границами своих транзакций. В этом приложении три обязательных зерна: `ObservationBean`, `QcObsBean` и `QcSetBean`. Первое было создано в главе 4 — построим остальные. В процессе разработки приложения раскрываются концепции ограничения клиентских транзакций и распространения контекста транзакций.

Еще раз о транзакциях в EJB

В модели EJB реализуется принцип двухфазного завершения: распространение контекста транзакций и их распределение. Однако производители EJB не обязаны реализовывать двухфазное завершение. Так, в базах данных Oracle8*i* (версий 8.1.5 и 8.1.6) реализуется только распространение контекста транзакций. В новой версии 8.1.7 будут поддерживаться как распространение контекста транзакций, так и распределение их.

Важной составляющей архитектуры EJB является область действия транзакции. В Enterprise JavaBeans область действия (scope) транзакции охватывает каждое зерно, участвующее в единице работы. Транзакция распространяется на него при вызове клиентом его методов. Таким образом, во время вызова вызываемые зерна включаются в область действия транзакции. При распространении транзакций следует учитывать два важных фактора — поток выполнения, в котором функционируют зерна, и транзакционные атрибуты EJB.

В главе 4 уже говорилось о значениях атрибутов транзакций (см. раздел "Еще раз о декларативных атрибутах транзакций"). В таблицах 5.1. и 5.2 представлены различные атрибуты транзакций и ситуации, в которых транзакция действует. В первой строке показано, что происходит при вызове клиентом методов зерна без контекста транзакции, а во второй — при вызове с контекстом. T1 и T2 обозначают две разных транзакции.

Таблица 5.1. *Транзакции в EJB*

Атрибут транзакций	Клиентская транзакция	Транзакция зерна
TX_NOT_SUPPORTED	Нет	Нет
TX_NOT_SUPPORTED	T1	Нет
TX_REQUIRED	Нет	T2
TX_REQUIRED	T1	T1
TX_SUPPORTS	Нет	Нет
TX_SUPPORTS	T1	T1
TX_NEW_REQUIRED	Нет	T2
TX_NEW_REQUIRED	T1	T2
TX_MANDATORY	Нет	Ошибка
TX_MANDATORY	T1	T1

Таблица 5.2. *Транзакции в зернах TX_BEAN_MANAGED*

Клиентская транзакция	Текущая транзакция экземпляра	Используемая транзакция
Нет	Нет	Нет
T1	Нет	Нет
Нет	T2	T2
T1	T2	T2

В таблице 5.2 представлены четыре возможных сценария:

1. В первом сценарии клиент и экземпляр не открывали контекста транзакций. Следовательно, метод зерна вызывается без контекста.
2. Во втором клиент открывает контекст транзакций, в отличие от экземпляра. Выполнение клиентской транзакции приостанавливается, и метод зерна вызывается без контекста.
3. В третьем клиент не открывает контекста транзакций, в отличие от экземпляра. Поэтому метод зерна вызывается в своем собственном контексте транзакций.
4. В четвертом и клиент, и экземпляр открывают контекст транзакций. Выполнение клиентской транзакции приостанавливается, метод вызывается в контексте транзакций зерна, и после выполнения метода возобновляется действие контекста транзакций клиента.

Обратите внимание на то, что транзакции TX_MANAGED_BEAN меняются в зависимости от типа сеансового зерна: STATELESS или STATEFUL. Так, зерно STATEFUL сохраняет открытую транзакцию во время вызова методов, а зерно STATELESS нет. В режиме открытой транзакции EJB-контейнер приостанавливает действие контекста зерна после выполнения каждого вызванного метода и возобновляет его при вызове следующего.

Теперь перейдем к построению транзакционного приложения, но сначала сформулируем требования, предъявляемые пользователями.

Требования пользователей

- Считать список наблюдений за атмосферой из базы данных Oracle.
- Проверить качество наблюдений за атмосферой и выбрать из них множество достоверных.
- Создать один новый и уникальный элемент в таблице QC_EVENT_LIST при условии, что он имеет Oracle-тип QUALITY_CONTROL_TYPE.
- Создать один новый и уникальный элемент в таблице PASSED_OBSERVATION_LIST, причем элемент должен состоять из атрибутов, связанных с конкретным QC_ID из таблицы QC_EVENT_LIST и множеством наблюдений за атмосферой, качество которых было проверено.
- Не разрешать приложению вводить данные в таблицу QC_EVENT_LIST, если пользователь не выбрал множество достоверных наблюдений или если данных о наблюдениях нет.
- Не разрешать приложению вводить данные в таблицу PASSED_OBSERVATION_LIST, если оно не смогло ввести данные в таблицу QC_EVENT_LIST.

Беря за основу приведенные выше требования, построим приложение, работающее с данными в границах одной базы данных Oracle. При этом важно учесть: транзакции будут ограничиваться на стороне клиента. Это гарантирует, что все изменения в БД будут реализовываться в рамках одной транзакции и все они будут либо фиксироваться, либо откатываться.

Приложение будет работать с данными двух таблиц Oracle: QC_EVENT_LIST и PASSED_OBSERVATION_LIST. Обе таблицы входят в состав научной схемы Observation (см. введение). В листинге 5.7 представлены типы и таблицы Oracle, используемые в QcSetBean и QcObsBean.

Листинг 5.7. Типы QUALITY_CONTROL_TYPE, PASSEDOBS и PASSEDOBSARRAY и таблицы QC_EVENT_LIST и PASSED_OBSERVATION_LIST

```
CREATE TYPE QUALITY_CONTROL_EVENT AS OBJECT(
   qc_id          NUMBER(8),
   when_t         DATE,
   at_time        CHAR(8),
   event_id       NUMBER(8),
   for_event      REF atmosevent,
   whom_id        NUMBER(6),
   by_whom        REF scientist);

CREATE TABLE QC_EVENT_LIST OF QUALITY_CONTROL_EVENT;

CREATE OR REPLACE TYPE PASSEDOBS AS OBJECT(
   obsid          NUMBER(8),
   when_t         DATE);

CREATE TYPE PASSEDOBSARRAY AS TABLE OF PASSEDOBS;

CREATE TABLE PASSED_OBSERVATION_LIST(
   qcid           NUMBER(8),
   when_t         DATE,
   at_time        CHAR(8),
   idobj          passedObsArray)
NESTED TABLE idobj STORE AS pobsid_list;
```

В зернах QcObsBean и QcSetBean используются процедуры и функции, PL/SQL-модуля QCACTIONS (см. введение).

В сеансовом зерне QcObsBean используются две процедуры PL/SQL:

- QCACTIONS.INSERTQCSETOBS(ArrayList qcObs) позволяет пользователю вводить идентификаторы obs_id наблюдений, прошедших проверку. Ее входным параметром является объект ArrayList из объектов QcObsHelper.
- QCACTIONS.GETQCOBS(int qc_id, OracleTypes.Cursor), считывает данные обо всех наблюдениях для конкретного идентификатора qc_id. Она принимает параметры qc_id и тип OracleTypes.Cursor и возвращает курсор, состоящий из множества данных о наблюдениях. Тип OracleTypes.Cursor используется для процедур и функций PL/SQL, которые возвращают Oracle-тип REFCURSOR. Результат получаем с помощью типа OracleResultSet, который аналогичен java.sql.ResultSet; он точно так же предоставляет метод next(), позволяющий считывать данные, получаемые из БД.

▼ ВНИМАНИЕ! *Тип данных OracleTypes.CURSOR специфичен только для JDBC-драйверов Oracle.*

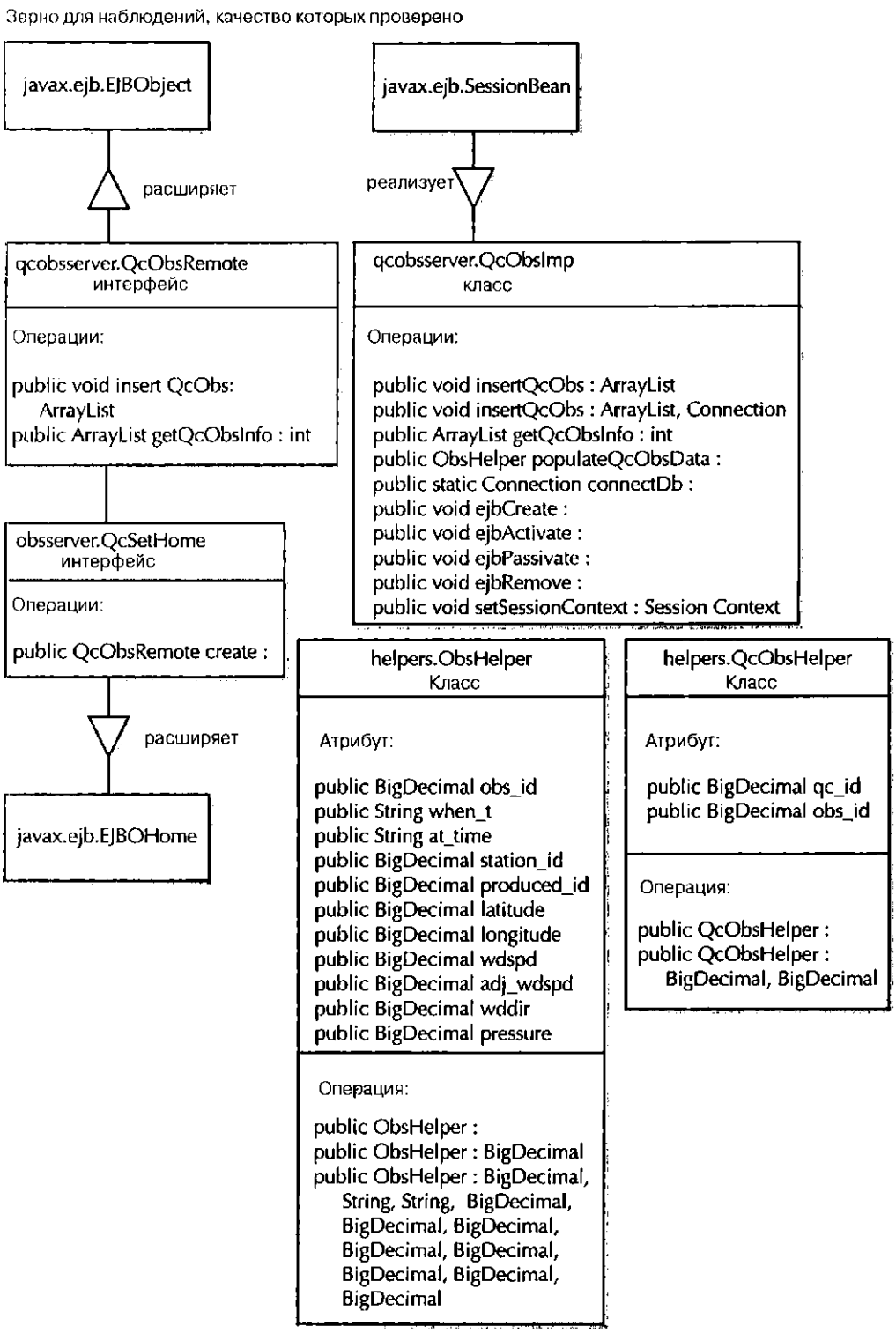
В сеансовом зерне QcSetBean используется одна функция и одна процедура PL/SQL:

- Функция QCACTIONS.GETNEWID() создает новый идентификатор QcSet.
- Процедура QCACTIONS.INSERTQCSET() вводит новую строку в таблицу QC_EVENT_LIST.

Общий вид зерен

На рис. 5.1, 5.2 и 5.3 показаны общий вид сеансовых зерен QcObsBean и QcSetBean, а также приложение QcClient. Общий вид сеансового зерна ObservationBean приведен на рис. 4.1 (глава 4).

Рис. 5.1. Общий вид зерна QcObsBeana



Заметим, что исходный текст всех программ, представленных в этой главе, содержится в файле Ch5ReaderCode.zip. Здесь находятся еще и реализации, в которых для всех разрабатываемых зерен используется SQLJ. Загрузить его можно с Web-сайта www.data-i.com или www.osborne.com.

Рис. 5.2.
Общий вид зерна QcSetBean

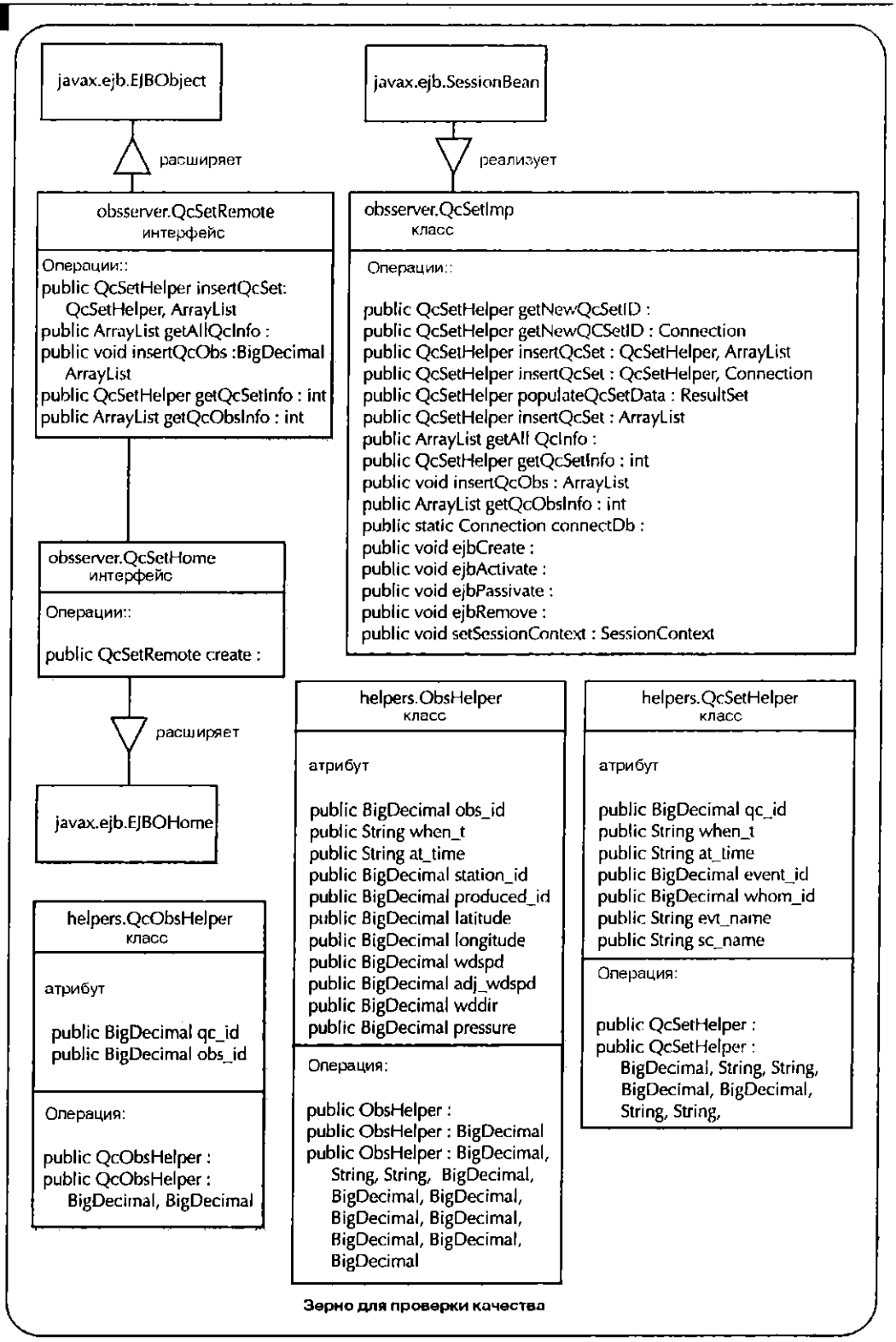
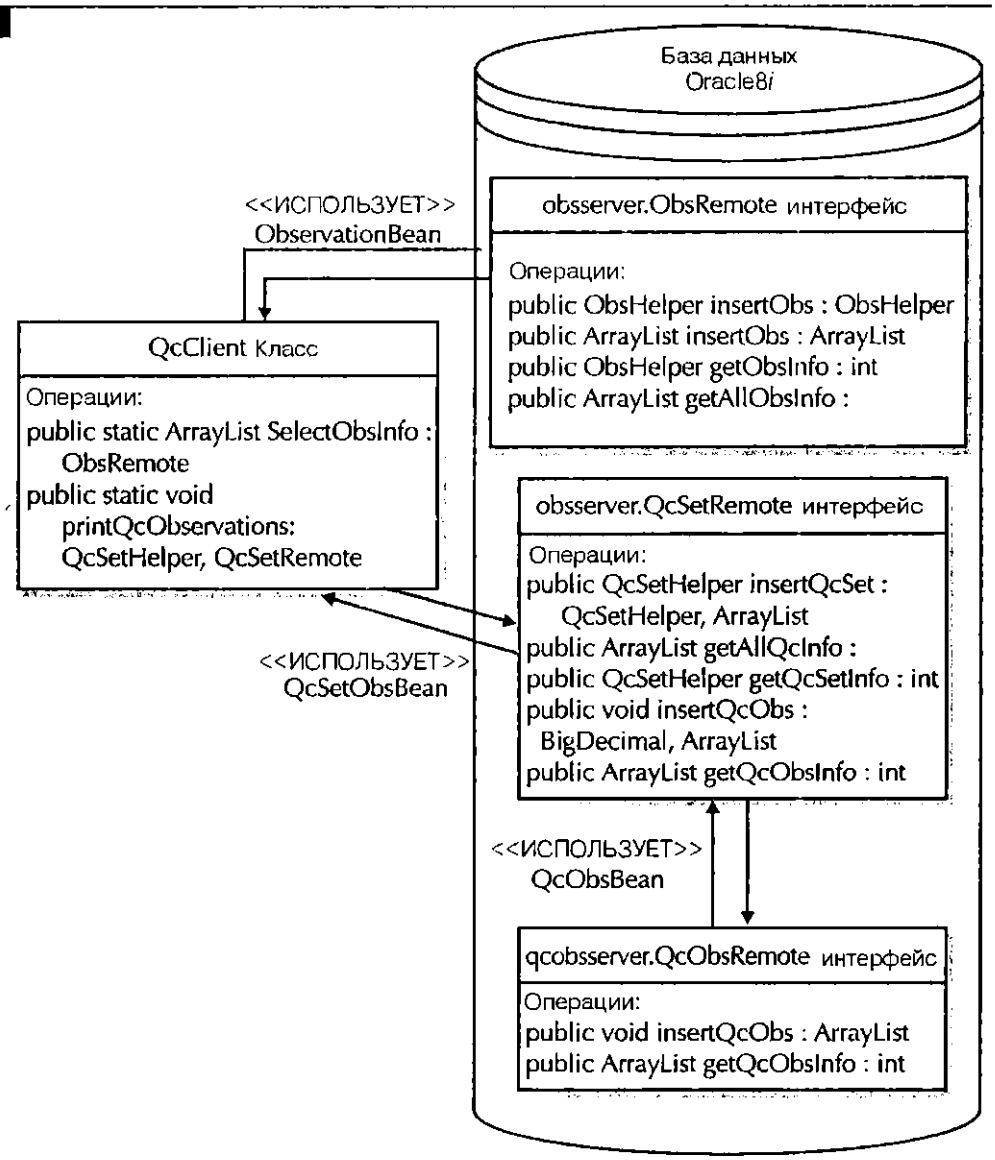


Рис. 5.3. Взаимодействие QcClient и зерен EJB



Общие классы: сериализуемые классы Java

`ObsHelper`, `QcObsHelper` и `QcSetHelper` используются для прямого и обратного упорядочения данных, пересылаемых из клиентских и серверных объектов и в них. Все три класса реализуют интерфейс `java.io.Serializable`. Понятие сериализуемости (последовательной передачи) очень важно при переносе объектов Java по сети, т.е. из одного адресного пространства в другое.

- ObsHelper — это класс-оболочка, отображающая атрибуты таблицы `OCEANIC_OBSERVATION_LIST`. Напомним, что этот класс был создан в главе 4. Данный класс используется сеансовыми зернами `QcObsBean`, `QcSetBean` и `ObservationBean`, а также приложением `QcClient`, которые будут созданы позже в этом разделе.
- `QcObsHelper` — это оболочка, отображающая некоторые атрибуты таблицы `PASSED_OBSERVATION_LIST`.
- `QcSetHelper` — это оболочка для таблицы `QC_EVENT_LIST`.
- `ObsException` — это описываемый пользователем Java-класс исключительной ситуации.

Создание класса QcObsHelper

Класс QcObsHelper отображает атрибуты таблицы PASSED_OBSERVATION_LIST и состоит из конструкторов по умолчанию и с параметрами. Вот его описание:

```
/*  Имя программы:   QcObsHelper.java
**
**  Назначение:      Серийный Helper-класс Java, используемый
**                   для передачи объектов параметров
**                   (PASSEDOBS) между клиентами,
**                   компонентами EJB и CORBA.
*/
package helpers;

import java.io.Serializable;
import java.math.BigDecimal;

public class QcObsHelper
    implements java.io.Serializable {

    public BigDecimal qc_id = null;
    public BigDecimal obs_id = null;

    // Конструктор по умолчанию
    public QcObsHelper() {
    }

    // Конструктор с параметрами
    public QcObsHelper ( BigDecimal qc_id, BigDecimal obs_id) {
          this.qc_id = qc_id;
          this.obs_id = obs_id;
    }
}
```

Создание класса QcSetHelper

Класс QcSetHelper отображает атрибуты таблицы QC_EVENT_LIST. Вот его описание:

```
/*  Имя программы:   QcSetHelper.java
**
**  Назначение:      Серийный Helper-класс Java,
**                   передающий объекты параметров
**                   (QUALITY_CONTROL_EVENT)
**                   между клиентами, компонентами EJB и CORBA.
*/
package helpers;

import java.io.Serializable;
import java.math.BigDecimal;

public class QcSetHelper
   implements java.io.Serializable {

    public BigDecimal qc_id = null;
    public String when_t = null;
    public String at_time = null;
    public BigDecimal event_id = null;
    public BigDecimal whom_id = null;
    public String evt_name = null;
    public String sc_name = null;

    // Конструктор по умолчанию
    public QcSetHelper() {
    }

    // Конструктор 1
    public QcSetHelper (BigDecimal qc_id, String when_t,
```

```
                        String at_time, BigDecimal event_id,
                        BigDecimal whom_id) {
        this.qc_id = qc_id;
        this.when_t = when_t;
        this.at_time = at_time;
        this.event_id = event_id;
        this.whom_id = whom_id;
    }
    // Конструктор 2
    public QcSetHelper (BigDecimal qc_id, String when_t,
         String at_time, BigDecimal event_id, BigDecimal whom_id,
         String evt_name, String sc_name) {

     this.qc_id = qc_id;
     this.when_t = when_t;
     this.at_time = at_time;
     this.event_id = event_id;
     this.whom_id = whom_id;
     this.evt_name = evt_name;
     this.sc_name = sc_name;
    }
    // Конструктор 3
    public ResetHelper ( BigDecimal qc_id ) {
     this.qc_id = qc_id;
    }
}
```

Создание класса ObsException

Некоторые методы сеансовых зерен QcSetBean и QcObsBean могут устанавливать исключительную ситуацию ObsException. Для передачи исключительной ситуации из серверных объектов клиенту необходимо описать специальный класс, как показано ниже:

```
/*  Имя программы:   ObsException.java
**
**  Назначение:      Класс исключительной ситуации,
**                   подлежащий переносу из серверных
**                   объектов клиентам.
*/
package helpers;

import java.rmi.*;

public class ObsException
    implements RemoteException {

    // Конструктор по умолчанию
    public ObsException (String msg ) {

        super(msg);
    }
}
```

Сеансовое зерно QcObsBean

Зерно QcObsBean дает пользователю возможность вводить новую запись в таблицу PASSED_OBSERVATION_LIST и считывать сведения обо всех проверенных наблюдениях за атмосферой для конкретного идентификатора QcSet.

Компонентами зерна QcObsBean являются:

- Класс ObsHelper
- Класс QcObsHelper
- Класс ObsException

- **Интерфейс** QcObsRemote
- **Интерфейс** QcObsHome
- Класс QcObsImp
- Дескриптор внедрения QcObs.ejb

Создание интерфейса QcObsRemote

В интерфейсе QcObsRemote **объявляются методы, которые могут вызываться клиентами. Вот его описание:**

```
// Имя программы:    QcObsRemote.java
// Название:         Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Компания:         Ваша компания
// Описание:
//
package qcobsserver;

// импортируем helper-классы
import helpers.QcObsHelper;
import helpers.ObsException;
import java.util.ArrayList;

import java.rmi.*;
import javax.ejb.*;
import java.sql.SQLException;

public interface QcObsRemote extends EJBObject {

    // Вставим объект QcObsHelper в базу данных Oracle8i
    public void insertQcObs(ArrayList qcObs)
        throws SQLException, RemoteException;

    // Получим сведения о наблюдении и возвратим объект
    // ObsHelper по идентификатору проверки.
    public ArrayList getQcObsInfo(int qcid)
        throws SQLException, ObsException, RemoteException;
    }
```

Создание интерфейса QcObsHome

Интерфейс QcObsHome **состоит из одного метода** create(), **который используется клиентами для создания экземпляров зерна** QcObsBean. **Приведем описание интерфейса:**

```
// Имя программы:    QcObsHome.java
// Название:         Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Компания:         Ваша компания
// Описание:
//
package qcobsserver;

import java.rmi.*;
import javax.ejb.*;

public interface QcObsHome extends EJBHome {

    // Создадим экземпляр интерфейса QcObsRemote
    public QcObsRemote create()
        throws CreateException, RemoteException;
}
```

Создание класса QcObsImp

Класс QcObsImp содержит тела методов для всех сигнатур методов, описанных в интерфейсе QcObsRemote. Метод insertQcObs() вызывает PL/SQL-процедуру QCACTIONS.INSERTQCSETOBS() для ввода данных в таблицу PASSED_OBSERVATION_LIST. Вот описание класса QcObsImp:

```
/* Имя программы:    QcObsImp.java
** Версия:
** Автор:            Ваше имя
** Описание:         Bean-класс EJB, позволяющий клиенту вводить
**                   объект QcObsHelper, ArrayList объектов
**                   QcObsHelper и считывать ArrayList объектов
**                   ObsHelper
**
*/
package qcobsserver;

// Импортируем helper-классы
import helpers.ObsHelper;
import helpers.QcObsHelper;
import helpers.ObsException;

import javax.ejb.SessionContext;
import javax.jts.UserTransaction;
import java.rmi.RemoteException;
import javax.ejb.*;

// Импортируем специфичную для приложения исключительную ситуацию
import java.sql.SQLException;
import java.sql.*;
import oracle.sql.*;

// Импортируем jdbc-драйвер Oracle
import oracle.jdbc.driver.*;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.math.*;
import java.util.ArrayList;

public class QcObsImp implements SessionBean {

SessionContext ctx;

// Введем новую строку данных в таблицу
// PASSED_OBSERVATION_LIST.
// Этот метод могут вызывать клиенты.
public void insertQcObs (ArrayList qcObs)
     throws SQLException, RemoteException {

    Connection conn = null;
    int arraySize = qcObs.size();
    int loopVar = 0;

    try {
        // С помощью внутреннего драйвера JDBC подключимся к БД
        conn = connectDb();

        while ( loopVar < arraySize ) {
             QcObsHelper anQcObs = (QcObsHelper)qcObs.get(loopVar);

             // Введем anQcObs во ВЛОЖЕННУЮ таблицу таблицы
             // PASSED_OBSERVATION_LIST
             insertQcObs(anQcObs, conn);
```

```
            loopVar++ ;
        } // Конец while
    } // Конец try
    catch (SQLException e) {
        throw e;
    } // Конец catch
    catch (java.lang.NullPointerException e) {
        throw e ;
    } // Конец catch
} // Конец insertQcObs(ArrayList qcObs)

// Введем новую строку данных в таблицу
// PASSED_OBSERVATION_LIST
public void insertQcObs (QcObsHelper qcObs, Connection conn)
        throws SQLException, RemoteException {
   // Создадим объект CallableStatement
   CallableStatement cstmt = null;
   try {
        // Подготовим String для вызова PL/SQL-процедуры
        // QCACTIONS.INSERTQCSETOBS().
        // Этот оператор готовит Java-объект типа String, который
        // состоит из оператора, необходимого для вызова PL/SQL-
        // процедуры QCACTIONS.INSERTQCSETOBS(), вводящей новую
        // запись в таблицу PASSED_OBSERVATION_LIST базы данных.
        String sql =
             "{call QCACTIONS.INSERTQCSETOBS(?,?)}";
        cstmt = conn.prepareCall(sql);
        cstmt.setInt(1, qcObs.qc_id.intValue());
        cstmt.setInt(2, qcObs.obs_id.intValue());
        cstmt.execute();
   } // Конец try
   catch (SQLException e) {
        throw e;
   } // Конец catch
   catch (java.lang.NullPointerException e) {
        throw e;
   } // Конец catch
   // Выполним очистку
   finally {
            if ( cstmt != null ) cstmt.close();
   } // Конец finally()
} // Конец insertQcObs(qcObs,conn)

// Получим сведения о наблюдении и возвратим массив из объектов
// QcObsHelper на основании идентификатора проверки.
public ArrayList getQcObsInfo(int qcid)
     throws SQLException, ObsException, RemoteException {
   // Создадим оператор CallableStatement
   CallableStatement cstmt = null;

   Connection conn = null;
   OracleResultSet rset = null;
   ArrayList obsList = null;
   try {
        conn = connectDb();
        // Подготовим String для вызова PL/SQL-процедуры
        // QCACTIONS.GETQCOBS().
        // Этот оператор готовит Java-объект типа String, который
        // состоит из оператора, необходимого для вызова PL/SQL-
        // процедуры QCACTIONS.GETQCOBS(), возвращающей все
        // элементы ВЛОЖЕННОЙ таблицы, атрибута таблицы
        // PASSED_OBSERVATION_LIST базы данных.
        String sql =
         "{call QCACTIONS.GETQCOBS(?,?)}";
```

```
        cstmt = conn.prepareCall(sql);
        // Объявим, что ? - возвращаемое значение типа Integer
        cstmt.setInt(1, qcid);
        // Объявим, что ? - возвращаемое значение типа
        // OracleTypes.CURSOR.
        // Процедура QCACTIONS.GETQCOBS() принимает два параметра:
        // int в качестве параметра IN (хранящего qc_id для QcSet)
        // и IN OUT параметр с типом данных OracleTypes.CURSOR.
        // Применяйте CURSOR, когда возвращаемый тип входного
        // параметра функции или процедуры PL/SQL - Oracle-тип
        // REF CURSOR. При использовании JDBC-драйвера Oracle тип
        // REF CURSOR требует применения OracleTypes.CURSOR в
        // качестве выходного параметра регистрации. Для получения
        // данных из результирующего курсора необходимо привести
        // его к типу OracleResultSet. Тип OracleResultSet
        // аналогичен Java.sql.ResultSet.
        cstmt.registerOutParameter(2, OracleTypes.CURSOR);
        cstmt.execute();
    } // Конец try
    catch (SQLException e) {
      //  e.printStackTrace();
        throw e;
    } // Конец catch
    try {
        // Приведем OracleTypes.CURSOR к типу OracleResultSet.

        // Этот оператор приводит OracleTypes.CURSOR к типу
        // OracleResultSet, содержащему еще и метод next().
        // Как и при использовании Java-типа ResultSet, курсор
        // опустошается с помощью метода next().
        rset =
            (OracleResultSet)cstmt.getObject(2);
        // Создадим экземпляр ArrayList obsList
        obsList = new ArrayList();
        while (rset.next()) {
            ObsHelper oneObs = populateQcObsData(rset);
            // Введем объект oneObs в ArrayList obsList
            obsList.add(oneObs);
        }  // Конец while
        // Уменьшим obsList до реального размера
        obsList.trimToSize();
        // Возвратим ArrayList из объектов ObsHelper вызывающему
        return obsList;
    } //  Конец try
    catch (java.lang.NullPointerException e) {
        throw new java.lang.NullPointerException (e.getMessage());
    } // Конец catch
    catch (SQLException e) {

        // Когда метод getObject() объекта CallableStatement
        // используется для заполнения rset OracleResultSet в
        // Oracle 8.1.6, при выполнении этого оператора можно
        // получить исключительную ситуацию SQLException, если
        // в программной конструкции PL/SQL не был открыт курсор.
        // Устанавливается SQLException с именем INVALID CURSOR.
        // Для корректного выхода из программы необходимо
        // перехватывать эту ошибку и задавать для OracleResultSet
        // rset null-значение.
            rset = null;
      // e.printStackTrace();
        throw new ObsException("No QcObs for this QcSet id: "
                +qcid );
    } // Конец catch
    // Выполним очистку
```

```
   finally {
             if ( cstmt != null ) cstmt.close();
             if ( rset != null ) rset.close();
   } // Конец finally()
} // Конец getQcObsInfo(int qcid)

// Этот метод заполняет объект ObsHelper результирующим
// набором данных.
public ObsHelper populateQcObsData (ResultSet rset)
     throws SQLException {
   try {
        // Объявим объект ObsHelper с помощью конструктора
        // по умолчанию.
        ObsHelper oneObs = new ObsHelper();
        // Установим переменные элементов объекта ObsHelper.
        // Отметим, что можно было бы воспользоваться
        // конструктором 1 с параметрами, описанным в классе
        // ObsHelper.
        oneobs.obs_id    = new BigDecimal(rset.getInt(1));
        oneobs.when_t    = rset.getString(2);
        oneobs.at_time   = rset.getString(3);
        rset.getInt(4);
        if  (rset.wasNull())
             oneObs.station_id = null;
        else
             oneObs.station_id = new BigDecimal(rset.getInt(4));
        rset.getInt(5);
        if  (rset.wasNull())
             oneObs.produced_id = null;
        else
             oneObs.produced_id = new BigDecimal(rset.getInt(5));
        rset.getInt(6);
        If  (rset.wasNull())
             oneObs.latitude = null;
        else
             oneObs.latitude = new BigDecimal(rset.getInt(6));
        rset.getInt(7);
        If  (rset.wasNull())
             oneObs.longitude = null;
        else
             oneObs.longitude = new BigDecimal(rset.getInt(7));
        rset.getInt(8);
        If  (rset.wasNull())
             oneObs.wdspd = null;
        else
             oneObs.wdspd = new BigDecimal(rset.getInt(8));
        rset.getInt(9);
        If  (rset.wasNull())
             oneObs.adj_wdspd = null;
        else
             oneObs.adj_wdspd = new BigDecimal(rset.getInt(9));
        rset.getInt(10);
        If  (rset.wasNull())
             oneObs.wddir = null;
        else
             oneObs.wddir = new BigDecimal(rset.getInt(10));
        rset.getInt(11);
        If  (rset.wasNull())
             oneObs.pressure = null;
        else
             oneObs.pressure = new BigDecimal(rset.getInt(11));
        return oneObs;
   } // Конец try
```

```
    catch (SQLException e) {
         throw e;
    } // Конец catch
    catch (java.lang.NullPointerException e) {
         throw e;
    } // Конец catch
} // Конец populateQcObsData()

// Этот метод вызывается всеми методами, которым нужно
// подключиться к БД с помощью внутреннего
// драйвера JDBC.
// Заметим, что программную конструкцию для получения объекта
// Connection можно размещать в методе ejbCreate().
public static Connection connectDb()
        throws SQLException, RemoteException {
    // Обратите внимание: этот метод доступен только методам
    // Bean-класса, но НЕ клиентам.
    // Создадим переменную java.sql.Connection.
    Connection conn = null;
        DriverManager.registerDriver
             (new oracle.jdbc.driver.OracleDriver());
    // Выполним конкретизацию с помощью внутреннего JDBC-драйвера Oracle8i.
    conn =
        DriverManager.getConnection("jdbc:default:connection:");
        // Возвратим объект java.sql.Connection
        return conn;
  } // Конец connectDb()

    public void ejbCreate()
      throws RemoteException, CreateException {
    } // Конец ejbCreate()
    public void ejbActivate()throws RemoteException {
    } // Конец ejbActivate()
    public void ejbPassivate() throws RemoteException {
    } // Конец ejbPassivate()
    public void ejbRemove() throws RemoteException {
    } // Конец ejbRemove ()
    public void setSessionContext(SessionContext ctxArg)
      throws RemoteException {
     ctx = ctxArg;
    } // Конец setSessionContext()
} // Конец класса QcObsImp
```

Сеансовое зерно QcSetBean

Зерно QcSetBean вводит данные в таблицу QC_EVENT_LIST, считывает данные из таблицы и вызывает методы QcObsBean для ввода данных в таблицу PASSED_OBSERVATION_LIST и для считывания данных из этой таблицы. Общий вид взаимодействия между приложением QcClient и сеансовыми зернами ObservationBean, QcSetBean и QcObsBean показан на рис. 5.3.

Компонентами зерна QcSetBean являются:

- Класс ObsHelper
- Класс QcObsHelper
- Класс ObsException
- **Интерфейс** QcSetRemote
- **Интерфейс** QcSetHome
- Класс QcSetImp
- Дескриптор внедрения QcSet.ejb

Создание интерфейса QcSetRemote

В интерфейсе QcSetRemote объявляются методы, работающие с таблицей QC_EVENT_LIST. Обратите внимание: методы insertQcObs() и getQcObsInfo() этого интерфейса неявно работают с таблицей PASSED_OBSERVATION_LIST. Их вызов приводит к вызову методов insertQcObs() и getQcObsInfo() зерна QcObsBean. Класс QcSetImp содержит тела методов insertQcObs() и getQcObsInfo(), указанных в интерфейсе QcSetRemote.

Приведем описание интерфейса QcSetRemote:

```
// Имя программы:    QcSetRemote.java
// Название:         Название вашего продукта
// Версия:
// Авторское право:  Copyright (c) 2000
// Автор:            Ваше имя
// Компания:         Ваша компания
// Описание:
//
package obsserver;

// импортируем helper-классы
import helpers.QcSetHelper;
import helpers.QcObsHelper;
import helpers.ObsHelper;
import helpers.ObsException;

import java.util.ArrayList;
import java.rmi.*;
import javax.ejb.*;

import java.sql.SQLException;

public interface QcSetRemote extends EJBObject {

   // Вставим объект QcSetHelper в базу данных Oracle8i
   // и связанные с ним проверенные наблюдения
   public QcSetHelper insertQcSet(QcSetHelper qc,ArrayList qcObs)
        throws SQLException, RemoteException;

   // Получим все сведения о QcSet и возвратим информацию в объект
   // ArrayList из объектов QcSetHelper. Этот метод возврата всех
   // данных из таблицы QC_EVENT_LIST использует динамический SQL.
   public ArrayList getAllQcInfo()
        throws SQLException, ObsException, RemoteException;

   // Получим сведения о QcSet и возвратим объект QcSetHelper
   // на основании идентификатора проверки.
   public QcSetHelper getQcSetInfo(int qcid)
        throws SQLException, ObsException, RemoteException;

   // Введем новую строку данных в таблицу PASSED_OBSERVATION_LIST
   // Этот метод вызывает insertQcObs() зерна QcObsBean
   public void insertQcObs (BigDecimal qc_id, ArrayList qcObs)
        throws SQLException, RemoteException;

   // Получим сведения о наблюдении и возвратим массив объектов
   // QcObsHelper по идентификатору проверки.
   public ArrayList getQcObsInfo(int qcid)
        throws SQLException, ObsException, RemoteException;
}
```

Создание интерфейса QcSetHome

Приведем описание интерфейса:

```
// Имя программы:    QcSetHome.java
```

```
// Название:        Название вашего продукта
// Версия:
// Авторское право: Copyright (c) 2000
// Автор:           Ваше имя
// Компания:        Ваша компания
// Описание:
//
package obsserver;
import java.rmi.*;
import javax.ejb.*;

public interface QcSetHome extends EJBHome {
    // Создадим экземпляр интерфейса QcSetRemote
    public QcSetRemote create()
        throws CreateException, RemoteException;
}
```

Создание класса QcSetImp

Класс QcSetImp является типичным примером размещенного на сервере клиентского объекта, иллюстрирующим процесс начала зерном EJB нового сеанса связи с базой данных с одновременным выполнением в текущем:

```
/* Имя программы:   QcSetImp.java
** Версия:
** Автор:           Ваше имя.
** Описание:        Bean-класс EJB, позволяющий клиенту вводить
**                  объект QcSetHelper, ArrayList объектов
**                  QcSetHelper и считывать ArrayList объектов
**                  QcObsHelper.
**                  Класс QcSetImp манипулирует данными таблицы
**                  Qc_event_list и использует зерно QcObsBean
**                  для манипулирования данными в таблице
**                  Passed_Observation_List.
*/
package obsserver;

// Импортируем helper-классы
import helpers.QcSetHelper;
import helpers.ObsHelper;
import helpers.QcObsHelper;

// Импортируем объекты qcobsserver.
// Этот и следующий операторы импортируют интерфейсы
// QcObsRemote и QcObsHome сеансового зерна QcObsBean. Класс
// QcSetImp - это клиент зерна QcObsBean. Обратите внимание:
// автономные клиенты Java и зерна EJB должны импортировать
// домашний и удаленный интерфейсы каждого опубликованного объекта,
// с которым они хотят работать.
import qcobsserver.QcObsRemote;
import qcobsserver.QcObsHome;

import javax.ejb.SessionContext;
import javax.jts.UserTransaction;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.rmi.RemoteException;
import javax.ejb.*;

// Импортируем специфичную для приложения исключительную ситуацию
import java.sql.SQLException;

import java.sql.*;
```

```
import oracle.sql.*;

// Импортируем jdbc-драйвер Oracle
import oracle.jdbc.driver.*;
// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.math.*;
import java.util.ArrayList;
import java.util.Hashtable;

public class QcSetImp implements SessionBean {

    // Создадим переменную типа SessionContext
    SessionContext ctx;
    // Создадим глобальную переменную типа QcObsRemote.
    // Этот оператор создает глобальную переменную qcObsObj типа
    // QcObsRemote. В ней будет храниться ссылка на интерфейс
    // QcObsRemote.
    // (см. пояснение 1)
    QcObsRemote qcObsObj = null;

    // Метод, который не могут вызывать клиенты, подключается
    // к БД и вызывает переопределяемый getNewQcSetId(conn)
    // для получения нового qc_id.
    public QcSetHelper getNewQcSetId()
        throws SQLException, RemoteException {
            Connection conn = null;
            try {
                // С помощью внутреннего драйвера JDBC подключимся к БД.
                conn = connectDb();
                QcSetHelper qc = getNewQcSetId(conn);
                return qc;
            } // Конец try
            catch(SQLException e) {
                throw e ;
            } // Конец catch()
    } // Конец getNewQcSetId()

    // Этот метод НЕ может вызываться клиентами
    // Поэтому он не указан в интерфейсе
    // QcSetRemote interface.
    public QcSetHelper getNewQcSetId(Connection conn)
          throws SQLException, RemoteException {
        // Создадим объект CallableStatement
        CallableStatement cstmt = null;

        // Вызовем getNewQcId() из PL/SQL-модуля QcActions,
        // чтобы получить новый qc_id
        try {
            // Подготовим String для вызова
            String sqlId =
               "{? = call QCACTIONS.GETNEWQCID}";
            // вызов, используя соединение JDBC
            // по умолчанию
            cstmt = conn.prepareCall(sqlId);
            // Объявим, что ? - возвращаемое значение типа Integer
            cstmt.registerOutParameter (1, Types.INTEGER);
            // Узнаем новый qcid, выполнив запрос
            cstmt.execute ();
            BigDecimal anQcId = new BigDecimal(cstmt.getInt (1));
            // Возвратим объект QcSetHelper с помощью
            // нового идентификатора
            return new QcSetHelper(anQcId);
        } // Конец try
```

```
    catch (SQLException e) {
         throw e;
    } // Конец catch
    // Выполним очистку
    finally {
            if ( cstmt != null ) cstmt.close();
    } // Конец finally()
} // Конец getNewQcSetId(conn)

// ИНИЦИИРУЕМАЯ КОНТЕЙНЕРОМ ТРАНЗАКЦИЯ
// Этот метод получает новый qc_id и добавляет новое событие
// QUALITY_CONTROL_EVENT в таблицу QC_EVENT_LIST. Его
// вызывают клиенты.
public QcSetHelper insertQcSet(QcSetHelper inQc)
    throws SQLException, RemoteException {
 QcSetHelper qc = inQc;
 Connection conn = null;
 try {
       // С помощью внутреннего драйвера JDBC подключимся к БД.
       conn = connectDb();
       // Узнаем новый new qc_id в базе данных
       QcSetHelper anQcid = getNewQcSetId(conn);
       // Обновим входной qc_id
       qc.qc_id = anQcid.qc_id;
       // Введем новый OCEANIC_OBSERVATION_TYPE
       anQcid = insertQcSet(qc,conn);

       // Возвратим объект QcSetHelper
       // Этот оператор возвращает вновь сгенерированный
       // идентификатор qc_id вызывающему
       return new QcSetHelper(anQcid.qc_id);
 } // Конец try
 catch (SQLException e) {
    throw e;
 } // Конец catch
 catch (java.lang.NullPointerException e) {
    throw e;
 } // Конец catch
} // Конец insertQcSet(qc)

public QcSetHelper insertQcSet(QcSetHelper qc, Connection conn)
    throws SQLException, RemoteException {
 // Создадим объект CallableStatement
 CallableStatement cstmt = null;
 try {
       // Подготовим String для вызова PL/SQL-процедуры
       // QCACTIONS.INSERTQCSET().
       String sql =
         "{call QCACTIONS.INSERTQCSET(?,?,?,?,?)}";
       cstmt = conn.prepareCall(sql);
       cstmt.setInt(1, qc.qc_id.intValue());
       cstmt.setString(2, qc.when_t);
       cstmt.setString(3, qc.at_time);
       cstmt.setInt(4, qc.event_id.intValue());
       cstmt.setInt(5, qc.whom_id.intValue());
       cstmt.execute();
       // Возвратим объект QcSetHelper
       return new QcSetHelper(qc.qc_id);
 }  // Конец try
 catch (SQLException e) {
    throw e;
 } // Конец catch
 // Выполним очистку
```

```
 finally {
        if ( cstmt != null ) cstmt.close();
 } // Конец finally()
} // Конец insertQcSet(qc,conn)

// Этот метод использует динамический SQL для возвращения всех данных
// из таблицы QC_EVENT_LIST
public ArrayList getAllQcInfo()
     throws SQLException, ObsExceptionRemoteException {
    Connection conn = null;
    // Для сохранения строк таблицы необходим вектор Java,
    // так как неизвестно, сколько строк считывается.
    ArrayList qcList = null;
    // Объявим Java-объект ResultSet
    ResultSet rset = null;
    // Объявим переменную PreparedStatement
    PreparedStatement pstmt = null;
    // Подготовим SQL-оператор SELECT
    String sql =
         "SELECT Q.qc_id, TO_CHAR(Q.when_t,'DD-MM-YYYY'),"
         + "Q.at_time, Q.event_id, Q.for_event.name, Q.whom_id, "
         + "Q.by_whom.lastname " + "FROM QC_EVENT_LIST Q";
    try {
         // С помощью внутреннего драйвера JDBC подключимся к БД
         conn = connectDb();
         // Подготовим оператор
         pstmt = conn.prepareStatement(sql);
         // Выполним запрос и сохраним результат в Java-объекте
         // ResultSet
         rset = pstmt.executeQuery();
         // Конкретизируем ArrayList qcList
         qcList = new ArrayList();
         while (rset.next()) {
            // Объявим объект QcSetHelper
            QcSetHelper oneQc = populateQcSetData(rset);
            // Введем объект oneObs в ArrayList qcList
            qcList.add(oneQc);
         } // Конец while
         // Проверим размер ArrayList
         if  (qcList.size() == 0)
              throw new ObsException("No QcSet Data")
         // Усечем размер qcList
         qcList.trimToSize();
         // Возвратим объект ArrayList вызывающему
         return qcList;
    } // Конец try
    catch (SQLException e) {
         throw e;
    } // Конец catch
    catch (java.lang.Exception e) {
         throw e;
    } // Конец catch
    // Выполним очистку
    finally {
          if ( rset != null ) rset.close();
          if ( pstmt != null ) pstmt.close();
    } // Конец finally()
} // Конец getAllQcInfo

// Получим сведения о наблюдении для конкретного qc_id
public QcSetHelper getQcSetInfo(int qcid)
     throws SQLException, RemoteException {
    Connection conn = null;
    int qc_id = qcid;
```

```
    // Объявим Java-объект ResultSet
    ResultSet rset = null;
    // Объявим переменную PreparedStatement
    PreparedStatement pstmt = null;
    // Подготовим SQL-оператор SELECT
    String sql =
         "SELECT Q.qc_id, TO_CHAR(Q.when_t,'DD-MM-YYYY'),"
         + "Q.at_time, Q.event_id, Q.for_event.name, Q.whom_id, "
         + "Q.by_whom.lastname " + "FROM QC_EVENT_LIST Q "
         + "WHERE Q.qc_id = ?";
    try {
         conn = connectDb();
         pstmt = conn.prepareStatement(sql);
         pstmt.setInt(1, qc_id);
         rset = pstmt.executeQuery();
         QcSetHelper oneQc = null;
         while (rset.next()) {
            oneQc = populateQcSetData(rset);
         } // Конец while
         return oneQc;
    } // Конец try
    catch (SQLException e) {
         throw e;
    } // Конец catch
    // Выполним очистку
    finally {
              if ( rset != null ) rset.close();
              if ( pstmt != null ) pstmt.close();
    } // Конец finally()
    } // Конец getQcSetInfo()

// Этот метод заполняет объект QcSetHelper данными
// результирующего набора.
public QcSetHelper populateQcSetData (ResultSet rset)
     throws SQLException {
    try {
         // Объявим объект QcSetHelper с помощью
         // конструктора по умолчанию.
         QcSetHelper oneQc = new QcSetHelper();
         // Установим переменные для элементов объекта QcSetHelper.
         // Заметьте, что можно было бы воспользоваться
         // конструктором 1 с параметрами, описанным в классе
         // QcSetHelper.
         oneqc.qc_id    = new BigDecimal(rset.getInt(1));
         oneqc.when_t   = rset.getString(2);
         oneqc.at_time  = rset.getString(3);
         rset.getInt(4);
         if    (rset.wasNull())
                oneQc.event_id = null;
         else
                oneQc.event_id = new BigDecimal(rset.getInt(4));
         oneQc.evt_name = rset.getString(5);
         rset.getInt(6);
         if    (rset.wasNull())
                oneQc.whom_id = null;
         else
            oneQc.whom_id = new BigDecimal(rset.getInt(6));
         oneQc.sc_name = rset.getString(7);
         return oneQc;
} // Конец try
catch (SQLException e) {
    throw e;
} // Конец catch
```

```
        catch (java.lang.NullPointerException e) {
            throw e;
        } // End catch
    } // End populateQcSetData()

    // Введем новую строку данных в таблицу PASSED_OBSERVATION_LIST
    public void insertQcObs (BigDecimal qc_id, ArrayList qcObs)
         throws SQLException, RemoteException {
     try {
          int arraySize = qcObs.size();
          // Создадим новый объект ArrayList с помощью qcObs.size().
          ArrayList arrayOfQcObsHelpers =
                       new ArrayList(arraySize);
          int loopVar = 0;
          while ( loopVar < arraySize ) {
             BigDecimal anObsid = (BigDecimal)qcObs.get(loopVar);
            // Создадим объект QcObsHelper с помощью qc_id и anObsid
            QcObsHelper aQcObsHelper =
                    new QcObsHelper(qc_id,anObsid);
            // Поместим объект QcObsHelper в объект ArrayList
            arrayOfQcObsHelpers.add(aQcObsHelper);
            loopVar++;
          } // End while
          // Вызовем insertQcObs (ArrayList qcObs) (см. пояснение 2).
          // Этот оператор использует ссылку на интерфейс
          // QcObsRemote для вызова метода insertQcObs() класса
          // QcObsImp. Это типичный пример ситуации, когда
          // существующее зерно EJB (клиентский объект на сервере)
          // вызывает другое, а также пример транзакций,
          // ограниченных на стороне сервера.
          // В дескрипторе внедрения QcObs.ejb для зерна QcObsBean
          // используется значение атрибута TX_REQUIRED_NEW.
          // Следовательно, зерно QcObsBean будет вызываться в новой
          // транзакции.

          qcObsObj.insertQcObs(qcObs);

     }  // Конец try
     catch (SQLException e) {
         throw e;
     }  // Конец catch
     catch (java.lang.NullPointerException e) {
         throw e;
     }  // Конец catch
    } // Конец insertQcObs(BigDecimal qc_id, ArrayList qcObs)

    // Получим сведения о наблюдениях и возвратим ArrayList объектов
    // ObsHelper на основании идентификатора проверки.
    // Этот оператор описывает метод getQcObsInfo() зерна
    // QcSetBean. При вызове первый обращается к методу
    // getQcObsInfo() сеансового зерна QcObsBean.
public ArrayList getQcObsInfo(int qcid)
         throws SQLException, RemoteException {
     ArrayList obsList = null;
     try {
          // Вызовем getQcObsInfo().
          // Этот оператор вызывает метод getQcObsInfo()
          // сеансового зерна QcObsBean и возвращает объект
          // ArrayList из объектов ObsHelper зерну QcSetBean.
          obsList = qcObsObj.getQcObsInfo(qcid);

          // Возвратим объект ArrayList
          return obsList;
     } // Конец try
```

```
    catch (java.lang.NullPointerException e) {
        throw e;
    } // Конец catch
    catch (SQLException e) {
      // e.printStackTrace();
        throw e;
    } // Конец catch
  } // Конец getQcObsInfo(int qcid)

  // Этот метод вызывается всеми методами, которым необходимо
  // подключиться к базе данных с помощью внутреннего драйвера JDBC
  public static Connection connectDb()
        throws SQLException, RemoteException {
    // Обратите внимание: этот метод доступен только методам
    // Bean-класса, но НЕ клиентам
    // Создадим переменную java.sql.Connection
    Connection conn = null;
      DriverManager.registerDriver
            (new oracle.jdbc.driver.OracleDriver());
      // Проведем конкретизацию, используя внутренний JDBC-драйвер Oracle8i
      conn =
            DriverManager.getConnection("jdbc:default:connection:");
      // Возвратим объект java.sql.Connection
      return conn;
  } // End connectDb()

  // (см. пояснение 3)
  public void ejbCreate()
        throws RemoteException, CreateException {
    try {
         // Создадим InitialContext (см. пояснение 4)
         Hashtable env = new Hashtable();
         env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
         Context ic = new InitialContext(env);
         // Создадим зерно QcObsImp (см. пояснение 5)
         String objectName = new String("/test/QcObsBean");

         // Начнем новый сеанс связи с базой данных, одновременно
         // стремясь найти ссылку на домашний интерфейс QcObsBean.
         // (см. пояснение 6)
         QcObsHome anQcObsHome =
             (QcObsHome)ic.lookup("sess_iiop://thisServer"
                   +objectName);

        // Для большей эффективности используйте тот же сеанс БД.
        // Достаточно набрать:
        // QcObsHome anQcObsHome =
        // (QcObsHome)ic.lookup("sess_iiop://thisServer/:thisSession"
        // +objectName);
         // (см. пояснение 7)
         qcObsObj = anQcObsHome.create();
    } // Конец try
    catch (NamingException e) {
        // e.printStackTrace();
    } // Конец catch
  } // Конец ejbCreate()
  public void ejbActivate() throws RemoteException {
  } // Конец ejbActivate()
  public void ejbPassivate() throws RemoteException {
  } // Конец ejbPassivate()
  public void ejbRemove() throws RemoteException {
  } // Конец ejbRemove()
  // (см. пояснение 8)
  public void setSessionContext(SessionContext ctx)
```

```
        throws RemoteException {
      this.ctx = ctx;
  } // Конец setSessionContext()
}
```

Пояснения к классу QcSetImp:

1. В глобальной переменной qcObsObj будет храниться ссылка на зерно QcObsBean. Ссылка на QcObsBean Bean получается, когда клиент вызывает метод create() зерна QcSetBean. Напомним, что при вызове клиентом метода create() домашнего интерфейса EJB EJB-контейнер вызывает метод ejbCreate(), связанный с первым и описанный в соответствующем Bean-классе. В данном примере при создании приложением QcClient экземпляра зерна QcSetBean класс QcSetImp автоматически и прозрачно создает экземпляр зерна QcObsBean.
2. В этом операторе ссылка на интерфейс QcObsRemote используется для вызова метода insertQcObs() класса QcObsImp. Напомним, что зерно QcSetBean получило экземпляр QcObsBean, когда объект QcClient вызвал метод create() интерфейса QcSetHome. Это типичный пример ситуации, когда существующее зерно EJB (клиентский объект на сервере) вызывает другое, а также пример транзакций, ограниченных на стороне сервера. В данном примере EJB-контейнер создаст новый объект контекста транзакций. Зерно QcObsBean входит в транзакцию, так как для вызывающего (зерна QcSetBean) транзакция действует благодаря атрибуту транзакций TX_SUPPORTS. Отметим, что контекст транзакций, открытый в зерне QcSetBean, не распространяется на QcObsBean. В дескрипторе внедрения QcObs.ejb для зерна QcObsBean используется значение атрибута TX_REQUIRED_NEW. Зерно QcObsBean будет вызываться в новой транзакции. При выполнении транзакция, начавшаяся из приложения QcClient и распространившаяся на QcSetBean, будет приостановлена и возобновлена после выполнения метода insertQcObs() класса QcObsImp.
3. Этот оператор описывает метод ejbCreate() зерна QcSetBean. Когда программа вызывает метод create() домашнего интерфейса EJB, EJB-контейнер вызывает для зерен STATELESS или STATEFUL методы:
   - newInstance(), который выделяет место и "оживляет" зерно
   - setSessionContext(), который предоставляет контекст сеанса Передаваемый объект SessionContext указывает на динамический объект, который обновляется EJB-контейнером каждый раз при вызове методов зерна. EJB-контейнер сохраняет объект SessionContext в течение всего существования сеансового зерна.
   - ejbCreate(), который подобно конструктору Java создаст экземпляр EJB и инициализирует внутренние переменные состояния экземпляра. Как и конструктор Java, метод ejbCreate() можно сделать таким, чтобы он инициализировал переменные, подключался к БД или создавал экземпляры других зерен.
4. В этом блоке try зерно QcSetBean создаст экземпляр QcObsBean. Подобно автономным клиентским программам Java, для получения ссылки на нужный опубликованный объект (в данном случае на зерно QcObsBean) зерно QcSetBean будет использовать JNDI-метод lookup(). Самый первый шаг – установка среды. Заметим, что устанавливается только переменная URL_PKG_PREFIXES – единственная переменная среды, которую необходимо установить, поскольку другие уже были установлены раньше самым первым клиентом (т.е. QcClient), вызвавшим метод create() интерфейса QcSetHome. За подробностями об установке переменных среды серверных объектов JNDI обратитесь к разделу "Серверный объект, начинающий новый сеанс" этой главы. После установки переменных среды с их помощью нужно создать JNDI-объект Context. Зерно QcSetBean начнет новый сеанс базы данных, используя комбинацию URL sess_iiop://thisServer и

имени опубликованного объекта /test/QcObsBean. Его служба транзакций создаст новый контекст транзакций, с помощью которого будут вызываться методы QcObsBean. При их выполнении будут приостанавливаться все транзакции, инициированные объектом QcClient, а возобновятся они после завершения методов. Заметим, что здесь для повышения эффективности работы можно не создавать новый сеанс связи с базой данных, а воспользоваться sess_iiop://thisServer/:thisSession в качестве URL (см. раздел "Серверный объект, обращающийся к тому же сеансу" этой главы).

5. Этот оператор создает строку символов (String) Java, содержащую опубликованное имя объекта /test/QcObsBean.
6. Этот оператор использует обозначение sess_iiop://thisServer в качестве URL, объединенного с именем /test/QcObsBean, получая ссылку на домашний интерфейс зерна. Напомним, что этот URL создает новый сеанс базы данных. Теперь существуют два сеанса БД: первый был установлен приложением QcClient при вызове им QcSetBean, а второй начался при выполнении EJB-контейнером метода ejbCreate() класса QcSetImp.
7. Этот оператор использует глобальную переменную qcObsObj для сохранения экземпляра интерфейса QcObsRemote. Через объект qcObsObj экземпляр сеансового зерна QcObsBean становится видимым на уровне класса, т.е. становится доступным исключительно всем методам зерна QcSetBean.
8. Этот оператор описывает метод setSessionContext(), с помощью которого мы получаем объект SessionContext от EJB-контейнера и сохраняем его в переменной экземпляра.

Файл дескриптора внедрения QcSet.ejb

В файле QcSet.ejb содержатся инструкции по внедрению зерна QcSetBean, а в файле QcObs.ejb — информация о внедрении зерна QcObsBean. Вот описание файла QcSet.ejb:

```
// Имя файла: QcSet.ejb (см. пояснение 1)
// Дескриптор внедрения зерна QcSetBean.
SessionBean obsserver.QcSetImp {
BeanHomeName = "test/QcSetBean";
   HomeInterfaceClassName = obsserver.QcSetHome;
   RemoteInterfaceClassName = obsserver.QcSetRemote;
   AllowedIdentities = {PUBLIC};
   SessionTimeout = 0;
   StateManagementType = STATEFUL_SESSION;
   RunAsMode = CLIENT_IDENTITY;
   TransactionAttribute = TX_SUPPORTS; // (см. пояснение 2)
 } // Конец obsserver.QcSetImp
```

Файл дескриптора внедрения QcObs.ejb

Приведем описание файла QcObs.ejb:

```
// Имя файла: QcObs.ejb (см. пояснение 3)
  // Дескриптор внедрения зерна QcObsBean.
  SessionBean qcobsserver.QcObsImp {
    BeanHomeName = "test/QcObsBean";
    HomeInterfaceClassName = qcobsserver.QcObsHome;
    RemoteInterfaceClassName = qcobsserver.QcObsRemote;
    AllowedIdentities = {PUBLIC};
    SessionTimeout = 0;
    StateManagementType = STATEFUL_SESSION;
    RunAsMode = CLIENT_IDENTITY;
    TransactionAttribute = TX_REQUIRED_NEW; // (см. пояснение 4)
  }
```

Пояснения к файлам `QcSet.ejb` **и** `QcObs.ejb`**:**

1. Класс `obsserver.QcSetImp SessionBean` описывает файл дескриптора для `QcSet.ejb` для сеансового зерна `QcSetBean`.
2. Этот оператор объявляет транзакционную политику всего зерна `QcSetBean`. Напомним, что атрибут транзакций `TX_SUPPORT` можно вызывать в предоставляемом клиентом контексте транзакций. Если клиент не предоставляет последнего, оператор вызывает метод зерна без контекста.
3. Класс `qcobsserver.QcObsImp SessionBean` описывает файл дескриптора `QcObs.ejb` для сеансового зерна `QcObsBean`.
4. Этот оператор объявляет транзакционную политику всего зерна `QcObsBean`. Зерна EJB с атрибутом `TX_REQUIRES_NEW` должны всегда вызываться в новом контексте транзакций, независимо от того, предоставляет ли клиент контекст транзакций или нет. В любом случае EJB-контейнер создает новый контекст, приостанавливая действие контекста транзакций клиента до выполнения транзакции зерна.

▼ ВНИМАНИЕ! *Порядок внедрения сеансовых зерен `QcSetBean` и `QcObsBean` очень важен. Зерно `QcSetBean` необходимо внедрять до внедрения зерна `QcObsBean`. Если поменять порядок на обратный, то во время вызова зерно `QcSetBean` не сможет найти домашний интерфейс зерна `QcObsBean`.*

Транзакционное приложение `QcClient`

Класс `QcClient` – это автономное транзакционное клиентское приложение Java, которое:

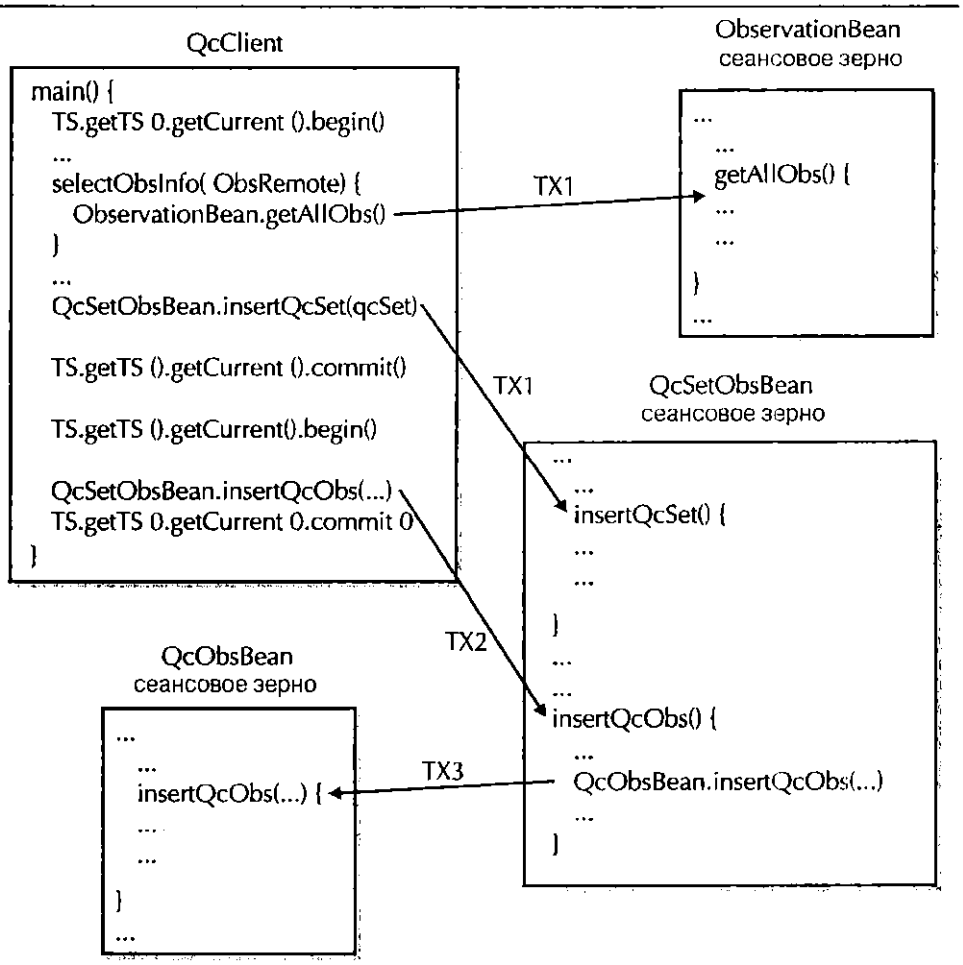
- Инициализирует службу транзакций клиента, вызывая метод `initialize()` класса `AuroraTransactionService`.
- Ограничивает и начинает транзакцию (`TX1`), вызывая JTS-метод `begin()`. На рис. 5.4 эта транзакция называется `TX1`.
- Получает экземпляры сеансовых зерен `ObservationBean` и `QcSetBean`.
- С помощью контекста транзакции `TX1`, создаваемого на шаге 2, по порядку вызывает метод `getAllObs()` зерна `ObservationBean` и метод `insertQcSet()` зерна `QcSetBean` (обратите внимание на то, что во время вызова метода `getAllObs()` контекст транзакции `TX1` распространяется на зерно `ObservationBean`; когда `QcClient` вызывает метод `insertQcSet()`, `TX1` распространяется и на зерно `QcSetBean`).
- Прекращает транзакцию `TX1`, начатую на шаге 2, вызывая JTS-метод `commit()`.
- Ограничивает и начинает новую транзакцию (`TX2`), снова вызывая JTS-метод `begin()`.
- С помощью контекста транзакции `TX2`, созданного на шаге 2, вызывает метод `insertQcObs()` зерна `QcSetBean`, который, в свою очередь, вызывает метод `insertQcObs()` зерна `QcObsBean`; когда сеансовое зерно `QcSetBean` вызывает метод `insert QcObs()` зерна `QcObsBean`, EJB-контейнер начинает новую транзакцию (называемую `TX3`).

На рис. 5.4 показаны различные области действия транзакций, в которых вызываются зерна.

Пояснения к рис. 5.4:

1. В локальном методе `selectObsInfo()` экземпляр зерна `ObservationBean` (обозначен как `ObservationBean`) используется для вызова своего метода `getAllObs()`.

Рис. 5.4. Области действия транзакций



2. Экземпляр зерна `QcSetBean` (обозначен как `QcSetBean`) применяется для вызова своих методов `insertQcSet()` и `insertQcObs()`, а экземпляр `QcObsBean` (обозначен как `QcObsBean`) — для вызова своего метода `insertQcObs()`.

Программа `QcClient` — это типичное клиентское приложение с явным ограничением транзакций, реализуемым программным путем. В приложениях, ограниченных на стороне клиента, транзакционный клиент явно охватывает один или несколько вызовов конкретного зерна с помощью методов ограничения `begin()` и `commit()` (или `rollback()`), которые начинают и заканчивают транзакции. Начальными и конечными *ограничителями* (demarcators) являются вызовы методов клиентской службы транзакций. Когда клиент открывает транзакционный контекст, тот передается от клиентского объекта серверному или от одного серверного объекта другому. Этот контекст переносит состояние транзакции. После инициализации клиентская служба транзакций неявно создает контекст, присваивая ему транзакционный идентификационный номер. Затем служба транзакций клиента распространяет контекст на каждого участника, т.е. на каждый объект EJB, вызываемый клиентом.

На стороне клиента контекст транзакций перехватывается, а затем переносится на каждый вызов метода серверного объекта. На стороне сервера транзакционный контекст извлекается и становится доступным для серверного объекта. В приложении `QcClient` для инициализации службы транзакций используется метод `initialize()` класса `AuroraTransactionService`. После инициализации служба транзакций неявно создает транзакционный контекст. Затем транзакция ограничивается выполнением соответствующего метода `begin()`, который начинает глобальную транзакцию и связывает ее с вызывающим потоком. Отметим, что связь транзакция-поток прозрачна. После открытия

транзакционного контекста служба транзакций QcClient распространяет его на зерна ObservationBean и QcSetBean. Приведем описание класса QcClient:

```
/* Имя программы:    QcClient.java
** Версия:
** Автор:            Ваше имя
** Описание:         Клиентское приложение Java, позволяющее
**                   пользователю получить список наблюдений
**                   за атмосферой, проверить их качество и
**                   одновременно ввести новую строку в таблицу
**                   QC_EVENT_LIST и таблицу
**                   PASSED_OBSERVATION_LIST. После ввода клиент
**                   вызывает методы, необходимые для отображения
**                   информации из таблицы PASSED_OBSERVATION_LIST.
**
*/
// Импортируем helper-классы
import helpers.QcSetHelper;
import helpers.ObsHelper;
import helpers.QcObsHelper;

// Импортируем объекты obsserver
import obsserver.ObsRemote;
import obsserver.ObsHome;
import obsserver.QcSetRemote;
import obsserver.QcSetHome;
import helpers.ObsException;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// (см. пояснение 1)
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.TS;

import javax.naming.Context;

import javax.naming.InitialContext;

// Импортируем вспомогательные классы Java
import java.math.BigDecimal;
import java.math.*;
import java.util.ArrayList;
import java.util.Hashtable;

public class QcClient {

  public static void main (String[] args) throws Exception {
    if (args.length != 5) {
        System.out.println ("usage: QcClient serviceURL "
        +" objectName user password");
        System. exit (1);
    }

    String serviceURL = args [0];
    String obsName = args [1]; // (см. пояснение 2)
    String qcSetName = args [2];
    String user = args [3];
    String password = args [4];

    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION,
                          ServiceCtx.NON_SSL_LOGIN);
    // Получим описатель для InitialContext
```

```
Context ic = new InitialContext (env);

// Получим описатель для источника транзакционных контекстов,
// или TX-Factory (см. пояснение 3)
AuroraTransactionService.initialize (ic, serviceURL);
ObsHome observationHome = null;
ObsRemote observationRemote = null;
QcSetHome qualityControlHome = null;
QcSetRemote qualityControlRemote = null;
try {
      // Ограничим TX (названный TX1 на рис. 5.4) (см. пояснение 4)
      TS.getTS ().getCurrent ().begin ();

      // С помощью домашнего интерфейса найдем объект
      // ObservationBean
      // (см. пояснение 5)
      observationHome =
        (ObsHome)ic.lookup (serviceURL + obsName);

      // Создадим экземпляр ObservationBean
      observationRemote = observationHome.create();
      // Получим сведения о наблюдениях (см. пояснение 6)
      // TX1 распространяется на ObservationBean.
      ArrayList qcObsIds = selectObsInfo (observationRemote);

      // С помощью домашнего интерфейса найдем объект QcObsBean
      qualityControlHome =
        (QcSetHome)ic.lookup (serviceURL + qcSetName);

      // Создадим экземпляр QcSetBean (см. пояснение 7)
      qualityControlRemote = qualityControlHome.create();

      // Установим переменные для создания объекта QcSetHelper
      BigDecimal qc_id = null;
      String when_t = "04-AUG-2000";
      String at_time = "085500";
      BigDecimal event_id = new BigDecimal(1);
      BigDecimal whom_id = new BigDecimal(1);

      // Воспользуемся конструктором с параметрами класса
      // QcSetHelper для создания объекта QcSetHelper
      QcSetHelper qcSet =
        new QcSetHelper(qc_id,when_t,at_time,event_id,whom_id);

      // Вызовем метод insertQcSet() сеансового зерна QcSetBean
      // (см. пояснение 8).
      // TX1 распространяется на QcSetBean.
      QcSetHelper returnQcSetId =
        qualityControlRemote.insertQcSet(qcSet);

      // (см. пояснение 9)
      // Зафиксируем изменения, внесенные во время транзакции TX1
      TS.getTS ().getCurrent ().commit (true);

      // Начнем новую транзакцию с именем TX2, которая связана
      // с QcSet из предыдущей транзакции (см. пояснение 10).
      TS.getTS ().getCurrent ().begin ();

      // TX2 распространяется на QcSetBean вызовом его метода insertQcObs().
      // Введем новую строку в таблицу PASSED_OBSERVATION_LIST
      qualityControlRemote.insertQcObs(returnQcSetId.qc_id,
                                       qcObsIds);

      // Получим все проверенные наблюдения для конкретного
      // идентификатора QcSet (см. пояснение 11)
```

```
            printQcObservations(returnQcSetId,qualityControlRemote);

            // Зафиксируем изменения

            // Значение параметра истинно (см. пояснение 12)
            TS.getTS ().getCurrent ().commit (true);

        } // Конец try ObsException
        catch (ObsException ex ) {
            System.err.println(ex.getMessage());
        } // Конец catch
        catch ( Exception ex ) {
            System.out.println("Cannot locate" +" or create object");
            System.err.println("error : " + ex);
            System.exit(1);
        } // Конец catch
    } // Конец main

    // (см. пояснение 13)
    public static ArrayList selectObsInfo (ObsRemote observationRemote)
          throws Exception {
        try {
            // Получим ArrayList наблюдений (см. пояснение 14)
            ArrayList arrayOfObs = observationRemote.getAllObs();

            // Выберем только два наблюдения (см. пояснение 15)
            ArrayList returnArrayOfObs = new ArrayList(2);

            // Создадим переменную obs1 типа ObsHelper
            ObsHelper obs1 = null;

            int arraySize = arrayOfObs.size();

            // Считаем идентификатор obs_id из ArrayList
            obs1 = (ObsHelper)arrayOfObs.get(arraySize - 1);

            // Получим последнее наблюдение из объекта ArrayList
            returnArrayOfObs.add(obs1.obs_id);

            // Получим первое наблюдение из объекта ArrayList
            obs1 = (ObsHelper)arrayOfObs.get(arraySize - arraySize);
            returnArrayOfObs.add(obs1.obs_id);

            return returnArrayOfObs;

        } // Конец try
        catch (SQLException e) {
            System.err.println("selectObsInfo : " + e.getMessage());
            e.printStackTrace ();
            System.exit(1);
        } // Конец catch
    } // Конец метода selectObsInfo()

    // (см. пояснение 16)
    public static void printQcObservations(QcSetHelper qcSet,
       QcSetRemote qualityControlRemote ) throws Exception {
        try {
            // Получим все сведения для конкретного qc_id
            int qc_id = qcSet.qc_id.intValue();
            // (см. пояснение 17)
            QcSetHelper returnQcSet =
               qualityControlRemote.getQcSetInfo(qcSet.qc_id);

            // Получим все проверенные наблюдения для указанного
            // выше qc_id (см. пояснение 18)
            ArrayList returnQcObs =
```

```
            qualityControlRemote.getQcObsInfo(qc_id);

          // Распечатаем информацию о QcSet
          System.out.println ("QcSet info: ");
          System.out.println ("QcSet id: " +returnQcSet.qc_id);
          System.out.println ("Date: " +returnQcSet.when_t);
          System.out.println ("at_time: " +returnQcSet.at_time);
          System.out.println ("Storm id: " +returnQcSet.event_id );
          System.out.println ("Scientist id: "
                      +returnQcSet.whom_id);
          System.out.println ("Storm Name: " +returnQcSet.evt_name);
          System.out.println ("Scientist Name:
                      +returnQcSet.sc_name);
          System.out.println ("********************");
          System.out.println ("Quality Controlled Observations"
            +" for QcSet id: " +returnQcSet.qc_id);
          System.out.println (" ");
          int loopVar = 0;
          int arraySize = returnQcObs.size();
          while ( loopVar < arraySize ) {
            // Считаем объект ObsHelper из ArrayList returnQcObs
            // (см. пояснение 19).
            ObsHelper obs = (ObsHelper)returnQcObs.get(loopVar);
            System.out.println("id: " +obs.obs_id);
            System.out.println("when_t: " +obs.when_t);
            System.out.println("at_time: " +obs.at_time);
            System.out.println("station_id: " +obs.station_id);
            System.out.println("produced_id: " +obs.produced_id);
            System.out.println("latitude: " +obs.latitude);
            System.out.println("longitude: " +obs.longitude);
            System.out.println("wdspd: " +obs.wdspd);
            System.out.println("adj_wdspd: " +obs.adj_wdspd);
            System.out.println("wddir: " +obs.wddir);
            System.out.println("pressure: " +obs.pressure);

            loopVar++;
          } // Конец while
        } // Конец try
      } // Конец метода printQcObservations()
} // Конец класса QcClient
```

Пояснения к приложению `QcClient`:

1. Этот и следующий операторы импортируют нужные классы соответственно из пакетов `oracle.aurora.jts.client.*` и `oracle.aurora.jts.util.*`. Для ограничения транзакций в автономных клиентах Java вам необходимо импортировать в свою программу классы `AuroraTransactionService` и `TS`, но это касается только баз данных Oracle8*i* версий 8.1.5 и 8.1.6. В новой редакции РСУБД Oracle версии 8.1.7 нужно будет импортировать в программу Java Transaction API (JTA) (`javax.transaction.*`).
2. Этот и следующий операторы считывают опубликованные имена объектов сеансовых зерен, используемых клиентом, т.е. соответственно `/test/ObservationBean` и `/test/QcSetBean`.
3. Здесь программа `QcClient` инициализирует клиентскую транзакционную службу. Чтобы ограничить транзакцию явным образом, клиенты должны инициализировать службу транзакций до начала транзакции. Если служба транзакций не инициализирована, вызов метода `begin()` не удастся, и установится исключительная ситуация `NotTransactionService`. Во время инициализации транзакции эта служба неявно создает объект транзакционного контекста и автоматически присваивает ему идентификационный номер. Такой контекст будет распространен на все остальные зерна, участвующие в транзакции (т.е. на каждый

объект EJB, вызываемый приложением QcClient). В этом случае служба транзакций QcClient распространит контекст транзакции на сеансовые зерна ObservationBean и QcSetBean.

4. Этот оператор явно ограничивает транзакцию вызовом метода begin() из класса oracle.aurora.jts.util.TS. После инициализации службы транзакций можно вызвать статический метод getCurrent() и получить текущий транзакционный контекст. Получив его, можно вызвать, начать, приостановить, возобновить, завершить и откатить текущую транзакцию в текущем транзакционном контексте. Во время выполнения этого метода служба транзакций начинает глобальную транзакцию и прозрачно управляется EJB-контейнером. Обратите внимание: транзакция начинается до считывания зерен, благодаря чему зерна автоматически становятся ее участниками. Зерна ObservationBean и QcSetBean являются частью одной и той же транзакции, начиная с момента своего создания до вызова их методов и до выполнения метода commit() или rollback(). Идея заключается в том, чтобы скомпоновать приложение в границах транзакции и гарантировать тем самым, что все обновляемые данные будут или зафиксированы, или отменены.

5. Этот оператор с помощью JNDI-метода lookup() получает ссылку на домашний интерфейс ObsHome. Следующий оператор получает экземпляр зерна ObservationBean.

6. Этот оператор вызывает метод selectObsInfo() (локальный метод приложения QcClient). Он использует объект observationRemote в качестве входного параметра и возвращает приложению QcClient объект ArrayList, являющийся набором объектов ObsHelper, каждый из которых представляет собой строку наблюдений за атмосферой из таблицы OCEANIC_OBSERVATION_LIST. Отметим, что в случае отсутствия наблюдений метод selectObInfo() устанавливает ObsException, и дальнейшая обработка информации не производится.

7. Этот оператор создает экземпляр зерна QcSetBean. Когда приложение QcClient вызывает метод create() домашнего интерфейса QcSetBean, EJB-контейнер обращается к соответствующему методу ejbCreate() класса QcSetImp. Затем метод ejbCreate() класса QcSetImp создает экземпляр QcObsBean. Таким образом, при вызове EJB-контейнером метода ejbCreate() класса QcSetImp создаются экземпляры зерен QcSetBean и QcObsBean.

8. Этот оператор вызывает метод insertQcSet() зерна QcSetBean. Текущий транзакционный контекст, созданный ранее, распространяется на зерно QcSetBean.

9. Метод-ограничитель commit() фиксирует изменения в базе данных и заканчивает транзакцию. Отметим, что метод commit() заканчивает транзакцию и принимает входной параметр с истинным (true) значением, если нужно зафиксировать изменения в БД. Если же значение параметра ложно (false), транзакция будет прервана без обновления базы.

10. Этот оператор начинает новую транзакцию. Данный пример показывает, как начинать несколько транзакций в одной программе. Служба транзакций приложения QcClient создает новый транзакционный контекст, который распространяется на QcSetBean. Следующий оператор вызывает метод insertQcObs() зерна QcSetBean, который, в свою очередь, вызывает метод insertQcObs() зерна QcObsBean. Напомним, что это сеансовое зерно с атрибутом TX_REQUIRED_NEW. Следовательно, EJB-контейнер будет всегда начинать новую транзакцию, независимо от того, использует ли клиент зерно QcObsBean или нет.

11. Этот оператор считывает набор наблюдений за атмосферой и отображает их.

12. Этот оператор заканчивает последнюю транзакцию, начало которой описано в пояснении 10.

13. Этот оператор описывает метод selectObsInfo(), вызывающий метод getAllOBS() сеансового зерна ObservationBean.
14. Этот оператор вызывает метод getAllObs() зерна ObservationBean. Если не возвращает никаких данных, устанавливается исключительная ситуация ObsException, и дальнейшая обработка информации не производится.
15. Этот оператор создает объект ArrayList для хранения проверенных наблюдений. Остальные операторы выбирают идентификаторы наблюдений и сохраняют их в объекте ArrayList.
16. Этот оператор описывает метод printQcObservations(), который вызывает методы getQcSetInfo() and getQcObsInfo() сеансовых зерен QcSetBean и QcObsBean соответственно.
17. Этот оператор вызывает метод getQcSetInfo() зерна QcSetBean и возвращает данные из таблицы QC_EVENT_LIST по конкретному qc_id.
18. Этот оператор вызывает метод getQcObsInfo() сеансового зерна QcSetBean. Обратите внимание на то, что метод getQcObsInfo(), в свою очередь, вызывает метод getQcObs() зерна QcObsBean. Метод getQcObs() возвращает объект ArrayList из объектов ObsHelper, каждый из которых представляет данные таблицы OCEANIC_OBSERVATION_LIST.
19. Этот оператор переносит каждый объект из объекта ArrayList в объект ObsHelper.

Итоги

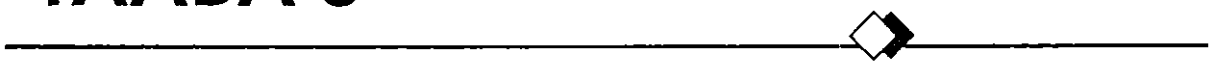
Данная глава завершает часть II книги. Здесь мы расскажем об использовании API службы транзакций Java (Java Transaction Service) для построения надежных транзакционных приложений. Были разработаны сеансовые зерна QcObsBean и QcSetBean. Первое работает с таблицей PASSED_OBSERVATION_LIST, а второе — с таблицей QC_EVENT_LIST. Более того, построенное автономное транзакционное клиентское Java-приложение QcClient использует три сеансовых зерна, находящихся в базе данных Oracle8*i*, причем сеансовое зерно QcSetBean является EJB-клиентом, работающим с другим зерном (QcObsBean).

В части III мы рассмотрим построение компонентов CORBA, выполняющихся в базе данных Oracle8*i*.

ЧАСТЬ III

Построение компонентов CORBA

ГЛАВА 6



Знакомство с CORBA

В этой части книги рассказано о способах применения CORBA в среде Oracle8*i*. В настоящей главе изложены основы CORBA, в частности:

- Общее представление о CORBA
- IDL
- Внедрение и использование серверных объектов CORBA
- Механизм обратных вызовов CORBA
- Механизм привязки CORBA
- Активация брокеров ORB и серверных объектов
- Создание спецификаций интерфейсов CORBA на Java
- Общие сведения об отдельных важных инструментах

В главе 7 рассматривается расширение в Oracle8*i* стандартной модели CORBA концепцией сеансов связи с базой данных, в которых активируются объекты CORBA.

В главе 8 рассказано о различных способах введения объектов CORBA в транзакции, что гарантирует сохранение согласованности информации.

Наконец, в главе 9 полученные знания будут применены в реализации компонентов с использованием серверных объектов CORBA. Будет построен ряд компонентов CORBA для базы данных Purchase Order.

Обзор CORBA

Общая архитектура посредника объектных запросов (CORBA, Common Object Request Broker Architecture) – это стандарт, принятый группой Object Management Group (OMG) для распределенных и неоднородных сетей объектов. Распределенной называется сеть, где объекты могут находиться на разных компьютерах, а неоднородной (гетерогенной) – сеть, где на компьютерах могут функционировать разные операционные системы, а объекты реализуются на разных языках программирования.

Интерфейсы для объектов создаются на нейтральном по отношению к языкам программирования языке описания интерфейсов (IDL, Interface Definition Language), который является частью стандарта CORBA.

Клиентом CORBA называется любая программная единица, задействующая объекты CORBA, предоставляемые по сети (*серверные объекты* CORBA).

Посредник объектных запросов (ORB, Object Request Broker) CORBA – это программная единица, которая является связующим звеном, перенаправляющим запросы клиентов CORBA серверным объектам CORBA. ORB делает все подробности обработки запросов прозрачными для клиентов. Например, вызовы методов, обращенные к удаленным серверным объектам сети, ничем не отличаются от вызова локальных методов. ORB – это фундамент архитектуры CORBA, управляющий всеми связями между клиентом и серверным объектом. В Oracle8*i* система ORB входит в состав JServer, который основан на Inprise's VisiBroker для Java. Существуют ORB, выполняющийся на стороне клиента, и ORB, который выполняется на стороне сервера. В Oracle8*i* запуск и работа ORB во многом прозрачны для реализации как клиентской, так и серверных программ.

В ORB реализуется набор *полностью определенных служб* (services), упрощающих работу в среде CORBA. Так, *службы именования* (naming services) позволяют клиентам без проблем ссылаться на серверные объекты, *службы транзакций* (transaction services) – разделять клиентские и серверные объекты на атомарные единицы, называемые *транзакциями* (transactions), а *службы событий* (event services) – делать доступными сведения о событиях, причем источник данных не обязан знать получателей. CORBA-клиенты Oracle8*i* в качестве интерфейса для поиска серверных объектов через CORBA-службу CosNaming применяют интерфейс именования и каталогов Java (JNDI, Java Naming and

Directory Interface). Кроме того, в Oracle8*i* предлагается интерфейс поставщика услуг (SPI, Service Provider Interface) JNDI, ориентированный на службу CosNaming. Транзакционные служебные интерфейсы API, применяемые в Oracle8i для CORBA, – это API службы транзакций Java (JTS, Java Transaction Service) и транзакционный API Java (JTA, Java Transaction API), по существу, заменивший JTS в Oracle версии 8.1.7. Об использовании JNDI говорится в этой главе, а более подробно – в главе 7. JTS и JTA обсуждаются в главе 8.

Помимо этого, для передачи ORB-запросов по сети TCP/IP в CORBA определен сетевой протокол Internet Inter-ORB Protocol (IIOP). О его применении рассказано в следующей главе.

IDL

Теперь мы поговорим о:

- Компиляторе IDL
- Конструкциях IDL

Компилятор IDL

Одним из главных достоинств CORBA является ее языковая независимость. Объекты CORBA, написанные на одном языке программирования, могут обращаться к методам объекта, написанного на другом. Такая независимость достигается благодаря использованию нейтрального в языковом отношении языка описания интерфейсов (IDL, Interface Definition Language). В нем не нужно реализовывать методы объектов – их следует лишь описать. Точнее, для каждого серверного объекта создается *IDL-интерфейс*, в котором перечисляются константы, типы, исключительные ситуации, атрибуты и методы, связанные с объектом. IDL был задуман так, чтобы соответствовать языку C++. Затем IDL-файл с интерфейсом компилируется, после чего интерфейс реализуется на конкретном языке программирования, например на Java. Для работы с Oracle8*i* используется IDL-компилятор `idl2java`, превращающий интерфейсы IDL в интерфейсы Java. После компиляции IDL-интерфейса полученный в результате Java-интерфейс реализуется в виде серверного объекта. Ниже приведен простой пример, иллюстрирующий процедуру компиляции и реализации. Базовой единицей компиляции IDL является *модуль* (module) – логически внутренне связанная совокупность описаний интерфейсов (и других описаний). Указанный ниже модуль состоит из одного интерфейса, который содержит один метод, не принимающий никаких аргументов и возвращающий строку символов:

```
/* Имя файла:    cool.idl
**
** Назначение:   В реализации cool() будет возвращать строку
**               символов "Java is cool."
**
*/
module cool { interface Cool { wstring isCool(); }; };
```

wstring (wide string – широкая строка) – это IDL-тип, соответствующий Java-типу **String**.

После компиляции `cool.idl` с помощью `idl2java` будут созданы восемь исходных файлов Java, располагающиеся в подкаталоге с именем `cool` в том же каталоге, что и `cool.idl`. Таким образом, впоследствии эти восемь исходных файлов Java составят пакет Java с именем `cool`. Перечислим их:

- `Cool.java`
- `CoolHolder.java`
- `CoolHelper.java`
- `_st_Cool.java`

- CoolImplBase.java
- CoolOperations.java
- _tie_Cool.java
- _example_Cool.java

Cool.java — это файл интерфейса:

```
package cool;
  public interface Cool
    extends org.omg.CORBA.Object {
       public java.lang.String isCool(); }
```

Обратите внимание: все базовые интерфейсные классы CORBA происходят из org.omg.CORBA.Object.

Для каждого интерфейсного класса модуля создается holder-класс (класс-хранитель) и helper-класс (класс-помощник). Первый используется в качестве оболочки, когда режим параметров метода интерфейса — *out* или *inout*.

Helper-класс применяется для чтения объекта из потока и для записи его в поток.

В *stub-файле* _st_Cool.java описывается *stub-класс*, устанавливаемый на станции клиента и выступающий в роли интерфейса связи клиента с удаленным объектом.

В скелетном файле _CoolImplBase.java описывается *скелетный класс* — серверный класс, взаимодействующий с клиентским stub-классом. Классы изолятора и скелета скрывают подробности удаленного вызова от разработчика, упрощая его задачу.

В файле _example_cool.java содержится пример реализации интерфейса Cool. Естественно, в нем нет тела метода isCool(). Этот файл можно переименовать (скажем, в CoolImpl.java), включив в его состав программную конструкцию, реализующую метод isCool(). Приведем файл _example_cool.java, автоматически генерируемый компилятором idl2java, и пример файла CoolImpl.java:

```
/* Имя файла: _example_Cool.java */

package cool;
public class _example_Cool extends _CoolImplBase {
  public _example_Cool( java.lang.String name ) { super( name ); }
  public _example_Cool() { super(); }
  public java.lang.String isCool() {}
}
```

```
/* Имя программы:    CoolImpl.java
**
** Назначение:       Реализует серверный объект.
**
*/
/* Создатель решил сохранить класс реализации в пакете с именем coolServer.
*/
package coolServer;
import cool.*;
public class CoolEmpl extends _CoolImplBase {
   public String isCool() { return "Java is cool."; }
}
```

Обратите внимание: класс реализации должен расширять скелетный класс.

Класс CoolOperations и _tie_Cool используются сервером для реализации привязки серверных объектов (см. раздел "Механизм привязки CORBA" этой главы).

Затем описанные выше классы Java необходимо скомпилировать при помощи javac (или sqlj), после чего соответствующие файлы классов должны быть загружены в JServer, а серверный объект должен быть опубликован, чтобы к нему могли обращаться клиенты (см. раздел "Внедрение и использование серверных объектов CORBA" этой главы).

В следующем разделе мы расскажем, каким образом конструкции IDL преобразуются в конструкции Java.

Конструкции IDL

Итак, мы поговорим о:

- Типах IDL
- Интерфейсах IDL
- Генерируемых holder- и helper-классах
- Исключительных ситуациях IDL
- Наследовании
- Модулях IDL

Типы IDL

В настоящем разделе описаны:

- Базовые типы IDL
- *Структуры* IDL
- *Перечни* IDL
- Объединения IDL
- *Последовательности*, *массивы* и *описания типов* IDL.

Базовые типы Соответствие базовых типов IDL и Java с указанием исключительных ситуаций CORBA, устанавливаемых при преобразовании этих типов, представлено вкратце в таблице 6.1 (взята из книги "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 3-11, 3-12, таблица 3.1]).

Таблица 6.1. *Соответствие типов данных IDL и Java*

IDL-тип данных CORBA	Тип данных Java	Исключительная ситуация
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	
octet	byte	
string	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	
unsigned short	short	
long	int	
unsigned long	long	
long long	long	
unsigned long long	long	
float	float	
double	double	

Структуры В IDL существуют структурные типы трех видов: *структура* (struct), *перечень* (enum) и *объединение* (union). О *структурах* речь пойдет в этом разделе, о *перечнях* и *объединениях* — в следующих двух.

IDL-*структура* аналогична структуре (struct) C++, в которой содержатся лишь элементы данных (т.е. элементов-методов она не содержит) и все элементы являются общими. IDL-*структуры* преобразуются в классы Java с полями для элементов *структуры* и двух конструкторов. Рассмотрим следующий IDL-модуль, который содержит IDL-*структуру* со сведениями о номере отдела (department number) типа *long* и о номере проекта (project number) также типа *long*:

```
module account {
  struct AccountInfo { long departmentno; long projectno; };
};
```

С помощью idl2java этот модуль превращается в следующий исходный файл Java:

```
package account;
final public class AccountInfo {
  public int departmentno;
  public int projectno;
  public AccountInfo() {}
  public AccountInfo ( int departmentno, int projectno ) {
    this.departmentno = departmentno;
    this.projectno = projectno;
  }
}
```

Первый конструктор позволяет клиенту создать новый неинициализированный экземпляр AccountInfo, а второй — создать и инициализировать новый экземпляр AccountInfo.

Перечни *Перечни* (перечисляемые IDL-типы) — это пользовательские типы, в которых пользователь указывает набор идентификаторов, составляющих значения типов. Например, в следующем модуле объявляется перечисляемый тип employee:

```
module hit {
  enum employee { watchman, janitor, manager, ceo };
};
```

Подобный тип *enum* преобразуется в конечный (final) класс Java с тем же именем, что и у *перечня*, и парой статических полей для каждого элемента *enum*. Одно из них имеет тип final int и используется в качестве индексного типа, например, когда тип *enum* используется как массив индексного типа. Тип другого поля является типом генерируемого класса Java и предназначен для передачи параметров. Помимо этого, здесь присутствует общий метод value(), который возвращает целое число (int), соответствующее текущему значению *перечня*, и конструктор, который устанавливает int, соответствующее текущему значению *перечня*. Кроме того, формируется общий статический метод from_int(), преобразующий значение int в соответствующий экземпляр генерируемого класса. Например, приведенный выше IDL-тип employee транслируется в следующий Java-класс employee:

```
final public class Employee {
  final public static int _watchman = 0;
  final public static int _janitor = 1;
  final public static int _manager = 2;
  final public static int _ceo = 3;
  final public static Employee watchman = new Employee( _watchman );
  final public static Employee janitor = new Employee( _janitor );
  final public static Employee manager = new Employee( _manager );
  final public static Employee ceo = new Employee( _ceo );
  private int_value;
  private Employee ( int value ) { _value = value; }
  public int value(){
```

```
        return_value;}
  public static Employee from_int( int $value ) {
    switch( $value ) {
        case _watchman: return watchman;
        case _janitor: return janitor;
        case _manager: return manager;
        case _ceo: return ceo;
        default: throw new org.omg.CORBA.BAD_PARAM ( "enum out of range:
                + (4 - 1) + "]: " + $value );
    }
  }
}
```

Объединения *Объединение* (селективный объединительный IDL-тип) позволяет связывать несколько типов в один, причем на тип, используемый в ссылке на конкретный экземпляр объединения, указывает специальный разделитель – дискриминатор (discriminator). Например, следующее *объединение* связывает типы *float* и *long*, а дискриминатор *long* указывает, что использовать – *float* или *long*:

```
union FloatorInt switch( long ) {
  case 0: float floatval;
  case 1: long intval;
};
```

Типы дискриминатора: *integer*, *char*, *boolean* или *enum*.

IDL-*объединение* преобразуется в конечный (final) класс Java с тем же именем, что и IDL-*объединение*, с null-конструктором, методом доступа (accessor) для дискриминатора, методом доступа для каждого оператора выбора, который возвращает значение объекта *union*, интерпретируемое как имеющее тип данного оператора, а также с методами-модификаторами для каждого из двух операторов выбора, присваивающих объекту *union* значение с типом данного оператора. Например, приведенный выше тип *union* транслируется в следующий класс Java :

```
final public class FloatorInt {

  // Текущее значение объекта union.
  private java.lang.Object _object;

  // Текущее значение дискриминатора.
  private long _disc;

  // null-конструктор.
  public FloatorInt() {}

  // Метод доступа дискриминатора.
  public long discriminator() { return _disc; }

  // Метод доступа для значения float.
  public float floatval() {
    if ( _disc != 0 ) {
      throw new org.omg.CORBA.BAD_OPERATION( "floatval" );
    }
    return( ( ( java.lang.Float ) _object).floatValue() );
  }

  // Метод доступа для значения int.
  public int intval() {
    if ( _disc != 1 ) {
      throw new org.omg.CORBA.BAD_OPERATION( "intval" );
    }
    return( ( ( java.lang.Int ) _object ).intValue() );
  }
```

```
  // Метод-модификатор для значения int.
  public void intvalue( int value ) {
    _disc = 1;
    _object = new java.lang.Integer( value );
  }

  /* Метод-модификатор для значения int, в котором устанавливаемое
     значение дискриминатора передается явным образом.
  */
  public void intval( int disc, int value ) {
    _disc = disc;
    _object = new java.lang.Integer( value );
  }

  // Метод-модификатор для значения float.
  public void floatval ( float value ){
    _disc = 0;
    _object = new java.lang.Float ( value ));

  /* Метод-модификатор для значения float, в котором устанавливаемое
     значение дискриминатора передается явным образом.
  */
  public void floatval( int disc, float value ) {
    _disc = disc;
    _object = new java.lang.Float( value );
  }
}
```

Последовательности, массивы и описания типов В IDL поддерживаются упорядоченные сборные конструкции двух типов: *последовательности* и *массивы*. IDL-*последовательность* (IDL sequence) — это одномерный массив, размер которого не нужно указывать в IDL-описании. IDL-*массив* (IDL array) — это многомерный массив, размеры которого указываются в IDL-описании. В обоих случаях для именования конкретных типов *sequence* и *array* используются IDL-*описания типов* (IDL typedefs). Как и в языке C, эти имена являются псевдонимами появляющихся в них типов. В следующем IDL-модуле содержатся типы *sequence* и *array*:

```
/* Имя файла: accounts.idl */

module accounts {
  struct AccountInfo { long departmentno; long projectno; }
  typedef sequence <AccountInfo> accountinfos;
  const long bound = 100;
  typedef AccountInfo AccountInfos2[bound];
  struct MyandYourAccounts {
    accountinfos myaccounts; accountinfos2 youraccounts;
  }
}
```

Для *структуры* MyandYourAccounts программа idl2java сгенерирует следующий исходный файл Java:

```
package accounts;
final public class MyandYourAccounts {
  public AccountInfo[] myaccounts;
  public AccountInfo[] youraccounts;
  public MyandYourAccounts() {}
  public MyandYourAccounts (
    AccountInfo[] myaccounts, AccountInfo[] youraccounts ) {
    this.myaccounts = myaccounts;
    this.youraccounts = youraccounts;
  }
}
```

Отметим, что единственной причиной использования *typedef* в IDL-файле является создание псевдонима типа, описываемого для ссылки на данный IDL-файл. Никакого класса Java для *typedef* не формируется. Заметим также, что в генерируемой программной конструкции Java отсутствует информация, связанная с массивом.

Интерфейсы IDL

Программа idl2java преобразует IDL-**интерфейс** в общий Java-интерфейс с тем же именем, что и у IDL-интерфейса. Такой Java-интерфейс реализуется на стороне клиента генерацией stub-класса, а на стороне сервера – генерацией скелетного класса вместе с классом серверного объекта, создаваемым программистом и расширяющим скелетный класс. Если IDL-интерфейс называется, скажем, interfacename, генерируемый Java- интерфейс также называется interfacename, генерируемый stub-класс – _st_interfacename, генерируемый скелетный класс – _interfacenameImplBase, а класс серверного объекта в среде программистов принято обозначать interfacenameImpl.

В IDL-интерфейсе могут присутствовать объявления типов, констант, исключительных ситуаций, атрибутов и методов. Примеры объявлений типов – объявления *enum*, *union* и *struct*, а также описания *typedef*. Константа, описываемая внутри интерфейса, в соответствующем Java-интерфейсе превращается в поле final public static с тем же именем, что и у константы. Типом поля является Java-тип, соответствующий IDL-типу константы. Например:

```
interface hit { const long c = 10; };
```

переводится в:

```
public interface hit extends org.omg.CORBA.Object {
    final public static int c = 10;
}
```

Исключительная ситуация (exception), описываемая в IDL-интерфейсе, преобразуется в описание Java-класса final public, расширяющее org.omg.UserException (см. следующий раздел "Объявление исключительных ситуаций"). Описание Java-класса содержится в Java-интерфейсе, генерируемом для IDL-интерфейса.

Атрибуты (attributes) – это поля данных экземпляра IDL-интерфейса (обратите внимание на то, что у Java-интерфейсов могут быть только поля данных final). Объявление *атрибута* состоит из ключевого слова *readonly* (только для чтения; необязательно), за которым следует ключевое слово *attribute*, а за ним IDL-тип и имя *атрибута*:

```
[ readonly ] attribute тип имя_атрибута;
```

IDL-*атрибут* в соответствующем генерируемом Java-интерфейсе преобразуется в метод доступа (для получения значения *атрибута*) и метод-модификатор (для установки значения *атрибута*). Если *атрибут* объявлен как *readonly*, создается только метод доступа. Например, IDL- *интерфейс*, содержащий *атрибуты* в приведенном ниже модуле account, переводится в следующий Java-интерфейс:

```
module account {
   interface AccountInfo { long departmentno; long projectno; }
};

public interface AccountInfo extends org.omg.CORBA.Object {
   public void departmentno( int departmentno );
   public int departmentno();
   public void projectno( int projectno );
   public int projectno();
}
```

Здесь методы Java с возвращаемым типом void – методы-модификаторы, а методы с возвращаемым типом int – методы доступа. Отметим, что создаваемый разработчиком класс серверного объекта, который "реализует" AccountInfo, расширяя скелетный класс, генерируемый для AccountInfo, может содержать поля данных для departmentno и projectno, а может и не содержать. Именно метод доступа и метод-модификатор считаются ответственными за реализацию *атрибута*. Использовать или нет поля данных Java для реализации – решает разработчик.

IDL-методы называются *операциями* (operations), хотя для ссылки на них обычно пользуются более известным термином "методы". Синтаксис объявления IDL-метода таков:

```
возвращаемый_тип имя_метода (список_параметров)
```

Список параметров либо пуст, либо содержит одно или несколько объявлений параметров, в последнем случае разделяемых запятыми. Вот синтаксис объявления параметра:

```
[ режим ] тип имя_параметра
```

Возможными режимами параметров методов являются *in*, *out* и *inout*. В режиме *in* значение фактического параметра передается параметру метода, но методом его изменить нельзя. В режиме *inout* значение фактического параметра передается параметру метода, а итоговое значение последнего передается обратно фактическому параметру. В режиме *out* значение фактического не передается параметру метода, но его итоговое значение передается обратно фактическому параметру.

Поскольку в Java вызовы поддерживаются только по значению (т.е. в режиме *in*), для каждого пользовательского типа, подобного *структуре* или *интерфейсу*, idl2java создает класс-оболочку, называемый классом- хранителем, или holder-классом. Затем такие классы-хранители применяются для параметров типов *inout* и *out*. Классы-хранители доступны и для стандартных (предопределенных) IDL-типов, например *long*. Так, ниже показаны объявление метода IDL-интерфейса и объявление Java-метода, в которое оно переводится:

```
interface account {
  void getAccountInfo(
    in long accountno, out long departmentno, out long projectno );
};

public interface account extends org.omg.CORBA.Object {
  public void getAccountInfo(
    int accountno,
    org.omg.CORBA.IntHolder departmentno,
    org.omg.CORBA.IntHolder projectno );
}
```

Более подробно о holder-классах рассказывается в следующем разделе.

Классы-хранители

Классы-хранители Java для базовых IDL-типов находятся в пакете org.omg.CORBA. Возьмем, к примеру, класс-хранитель для int:

```
final public class IntHolder {
  public int value;
  public Intholder() {}
  public Intholder( int initial ) { value = initial; }
}
```

Аналогичные holder-классы генерируются для IDL-типов, описываемых пользователями. Поскольку в методе Java-интерфейса для каждого метода IDL-интерфейса с параметром *inout* или *out* будет существовать соответствующий параметр holder-класса, в программе Java для класса серверного объекта интерфейса значение параметра будет задаваться установкой поля значения параметра holder-класса.

Отметим, что классы-хранители автоматически создаются для интерфейсов, *структур*, *объединений*, *перечней* и *исключительных ситуаций* IDL.

Объявление исключительных ситуаций

Исключительные ситуации, или исключения (exceptions),— это конструкции, соответствующие ошибочным условиям. В IDL поддерживаются стандартные исключительные ситуации, называемые системными, а также описываемые пользователями.

Системные исключительные ситуации преобразуются в исключительные ситуации Java, описанные в пакете `org.omg.CORBA`. Пользовательские — в генерируемый класс исключительной ситуации (т.е. в класс, который расширяет класс `org.omg.CORBA.UserException`). Такая ситуация объявляется с ключевым словом *exception*, а синтаксис объявления тот же, что и для *struct*, кроме, естественно, начального ключевого слова (*exception* вместо *struct*). В атрибутах исключительной ситуации хранятся сообщения об ошибках и другие значения, связанные с ошибками. В нижеприведенном примере показаны объявление исключительной ситуации и генерируемый при его преобразовании класс Java:

```
exception MyError { wstring mess; };

final public class MyError extends org.omg.CORBA.UserException {
  public java.lang.String mess;
  public MyError() {}
  public MyError( java.lang.String mess) { this.mess = mess; }
}
```

Отметим, что `CORBA.UserException` — это подкласс `java.lang.Exception`, поэтому он ведет себя как истинный класс исключительной ситуации Java.

Методы IDL можно объявлять способными устанавливать конкретную исключительную ситуацию, для чего используется конструкция *raises*:

```
void myfunc() raises( MyError );
```

Подобное объявление метода транслируется в объявление метода Java с конструкцией `throws`:

```
public void myfunc() throws MyError;
```

Наследование

Хотя в IDL допускается и множественное наследование интерфейсов, для простоты в этом разделе рассмотрим только одиночное наследование. Подробнее о множественном наследовании см. книгу "Java Programming with CORBA" [62].

Предположим, что структура наследования имеет следующий вид:

```
interface B:A { ... }
```

(где `B:A` означает происхождение интерфейса `B` из интерфейса `A`). Таким образом, `B` наследует все конструкции, объявленные в `A`. Соответствующие генерируемые интерфейсы Java также связаны отношением подкласса:

```
public interface B extends A { ... }
```

Модули

Итак, как мы уже видели, IDL-*модуль* (IDL module) — это единица компиляции в IDL. Синтаксис объявления модуля таков:

```
module имя_модуля {
  объявления типов;
  объявления констант;
  объявления исключительных ситуаций;
  объявления интерфейсов;
```

```
    объявления модулей;
};
```

Синтаксис объявления в модулях *типов*, *исключительных* ситуаций и *констант* такой же, что и при объявлении их в интерфейсах. Однако следует заметить, что объявления констант преобразуются в конструкции Java несколько иначе. Константы интерфейсов транслируются в соответствующих интерфейсах Java в поля `final public static`, а константы, содержащиеся непосредственно в модуле,— в общий интерфейс (`public interface`) Java, содержащий вышеуказанное поле с именем `value`, где хранится значение константы. Например:

```
const long MyConstant = 10;
```

преобразуется в:

```
public interface MyConstant {
   final public static int value = 10;
}
```

Внедрение и использование объектов CORBA

В этом разделе на примере account будет показано, как:

- Программировать *модуль* IDL
- Использовать `idl2java` для надлежащей генерации исходных файлов Java из *модуля* IDL
- Реализовывать серверные объекты CORBA в Java
- Компилировать файлы Java и загружать генерируемые классы Java в JServer
- Публиковать имена для каждого серверного объекта CORBA
- Программировать и компилировать клиент Java, вызывающий методы серверного объекта CORBA
- Выполнять клиент

Программирование модуля IDL

Цель этого приложения — узнать из таблицы `ACCOUNT_LIST` номер отдела и номер проекта для указанного номера счета. Начнем с программирования следующего IDL-модуля:

```
/* Имя файла: account.idl */

module account {
   struct AccountInfo { long departmentno; long projectno; };
   exception NoAccountError { wstring mess; };
   interface Account {
     AccountInfo getAccountInfo( in long accountno )
       raises( NoAccountError );
   };
};
```

`getAccount` — это метод, который возвращает структуру `AccountInfo`, содержащую значение `departmentno` и `projectno` для записи `ACCOUNT_LIST` с указанным значением `accountno`. Если такой записи в `ACCOUNT_LIST` не существует, устанавливается исключительная ситуация `NoAccountError`.

Создание исходных файлов Java из файла IDL с помощью idl2java

Чтобы из `account.idl` сгенерировать конструкцию Java, нужно лишь выполнить команду:

```
idl2java _no_tie account.idl
```

Параметр _no_tie указывает на то, что механизм привязки (см. следующий раздел) применяться не будет, поэтому сгенерируется меньшее число исходных файлов Java. Программа idl2java создаст для модуля 12 исходных файлов Java, причем все они будут содержаться в каталоге account:

- Интерфейс Java, helper-, holder-, stub- и скелетный файлы для интерфейса Account
- Исходный файл Java с описанием класса, а также helper- и holder- файлы Java для *структуры* AccountInfo и исключительной ситуации NoAccountError
- Файл _example_Account, содержащий неполный пример Java-реализации серверного объекта Account.

Программирование Java-реализации серверного объекта

Далее приведена SQLJ-реализация серверного объекта. В следующем разделе этой главы будет представлен JDBC-вариант данной программы:

```
/* Имя программы:    AccountImpl.sqlj
**
** Назначение:       Получить информацию о счете по данному номеру
**
*/
package accountServer;
import account.*;
import java.sql.*;
public class AccountImpl extends _AccountImplBase {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {

         //* Необходимо инициализировать переменные, чтобы выбрать из них информацию.
         int departmentno = 0;
         int projectno = 0;

         #sql {
           SELECT departmentno, projectno
              INTO :departmentno, :projectno
                 FROM ACCOUNT_LIST
                    WHERE accountno = :accountno
         };
         return new AccountInfo( departmentno, projectno );
     }
     catch( SQLException e ) {
         throw new NoAccountError( e.getMessage() );
     }
   }
}
```

Компиляция файлов Java и загрузка созданных файлов классов в JServer

Сначала скомпилируем исходные файлы Java:

```
javac account/Account.java\
   account/AccountHolder.java\
   account/AccountHelper.java
javac account/AccountInfo.java\
   account/AccountInfoHolder.java\
   account/AccountInfoHelper.java
javac account/NoAccountError.java\
   account/NoAccountErrorHolder.java\
```

```
  account/NoAccountErrorHelper.java
javac account/_EmployeeImplBase.java\
  account/_st_Account.java
sqlj -J-classpath\
  .:$(ORACLE_HOME)/lib/aurora_client.jar:\
  $(ORACLE_HOME)/jdbc/lib/classes111.zip:\
  $(ORACLE_HOME)/sqlj/lib/translator.zip$(ORACLE_HOME)/lib/vbjorb.jar:\
  $(ORACLE_HOME)/lib/vbjapp.jar$(JDK_HOME)/lib/classes.zip -ser2class\
  accountServer/AccountImpl.sqlj
```

Обратите внимание на то, что командой `sqlj` генерируются два дополнительных файла классов: `AccountImpl_SJProfile0` и `AccountImpl_SJProfilekeys`, которые также необходимо загружать на сервер. После этого можно создать файл Java-архива (`.jar`), содержащий все классы, загружаемые на сервер. Рекомендуется загружать их по очереди.

```
jar -cf0 accountjar.jar\
  account/Account*.class\
  account/NoAccountError*.class\
  account/_AccountImplBase.class\
  _st_Account.class\
  accountServer/AccountImpl*. class
```

Отметим, что трафаретный символ (*) используется при условии, что в каталоге нет других файлов с аналогичными именами. Наконец, загружаем все классы из файла `.jar` в JServer:

```
loadjava -oracleresolver -resolve -user scott/tiger accountjar.jar
```

Синтаксис команды `loadjava` приведен в следующем разделе этой главы.

Публикация имени серверного объекта CORBA

Последний этап подготовки серверного объекта – публикация его имени, которое сможет находить клиентская программа. Для этого используется команда `publish`:

```
publish -republish -user scott -password tiger -schema scott -service\
  sess_iiop://localhost:2481:ORCL\
  /test/Account AccountImpl account.AccountHelper
```

где параметры и аргументы имеют значения:

- `-republish` – перезаписать опубликованный объект с тем же именем.
- `-user scott: scott` – имя пользователя публикующей схемы.
- `-password tiger: tiger` – пароль для scott.
- `-schema scott:` имя публикующей схемы.
- `-service sess_iiop://localhost:2481:ORCL` указывает URL, идентифицирующий базу данных, чье пространство имен сеансов нужно открыть. Хост-имя (здесь `localhost`) идентифицирует компьютер, на котором находится БД (`localhost` возвращает хост-имя текущего компьютера), `2481` – порт прослушивающего процесса для `iiop`, а `ORCL` – системный идентификатор (SID) базы данных.
- `/test/Account:` имя публикуемого объекта. Каталог `/test` в пространстве имен – область, где публикуется большинство объектов. Публикуемое имя серверного объекта необязательно должно совпадать с именем интерфейса.
- `AccountImpl`: имя Java-класса, реализующего серверный объект.
- `AccountHelper`: имя helper-класса для `Account`.

Синтаксис команды `publish` приведен в следующем разделе.

Программирование и компиляция клиента Java

Чтобы обращаться к серверному объекту по его опубликованному имени, клиентская программа должна:

- Конкретизировать JNDI-объект InitialContext и заполнить его необходимыми свойствами соединения.
- Вызвать метод lookup() объекта InitialContext, передавая параметр URL с указанием искомых имени службы IIOP и имени серверного объекта. Метод lookup() возвращает ссылку на нужный серверный объект.
- Вызвать нужный метод серверного объекта по ссылке.

Следующая клиентская программа AccountUser.java выполняет все эти действия, чтобы вызвать метод getAccount() по ссылке на серверный объект Account:

```
/* Имя программы:    AccountUser.java
**
** Назначение:       Получить сведения о счете 1056 с помощью
**                   серверного обьекта Account
**
*/
import account.*;
import accountServer.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

/* Импортируем контекстный интерфейс JNDI.
   (см. пояснение 1)
*/
import javax.naming.Context;

/* Импортируем класс InitialContext, реализующий
   контекстный интерфейс.
   (см. пояснение 2)
*/
import javax.naming.InitialContext;

/* Импортируем класс хэш-таблицы для хранения свойств Среды
   начального контекста.
*/
import java.util.Hashtable;

public class AccountUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    /* Конкретизируем и заполним обьект InitialContext.
       (см. пояснение 3)
    */
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Найдем имя обьекта, получив ссылку на обьект.
    Account account = ( Account )
        ic.lookup( "sess_iiop://localhost:2481:ORCL/test/Account" );

    // Вызовем метод getAccountInfo().
    AccountInfo ai = account.getAccountInfo( 1056 );

    // Распечатаем сведения о счете.
```

```
        System.out.println( "Department number = " + ai.departmentno +
                            " Project number = " + ai.projectno );
    }
}
```

Пояснение к AccountUser.java:

1. Все EJB- и CORBA-клиенты Oracle8*i*, использующие для поиска серверных объектов методы JNDI, должны импортировать JNDI-интерфейс Context. Он формирует основу для JNDI-операций, управляющих службами и сеансами в Oracle8i ORB.
2. Все операции именования выполняются в соответствии с контекстом. Начальный контекст является отправной точкой в процессе определения имен.
3. Сначала создается новая Hashtable, в которой будут храниться свойства среды. В частности, в URL_PKG_PREFIXES – имя свойства среды, указывающее на список префиксов пакетов, используемых при загрузке контекстных средств URL. Сейчас для этого свойства должно быть установлено "oracle.aurora.jndi". Назначение других свойств – аутентификация:
   - SECURITY_PRINCIPAL хранит имя пользователя БД.
   - SECURITY_CREDENTIALS хранит пароль БД.
   - SECURITY_AUTHENTICATION хранит тип используемой аутентификации. Существует три возможных вида аутентификации, значения которых описаны в классе ServiceContext: NON_SSL_LOGIN, SSL_CREDENTIAL и SSL_LOGIN. Вообще, соединение уровня защиты сокетов (SSL – Secure Socket Layer) предоставляет две возможности: защищенный канал (шифрованную линию связи), который организуется всегда, и средство аутентификации (проверки подлинности идентификационных данных клиента), требующее у клиента SSL-сертификата. Клиент без сертификата аутентификацию не проходит, но защищенным соединением, тем не менее, пользоваться может. Режим SSL_LOGIN основан на использовании защищенного канала, но для аутентификации при этом применяется протокол регистрации (обычный пароль пользователя). В режиме же SSL_CREDENTIAL SSL-сертификат обязателен. В режиме NON_SSL_LOGIN для аутентификации применяется не соединение Secure Socket Layer, а обычный протокол регистрации по TCP/IP.

Создав программу для клиента Java, можно скомпилировать ее с помощью javac:

- `javac AccountUser.java`

Выполнение клиента

Выполним клиентский класс AccountUser командой java. Необходимо установить для этой команды classpath, указав classes.zip (стандартный библиотечный архив Java), классы, подобные тем, что находятся в vbjapp.jar и vbjorb.jar и применяются клиентом ORB, а также поставляемый вместе с Oracle8*i* .jar-файл aurora_client.jar. Вот выполнения нужного класса Java:

- ```
  % java -classpath\
    .:$(ORACLE_HOME)/lib/aurora_client.jar:\
    $(ORACLE_HOME)/jdbc/lib/classes111.zip:\
    $(ORACLE_HOME)/sqlj/lib/translator.zip:\
    $(ORACLE_HOME)/lib/vbjorb.jar:\
    $(ORACLE_HOME)/lib/vbjapp.jar:$(JDK_HOME)/lib/classes.zipAccountUser
  ```

Здесь JDK_HOME – место установки инструментария разработки Java Development Kit (JDK), а ORACLE_HOME – домашний каталог Oracle.
  ```

Реализация серверного объекта CORBA с помощью JDBC

В этом разделе приведена реализация AccountImpl с использованием JDBC:

```
/* Имя программы: AccountImpl.java */

package accountServer;
import account.*;
import java.sql.*;
public class AccountImpl extends _AccountImplBase {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {
         Connection conn =
           new oracle.jdbc.driver.OracleDriver().defaultConnection();
         PreparedStatement ps = conn.prepareStatement
           ( "SELECT departmentno, projectno
                FROM ACCOUNT_LIST WHERE accountno = ?" );
         try {
           ps.setInt( 1, accountno );
           ResultSet rset = ps.executeQuery();
           if ( !rset.next() )
              throw new NoAccountError( "No such account." );
           return new AccountInfo( rset.getInt(1),
              rset.getInt(2) );
         }
         finally { ps.close(); }
     }
     catch( SQLException e ) {
         throw new NoAccountError( e.getMessage() );
     }
   }
}
```

Механизм обратных вызовов CORBA

Служба обратных вызовов (callback) позволяет серверному объекту вызывать методы клиентского объекта. Чтобы обратный вызов был возможен, необходимо иметь:

- Клиентский объект обратного вызова, содержащий метод, вызываемый серверным объектом в "обратном порядке"
- Серверный объект, содержащий метод с параметром, т.е. ссылку на объект обратного вызова клиента (этот метод можно считать серверным методом обратного вызова, поскольку именно он будет выполнять обратный вызов)
- Реализацию клиентского объекта обратного вызова
- Реализацию серверного объекта
- Клиентскую программу, обращающуюся к серверному методу обратного вызова

Конечно, совсем не обязательно вызывать метод клиента напрямую, т.е. серверный метод обратного вызова может вызывать и другой, который вызывает третий и т.д., и на самом деле вызывается метод клиентского объекта обратного вызова. Но серверный метод обратного вызова позволяет программе клиента обращаться на клиентский объект обратного вызова, чтобы передать ссылку.

Ниже приведен простой пример, иллюстрирующий механизм обратных вызовов CORBA, где программные конструкции Java и IDL используются для возврата серверным методом строки символов "Java is cool.", а клиентским методом обратного вызова — строки символов "You said it.".

Клиентский объект обратного вызова

Следующий IDL-файл описывает клиентский объект обратного вызова:

```
/* Имя файла:    clientcallback.idl
**
** Назначение:   Описание клиентского объекта обратного вызова.
**
*/
module clientcallback {
   interface Clientcallback { wstring saidit(); };
};
```

Серверный объект

В следующем IDL-файле описывается серверный объект:

```
/* Имя файла: server.idl */

#include <clientcallback.idl>
module server {
   interface Server {
     wstring isCool( in clientcallback::ClientCallback objectref );
   };
};
```

Обратите внимание: серверный объект может ссылаться на объект ClientCallback потому, что по правилам C++ в состав серверного IDL-файла включен IDL-файл clientcallback. Отметим также, что ссылка на объект ClientCallback делается операцией определения области действия языка C++ (::).

Реализация клиентского объекта обратного вызова

В этом файле Java реализуется объект ClientCallback:

```
/* Имя программы:    ClientCallbackImpl.java
**
** Назначение:       Реализация объекта ClientCallback.
**
*/
import clientcallback.*;
public class ClientCallbackImpl extends _ClientCallbackImplBase {
   public String saidit() { return "You said it."; }
}
```

Отметим, что клиентский объект обратного вызова вовсе не обязательно загружать в JServer, так как он работает на станции клиента.

Реализация серверного объекта

В следующем файле Java реализуется серверный объект:

```
/* Имя программы: ServerImpl.java */

package callServer;
import server.*;
import clientcallback.*;
public class ServerImpl extends _ServerImplBase {
   public String isCool ( ClientCallback clientcallback) {
     return "Java is cool ." + clientcallback.saidit();
   }
}
```

Отметим, что необходимы Java-классы из обоих пакетов clientcallback и server (которые были созданы в результате выполнения javac над соответствующими файлами Java).

Клиентская программа, обращающаяся к серверному методу обратного вызова

В приведенном ниже файле Java содержится клиентская программа, начинающая свою работу с вызова метода isCool() серверного объекта. С помощью дополнительных программных конструкций, необходимых для обратного вызова, создается клиентский объект обратного вызова, который затем публикуется для ORB на станции клиента. В этих конструкциях есть ссылка на так называемый базовый объектный адаптер (BOA, basic object adapter), его назначение – создать и интерпретировать объектные ссылки, а также активировать и дезактивировать реализации объектов. BOA – это компонент ORB, гарантирующий, что вызов по объектной ссылке всегда достигнет активированного объекта, который способен на это ответить.

```
/* Имя программы:    Client.java
**
** Назначение:       Обращение к серверному методу обратного вызова.
**
*/
import server.*;
import clientcallback.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client {
   public static void main( String[] args ) throws Exception {
     Hashtable env = new Hashtable();
     env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
     env.put( Context.SECURITY_PRINCIPAL, "scott" );
     env.put( Context.SECURITY_CREDENTIALS, "tiger" );
     env.put( Context.SECURITY_AUTHENTICATION,
         ServiceCtx.NON_SSL_LOGIN );
     Context ic = new InitialContext( env );

     /* Для правильной инициализации ORB нужно получить серверный
        объект до подготовки клиентского объекта обратного вызова.
     */
     Server server = ( Server ) ic.lookup
      ( "sess_iiop://localhost:2481:ORCL/test/Server" );

     /* Создадим клиентский объект и опубликуем его для ORB на
        станции клиента.
     */
     /* Вызовем метод init() псевдообъекта ORB, чтобы узнать ссылку
        на существующий клиентский ORB. Необязательно, чтобы
        в псевдообъекте были задействованы типы CORBA.
     */
     org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();

     // С помощью ссылки ORB инициализируем клиентский BOA.
     org.omg.CORBA.BOA boa = orb.BOA_init();

     // Конкретизируем новый клиентский объект обратного вызова.
     ClientCallbackImpl clientcallback = new ClientCallbackImpl();

     // Зарегистрируем клиентский объект обратного вызова с помощью клиентского BOA.
     boa.obj_is_ready( clientcallback );


     // Вызовем серверный метод обратного вызова.
     System.out.println( server.isCool( clientcallback ) );
   }
}
```

Выполнив эту программу, пользователь увидит следующий результат:

```
Java is cool. You said it.
```

Механизм привязки CORBA

Существует альтернативный способ реализации – это механизм привязки CORBA серверного объекта, отличный от расширения генерируемого скелетного класса. При его использовании не происходит прямого расширения скелетного класса, а реализуется Java-интерфейс *имя_интерфейса Operations*, генерируемый программой idl2java из IDL-интерфейса. Достоинством механизма привязки является свобода в использовании серверного класса реализации, который может расширять другой класс, поскольку указывается ключевое слово "extends", а не "implements". Обычный серверный класс реализации расширять нельзя, так как он уже расширяет скелетный класс.

В качестве примера рассмотрим модуль cool из раздела "Компилятор IDL" этой главы. Для реализации объекта Cool при помощи механизма привязки нужно изменить файл CoolImpl.java, однако остальные файлы IDL и Java можно оставить абсолютно без изменений. Если точнее, то в режиме привязки CoolImpl необходимо реализовать интерфейс oracle.aurora.AuroraServices.ActivateObject, содержащий один метод без параметров – _initializeAuroraObject(). Обычно этот метод реализуется для классов привязки возвратом нового экземпляра _tie_имя_обьекта, как показано в следующем примере:

```
/* Имя программы: CoolImpl.java */

package coolServer;
import cool.*;
import oracle.aurora.AuroraServices.ActivatableObject;
  public class CoolImpl implements CoolOperations, ActivatableObject {
  public String isCool() { return "Java is cool."; };
  public org.omg.CORBA.Object _initializeAuroraObject() {
    return new _tie_Cool( this ) ;
  }
}
```

Естественно, при выполнении idl2java над серверным модулем параметр _no_tie указывать не нужно.

Активация посредников ORB и серверных объектов

В этом разделе нас интересуют:

- Активация ORB
- Активация серверного объекта

Активация ORB

В приложениях CORBA требуется, чтобы ORB был активен на стороне как клиента, так и сервера. Рассмотрим подробнее процесс активации ORB в Oracle8*i*.

Клиентский ORB активируется, когда клиент впервые вызывает метод lookup(). Если необходима ссылка на клиентский ORB (см. предыдущий пример с обратным вызовом), она возвращается методом init() псевдообъекта ORB. Заметим, что не все операции можно выполнять по ссылке, возвращаемой методом ORB.init(). Это особенно важно при работе с апплетами, когда особое внимание уделяется вопросам, связанным с безопасностью информации.

Серверный ORB активируется при создании сеанса, презентацией, обрабатывающей запросы IIOP.

Активация серверного объекта

Серверный объект автоматически активируется, когда клиент выполненяет его поиск (lookup).

Постоянные объекты JServer ORB регистрирует автоматически с помощью BOA. *Переходные* же (т.е. те, что действуют в течение работы создавшего их процесса), генерируемые другими объектами, необходимо регистрировать с помощью BOA, вызывая метод obj_is_ready() объекта BOA. Напомним, что метод вызывался клиентской программой в примере обратного вызова для регистрации "переходного" клиентского объекта Callback. В приведенном ниже примере показан процесс создания переходных объектов, где метод одного серверного объекта динамически создает метод другого и фигурируют:

- IDL-модуль, который описывает два серверных объекта, объект с именем CoolMaker, создающий экземпляр объекта с именем Cool
- Реализации этих серверных объектов
- Клиентская программная конструкция, вызывающая метод объекта CoolMaker (который, в свою очередь, создает новый экземпляр Cool)

IDL-модуль

Сначала представим IDL-модуль:

```
/* Имя файла: maker.idl */

module maker {
   interface Cool { wstring isCool(); };
   interface CoolMaker { Cool make( in wstring mess ); };
};
```

Метод make() создает новый экземпляр Cool, инициализируемый указанным сообщением. После этого клиент сможет вызывать метод isCool() экземпляра, возвращая хранимое там сообщение.

Реализация серверных объектов

Далее представим Java-реализацию серверных объектов:

```
/* Имя программы:    CoolImpl.java
**
** Назначение:       Реализация объекта Cool.
**
*/
package makerServer;
import maker.*;
public class CoolImpl extends _CoolImplBase {
   String mess;

   // Конструктор для инициализации объекта сообщением.
   public CoolImpl( String mess ) { this.mess = mess; }
   public String isCool() { return mess; }
}

/* Имя программы:    CoolMakerImpl.java
**
** Назначение:       Реализация объекта CoolMaker.
**
*/
package makerServer;
import maker.*;
import oracle.aurora.AuroraServices.ActivatableObject;
public class CoolMakerImpl extends _CoolMakerImplBase
   implements ActivatableObject {
   public Cool make( String mess ) {

      // Создадим переходной объект Cool.
```

```
    CoolImpl cool = new CoolImpl( mess );

    /* Зарегистрируем этот объект с помощью серверного BOA,
       используя метод connect(), который является переносимым
       вариантом obj_is_ready().
    */
    _orb().connect( cool );
    return cool;
  }
  public org.omg.CORBA.Object _initializeAuroraObject() {
    return this;
  }
}
```

Клиентская программа

Наконец, представим клиентскую программу:

```
/* Имя программы: CoolClient.java */

import maker.*;
import makerServer.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class CoolClient {
  public static void main( String[] args ) throws Exception {
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" ) ;
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Установим объект CoolMaker.
    CoolMaker maker = ( CoolMaker )
        ic.lookup( "sess_iiop://localhost:2481:ORCL/test/CoolMaker" );

    /* Вызовем метод make для создания объектов cool,
       инициализируемых нужными сообщениями.
    */
    Cool cool = maker.make( "Java is cool." );
    Cool cools = maker.make( "SQL is cool too." );

    // Вызовем метод isCool() в экземплярах Cool для получения сообщений.
    System.out.println( cool.isCool() );
    System.out.println( cools.isCool() );
  }
}
```

Программирование спецификаций интерфейсов на Java

В среде разработки Java в Oracle8*i* можно пользоваться инструментальными средствами Inprise Caffeine: java2iiop и java2idl. Из спецификаций интерфейсов, написанных на Java, первое напрямую генерирует совместимые с CORBA изоляторы и скелеты Java, а второе — спецификации IDL (см. главу 6 руководства "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide" [52]).

Краткие сведения о некоторых важных инструментах

Теперь рассмотрим следующие инструменты, выполняемые из приглашения системы:

- loadjava
- dropjava
- publish
- remove

loadjava

Утилита loadjava используется для преобразования .class-файлов в библиотечные единицы базы данных, называемые *схемными объектами Java-классов* (Java class schema objects) и хранимые на сервере, а также .ser-файлов в схожие единицы, называемые *схемными объектами ресурсов* (resource schema objects), тоже хранимые на сервере, и исходных файлов в библиотечные единицы, называемые *схемными объектами источников* (source schema objects) и хранимые на сервере. Для каждого .class-файла, каждого .ser-файла и каждого .java- файла источника создается отличный от других схемный объект.

В командной строке loadjava можно указать каждый .class- или .ser- файл по отдельности или общий .jar-файл (архив Java), объединив сначала все файлы. Для объединения отдельных файлов в .jar-файл применяется утилита jar. В .jar-файле можно объединить файлы классов и ресурсов или файлы источников и ресурсов, но только не файлы источников и классов. Синтаксис инструмента loadjava описан в "Oracle8*i* SQLJ Programming" [37, стр. 525]:

```
loadjava {-user | -u} <user>/<password>[@<database>] [options]
{<file>.java | <file>.class | <file>.jar | <file>.zip | <file>.sqlj |
<resourcefile>} ...
```

Приведенные здесь возможные параметры взяты из "Oracle8*i* SQLJ Programming" [37, стр. 526,527]:

Параметр	Описание
<filenames>	Можно указать любое число и любую комбинацию имен файлов .java, .class, .sqlj, .jar, .zip и ресурсов в любом порядке.
-andresolve	Приказывает loadjava компилировать источники, если те были загружены, и определять внешние ссылки в каждом классе по мере его загрузки. Параметры -andresolve и -resolve являются взаимоисключающими.
-debug	Приказывает компилятору Java генерировать отладочную информацию.
-definer	По умолчанию схемные объекты классов действуют с привилегиями их вызывающего. Этот параметр предоставляет классам привилегии определяющего (описывающего).
-grant <grants>	Предоставляет привилегию EXECUTE (выполнения) на загруженные классы указанным пользователям и ролям. Можно задавать любое число пользователей и ролей в любой комбинации, отделяя их друг от друга запятыми, но не пробелами (-grant Bob,Betty, но не -grant Bob, Betty).
-oci8	Использует JDBC-драйвер OCI; -oci8 и -thin взаимно исключают друг друга; если ничего не указано, по умолчанию используется -oci8.
-oracleresolver	Использует определитель, требующий, чтобы были найдены все ссылки на классы.
-resolve	Компилирует (при необходимости) и определяет внешние ссылки в классах после загрузки всех классов из командной строки. Параметры -andresolve и -resolve являются взаимоисключающими.

Параметр	Описание
-resolver <resolver>	Использует определитель, требующий, чтобы были найдены все ссылки на классы.
-schema	Называет схему, в которой создаются схемные объекты. Если схема не указана, используется схема регистрации (входа в систему). Для создания схемного объекта не в собственной схеме необходимо иметь привилегию CREATE PROCEDURE или CREATE ANY PROCEDURE.
-synonym	Создает общий (PUBLIC) синоним для загруженных классов, делая их доступными извне схемы, в которую они загружаются. Необходимо иметь привилегию CREATE PUBLIC SYNONYM.
-thin	Использует тонкий JDBC-драйвер. -oci8 и -thin взаимно исключают друг друга; если ничего не указано, по умолчанию используется -oci8.
-user, -u	Указывает пользователя, пароль и строку соединения с базой данных. Вид аргумента: <username>/<password>[@<database>]
-verbose	Приказывает loadjava во время работы выдавать сообщения с подробным описанием состояния

dropjava

Данная утилита используется для удаления схемных объектов Java с сервера. Это инверсия команды loadjava (см. книгу "Oracle8*i* SQLJ Programming" [37, стр. 527]):

```
dropjava {-u | -user} <user>/<password>[@<database>] [options]
{<file>.java | <file>.class | <file>.sqlj | <file>.jar | <file>.zip |
<resourcefile>} ...
```

Приведенные здесь возможные параметры описаны в книге "Oracle8*i* SQLJ Programming" [37, стр. 528]:

Параметр	Описание
-user	Указывает пользователя, пароль и необязательную строку соединения с базой данных.
<filenames>	Можно указать любое число и любую комбинацию имен файлов .java, .class, .sqlj, .jar, .zip и ресурсов в любом порядке. Файлы .jar и .zip должны быть распакованы.
-oci8	Использует JDBC-драйвер OCI: -oci8 и -thin взаимно исключают друг друга; если ничего не указано, по умолчанию применяется -oci8.
-schema	Называет схему, из которой удаляются схемные объекты. Если схема не указана, задействуется схема регистрации (входа в систему). Для удаления схемного объекта из чужой схемы необходимо иметь системную привилегию DROP ANY PROCEDURE.
-thin	Использует тонкий JDBC-драйвер: -oci8 и -thin взаимно исключают друг друга; если ничего не указано, по умолчанию применяется -oci8
-verbose	Приказывает dropjava во время работы выдавать сообщения с подробным описанием состояния

publish

У каждого экземпляра базы данных, в котором функционирует программное обеспечение JServer, есть *пространство имен сеансов* (session namespace), используемое Oracle8*i* ORB для активации серверных объектов CORBA,– иерархическая совокупность объектов, называемых *опубликованными объектами* (PublishedObjects) и *публикующими контекстами* (PublishingContexts).

PublishedObjects – это нижний уровень иерархии, а PublishingContexts – узлы верхнего уровня, аналогичные системным файлам и каталогам UNIX. Каждый PublishedObject соответствует схемному объекту класса, представляющему

CORBA-реализацию объекта. Для активации серверного объекта CORBA клиент ссылается на имя соответствующего опубликованного объекта.

Процесс создания PublishedObject называется *публикацией* (publishing) и выполняется с помощью утилиты publish (описываемой в этом разделе) или интерактивной оболочки сеансов (описанной в приложении F). Опубликованные объекты CORBA создаются ими после загрузки на сервер реализаций соответствующих объектов с помощью утилиты loadjava.

Атрибуты PublishedObject описаны в "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-18]:

- **Schema object name** Имя схемного объекта Java-класса, связанного с PublishedObject.
- **Schema** Имя схемы, содержащей соответствующий схемный объект класса.
- **Helper schema object name** Имя helper-класса, используемого Oracle8*i* ORB для автоматического сужения ссылки на экземпляр объекта CORBA.

У PublishedObjects и PublishingContexts есть владельцы и привилегии. Владельцем (owner) может быть имя пользователя или роли. Изменять право владения привилегиями конкретного PublishedObject или PublishingContext может только владелец PublishedObject или PublishingContext. Права на пространство имен сеансов описаны в таблице 6.2, которая взята из "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-18, таблица 6-4].

Таблица 6.2. *Права PublishingContext и PublishingObject*

Право	Значение для PublishingContext	Значение для PublishedObject
read	Просмотр содержимого и атрибутов (тип, права и время создания)	Просмотр атрибутов объекта (тип, схемный объект, схема, helper, права, время создания)
write	Создание PublishedObject или PublishingContext в PublishingContext	Повторная публикация объекта
execute	Использование содержимого для определения имени	Активация ассоциированного класса

Oracle8*i* создает пространство имен автоматически при конфигурировании Oracle8*i* ORB. Контексты PublishingContext, представленные в таблице 6.3 ("Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-18, таблица 6-5]), существуют во всех пространствах имен сеансов.

Таблица 6.3. *Начальные публикующие контексты и права*

Имя	Владелец	Read	Write	Execute
/	SYS	PUBLIC	SYS	PUBLIC
/bin	SYS	PUBLIC	SYS	PUBLIC
/etc	SYS	PUBLIC	SYS	PUBLIC
/test	SYS	PUBLIC	PUBLIC	PUBLIC

Поскольку PUBLIC может осуществлять запись только в /test, PublishingContexts и PublishedObjects создаются, как правило, именно в /test. Утилита publish создает или заменяет (повторно публикует (*republish*)) PublishedObject в PublishingContext. При обновлении схемного объекта Java-класса повторная публикация необязательна. Она нужна лишь при изменении атрибутов опубликованного объекта. Синтаксис команды publish взят из "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-19]:

```
publish <name> <class> [<helper>] -user <username> -password <password>
-service <serviceURL> [options]
```

```
[-describe]
[{-g | -grant} {<user> | <role>}[,{<user> | <role>}]...]
[{-h | -help}]
[-iiop]
[-role <role>]
[-republish]
[-schema <schema>]
[-ssl]
 [-version]
```

Аргументы командной строки publish подытожены в таблице 6.4 ("Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-19–6-21, таблица 6-6]):

Таблица 6.4. *Краткие сведения об аргументах инструментального средства publish*

Параметр	Описание
<name>	Имя создаваемого или повторно публикуемого PublishedObject; при необходимости создаются PublishingContexts.
<class>	Имя схемного объекта класса, соответствующего <name>.
<helper>	Имя схемного объекта Java-класса, реализующего метод narrow() для <class>.
-user	Указывает идентификационные характеристики для регистрации в экземпляре базы данных, указанном в -service.
-password	Указывает аутентифицирующий пароль для имени пользователя, заданного в -user.
-service	URL, идентифицирующий базу данных, пространство имен сеансов которой "открывается" с помощью sess_sh. serviceURL имеет вид: sess_iiop://<*host*>:<*lport*>:<*sid*> <host> — это компьютер, на котором расположена целевая база данных; <lport> — порт прослушивающего процесса, установленный для прослушивания сеансового IIOP; <sid> — идентификатор экземпляра базы данных. Например, sess_iiop://localhost:2481:ORCL соответствует установке по умолчанию на машине вызывающего.
-describe	Кратко описывает действие инструмента, а затем осуществляет выход из программы.
-grant	После создания или повторной публикации объекта PublishedObject предоставляет права read и execute последовательности имен <user> и <role>. При повторной публикации заменяет существующих пользователей/роли с правами read/execute на имена <user> и <role>. Для селективного изменения прав PublishedObject используется команда chmod утилиты sess_sh. Обратите внимание: для активации объекта CORBA или зерна EJB пользователь должен иметь право execute как на PublishedObject, так и на соответствующий схемный объект класса.
-help	Кратко описывает синтаксис инструмента, а затем осуществляет выход из программы.
-iiop	Соединяется с целевой базой данных с помощью IIOP вместо заданного по умолчанию сеансового IIOP; используется при публикации на сервере базы данных, сконфигурированном без сеансового IIOP.
-role	Роль, принимаемая для publish; значения по умолчанию нет.

Таблица 6.4. *Краткие сведения об аргументах инструментального средства* `publish` *(продолжение)*

Параметр	Описание
`-republish`	Обязывает инструмент `publish` заменять существующий `PublishedObject`; без этого параметра `publish` отказывается от попытки публикации существующего имени. Если `PublishedObject` не существует, `publish` создает его. При повторной публикации права, не принадлежащие владельцу, удаляются; для добавления прав read/execute при повторной публикации служит параметр `-grant`.
`-schema`	Схема, содержащая схемный объект `<class>` Java. Если не указать его, инструмент `publish` применяет схему вызывающего.
`-ssl`	Соединяется с базой данных посредством серверной SSL-аутентификации. Для его использования необходимо настроить БД на работу с SSL и указать порт прослушивающего процесса SSL в `-service`.
`-version`	Показывает версию инструмента, а затем осуществляет выход из программы.

remove

Утилита `remove` удаляет `PublishedObject` или `PublishingContext` из пространства имен сеансов. Обратите внимание: соответствующий объект Java с сервера не удаляется — для этого служит утилита `dropjava`. Синтаксис команды `remove` представлен в "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-21]:

```
remove <name> -user <username> -password <password>
-service <serviceURL> [options]
  [{-d | -describe}]
  [{-h | -help}]
  [-iiop]
  [{-r | -recurse]
  [-role <role>]
  [-ssl]
  [-version]
```

Аргументы `remove` подытожены в таблице 6.5 ("Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-21, 6-22, таблица 6-7]).

Приведем пример вызова `remove`:

```
remove /test/Account -user scott -password tiger\
  -service sess_iiop://localhost:2481:ORCL
```

Эта команда удалит из пространства имен сеансов `PublishedObject` с именем `/test/Account`.

Таблица 6.5. *Краткие сведения об аргументах* `remove`

Параметр	Описание
`<name>`	Имя удаляемого `PublishingContext` или `PublishedObject`.
`-user`	Указывает идентификационные характеристики для регистрации в экземпляре, указанном в `-service`.
`-password`	Указывает аутентифицирующий пароль для `<username>`, заданного в `-user`.

Таблица 6.5. *Краткие сведения об аргументах* `remove` *(продолжение)*

Параметр	Описание

`-service`	URL, идентифицирующий базу данных, пространство имен сеансов которой "открывается" с помощью sess_sh. serviceURL имеет вид:
	`sess_iiop://<host>:<lport>:<sid>`
	`<host>` — это компьютер, на котором расположена целевая база данных; `<lport>` — порт прослушивающего процесса, установленный для прослушивания сеансового IIOP; `<sid>` — идентификатор экземпляра базы данных. Пример:
	`sess_iiop://localhost:2481:ORCL`
	что соответствует установке по умолчанию на машине вызывающего.

`-describe`	Кратко описывает действие инструмента, а затем осуществляет выход из программы.
`-help`	Кратко описывает синтаксис инструмента, а затем осуществляет выход из программы.
`-iiop`	Соединяется с целевой базой данных с помощью IIOP вместо заданного по умолчанию сеансового IIOP. Этот параметр используется при удалении с сервера базы данных, сконфигурированного без сеансового IIOP.
`-recurse`	Рекурсивно удаляет `<name>` и все подчиненные контексты `PublishingContext`; обязателен для удаления `PublishingContext`.
`-role`	Роль, принимаемая для `remove`; значения по умолчанию нет.
`-ssl`	Соединяется с базой данных посредством серверной SSL-аутентификации. Для использования этого параметра необходимо настроить БД на работу с SSL.
`-version`	Показывает версию инструмента, а затем осуществляет выход из программы.

Итоги

В этой главе обсуждалось внедрение и использование простых серверных объектов CORBA. Было рассказано о главных достоинствах серверных объектов CORBA, в том числе об их неоднородности (так, серверные объекты, описанные на нейтральном по отношению к языкам программирования IDL, можно реализовывать на разных языках) и прозрачности местонахождения, т.е. клиент не обязан знать о месте применяемого им серверного объекта.

Рассматривались методы программирования CORBA, в том числе механизм обратных вызовов, позволяющий серверному объекту вызывать метод клиента, механизм привязки, предоставляющий альтернативный способ реализации серверного объекта (отличный от расширения генерируемого скелетного класса) и метод активации переходных объектов предварительной их регистрацией с помощью BOA.

В следующей главе мы более подробно расскажем о создании сеансов, в которых участвуют объекты CORBA, и управлении ими.

ГЛАВА 7

Сеансы CORBA

Несмотря на то что в стандарте CORBA концепция сеансов не принята, она является фундаментальной для баз данных Oracle8*i* и JServer ORB. Каждый серверный CORBA-объект Oracle8*i* работает в конкретном сеансе. Нередко все серверные объекты активируются в одном сеансе, и в этом случае для управления сеансами вполне достаточно службы стандартного IIOP. Если же объекты необходимо активировать в различных сеансах, потребуется служба сеансового IIOP. В этой главе говорится о том, что имеет отношение к сеансам, в частности о:

- Службах соединений: TTC и IIOP
- Характеристиках JNDI
- CORBA-сеансах Oracle8*i*
- Управлении сеансами
- Аутентификации
- Клиентах не-JNDI

Службы соединений: TTC и IIOP

Презентацией (presentation) в Oracle называется служебный протокол, принимающий входящие сетевые запросы и активирующий для обработки запроса служебные подпрограммы в ядре базы данных или виртуальной машине Java. Служба (service) реализует служебный протокол. В Oracle8*i* JServer поддерживаются три службы: служба двухзадачного общего (TTC, two-task common) уровня, унаследованная из Oracle версии 8.0.5, служба сеансового CORBA-протокола Internet Inter-ORB Protocol (сеансового IIOP) и служба стандартного IIOP. В Oracle версии 8.1.7 для сервлетов применяется и четвертая презентация – HTTP. Кроме того, в Oracle8*i* пользователи могут описывать свои собственные презентации.

Служба TTC обрабатывает входящие запросы Net8, обращенные к SQL-службам БД и поступающие от инструментов Oracle, подобных SQL*PLUS, и специальных приложений, создаваемых с помощью таких инструментов, как Oracle Forms, PRO*C и SQLJ. Службы IIOP обрабатывают запросы TCP/IP, направляемые в точку входа в службу прослушивающим процессом и диспетчером и начинают сеансы связи с базой данных Oracle8*i*, управляют ими и завершают их. Через такие сеансы обращение к объектам CORBA происходит так же, как и через службу TTC, которая начинает обработку запросов, полученных от SQL*PLUS или подобных инструментов, управляет ей и завершает ее. Средства типа `publish`, `deployejb` и `session shell` обращаются к объектам CORBA, тем самым соединяясь с базой данных через порт IIOP. Кроме того, этот порт используется клиентами EJB и CORBA, а также распределенными объектами, обращающимися к другим распределенным объектам, при отправке запросов Oracle. Но инструменты типа `loadjava` и `dropjava`, которые не обращаются к распределенным объектам, подключаются через порт TTC. Заметим, что по умолчанию номер порта TTC – 1521, а порта IIOP – 2481.

Принципиально службы стандартного и сеансового IIOP отличаются лишь в одном: первая позволяет клиентам обращаться только к тем серверным объектам, которые функционируют в одном сеансе, а вторая – к серверным объектам в нескольких сеансах. В сеансовом IIOP, соединяясь с несколькими сеансами, можно встраивать идентификатор сеанса в объектные ссылки, что позволяет связывать эти ссылки с разными сеансами одновременно.

Характеристики JNDI

Теперь разберем, как использовать интерфейс именования и каталоги Java (JNDI, Java Naming and Directory Interface) при поиске и активации опубликованных объектов в пространстве имен сеансов. JNDI – это продукт компании Sun Microsystems. При его использовании в клиентской программе или

программе реализации серверного объекта необходимо включить в каждый исходный файл операторы импорта:

```
// JNDI-интерфейс Context.
import javax.naming.Context;

/* Импортируем класс InitialContext, который реализует интерфейс
   Context interface.
*/
import javax.naming.InitialContext;

// Константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Хэш-таблица для среды начального контекста.
import java.util.Hashtable;
```

В этом разделе рассматриваются:

- JNDI-интерфейс Context
- JNDI-класс InitialContext

JNDI-интерфейс Context

Все EJB- и CORBA-клиенты Oracle8*i*, использующие JNDI в поиске и активации серверных объектов, должны импортировать интерфейс javax.naming.Context. Этот интерфейс является основой для операций JNDI, управляющих службами и сеансами в Oracle8*i* ORB.

Перед использованием JNDI для соединения клиентской программы с сервером Oracle8*i* необходимо установить среду для контекста JNDI. Свойства среды сохраняются, как правило, в хэш-таблице, хотя для этой цели можно использовать и список свойств (см. главу 6). Хэш-таблица создается в программе Java оператором:

```
Hashtable environment = new Hashtable();
```

Советуем перечитать пояснение 3 к программе в разделе "Программирование и компиляция клиента Java" главы 6, где говорится о свойствах среды и о том, как их установить.

В интерфейсе Context содержится несколько методов, реализуемых в классах ServiceCtx, SessionCtx и InitialContext. Эти классы описаны в следующем разделе.

JNDI-класс InitialContext

InitialContext — это класс пакета javax.naming, реализующий интерфейс Context. Объект InitialContext создается для среды, и при вызове его метода lookup() возвращается ссылка на опубликованный объект. Например, следующий программный фрагмент взят из программы AccountUser.java (см. главу 6):

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext(env);
Account account = (Account) ic.lookup(
    "sess_iiop://localhost:2481:ORCL/test/Account" );
```

Кроме того, метод lookup() объекта InitialContext возвращает новый объект служебного контекста.

```
ServiceCtx service = (ServiceCtx)ic.lookup(
    "sess_iiop://localhost:2481:ORCL" );
```

В данном случае в URL нет именованного объекта (такой URL называется служебным (serviceURL)), и объект, возвращаемый методом lookup(), передается в ServiceCtx. Этот фрагмент запускает службу сеансового IIOP. Объект служебного контекста представляет имя службы, которая была запущена вызовом lookup(). Класс ServiceCtx расширяет класс Context. Аналогичным образом, класс SessionCtx расширяет класс Context, а объект SessionCtx представляет сеанс.

CORBA-сеансы Oracle8*i*

Сеанс (session) – это соединение клиента со службой. Например, когда инструмент, подобный SQL*PLUS, через Net8 соединяется с TTC-портом прослушивающего процесса, Oracle8*i* устанавливает новый сеанс связи с базой данных, чтобы работать с соединением и обеспечивать SQL-поддержку. Точно так же, новый сеанс может быть установлен со стороны Oracle8*i* для обработки входящего запроса от клиентской программы CORBA, скажем при исполнении метода lookup() контекста для установки нового соединения. Всякий раз, когда InitialContext.lookup() активирует серверный объект, активация ограничится либо новым, либо существующим сеансом. Таким образом, серверные объекты активируются внутри сеансов, поэтому ORB и БД должны отличать объекты в одном и том же серверном процессе, беря за основу сеансы, в котором те активированы.

В стандартном IIOP соединение идентифицируется по его хосту и номеру порта, которые также включаются в объектные ссылки. В сеансовом же IIOP клиент в границах службы может подключаться к нескольким сеансам. Следовательно, служба сеансового IIOP должна различать сеансы, как, впрочем, и службы. Позволяет это сделать *тэг компонентов* (component tag) сеансового IIOP – SessionIIOP. В нем содержится информация, однозначно идентифицирующая сеанс, в котором был активирован объект. Она используется выполняющей системой ORB клиента для отправки запросов на получение нужных объектов в нужном сеансе. Заметим, однако, что включение идентификационной информации о сеансах является единственным отличием сеансового IIOP от стандартного.

Когда клиентская программа устанавливает IIOP-соединение с базой данных, серверная программа должна решить, начинать ли для обработки соединения новый сеанс или воспользоваться существующим. Если клиент с помощью метода lookup() посылает новый запрос на соединение, а активных сеансов для последнего нет, автоматически стартует новый сеанс.

Для направления запроса на соединение существующему сеансу вовсе не обязательно применять сеансовый IIOP – достаточно стандартного. Сеансовый обязателен лишь тогда, когда один клиент должен обратиться к серверным объектам, находящимся в нескольких сеансах (например, при подключении к Oracle8*i* многоуровневого сервера от имени нескольких клиентов).

Далее рассматриваются:

- Прослушивающие процессы и диспетчеры БД
- Синтаксис URL для служб и сеансов
- Компоненты и классы URL
- Класс служебного контекста
- Класс сеансового контекста
- Пример приложения

Прослушивающие процессы и диспетчеры базы данных

После получения клиентского запроса на IIOP-соединение прослушивающий процесс выделяет IIOP-диспетчера и посылает клиенту IIOP-ответ, чтобы тот установил с ним соединение.

Разделяемый сервер, обслуживая новое IIOP-соединение, сначала создает для него новый сеанс связи с базой данных, а затем активирует в сеансе ORB, где ORB отвечает за чтение входящих IIOP- сообщений, за аутентификацию клиента, за нахождение и активацию соответствующих серверных объектов и за отправку нужных IIOP-сообщений клиенту.

Потом IIOP-сообщения от того же клиента сразу направляются существующему сеансу и так же обрабатываются ORB.

Когда прослушивающий процесс настроен на соединение Net8 или IIOP, различий между сеансовым IIOP и стандартным нет. Сообщения обоих типов обрабатываются прослушивающим процессом в одном и том же порту. Но для соединений Secure Socket Layer (SSL) необходим отдельный порт (далее см. главу 6).

Синтаксис URL для служб и сеансов

В Oracle8*i* универсальные указатели ресурсов (URL, universal resource locator) имеют специальный синтаксис, позволяющий с помощью JNDI-запросов запускать службы и сеансы и обращаться к серверным объектам, опубликованным в экземпляре базы данных. Мы уже продемонстрировали указатели URL, передаваемые в качестве аргументов метода `lookup()` класса `InitialContext`. Указатель serviceURL используется для запуска такой службы, как `sess_iiop`. Подобный URL состоит из:

- Имени службы, за которым следуют двоеточие и две косых черты: `sess_iiop://`
- Системного имени (хост-имени) (можно использовать или настоящее хост-имя, или `localhost` (локальное хост-имя), или IP-адрес хоста в числовом виде)
- Номера порта прослушивающего процесса для служб IIOP (по умолчанию 2481)
- Системного идентификатора (SID), например ORCL

Для разделения хост-имени, порта и SID используются двоеточия. Например:

```
sess_iiop://local-host:2481:ORCL
```

Перед нами serviceURL. URL для активации серверного объекта имеет вид serviceURL плюс имя объекта:

```
sess_iiop://localhost:2481:ORCL/test/Account
```

Компоненты и классы URL

Как видно, при установлении соединения с Oracle и поиске опубликованного объекта с помощью JNDI используется URL, указывающий не только имя опубликованного объекта, который нужно найти и активировать, но и службу (ее имя, хост, порт и SID). В таком URL можно ввести и имя сеанса:

```
sess_iiop://localhost:2481:ORCL/:default/test/Account
```

Здесь `sess_iiop://localhost:2481:ORCL` — имя службы, `:default` — сеанс по умолчанию (если сеанс был установлен ранее), а `/test/Account` — полный путь к пространству имен искомого серверного объекта. На станции клиента сеансом по умолчанию считается тот, что установлен первым, и он автоматически называется `:default`. Любой серверный объект, активируемый впоследствии со станции клиента вызовом `InitialContext.lookup()`, автоматически активируется в сеансе по умолчанию, только если в URL, передаваемом методу `lookup()` не указано имя другого сеанса явным образом. Заметим, что если в URL, содержащем имя серверного объекта, имя сеанса не указано, серверный объект, тем не менее, активируется в сеансе по умолчанию. Следовательно, явно указывать `:default` в URL не обязательно.

Каждый из этих компонентов URL является Java-классом. Так, имя службы представлено объектом `ServiceCtx`, а сеанс — объектом `SessionCtx`.

Теперь рассмотрим:

- Класс служебного контекста
- Класс сеансового контекста

Класс служебного контекста

Класс `ServiceCtx` расширяет JNDI-класс `Context`. Как мы уже видели, в классе `ServiceCtx` описывается несколько переменных типа `final public static` (т.е. констант), сведения о которых приведены в таблице 7.1, взятой из "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide, Release 8.1.5" [стр. 4-14, 4-15, таблица 4.1].

Таблица 7.1. *Общие переменные ServiceCtx*

Строка символов	Значение
DEFAULT_SESSION	": default"
NON_SSL_CREDENTIAL	"Credential"
NON_SSL_LOGIN	"Login"
SSL_CREDENTIAL	"SecureCredential"
SSL_LOGIN	"SecureLogin"
SSL_30	"30"
SSL_20	"20"
SSL_30_WITH_20_HELLO	"30_WITH_20_HELLO"
THIS_SERVER	":thisServer"
THIS_SESSION	":thisSession"
Целочисленное название	**Целочисленный конструктор**
SESS_IIOP	New Integer (2)
IIOP	New Integer(1)

Для приложений CORBA и EJB доступны описанные далее методы `ServiceCtx`.

- `public Context createSubcontext (String name)`

Таким методом создается новый именованный сеанс. Параметр name – это имя сеанса, а возвращаемый объект – сеанс. Данный метод в совокупности с методом `SessionCtx.activate()`, о котором говорится в следующем разделе, используется для активации серверного объекта в новом сеансе, а не в сеансе по умолчанию.

- ```
  public static org.omg.CORBA.ORB init (
      String username,
      String password,
      String role,
      boolean ssl,
      java.util.Properties props )
  ```

Этот метод возвращает ссылку на ORB, создаваемый при поиске. Параметр `ssl` нужно устанавливать равным `true` для SSL-аутентификации. Метод используется клиентами не-JNDI (см. следующий раздел).

- `public synchronized SessionCtx login()`

Этот метод проверяет подлинность вызывающего, использующего свойства в среде начального контекста, а затем активирует новый сеанс и возвращает объект сеансового контекста.
  ```

- `public Object lookup ( String name )`

Этот метод ищет в экземпляре базы данных серверный объект, связанный с серверным контекстом и возвращает активированный экземпляр объекта. Следовательно, вызовом `lookup()` объекта `InitialContext` с `serviceURL` можно активировать службу и возвратить объект `ServiceCtx`, а последующим вызовом `lookup()` объекта `ServiceCtx` активировать указанный серверный объект (а не делать обе операции в одном вызове `lookup()`).

- `public Object _lookup ( String name )`

Этот метод идентичен методу `lookup()` за исключением того, что серверный объект не активируется.

Класс сеансового контекста

Класс `SessionCtx` расширяет JNDI-класс Context. Объекты `SessionCtx` представляют сеансы и содержат методы, позволяющие выполнять такие сеансовые операции, как аутентификация клиента или активация объектов.

- `public synchronized boolean login()`

Этот метод проверяет подлинность клиента, используя свойства среды начального контекста, переданные конструктору `InitialContext`: имя пользователя, пароль и роль.

- ```
  public synchronized boolean login (
  String username, String password, String role )
  ```

Этот метод проверяет подлинность клиента при помощи имени пользователя, пароля и роли, переданных в качестве параметров (роль необязательна и часто передается как `null`), и применяется обычно для аутентификации клиента в сеансе, явно начатом посредством `Context.createSubcontext()`.

- `public Object activate ( String name )`

Этот метод ищет и активирует серверный объект по указанному имени и используется, как правило, для активации серверного объекта в сеансе, явно начатом посредством `Context.createSubcontext()`.

Пример приложения

В этом разделе приведен пример приложения, в котором ищется `serviceURL`, после чего сеанс явно создается с помощью метода `SessionCtx.createSubcontext()`, затем клиент аутентифицируется методом `SessionCtx.login()`, а серверный объект явно активируется методом `SessionCtx.activate()`. Используется здесь серверный объект `Account` из главы 6. Для полноты картины включим IDL-файл для данного серверного объекта:

```
/* Имя файла: account.idl */

module account {
  struct AccountInfo { long departmentno; long projectno; };
  exception NoAccountError { wstring mess; };
  interface Account {
    AccountInfo getAccountInfo( in long accountno )
        raises( NoAccountError );
  };
};
```

Файл `AccountImpl.sqlj` нужно создать таким, чтобы объекты `Account` можно было активировать явным образом. Для этого `AccountImpl` должен реализовывать интерфейс `oracle.aurora.AuroraServices.ActivatableObject`, причем должен быть реализован метод `_intializeAuroraObject()` этого интерфейса (см. главу 6).

```
/* Имя программы:  AccountImpl.sqlj
**
```
```

```
** Назначение: Получить информацию о счете для данного номера счета.
**
*/
package accountServer;
import account.*;
import java.sql.*;
import oracle.aurora.AuroraServices.ActivatableObject;
public class AccountImpl extends _AccountImplBase {
  implements ActivatableObject {
  public AccountInfo getAccountInfo( int accountno )
    throws NoAccountError {
    try {

        // Необходимо инициализировать переменные для выбора в них информации.
        int departmentno = 0;
        int projectno = 0;

        #sql {
          SELECT departmentno, projectno
             INTO :departmentno, :projectno
                FROM ACCOUNT_LIST
                        WHERE accountno = :accountno
        };
        return new AccountInfo( departmentno, projectno );
    }
    catch( SQLException e ) {
        throw new NoAccountError( e.getMessage() );
    }
  }
    // Реализуем ActivatableObject._intializeAuroraObject().
    public org.omg.CORBA.Object _initializeAuroraObject() {return this;}
  }

  /* Имя программы: ExplicitAccountUser.java
  **
  ** Назначение:        С помощью серверного объекта Account получить
  **                    сведения о счете 1056, иллюстрируя этапы явного
  **                    создания, активации и аутентификации.
  **
  */
  import account.*;
  import accountServer.*;

  // Импортируем класс ServiceCtx и класс SessionCtx.
  import oracle.aurora.jndi.sess_iiop.ServiceCtx;
  import oracle.aurora.jndi.sess_iiop.SessionCtx;

  // Импортируем JNDI-интерфейс Context.
  import javax.naming.Context;

  /* Импортируем класс initialContext, реализующий интерфейс
     Context.
  */
  import javax.naming.InitialContext;

  /* Импортируем класс хэш-таблицы для хранения свойств Среды
     начального контекста.
  */
  import java.util.Hashtable;

  public class ExplicitAccountUser {

    // main устанавливает, а не перехватывает исключительные ситуации.
    public static void main( String[] args ) throws Exception {
```

```
            /* Подготовим упрощенный начальный контекст. Поскольку сеанс
               начинается явным образом, аутентификация должна быть явной.
            */
            Hashtable env = new Hashtable();
            env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
            Context ic = new InitialContext( env );

            /* Получим объект SessionCtx, представляющий экземпляр базы
               данных, с помощью serviceURL.
            */
            ServiceCtx service = ( ServiceCtx )
            ic.lookup( "sess_iiop://localhost:2481:ORCL" );

        // Создадим сеанс с именем :mysession в экземпляре.
        SessionCtx session = ( SessionCtx )
            service.createSubcontext( ":mysession" );

        // Выполним аутентификацию.
        session.login( "scott", "tiger", null );

        // Активируем объект Account.
        Account account = ( Account )
            session.activate( "/test/Account" );

        // Вызовем метод getAccountInfo().
        AccountInfo ai = account.getAccountInfo( 1056 );

        // Распечатаем сведения о счете.
        System.out.println( "Department number = " + ai.departmentno +
                            " Project number = " + ai.projectno );
    }
}
```

Управление сеансами

В этом разделе описан процесс явного управления сеансами, в частности:

- Запуск сеанса из серверного объекта
- Запуск именованного сеанса со стороны клиента
- Пример двухсеансовой программы
- Пример программы с указателями URL, содержащими имена сеансов
- Управление продолжительностью сеанса и окончание сеанса

Запуск сеанса из серверного объекта

Когда один серверный объект активирует другой с помощью того же URL, что и у клиента, новый серверный объект активируется в другом сеансе, нежели тот, где функционирует серверный объект, выполнявший поиск:

```
Hashtable env = new Hashtable();
env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
env.put( Context.SECURITY_PRINCIPAL, "scott" );
env.put( Context.SECURITY_CREDENTIALS, "tiger" );
env.put( Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext( env );
Account = ( Account ) ic.lookup(
    "sess_iiop://localhost:2481:ORCL/test/Account );
```

Если же нужно, чтобы новый серверный объект работал в том же сеансе, что и вызывающий, в URL следует указать не `хост_имя:порт:SID`, а `thisServer/:thisSession`. Изменим вызов `lookup()` на следующий:

```
Hashtable env = new Hashtable();
```

```
env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
Context ic = new InitialContext( env );
Account myObj = ( Account ) ic.lookup(
    "sess_iiop://thisServer/:this Session/test/Account );
```

Обратите внимание: здесь не нужно выполнять аутентификацию регистрации, так как клиент (в данном случае серверный объект) уже аутентифицирован. Напомним, что проверку подлинности для сеанса проходят не объекты, а клиенты. При каждом запуске отдельного сеанса, чего в данном случае не происходит, необходимо выполнять или аутентификацию регистрации, или SSL-аутентификацию.

Запуск именованного сеанса со стороны клиента

Если в URL, передаваемом методу `InitialContext.lookup()`, указать имя серверного объекта, но не указать явное имя сеанса (или указать сеанс по умолчанию), серверный объект активируется в сеансе `:default`. Если впоследствии клиент активирует дополнительные серверные объекты, они будут функционировать в том же сеансе по умолчанию. Для активации же серверного объекта в отдельном сеансе нужно:

1. Создать новый служебный контекст, вызвав `lookup()` с помощью service-URL:

   ```
   ServiceCtx service = ( Servicectx ) ic.lookup(
          "sess_iiop://localhost:2481:ORCL" );
   ```

2. Создать контекст нового сеанса, вызвав метод `createSubcontext()` служебного контекста:

   ```
   SessionCtx newsess = ( SessionCtx )
     service.createSubcontext ( ":mysession" );
   ```

3. Аутентифицировать клиента, вызвав метод `login()` нового сеанса:

   ```
   newsess.login( "scott", "tiger", "null" );
   ```

В следующем разделе приведен завершенный пример явного запуска двух сеансов.

Пример двухсеансовой программы

Здесь клиент явно начинает два различных сеанса и явно активирует серверный объект в каждом из них. Этот пример очень похож на предыдущий пример приложения за исключением того, что запускается не один, а два сеанса. Вновь воспользуемся объектом `Account` из главы 6. Для полноты картины повторим IDL-файл.

```
/* Имя файла: account.idl */

module account {
   struct AccountInfo { long departmentno; long projectno; };
   exception NoAccountError { wstring mess; };
   interface Account {
     AccountInfo getAccountInfo( in long accountno )
         raises( NoAccountError );
   };
};

/* Имя программы: AccounImpl.sqlj
**
** Назначение: Получить информацию о счете по данному номеру
**
*/
package accountServer;
import account.*;
import java.sql.*;
```

```
import oracle.aurora.AuroraServices.ActivatableObject;
public class AccountImpl extends _AccountImplBase {
   implements ActivatableObject {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {

         // Необходимо инициализировать переменные для выбора в них информации.
         int departmentno = 0;
         int projectno = 0;

         #sql {
           SELECT departmentno, projectno
              INTO :departmentno, :projectno
               FROM ACCOUNT_LIST
                   WHERE accountno = :accountno
         };
         return new AccountInfo( departmentno, projectno );
     }
     catch( SQLException e ) {
         throw new NoAccountError( e.getMessage() );
     }
   }

   // Реализуем ActivatableObject._intializeAuroraObject().
   public org.omg.CORBA.Object _initializeAuroraObject() {return this;}
}

/* Имя программы: AccountUser2.java
**
** Назначение:   С помощью серверного объекта Account получить
**               сведения о счетах 1056 и 2001.
**
*/
import account.*;
import accountServer.*;

// Импортируем класс ServiceCtx и класс SessionCtx.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс initialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class AccountUser2 {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {
    Hashtable env = new Hashtable();

    /* Подготовим упрощенный начальный контекст. Поскольку сеанс
       начинается явным образом, аутентификация должна быть явной.
    */
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    Context ic = new InitialContext( env );

    // Получим объект SessionCtx, представляющий экземпляр базы данных.
    ServiceCtx service = ( ServiceCtx )
        ic.lookup( "sess_iiop://localhost:2481:ORCL" );
```

```
        // Создадим сеанс с именем :mysession1 в экземпляре.
        SessionCtx session1 = ( SessionCtx )
            service.createSubcontext( ":mysession1" );

        // Выполним аутентификацию.
        session1.login( "scott", "tiger", null );

        // Создадим другой сеанс с именем :mysession2 в экземпляре.
        SessionCtx session2 = ( SessionCtx )
            service.createSubcontext( ":mysession2" );

        // Выполним аутентификацию.
        session2.login( "scott", "tiger", null );

        // Активируем объект Account в :mysession1.
        Account account1 =
            ( Account ) session1.activate( "/test/Account" );

        // Активируем объект Account в :mysession2.
        Account account2 =
            ( Account ) session2.activate( "/test/Account" );

        // Вызовем метод getAccountInfo() для account1 и account2.
        AccountInfo ai1 = account1.getAccountInfo( 1056 );
        AccountInfo ai2 = account2.getAccountInfo( 2001 );

        // Распечатаем сведения о счетах.
        System.out.println( "For account 1056: " +
                            "Department number = " + ai1.departmentno +
                            " Project number = " + ai1.projectno );
        System.out.println( "For account 2001: " +
                            "Department number = " + ai2.departmentno +
                            " Project number = " + ai2.projectno ); }
}
```

Пример программы с указателями URL, содержащими имена сеансов

Этот пример похож на предыдущий, за исключением того, что для активации серверных объектов используется метод `InitialContext.lookup()`, а не метод `SessionCtx activate()`. Для полноты картины вновь повторим IDL-файл:

```
/* Имя файла: account.idl */

module account {
   struct AccountInfo { long departmentno; long projectno; };
   exception NoAccountError { wstring mess; };
   interface Account {
     AccountInfo getAccountInfo( in long accountno )
         raises( NoAccountError );
   };
};
```

```
/* Имя программы:  AccounImpl.sqlj
**
** Назначение:  Получить информацию о счете по данному номеру.
**
*/
package accountServer;
import account.*;
import java.sql.*;

/* Обратите внимание: в AccountUser3 класс ActivatableObject
   необязателен, так как серверный объект активируется в вызовах
   lookup().
```

```
*/
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountImpl extends _AccountImplBase {
   implements ActivatableObject {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {

         // Необходимо инициализировать переменные для выбора в них информации.
         int departmentno = 0;
         int projectno = 0;

         #sql {
           SELECT departmentno, projectno
           INTO :departmentno, :projectno
              FROM ACCOUNT_LIST
                WHERE accountno = :accountno
         };
         return new AccountInfo( departmentno, projectno );
     }
     catch( SQLException e ) {
         throw new NoAccountError( e.getMessage() );
     }
   }

   // Реализуем ActivatableObject._intializeAuroraObject().
   */ Обратите внимание: в AccountUser3 этот метод необязателен, так
      как серверный объект активируется в вызовах lookup().
   */
   public org.omg.CORBA.Object _initializeAuroraObject() {return this;}
}

/* Имя программы:  AccountUser3.java
**
** Назначение:        С помощью серверного объекта Account получить
**                    сведения о счетах 1056 и 2001.
**
*/
import account.*;
import accountServer.*;

// Импортируем класс ServiceCtx и класс SessionCtx.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

/* Импортируем класс initialContext, реализующий интерфейс
   Context.
*/
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class AccountUser3 {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    /* Подготовим упрощенный начальный контекст. Поскольку сеанс
       начинается явным образом, аутентификация должна быть явной.
    */
```

```
        Hashtable env = new Hashtable ();
        env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
        Context ic = new InitialContext( env );

        /* Получим объект SessionCtx, представляющий экземпляр БД.
        ServiceCtx service = ( ServiceCtx )
            ic.lookup( "sess_iiop://localhost:2481:ORCL" );

        // Создадим сеанс с именем :mysession1 в экземпляре.
        SessionCtx session1 =
            ( SessionCtx ) service.createSubcontext( ":mysession1" );

        // Выполним аутентификацию.
        session1.login( "scott", "tiger", null );

        // Создадим другой сеанс с именем :mysession2 в экземпляре.
        SessionCtx session2 =
            ( SessionCtx ) service.createSubcontext( ":mysession2" );

        // Выполним аутентификацию.
        session2.login( "scott", "tiger", null );

        /* Активируем объекты с помощью полностью определенного URL,
           содержащего имя сеанса.
        */
        /* Обратите внимание: в начальном контексте необязательны
           сведения об аутентификации, так как клиент уже был проверен в
           сеансах.
        */
        Account account1 = ( Account ) ic.lookup(
            "sess_iiop://localhost:2481:ORCL/:mysession1/test/Account" );
        Account account2 = ( Account ) ic.lookup(
            "sess_iiop://localhost:2481:ORCL/:mysession2/test/Account" );

        // Вызовем метод getAccountInfo() для account1 и account2.
        AccountInfo ai1 = account1.getAccountInfo( 1056 );
        AccountInfo ai2 = account2.getAccountInfo( 2001 );

        // Распечатаем сведения о счетах.
        System.out.println( "For account 1056: " +
                            "Department number = " + ai1.departmentno +
                            " Project number = " + ai1.projectno );
        System.out.println( "For account 2001: " +
                            "Department number = " + ai2.departmentno +
                            " Project number = " + ai2.projectno );
    }
}
```

Управление продолжительностью сеанса и его завершение

В обычном режиме сеанс заканчивается тогда, когда закрывается последнее клиентское соединение. Однако метод `oracle.aurora.net.Presentation.sessionTimeout()` предоставляет серверному объекту возможность управлять продолжительностью сеанса. Параметром этого метода является значение тайм-аута (длительности) сеанса в секундах. Отсчет времени начинается после выполнения последних клиентских запросов. Таким образом, если значение таймаута равно 30, сеанс будет завершаться через каждые 30 секунд после обработки последнего клиентского запроса:

```
oracle.aurora.net.Presentation.sessionTimeout(30);
```

Заметим, что при использовании метода `sessionTimeout()` необходимо к своему `classpath` добавить:

- `$(ORACLE_HOME)/lib/aurora.zip`

Метод `exitSession()` из `oracle.aurora.vm.OracleRuntime` завершает сеанс связи с базой данных, что очень напоминает завершение программы с помощью `System.exit()`:

- `oracle.aurora.vm.OracleRuntime.exitSession(1);`

Целочисленный (`int`) параметр данного метода — это значение выхода, аналогичное значению, указываемому в `System.exit()`.

Аутентификация

Поскольку сервер Oracle — это защищенный сервер, клиент не может обращаться к данным, хранимым в базе, без предварительной аутентификации со стороны сервера Oracle. Серверные CORBA-объекты и зерна Enterprise JavaBean в Oracle8*i* исполняются в границах сервера Oracle, поэтому чтобы клиент мог активировать серверные объекты и вызывать их методы, он должен иметь:

- Возможность удостоверять перед сервером себя самого, передавая имя пользователя и пароль
- Право доступа к любому активируемому им объекту
- В некоторых случаях — привилегии execute на сам метод (например, в EJB права доступа можно устанавливать по типу "метод-метод")

Теперь поговорим о:

- Способах аутентификации клиента
- Правах доступа к объектам базы данных

Способы аутентификации клиента

Клиент может удостоверять себя перед сервером тремя способами, с помощью:

- Протокола регистрации Oracle8i по стандартному (не-SSL) соединению TCP/IP
- Протокола регистрации по соединению Secure Socket Layer
- Аутентификации на основе мандата по соединению Secure Socket Layer

Нужный способ аутентификации определяется по значению, атрибута `javax.naming.Context.SECURITY_AUTHENTICATION`, установленному при описании начального контекста JNDI. Допустимы следующие три значения:

- `ServiceCtx.NON_SSL_LOGIN` — протокол регистрации Oracle8*i* используется в стандартном соединении TCP/IP
- `ServiceCtx.SSL_LOGIN` — протокол регистрации используется в соединении Secure Socket Layer
- `ServiceCtx.SSL_CREDENTIAL` — мандатный протокол используется в соединении Secure Socket Layer

Если для атрибута `javax.naming.Context.SECURITY_AUTHENTICATION` никакого значения не задано, клиент будет активировать непосредственно протокол регистрации.

Теперь более внимательно рассмотрим:

- Протокол регистрации
- Мандатный протокол

Протокол регистрации

Протокол регистрации, или входа в систему (login), используется вместе с шифрованием Secure Socket Layer (SSL) или без него. Для приложения,

данные которого требуют защиты, следует воспользоваться SSL-соединением и установить значение SSL_LOGIN для атрибута SECONDARY_AUTHENTICATION. Допустимо и значение NON_SSL_LOGIN – тогда шифрованием защищается квитирование (handshaking) регистрации, но остальные операции взаимодействия клиента с сервером защищаются в меньшей степени.

Указав значения для каждой из четырех переменных JNDI-контекста (URL_PKG_PREFIXES, SECURITY_PRINCIPAL, SECURITY_CREDENTIALS, SECURITY_AUHENTICATION), мы запустим автоматическую регистрацию при первом вызове метода Context.lookup().

Заметим, что для протокола регистрации необходимы два компонента: клиентский и серверный. Клиентский компонент (Login) реализует клиентскую часть протокола квитирования регистрации и выступает в роли объекта-посредника для вызова серверного объекта регистрации. Объект Login находится в пакете oracle.aurora.client.

Серверный объект регистрации называется LoginServer и находится в пакете oracle.aurora.AuroraServices. В следующем варианте AccountUser показано, как явно аутентифицировать клиента с помощью объектов регистрации, а не вызовом метода Sessionctx.login(), который вызывает объекты регистрации автоматически:

```
/* Имя программы: AccounUser4.java
**
** Назначение:     С помощью серверного объекта Account получить
**                 информацию о счете 1056.
**
*/
import account.*;
import accountServer.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
public class AccountUser4 {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    // Подготовим упрощенный начальный контекст.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    Context ic = new InitialContext( env );

    // Получим объект SessionCtx, представляющий экземпляр БД.
    ServiceCtx service = ( ServiceCtx )
        ic.lookup( "sess_iiop://localhost:2481:ORCL" );

    // Создадим сеанс с именем :mysession в экземпляре.
    SessionCtx session1 =
        ( SessionCtx ) service.createSubcontext( ":mysession" );

    // Активируем объект LoginServer по стандартному имени etc/login.
    LoginServer login_server =
        ( LoginServer ) session.activate( "etc/login" );
    /* Создадим клиент регистрации и выполним аутентификацию с
       помощью протокола регистрации.
    */
    Login login = new Login( login_server );
    login.authenticate( "scott", "tiger", null );

    // Активируем объект Account.
```

```
        Account account = ( Account ) session.activate("/test/Account");

        // Вызовем метод getAccountInfo().
        AccountInfo ai = account.getAccountInfo( 1056 );

        // Распечатаем сведения о счете.
        System.out.println( "Department number = " + ai.departmentno +
                            " Project number = " + ai.projectno );
    }
}
```

Мандатный протокол

Применение типа аутентификации `ServiceCtx.SSL_CREDENTIAL` означает, что имя пользователя, пароль и роль передаются серверу при первом вызове метода по SSL-соединению. Поскольку SSL-соединение требует, чтобы пароль шифровался протоколом передачи данных, необходимости в квитировании, используемом протоколом регистрации, нет, и поэтому для SSL-соединений мандатный (credential) протокол несколько эффективнее.

Права доступа к объектам базы данных

Активировать объект и обращаться к его методам могут только те пользователи, которым были предоставлены права `EXECUTE` на Java-класс объекта, хранимого в базе.

Назначаются права `EXECUTE` на загружаемые Java-классы с помощью аргумента `-grant` команды `loadjava`.

Кроме того, предоставить полномочие `EXECUTE` на Java-класс, уже загруженный в базу данных, можно SQL-командой `GRANT`:

```
GRANT EXECUTE on "Account" to George
```

К опубликованным объектам вправе обращаться все пользователи, если только полномочия не управляются явным образом. Управлять полномочиями можно двумя способами:

- С помощью параметра `-grant` инструмента `publish` (см. главу 6)
- С помощью команд `chmod` и `chown` оболочки сеансов (см. приложение F)

Отметим, что полномочия опубликованных объектов могут отличаться от полномочий на базовые классы. Пользователь, желающий активировать опубликованный объект, должен иметь полномочие `EXECUTE` на него, а также на базовые классы.

Клиенты не-JNDI

Клиенты могут полностью отказаться от использования JNDI и подключаться к серверу Oracle, используя непосредственно методы `CosNaming`. Заметим, что в этом случае для аутентификации необходим класс Login. Приведем пример:

```
/* Имя программы: AccountUser5.java */

// Import объект CORBA и классы CosNaming.
import org.omg.CORBA.Object;
import org.omg.CosNaming.*;

import oracle.aurora.AuroraServices.*;
import oracle.aurora.client.Login;
import account.*;
import org.omg.CORBA.SystemException;
import accountServer.*;

public class AccountUser5 {
  public static void main( String[] args ) throws Exception {

    // Получим ссылку на объект NameService.
```

```
        PublishingContext pc = null;
        pc = VisiAurora.getNameService( "localhost", 2481, "ORCL" );

        // Получим ссылку на ранее опубликованный объект регистрации.
        PublishedObject lob = null;
        LoginServer serv = null;
        NameComponent[] name = new NameComponent[2];
        name[0] = new NameComponent( "etc", "" );
        name[1] = new NameComponent( "login", "" );

        // Найдем этот объект в службе имен.
        Object l = pc.resolve( name );

        // Убедимся в том, что объект опубликован.
        lob = PublishedObjectHelper.narrow(l);

        // Создадим и активируем этот объект.
        l = lob.activate_no_helper();
        serv = LoginServerHelper.narrow(l);

        // Создадим клиентский объект-посредник регистрации и выполним аутентификацию.
        Login login = new Login( serv );
        login.authenticate( "scott", "tiger", null );

        // Создадим и получим ссылку на объект Account.
      PublishedObject accountObj = null;
        name[0] = new NameComponent( "test", "" );
        name[1] = new NameComponent( "Account", "" );

        // Найдем этот объект в пространстве имен.
        Object a = pc.resolve( name );

        // Убедимся в том, что объект опубликован.
        accountObj = PublishedObjectHelper.narrow(a);

        // Создадим и активируем этот объект.
        a = accountObj.activate_no_helper();

        /* С помощью метода narrow() helper-класса передадим объект в объект Account.
           При использовании JNDI-метода  InitialContext.lookup() для получения ссылки
           на Account этот вызов выполняется автоматически.
        */
        Account account = null;
        account = AccountHelper.narrow(a);

        // Вызовем метод getAccountInfo().
        AccountInfo ai = account.getAccountInfo( 1056 );

        // Распечатаем сведения о счете.
        System.out.println( "Department number = " + ai.departmentno +
                            " Project number = " + ai.projectno );
    }
}
```

Итоги

В этой главе рассматривались различные сценарии аутентификации клиентов и активации серверных объектов. В частности, говорилось об исполнении серверных объектов в разнообразных конфигурациях сеансов. Кроме того, вы узнаете о том, как отказаться от использования JNDI и напрямую находить объекты с помощью методов CosNaming, и о том, что в этом случае аутентификация должна выполняться с применением класса Login.

В следующей главе мы расскажем об управлении транзакциями в приложениях CORBA, что гарантирует согласованную работу базы данных при выполнении последних. Управление транзакциями очень важно в тех конфигурациях сеансов, где функционируют серверные объекты. В частности, при использовании JTS в качестве транзакционного API все серверные объекты в транзакции должны работать в одном сеансе.

ГЛАВА 8

Управление транзакциями в CORBA

Тема главы – управление транзакциями в приложениях CORBA. Об управлении транзакциями в приложениях EJB уже шла речь в части II книги. По большому счету, транзакции в приложениях CORBA и EJB управляются одинаково, но есть ряд различий. В частности, для зерен EJB существует *декларативное транзакционное* (declarative transactional) средство, позволяющее разработчику объявлять (декларировать) транзакционные свойства приложения во время внедрения зерна, не создавая транзакционных программных конструкций. Однако разработчик CORBA, разрешающий транзакционные свойства серверных объектов, обязан обращаться к службе транзакций. Для этого служат транзакционные интерфейсы API. Обычно транзакционный API является отражением подмножества API службы объектных транзакций (OTS, Object Transaction Service) OMG. Два подобных отражения предлагаются сервером Oracle8*i*: служба транзакций Java (JTS, Java Transaction Service), входящая в Oracle версий 8.1.5 и 8.1.6, и API транзакций Java (JTA, Java Transaction API), по существу, заменивший JTS в Oracle 8.1.7 и специфицированный компанией Sun Microsystems. В этой главе мы расскажем о JTS и отличиях его от JTA.

Транзакция (transaction) – это такая единица работы в базе данных, которая:

- **Atomic (атомарна, т.е. неделима)** Все внесенные во время транзакции изменения отменяются, если неуспешно любое из них.
- **Consistent (согласована)** Транзакция сохраняет согласованность, т.е. переводит БД из одного согласованного состояния в другое.
- **Isolated (изолирована)** Одновременно выполняемые транзакции не мешают друг другу, т.е. промежуточные операции одной транзакции невидимы в другой, которая выполняется одновременно.
- **Durable (долговременна)** После завершения транзакции ее результаты сохраняются в базе данных.

В реляционной СУБД так называемые свойства ACID реализуются комбинацией механизмов восстановления в случае аварии и одновременного управления. В этой главе вы узнаете об управлении транзакциями CORBA в Oracle8*i*, а именно о:

- Ограниченных возможностях JTS
- Ограничении транзакций и контексте транзакций
- Интерфейсах службы транзакций
- Методах службы транзакций Java
- Использовании методов JTS в клиентских программных конструкциях CORBA
- Ограничении на стороне клиента
- Использовании методов JTS в серверных программных конструкциях CORBA
- Ограничении на стороне сервера
- Компоненте серверного объекта CORBA
- Клиенте, обращающемся к компоненту EJB посредством ограничения транзакций через класс TransactionService
- Клиенте CORBA, обращающемся к компоненту EJB и компоненту серверного объекта CORBA
- JTA

Ограниченные возможности JTS

Возможности реализации JTS в Oracle8*i* несколько сужены, поскольку эта служба предназначена по большей части для поддержки ограничения на стороне клиента. Ее недостатки:

- **Невозможность транзакций с несколькими ресурсами** JTS-транзакция Oracle8*i* не может охватывать несколько баз данных, сеансов и серверов, но способна охватывать несколько серверных объектов или несколько зерен EJB, только если они работают в одном сеансе на одном и том же сервере.

  Область действия транзакции определяется *контекстом транзакции* (transaction context). Транзакция может работать с несколькими объектами, и ее контекст разделяется между этими объектами. Однако транзакционные контексты никогда не распространяются за пределы сервера. Если серверный объект вызывает объект другого сервера, контекст транзакции не переносится.

  Следует заметить, что в API транзакций Java (JTA), речь о котором пойдет ниже, транзакции могут охватывать несколько баз данных и несколько серверов.

- **Невозможность вложения транзакций** При попытке начать новую транзакцию до завершения или отката текущей устанавливается исключительная ситуация `SubtransactionsUnavailable`.
- **Невозможность тайм-аута (задержки по времени)** Методы JTS, работающие с ним, например `set_timeout()`, действуют в режиме "отсутствие операций" ("no ops"), т.е. вызываются и не устанавливают исключительных ситуаций, но ничего не выполняют. В JTA тайм-аут действует.
- **Невозможность взаимодействия** Службы транзакций не взаимодействуют с другими реализациями OTS.

Ограничения транзакций и контекст транзакций

Транзакция характерна тем, что ее можно сделать *ограниченной* (demarcated), т.е. четко указать ее начало и окончание. JTS-транзакция начинается вызовом метода `TS.begin()`, а заканчивается вызовом метода `TS.commit()` или метода `TS.rollback()` (см. раздел "Методы службы транзакций Java" этой главы).

Обратите внимание: каждый SQL-оператор `INSERT`, `DELETE` и `UPDATE` серверного объекта должен исполняться в границах активной транзакции. Такие транзакции завершаются или откатываются JTS-методами `commit()` или `rollback()`. Завершение и откат не могут выполняться SQL-операторами `COMMIT` и `ROLLBACK`, которые неприменимы по отношению к "глобальной" транзакции, состоящей из серверных объектов CORBA и EJB. Однако оператор `INSERT`, `DELETE` или `UPDATE`, явно исполняемый клиентом, не является частью глобальной транзакции (он исполняется в отдельном сеансе, а не в сеансе серверных объектов), а значит, должен завершаться или откатываться SQL- операторами `COMMIT` и `ROLLBACK`.

Транзакции можно описывать как ограниченные на стороне клиента или на стороне сервера. В случае клиентского ограничения транзакционный клиент явно выполняет методы ограничения, начинающие и заканчивающие транзакции. Начало и окончание ограничения – это вызовы методов клиентской службы транзакций (см. раздел "Использование методов JTS в клиентских программных конструкциях CORBA" этой главы и пример ниже). Напомним, что даже если глобальная транзакция (операторы, к которым применимы методы JTS) ограничена на стороне клиента, она содержит только те операторы `INSERT`, `DELETE` и `UPDATE`, которые фигурируют в серверных объектах, а не те, что указаны непосредственно клиентом.

Транзакцию, ограниченную на стороне сервера, начинает и заканчивает серверный объект (см. раздел "Использование методов JTS в серверных

программных конструкциях CORBA" этой главы). Отметим, что транзакция может охватывать несколько объектов, любой из которых может заканчивать транзакцию.

Контекст транзакции (transaction context) – это псевдообъект, передаваемый клиентом серверному объекту или одним серверным объектом другому (когда серверный объект выступает в роли клиента, вызывая метод другого серверного объекта). Контекст фиксирует состояние транзакции.

После инициализации клиентской службы транзакций и вызова метода, начинающего транзакцию, эта служба неявно создает контекст транзакции и присваивает ему идентификационный номер транзакции. Затем клиентская служба транзакций распространяет контекст на каждый объект, вызываемый клиентом.

Контекст незаметно распространяется от инициатора транзакции на серверный объект. На стороне клиента контекст переносится на каждый вызов метода серверного объекта. На стороне сервера информация о контексте транзакции извлекается и делается доступной для серверного объекта.

Отметим, что при вызове клиентом нескольких серверных объектов CORBA и EJB контекст транзакции, начинающейся в одном из этих объектов, не распространяется на другие серверные объекты. Следовательно, транзакция, которая начинается в одном серверном объекте, вызываемом клиентом, не содержит обновлений, реализуемых в других серверных объектах, вызываемых тем же клиентом.

Интерфейсы службы транзакций

Для связи с JTS разработчик приложения может применять два класса:

- `TransactionService`.
- `UserTransaction`, реализуемый с помощью `oracle.aurora.jts.client.AuroraTransactionService`. Класс `UserTransaction` используется с зернами EJB (см. главу 4).

Класс `TransactionService` применяется для инициализации транзакции на стороне клиента. Для этого в клиентскую исходную программу необходимо импортировать несколько классов:

```
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.jts.*;
import oracle.aurora.jts.util.*;
```

Эти классы включаются в библиотечный файл `aurora_client.jar`, который должен быть указан в `classpath` при компиляции и исполнении всех исходных файлов, использующих JTS.

Вызывать можно следующий метод `AuroraTransactionService`:

```
public synchronized static void initialize(
  Context initialContext, String serviceName )
```

Назначением этого метода является инициализация контекста транзакции на стороне клиента. Параметр `initialContext` – это объект контекста, возвращаемый JNDI-конструктором `Context`, а параметр `serviceName` – полное имя службы, например:

```
sess_iiop://localhost:2481:ORCL
```

Так, в следующий программный фрагмент включен вызов метода `initialize()`:

```
Hashtable env = new Hashtable();
env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
env.put( Context.SECURITY_PRINCIPAL, "scott" ) ;
env.put( Context.SECURITY_CREDENTIALS, "tiger" );
env.put( Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context initialContext = new InitialContext( env );
```

```
AuroraTransactionService.initialize(
  initialContext, "sess_iiop://localhost:2481:ORCL" );
```

Методы службы транзакций Java

В пакете jts содержатся методы, используемые клиентом или серверным объектом для начала, завершения, отката транзакций и для выполнения служебных функций, например для установки таймаута транзакции. Эти JTS- методы могут применяться клиентами CORBA, клиентами EJB и серверными объектами CORBA. Разработчикам EJB, которым нужно управлять транзакциями с помощью зерен, следует пользоваться интерфейсом UserTransaction в состоянии управления со стороны зерна (см. главу 4).

Для работы с методами JTS нужно включить в исходный программный текст следующие операторы import:

```
import oracle.aurora.jts.util.TS;
import javax.jts.util.*;
import org.omg.CosTransactions.*;
```

Пакет oracle.aurora.jts.util входит в состав библиотечного файла aurora_client.jar, который должен быть указан в classpath для всех источников Java, использующих пакет jts.

Служба транзакций представлена следующими статическими (static) методами класса TS:

public static synchronized TransactionService getTS()

Метод getTS() возвращает объект службы транзакций. После этого для данного объекта вызывается метод getCurrent(), который возвращает псевдообъект Current — контекст транзакции. Именно методы commit() и rollback() псевдообъекта Current вызываются для настоящей обработки транзакции.

Следующий программный фрагмент иллюстрирует старт новой транзакции на стороне клиента, причем в первую очередь возвращается начальный контекст JNDI:

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.clien .AuroraTransactionService;
import javax.naming.Context;
import javax.naming.InitialCo .ext;
import java.util.Hashtable;
...
Context ic = new InitialContex( env );
...
AuroraTransactionService.initialize(
    ic, "sess_iiop://localhost:2481:ORCL" );
...
Account account = ( Account )ic.lookup(
    "sess_iiop://localhost:2481:ORCL/test/Account" );
oracle.aurora.jts.util.TS.getTS().getCurrent().begin() ;
```

Если служба транзакций недоступна, getTS() устанавливает исключительную ситуацию NoTransactionService.

Класс Current содержит следующие методы:

public void begin()

Этот метод начинает новую транзакцию и может устанавливать такие исключительные ситуации:

- NoTransactionService — устанавливается, если контекст транзакции не инициализирован.
- SubtransactionsUnavailable — устанавливается, если begin() вызывается до завершения или отката текущей транзакции.

public Control suspend()

Этот метод приостанавливает текущую транзакцию сеанса, разрешая начать новую транзакцию. Он возвращает псевдообъект Control, над которым будут выполняться все последующие вызовы resume(). Проиллюстрируем вызов suspend():

```
org.omg.CosTransactions.Control control =
  oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
```

Метод suspend() может устанавливать следующие исключительные ситуации:

- NoTransactionService — устанавливается, если контекст транзакции не инициализирован.
- TransactionDoesNotExist — устанавливается, если нет активного контекста транзакции. Это происходит, когда вызов suspend() следует за другим вызовом suspend(), без промежуточного вызова resume().

public void resume(Control control)

Этот метод возобновляет приостановленную транзакцию. Он вызывается после suspend(), чтобы снова начал действовать указанный контекст транзакции. Параметр control должен быть объектом Control транзакции, возвращенным предыдущим вызовом suspend() в том же сеансе. Проиллюстрируем вызов метода resume():

```
org.omg.CosTransactions.Control control =
   oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
...
oracle.aurora.jts.util.TS.getTS().getCurrent().resume( control );
```

Метод resume() устанавливает исключительную ситуацию InvalidControl, если значение параметра control равно null или неправильно.

public void commit(boolean report_heuristics)

Этот метод завершает (фиксирует) текущую транзакцию. Для параметра report_heuristics должно быть установлено значение false. Значение true для параметра report_heuristics устанавливалось бы, если бы требовалась дополнительная информация по двухфазному завершению. Однако в настоящее время JServer не поддерживает протокол двухфазного завершения для распределенных объектов, поэтому нужно обязательно устанавливать для report_heuristics значение false. Этот параметр введен для совместимости с будущими версиями JServer.

Метод commit() может устанавливать следующие исключительные ситуации:

- HeuristicMixed — устанавливается, если для report_heuristics было задано true, двухфазное завершение находится в процессе работы, эвристическое решение было принято, в транзакции были завершены какие-либо нужные операции обновления, а для ряда других в это время был произведен откат.
- HeuristicHazard — устанавливается, если для report_heuristics было задано true, двухфазное завершение находится в процессе работы, а для ряда нужных операций обновления был произведен откат.
- NoTransaction — устанавливается, если активной транзакции нет.

public void rollback()

Этот метод производит откат результатов текущей транзакции. Поскольку вызов rollback(), как и вызов commit(), заканчивает текущую транзакцию, при вызове после rollback() любого JTS-метода, кроме begin(), устанавливается исключительная ситуация NoTransaction.

Метод rollback() устанавливает исключительную ситуацию NoTransaction, если транзакция не находится в своем контексте.

public void rollback_only()

Этот метод изменяет транзакцию, связанную с текущим потоком, так, что единственным возможным результатом становится откат транзакции. Метод rollback_only() устанавливает исключительную ситуацию NoTransaction, если транзакция не находится в своем контексте.

public void set_timeout(int seconds)

Этот метод в настоящее время не поддерживается и при вызове не действует. Во всех случаях значением таймаута по умолчанию является 60 секунд.

public Status get_status()

Этот метод вызывается для выявления состояния текущей транзакции. Приведем несколько возможных возвращаемых значений:

- javax.transaction.Status.StatusActive
- javax.transaction.Status.StatusMarkedRollback
- javax.transaction.Status.StatusNoTransaction

Полный набор целочисленных значений состояния приведен в javax.transaction.Status.

public String get_transaction_name()

Этот метод возвращает имя текущей транзакции. Если get_transaction_name() вызывается при неактивной транзакции или вне контекста транзакции, возвращается пустая (null) строка символов.

Использование методов JTS в клиентских программных конструкциях CORBA

Для использования методов JTS в клиентских программах CORBA нужно:

1. Импортировать следующие пакеты:

   ```
   oracle.aurora.jts.client.AuroraTransactionService
   oracle.aurora.jts.util.TS
   org.omg.CosTransactions
   ```

2. Вызвать метод AuroraTransactionService.initialize(), передав ему параметры serviceURL и начального JNDI-контекста.
3. Начать транзакцию, вызвав:

   ```
   oracle.aurora.jts.util.TS.getTS().getCurrent().begin()
   ```

4. Закончить транзакцию, вызвав TS.getTS().getCurrent().commit(false) или TS.getTS().getCurrent().rollback().

Проиллюстрируем эти операции:

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.TS;
import org.omg.CosTransactions.*;
...
// Инициализируем контекст транзакции.
AuroraTransactionService.initialize( ic,
   "sess_iiop://localhost:2481:ORCL" );

// Начнем транзакцию.
TS.getTS().getCurrent().begin();
```

```
... //
Завершим транзакцию.
TS.getTS().getCurrent().commit( false );
```

Полный пример ограничения транзакции на стороне клиента приведен в следующем разделе.

Пример ограничения на стороне клиента

В этом примере к объекту Account из главы 6 добавляется метод deleteAccount(), который удаляет строку счета из таблицы ACCOUNT_LIST по указанному номеру счета этой строки. Метод deleteAccount() реализован с помощью SQLJ. Затем программируется клиент CORBA, вызывающий метод deleteAccount() в контексте транзакции.

Первый шаг — создать IDL-файл:

```
/* Имя файла: account.idl */

module account {
    struct AccountInfo { long departmentno; long projectno; };

    exception NoAccountError { wstring mess; };
    exception DeleteAccountError { wstring mess; };

    interface Account {
        AccountInfo getAccountInfo( in long accountno )
              raises( NoAccountError );
        void deleteAccount( in long accountno )
              raises( NoAccountError, DeleteAccountError );
    };
};
```

Следующий шаг — реализовать объект Account на SQLJ. Реализация на JDBC содержится на сайте издательства "Лори" (www.lory-press.ru) и на web-сайтах, указанных во введении к книге.

```
/* Имя программы: AccountImpl.sqlj
**
Назначение:   Реализовать объект Account. Предлагаются два метода:
**            один - для возвращения информации о счете,
**            другой - для удаления строки из ACCOUNT_LIST.
**
*/

package accountServer;

import account.*;
import java.sql.*;

public class AccountImpl extends _AccountImplBase {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {

         // Необходимо инициализировать переменные для выбора в них информации.
         int departmentno = 0;
         int projectno = 0;

         #sql {
           SELECT departmentno, projectno
              INTO :departmentno, :projectno
                 FROM ACCOUNT_LIST
                    WHERE accountno = :accountno
         };
         return new AccountInfo( departmentno, projectno );
```

```
        }

        catch( SQLException e ) {
            throw new NoAccountError( e.getMessage() );
        }
      }

      // Удалим нужный счет.
      public void deleteAccount( int accountno )
        throws NoAccountError, DeleteAccountError {
        try {
            int acctno = 0;
            #sql {
              SELECT accountno
                 INTO :acctno
                   FROM ACCOUNT_LIST
                      WHERE accountno = :accountno
            };
        }

        // Операция SELECT INTO закончилась неудачно из-за (возможно) отсутствия строки.
        catch( SQLException e ) {
            throw new NoAccountError( e.getMessage() );
        }

        try {
            #sql {
              DELETE FROM ACCOUNT_LIST
                 WHERE accountno = :accountno
            };
        }

        // Удаление неудачно.
        catch( SQLException e ) {
            throw new DeleteAccountError( e.getMessage() );
        }
      }
}
```

Наконец, создадим клиентскую программу:

```
/* Имя программы:  AccountUser6.java
**
** Назначение:     С помощью серверного объекта Account удалить
**                 счет 1056 из таблицы ACCOUNT_LIST.
**
*/
import account.*;
import accountServer.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс initialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

/* Импортируем класс хэш-таблицы для хранения свойств среды
   начального контекста.
*/
import java.util.Hashtable;

// Импортируем классы для обработки транзакций.
```

```
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

public class AccountUser6 {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    // Конкретизируем и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Инициализируем службу транзакций Aurora.
    AuroraTransactionService.initialize(
        ic, "sess_iiop://localhost:2481:ORCL" );

    // Найдем имя объекта, получив ссылку на объект.
    Account account = ( Account )
        ic.lookup( "sess_iiop://localhost:2481:ORCL/test/Account" );

    // Начнем транзакцию.
    TS.getTS().getCurrent().begin();

    // Удалим счет 1056.
    account.deleteAccount( 1056 );

    // Завершим транзакцию.
    TS.getTS().getCurrent().commit( false );
  }
}
```

Использование методов JTS в серверных программных конструкциях CORBA

Далее показано, как использовать методы JTS в серверных программах CORBA:

1. Импортировать `oracle.aurora.jts.TS` и `org.omg.CosTransactions.*`.
2. Не нужно инициализировать службу транзакций aurora, вызывая `AuroraTransactionService.initialize()` на сервере, так сервер делает это самостоятельно.
3. Начать транзакцию, вызвав `TS.getTS().getCurrent().begin()`. Это можно сделать в отдельном методе или в методе, исполняющем первую SQL-команду языка DML, с помощью SQLJ или JDBC. Транзакция охватывает методы, так как она действует в границах сеанса, где началась.
4. Закончить транзакцию, вызвав `TS.getTS().getCurrent().commit( false )` или `TS.getTS.getCurrent().rollback()`.

Полный пример ограничения транзакции на стороне сервера приведен в следующем разделе.

Пример ограничения на стороне сервера

В этом примере все методы для обработки транзакции вызываются в реализации серверного объекта. IDL-файл из предыдущего примера изменен так, чтобы обрабатывать исключительную ситуацию, которая будет передавать

обратно клиенту сообщения, переданные серверному объекту в результате неуспешного выполнения метода транзакции.

```
/* Имя файла: account.idl */

module account {

   struct AccountInfo { long departmentno; long projectno; };

   exception NoAccountError { wstring mess; };
   exception DeleteAccountError { wstring mess; };
   exception TransactionError { wstring mess; };

   interface Account {
     AccountInfo getAccountInfo( in long accountno )
         raises( NoAccountError );
     void deleteAccount( in long accountno )
         raises( NoAccountError, DeleteAccountError, TransactionError);
   };
};
```

Затем серверный объект реализуется для обработки транзакции. Здесь для реализации применяется SQLJ. Реализация на JDBC содержится на сайте издательства "Лори" (www.lory-press.ru) и на Web-сайтах, указанных во введении к книге.

```
/* Имя программы:  AccountImpl.sqlj
**
** Назначение:     Реализовать обьект Account. Предлагаются два метода:
**                 один - для возвращения информации о счете,
**                 другой - для удаления строки из ACCOUNT_LIST. Метод
**                 удаления вызывает методы обработки транзакции.
**
*/

package accountServer;

import account.*;
import java.sql.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

public class AccountImpl extends _AccountImplBase {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {

         // Необходимо инициализировать переменные для выбора в них информации.
         int departmentno = 0;
         int projectno = 0;

         #sql {
           SELECT departmentno, projectno
              INTO :departmentno, :projectno
                FROM ACCOUNT_LIST
                   WHERE accountno = :accountno
         };

         return new AccountInfo( departmentno, projectno );
     }

     catch( SQLException e ) {
         throw new NoAccountError( e.getMessage() );
     }
```

```
    }

    // Удалим нужный счет.
    public void deleteAccount( int accountno )
      throws NoAccountError, DeleteAccountError, TransactionError {

      // Начнем транзакцию.
      try {
          TS.getTS().getCurrent().begin();
      }

      // Начало неудачно.
      catch( Exception e ) {
          throw new TransactionError( "Begin failed: " + e);
      }

      try {
          int acctno = 0;
          #sql {
            SELECT accountno
               INTO :acctno
                  FROM ACCOUNT_LIST
                    WHERE accountno = :accountno
          };
      }

      /* Операция SELECT INTO закончилась неудачно из-за (возможно)
         отсутствия строки.
      */
      catch( SQLException e ) {
          throw new NoAccountError( e.getMessage() );
      }

      try {
          #sql {
            DELETE FROM ACCOUNT_LIST
               WHERE accountno = :accountno
          };
      }

      // Удаление неудачно.
      catch( SQLException e ) {
          throw new DeleteAccountError( e.getMessage() );
      }

      // Завершим транзакцию.
      try {
          TS.getTS().getCurrent().commit( false );
      }

      // Завершение неудачно.
      catch( Exception e ) {
          throw new TransactionError( "Commit failed: " + e);
      }
    }
}
```

Теперь клиентская программа освобождается от вызова методов для обработки транзакции:

```
/* Имя программы: AccountUser7.java
**

** Назначение:    С помощью серверного обьекта Account удалить
**                счет 1056 из таблицы ACCOUNT_LIST.
```

```
**
*/
import account.*;
import accountServer.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

/* Импортируем класс хэш-таблицы для хранения свойств среды
   начального контекста.
*/
import java.util.Hashtable;

public class AccountUser7 {

   // main устанавливает, а не перехватывает исключительные ситуации.
   public static void main( String[] args ) throws Exception {

     // Конкретизируем и заполним объект InitialContext.
     Hashtable env = new Hashtable();
     env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
     env.put( Context.SECURITY_PRINCIPAL, "scott" );
     env.put( Context.SECURITY_CREDENTIALS, "tiger" );
     env.put( Context.SECURITY_AUTHENTICATION,
         ServiceCtx.NON_SSL_LOGIN );
     Context ic = new InitialContext( env );

     // Найдем имя объекта, получив ссылку на объект.
     Account account = ( Account )
         ic.lookup( "sess_iiop://localhost:2481:ORCL/test/Account" );

     // Удалим счет 1056.
     account.deleteAccount( 1056 );
   }
}
```

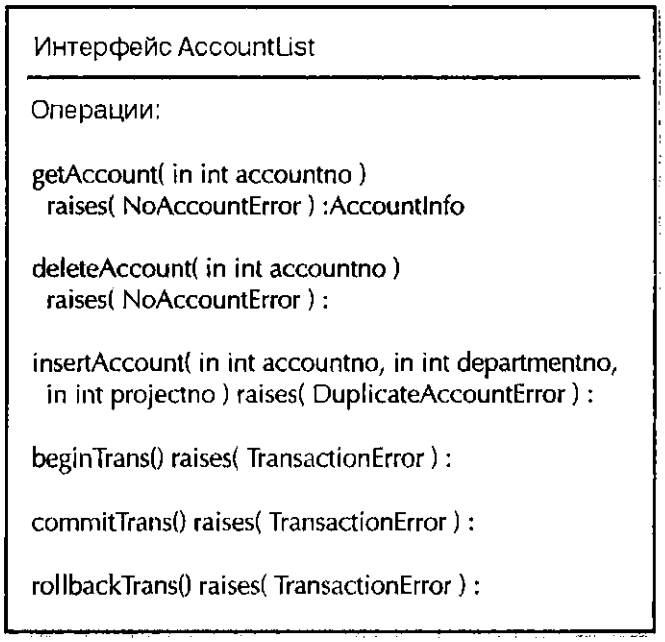
Компонент серверного объекта CORBA

В этом разделе создадим компонент серверного объекта CORBA для работы с таблицей ACCOUNT_LIST. Этот серверный объект расширяет и уточняет серверный объект, описанный в предыдущем разделе. Как и раньше, вся обработка транзакции выполняется в серверном объекте.

Что такое компонент? Как известно из прошлых глав книги, *компонент* (component) — это структурная единица с заранее оговоренными интерфейсами и с явными контекстными зависимостями. Компонент может быть внедрен независимо от других или включен в конструкцию посторонними (см. "Oracle8*i* SQLJ Programming [37, стр. 308]). Это значит, что компонент является программной единицей, которая достаточно хорошо описана и автономна и может быть без проблем внедрена отдельно или интегрирована с другими программными единицами. Компонент CORBA можно реализовать как одиночный серверный объект CORBA или как логически взаимосвязанную совокупность серверных объектов.

В этом разделе компонент для манипулирования таблицей ACCOUNT_LIST реализуется как одиночный серверный объект с именем AccountList. Этот серверный объект будет содержать методы для считывания строки в таблице ACCOUNT_LIST, для ввода строки в таблицу ACCOUNT_LIST, для удаления строки из таблицы ACCOUNT_LIST, а также для начала, завершения и отката транзакции

Рис. 8.1.
Интерфейс AccountList



(рис. 8.1). Кроме того, будет представлен клиент CORBA, использующий данный компонент.

В этом разделе описаны:

- Файл `accountlist.idl` для серверного объекта `AccountList`
- Реализация серверного объекта `AccountList`
- Клиент CORBA, использующий серверный объект `AccountList`

accountlist.idl

Далее приведен IDL-файл для серверного объекта `AccountList`:

```
/* Имя файла: accountlist.idl */

module accountlist {

   struct AccountInfo { long departmentno; long projectno; }

   exception NoAccountError { wstring mess; };
   exception TransactionError { wstring mess; };
   exception DuplicateAccountError { wstring mess; };
   exception DeleteAccountError { wstring mess; };

   interface AccountList {
     AccountInfo getAccountInfo( in long accountno )
         raises( NoAccountError );
     void deleteAccount( in long accountno )
         raises( NoAccountError, DeleteAccountError );
     void insertAccount(
         in long accountno, in long departmentno, in long projectno )
           raises( DuplicateAccountError );
     void beginTrans() raises( TransactionError );
     void commitTrans() raises( TransactionError );
     void rollbackTrans() raises( TransactionError );
   };
};
```

AccountListImpl.sqlj

Далее приведен класс реализации для серверного объекта AccountList, созданный на SQLJ. Реализация на JDBC содержится на сайте издательства "Лори" (www.lory-press.ru) и на Web-сайтах, указанных во введении к книге.

```
/* Имя программы: AccountListImpl.sqlj
**
** Назначение:   Реализовать объект AccountList для работы с таблицей
**               ACCOUNT_LIST.
**
*/

package accountlistServer;

import accountlist.*;
import java.sql.*;

import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

public class AccountListImpl extends _AccountListImplBase {

  // Выберем нужный счет.
  public AccountInfo getAccountInfo( int accountno )
    throws NoAccountError {
    try {

        // Необходимо инициализировать переменные для выбора в них информации.
        int departmentno = 0;
        int projectno = 0;

        #sql {
          SELECT departmentno, projectno
             INTO :departmentno, :projectno
               FROM ACCOUNT_LIST
                 WHERE accountno = :accountno
        };
        return new AccountInfo( departmentno, projectno );
    }

    // Выбор неудачен (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoAccountError( e.getMessage() );
    }
  }

  // Удалим нужный счет.
  public void deleteAccount( int accountno )
  throws NoAccountError, DeleteAccountError {
    try {
        int acctno = 0;
        #sql {
          SELECT accountno
             INTO :acctno
               FROM ACCOUNT_LIST
                 WHERE accountno = :accountno
        };
    }

    /* Операция SELECT INTO закончилась неудачно из-за (возможно)
       отсутствия строки.
    */
    catch( SQLException e ) {
        throw new NoAccountError( e.getMessage() );
```

```
  }

  try {
      #sql {
        DELETE FROM ACCOUNT_LIST
           WHERE accountno = :accountno
      };
  }

  // Удаление неудачно.
  catch( SQLException e ) {
      throw new DeleteAccountError( e.getMessage() );
  }
}

// Введем нужный счет.
public void insertAccount(
  int accountno, int departmentno, int projectno )
      throws DuplicateAccountError {
  try {
      #sql {
        INSERT INTO ACCOUNT_LIST
           VALUES(:accountno, :departmentno, :projectno )
      };
  }

  /* Ввод неудачен (возможно) из-за того, что номер счета уже
     присутствует в таблице.
  */
  catch( SQLException e ) {
      throw new DuplicateAccountError( e.getMessage() );
  }
}

// Начнем транзакцию.
public void beginTrans() throws TransactionError {
  try {
      TS.getTS().getCurrent().begin();
  }
  // Начало неудачно.
  catch( Exception e ) {
      throw new TransactionError( e.getMessage() );
  }
}

// Завершим транзакцию.
public void commitTrans() throws TransactionError {
  try {
      TS.getTS().getCurrent().commit( false );
  }
  // Завершение неудачно.
  catch( Exception e ) {
      throw new TransactionError( e.getMessage() );
  }
}

// Выполним откат транзакции.
public void rollbackTrans() throws TransactionError {
  try {
      TS.getTS().getCurrent().rollback();
  }
  // Откат неудачен.
  catch( Exception e ) {
```

```
            throw new TransactionError( e.getMessage() );
        }
      }
    }
```

Клиент CORBA для компонента AccountList

Далее описан клиент CORBA для серверного объекта AccountList:

```
/* Имя программы: AccountListUser.java
**
** Назначение:  С помощью компонента AccountList получить сведения о
**              счете 1056, удалить счет 1056 и ввести счет 2056.
**
*/
import accountlist.*;
import accountlistServer.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class AccountListUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
  throws Exception {

    // Конкретизируем и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Найдем имя объекта, получив ссылку на объект.
    AccountList account = ( AccountList )
        ic.lookup( "sess_iiop://localhost:2481:ORCL/test/AccountList");

    // Вызовем метод getAccountInfo() для получения сведений о счете 1056.
    AccountInfo ai = account.getAccountInfo( 1056 );

    // Распечатаем сведения о счете.
    System.out.println( "Department number = " + ai.departmentno +
                        " Project number = " + ai.projectno );

    // Начнем транзакцию.
    account.beginTrans();

    // Удалим счет 1056.
    account.deleteAccount( 1056 );

    // Введем счет 2056.
    account.insertAccount( 2056, 200, 200 );

    // Завершим транзакцию, состоящую из операций удаления и ввода.
```

```
        account.commitTrans();
      }
    }
```

Пример с приостановкой и возобновлением транзакции

Этот пример иллюстрирует приостановку текущей транзакции, начало и завершение новой транзакции и возобновление первоначальной транзакции с ее последующим завершением. Пример состоит из клиентской программы, в которой используется компонент AccountList, реализованный в предыдущем разделе и вводящий строки в таблицу ACCOUNT_LIST. Клиент применяет этот компонент для ввода еще одной строки в ACCOUNT_LIST во второй транзакции, а затем — для ввода строки в ACCOUNT_LIST в первоначальной транзакции.

В клиентской программе нельзя применять методы beginTrans() и commitTrans() серверного объекта AccountList, так как JTS-методы suspend() и resume() не работают с ограничением на стороне сервера.

```
/* Имя программы: AccountListUser2.java
**
** Назначение:      С помощью серверного объекта AccountList ввести
**                  строки в таблицу ACCOUNT_LIST в рамках одной
**                  транзакции, приостановить ее, начать вторую, ввести
**                  строку в ACCOUNT_LIST в рамках второй транзакции, а
**                  затем ввести строку в ACCOUNT_LIST в рамках первой
**                  транзакции.
**
*/
import accountlist.*;
import accountlistServer.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class AccountListUser2 {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
    throws Exception {

    // Конкретизируем и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Инициализируем службу транзакций Aurora.
```

```
        AuroraTransactionService.initialize(
            ic, "sess_iiop://localhost:2481:ORCL" );

        // Найдем имя объекта, получив ссылку на объект.
        AccountList account = ( AccountList )
            ic.lookup( "sess_iiop://localhost:2481:ORCL/test/AccountList");

        // Начнем транзакцию ACCOUNT_LIST.
        TS.getTS().getCurrent().begin();

        // Введем три записи в ACCOUNT_LIST.
        account.insertAccount( 3056, 200, 200 );
        account.insertAccount( 4056, 200, 200 );
        account.insertAccount( 5056, 200, 200 );

        // Приостановим первую транзакцию.
        Control c = TS.getTS().getCurrent().suspend();

        // Начнем вторую транзакцию.
        TS.getTS().getCurrent().begin();

        // Введем строку в ACCOUNT_LIST.
        account.insertAccount( 7056, 200, 200 );

        // Завершим вторую транзакцию.
        TS.getTS().getCurrent().commit( false );

        // Возобновим приостановленную транзакцию ACCOUNT_LIST.
        TS.getTS().getCurrent().resume(c) ;

        // Введем строку в ACCOUNT_LIST.
        account.insertAccount( 6056, 200, 200 );

        // Завершим первую транзакцию ACCOUNT_LIST.
        TS.getTS().getCurrent().commit( false );
      }
    }
```

Клиент, обращающийся к компоненту EJB

В этом разделе представлена клиентская программа, использующая EJB-компонент ObservationBean из главы 4 для ввода строки в таблицу OCEANIC_OBSERVATION_LIST. Эта программа выполняет всю обработку транзакции ввода с помощью CORBA-службы TransactionService, а не EJB-службы UserTransaction.

```
/* Имя программы: ObsRemoteUser.Java
**
** Назначение:   С помощью интерфейса ObsRemote ввести строку в
**               OCEANIC_OBSERVATION_LIST.
**
**
*/
import java.sql.*;
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем helper-классы.
import helpers.ObsHelper;

import obsserver.ObsRemote;
import obsserver.ObsHome;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
```

```
// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class ObsRemoteUser {

   // main устанавливает, а не перехватывает исключительные ситуации.
   public static void main( String[] args ) throws Exception {

     // Конкретизируем и заполним объект InitialContext.
     Hashtable env = new Hashtable();
     env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
     env.put( Context.SECURITY_PRINCIPAL, "scott" );
     env.put( Context.SECURITY_CREDENTIALS, "tiger" );
     env.put( Context.SECURITY_AUTHENTICATION,
         ServiceCtx.NON_SSL_LOGIN );
     Context ic = new InitialContext( env );

     // Инициализируем службу транзакций Aurora.
     AuroraTransactionService.initialize(
         ic, "sess_iiop://localhost:2481:ORCL" );

     ObsHome myObsHome = null;
     ObsRemote myObservationBean = null;

     // С помощью домашнего интерфейса найдем объект EJB.
     myObsHome = ( ObsHome )
         ic.lookup(
           "sess_iiop://localhost:2481:ORCL/test/ObservationBean" );

     // Создадим экземпляр зерна.
     myObservationBean = myObsHome.create();

     /* Вызовем метод Bean-класса для считывания строки
        OCEANIC_OBSRVATION_LIST с указанным obs_id = 2.
        Значением pressure будет 0, а не null.
     */
     ObsHelper obs = myObservationBean.getObsInfo(2);

     // Изменим дату.
     obs.when_t="07-JUN-2000";

     // Начнем транзакцию.
     TS.getTS().getCurrent().begin();

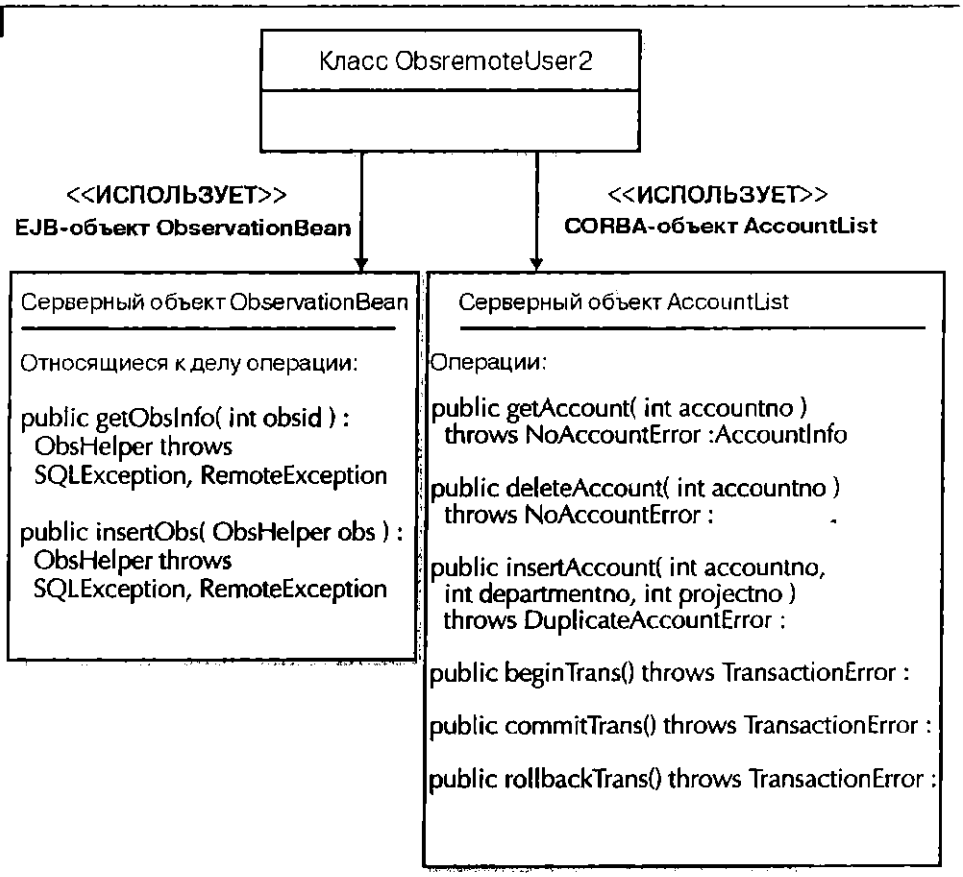
     /* Вызовем метод insertObj() для получения нового obs_id и
        ввода новой строки с этим obs_id в таблицу
        OCEANIC_OBSERVATION_LIST; в других отношениях строка такая
        же, что и строка для obs_id = 2, за исключением того, что
        у нее другая дата и значение pressure равно нулю.
     */
     ObsHelper oh = myObservationBean.insertObs( obs );

     // Завершим транзакцию.
     TS.getTS().getCurrent().commit( false );
   }
}
```

Клиент CORBA, обращающийся к компоненту EJB и компоненту серверного объекта CORBA

В этом разделе описана клиентская программа CORBA, в которой, как и в предыдущем примере, используется EJB-компонент `ObservationBean`, а также CORBA-компонент `AccountList` (рис. 8.2). Вся транзакционная обработка выполняется клиентом, а не транзакционными методами `AccountList`, так как вызов EJB клиентом не будет действовать, если ограничение транзакции JTS на стороне сервера выполнялось в вызываемом серверном объекте CORBA. Дело в том, что контекст транзакции не распространяется с серверного объекта на EJB.

Рис. 8.2. Клиент ObsRemoteUser2, использующий серверные объекты EJB и CORBA




```
/* Имя программы: ObsRemoteUser2.Java
**
** Назначение:   С помощью интерфейса ObsRemote ввести строку в
**               OCEANIC_OBSERVATION_LIST, а с помощью серверного
**               объекта AccountList ввести строку в ACCOUNT_LIST.
**
*/
import accountlist.*;
import accountlistServer.*,

import java.sql.*;
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем helper-классы.
import helpers.ObsHelper;
```

```
import obsserver.ObsRemote;
import obsserver.ObsHome;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class ObsRemoteUser2 {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    // Конкретизируем и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Инициализируем службу транзакций Aurora.
    AuroraTransactionService.initialize(
        ic, "sess_iiop://localhost:2481:ORCL" );

    // Найдем имя объекта AccountList, получив ссылку на объект.
    AccountList account = ( AccountList )
        ic.lookup( "sess_iiop://localhost:2481:ORCL/test/AccountList" );
    ObsHome myObsHome = null;
    ObsRemote myObservationBean = null;

    // С помощью домашнего интерфейса найдем объект EJB.
    myObsHome = ( ObsHome )
        ic.lookup(
          "sess_iiop://localhost:2481:ORCL/test/ObservationBean" );

    // Создадим экземпляр зерна.
    myObservationBean = myObsHome.create();
    /* Вызовем метод Bean-класса для считывания строки
       OCEANIC_OBSRVATION_LIST с указанным obs_id = 2.
       Значением pressure будет 0, а не null.
    */
    ObsHelper obs = myObservationBean.getObsInfo(2);

    // Изменим дату.
    obs.when_t="07-JUN-2000";

    // Начнем транзакцию.
    TS.getTS().getCurrent().begin();

    /* Вызовем метод insertObj() для получения нового obs_id и
       ввода новой строки с этим obs_id в таблицу
       OCEANIC_OBSERVATION_LIST; в других отношениях строка такая
       же, что и строка для obs_id = 2, за исключением того, что
       у нее другая дата и значение pressure равно нулю.
    */
    ObsHelper oh = myObservationBean.insertObs( obs );
```

```
        // Введем счет 7056.
        account.insertAccount( 7056, 200, 200 );

        // Завершим транзакцию.
        TS.getTS().getCurrent().commit( false );
      }
    }
```

JTA

В Oracle версии 8.1.7 главным транзакционным API, заменившим JTS, стал JTA. О некоторых преимуществах JTA по сравнению с JTS говорилось в разделе "Ограниченные возможности JTS" этой главы. В настоящем разделе рассказано о том, как реализовывать ограничение транзакций на стороне клиента и на стороне сервера с помощью JTA. В частности, здесь описаны:

- Структура клиента с ограничением JTA на стороне клиента
- Пример ограничения JTA на стороне клиента
- Структура реализации серверного объекта с ограничением JTA на стороне сервера

Структура клиента с ограничением JTA на стороне клиента

При использовании транзакционного прикладного интерфейса JTA для приложений CORBA, как и для приложений CORBA, применяется класс UserTransaction. Чтобы клиент смог приступить к работе, сначала необходимо осуществить привязку объекта UserTransaction в пространстве имен. Для этого служит команда bindut инструментального средства sess_sh. Например, чтобы привязать объект UserTransaction к имени "/test/hit" в пространстве имен, находящемся на хосте data-i.com, нужно выполнить следующее:

```
sess_sh -service jdbc:oracle:thin:@data-i.com:2481:ORCL
-user SCOTT -password TIGER & bindut /test/hit -url

jdbc:oracle:thin@data-i.com:2481:ORCL
```

В отношении обработки транзакций JTA структуру клиента нужно организовать следующим образом:

1. Указать местонахождение пространства имен, в котором привязан транзакционный объект. Это значение вводится в хэш-таблицу, которая будет использоваться для инициализации объекта InitialContext:

   ```
   Hashtable env = new Hashtable();
   env.put( jdbc_accessURLContextFactory.CONNECTION_URL_PROP,
     "jdbc:oracle:thin:data-i.com:2481:ORCL" );
    ...
   Context ic = new InitialContext ( env );
   ```

2. Зарегистрировать JDBC-драйвер OracleDriver:

   ```
   DriverManager.registerDriver( new
   oracle.jdbc.driver.OracleDriver() );
   ```

3. Считать объект UserTransaction в пространстве имен JNDI:

   ```
   UserTransaction ut = ( UserTransaction )
     ic.lookup( "jdbc_access:/test/hit" );
   ```

4. Начать транзакцию:

   ```
   ut.begin()
   ```

5. Считать серверные объекты и выполнить их методы. Это делается так же, как и при использовании JTS.

6. Завершить транзакцию:

```
ut.commit();
```

Пример ограничения JTA на стороне клиента

Отличия ограничения на стороне клиента транзакций JTA от ограничения на стороне клиента транзакций JTS рассмотрим на примере модифицированного клиента JTS из раздела "Пример ограничения на стороне клиента" этой главы.

```
/* Имя программы:    AccountUserJTA.java
**
** Назначение:       С помощью серверного объекта Account удалить
**                   счет 1056 из таблицы ACCOUNT_LIST.
**
*/
import account.*;
import accountServer.*;

import java.sql.DriverManager;
import javax.transaction.UserTransaction;
import javax.naming.NamingException;
import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс initialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

/* Импортируем класс хэш-таблицы для хранения свойств среды
   начального контекста.
*/
import java.util.Hashtable;

public class AccountUserJTA {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    // Конкретизируем и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    env.put( jdbc_accessURLContextFactory.CONNECTION_URL_PROP,
        "jdbc:oracle:thin:@data-i.com:2481:ORCL" );
    Context ic = new InitialContext( env );

    // Зарегистрируем JDBC-драйвер OracleDriver.
    DriverManager.registerDriver(
        new oracle.jdbc.driver.OracleDriver() );

    // Считаем объект UserTransaction в пространстве имен JNDI.
    UserTransaction ut = ( UserTransaction )
        ic.lookup ( "jdbc_access:/test/hit" );

    // Найдем имя серверного объекта, получив ссылку на объект.
    Account account = ( Account )
```

```
            ic.lookup( "sess_iiop://localhost:2481:ORCL/test/Account" );

        // Начнем транзакцию.
        ut.begin();

        // Удалим счет 1056.
        account.deleteAccount( 1056 );

        // Завершим транзакцию.
        ut.commit( false );
    }
}
```

Структура реализации серверного объекта с ограничением JTA на стороне сервера

Структура реализации серверного объекта с ограничением JTA на стороне сервера идентична структуре предыдущей программы, за исключением того, что при локальной привязке объекта UserTransaction необходимо лишь его JNDI-имя, а начальный контекст можно создавать без установки среды, так как значения среды уже известны:

```
InitialContext ic = new InitialContext();
UserTransaction ut = ( UserTransaction ) ic.lookup( "/test/hit" );
```

Выше говорилось об обработке транзакций JTA для локальной базы данных. За сведениями об обработке транзакций JTA, в которых задействовано несколько удаленных баз, рекомендуем обращаться к "CORBA Developer's Guide, version 8.1.7" [32].

Итоги

В этой главе говорилось о транзакционном управлении компонентами при помощи CORBA. В частности, были представлены недостатки API JTS (некоторые из них устранены в JTA, который будет доступен в Oracle версии 8.1.7) , а также примеры, иллюстрирующие ограничение транзакций JTS на стороне клиента и на стороне сервера. В заключение несколько слов было сказано о JTA, который, по существу, заменил JTS в Oracle версии 8.1.7, что в особенности касается приложений с ограничением на стороне сервера.

В следующей главе компоненты CORBA будут подвергнуты дальнейшему исследованию, для чего реализуется набор компонентов схемы Purchase Order.

ГЛАВА 9

Компоненты схемы Purchase Order

В этой главе мы перейдем на новый уровень проектирования и реализации компонентов CORBA, создав набор компонентов для обработки таблиц схемы Purchase Order (заказ на покупку), представленной в главе 1. С каждой таблицей будет работать один компонент. Если в таблице содержится внешний ключ, один из методов ее компонента будет возвращать всю информацию для указанной строки таблицы, в том числе информацию, содержащуюся в строке, на которую ссылается значение внешнего ключа данной строки. Кроме того, в компонент для таблицы PURCHASE_LIST будет добавлен метод, возвращающий все сведения о заказе на покупку, включая набор записей о пунктах данного заказа. Последний метод основан на инверсии "один-ко-многим" взаимоотношения внешнего ключа между пунктами ассортимента и покупками.

Для каждого компонента опишем требования, предъявляемые пользователями, и представим общий вид, определяющий структуру компонента. Рассмотрим реализацию и пример клиента для компонента. Заметим, что все транзакции будут ограничиваться в клиентских программах, поскольку ограничение на стороне сервера неэффективно (см. разделы "Ограничение транзакций и контекст транзакций" и "Пример с приостановкой и возобновлением транзакции" главы 8).

Будут созданы компоненты для всех таблиц схемы Purchase Order, кроме таблиц CREDITCARD_LIST, CHECKACCOUNT_LIST и ACCOUNT_LIST (компонент для таблицы ACCOUNT_LIST представлен в разделе "Компонент серверного объекта CORBA" главы 8). Компоненты для указанных таблиц можно создать аналогично компонентам для других таблиц, и пусть эта задача станет упражнением для читателя.

Итак, в этой главе создаются:

- CORBA-компонент DepartmentList (список отделов)
- CORBA-компонент EmployeeList (список служащих)
- CORBA-компонент VendorList (список поставщиков)
- CORBA-компонент ProjectList (список проектов)
- CORBA-компоненты PurchaseList (список покупок) и LineItemList (список пунктов ассортимента)

CORBA-компонент DepartmentList

Компонент DepartmentList создается в виде серверного объекта CORBA. Создание и применение такого компонента предполагают выполнение следующих действий:

- Определение требований, предъявляемых к компоненту пользователями
- Разработка общего вида компонента, являющегося его описанием
- Реализация общего вида
- Создание клиентов, использующих компонент

Хотя компонент не связывается с конкретным приложением, он создается для реализации требований пользователей, выполняющих определенную задачу (в этом разделе – задачу обработки таблицы DEPARTMENT_LIST). Пользователи, выполняющие задачу, предъявляют разработчикам свои требования.

Поскольку каждый компонент реализуется как серверный объект CORBA, в общем виде будут указаны операции (методы), которые должен предоставить интерфейс серверного объекта, а также будет определено взаимоотношение, существующее между интерфейсом серверного объекта для данного компонента и интерфейсами серверных объектов для других компонентов, необходимых данному.

Затем серверный объект материализуется созданием для него IDL-файла и класса его реализации. В IDL-файле серверный объект определяется как

интерфейс, который поддерживает методы, указанные в общем виде (вместе с полным описанием типов, необходимых интерфейсу), а класс реализации серверного объекта реализует эти методы с помощью Java.

В заключение компонент будет использован клиентом Java.

В этой главе описаны:

- Требования пользователей DepartmentList
- Общий вид DepartmentList
- IDL-файл для DepartmentList
- Реализация серверного объекта для DepartmentList
- Клиент для DepartmentList

Требования пользователей DepartmentList

Предположим, что пользователи хотят выполнять над таблицей DEPARTMENT_LIST следующие операции:

- По номеру отдела узнавать значения столбцов, не являющихся первичным ключом в строке DEPARTMENT_LIST.
- По номеру отдела удалять строку из DEPARTMENT_LIST.
- Вводить в таблицу DEPARTMENT_LIST строку с указанными значениями столбцов.

Общий вид DepartmentList

На рис. 9.1 представлен общий вид компонента, который отвечает указанным выше требованиям пользователей. Обратите внимание: на этом этапе можно определять исключительные ситуации, соответствующие ошибочным условиям.

Рис. 9.1. Общий вид DepartmentList

Интерфейс DepartmentList

Операции:

getDepartment(in long departmentno)
raises(NoDepartmentError, DeleteDepartmentError)

deleteDepartment(in long departmentno)
raises(NoDepartmentError) :

insertDepartment(
in long departmentno,
in wstring shortname,
in wstring longname)
raises(DuplicateDepartmentError) :

IDL-файл для DepartmentList

Следующий IDL-файл является спецификацией компонента DepartmentList, приведенного на рис. 9.1:

```
/* Имя файла: departmentlist.idl */

module departmentlist {
    struct DepartmentInfo { wstring shortname; wstring longname; };

    exception NoDepartmentError { wstring mess; };
    exception DuplicateDepartmentError { wstring mess; };
    exception DeleteDepartmentError { wstring mess; };
```

```
    interface DepartmentList {
        DepartmentInfo getDepartmentInfo( in long departmentno )
              raises( NoDepartmentError );
        void deleteDepartment( in long departmentno )
              raises( NoDepartmentError, DeleteDepartmentError );
        void insertDepartment( in long departmentno,
              in wstring shortname, in wstring longname )
                raises( DuplicateDepartmentError );
    };
};
```

Реализация серверного объекта для DepartmentList

В следующем файле SQLJ содержится программный текст класса DepartmentListImpl, который реализует операции, указанные в приведенном выше IDL-файле:

```
/* Имя программы:    DepartmentListImpl.sqlj
**
** Назначение:       Реализовать серверный объект DepartmentList для
**                   работы с таблицей DEPARTMENT_LIST.
**
*/

package departmentlistServer;

import departmentlist.*;
import java.sql.*;

public class DepartmentListImpl extends _DepartmentListImplBase {

  // Считываем сведения о нужном отделе.
  public DepartmentInfo getDepartmentInfo( int departmentno )
    throws NoDepartmentError {
    try {
        // Необходимо инициализировать переменные для размещения в них информации.
        String shortname = null;
        String longname = null;

        #sql {
          SELECT shortname, longname
             INTO :shortname, :longname
               FROM DEPARTMENT_LIST
                 WHERE departmentno = :departmentno
        };
        return new DepartmentInfo( shortname, longname );
    }

    // Выбор неудачен (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoDepartmentError( e.getMessage() );
    }
  }

  // Удалим сведения об отделе.
  public void deleteDepartment( int departmentno )
    throws NoDepartmentError, DeleteDepartmentError {
    try {
        int deptno = 0 ;
        #sql {
          SELECT departmentno
             INTO :deptno
               FROM DEPARTMENT_LIST
                 WHERE departmentno = :departmentno
        };
```

```
        }

        // Операция SELECT INTO неудачна (возможно) из-за отсутствия строки.
        catch( SQLException e ) {
            throw new NoDepartmentError( e.getMessage() );
        }

        try {
            #sql {
              DELETE FROM DEPARTMENT_LIST
                 WHERE departmentno = :departmentno
            };
        }

        // Удаление неудачно.
        catch( SQLException e ) {
            throw new DeleteDepartmentError( e.getMessage() );
        }
      }

      // Введем сведения о новом отделе.
      public void insertDepartment(
        int departmentno, String shortname, String longname )
            throws DuplicateDepartmentError {
        try {
            #sql {
              INSERT INTO DEPARTMENT_LIST
                 VALUES( :departmentno, :shortname, :longname )
            };
        }

        /* Ввод неудачен (возможно) из-за того, что номер отдела уже
           присутствует в таблице.
        */
        catch( SQLException e ) {
            throw new DuplicateDepartmentError( e.getMessage() );
        }
      }
}
```

Клиент для DepartmentList

Следующий исходный Java-файл содержит Java-класс, который использует компонент DepartmentList, вызывая методы getDepartmentInfo(), deleteDepartment() и insertDepartment():

```
/* Имя программы:    DepartmentListUser.java
**
** Назначение:       С помощью компонента DepartmentList получить сведения
**                   об отделе 200, удалить сведения об отделе 200 и
**                   ввести сведения об отделе 2056.
**
*/

// Импортируем классы компонента.
import departmentlist.*;
import departmentlistServer.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
```

```
// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class DepartmentListUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
    throws Exception {

    // Создадим экземпляр и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Инициализируем службу транзакций Aurora.
    AuroraTransactionService.initialize(
        ic, "sess_iiop://localhost:2481:ORCL" );

    // Найдем имя объекта, получив ссылку на объект.
    DepartmentList department = ( DepartmentList )
        ic.lookup(
          "sess_iiop://localhost:2481:ORCL/test/DepartmentList");

    // Вызовем метод getDepartmentInfo() для получения сведений об отделе 200.
    DepartmentInfo di = department.getDepartmentInfo( 200 );

    // Распечатаем сведения об отделе.
    System.out.println( "Short name = " + di.shortname +
                        " Long name = " + di.longname );

    // Начнем транзакцию.
    TS.getTS().getCurrent().begin();

    // Удалим сведения об отделе 200.
    department.deleteDepartment( 200 );

    // Введем сведения об отделе 2056.
    department.insertDepartment( 2056, "M", "Morgue" );

    // Завершим транзакцию, состоящую из операций удаления и ввода.
    TS.getTS().getCurrent().commit( false );
  }
}
```

CORBA-компонент EmployeeList

CORBA-компонент EmployeeList несколько сложнее компонента DepartmentList тем, что нужно считывать еще и информацию об отделе, на который ссылается запись о служащем. Это осуществляется при помощи внешнего ключа departmentno и обусловливается требованиями, предъявляемыми пользователями.

В контексте изучения того, как описывается и реализуется данная операция, рассматриваются следующие моменты:

- Требования пользователей EmployeeList
- Общий вид EmployeeList

- IDL-файл для EmployeeList
- Реализация серверного объекта для EmployeeList
- Клиент для EmployeeList

Требования пользователей EmployeeList

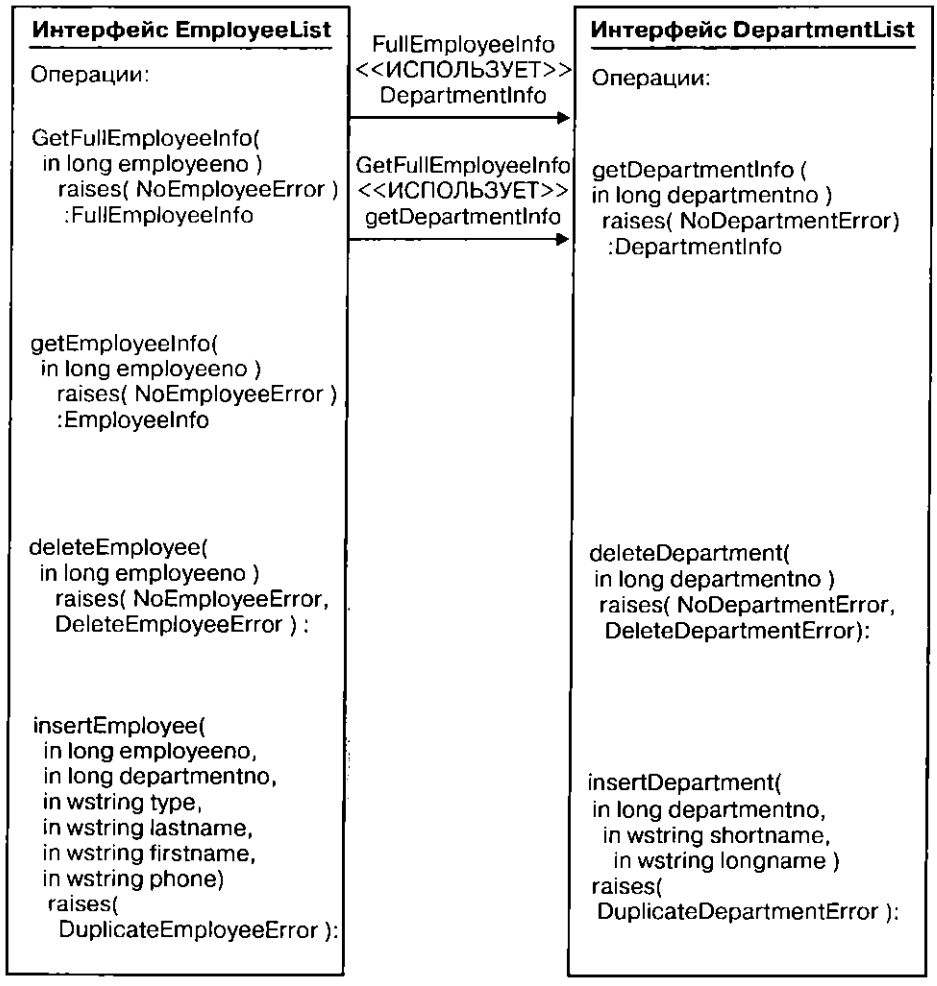
Пользователи хотят выполнять над компонентом следующие операции:

- По номеру служащего получать другие значения строки таблицы EMPLOYEE_LIST.
- По номеру служащего получать всю информацию о нем, в том числе значения столбцов, не являющихся первичным ключом, в соответствующей строке EMPLOYEE_LIST, и значения столбцов, не являющихся первичным ключом, в строке DEPARTMENT_LIST, на которую ссылается поле departmentno строки EMPLOYEE_LIST.
- По номеру служащего удалять соответствующую строку из EMPLOYEE_LIST.
- Вводить в EMPLOYEE_LIST строку с указанными значениями столбцов.

Общий вид EmployeeList

На рис. 9.2 представлен общий вид компонента EmployeeList, который отвечает требованиям пользователей. Решено, что операция, выдающая всю информацию о служащем, будет возвращать объект, который содержит ссылку на объект DepartmentInfo, описанный для компонента DepartmentList, и вызывать метод getDepartmentInfo() из DepartmentList, получая объект DepartmentInfo.

Рис. 9.2. Общий вид EmployeeList



IDL-файл для EmployeeList

В следующем IDL-файле описывается большая часть спецификации компонента EmployeeList, представленного на рис. 9.2 (тот факт, что getFullEmployeeInfo() вызывает getDepartmentInfo(), не отражен в IDL-файле).

Поскольку методу getFullEmployeeInfo() компонента EmployeeList необходима структура DepartmentInfo компонента DepartmentList, IDL-файл для компонента EmployeeList должен содержать IDL-файл для компонента DepartmentList.

```
/* Имя файла: employeelist.idl */

// Подключим IDL-файл для объекта, на который производится ссылка.
#include <departmentlist.idl>

module employeelist {

   struct EmployeeInfo {
     long departmentno; wstring type;
         wstring lastname; wstring firstname; wstring phone;
   };

   /* В FullEmployeeInfo хранится ссылка на объект DepartmentInfo для
      служащего, а также информация из EMPLOYEE_LIST.
     */
   struct FullEmployeeInfo {
     departmentlist::DepartmentInfo dinfo;
     long departmentno;
         wstring type; wstring lastname;
           wstring firstname; wstring phone;
   };

   exception NoEmployeeError { wstring mess; };
   exception DuplicateEmployeeError { wstring mess; };
   exception DeleteEmployeeError { wstring mess; };

   interface EmployeeList {
     EmployeeInfo getEmployeeInfo( in long employeeno )
         raises( NoEmployeeError );
     FullEmployeeInfo getFullEmployeeInfo( in long employeeno )
         raises( NoEmployeeError );
     void deleteEmployee( in long employeeno )
         raises( NoEmployeeError, DeleteEmployeeError );
     void insertEmployee(
         in long employeeno, in long departmentno, in wstring type,
           in wstring lastname, in wstring firstname, in wstring phone)
             raises( DuplicateEmployeeError );
   };
};
```

Реализация серверного объекта для EmployeeList

Самым сложным для реализации методом EmployeeListImpl.sqlj является метод getFullEmployeeInfo(). Он обращается к другому серверному объекту – к объекту DepartmentList, чтобы, вызвав его метод getDepartmentInfo(), получить сведения об отделе для служащего. Отметим, что в serviceURL поиска в DEPARTMENT_LIST указано thisServer, что вынуждает серверный объект DepartmentList выполняться в том же сеансе, в каком действует серверный объект EmployeeList (см. раздел "Запуск сеанса из серверного объекта" главы 7). Без указания thisServer пример не будет работать, так как транзакция JTS не может охватывать несколько сеансов.

```
/* Имя программы: EmployeeListImpl.sqlj
**
**
** Назначение:      Реализовать серверный объект EmployeeList для работы
```

```
**                  с таблицей EMPLOYEE_LIST.
**
*/

package employeelistServer;

import java.sql.*;
import employeelist.*;

// Импортируем классы для компонента DepartmentList.
import departmentlist.*;
import departmentlistServer.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class EmployeeListImpl extends _EmployeeListImplBase {

  // Считываем сведения о нужном служащем.
  public EmployeeInfo getEmployeeInfo( int employeeno )
    throws NoEmployeeError {
    try {

        // Необходимо инициализировать переменные для размещения в них информации.
        int departmentno = 0;
        String type = null;
        String lastname = null;
        String firstname = null;
        String phone = null;

        #sql {
          SELECT departmentno, type, lastname, firstname, phone
             INTO :departmentno, :type, :lastname, :firstname, :phone
               FROM EMPLOYEE_LIST
                 WHERE employeeno = :employeeno
        };
        return new EmployeeInfo(
          departmentno, type, lastname, firstname, phone );
    }

    // Выбор неудачен (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoEmployeeError( e.getMessage() );
    }
  }

  // Считываем всю информацию о служащем, включая сведения о его отделе.
  public FullEmployeeInfo getFullEmployeeInfo( int employeeno )
    throws NoEmployeeError {
    try {

        // Получим сведения о служащем.
        EmployeeInfo ei = getEmployeeInfo( employeeno );

        // Получим сведения об отделе служащего.

        // Создадим экземпляр и заполним объект InitialContext.
```

```
        Hashtable env = new Hashtable();
        env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
        Context ic = new InitialContext( env );

        // Найдем имя объекта DepartmentList, получив ссылку на объект.
        DepartmentList department = ( DepartmentList )
          ic.lookup( "sess_iiop://thisServer/test/DepartmentList" );

        // Вызовем метод getDepartment() для получения сведений об отделе.
        DepartmentInfo di =
          department.getDepartmentInfo( ei.departmentno );

        /* Возвратим объект FullEmployeeInfo, содержащий ссылку на
           объект DepartmentInfo для отдела служащего.
        */
        return(
          new FullEmployeeInfo(
            di, ei.departmentno, ei.type,
              ei.lastname, ei.firstname, ei.phone ) );
      }

      // Перехватим все установленные исключительные ситуации.
      catch( Exception e ) {
        throw new NoEmployeeError( e.getMessage() );
      }
    }

    // Удалим сведения о нужном служащем.
    public void deleteEmployee( int employeeno )
      throws NoEmployeeError, DeleteEmployeeError {
      try {
        int empno = 0;
        #sql {
          SELECT employeeno
            INTO :empno
              FROM EMPLOYEE_LIST
                WHERE employeeno = :employeeno
        };
      }

      // Операция SELECT INTO неудачна (возможно) из-за отсутствия строки.
      catch( SQLException e ) {
        throw new NoEmployeeError( e.getMessage() );
      }
      try {
        #sql {
          DELETE FROM EMPLOYEE_LIST
            WHERE employeeno = :employeeno
          };
        }

        // Удаление неудачно.
        catch( SQLException e ) {
          throw new DeleteEmployeeError( e.getMessage() );
        }
      }

      // Введем сведения о новом служащем.
      public void insertEmployee(
        int employeeno, int departmentno, String type,
          String lastname, String firstname, String phone )
            throws DuplicateEmployeeError {
        try {
        #sql {
          INSERT INTO EMPLOYEE_LIST
```

```
                VALUES( :employeeeno, :departmentno, :type,
                  :lastname, :firstname, :phone )
            };
        }

        /* Ввод неудачен (возможно) из-за того, что номер служащего уже
           присутствует в таблице.
        */
        catch( SQLException e ) {
            throw new DuplicateEmployeeError( e.getMessage() );
        }
      }
}
```

Клиент для EmployeeList

Следующая клиентская программа CORBA обращается к серверному объекту EmployeeList, вызывает метод getFullEmployee(), чтобы получить сведения о конкретном служащем (в том числе сведения о его отделе), удаляет сведения о том же служащем из таблицы EMPLOYEE_LIST и вводит нового служащего в таблицу EMPLOYEE_LIST:

```
/* Имя программы: EmployeeListUser.java
**
** Назначение:      С помощью компонента EmployeeList получить все
**                  сведения о служащем 104, удалить сведения о служащем
**                  104 и ввести сведения о служащем 2056.
**
*/

// Импортируем классы компонента.
import employeelist.*;
import employeelistServer.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class EmployeeListUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
    throws Exception {

    // Создадим экземпляр и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );
```

```
        // Инициализируем службу транзакций Aurora.
        AuroraTransactionService.initialize(
            ic, "sess_iiop://localhost:2481:ORCL" );

        // Найдем имя объекта, получив ссылку на объект.
        EmployeeList employee = ( EmployeeList )
            ic.lookup(
              "sess_iiop://localhost:2481:ORCL/test/EmployeeList" );

        /* Вызовем метод getFullEmployeeInfo() для получения сведений о
           служащем 104.
        */
        FullEmployeeInfo fi = employee.getFullEmployeeInfo( 104 );

        // Распечатаем сведения о служащем.
        System.out.println( "Type = " + fi.type +
                            " Last name = " + fi.lastname +
                            " First name = " + fi.firstname +
                            " Phone = " + fi.phone );

        // Распечатаем сведения об отделе служащего.
        System.out.println( "Department number = " + fi.departmentno +
                            " Short name = " + fi.dinfo.shortname +
                            " Long name = " + fi.dinfo.longname );

        // Начнем транзакцию.
        TS.getTS().getCurrent().begin() ;

        // Удалим сведения о служащем 104.
        employee.deleteEmployee( 104 );

        // Введем сведения о служащем 2056.
        employee.insertEmployee( 2056, 200, "Surgeon",
            "Smith", "Garrett", "999-9999" );

        // Завершим транзакцию, состоящую из операций удаления и ввода.
        TS.getTS().getCurrent().commit( false );
    }
}
```

CORBA-компонент VendorList

Поскольку в таблице VENDOR_LIST нет внешних ключей, компонент VendorList довольно прост и похож на компонент DepartmentList. Ниже описываются:

- Требования пользователей VendorList
- Общий вид VendorList
- IDL-файл для VendorList
- Реализация серверного объекта для VendorList
- Клиент для VendorList

Требования пользователей VendorList

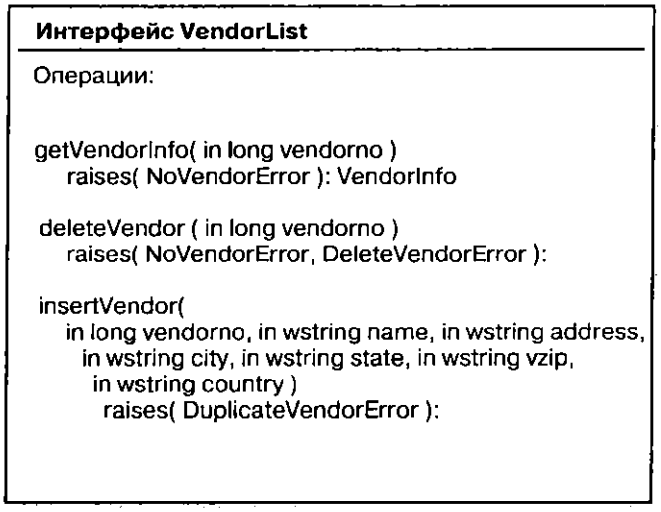
Требования пользователей VendorList схожи с требованиями пользователей DepartmentList. Пользователи хотят выполнять следующие операции:

- По номеру поставщика узнавать значения столбцов, не являющихся первичным ключом, в строке VENDOR_LIST.
- По номеру поставщика удалять строку из VENDOR_LIST.
- Вводить в таблицу VENDOR_LIST строку с указанными значениями столбцов.

Общий вид VendorList

На рис. 9.3 представлен приемлемый общий вид VendorList, который соответствует требованиям, предъявляемым пользователями.

Рис. 9.3. Общий вид VendorList



IDL-файл для VendorList

Следующий IDL-файл отражает общий вид компонента VendorList:

```
/* Имя файла: vendorlist.idl */

module vendorlist {

   struct VendorInfo {
     wstring name; wstring address; wstring city;
         wstring state; wstring vzip; wstring country;
   };

   exception NoVendorError { wstring mess; };
   exception DuplicateVendorError { wstring mess; };
   exception DeleteVendorError { wstring mess; };

   interface VendorList {
     VendorInfo getVendorInfo( in long vendorno )
         raises( NoVendorError );
     void deleteVendor ( in long vendorno )
         raises( NoVendorError, DeleteVendorError );
     void insertVendor(
         in long vendorno, in wstring name, in wstring address,
           in wstring city, in wstring state, in wstring vzip,
             in wstring country )
               raises( DuplicateVendorError );
   };
};
```

Реализация серверного объекта для VendorList

В следующем файле SQLJ содержится класс VendorListImpl, реализующий серверный объект:

```
/* Имя программы: VendorListImpl.sqlj
**
** Назначение:   Реализовать серверный объект VendorList для
**               работы с таблицей VENDOR_LIST.
**
```

```
*/

package vendorlistServer;

import vendorlist.*;
import java.sql.*;

public class VendorListImpl extends _VendorListImplBase {

   // Считываем сведения о нужном поставщике.
   public VendorInfo getVendorInfo( int vendorno )
     throws NoVendorError {
     try {

         /* Необходимо инициализировать переменные для размещения в них
            информации.
         */
         String name = null;
         String address = null;
         String sity = null;
         String state = null;
         String vzip = null;
         String country = null;

         #sql {
           SELECT name, address, city, state, vzip, country
              INTO :name, :address, :city, :state, :vzip, :country
                FROM VENDOR_LIST
                  WHERE vendorno = :vendorno
         };
         return new VendorInfo(
           name, address, city, state, vzip, country );
     }

     // Выбор неудачен (возможно) из-за отсутствия строки.
     catch( SQLException e ) {
         throw new NoVendorError( e.getMessage() );
     }
   }

   // Удалим сведения о поставщике.
   public void deleteVendor ( int vendorno )
     throws NoVendorError, DeleteVendorError {
     try {
         int vendno = 0 ;
         #sql {
           SELECT vendorno
              INTO :vendno
                FROM VENDOR_LIST
                   WHERE vendorno = :vendorno
         };
     }

     // Операция SELECT INTO неудачна (возможно) из-за отсутствия строки.
     catch( SQLException e ) {
         throw new NoVendorError( e.getMessage() );
     }
     try {
         #sql {
           DELETE FROM VENDOR_LIST
              WHERE vendorno = :vendorno
         };
     }

     // Удаление неудачно.
```

```
      catch( SQLException e ) {
          throw new DeleteVendorError( e.getMessage() );
      }
    }

    // Введем сведения о новом поставщике.
    public void insertVendor (
      int vendorno, String name, String address, String city,
          String state, String vzip, String country )
            throws DuplicateVendorError {
      try {
          #sql {
            INSERT INTO VENDOR_LIST
               VALUES( :vendorno, :name, :address,
                 :city, :state, :vzip, :country )
          };
      }

      /* Ввод неудачен (возможно) из-за того, что номер поставщика уже
         присутствует в таблице.
      */
      catch( SQLException e ) {
          throw new DuplicateVendorError( e.getMessage() );
      }
    }
}
```

Клиент для VendorList

Следующая программа Java использует компонент VendorList:

```
/* Имя программы: VendorListUser.java
**
** Назначение:   С помощью компонента VendorList получить сведения
**               о поставщике 401 и ввести сведения о поставщике 2056.
**
*/

// Импортируем классы компонента.
import vendorlist.*;
import vendorlistServer.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class VendorListUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
    throws Exception {

    // Создадим экземпляр и заполним объект InitialContext.
```

```
        Hashtable env = new Hashtable();
        env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
        env.put( Context.SECURITY_PRINCIPAL, "scott" );
        env.put( Context.SECURITY_CREDENTIALS, "tiger" );
        env.put( Context.SECURITY_AUTHENTICATION,
            ServiceCtx.NON_SSL_LOGIN );
        Context ic = new InitialContext( env );

        // Инициализируем службу транзакций Aurora.
        AuroraTransactionService.initialize(
            ic, "sess_iiop://localhost:2481:ORCL" );

        // Найдем имя объекта, получив ссылку на объект.
        VendorList vendor = ( VendorList )
            ic.lookup( "sess_iiop://localhost:2481:ORCL/test/VendorList" );

        // Вызовем метод getVendorInfo() для получения сведений о поставщике 401.
        VendorInfo vi = vendor.getVendorInfo( 401 );

        // Распечатаем сведения о поставщике.
        System.out.println( "Name = " + vi.name +
                            " Address = " + vi.address +
                            " City = " + vi.city +
                            " State = " + vi.state +
                            " Zip = " + vi.vzip +
                            " Country = " + vi.country );

        // Начнем транзакцию.
        TS.getTS().getCurrent().begin();

        // Введем сведения о поставщике 2056.
        Vendor.insertVendor( 2056, "Shell Shop",
                             "105 S. Ocean Drive",
                             "Miami Beach", "Florida",
                             "77777" , "USA" );

        // Завершим транзакцию.
        TS.getTS().getCurrent().commit( false );
    }
}
```

CORBA-компонент ProjectList

Этот компонент похож на компоненты DepartmentList и VendorList. VendorList и ProjectList будут использоваться компонентами, разрабатываемыми в последнем разделе этой главы.

Ниже рассматриваются:

- Требования пользователей ProjectList
- Общий вид ProjectList
- IDL-файл для ProjectList
- Реализация серверного объекта для ProjectList
- Клиент для ProjectList

Требования пользователей ProjectList

Требования пользователей ProjectList схожи с требованиями пользователей DepartmentList. Пользователи хотят выполнять следующие операции:

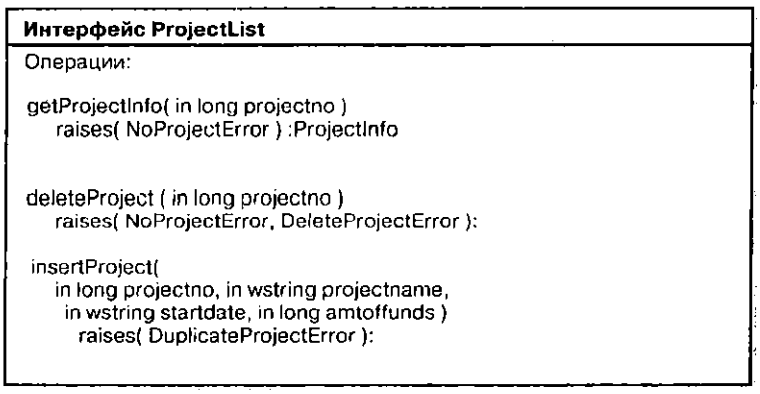
- По номеру проекта узнавать значения столбцов, не являющихся первичным ключом, в строке PROJECT_LIST.
- По номеру проекта удалять строку из PROJECT_LIST.

- Вводить в таблицу PROJECT_LIST строку с указанными значениями столбцов.

Общий вид ProjectList

Общий вид, представленный на рис. 9.4, описывает компонент ProjectList, отвечающий требованиям пользователей.

Рис. 9.4. Общий вид ProjectList



IDL-файл для ProjectList

Следующий IDL-файл отражает спецификацию ProjectList, приведенную на рис. 9.4:

```
/* Имя файла: projectlist.idl */

module projectlist {

   struct ProjectInfo {
     wstring projectname; wstring startdate; long amtoffunds;
   };

   exception NoProjectError { wstring mess; };
   exception DuplicateProjectError { wstring mess; };
   exception DeleteProjectError { wstring mess; };

   interface ProjectList {
     ProjectInfo getProjectInfo( in long projectno )
         raises( NoProjectError );
     void deleteProject ( in long projectno )
         raises( NoProjectError, DeleteProjectError );
     void insertProject(
         in long projectno, in wstring projectname,
           in wstring startdate, in long amtoffunds )
             raises( DuplicateProjectError );
   };
};
```

Реализация серверного объекта для ProjectList

Методы, описанные в ProjectList.idl, реализуются в ProjectListImpl.sqlj.

```
/* Имя программы: ProjectListImpl.sqlj
**
** Назначение:   Реализовать объект ProjectList для
**               работы с таблицей PROJECT_LIST.
**
*/

package projectlistServer;
```

```
import projectlist.*;
import java.sql.*;

public class ProjectListImpl extends _ProjectListImplBase {

  // Считываем сведения о нужном проекте.
  public ProjectInfo getProjectInfo( int projectno )
    throws NoProjectError {
    try {

        // Необходимо инициализировать переменные для размещения в них информации.
        String projectname = null;
        String startdate = null;
        int amtoffunds = 0;

        #sql {
          SELECT projectname, start_date, amt_of_funds
             INTO :projectname, :startdate, :amtoffunds
               FROM PROJECT_LIST
                 WHERE projectno = :projectno
        };
        return new ProjectInfo(projectname, startdate, amtoffunds );
    }

    // Выбор неудачен (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoProjectError( e.getMessage() );
    }
  }

  // Удалим сведения о проекте.
  public void deleteProject ( int projectno )
    throws NoProjectError, DeleteProjectError {
    try {
        int projno = 0 ;
        #sql {
          SELECT projectno
             INTO :projno
               FROM PROJECT_LIST
                 WHERE projectno = :projectno
        };
    }

    // Операция SELECT INTO неудачна (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoProjectError( e.getMessage() );
    }
    try {
        #sql {
          DELETE FROM PROJECT_LIST
             WHERE projectno = :projectno
        };
    }

    // Удаление неудачно.
    catch( SQLException e ) {
        throw new DeleteProjectError( e.getMessage() );
    }
  }

  // Введем сведения о новом проекте.
  public void insertProject (
    int projectno, String projectname,
        String startdate, int amtoffunds )
          throws DuplicateProjectError {
```

```
        try {
            #sql {
              INSERT INTO PROJECT_LIST
                 VALUES(
                    :projectno, :projectname, :startdate, :amtoffunds )
            };
        }

        /* Ввод неудачен (возможно) из-за того, что номер проекта уже
           присутствует в таблице.
        */
        catch( SQLException e ) {
            throw new DuplicateProjectError( e.getMessage() );
        }
      }
}
```

Клиент для ProjectList

Следующая программа Java использует компонент ProjectList:

```
/* Имя программы: ProjectListUser.java
**
** Назначение:  С помощью компонента ProjectList получить сведения
**              о проекте 300, удалить сведения о проекте 300 и
**              ввести сведения о проекте 2056.
**
*/

// Импортируем классы компонента.
import projectlist.*;
import projectlistServer.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class ProjectListUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
    throws Exception {

    // Создадим экземпляр и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Инициализируем службу транзакций Aurora.
```

```
        AuroraTransactionService.initialize(
            ic, "sess_iiop://localhost:2481:ORCL" );

        // Найдем имя объекта, получив ссылку на объект.
        ProjectList project = ( ProjectList )
            ic.lookup( "sess_iiop://localhost:2481:ORCL/test/ProjectList" );

        // Вызовем метод getProjectInfo() для получения сведений о проекте 300.
        ProjectInfo pi = project.getProjectInfo( 300 );

        // Распечатаем сведения о проекте.
        System.out.println( "Project name = " + pi.projectname +
                            " Start date = " + pi.startdate +
                            " Amount of funds = " + pi.amtoffunds );

        // Начнем транзакцию.
        TS.getTS().getCurrent().begin();

        // Удалим сведения о проекте 300.
        project.deleteProject( 300 );

        // Введем сведения о проекте 2056.
        project.insertProject(
            2056, "Neonatal Expansion", "07-Jun-2000", 0 );

        // Завершим транзакцию, состоящую из операций удаления и ввода.
        TS.getTS().getCurrent().commit( false );
    }
}
```

CORBA-компоненты PurchaseList и LineItemList

Эти компоненты тесно взаимосвязаны, поэтому представлены вместе. В частности, они ссылаются друг на друга, так что при описании их в отдельных IDL-файлах оба файла пришлось бы генерировать до компиляции любого из файлов реализации компонентов. Следовательно, эти компоненты должны быть описаны в одном IDL-файле.

Посмотрим, каким образом эти компоненты ссылаются друг на друга. С каждым заказом на покупку связан набор пунктов ассортимента. В записи LINEITEM_LIST присутствует столбец requestno, который является внешним ключом, ссылающимся на запись PURCHASE_LIST пункта. Пользователи LineItemList требуют, чтобы в компоненте содержалась операция, возвращающая, помимо всего прочего, сведения из PURCHASE_LIST для записи PURCHASE_LIST, на которую производится ссылка (аналогичная операция выполняется в компоненте EmployeeList). С другой стороны, пользователи PurchaseList требуют, чтобы операция PurchaseList возвращала сведения из LINEITEM_LIST для всех пунктов, имеющих отношение к указанному заказу на покупку. В общем виде описывается, что эти два пользовательских требования реализуются так, чтобы сделать компоненты взаимоссылающимися. Обратите внимание на то, что вторая операция основана на инверсии "один-ко-многим" взаимоотношения внешнего ключа между пунктами ассортимента и покупками.

Ниже рассматриваются:

- Требования пользователей PurchaseList и LineItemList
- Общий вид PurchaseList и LineItemList
- IDL-файл для PurchaseList и LineItemList
- Реализация серверного объекта для PurchaseList
- Реализация серверного объекта для LineItemList
- Клиент для PurchaseList

Требования пользователей PurchaseList и LineItemList

Сначала посмотрим, какие операции требуют пользователи от PurchaseList:

- По номеру запроса возвращать значения других столбцов строки PURCHASE_LIST.
- По номеру запроса возвращать всю информацию о заказе на покупку, в том числе информацию, не имеющую отношения к первичному ключу, соответствующей строки служащего, информацию, не имеющую отношения к первичному ключу, соответствующей строки поставщика, а также данные строк пунктов ассортимента для заказа на покупку.
- По номеру запроса удалять соответствующую строку из PURCHASE_LIST.
- Вводить новую строку в PURCHASE_LIST с указанными значениями столбцов.

Теперь приведем операции, обязательные для компонента LineItemList:

- По номеру запроса, номеру в ассортименте и номеру проекта возвращать значения столбцов, не являющихся первичным ключом, для пункта ассортимента.
- По номеру запроса, номеру в ассортименте и номеру проекта возвращать всю информацию о пункте ассортимента, в том числе значения столбцов, не являющихся первичным ключом, в соответствующей строке PURCHASE_LIST, и значения столбцов, не являющихся первичным ключом, в соответствующей строке PROJECT_LIST.

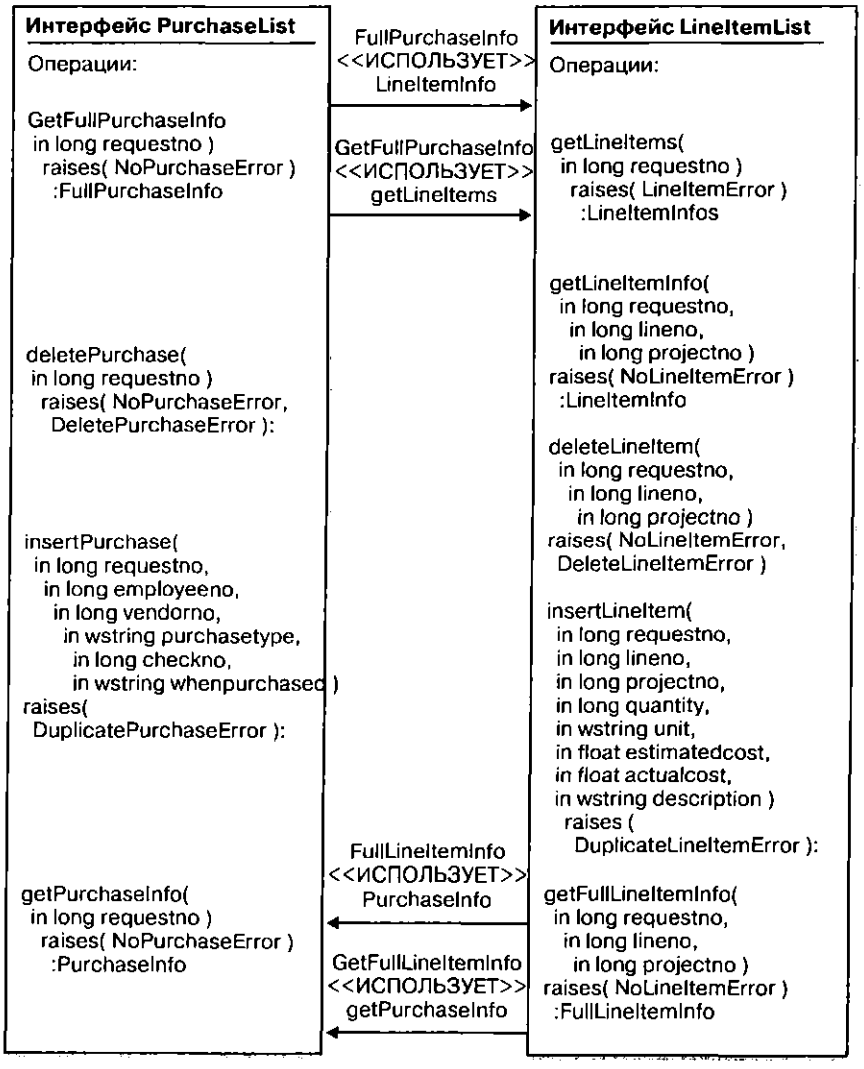


Рис. 9.5. Общий вид взаимоотношения между PurchaseList и LineItemList

- По номеру запроса, номеру в ассортименте и номеру проекта удалять соответствующую строку из LINEITEM_LIST.
- Вводить в LINEITEM_LIST новую строку с указанными значениями столбцов.
- По номеру запроса возвращать значения столбцов, отличных от requestno, соответствующих строк LINEITEM_LIST.

Общий вид PurchaseList и LineItemList

Эти компоненты сложнее, чем рассматривавшиеся выше, и для представления их общего вида используется несколько рисунков. На рис. 9.5 изображено взаимоотношение, существующее между PurchaseList и LineItemList.

На рис. 9.6 изображены взаимоотношения между PurchaseList, EmployeeList и VendorList.

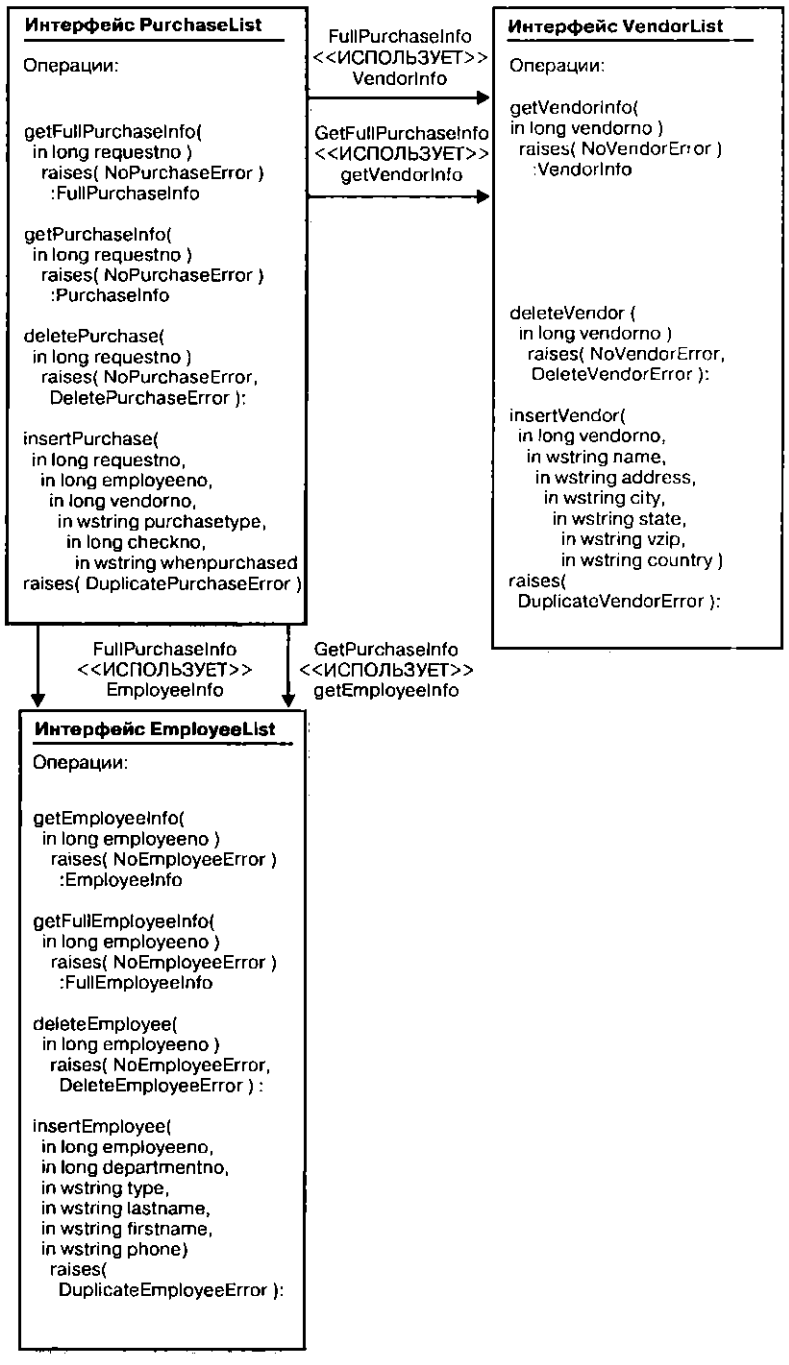


Рис. 9.6. Общий вид взаимоотношений между PurchaseList и VendorList и между PurchaseList и EmployeeList

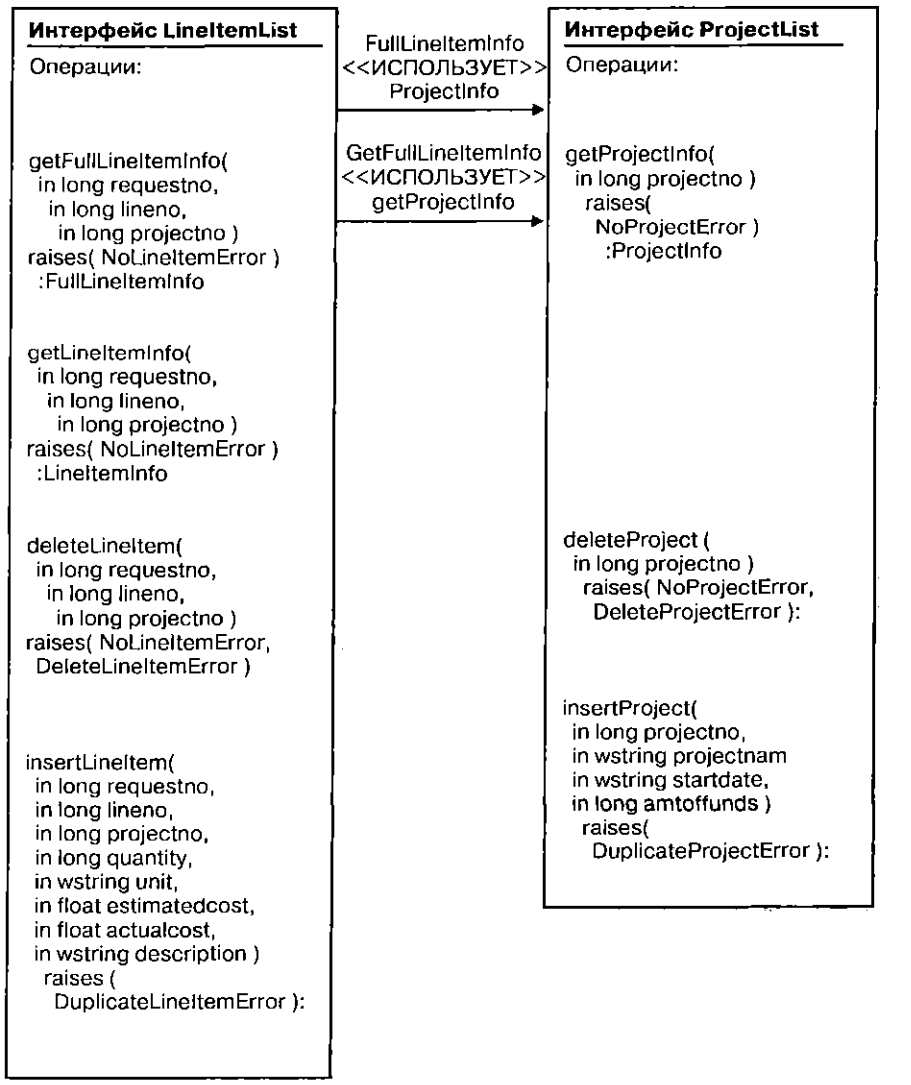


Рис. 9.7. Общий вид взаимоотношения между LineItemList и ProjectList

На рис. 9.7 изображено взаимоотношение между LineItemList и ProjectList.

IDL-файл для PurchaseList и LineItemList

IDL-файл состоит из двух разделов: один для типов PurchaseList, другой для типов LineItemList. Однако typedef, описание PurchaseInfo и описание POLineItemInfo относятся к обоим компонентам.

Информация о пунктах ассортимента, содержащаяся в типе FullPurchaseInfo и возвращаемая методом getLineItems(), описывается как массив объектов POLineItemInfo. В объекте POLineItemInfo содержится вся информация одной строки LINEITEM_LIST:

```
/* Имя файла: purchaseorder.idl */

// Подключим IDL-файлы для типов, на которые производятся ссылки.
#include <employeelist.idl>
#include <vendorlist.idl>
#include <projectlist.idl>

module purchaseorder {

   // Раздел PurchaseList.
```

```
struct PurchaseInfo {
  long employeeno; long vendorno;
      wstring purchasetype; long checkno;
        wstring whenpurchased;
};

/* В POLineItemInfo хранится информация из LineItemInfo, а также
   три значения полей первичного ключа таблицы LINEITEM_LIST.
*/
struct POLineItemInfo {
  long requestno; long lineno; long projectno;
      long quantity; wstring unit; float estimatedcost;
        float actualcost; wstring description;
};

/* typedef для массива объектов с информацией о пунктах
   ассортимента для заказа на покупку.
*/
typedef sequence <POLineItemInfo> LineItemInfos;

/* В FullPurchaseInfo хранятся ссылки на объекты EmployeeInfo и
   VendorInfo для заказа на покупку, а также ссылка на массив
   информации о пунктах ассортимента для объектов заказов на
   покупку плюс информация из PURCHASE_LIST.
*/
struct FullPurchaseInfo {
  employeelist::EmployeeInfo einfo;
      vendorlist::VendorInfo vinfo;
        LineItemInfos linfo; long employeeno; long vendorno;
          wstring purchasetype; long checkno; wstring whenpurchased;
};

exception NoPurchaseError { wstring mess; };
exception DuplicatePurchaseError { wstring mess; };
exception DeletePurchaseError { wstring mess; };

interface PurchaseList {
  PurchaseInfo getPurchaseInfo( in long requestno )
      raises( NoPurchaseError );
  FullPurchaseInfo getFullPurchaseInfo( in long requestno )
      raises( NoPurchaseError );
  void deletePurchase( in long requestno )
      raises( NoPurchaseError, DeletePurchaseError );
  void insertPurchase(
      in long requestno, in long employeeno, in long vendorno,
        in wstring purchasetype, in long checkno,
          in wstring whenpurchased )
              raises( DuplicatePurchaseError );
};

// Раздел LineItemList.
struct LineItemInfo {
  long quantity; wstring unit; float estimatedcost;
      float actualcost; wstring description;
};

/* В FullLineItemInfo хранятся ссылки на объекты PurchaseInfo и
   ProjectInfo для пункта ассортимента, а также информация из
   LINEITEM_LIST.
*/
struct FullLineItemInfo {
  PurchaseInfo pinfo;
      projectlist::ProjectInfo prinfo;
        long quantity; wstring unit; float estimatedcost;
          float actualcost; wstring description;
```

```
    };

    exception NoLineItemError { wstring mess; };
    exception DeleteLineItemError { wstring mess; };
    exception DuplicateLineItemError { wstring mess; };
    exception LineItemError { wstring mess; };

    interface LineItemList {
      LineItemInfo getLineItemInfo(
          in long requestno, in long lineno, in long projectno )
            raises( NoLineItemError );
      FullLineItemInfo getFullLineItemInfo(
          in long requestno, in long lineno, in long projectno )
            raises( NoLineItemError );
      void deleteLineItem(
          in long requestno, in long lineno, in long projectno )
            raises( NoLineItemError, DeleteLineItemError );
      void insertLineItem(
          in long requestno, in long lineno, in long projectno,
            in long quantity, in wstring unit, in float estimatedcost,
              in float actualcost, in wstring description )
                raises ( DuplicateLineItemError );

      /* getLineItems возвращает массив информации о пунктах
         ассортимента для заказа на покупку.
      */
      LineItemInfos getLineItems( in long requestno )
          raises( LineItemError );
    };
};
```

Реализация серверного объекта для PurchaseList

Самым сложным для реализации в компоненте является метод getFullPurchaseInfo(). В этом методе информация из EMPLOYEE_LIST получается в результате обращения к компоненту EmployeeList, информация из VENDOR_LIST — в результате обращения к компоненту VendorList, а информация из LINEITEM_LIST — в результате обращения к компоненту LineItemList. Информация из EMPLOYEE_LIST и VENDOR_LIST сохраняется в объектах типов EmployeeInfo и VendorInfo соответственно. Информация из LINEITEM_LIST сохраняется в массиве (объектов POLineItemInfo). Все ссылки на них возвращаются методом getFullPurchaseInfo() объекта FullPurchaseInfo:

```
/* Имя программы:    PurchaseListImpl.sqlj
**
** Назначение:       Реализовать объект PurchaseList для работы с таблицей
**                   PURCHASE_LIST.
**
*/

package purchaseorderServer;

import java.sql.*;
import purchaseorder.*;

// Импортируем классы для компонентов EmployeeList, VendorList и LineItemList.
import employeelist.*;
import employeelistServer.*;
import vendorlist.*;
import vendorlistServer.*;
import lineitemlist.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
```

```
// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class PurchaseListImpl extends _PurchaseListImplBase {

  // Считываем сведения о нужном заказе на покупку.
  public PurchaseInfo getPurchaseInfo( int requestno )
    throws NoPurchaseError {
    try {

        // Необходимо инициализировать переменные для размещения в них информации.
        int employeeno = 0;
        int vendorno = 0;
        String purchasetype = null;
        int checkno = 0 ;
        String whenpurchased = null;

        #sql {
          SELECT employeeno, vendorno, purchasetype,
                 checkno, whenpurchased
          INTO :employeeno, :vendorno, :purchasetype,
               :checkno, :whenpurchased
             FROM PURCHASE_LIST
               WHERE requestno = :requestno
        };
        return new PurchaseInfo(
          employeeno, vendorno, purchasetype,
            checkno, whenpurchased );
    }

    // Выбор неудачен (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoPurchaseError( e.getMessage() );
    }
  }

  /* Считываем всю информацию о нужном заказе на покупку, в том числе
     информацию о служащем, поставщике и пунктах ассортимента для заказа.
  */
  public FullPurchaseInfo getFullPurchaseInfo( int requestno )
    throws NoPurchaseError {
    try {

        // Получим сведения о покупке.
        pi = getPurchaseInfo( requestno );

        // Создадим экземпляр и заполним объект InitialContext.
        Hashtable env = new Hashtable();
        env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
        Context ic = new InitialContext( env );

        // Получим сведения о служащем для заказа на покупку.

        // Найдем имя объекта EmployeeList, получив ссылку на объект.
        EmployeeList employee = ( EmployeeList )
          ic.lookup( "sess_iiop://thisServer/test/EmployeeList" );

        // Вызовем метод getEmployeeInfo(), получив сведения о служащем.
        EmployeeInfo ei = employee.getEmployeeInfo( pi.employeeno );
```

```
        // Получим сведения о поставщике для заказа на покупку.

        // Найдем имя объекта VendorList, получив ссылку на объект.
        VendorList vendor = ( VendorList )
          ic.lookup( "sess_iiop://thisServer/test/VendorList" );

        // Вызовем метод getVendorInfo(), получив сведения о поставщике.
        VendorInfo vi = vendor.getVendorInfo( pi.vendorno );

        // Получим сведения о пунктах ассортимента для заказа на покупку.

        // Найдем имя объекта LineItemList, получив ссылку на объект.
        LineItemList lineitem = ( LineItemList )
          ic.lookup( "sess_iiop://thisServer/test/LineItemList" );

        /* Вызовем метод getLineItems() для получения сведений о пунктах
           ассортимента.
        */
        POLineItemInfo[] lis = lineitem.getLineItems( requestno );

        return new FullPurchaseInfo(
          ei, vi, lis, pi.employeeno, pi.vendorno,
            pi.purchasetype, pi.checkno, pi.whenpurchased );
      }

      // Неудача.
      catch( Exception e ) {
          throw new NoPurchaseError( e.getMessage() );
      }
    }

    // Удалим запись о покупке.
    public void deletePurchase( int requestno )
      throws NoPurchaseError, DeletePurchaseError {
      try {
          int reqno = 0;
          #sql {
            SELECT requestno
               INTO :reqno
                 FROM PURCHASE_LIST
                   WHERE requestno = :requestno
          };
      }

      // Операция SELECT INTO неудачна (возможно) из-за отсутствия строки.
      catch( SQLException e ) {
          throw new NoPurchaseError( e.getMessage() );
      }
      try {
          #sql {
            DELETE FROM PURCHASE_LIST
               WHERE requestno = :requestno
          };
      }

      // Удаление неудачно.
      catch( SQLException e ) {
          throw new DeletePurchaseError( e.getMessage() );
      }
    }

    // Введем запись о новой покупке.
    public void insertPurchase(
      int requestno, int employeeno, int vendorno,
          String purchasetype, int checkno, String whenpurchased )
```

```
        throws DuplicatePurchaseError {
    try {
        #sql {
          INSERT INTO PURCHASE_LIST
            VALUES( :requestno, :employeeno, :vendorno,
              :purchasetype, :checkno, :whenpurchased )
        };
    }

    /* Ввод неудачен (возможно) из-за того, что номер запроса уже
       присутствует в таблице.
    */
    catch( SQLException e ) {
        throw new DuplicatePurchaseError( e.getMessage() );
    }
  }
}
```

Реализация серверного объекта для LineItemList

Двумя самыми сложными методами для реализации в этом классе являются getFullLineInfo() и getLineItems(). Структура getFullLineInfo() аналогична структуре метода getFullEmployeeInfo() (см. выше). Поэтому сосредоточим свое внимание на методе getLineItems().

Сначала сведения о пунктах ассортимента для нужного заказа переносятся в итератор. Затем строка за строкой эти сведения переносятся в вектор. Вектор (Vector) — это динамический массив. В данном случае используется именно вектор, а не массив с фиксированной границей, потому что заранее определять число строк, загруженных в итератор, с помощью оператора SELECT INTO нерационально. Размер вектора увеличивается с вводом в него элементов.

После загрузки всех строк из итератора в вектор создается массив нужного размера, и информация из вектора копируется в массив. Ссылка именно на этот массив и возвращается методом:

```
/* Имя программы:    LineItemListImpl.sqlj
**
** Назначение:       Реализовать объект LineItemList для работы с таблицей
**                   LINEITEM_LIST.
**
*/

package purchaseorderServer;

import lineitemlist.*;

// Импортируем классы для компонентов EmployeeList, PurchaseList, и ProjectList.
import employeelist.*;
import purchaseorder.*;
import projectlist.*;
import projectlistServer.*;

import java.sql.*;
import java.util.Vector;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;
```

```
// В LIter будут храниться записи о пунктах для заказа на покупку.
#sql iterator LIter (
    int lineno, int projectno, int quantity, String unit,
        float estimatedcost, float actualcost, String description );

public class LineItemListImpl extends _LineItemListImplBase {

  // Считываем сведения о нужном пункте ассортимента.
  public LineItemInfo getLineItemInfo(
    int requestno, int lineno, int projectno )
        throws NoLineItemError {
    try {

        // Необходимо инициализировать переменные для размещения в них информации.
        int quantity = 0;
        String unit = null;
        float estimatedcost = (float)0.0;
        float actualcost = (float)0.0;
        String description = null;

        #sql {
          SELECT quantity, unit, estimatedcost,
             actualcost, description
          INTO :quantity, :unit, :estimatedcost,
               :actualcost, description
             FROM LINEITEM_LIST
               WHERE requestno = :requestno
                 AND lineno = :lineno
                   AND projectno = :projectno
        };
        return new LineItemInfo(
          quantity, unit, estimatedcost, actualcost, description );
    }

    // Выбор неудачен (возможно) из-за отсутствия строки.
    catch( SQLException e ) {
        throw new NoLineItemError( e.getMessage() );
    }
  }

  /* Считываем всю информацию о нужном пункте ассортимента, в том
     числе информацию о покупке и проекте для пункта.
  */
  public FullLineItemInfo getFullLineItemInfo(
    int requestno, int lineno, int projectno )
        throws NoLineItemError {
    try {

        // Получим сведения о пункте ассортимента.
        LineItemInfo li = getLineItemInfo(
          requestno, lineno, projectno );

        // Получим сведения о покупке для пункта ассортимента.

        // Создадим экземпляр и заполним объект InitialContext.
        Hashtable env = new Hashtable();
        env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
        Context ic = new InitialContext( env );

        // Найдем имя объекта покупки, получив ссылку на объект.
        PurchaseList purchase = ( PurchaseList )
          ic.lookup( "sess_iiop://thisServer/test/PurchaseList" );

        // Вызовем метод getPurchaseInfo() для получения сведений о покупке.
        PurchaseInfo pi = purchase.getPurchaseInfo( requestno );
```

```
        // Получим сведения о проекте для пункта ассортимента.

        // Найдем имя объекта проекта, получив ссылку на объект.
        ProjectList project = ( ProjectList )
          ic.lookup( "sess_iiop://thisServer/test/ProjectList" );

        // Вызовем метод getProjectInfo() для получения сведений о проекте.
        ProjectInfo pri = project.getProjectInfo( projectno );

        return(
          new FullLineItemInfo(
             pi, pri, li.quantity, li.unit, li.estimatedcost,
               li.actualcost, li.description )
        );
      }

      // Неудача.
      catch( Exception e ) {
        throw new NoLineItemError( e.getMessage() );
      }
    }

    // Удалим сведения о пункте ассортимента.
    public void deleteLineItem(
      int requestno, int lineno, int projectno )
        throws NoLineItemError, DeleteLineItemError {
      try {

        // Необходимо инициализировать переменные для размещения в них информации.
        int quantity = 0;
        String unit = null;
        float estimatedcost = (float)0.0;
        float actualcost = (float)0.0;
        String description = null;

        #sql {
          SELECT quantity, unit, estimatedcost,
                 actualcost, description
             INTO :quantity, :unit, :estimatedcost,
                  :actualcost, :description
               FROM LINEITEM_LIST
                 WHERE requestno = :requestno
                   AND lineno = :lineno
                     AND projectno = :projectno
        };
      }

      // Операция SELECT INTO неудачна (возможно) из-за отсутствия строки.
      catch( SQLException e ) {
        throw new NoLineItemError( e.getMessage() );
      }
      try {
        #sql {
          DELETE FROM LINEITEM_LIST
             WHERE requestno = :requestno
               AND lineno = :lineno
                 AND projectno = :projectno
        };
      }

      // Удаление неудачно.
      catch( SQLException e ) {
        throw new DeleteLineItemError( e.getMessage() );
      }
    }
```

```
// Введем новый пункт ассортимента.
public void insertLineItem(
  int requestno, int lineno, int projectno,
      int quantity, String unit, float estimatedcost,
        float actualcost, String description )
          throws DuplicateLineItemError {
  try {
      #sql {
        INSERT INTO LINEITEM_LIST
          VALUES( :requestno, :lineno, :projectno,
            :quantity, :unit, :estimatedcost,
                :actualcost, description )
      };
  }

  /* Ввод неудачен (возможно) из-за того, что пункт ассортимента
     уже присутствует в таблице.
  */
  catch( SQLException e ) {
      throw new DuplicateLineItemError( e.getMessage() );
  }
}

// Получим сведения о пунктах ассортимента для нужного заказа на покупку.
public POLineItemInfo[] getLineItems( int requestno )
  throws LineItemError {
  try {

      /* Поместим считанные сведения о пунктах ассортимента в
         вектор, так как вычислять количество пунктов для заказа на
         покупку слишком долго.
      */
      Vector polineitemVector = new Vector();
      LIter anLIter = null;
      #sql anLIter =
        { SELECT lineno, projectno, quantity, unit,
          estimatedcost, actualcost, description
            FROM LINEITEM_LIST
              WHERE requestno = :requestno
        };

      // Добавим считанную запись в вектор.
      while( anLIter.next() ) {
        polineitemVector.addElement(
          new POLineItemInfo(
            requestno,
            anLIter.lineno(),
            anLIter.projectno(),
            anLIter.quantity(),
            anLIter.unit(),
            anLIter.estimatedcost(),
            anLIter.actualcost(),
            anLIter.description() ) );
      }
      anLIter.close();

      // Перенесем записи из вектора в массив.
      POLineItemInfo[] returnarray =
        new POLineItemInfo[ polineitemVector.size() ]
      polineitemVector.copyInto( returnarray );

      // Возвратим ссылку на массив.
      return returnarray;
  }
```

```
        // Неудача.
        catch( SQLException e ) {
            throw new LineItemError( e.getMessage() );
        }
      }
}
```

Клиент для PurchaseList

Для иллюстрации достаточно создать клиента, применяющего только компонент PurchaseList, так как сам PurchaseList пользуется компонентом LineItemList.

Клиент обращается к компоненту PurchaseList и вызывает его метод getFullPurchaseInfo(). Этот метод возвращает объект FullPurchaseInfo, который содержит значения столбцов, не являющихся первичным ключом, в нужной строке PURCHASE_LIST, а также ссылки на соответствующие строки EMPLOYEE_LIST и VENDOR_LIST плюс массив ссылок POLineItemInfo со сведениями о пунктах ассортимента для заказа на покупку. Затем клиент распечатывает всю эту информацию. Для печати массива используется цикл for.

Наконец, с помощью компонента PurchaseList клиент удаляет строку из таблицы PURCHASE_LIST и вводит в нее новую строку:

```
/* Имя программы:    PurchaseListUser.java
**
** Назначение:       С помощью компонента PurchaseList получить сведения
**                   о заказе на покупку 500, удалить запись о покупке
**                   500 и ввести запись о покупке 2056.
**
*/

import java.util.Vector;

// Импортируем классы компонента.
import purchaseorder.*;
import purchaseorderServer.*;

// Импортируем классы для обработки транзакций.
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;
import oracle.aurora.jts.client.AuroraTransactionService;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class PurchaseListUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args )
    throws Exception {

    // Создадим экземпляр и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
```

```
    ServiceCtx.NON_SSL_LOGIN );
Context ic = new InitialContext( env );

// Инициализируем службу транзакций Aurora.
AuroraTransactionService.initialize(
    ic, "sess_iiop://localhost:2481:ORCL" );
    Vector polineitemVector = new Vector();

// Найдем имя объекта, получив ссылку на объект.
PurchaseList purchase = ( PurchaseList )
    ic.lookup( "sess_iiop://localhost:2481:ORCL/test/PurchaseList" );

/* Вызовем метод getFullPurchaseInfo() для получения сведений о
   покупке 500, в том числе сведений о служащем, поставщике и
   пунктах ассортимента.
*/
FullPurchaseInfo fi = purchase.getFullPurchaseInfo( 500 );

// Распечатаем сведения о заказе на покупку.
System.out.println(
    "Employee number = " + fi.employeeno +
    " Vendor number = " + fi.vendorno +
    " Purchase type = " + fi.purchasetype +
    " Check number = " + fi.checkno +
    " Purchase date = " + fi.whenpurchased );

// Распечатаем сведения о служащем для заказа на покупку.
System.out.println(
    "Last name = " + fi.einfo.lastname +
    " First name = " + fi.einfo.firstname +
    " Type = " + fi.einfo.type +
    " Department number = " + fi.einfo.departmentno +
    " Phone = " + fi.einfo.phone );

// Распечатаем сведения о поставщике для заказа на покупку.
System.out.println(
    "Name = " + fi.vinfo.name +
    " Address = " + fi.vinfo.address +
    " City = " + fi.vinfo.city +
    " Zip = " + fi.vinfo.vzip +
    " Country = "+ fi.vinfo.country );

// Распечатаем сведения о пунктах ассортимента для заказа на покупку.
for( int i = 0; i < fi.linfo.length; i++ ) {
    System.out.println(
      "Line number = " + fi.linfo[i].lineno +
      " Project number = " + fi.linfo[i].projectno +
      " Quantity = " + fi.linfo[i].quantity );

    System.out.println(
      "Unit = " + fi.linfo[i].unit +
      " Estimated cost = " + fi.linfo[i].estimatedcost +
      " Actual cost = " + fi.linfo[i].actualcost +
      " Description = " + fi.linfo[i].description );
}

// Начнем транзакцию.
TS.getTS().getCurrent().begin();

// Удалим запись о покупке 500.
purchase.deletePurchase( 500 );

// Введем запись о покупке 2056.
purchase.insertPurchase(
    2056, 102, 402, null, 0, "07-JUN-2000" );
```

```
            // Завершим транзакцию, состоящую из операций удаления и ввода.
            TS.getTS().getCurrent().commit( false );
        }
    }
```

Итоги

На этом заканчивается часть III нашей книги, в которой говорилось о CORBA.

В главе 6 были представлены основы реализации и использования серверных объектов CORBA в Java. В главе 7 рассматривались различные способы вызова серверных объектов CORBA в сеансах связи с базой данных. В главе 8 говорилось о проблемах вызова серверных объектов в глобальных транзакциях. В главе 9 были приведены примеры проектирования и реализации компонентов CORBA, связанных друг с другом посредством взаимоотношений внешнего ключа и "один-ко-многим".

В части IV рассказывается о реализации страниц JSP (JavaServer Pages), обращающихся к компонентам EJB и CORBA.

ЧАСТЬ IV

Построение web-приложений с помощью страниц JSP

ГЛАВА 10

Введение в программирование с использованием страниц JSP

Web-программирование с использованием сервлетов и страниц JSP рассматривалось в главе 2. Там говорилось о различиях между двумя моделями и об общей архитектуре web-приложений. Напомним, что в JSP поддерживается написание на HTML-страницах сценариев, основанных на Java, что позволяет без труда создавать web-страницы с динамическим содержимым. Например, при помощи JSP можно обратиться к базе данных с запросом и вывести результаты в виде HTML-таблицы. Основой страниц JSP являются сервлеты – программы, использующие только Java и исполняемые на web-сервере для создания динамических страниц. Главная цель спецификации JSP – повысить уровень абстракции по сравнению с сервлетами и сделать модель программирования более удобной. По своей структуре страницы JSP предназначены для компонентного программирования, для чего служат тэги сценариев и JavaBean. Эти страницы обеспечивают четкое отделение статического содержимого web-страницы от программной логики, генерирующей динамическое содержимое. Страницы JSP преобразуются в программы Java и используют ту же структуру исполнения, что и сервлеты, а производительность повышается такими способами, как автоматическая повторная трансляция JSP-файлов, что способствует быстрой разработке web-приложений.

В этой части книги JSP-программирование рассматривается более глубоко. В главе 10 даются основы разработки программ с помощью JSP-страниц. В главе 11 говорится о компонентном программировании с использованием JavaBean и JSP. Основное внимание уделяется приложениям баз данных, где применяются компоненты JavaBean, написанные на SQLJ и JDBC. В главе 12 описываются компоненты таких типов, как зерна Enterprise JavaBean (EJB) и CORBA-объекты Java, внедряемые в Oracle8*i*, а также библиотеки тэгов JSP. Будут рассмотрены различные способы объединения этих компонентов на страницах JSP и эффективного использования возможностей Oracle8*i* и Java для построения модульных и мощных web-приложений баз данных.

Цель этой главы – представить некоторые ключевые концепции JSP-программирования и дать общий обзор методов JSP-программирования. На примере нескольких программ описываются базовый синтаксис JSP и процесс разработки приложений. Кроме того, рассказывается об использовании JSP- страниц вместе с другими JSP-страницами и сервлетами. В частности, говорится о том, как:

- Написать JSP-страницу `Hello` с помощью элементов сценариев, неявных объектов и HTTP-формы
- Преобразовывать и исполнять JSP
- Использовать директивы для импорта пакетов Java и для подключения других страниц во время преобразования JSP
- Использовать стандартные функциональные тэги `<jsp:include>` и `<jsp:forward>` для вызова других страниц с JSP во время ее выполнения
- Вызывать сервлет с JSP и наоборот, чтобы комбинировать в web-приложении программные конструкции двух типов
- Использовать страницы ошибок для обработки ошибок этапа выполнения на JSP
- Использовать на JSP программные конструкции SQLJ для исполнения статического SQL-запроса, обращенного к базе данных Oracle
- Использовать транслятор командной строки в JSP-реализации Oracle

Большинство примеров иллюстрирует стандартные свойства спецификаций JSP 1.0 и 1.1, за исключением JSP-транслятора командной строки и использования SQLJ в JSP-файлах. Эти свойства весьма полезны и расширяют JSP-реализацию Oracle.

Помимо упомянутых выше, существует ряд других стандартных тэгов, доступных в JSP. О тэгах для компонентного программирования с использованием JavaBean, например <jsp:useBean> и <jsp:setProperty>, говорится в следующей главе. Полный набор JSP-тэгов приведен в приложении А. Кроме набора стандартных тэгов, в JSP 1.1 описывается схема расширения для *специальных* (custom) тэгов. Это свойство является главным отличием спецификации JSP 1.1 от JSP 1.0 и мощным механизмом расширения JSP посредством *библиотек тэгов* (tag libraries) третьих фирм (см. раздел "Использование библиотек тэгов JSP" главы 12).

Теоретически спецификация JSP допускает использование языков написания сценариев, отличных от Java, однако Java дает основу для понимания программирования с помощью JSP и сервлетов. В этой книге в качестве языка сценариев применяется или Java, или SQLJ (расширение Java встроенными SQL-операторами). Кроме того, спецификации JSP и сервлетов разрешают использование протоколов, отличных от HTTP. В книге рассматривается HTTP, так как на нем основано большинство реализаций сервлетов и JSP.

Основы JSP-программирования

Для построения JSP-приложений применяются программные элементы двух видов:

- *Элементы сценариев* (scripting elements) JSP, используемые для встраивания программных конструкций Java в JSP-файл
- *Неявные объекты* (implicit objects) JSP, автоматически доступные в рамках JSP для обращения к базовым объектам Java, например HTML-запрос и HTML-ответ

Применим эти два средства JSP для написания простой JSP-программы, которая выводит приветствие "Hello", текущие дату и время. Данный пример показывает, как можно быстро и просто разработать базовое JSP-приложение, используя для этого встроенные средства.

Элементы сценариев

Для JSP доступны элементы сценариев трех типов:

- Объявления
- Скриптлеты
- Выражения

Для каждого типа элементов сценариев в спецификации JSP предусмотрены два вида синтаксического обозначения. Для представления начала и окончания элемента в одном случае используются ограничители типа <% и %>, а в другом – синтаксис, совместимый с XML. Причина создания эквивалентного XML-обозначения состоит в том, чтобы разрешить написание JSP-страниц при помощи инструментов создания документов XML. Мы познакомимся с синтаксисом обоих видов.

Ниже следует краткое описание каждого типа элементов сценариев JSP.

Объявления

Это объявления (declarations) переменных и методов Java, которые будут использоваться в программных конструкциях Java JSP-файла. Они ограничиваются тэгами <%! и %> и могут применяться на всей странице. Например, в следующем блоке объявляются две переменные Java (`firstName` и `lastName`) со строковыми значениями:

```
<%! String firstName;
    String lastName;
%>
```

В обозначении, совместимом с XML, эти объявления принимают следующий вид:

```
<jsp:declaration>
    String firstName;
    String lastName;
</jsp:declaration>
```

Объявленные подобным образом переменные инициализируются при загрузке и инициализации JSP-страницы (см. главу 2). Сами объявления переменных и методов не приводят на странице к какому-либо результату, но они используются другими элементами JSP (выражениями и скриптлетами) для создания динамического содержимого.

Выражения

Это выражения (expressions) Java, которые динамически вычисляются на этапе выполнения и распечатываются на выходной странице в виде символьных строк. Они окружаются тэгами <%= и %>, которые чередуются со статическим содержимым. Например, следующим оператором Java-выражение `firstName` встраивается в HTML-текст, и в результате выводится приветствие:

```
Hello   <%= firstName %> !   Have a nice day!
```

В альтернативном XML-обозначении этот фрагмент можно записать так:

```
Hello   <jsp:expression> firstName </jsp:expression> ! Have a nice day
```

При исполнении этого программного фрагмента значение переменной вычисляется, и строковое значение вводится в нужное место генерируемого результата. Обратите внимание: встраиваемое выражение не заканчивается точкой с запятой; дело в том, что в генерируемой программной конструкции оно используется в качестве аргумента при вызове метода `print(...)`.

Скриптлеты

Скриптлеты (scriptlets) JSP – это программные фрагменты Java, исполняемые при вызове JSP. Они ограничиваются тэгами <% и %>, чередуемыми со статическим содержимым. Например, следующая программная конструкция проверяет, больше ли нуля значение переменной itemCount, и выводит на экран соответствующее сообщение:

```
<%  if  (itemCount > 0) { %>
        <B> You selected <%= itemCount %> items. </B>
<%  }   else { %>
        <B> You have not selected any items. </B>
<%      } %>
```

В этом примере использованы три разных скриптлета: оператор `if`, оператор `else` и завершающая фигурная скобка для оператора `if`. Кроме того, для вывода значения `itemCount` применяется встроенное JSP-выражение. Скриптлеты и выражения чередуются с HTML-текстом в порядке их исполнения. Заметим, что вовсе не обязательно, чтобы скриптлетами завершались блоки текста Java,– первый скриптлет начинается с оператора `if` и заканчивается открывающей фигурной скобкой. За этим скриптлетом следует статический HTML-текст с встроенным JSP-выражением, вслед за которым идет оператор `else`, продолжающий оператор `if` предыдущего скриптлета. При преобразовании JSP-страницы отдельные программные фрагменты объединяются, создавая в итоге программную конструкцию Java. Этот программный блок должен быть синтаксически правилен, и только тогда он будет успешно скомпилирован. Следовательно, нужно закончить скриптлет точкой с запятой или опустить точку с запятой – это зависит от синтаксических правил Java.

В альтернативном XML-обозначении приведенный выше скриптлет можно записать так:

```
<jsp:scriptlet> if  (itemCount > 0)  { </jsp:scriptlet>
  <B>You selected <jsp:expression>itemCount</jsp:expression> items.</B>
<jsp:scriptlet> }   else { </jsp:scriptlet>
    <B:>You have not selected any items.</B>
<jsp:scriptlet>     } </jsp:scriptlet>
```

Обратите внимание на то, что для JSP-тэгов учитывается регистр символов. Полный перечень элементов сценариев JSP приводится в приложении D. В JSP-программе Hello.jsp используются элементы сценариев всех трех видов. Но прежде чем создать ее, выясним, какие объекты применяются в JSP неявно.

Неявные объекты JSP

Для удобства программирования существует ряд объектов Java, описываемых автоматически и используемых в скриптлетах и выражениях JSP без явного объявления программных переменных. Среди них:

- `request`. Представляет HTTP-запрос для клиентского вызова. Это экземпляр типа `javax.servlet.http.HttpServletRequest`. Его методы `getParameter()`, `getParameterNames()` и `getParameterValues()` позволяют считывать параметры вызова из HTTP-запроса.
- `response`. Представляет HTTP-ответ от сервера. Это экземпляр типа `javax.servlet.http.HttpServletResponse`. Его методы, например `setContentType()`, позволяют устанавливать кодировку символов выходных данных.
- `session`. Представляет HTTP-сеанс (если он есть), связанный с клиентским запросом. Доступен, если JSP является частью HTTP-сеанса (необязательное свойство web-страницы). Это экземпляр типа `javax.servlet.http.HttpSession`, и его методы, например `setMaxInactiveInterval()`, позволяют устанавливать таймаут сеанса.
- `application`. Представляет web-приложение (контекст сервлетов), в котором работает JSP. Это экземпляр типа `javax.servlet.ServletContext`, и его методы, например `log()`, позволяют записывать ошибки и другие сообщения в файл журнала.
- `out`. Представляет выходной поток для JSP. Используется для вывода статических и динамических данных. Это экземпляр типа `javax.servlet.jsp.JspWriter`, и его методы, например `print()`, `println()`, `clear()`, `flush()` и `close()`, позволяют записывать данные.

В спецификации JSP описаны и другие неявные объекты, например `exception`, `pageContext`, `config` и `page`. Об использовании объекта `exception` рассказывается ниже. Полный перечень неявных объектов JSP приводится в приложении D. Описания соответствующих типов в Servlet 2.2 API даются в приложении A.

В следующем примере JSP-скриптлета используется неявный объект `request`:

```
<% String user = request.getParameter("username"); %>
```

Этот скриптлет вызывает метод `getParameter()` объекта `request`, заполняя строковую Java-переменную `user`. Аргументом метода `getParameter()` является строка символов, представляющая параметр, в данном случае `username`. В результате вызова этот метод возвращает строковое значение, если параметр присутствует в списке параметров запросов JSP, и `null` в противном случае. Заметим, что в операции поиска имени-значения учитывается регистр символов.

Написание `Hello.jsp`

Теперь создадим завершенную JSP-страницу. Назначение этой программы — вывести приветствие клиенту, а также текущие дату и время. Ее текст (с использованием не-XML обозначения `<%...%>`) приведен ниже и снабжен комментариями. Обратите внимание: комментарии можно писать с использованием

синтаксиса HTML, соглашений Java или JSP-тэгов. Комментарии синтаксиса HTML или XML ограничиваются тэгами <!-- и -- >, и в них можно встраивать JSP-выражения. Эти комментарии являются частью генерируемой выходной страницы. В Java комментарий может соответствовать принятому соглашению: /*...*/ или // (как удобнее); такие комментарии на генерируемой странице не появляются. Кроме того, можно пользоваться тэгами комментариев JSP <% --...-- %>. Подобные комментарии не интерпретируются транслятором JSP.

```
<%-- Имя программы:  Hello.jsp
  -- Назначение:         Распечатать приветствие и текущее время.
  --%>
<HTML>                                  <%-- (см. пояснение 1) --%>
  <HEAD> <TITLE> The Hello JSP </TITLE>
  </HEAD>
  <!-- Это JSP-тело выводит пользователю сообщение, содержащее приветствие
    -- и текущее время --!>
  <BODY BGCOLOR="white">                <%-- (см. пояснение 2) --%>
    <%! String defaultName = "there";       /* (см. пояснение 3) */ %>
    <%  String name = request.getParameter("username");
                                          /* (см. пояснение 4) */ %>
    <H3> Hello <%= (name == null)? defaultName : name %> ! </H3>
                                        <%-- (см. пояснение 5) --%>
    <P><B> The current time is <%= new java.util.Date() %>
                                        <%-- (см. пояснение 6) --%>
    </B></P>
    <B>May I have your name?</B>
    <FORM METHOD=GET>                   <%-- (см. пояснение 7) --%>
        <INPUT TYPE="text"
               NAME="username"
               SIZE=15>                 <%-- (см. пояснение 8) --%>
    <INPUT TYPE="submit"
               VALUE="Submit name"><%-- (см. пояснение 9) --%>
    </FORM>
    <P><B> Have a good day! :-) </B></P>
  </BODY>
</HTML>
```

Пояснения к программе Hello.jsp**:**

1. Тэг <HTML> определяет начало HTML-текста. Следующие три оператора – это обычные HTML-конструкции, описывающие начало страницы и ее название с помощью тэгов <HEAD> и <TITLE>.
2. Тэг <BODY> определяет начало сегмента тела HTML. Здесь с помощью атрибута BGCOLOR фоновым назначается белый цвет.
3. Это JSP-объявление строковой переменной с именем defaultName, которой присваивается постоянное значение "there". В этом примере defaultName можно было бы объявить в скриптлете, так как она используется только в главном теле страницы (других явно объявляемых методов Java здесь нет).
4. Этот оператор устанавливает переменную name в значение username параметра запроса, вызывая метод getParameter() неявного объекта request.
5. Этот оператор с помощью встроенного Java-выражения выводит сообщение "Hello" с именем пользователя, если оно не является null-значением, или "Hello there", если имя пользователя не указано (в этом случае метод request.getParameter("username") возвращает null).
6. Этот метод получает текущее время, используя встроенное Java-выражение new java.util.Date(). Возвращаемый этим конструктором результат автоматически преобразуется в строку символов Java с датой и временем и помещается на выходную страницу.

7. Этот оператор отмечает начало сегмента формы HTML. С помощью форм в HTML-программировании часто получают данные из браузеров клиентов (см. главу 2). Посмотрим в общих чертах, как действует HTML-форма (за дополнительной информацией обращайтесь к специальным книгам по HTML). Сегмент формы ограничивается начальным и конечным тэгами <FORM> и </FORM>. В форме можно описать одно или несколько полей ввода и одну или несколько кнопок. С каждым полем ввода связывается имя и, при желании, поля TYPE и SIZE, определяющие тип и размер входного окна. По щелчку мыши на функциональной кнопке серверу направляется новый HTTP-запрос.

   Направление и способ посылки нового запроса определяются:

   - Атрибутом ACTION тэга <FORM>, который задается как URL для программы, вызываемой на сервере. Если этого атрибута нет, как в программе Hello.jsp, то по умолчанию запрос направляется на ту же страницу, т.е. на Hello.jsp.
   - Полем METHOD, которое является необязательным атрибутом тэга <FORM>, указывающим HTTP-метод для передачи данных web-серверу. Его значением может быть или POST, или GET. Если этот атрибут опущен, значением по умолчанию становится GET. Метод GET добавляет вводимые в форму данные (называемые *строкой запроса* (query string)) к функциональному URL в виде пар *ключ=значение*. Если полей формы несколько, они разделяются символами '&', каждый пробел преобразуется в символ '+', а каждый специальный символ представляется своим шестнадцатеричным эквивалентом. В противоположность этому, метод POST посылает данные формы как часть тела HTTP-запроса, что не видно в URL запроса.

   В тэге <FORM> примера Hello.jsp используется метод GET для демонстрации того, каким образом данные формы, введенные на экране, посылаются как часть URL запроса. Этот URL можно пометить закладкой и вызывать напрямую. Однако метод GET не подходит для передачи больших объемов данных или конфиденциальной информации, например номера службы социальной защиты (в США); для этого следует применять метод POST. Для обоих методов данные формы можно считывать в HTTP-объекте request методом getParameter(). В соответствии с этим в программе Hello.jsp для считывания имени пользователя, введенного на экране, вызывается метод request.getParameter("username") (см. пояснение 4).

8. Этот тэг <INPUT> определяет текстовое поле ввода с именем username и длиной 15 символов.

9. Этот тэг <INPUT> определяет функциональную кнопку с именем "Submit name". Если ввести имя (скажем, Pam) в текстовом окне и щелкнуть мышью на кнопке на экране, web-серверу будет направлен новый запрос с указанными данными. Поскольку методом формы является GET (см. пояснение 7), в новом URL содержится строка запроса username=Pam, которая добавляется к исходному URL.

Выполнение Hello.jsp

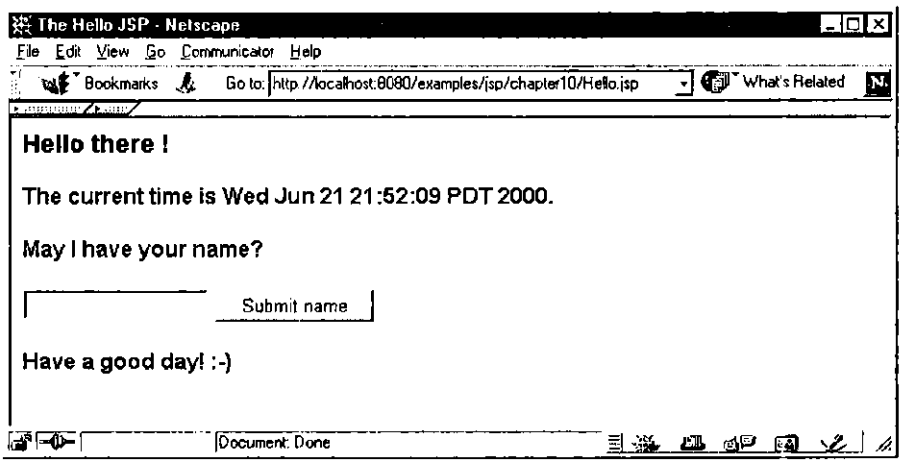
Сначала установим программу Hello.jsp на web-сервере. Шаги, выполняемые во время инсталляции, в большой степени зависят от применяемого web-сервера. Операции установки для ряда популярных web-серверов приведены в приложении E. Предположим, что IDL-файл установлен как examples/jsp/chapter10/Hello.jsp относительно корня приложений web-сервера по умолчанию. Теперь можно выполнить эту JSP, указав браузеру соответствующий URL:

- http://localhost:8080/examples/jsp/chapter10/Hello.jsp

Такой запрос приводит к тому, что двигатель JSP автоматически запускает транслятор JSP, если это первый вызов страницы. Генерируемый код

компилируется компилятором Java, а полученный в результате байтовый код Java исполняется web-сервером. Если файл `Hello.jsp` не изменяется, в последующих запросах, обращенных к той же странице, не происходит повторной трансляции JSP, а исполняется ранее скомпилированный байтовый код Java. Типичный результат для этой страницы, выдаваемый браузером, показан на рис. 10.1

Рис. 10.1. Выходные данные `Hello.jsp`



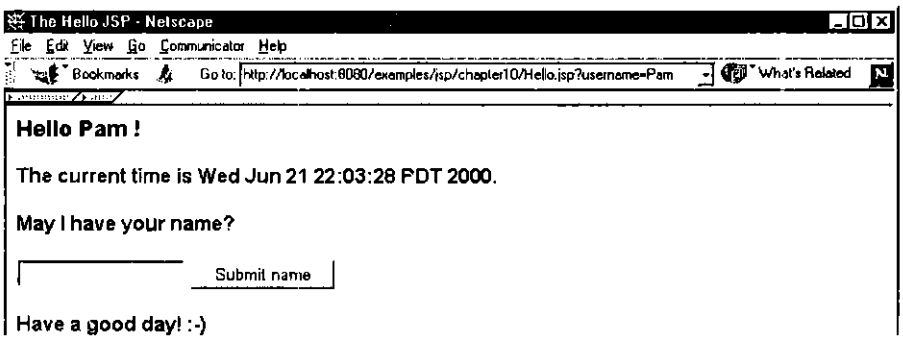
В данном случае URL запроса не содержит параметра `username`. Поэтому программа `Hello.jsp` лишь распечатывает сообщение "Hello there" с текущим временем и отображает текстовое поле и кнопку передачи для формы ввода данных. При повторном вызове страницы с тем же URL временная метка на выходной странице будет иной.

Если теперь в текстовом окне формы ввести имя **Pam** и щелкнуть мышью на кнопке Submit name, web-серверу будет послан новый запрос как результат исполнения функции HTML-формы:

- `http://localhost:8080/examples/jsp/chapter10/Hello.jsp?username=Pam`

Обратите внимание на то, что к первоначальному URL вызова `Hello.jsp` добавляются символ `'?'` и строка запроса `username=Pam`, это соответствует правилам метода `GET`. Символ `'?'` выступает в роли разделителя URL страницы и параметров HTTP-запроса. На рис. 10.2 показан результат, генерируемый в браузере этим запросом. Как видим, JSP успешно считала параметр `username` в URL запроса и распечатала приветствие "Hello Pam", а не "Hello there", как это было на рис. 10.1.

Рис. 10.2. Выходные данные `Hello.jsp` после ввода Pam в текстовом окне



Заглянем внутрь

Теперь мы видим, насколько прост процесс создания и выполнения программы Hello.jsp, однако стоит посмотреть, что происходит "за кулисами". Двигатель JSP (называемый также контейнером JSP) несет ответственность за управление программами JSP и за их исполнение (см. главу 2). Обычно он работает как сервлет с отображенным расширением, т.е. вызывается базовым web-сервером и двигателем сервлетов, если URL запроса заканчивается расширением .jsp. Двигатель JSP выполняет некоторые операции автоматически, скрывая многие детали и ускоряя процесс разработки приложения. После первого вызова страницы программа JSP преобразуется в код сервлетов Java, который затем компилируется при помощи компилятора Java, и в результате для класса сервлета генерируется байтовый код. Затем сгенерированный класс сервлета загружается, инициализируется и выполняется, создавая выходную страницу, отсылаемую обратно HTTP-клиенту. Позже класс сервлета можно загрузить еще раз или выгрузить с помощью двигателя JSP.

Заметим, что JSP-файл можно преобразовать заранее, предотвратив задержку ответа из-за накладных расходов, затрачиваемых на трансляцию при первом вызове JSP. Для этой цели в состав JSP-реализации Oracle входит инструментальное средство командной строки, называемое ojspc (см. ниже раздел "Реализация JSP в Oracle").

Интересно заглянуть в Java-программу Hello.java, генерируемую для Hello.jsp. Элементы генерируемой программы зависят от реализации, но основным соглашениям, принятым в спецификации JSP, должны удовлетворять все двигатели JSP. В листинге 10.1 представлена программа, генерируемая транслятором Oracle JSP 1.0. Она сопровождается ссылками на подробные пояснения, следующие за листингом. Отметим, что в генерируемой программе для решения задач, специфичных для реализации, используются библиотеки Oracle JSP. Кроме того, в Oracle JSP 1.1 программа будет иной, хотя концептуально структура сохранится. К важным моментам листинга 10.1 относится следующее:

- **Для JSP генерируется класс Java.** Имя генерируемого класса зависит от реализации; для простоты предположим, что он называется Hello. Этот класс можно разместить в пакете, имя которого также определяется реализацией; специального обязательного алгоритма для этого в спецификации JSP нет. Обычно имя пакета используется только двигателем JSP и не имеет прямого отношения к программисту, создающему JSP.
- **В генерируемом классе описывается метод** `_jspService()`. Этот метод является главной точкой входа для вызова JSP. Он содержит программные конструкции для записи в выходной поток как статического, так и динамического содержимого JSP-страницы. В частности, все скриптлеты и выражения JSP входят в состав тела этого метода. Имя и сигнатура метода соответствуют стандартному соглашению (контракту), описанному в спецификации JSP. Он принимает два аргумента: HTTP-объекты request и response. При исполнении он может устанавливать исключительные ситуации.

Листинг 10.1. Программа, генерируемая из Hello.jsp

```
package examples.jsp.chapter10; // (см. пояснение 1)

import oracle.jsp.runtime.*; // (см. пояснение 2)
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import java.beans.*;
```

```
public class Hello extends oracle.jsp.runtime.HttpJsp {
                                        // (см. пояснение 3)
  public final String _globalsClassName = null;
  // ** Начало объявлений
  String defaultName = "there";  // (см. пояснение 4)
  // ** Конец объявлений
  public void _jspService(HttpServletRequest request,
                          HttpServletResponse response)
                throws IOException, ServletException {
                                        // (см. пояснение 5)
    /* Установим внутренние переменные с помощью pageContext:
    ** session = HttpSession
    ** application = ServletContext
    ** out = JspWriter
    ** page = this
    ** config = ServletConfig
    ** все сеансовые/прикладные зерна, объявленные в globals.jsa
    */
      JspFactory factory = JspFactory.getDefaultFactory();
                                        // (см. пояснение 6)
      PageContext pageContext = factory.getPageContext(this,
          request, response, null, true,JspWriter.DEFAULT_BUFFER, true);
      // Обратите внимание: это не указывается, если директива
      // сеанса == false
      HttpSession session = pageContext.getSession();
      ServletContext application = pageContext.getServletContext();
      JspWriter out = pageContext.getOut();
      Hello page = this;
      ServletConfig config = pageContext.getServletConfig();
      try {
          out.println( "<HTML>"); // (см. пояснение 7)
      out.println( "    <HEAD> ");
          out.println( "      <TITLE> The Hello JSP </TITLE> ");
          out.println( " </HEAD> ");
          out.println( " <BODY BGCOLOR=\"white\">");
          out.print ( "  ");
          out.println( "");
          out.print ( " " );
          String name = request.getParameter("username");
                                        // (см. пояснение 8)
          out.println( "");
          out.print( "   <H3> Hello ");
          Out.print(  (name == null)? defaultName : name );
                                        // (см. пояснение 9)
          out.println( " ! </H3>");
          out.print ( "   <P><B> The current time is ");
          Out.print(  new java.util.Date() ); // (см. пояснение 10)
          out.println( ". ");
          out.println( "      </B></P>");
          out.println( "      <B>May I have your name?</B>");
          out.println( "      <FORM METHOD=GET> ");
          out.println( "          <INPUT TYPE=\"text\"");
          out.println( "                 NAME=\"username\"");
          out.println( "                 SIZE=15>");
          out.println( "          <INPUT TYPE=\"submit\"");
          out.println( "              VALUE=\"Submit name\">");
          out.println( "          </FORM>");
          out.println( "      <P><B> Have a good day! :-) </B></P>");
          out.println( "  </BODY>");
          out.println( "</HTML>");
          Out.flush();    // (см. пояснение 11)
      }
      catch( Exception e) {  // (см. пояснение 12)
```

```
            try { if (out != null) out.clear();}
            catch( Exception clearException) {
            }
            pageContext.handlePageException( e) ;
        }
        finally {  // (см. пояснение 13)
            if (out != null) out.close();
            factory.releasePageContext(pageContext);
        }
    }
}
```

Пояснения к программе `Hello.java`, генерируемой из `Hello.jsp`:

1. Это объявление пакета для программы `Hello.jsp`. В Oracle JSP 1.0 он извлекается из имени маршрута к области установки JSP на web-сервере.
2. Эти операторы `import` генерируются автоматически транслятором JSP для импорта необходимых библиотек JSP, сервлетов и других служебных пакетов.
3. Этот оператор объявляет общий класс Java с именем `Hello`, которое в Oracle JSP 1.0 является базовым именем JSP-файла `Hello.jsp`. Класс `Hello` расширяет класс `oracle.jsp.runtime.HttpJsp` — абстрактный класс, который описан в библиотеке JSP этапа выполнения, предоставляемой в Oracle. В нем присутствуют стандартные (пустые) реализации таких методов, как `jspInit()` и `jspDestroy()`, которые при необходимости вызываются двигателем JSP для инициализации и уничтожения экземпляра JSP-страницы. К тому же эти методы могут быть переопределены программистом явным объявлением методов Java на JSP-странице, хотя подобное встречается довольно редко. Точное описание абстрактного класса `HttpJsp` зависит от реализации, но в нем должен присутствовать стандартный интерфейс `javax.servlet.jsp.HttpJspPage`, как того требует спецификация Servlet 2.2 API (см. приложение А).
4. Этот оператор соответствует объявлению Java-переменной `defaultName`, ограниченной тэгами `<%!...%>`, в программе `Hello.jsp`. Заметим, что это — объявление уровня класса, т.е. имя переменной доступно всем методам класса `Hello`.
5. Этот оператор описывает общий метод с именем `_jspService()`, который вызывается для обработки HTTP-запроса клиента. Тело этого метода автоматически генерируется транслятором JSP на основе содержимого JSP- страницы. Как правило, тело метода состоит из трех частей:
   - **Программные конструкции установки**, где инициализируются необходимые объекты HTTP и JSP этапа выполнения.
   - **Программные конструкции пользователя**, основанные на JSP-странице. В листинге 10.1 эти конструкции представлены в виде серии операторов `out.print()` и `out.println()`. Возможны и другие схемы генерации программных конструкций; в Oracle JSP 1.1 они оптимизируются за счет использования для постоянного HTML-текста символьных массивов вместо строк символов. Несмотря на различия генерируемых программ, в разных JSP-реализациях сохраняется одна и та же основная характеристика: статическое содержимое и динамические скриптлеты и выражения располагаются в том же порядке, что и на исходной JSP- странице. При обнаружении ошибок на этапе выполнения могут устанавливаться исключительные ситуации (например, `javax.servlet.ServletException` или `java.io.IOException`).
   - **Программные конструкции обработки исключительных ситуаций и придания окончательного вида**. Перед завершением программы выходной поток очищается или сбрасывается на диск, а внутренние ресурсы высвобождаются.

6. Этот оператор входит в состав программных конструкций установки. Он извлекает источник по умолчанию из класса `javax.servlet.jsp.JspFactory` (см. приложение D). Этот источник используется для создания объекта `pageContext`, связываемого с JSP-страницей. Объект `pageContext` содержит разнообразные зависящие от реализации свойства двигателя JSP и управляет пространствами имен области действия (`request`, `page`, `session` и `application`). Следующие пять операторов также являются частью фазы установки JSP, на которой с помощью контекста страницы инициализируются такие объекты, как HTTP-сеанс и выходной поток, путем вызова соответствующих методов.
7. Этот оператор является началом HTML-текста на JSP. Он записывается в выходной поток.
8. Этот оператор генерируется из JSP-скриптлета, который считывает значение параметра `username` в объекте `request`.
9. Этот оператор генерируется из JSP-выражения, которое встроено в приветственное сообщение.
10. Этот оператор генерируется из JSP-выражения, которое распечатывает текущее время.
11. Этот оператор указывает на окончание программных конструкций пользователя и сбрасывает информацию выходного буфера.
12. Этот оператор является частью третьего и последнего раздела метода `_jspService()`. Он автоматически генерируется для перехвата всех исключительных ситуаций, которые могут возникнуть во время исполнения JSP. В случае ошибки выходной буфер очищается, и исключительная ситуация передается объекту `pageContext` для обработки и выдачи отчета об ошибках.
13. Это программная конструкция придания окончательного вида, выполняемая перед выходом из JSP-программы. Выходной буфер закрывается, а объект `pageContext` освобождается.

В процессе создания приложений JSP обычно не требуется знать все особенности генерируемой программы, описанные выше. Отладка JSP-программ поддерживается в Oracle JDeveloper; можно установить в исходном тексте JSP-страницы точки прерывания и выполнить программу в локальном двигателе сервлетов. Такая функциональная возможность доступна в JDeveloper версии 3.1 и выше.

Директивы JSP

В предыдущем разделе было показано, как встраиваются в программный текст Java объявления, скриптлеты и выражения JSP. Теперь рассмотрим директивы JSP, которые используются для передачи информации, необходимой для трансляции JSP, такой как:

- Java-операторы `import`
- Файлы, статически подключаемые во время трансляции JSP
- Тип содержимого (MIME-тип и набор символов) JSP и выходной страницы

В этом разделе рассказывается о каждом из указанных видов информации. Другие способы использования директив JSP будут продемонстрированы в последующих разделах этой главы.

Директива JSP ограничивается тэгами `<%@...%>`. Ее общий синтаксис:

- `<%@ директива атрибут1="значение1" атрибут2="значение2" ... ... %>`

Цель директивы JSP – предоставить информацию транслятору JSP. Она воздействует на программный текст, генерируемый для JSP, но не создает непосредственно содержимое выходной страницы. Значениями атрибутов не могут быть параметры этапа выполнения из HTTP-запроса.

Существуют три типа директив JSP: директива page, директива include и директива taglib. Набор разрешенных атрибутов определяется типом используемой директивы. Ниже рассказывается о применении директив page и include и их атрибутов. Директива taglib рассматривается в главе 12. Описание всех директив JSP и их атрибутов дается в приложении D.

Директива page

Директива page используется для предоставления транслятору JSP информации, применяемой по отношению ко всей странице. У нее несколько атрибутов; в этом разделе говорится о наиболее важных: import и contentType. Полный перечень атрибутов директивы page приведен в приложении D.

На JSP-странице и на других страницах, включаемых в нее статически, может присутствовать одна или несколько директив page. Однако каждый из атрибутов, за исключением атрибута import, можно указывать лишь однажды. Приведем описание наиболее часто используемых атрибутов:

- import. Этот атрибут указывает один или несколько классов Java, которые нужно импортировать в транслируемую программу. Например, следующая директива page импортирует библиотеки java.sql.*:

  ```
  <%@ page import="java.sql.*" %>
  ```

  Эту директиву можно записать и в альтернативном XML-обозначении:

  ```
  <jsp:directive.page import="java.sql.*" />
  ```

  В соответствии с соглашениями Java директивы page для импорта пакетов размещаются, как правило, в верхнем разделе JSP. Одной директивой page можно импортировать несколько пакетов, для чего указывается список классов Java и пакетов, разделенных запятыми:

  ```
  <%@ page import="java.sql.*,sqlj.runtime.ref.DefaultContext" %>
  ```

  Такой же результат дает применение нескольких директив page на разных строках с отдельными атрибутами import, например:

  ```
  <%@ page import="java.sql.*" %>
  <%@ page import="sqlj.runtime.ref.DefaultContext" %>
  ```

- language. Этот атрибут описывает язык сценариев, используемый для JSP. Стандартным значением по умолчанию данного атрибута является java. В JSP-реализации Oracle допустимо и значение sqlj:

  ```
  <%@ page language="sqlj" %>
  ```

  Эта директива разрешает применение операторов #sql в JSP, что позволяет встраивать SQL-операторы. В этом случае во время преобразования страницы транслятор JSP вызывает транслятор SQLJ. Использование SQLJ-операторов в JSP продемонстрировано ниже.

- buffer. Этот атрибут определяет, буферизовать ли выходной поток, используемый для записи конечного содержимого страницы. Режимом по умолчанию является буферизация выходного потока, что, как правило, и требуется. Значение none задается, если буферизация не нужна. В этой директиве page указывается буфер размером 20 Кбайт:

  ```
  <%@ page buffer="20kb" %>
  ```

  Размер буфера по умолчанию определяется реализацией, но должен быть равен 8 Кбайт или больше.

- autoFlush. Этот логический атрибут связан с атрибутом buffer. Он определяет поведение буферизуемого выходного потока: сбрасывать ли при заполнении содержимое буфера автоматически или устанавливать исключительную ситуацию, сигнализирующую о переполнении. Значение по умолчанию этого атрибута — true.

- isErrorPage. Этот логический атрибут указывает, может ли данная JSP-страница обращаться к исключительной ситуации, установленной на

другой странице притом, что данная страница вызвана в результате этой исключительной ситуации. Значением по умолчанию является false, и в этом случае неявный объект exception нельзя использовать на JSP (он будет вызывать ошибку трансляции). Если установить этот атрибут в значение true, информация об исключительной ситуации будет доступна скриптлетам и выражениям данной JSP через неявно определяемый объект exception. Данный атрибут связан с атрибутом errorPage, описанным ниже.

- errorPage. Этот атрибут описывает URL другой JSP для обработки ошибок. При возникновении ошибки на JSP-странице, где есть эта директива, обработка передается по указанному URL, причем сообщаются еще и характеристики установленной исключительной ситуации. JSP, которой передается ошибка, описывается обычно как страница ошибок, т.е. на ней присутствует директива page, атрибут isErrorPage которой установлен в значение true. URL по умолчанию для обработки ошибок определяется реализацией JSP. В Oracle JSP для обработки неперехваченных исключительных ситуаций описывается собственная страница ошибок. Пример использования атрибутов errorPage и isErrorPage приведен ниже в разделе "Обработка ошибок JSP".
- contentType. Этот атрибут определяет MIME-тип и набор символов JSP-страницы. Он применяется к тексту страницы во время трансляции, а также к выходным данным во время исполнения. Значение по умолчанию для MIME-типа — text/html, а для кодировки набора символов — ISO-8859-1 (или Latin-1). Название набора символов — это имя IANA (Internet Assigned Numbers Authority — управление по назначению номеров в Интернете; имена IANA для различных наборов символов можно узнать на web-странице www.iana.org/numbers.htm). Например, следующим оператором устанавливается японская кодировка набора символов Shift_JIS:

```
<%@ page contentType="text/html;charset=Shift_JIS" %>
```

Для правильной трансляции директива page с установкой contentType должна указываться в начале JSP, перед использованием символов, не относящихся к Latin-1, и перед сбросом выходного буфера (когда выходной поток инициализируется информацией о наборе символов).

Директива include

Директива include используется для физического ввода содержимого некоторого файла во время трансляции JSP. Будьте внимательны и не путайте эту директиву с функциональным тэгом <jsp:include>, который применяется для подключения файлов во время исполнения JSP. О тэге include говорится в разделе "Вызов других страниц с JSP". Директиву include иногда называют статическим подключением, а функциональный тэг include — *динамическим* подключением.

Синтаксис директивы include:

❒ `<%@ include file="URL_файла" %>`

В совместимом с XML обозначении директива include записывается следующим образом:

❒ `<jsp:directive.include file="URL_файла" />`

У этой директивы один обязательный атрибут — подключаемый файл. Файл указывается с *относительным* URL, который может задаваться:

- **Относительно страницы**, при этом он не начинается с символа '/'. В данном случае путь к файлу определяется относительно местонахождения текущей JSP.

- **Относительно приложения или относительно контекста**, при этом он начинается с символа '/'. Такой синтаксис URL предполагает, что путь к файлу определяется относительно корня JSP-приложения. Как говорилось в главе 2, в спецификации Servlet 2.2 описано понятие корня web-приложения, которое связано с контекстом сервлетов для данного приложения. Для сред сервлетов, основанных на файловых системах (например, для web-сервера Apache с двигателем сервлетов Tomcat), корень приложения соответствует каталогу файловой системы. Все файлы, принадлежащие web-приложению, размещаются в этом каталоге.

Директива include обязывает транслятор JSP вводить содержимое указанного файла в то место, где записана директива. Другими словами, во время трансляции JSP эта директива замещается содержимым файла. Обычно include применяется для вставки общего "заголовочного" текста, например сообщений об авторских правах и информации баннеров. Подключаемый файл удобно также использовать для хранения директив page, которые являются общими для группы JSP-страниц, например, при импорте классов и пакетов Java.

Ниже приводится пример простой JSP-программы StaticInclude.jsp, где директива include используется для ввода двух файлов, Imports.jsp и Banner.htm, в основной текст JSP:

```
<%-- Имя программы:  StaticInclude.jsp
  -- Назначение:     Подключить другие страницы во время трансляции.
  --%>
<HTML>
  <HEAD> <TITLE> The StaticInclude JSP </TITLE> </HEAD>
  <BODY>
    <%@ include file="Imports.jsp"%>
<H3> Hello! The current time is: <%= new Date() %> </H3> <BR><HR>
    <%@ include file="common/Banner.htm" %>
  </BODY>
</HTML>
```

Предполагается, что файл Imports.jsp находится в том же каталоге, что и основная JSP, а файл Banner.htm — в подкаталоге common относительно местоположения файла StaticInclude.jsp. Для выполнения этого примера опишем простой файл:

```
<!-- Banner.htm -->
<H2><I>...Информация баннера... </I></H2>
```

Файл Imports.jsp предназначен для хранения общих операторов импорта, например:

```
<!-- Imports.jsp -->
<%@ page import="java.sql.*, java.util.Date" %>
<%@ page import="sqlj.runtime.ref.DefaultContext" %>
```

На рис. 10.3 показан результат, получаемый в браузере после исполнения StaticInclude.jsp. Обратите внимание на то, что информация баннера размещается внизу страницы.

Вызов других страниц с JSP

JSP-программа может вызывать другие страницы. Для этого применяются два функциональных тэга:

- Тэг <jsp:include>, который вызывает другую страницу, а затем возвращает управление вызывающей JSP.
- Тэг <jsp:forward>, который передает управление другой странице и закрывает вызывающую JSP.

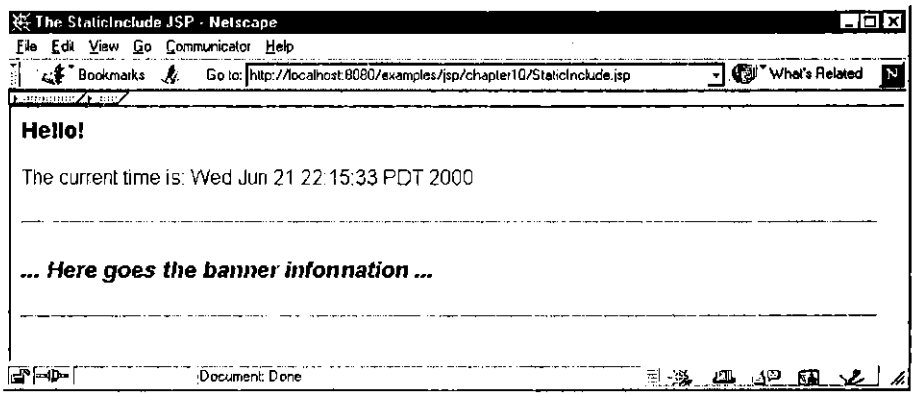


Рис. 10.3. Выходные данные после исполнения StaticInclude.jsp

Эти тэги используются, как правило, для перехода от одной страницы к другой на основе динамических условий. В следующих примерах показано применение тэгов в приложении JSP.

Тэг <jsp:include>

Тэг <jsp:include> используется для посылки HTTP-запроса с основной JSP на другую страницу. Подключаемой страницей может быть статический текст или HTML-файл, а также динамическая программа, например другая JSP или сервлет. Подключаемая страница обрабатывает запрос и обычно генерирует некие данные, которые вводятся в выходной поток как часть конечной страницы. Вызывающая страница возобновляет свое исполнение после завершения действия тэга include. Синтаксис тэга include:

```
<jsp:include page ="includeURL" flush="true" />
```

Заметим, что синтаксис тэга include совместим с XML. У этого тэга два атрибута: page и flush. Атрибут page указывает относительный URL вызываемой страницы, который может задаваться в виде постоянной строки символов или вычисляться на этапе исполнения при помощи встроенного JSP-выражения. Значением логического атрибута flush всегда должно быть true. Это значит, что буфер выходного потока основной JSP должен сбрасываться до передачи управления подключаемой странице. Сбрасывать буфер нужно обязательно, чтобы подключаемая страница могла вводить свои собственные выходные данные в нужную позицию HTTP-ответа.

Приведем пример программы DynamicInclude.jsp, которая подключает выходные данные HTML-страницы IncludedHello.htm и JSP-программы Time.jsp, вызывая их при помощи тэга include:

```
<%-- Имя программы:  DynamicInclude.jsp
  -- Назначение:     Вызвать другие страницы на этапе выполнения
  --%>
<HTML>
  <HEAD> <TITLE> The Dynamic Include JSP </TITLE> </HEAD>
  <BODY BGCOLOR="white">
    <% String includedHtml = "include/IncludedHello.htm"; %>
    <P> <jsp:include page="<%= includedHtml %>" flush="true"/>
    <B> <jsp:include page="Time.jsp" flush="true"/> </B>
  </BODY>
</HTML>
```

Обратите внимание: URL подключаемой HTML-страницы устанавливается динамически при помощи JSP-выражения, причем за основу берется значение, присваиваемое переменной includedHtml в JSP-скриптлете. В данном примере присваиваемое значение — это постоянная строка символов "include/IncludedHello.htm", но вообще его можно вычислять по динамическому условию. Возможность динамического определения URL подключаемой страницы является основным отличием тэга <jsp:include> от директивы include этапа трансляции (см. выше).

Файл IncludedHello.htm должен находиться в подкаталоге include относительно местоположения DynamicInclude.jsp. Создадим простой файл IncludedHello.htm:

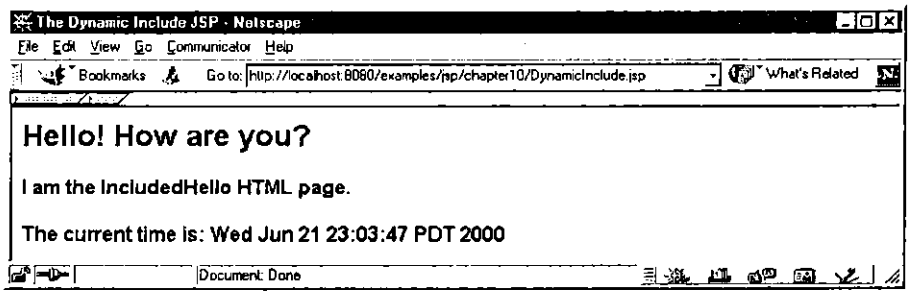
```
<!-- Имя программы:  IncludedHello.htm
  -- Назначение:     Распечатать статический текст
  --!>
<H2> Hello! How are you? </H2>
<P><B> I am the IncludedHello HTML page.
<P>The current time is: </B>
```

Судя по URL страницы, используемому в тэге include, файл Time.jsp должен находиться в том же каталоге, что и файл основной JSP. На странице Time.jsp содержится следующий программный текст:

```
<%-- Имя программы: Time.jsp --%>
<%= new java.util.Date() %>
```

Если теперь указать в браузере URL для DynamicInclude.jsp, к выходной странице подключится содержимое файла IncludedHello.htm, за которым будет следовать текущее время, вычисляемое программой Time.jsp. Отметим, что подключаемая JSP будет автоматически преобразовываться перед вызовом, если, конечно, она не была преобразована заранее. Типичный результат вызова DynamicInclude.jsp, отображаемый браузером, показан на рис. 10.4.

Рис. 10.4. Выходные данные после выполнения DynamicInclude.jsp



В примере не показано, но стоит сказать, что в JSP-приложении разрешается применять несколько уровней функции include. Это может стать для читателя поводом попробовать свои силы в программировании.

В некоторых приложениях необходимо вводить дополнительную информацию в HTTP-запрос перед его отправкой на обработку подключаемой странице. Для этого в спецификации JSP 1.1 предусмотрен специальный тэг <jsp:param> (в JSP 1.0 этот тэг недоступен). Его используют для указания дополнительных параметров запроса в виде пар имя=значение, которые добавляются к исходному HTTP-объекту request вызывающей JSP. Синтаксис тэга <jsp:param>:

```
<jsp:param name="имя_параметра" value="значение_параметра" />
```

Синтаксис тэга include при использовании дополнительных параметров запроса:

```
<jsp:include page ="includeURL" flush="true" >
    <jsp:param name= "новый_параметр1" value="новое_значение1" />
    <jsp:param name= "новый_параметр2" value="новое_значение2" />
</jsp:include>
```

В качестве упражнения читатель может написать JSP с использованием дополнительных параметров запроса для вызова подключаемой страницы.

Тэг <jsp:forward>

Синтаксис функционального тэга для передачи запроса:

```
<jsp:forward page="передаваемый_URL" />
```

Результатом работы этой функции являются отправка HTTP-запроса по URL указанной страницы и прекращение исполнения текущей страницы. Страница, которой переходит управление, может быть статическим HTML- или текстовым файлом, а также программой JSP или сервлетом. Если вызывающая JSP уже записала какие-то данные в выходной буфер, то перед передачей запроса буфер очищается. Для добавления параметров к HTTP-запросу перед его передачей применяются вложенные тэги <jsp:param>, для чего используется синтаксис, схожий с тем, что был приведен выше для тэга <jsp:include>.

Рассмотрим страницу Forward.jsp, иллюстрирующую концепцию передачи запроса:

```
<!-- Имя программы:  Forward.jsp
  -- Назначение:     Передать обработку по условию другим страницам.
  -- ! >
<HTML>
  <HEAD><TITLE> The Forward JSP </TITLE></HEAD>
  <BODY>
    <%  if (request.getParameter("username") == null) { %>
          <jsp:forward page="forward/ForwardedHello.htm" />
    <% } else { %>
          <jsp:forward page="forward/WelcomeUser.jsp" />
    <% } %>
  </BODY>
</HTML>
```

Приложение Forward.jsp проверяет, есть ли в HTTP-запросе параметр с именем username. Если этого параметра нет или если его значение null, запрос передается статическому HTML-файлу ForwardedHello.htm, находящемуся в подкаталоге forward относительно текущей страницы. Если значение параметра username не равно null, запрос передается программе WelcomeUser.jsp, хранящейся в подкаталоге forward. Файл ForwardedHello.htm представлен ниже:

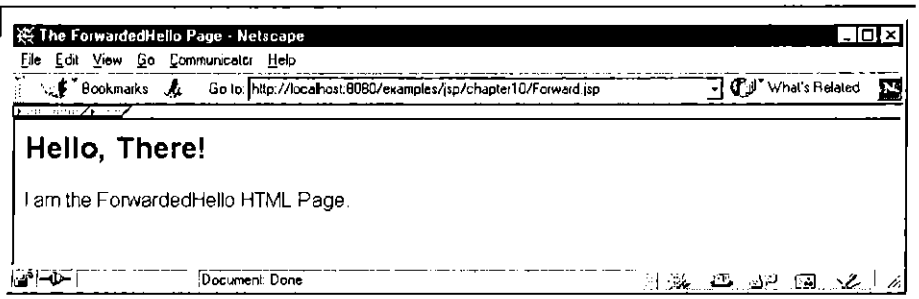
```
<!-- Имя программы:   ForwardedHello.htm
  -- Назначение:      Распечатать статический текст.
  --!>
<HTML>
  <HEAD><TITLE> The ForwardedHello Page </TITLE></HEAD>
  <BODY BGCOLOR="white">
    <H2> Hello, There!  </H2>
    <P> I am the ForwardedHello HTML Page.
  </BODY>
</HTML>
```

Назначение страницы WelcomeUser.jsp — вывод имени пользователя в приветственном сообщении. Приведем ее программный текст:

```
<%-- Имя программы: WelcomeUser.jsp
  -- Назначение: Распечатать приветственное сообщение для
  -- параметра username.
  --%>
<HTML>
  <HEAD><TITLE>The WelcomeUser JSP </TITLE></HEAD>
  <BODY>
      <H2> Welcome <%= request.getParameter("username") %> ! </H2>
  </BODY>
</HTML>
```

Для выполнения этой JSP файл Forward.jsp следует установить в каталоге examples/jsp/chapter10 web-сервера, а файлы ForwardedHello.htm и WelcomeUser.jsp — в подкаталоге forward относительно Forward.jsp (см. приложение А). Если затем указать браузеру URL для Forward.jsp, тот отобразит выходную страницу, приведенную на рис. 10.5.

Рис. 10.5. Выходные данные после исполнения Forward.jsp без параметров запроса



Стоит отметить два важных момента относительно выходных данных, генерируемых страницей Forward.jsp:

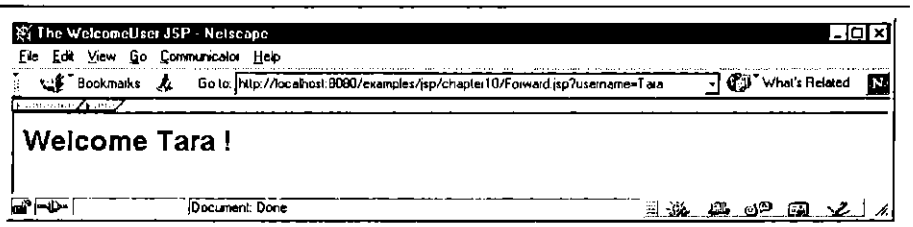
- Поскольку URL вызова не содержит параметров запроса, управление исполнением передается странице ForwardedHello.htm, содержимое которой отображается браузером.
- Название выходной страницы – "The ForwardedHello Page". Заметьте, что строка с названием вызывающей JSP (т.е. Forward.jsp) – "The Forward JSP" – расположена до операции forward. Однако в соответствии с семантикой forward выходной буфер перед передачей очищается. Следовательно, название конечной страницы определяется страницей, которой передается управление, т.е. в данном случае – страницей ForwardedHello.htm. Если исследовать источник генерируемой страницы, обнаружится, что он состоит только из содержимого данной страницы.

На WelcomeUser.jsp применяется другой вариант передачи – вызывается Forward.jsp с параметром запроса username, содержащим не null-значение. Укажем браузеру соответствующий URL, например:

```
http://localhost:8080/examples/jsp/chapter10/Forward.jsp?username=Tara
```

Параметр запроса и его значение добавляются к базовому URL в виде пары *имя=значение* после разделительного символа '?', что соответствует соглашению, принятому для HTTP-метода GET. Результат, выдаваемый браузером в этом случае, показан на рис. 10.6. Еще раз отметим, что название, отображаемое в заголовке окна браузера,– это название страницы, которой передается управление, а не вызывающей JSP.

Рис. 10.6. Выходные данные после исполнения Forward.jsp с параметром username



Использование JSP-страниц вместе с сервлетами

Важно понимать, каким образом JSP-страницы и сервлеты работают совместно. В некоторых случаях удобнее создать сервлет, а не JSP (см. раздел "Сравнение JSP и сервлета" главы 2). Например, для получения двоичных данных механизм JSP не совсем удобен. Дело в том, что в JSP для вывода данных используется объект javax.servlet.jsp.JspWriter, а его методы хорошо применимы только по отношению к простому тексту (именно для этого в первую очередь и предназначены JSP-страницы). Для получения двоичных данных обычно приходится применять сервлет. При этом выходной поток, используемый для вывода двоичных данных, можно получить с помощью метода getOutputStream() HTTP-объекта response. Таким образом, в ряде случаев нужно применять в web-приложении как сервлеты, так и JSP-страницы.

Посмотрим, как можно вызывать JSP из сервлета и наоборот. В приводимых примерах предполагается, что используется двигатель сервлетов, поддерживающий Servlet 2.1 API или выше, например JSWDK или Tomcat. Дело в том, что в спецификации Servlet 2.0 (это версия, реализуемая исполнителем сервлетов JServ для Apache) некоторые свойства отсутствуют. Так, в спецификации Servlet 2.0 API не доступен интерфейс `javax.servlet.RequestDispatcher`; он появился только в Servlet 2.1 API.

Вызов сервлета с JSP-страницы

В реализациях сервлетов, соответствующих Servlet 2.1 и выше, для вызова сервлета с JSP-страницы используются стандартные тэги `<jsp:include>` и `<jsp:forward>`. Ниже приводится текст JSP-программы `CallServlet.jsp`, в которую включаются выходные данные, генерируемые классом сервлета с именем `CalledServlet`. Общая структура программного текста JSP должна быть вам уже хорошо знакома. Заметим, что выходные данные сервлета вводятся в выходные данные JSP посредством тэга `<jsp:include>`, который действует как вызов метода:

```
<%-- Имя программы:  CallServlet.jsp
  -- Назначение:     Вызвать сервлет с JSP.
  --%>
<HTML>
  <HEAD> <TITLE> The CallServlet JSP </TITLE> </HEAD>
  <BODY BGCOLOR="white">

<H3> Hello, I am the CallServlet JSP. </H3>
<p> <b>Below I will include output generated by a servlet .</b><br>
<br><hr><i>
<jsp:include page="/servlet/chapter10.CalledServlet" flush="true" />
</i><br><hr>

<p> <b>You are now back to the CallServlet JSP.</b>
  </BODY>
</HTML>
```

Далее приводится текст программы `CalledServlet.java`. Этот сервлет извлекает объект `PrintWriter` из аргумента `response` и распечатывает сообщение:

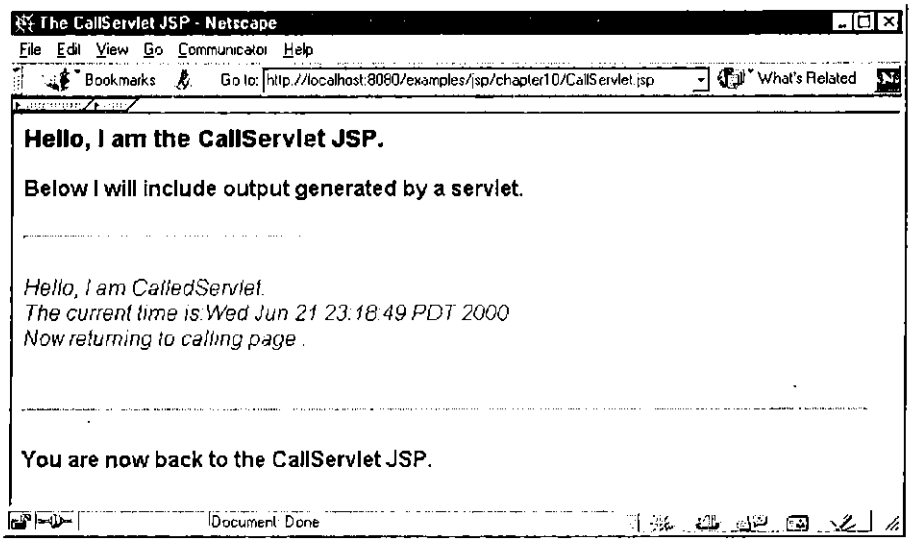
```
/** Имя программы: CalledServlet.java
** Назначение: Распечатать приветственное сообщение.
**/
package chapter 10;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;

public class CalledServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
                throws IOException {
        PrintWriter out= response.getWriter();
        out.println("Hello, I am CalledServlet.");
        out.println("<BR>The current time is:"+ new java.util.Date()
                +"<BR>");
        out.println("Now returning to calling page..");
    }
}
```

Установите эти программы JSP и сервлет на своем web-сервере (см. приложение E). На рис. 10.7 показан результат, генерируемый при вызове `CallServlet.jsp`.

Рис. 10.7. Выходные данные CallServlet.jsp



С помощью тэга <jsp:forward> можно передавать управление исполнением с JSP-страницы сервлету. Напомним, чем include отличается от forward. Среди выходных данных forward содержится информация только той страницы, которой передано управление, а информации страницы-источника нет. В противоположность этому, выходные данные оператора <jsp:include> физически размещаются в том месте, где оператор указан на JSP, и являются лишь частью конечного ответа, посылаемого браузеру. После возврата управления от подключаемой программы вызывающая программа продолжает свое исполнение, как и при вызове метода.

Возникает вопрос: как же передавать информацию с вызывающей страницы на подключаемую или на новую управляющую страницу. Одним из способов является использование тэга <jsp:param>, который доступен в реализациях JSP 1.1. Существует несколько других способов, например с помощью HTTP-объектов session или request либо посредством JavaBean. О применении атрибутов объекта request для передачи данных говорится в следующем разделе; этот вариант можно применять при вызове с JSP-страницы сервлета или другой JSP-страницы. О совместном использовании информации страниц при помощи JavaBean с правильной областью действия рассказывается в главе 11.

Вызов JSP из сервлета

Можно также подключить или передать HTTP-запрос из сервлета на JSP, что требует использования интерфейса javax.servlet.RequestDispatcher, появившегося в спецификации Servlet 2.1 API (см. ниже). При помощи этого интерфейса реализуются тэги <jsp:include> и <jsp:forward>. Следовательно, многое из того, о чем говорилось выше, вполне применимо и к функциональным тэгам include и forward. Хотя большинство операций производится исполняющей системой JSP и не программируется вручную, знание интерфейса RequestDispatcher() поможет лучше понять принцип действия тэгов <jsp:include> и <jsp:forward>.

Интерфейс RequestDispatcher

Интерфейс RequestDispatcher реализует механизм вызова ресурсов web-приложения из сервлета или с JSP. Методы этого интерфейса служат для передачи текущего HTTP-запроса другой программе – обычно сервлету или JSP – для дальнейшей обработки. Методы RequestDispatcher и других интерфейсов Servlet 2.2 описаны в приложении А. Процесс диспетчирования (передачи запроса другой программе) при помощи этого интерфейса состоит, как правило, из четырех операций:

1. Сначала нужно получить объект ServletContext. Это можно сделать, обратившись к экземпляру HTTP-сервлета с помощью его метода getServletContext():

```
javax.servlet.ServletContext sc = getServletContext();
```

2. Затем нужно извлечь объект RequestDispatcher из объекта ServletContext, вызвав его метод getRequestDispatcher() и указав в качестве аргумента относительный для приложения URL вызываемой программы:

```
javax.servlet.RequestDispatcher rd =
            sc.getRequestDispatcher("/jsp/chapter10/CalledJSP.jsp")
```

3. Нередко требуется совместно использовать данные подключаемой или новой управляющей страницы и вызывающей страницы. Для этого существует несколько способов:

   - В вызове метода getRequestDispatcher() можно добавить к URL страницы строку запроса, используя соглашение '?' и пары имя-значение, например:

```
javax.servlet.RequestDispatcher rd =
  sc.getRequestDispatcher(
              "/jsp/chapter10/CalledJSP.jsp?username=jspuser");
```

     В этом случае параметр username и его значение становятся частью HTTP-запроса и могут быть считаны методом request.getParameter(). Такой же способ можно применять в тэгах <jsp:include> и <jsp:forward>. Обратите, однако, внимание на ограничение при передаче дополнительной информации: в этом методе она может быть представлена только строкой символов, так как является частью URL.

   - С помощью метода request.setAttribute() можно связать любой объект Java с HTTP-объектом request до вызова другой страницы. Например:

```
Order o = new Order(); request.setAttribute("cart", o);
```

     После этого атрибут становится доступным на подключаемой или новой управляющей странице и вызывается методом request.getAttribute(). Например:

```
Order o = (Order) request.getAttribute("cart");
```

   - Можно использовать HTTP-объект session и методы session.setAttribute() и session.getAttribute(). Эти методы аналогичны соответствующим методам объекта request; важное отличие их заключается в том, что они доступны на любой странице, вызываемой в одном и том же HTTP-сеансе, а не только для одного HTTP-запроса.

   - Можно применить JavaBean с областью действия сеанса, когда зерно используется всеми страницами HTTP-сеанса (см. главу 11).

     Выбор подхода зависит от требований и структуры web-приложения. Если информация должна разделяться только во время HTTP-запроса, для хранения данных следует использовать объект request. Если же информация имеет отношение ко всем страницам приложения, применяйте объект session или JavaBean с областью действия сеанса.

4. Наконец, после установки совместно используемых данных и диспетчера запросов нужно вызвать метод include() или forward() объекта RequestDispatcher, указав в качестве параметров HTTP-объекты request и response:

```
rd.include(request, response); // подключение страницы
rd.forward(request, response); // переход к странице
```

При использовании метода forward(), в противоположность методу include(), управление процессом исполнения передается вызываемой программе, а вызывающая программа закрывается.

Теперь объединим все эти этапы и напишем завершенный сервлет, который обращается к JSP и передает ей некоторую информацию. Ниже приводится

программный текст сервлета CallJSP, который устанавливает атрибут username в объекте request, а затем вызывает программу CalledJSP, включая ее выходные данные в генерируемую страницу:

```
/** Имя программы: CallJSP Servlet
 ** Назначение: Вызвать JSP из сервлета.
 **/
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
public class CallJSP extends HttpServlet {
    public void doGet (HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException {
             response.setContentType("text/html");
             PrintWriter out = response.getWriter();
             out.println("<html><body bgcolor=\"white\">");
             out.println("<head><title>The CallJSP Servlet</title>");
             out.println("</head><body>");

             out.println("<H3>I am the CallJSP Servlet.</H3>");
             out.println("<B>I will now include the CalledJSP page..</B>");
             out.println("<BR><BR><HR><i>");

             request.setAttribute("username", "jspuser");
             getServletContext();
               getRequestDispatcher("/jsp/chapter10/CalledJSP.jsp");
               include(request, response);

             out.println("</I><HR><BR>");
             out.println ("<P><B>Now you are back to CallJSP servlet.</B>");
             out.println("</body>");
             out.println("</html>");
        }
}
```

Приведем текст программы CalledJSP. Эта страница распечатывает приветственное сообщение и текущее время. Для считывания значения username, установленного сервлетом CallJSP, она использует метод getAttribute() HTTP-объекта request:

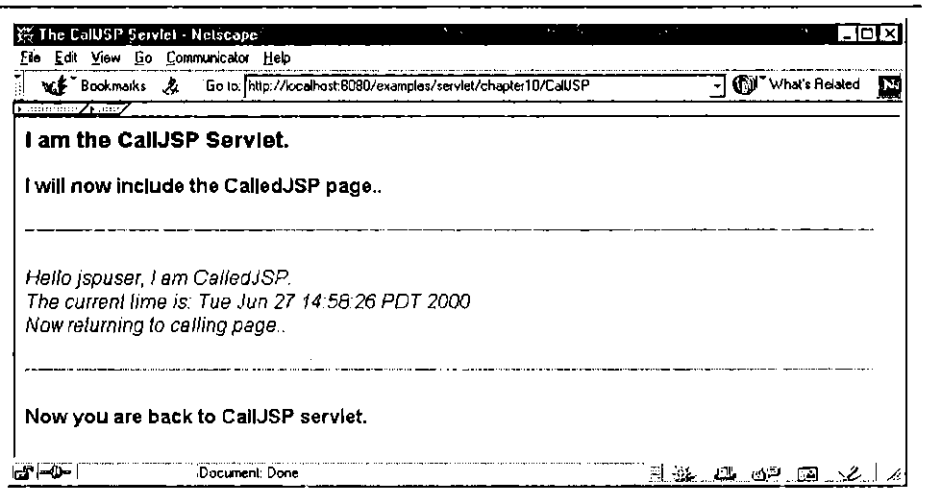
```
/** Имя программы:   CalledJSP.jsp
 ** Назначение:      Извлечь атрибут username из request и распечатать
 **                  приветствие и текущее время.
 **/
<P>Hello <%= request.getAttribute("username") %>, I am CalledJSP.
<P>Hello, I am CalledJSP.
<BR>The current time is: <%= new java.util.Date() %>
<BR>Now returning to calling page..<P>
```

Результат вызова сервлета CallJSP показан на рис. 10.8. Отметим, что после выполнения программы CalledJSP управление возвращается сервлету. В качестве упражнения читателю предлагается написать программу сервлета, который переходит к JSP (а не подключает ее).

Особенности сред Servlet 2.0

В некоторых популярных исполнителях сервлетов, например в JServ, реализуется только Servlet 2.0 API. В таких средах, как Oracle Internet Application Server версии 1.0, основой которого является web-сервер Apache и JServ, интерфейс RequestDispatcher недоступен. Тем не менее тэги <jsp:include> и <jsp:forward> в JSP-реализации Oracle поддерживаются, что достигается эмуляцией некоторых функциональных свойств интерфейса RequestDispatcher в среде

Рис. 10.8. Выходные данные сервлета CallJSP



Servlet 2.0. Однако ограничением является то, что вызывать с помощью этих тэгов можно только JSP-страницы (в Oracle JSP 1.1 посредством функциональных тэгов сред Servlet 2.0 можно вызывать еще и HTML-страницы; такая функциональная возможность была добавлена, когда писалась эта книга). HTML-страницы можно переименовывать или копировать, устанавливая расширения .jsp. Обойти проблему для сервлетов можно, создав оболочку, т.е. "завернув" сервлет в JSP так, чтобы его было удобно вызывать с JSP-страницы при помощи функциональных тэгов include и forward. Ниже приводится программный фрагмент, где показано, как JSP-страница ServletWrapper.jsp при обращении к ней вызывает базовую программу сервлета CalledServlet (см. выше). Она инициализирует и уничтожает внутренний сервлет методами jspInit() и jspDestroy() соответственно и вызывает метод doGet() сервлета CalledServlet через JSP-скриптлет как часть служебного запроса JSP.

```
<%-- Имя программы:      ServletWrapper.jsp
  -- Назначение:         Вызвать программу сервлета без применения JSP-тэгов и
  --                     интерфейса RequestDispatcher. Это удобно делать
  --                     в средах Servlet 2.0.
  --%>
<%@ page isThreadSafe="true" import="CalledServlet" %>
<%!  CalledServlet calledServlet = null;
     public void jspInit() {
         calledServlet = new CalledServlet();
         try {
           calledServlet.init(this.getServletConfig());
         } catch (ServletException se) {
             calledServlet = null;
         }
     }
     public void jspDestroy() {
           if (calledServlet != null)
               calledServlet.destroy();
     }
 %>

<% if (calledServlet != null) {
      calledServlet.doGet(request, response);
   }
   else {
    throw new JspException("Error initializing servlet!!");
   }
%>
```

Обратите внимание на использование атрибута isThreadSafe директивы page в ServletWrapper.jsp. По умолчанию значение этого атрибута — true, но если поток базового сервлета не защищен, это значение следует изменить на false. Теперь можно вызвать страницу ServletWrapper.jsp с помощью тэга <jsp:include>:

```
<jsp:include page="ServletWrapper.jsp">
```

Этот оператор include будет действовать во всех средах JSP и сервлетов, в том числе в реализации Apache/JServ с Servlet 2.0.

Обработка ошибок JSP

В этом разделе говорится об использовании страниц ошибок JSP для обработки исключительных ситуаций, возникающих во время исполнения JSP. Поскольку нередко в скриптлетах JSP-страниц присутствуют программные конструкции Java, можно написать блок try...catch для перехвата исключительных ситуаций и вывода соответствующих сообщений об ошибках. Кроме того, JSP позволяет создавать модульные страницы ошибок, специально предназначенные для форматирования исключительных ситуаций и для создания отчетов о них. Для этого используются атрибуты errorPage и isErrorPage директивы *page*. Синтаксис этих атрибутов рассматривался выше в разделе "Директивы JSP". Теперь посмотрим их в действии.

Ниже описывается JSP-страница ErrorGenerator.jsp, генерирующая сообщение об ошибке, если в HTTP-объекте request отсутствует параметр username. Подобная ситуация может возникнуть, к примеру, на странице входа в систему, где содержится форма для ввода имени и пароля пользователя. Рекомендательное сообщение об ошибке необходимо на тот случай, если такая информация не указана.

Элементы программы ErrorGenerator.jsp, имеющие отношение к обработке ошибок, выделены жирным шрифтом. Важной характеристикой предлагаемой JSP является возможность использования директивы page для указания страницы ErrorHandler.jsp. HTTP-запрос направляется на эту страницу ошибок всякий раз, когда на главной JSP-странице возникает ошибка. Одну и ту же страницу обработки ошибок можно применять для нескольких JSP-страниц.

```
<%-- Имя программы:      ErrorGenerator.jsp
  -- Назначение:         Генерация ошибки для иллюстрации использования
  --                     страницы ошибок.
  --%>
<%@ page errorPage="ErrorHandler.jsp" %>
<HTML>
  <HEAD> <TITLE> The ErrorGenerator JSP </TITLE>
  </HEAD>
  <BODY BGCOLOR="white">
    <% //Проверим, указан ли username, если нет - установим
       // исключительную ситуацию
       String user = request.getParameter("username");
       if (user == null) {
         throw new Exception("You gave no user name!!"); }
       else %>
    Hello <%= user %>!
  </BODY>
</HTML>
```

Приведем программный текст страницы ErrorHandler.jsp, которая форматирует сообщение об ошибке и передает его:

```
<%-- Имя программы:  ErrorHandler.jsp
  -- Назначение:     Вывести сообщения об ошибках в удобном для пользователя виде.
  --%>
<%@ page isErrorPage="true" %>
<HTML>
```

```
<HEAD><TITLE> The ErrorHandler JSP </TITLE></HEAD>
<BODY BGCOLOR="white">
  <P><B> Hello! I am the ErrorHandler JSP.
  <P>    Sorry, there was a problem:
              <%= exception.getLocalizedMessage() %> </B>
</BODY>
</HTML>
```

Программа ErrorHandler.jsp объявлена как страница обработки ошибок: атрибут isErrorPage директивы page установлен в значение true. Напомним, что в результате такой установки объект exception будет неявно доступен на странице. Программа ErrorHandler.jsp обращается к этому объекту, извлекая и распечатывая сообщение об ошибке, сгенерированное главной JSP-страницей. Для этого вызывается метод getLocalizedMessage() объекта exception. Если характеристики локальной среды не имеют особого значения при составлении отчетов об ошибках, можно вызвать метод getMessage(). На рис. 10.9 показан результат вызова страницы ErrorGenerator.jsp без параметра username, когда устанавливается исключительная ситуация, о которой сообщает страница ErrorHandler.jsp.

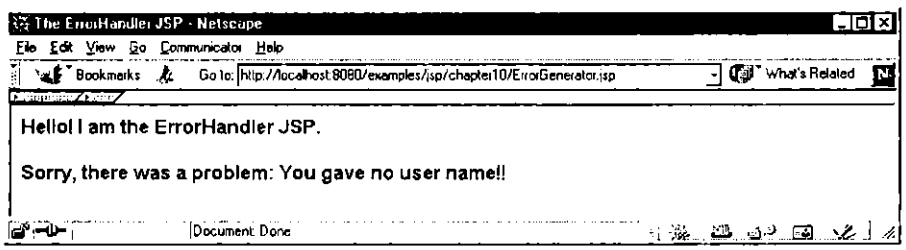


Рис. 10.9. Выходные данные после вызова ErrorGenerator.jsp без параметров

И последнее замечание о страницах ошибок. Будут ли доступны генерируемые исключительные ситуации еще и сервлетам, которые являются частью того же самого web-приложения? Да. К объекту exception можно обратиться не только при помощи неявной переменной exception на странице ошибок, но и через атрибут javax.servlet.jsp.jspException HTTP-объекта request:

```
<% Exception e = (Exception)
          request.getAttribute("javax.servlet.jsp.jspException"); %>
<P> Sorry, there was a problem: <%= e.getLocalizedMessage() %>
```

Для обращения к JSP-странице ошибок сервлет должен вызвать метод request.setAttribute() с генерируемой исключительной ситуацией, а затем передать обработку запроса JSP-странице ошибок с помощью метода forward() интерфейса RequestDispatcher (см. выше). Написать сервлет, передающий ошибки на JSP-страницу обработки ошибок подобным образом, предлагаем читателю.

JSP-реализация Oracle

В Oracle спецификации JSP 1.0 и 1.1 реализованы по-своему. Их можно бесплатно загрузить по сети Oracle Technology Network (technet.oracle.com/tech/java/servlets). Отличительной характеристикой двигателя JSP в Oracle является способность работать с разными версиями Servlet API (2.0, 2.1 и 2.2). Поэтому с JServ как с исполнителем сервлетов (Servlet 2.0 API) на web-сервере Apache и с Tomcat (Servlet 2.2) будут работать одни и те же библиотеки Oracle JSP. Такая независимость от версии Servlet API делает двигатель Oracle JSP переносимым в среде самых различных исполнителей сервлетов. Об установке Oracle JSP в средах популярных web-серверов рассказывается в приложении E.
В Oracle 8.1.7 сервлеты и JSP-страницы можно исполнять в границах виртуальной машины Java, встроенной в базу данных (JServer). При этом также используется JSP-двигатель Oracle, который выполняет в базе данных оптимизированный код JSP. Во время написания книги этот продукт находился в стадии разработки (ожидается, что он появится в начале осени 2000 г.).

JSP-реализация Oracle не только полностью соответствует спецификациям JSP, но и имеет ряд расширений, упрощающих процесс программирования. Из них главными считаются два:

- Поддержка SQLJ-конструкций в JSP-скриптлетах
- Инструмент-транслятор ojspc, позволяющий заранее преобразовывать файлы из командной строки

Использование SQLJ в JSP

SQLJ – это стандарт ANSI/ISO для встраивания статических SQL-операторов в программы Java при помощи специального обозначения #sql. Операторы #sql преобразуются в программные конструкции Java транслятором SQLJ. Скомпилированная программа выполняется в базе данных с помощью драйвера JDBC. SQLJ удобен тем, что краткость обозначений облегчает процесс программирования и снижает вероятность ошибок. Обычно программы SQLJ намного короче программ JDBC, и их написание и обслуживание занимают меньше времени. Кроме того, при использовании SQLJ можно проверять статические SQL-операторы в базе данных, фиксировать синтаксические ошибки SQL и несоответствие типов во время компиляции, а не исполнения программы (за дополнительными сведениями обращайтесь к книге "Oracle8*i* SQLJ Programming", OMH, ноябрь 1999 г.). Но если SQL- команда динамическая, т.е. перед исполнением статически не определена, придется применять вызовы JDBC.

Использование SQLJ-конструкций на JSP-странице задается одним из следующих способов:

- Путем указания для исходного файла расширения .sqljsp, вместо .jsp
- Путем указания language="sqlj" в директиве page на JSP (см. выше)

Хотя в JSP-реализации Oracle не возбраняется пользоваться SQLJ-конструкциями в JSP-скриптлетах, следует понимать, что лучше всего программно обращаться к базе данных при помощи компонентов JavaBean. SQLJ-конструкции можно ввести в JavaBean, сделав программу более модульной и многократно используемой. Кроме того, это удаляет с JSP громоздкую программную логику, смешанную со статическим текстом. О применении зерен JavaBean вместе с JSP подробно рассказывается в главе 11.

Ниже приводится программный текст страницы SQLJQuery.jsp, которая использует SQLJ для обращения к базе данных и отображает результаты статического SQLJ-запроса. Цель этой программы – по номеру служащего, указываемому в качестве параметра запроса, считать его полное имя, описание рабочих обязанностей и название отдела в базе данных заказов на покупку. Схема этой базы данных описана во введении. Для удобства воспроизведем нужные таблицы EMPLOYEE_LIST и DEPARTMENT_LIST:

```
CREATE TABLE EMPLOYEE_LIST(
  employeeno      NUMBER(7),
  departmentno NUMBER(5),
  type       VARCHAR2(30),
  lastname   VARCHAR2(30),
  firstname VARCHAR2(30),
  phone VARCHAR2(10));
CREATE TABLE DEPARTMENT_LIST(
  departmentno NUMBER(5),
  shortname VARCHAR2(6),
  longname VARCHAR2(20));
```

Предположим, что эти таблицы установлены в учетной записи jspuser с паролем jsp. Напишем программу для выбора нужной информации с помощью SQLJ:

```
<%-- Имя программы:  SQLJQuery.jsp
  -- Назначение:     Выбрать информацию в базе данных заказов на покупку.
```

```
  --%>
<%@ page language="sqlj" %>      <!" (см. пояснение 1) -->
<%@ page import="sqlj.runtime.ref.DefaultContext" %>
<%@ page import="oracle.sqlj.runtime.Oracle" %>
<HTML> <HEAD> <TITLE> The SQLJQuery JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">
   <H3> Hello, I can do a SQLJ query to get employee data...</H3>
   <%   String empno = request.getParameter("empno");    // (см. пояснение 2)
        if (empno != null) { %>
   <H2> Employee # <%=empno %> Details: </H2>
   <%   DefaultContext dctx = null;          <!-- (см. пояснение 3) -->
        try { // установим соединение с базой данных
          dctx = Oracle.getConnection("jdbc:oracle:oci8:@",
                                      "jspuser", "jsp"); // (см. пояснение 4)
          String job = null;
          String deptName = null;
          String empName = null;
          #sql [dctx] {  /* (см. пояснение 5) */
                         SELECT type, department_list.longname,
                                lastname||', '||firstname
                         INTO :job, :deptName, :empName
                         FROM EMPLOYEE_LIST, DEPARTMENT_LIST
                         WHERE employeeno= :empno
                         AND DEPARTMENT_LIST.deptno =
                         EMPLOYEE_LIST.deptno}; %>
        <H3><BLOCKQUOTE><PRE>
          Name          : <%= empName %>     <!-- (см. пояснение 6) -->
          Job           : <%= job %>
          Department    : <%= deptName %>
        </PRE></BLOCKQUOTE></H3>
    <%
        } catch (java.sql.SQLException e) { %>  <!-- (см. пояснение 7) -->
          <P> SQL error: <PRE> <%= e %> </PRE> </P>
        <%   } finally { if (dctx != null) dctx.close(); } } %>
    <HR><BR>
    <B>Enter an Employee Number:</B>
    <FORM METHOD=GET>   <!-- (см. пояснение 8) -->
        <INPUT TYPE="text" NAME="empno" SIZE=15 VALUE="111">
        <INPUT TYPE="submit" VALUE="Ask Oracle">
    </FORM>
  </BODY>
</HTML>
```

Пояснения к программе SQLJQuery.jsp:

1. Эта директива page объявляет об использовании SQLJ на JSP. В следующих двух строках также применяется директива page — для импорта необходимых классов SQLJ.
2. Этот скриптлет извлекает значение параметра с именем empno из объекта request. Если значение этого параметра не равно null, JSP исполняет SQL-запрос.
3. Этот оператор является частью главного JSP-скриптлета и описывает явный объект DefaultContext, представляющий SQLJ-соединение с базой данных. В SQLJ поддерживается понятие стандартного соединения (соединения по умолчанию) для исполнения SQL-конструкций, но его не следует применять в программировании с использованием JSP или сервлетов. Дело в том, что в исполняющей системе SQLJ соединение по умолчанию представляется в виде статической переменной, и одновременно выполняющиеся сервлеты и JSP-страницы, работающие в одной и той же виртуальной машине Java, будут перезаписывать эту статическую переменную, так как она одна. Выйти из этой ситуации можно, воспользовавшись для исполнения SQL-операторов явным объектом DefaultContext, как показано в данной программе.

4. Этот оператор Java инициализирует SQLJ-объект как соединение с базой данных. Обратите внимание на то, что для соединения с учетной записью jspuser/jsp используется драйвер JDBC-OCI.
5. Это статический SQL-запрос, написанный на SQLJ. В нем в качестве контекста соединения для исполнения запроса используется dctx. Параметр запроса empno в конструкции WHERE применяется для считывания информации о соответствующем служащем в таблицах EMPLOYEE_LIST и DEPARTMENT_LIST. Выбираемые данные помещаются в три переменные Java: job, deptName и empName, которые были объявлены ранее в этом скриптлете.
6. Этот HTML-оператор выводит значение переменной empName, присвоенное ей в предыдущем SQL-запросе. В следующих двух строках HTTP-текста отображаются соответственно рабочие обязанности и название отдела.
7. Этот оператор Java используется для перехвата всех исключительных ситуаций SQL, которые могут устанавливаться в программном тексте SQLJ. Блок catch распечатывает информацию об исключительной ситуации. За ним следует блок finally, закрывающий ранее открытый контекст соединения.
8. Этот оператор указывает на начало сегмента HTML-формы. Он применяется для получения номера служащего от пользователя по мере ввода данных на экране. Впоследствии этот параметр используется в SQL-запросе. Обратите внимание: в качестве номера служащего по умолчанию в форме указан номер 111.

На рис. 10.10 показан результат, выдаваемый браузером после вызова SQLJQuery.jsp без параметров.

Если ввести номер служащего в текстовом окне и щелкнуть мышью на кнопке Ask Oracle на экране, программа SQLJQuery.jsp будет вызвана с параметром запроса empno, установленным в значение 111. Это вызовет передачу SQL-запроса в базу данных, и браузер отобразит выходную страницу, представленную на рис. 10.11.

Использование транслятора командной строки ojspc

В двигателях JSP поддерживается автоматическое преобразование установленных JSP-страниц при их вызове. Это очень удобно при разработке приложений JSP. Однако в процессе внедрения приложений JSP нежелательно, чтобы JSP преобразовывалась при ее первом вызове конечным пользователем. Преобразование JSP отнимает значительное время и может замедлить реакцию приложения в сравнении с ранее скомпилированными страницами. Следовательно, после разработки приложения было бы полезно заранее транслировать JSP-страницы, чтобы избежать их преобразования на этапе исполнения. Именно для этого в JSP-реализации Oracle существует инструментальное средство трансляции ojspc, выполняемое из командной строки. Познакомимся с синтаксисом и параметрами этого инструмента.

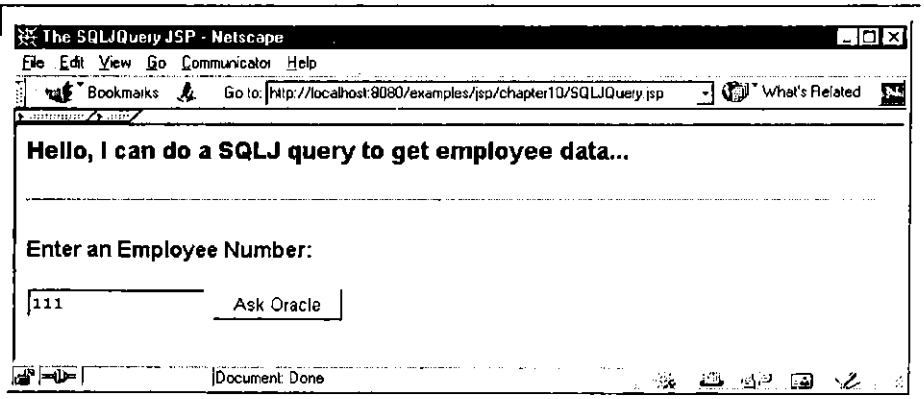


Рис. 10.10. Выходные данные после исполнения SQLJQuery.jsp без параметров

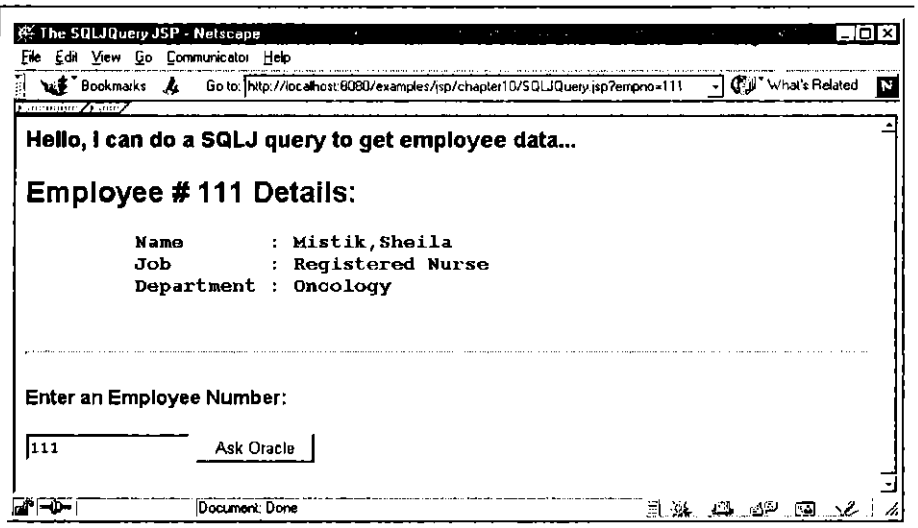


Рис. 10.11. Результат запроса данных о служащем 111 при помощи SQLJQuery.jsp

Команда ojspc – обычный сценарий оболочки UNIX (и командный файл в Windows), который вызывает Java-класс oracle.jsp.tools.Jspc. Этот класс является частью JSP-библиотек Oracle. Для применения данного инструмента нужно, чтобы в переменной среды CLASSPATH были указаны эти библиотеки, а также библиотеки сервлетов. Кроме того, при преобразовании с помощью ojspc файлов .sqljsp нужно убедиться в том, что JDBC-классы (classes111.zip или classes12.zip, что определяется использованием JDK 1.1 либо JDK 1.2) и SQLJ-библиотеки (translator.zip) доступны в CLASSPATH. Синтаксис вызова ojspc:

- `ojspc [`*параметры*`] `*список_файлов*

В средах оболочек UNIX разрешается указывать трафаретные символы в списке исходных файлов. Допустимыми расширениями исходных файлов являются .jsp, .sqljsp, .sqlj и .java. Файлы .sqljsp могут содержать в своем составе SQL-операторы в обозначении #sql. Для таких файлов транслятор JSP автоматически вызывает транслятор SQLJ. Во время SQLJ- и JSP-трансляции ссылки на типы определяются при помощи всех файлов SQLJ и Java, указанных в командной строке. Помимо этого, инструмент ojspc вызывает компилятор Java (по умолчанию javac), чтобы создать непосредственно байтовый код; автоматическая компиляция генерируемого текста отключается параметром -noCompile.

Для вывода перечня параметров ojspc применяется команда:

- `ojspc -help`

Основными параметрами являются:

- -addclasspath. Этот параметр задает дополнительные элементы CLASSPATH для Java-компиляции генерируемого текста. Если не указан, используется системная установка CLASSPATH.
- -appRoot. Этот параметр определяет корневой каталог приложения для файлов, подключаемых статически при помощи относительных для приложения указателей URL JSP-директивой include. В Oracle JSP версии 1.0 этот параметр назывался -includePath, в версии 1.1 он был переименован в -appRoot. Значением по умолчанию является текущий каталог.
- -d, -dir. Этот параметр определяет выходной каталог для генерируемого байтового кода. По умолчанию байтовый код помещается в текущий каталог.

- `-extres`. Этот флаг указывает на то, что для статического текста в JSP-источнике нужно генерировать файл внешнего ресурса. Флаг удобен в работе, если тело метода Java имеет длину около 64 Кбайт, что является системным ограничением (напомним, что в генерируемом тексте статический текст размещается в методе `_jspService()`). Кроме того, это ускоряет процесс Java-компиляции, так как постоянные строки символов помещаются в файл внешнего ресурса.
- `-noCompile`. Этот флаг указывает на то, что компилировать генерируемый текст не нужно. Впоследствии можно самостоятельно скомпилировать сгенерированные файлы, возможно, компилятором, отличным от стандартного `javac`.
- `-packageName`. Этот параметр указывает имя пакета для генерируемого класса. Обычно имя класса соответствует маршруту к странице, по которому JSP установлена на web-сервере. Таким образом, для JSP-страницы, установленной в подкаталоге `examples/jsp/chapter10` каталога документов web-сервера, во время предварительной трансляции с помощью `ojspc` нужно было бы указать имя пакета `examples.jsp.chapter10`. Именно так JSP-двигатель Oracle отображает маршруты к страницам в URL запроса на имена пакетов Java. Точной схемы такого отображения в спецификации JSP нет, и она определяется реализацией.
- *`-S-параметр_sqlj`*. Этот параметр применим только по отношению к JSP-страницам с программными конструкциями SQLJ и используется для указания параметров транслятора SQLJ. За префиксом `-S` должно следовать имя SQLJ-параметра, символ пробела, а затем значение параметра (если оно есть).
- `-srcdir`. Определяет каталог, где будут размещены генерируемые исходные файлы. Не путайте этот параметр с описанным выше параметром `-d`, который задает местонахождение генерируемого байтового кода.
- `-verbose`. Этот флаг используется для получения подробной информации об этапах трансляции.
- `-version`. Этот флаг отображает номер версии библиотеки Oracle JSP.

Приведем несколько примеров, иллюстрирующих использование инструмента `ojspc`. Заметим, что для отделения параметра от его значения (если оно есть) применяются пробелы. Строковые значения для таких параметров, как имена пакетов и маршруты к каталогам, следует заключать в двойные кавычки, и помните о регистре символов. Для имен параметров регистр символов не учитывается.

Для получения информации об этапах трансляции во время компиляции JSP-файла `Foo.jsp` введем следующее:

```
ojspc -verbose Foo.jsp
```

Для компиляции файлов `Bar.jsp` и `Query.sqljsp` введем:

```
ojspc Bar.jsp Query.sqljsp
```

Маршруты к исходным файлам можно указывать либо относительно текущего каталога, в котором вызывается `ojspc`, либо как абсолютные в файловой системе. По умолчанию выходным каталогом для генерируемых классов является текущий каталог, откуда вызывается `ojspc`. Местоположение генерируемых исходных файлов и компилируемого байтового кода можно изменить с помощью параметров `-d` и `-srcdir`:

```
ojspc -d _pages -srcdir /home/mywebpages *.jsp
```

Параметры транслятора SQLJ устанавливаются с помощью флага `-S`. Например, чтобы транслятор SQLJ считывал параметры из файла `sqlj.properties`, нужно ввести:

- `ojspc -S-props sqlj.properties Query.sqljsp`

Полный перечень параметров `ojspc` приведен в "Oracle JSP Developer Guide and Reference, Release 8.1.7".

Итоги

В этой главе описаны элементы сценариев и другие компоненты программирования на Java. Было показано, насколько прост процесс написания и исполнения JSP-программ. Двигатель JSP автоматически преобразует, загружает и выполняет JSP-страницы, позволяя быстро разрабатывать web-приложения. Объявления, скриптлеты и выражения JSP позволяют без труда вводить программные конструкции Java в JSP-страницы и динамически генерировать разделы web-страниц. JSP-директива `page` используется для передачи информации, связанной со страницей, например кодировки символов, транслятору JSP. Существуют специальные атрибуты `errorPage` и `isErrorpage` директивы `page`, применяемые для обработки ошибок этапа выполнения JSP. JSP-директива `include` служит для подключения содержимого других страниц статически на этапе компиляции. Кроме того, рассматривалось применение стандартных функциональных JSP-тэгов `<jsp:include>` и `<jsp:forward>`, позволяющих вызывать с JSP-страницы другие программы JSP и сервлеты. Было показано, как с помощью интерфейса `RequestDispatcher` вызывать JSP-страницы и другие web-ресурсы из сервлета.

В заключение были описаны JSP-реализация Oracle и ее свойства, которые дополняют функциональные возможности, полностью соответствующие спецификации JSP. В большинстве JSP-реализаций Java – единственный язык сценариев. В JSP-реализации Oracle в JSP-скриптлетах можно использовать и конструкции SQLJ, что упрощает процесс создания программ для работы с базами данных. Для предварительной трансляции JSP-страниц перед внедрением предлагается транслятор командной строки `ojspc`.

Теперь вы умеете создавать JSP-страницы с использованием элементов сценариев, объявлений, выражений и неявных объектов JSP, которые определяют содержимое страниц динамически, исходя из параметров запроса, вводимых посредством HTML-форм. Кроме того, вы должны понимать, каким образом в процессе трансляции и исполнения JSP генерируется выходная страница. В этой главе было показано, как создавать JSP-страницы, вызывающие другие JSP-страницы, HTML-страницы и сервлеты, а также сервлеты, вызывающие JSP-страницы. Была создана JSP, которая выполняет статический запрос, обращенный к базе данных Oracle, и освещен вопрос предварительной трансляции программных файлов JSP с помощью транслятора командной строки `ojspc`.

Овладев основами JSP-программирования, мы можем перейти к написанию web-страниц. В следующей главе показано, как с помощью компонентов JavaBean четко отделять в JSP-программах статический текст от динамического содержимого. Рассказывается о таких тэгах, связанных с зернами, как `<jsp:useBean>` и `<jsp:setProperty>`, которые являются стандартными тэгами JSP, облегчающими создание зерен JavaBean и работу с ними и с их свойствами. Приемы программирования JSP, о которых говорилось в настоящей главе, можно объединять с компонентами JavaBean, создавая многоцелевые web-приложения, компактные и модульные, простые в написании и обслуживании.

ГЛАВА 11

Использование JSP вместе с зернами JavaBean

В предыдущей главе были представлены базовый синтаксис и свойства JSP. В этой главе говорится о применении JSP-страниц в компонентном программировании совместно с зернами JavaBean. Вы узнаете об основных концепциях зерен JavaBean и о том, как грамотно использовать базу данных Oracle в работе с компонентами JavaBean. Кроме того, описывается процесс генерации XML-текста при помощи JSP-страниц и зерен JavaBean. Вы изучите, как:

- Применять компонент JavaBean с помощью соответствующих встроенных JSP-тэгов
- Разрабатывать сеансовое JSP-приложение, использующее "корзину" для ввода заказа
- Исполнять операции обновления базы данных, которые основаны на данных, вводимых пользователем в HTML-формах
- Исполнять статический SQL-запрос с помощью SQLJ для взаимодействия с базой данных
- Исполнять динамические SQL-запросы с помощью JDBC и форматировать результаты запроса в виде HTML-таблицы
- Генерировать XML-текст из результатов SQL-запроса
- Оптимизировать производительность доступа к базе данных с помощью пулинга соединений на уровне приложения

Разработка JSP-приложений с помощью компонентов JavaBean

Фундамент JSP был создан для работы с модульными и многократно используемыми программными компонентами, такими как зерна JavaBean. Применение зерен JavaBean в JSP позволяет четко отделять в HTML программные конструкции Java, генерирующие динамическое содержимое, от логики представления информации. Удобно то, что программные конструкции Java, вычисляющие динамическое содержимое, можно разместить в одном или в нескольких зернах JavaBean, которые "вставляются" в JSP для упрощения манипуляции данными. Такое разделение позволяет программисту Java создавать вычислительные логические схемы в компонентах JavaBean, а web-дизайнеру формировать вид представления информации. Впоследствии компоненты можно объединить надлежащим образом на JSP-странице.

В этом разделе рассказывается о том, как создавать простые компоненты JavaBean и применять их в JSP-программах, используя компактный и эффективный синтаксис.

Знакомство с JavaBean

JavaBean – это класс Java с определенной структурной схемой. Схема имеет хорошо описанную семантику, что позволяет определять структуру JavaBean через анализ внутренней организации его скомпилированного класса при помощи API *отражения* (reflection) Java. Главные требования, предъявляемые к JavaBean, таковы:

- Оно должно быть общим (`public`) классом Java.
- Оно должно содержать общий конструктор без аргументов.
- Оно может обладать одним или несколькими *свойствами* (properties), или атрибутами, работа с которыми организуется посредством общих методов *доступа* (accessor).

Свойства зерна реализуются обычно как частные (`private`) переменные Java или поля, а методы доступа используются для присвоения и для считывания их значений. По умолчанию методы доступа связываются с соответствующими

свойствами на основе простого соглашения по именованию: методами доступа для свойства, объявленного как `int x`, являются `getX()` и `setX(int newX)`. Такие правила проектирования обеспечивают простоту разработки приложений, а также динамическое определение свойств зерна во внешних средах, куда оно встраивается, например в графических инструментах разработки или как компонента в JSP.

Помимо свойств, в JavaBean, как и в любом другом классе Java, описываются еще и методы. Не забывайте о том, что JavaBean — это обычный класс Java с набором специальных характеристик.

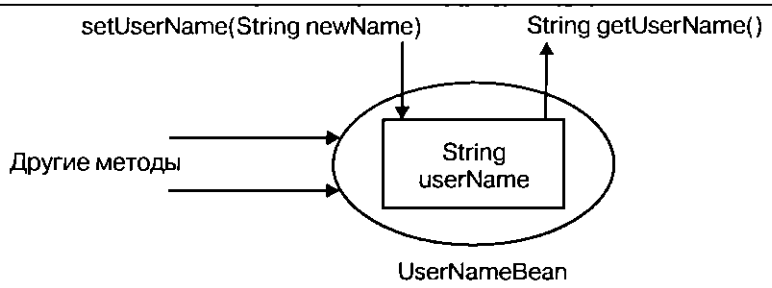
Еще одной важной характеристикой зерна является его способность реагировать на события (event) окружающей среды, для чего реализуются *интерфейсы приемника* (прослушивающего процесса) событий (event listener). Например, зерно, обращающееся к базе данных с запросом, требует, чтобы при завершении HTTP-сеанса базовый курсор закрывался. Зерно будет получать уведомление о событии завершения, если зарегистрируется в списке объектов, заинтересованных в данном событии (см. ниже раздел "События и интерфейсы приемников событий").

В редких случаях приходится использовать в зерне нестандартные имена для методов доступа или какую-то нестандартную структурную схему. Фундамент зерен JavaBean достаточно гибок для того, чтобы создать такое JavaBean, но после этого необходимо сформировать специальный класс *дескриптора* (descriptor) зерна, предоставив информацию о его структуре. Имя класса дескриптора образуется добавлением суффикса BeanInfo к имени класса зерна, причем класс дескриптора должен реализовывать интерфейс `java.beans.BeanInfo`, который содержит методы для получения необходимой информации об атрибутах, методах и событиях для JavaBean. За дополнительными сведениями обращайтесь к литературе по JavaBean, например к книге "Developing JavaBeans" Роберта Ингландера, издательство O'Reilly Press. В примерах этой книги соблюдаются стандартные правила именования `get` и `set` для методов доступа JavaBean.

Создание простого JavaBean

В качестве первого шага создадим простое зерно `UserNameBean`, которое обладает единственным свойством `userName` со строковым значением, представляющим имя пользователя. Методы доступа этого свойства описываются в соответствии со стандартным соглашением по именованию. Базовая структура этого зерна-компонента представлена на диаграмме (см. рис. 11.1).

Рис. 11.1. Структура компонента `UserNameBean`



Ниже приводится программный текст этого зерна — класс `UserNameBean` с частным полем `userName` и методами `getUserName()` и `setUserName(String newName)` для считывания и присвоения значения этого свойства. Здесь описан дополнительный метод `noUser()`, возвращающий true, если имя пользователя — `null` или пустая строка символов.

```
package mybeans;

public class UserNameBean {
  private String userName = "";
  public String getUserName() {
    return userName;
  }
```

```
  public void synchronized setUserName(String newName) {
    userName = newName;
  }
  public boolean noUser() {
    return ((userName == null) || (userName.equals("")));
  }
}
```

В этой программе класс UserNameBean помещен в пакет mybeans. Указывать имя пакета для JavaBean необязательно, как, впрочем, и для любого другого класса Java. Явного конструктора здесь нет, т.е. для класса используется конструктор по умолчанию. Напомним, что JavaBean — обычный класс Java с набором дополнительных правил, определяющих его структуру, поэтому при его написании соблюдаются обычные правила синтаксиса и семантики Java. Заметим также, что для свойства зерна, используемого в JSP, совсем не обязательно указывать оба метода get и set; какой из методов доступа может быть опущен, зависит от способа использования свойства. Обычно метод set защищается от многопоточного доступа объявлением его как синхронизированного (synchronized) метода.

Зерно компилируется так же, как любой другой класс Java. Для использования зерна в JSP-приложении класс зерна необходимо установить на web-сервере (см. приложение А).

Стандартные JSP-тэги для зерен JavaBean

После написания JavaBean его можно ввести в JSP при помощи стандартных тэгов. В спецификации JSP описаны три встроенных тэга, упрощающих работу с зернами:

- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>

Эти тэги применяются для интеграции на JSP-страницах логики Java зерен со статическим HTML-текстом. Теперь посмотрим, каковы синтаксис и семантика этих тэгов и как их грамотно использовать в JSP-приложениях.

Тэг <jsp:useBean>

Этот тэг указывает на то, что определенное зерно используется на JSP-странице. Общий формат тэга:

❐ `<jsp:useBean id="переменная" другие_атрибуты />`

К другим атрибутам, помимо прочих, относятся class и scope. Наиболее часто используются атрибуты id, scope и class.

Атрибут class описывает полностью определенное имя класса для JavaBean; в данном случае — mybeans.UserNameBean.

Атрибут id описывает идентификатор Java для экземпляра зерна. С помощью этого идентификатора можно ссылаться на экземпляр зерна в той области действия Java, в которой присутствует тэг <jsp:useBean>. Например, следующий оператор JSP-программы указывает на то, что она использует экземпляр класса зерна mybeans.UserNameBean:

❐ `<jsp:useBean id= "userBean" class="mybeans.UserNameBean" />`

Здесь идентификатор userBean определяет имя экземпляра зерна. Согласно правилам области действия переменных в Java, можно написать выражение или скриптлет Java, где фигурирует данный идентификатор, например:

❐ `<%= userBean.getUserName() %>`

Атрибут scope необязателен, но довольно важен. Он определяет *время жизни* (lifetime) компонента-зерна, используемого на JSP-странице, т.е. продолжительность его существования в качестве программной переменной. Этот атрибут может иметь одно из четырех значений:

- page
- request
- session
- application

Значение по умолчанию для scope — page. Рассмотрим подробнее каждое из этих четырех значений:

- page. Во время исполнения оператора <jsp:useBean> JSP-контейнер просматривает объект page в поисках экземпляра зерна с таким же идентификатором. Если такого зерна не обнаруживается, создается новый экземпляр зерна, который связывается с объектом page при помощи идентификатора, указанного атрибутом id. В этом правиле поиска предполагается, что обращаться к зерну с областью действия page на других страницах, вызываемых с данной JSP, нельзя; это относится, например, к страницам, вызываемым посредством тэгов <jsp:include> и <jsp: forward> (см. главу 10). При повторном вызове этой JSP тот же самый идентификатор будет ссылаться на совершенно иной экземпляр зерна.
- request. Эта область действия похожа на область действия page с тем важным отличием, что экземпляр зерна связывается с HTTP-объектом request. Следовательно, к зерну с областью действия request можно обращаться и на другой странице, вызываемой (подключаемой или новой управляющей) с текущей JSP. Идентификатор зерна с областью действия request должен быть уникален среди всех страниц, обрабатывающих один и тот же HTTP-запрос.
- session. В этом случае экземпляр зерна связывается с объектом session и существует во время HTTP-сеанса, в котором вызывается данная JSP. При исполнении тэга <jsp:useBean> в объекте session создается экземпляр зерна, если, конечно, его там еще нет. Идентификатор зерна с областью действия session должен быть уникален среди всех страниц, вызываемых в границах одного и того же HTTP-сеанса. Зерна с областью действия session часто используются в приложениях с контролем за сеансами, например в таких, где реализуется функция "корзины магазина". Пример подобного зерна приводится ниже.
- application. Эта область действия похожа на другие, но связывает экземпляр зерна с объектом application (см. главу 10). При ее использовании экземпляр зерна становится доступен всем страницам, которые исполняются в одной и той же виртуальной машине Java и являются частью одного и того же web-приложения, в качестве текущей JSP.

Помимо атрибутов, у тэга <jsp:useBean> может быть и тело, содержащее другие JSP-тэги и скриптлеты для работы с экземпляром зерна. Например:

```
<jsp:useBean id="userBean" class="mybeans.UserNameBean" >
    Initial value of user name is: <%= userBean.getUserName() %>
</jsp:useBean>
```

Тэг <jsp:setProperty>

Стандартный тэг <jsp:setProperty> применяется для установки свойств компонента JavaBean, используемого на JSP-странице. Общий формат этого тэга:

```
<jsp:setProperty name="имя_зерна" выражение_для_свойств />
```

где *имя_зерна* — идентификатор экземпляра JavaBean, описываемый обычно в тэге <jsp:useBean> выше на JSP. Конструкция выражение_*для_свойств* может записываться в одном из следующих четырех форматов:

- property="*имя_свойства*". В этом формате указывается имя одного свойства JavaBean, а его значение устанавливается на основании значения соответствующего параметра HTTP-объекта request. Другими словами,

метод set этого свойства будет вызываться со значением соответствующего параметра запроса в качества аргумента. Например, чтобы установить свойство userName ранее описанного UserNameBean на основании соответствующего параметра HTTP-запроса, нужно ввести:

```
<jsp:setProperty name="userBean" property="userName" />
```

- property="*". Это сокращенное выражение говорит о том, что свойства JavaBean будут устанавливаться на основании значений соответствующих параметров запроса путем сопоставления их имен. Другими словами, анализируется каждый параметр HTTP-объекта request, и при наличии соответствующего свойства зерна его метод set вызывается со значением параметра запроса в качестве аргумента.
- property="*имя_свойства*" *param="имя_параметра"*. Этот формат используется для указания пары свойство-параметр запроса. Свойству зерна присваивается значение указанного параметра запроса вызовом соответствующего метода set со значением параметра в качестве аргумента. Например, если в HTTP-объекте request для представления имени пользователя применяется другой идентификатор, скажем, newName, то для присвоения значения свойству userName зерна значения параметра нужно указать:

```
<jsp:setProperty name="userBean" property="userName"
                                        parameter="newName" />
```

- property="имя_свойства" *value="новое_значение"*. Этот формат используется для присвоения указанного значения данному свойству зерна. *Новое_значение* может быть постоянной строкой символов или JSP-выражением, вычисляемым во время запроса.

Тэг <jsp:getProperty>

Этот оператор выводит значение указанного свойства JavaBean. Результатом его является вызов метода get свойства, преобразование выданного значения в строку символов и запись ее в объект out (представляющий поток ответа). Общий формат этого тэга:

❒ `<jsp:getProperty name="имя_зерна" property="имя_свойства" />`

Здесь атрибут name ссылается на идентификатор зерна, ранее определенный в тэге <jsp:useBean> в границах JSP, а атрибут property указывает свойство этого зерна. Например, для отображения свойства userName зерна UserNameBean нужно ввести:

❒ `<jsp:getProperty name="userBean" property="userName" />`

Возникает вопрос: а нужен ли вообще тэг <jsp:getProperty>? Ведь, по всей видимости, такой же эффект будет иметь выполнение эквивалентного JSP-выражения. Например, можно переписать предыдущий оператор, и результат будет таким же:

❒ `<%= userBean.getUserName() %>`

В чем же отличие этих двух операторов? В некоторых случаях использование тэга <jsp:getProperty> является более гибким способом получения значения свойства. Так, иногда для методов доступа зерна применяются нестандартные имена (как указано в классе дескриптора BeanInfo), которые неизвестны JSP-программисту. Настоящее имя метода доступа, соответствующего указанному свойству, выясняется во время JSP-трансляции при внутреннем анализе класса зерна и соответствующего класса дескриптора зерна. В такой ситуации удобно применять тэг <jsp:getProperty>. Кроме того, можно изменять имена методов доступа без модификации JSP-программы.

Индексируемые свойства

Имя индексируемого свойства связывается с несколькими значениями. Индексируемое свойство объявляется как массив, например:

```
private String[] phones;
```

В спецификации JavaBean описывается стандартная структурная схема для методов доступа индексируемых свойств. Методы доступа могут обладать дополнительным атрибутом — целочисленным индексом массива, что позволяет обращаться к отдельным значениям массива. Методы доступа set и get могут иметь и другой вид и применяться ко всему массиву. Например, следующие сигнатуры методов — это стандартная схема доступа для объявленного выше свойства phones:

```
public String[] getPhones();
public void setPhones(String[] newPhones);
public String getPhones(int index);
public void setPhones(int index, String newPhones);
```

В спецификации JSP 1.1 поддерживаются индексируемые свойства в стандартном тэге <jsp:setProperty>. В качестве примера предположим, что индексируемое свойство phones объявлено как часть экземпляра зерна addrBean. Для установки и получения значений этого индексируемого свойства на JSP можно воспользоваться следующими операторами:

```
<% String[] thePhones = new String[]{"123-4567", "456-7890"}; %>
<jsp:setProperty name="addrBean" property="phones"
                 value="<%= thePhones %>" />
<%=addrBean.getPhones(0)%>,<%=addrBean.getPhones(1)%>
```

Создание HelloBean.jsp с помощью UserNameBean

Теперь создадим JSP, на которой используется описанный выше компонент UserNameBean. Эта JSP с именем HelloBean.jsp по существу выполняет ту же самую функцию, что и программа Hello.jsp из главы 10, с тем отличием, что в HelloBean.jsp используются JavaBean и стандартные JSP-тэги зерен, а не программные конструкции Java в JSP-скриптлетах. Приведем текст этой JSP :

```
<!-- Имя программы:  HelloBean.  p
  -- Назначение:     Сохранить   считать имя пользователя при помощи
  --                 компонента UserNameBean и распечатать приветствие.
-->
<!-- (см. пояснение 1) -->
<jsp:useBean id="nameBean" class="mybeans.UserNameBean"/>
<!-- (см. пояснение 2) ->
<jsp:setProperty name="nameBean" property="*" />

<HTML>
  <HEAD>
    <TITLE> The HelloBean JSP </TITLE>
  </HEAD>
  <BODY  BGCOLOR="white">
                <!--- (см. пояснение 3) -->
    <H3> Hello  <%= nameBean.getUserName() %> ! </H3>
    <P><B> The current time is <%= new java.util.Date() %>.
          Have a nice day! </B></P>
    <B> Please type in your user name: </B>
      <!-- (см. пояснение 4) -->
      <FORM METHOD=GET>
        <INPUT TYPE="text" NAME="username" SIZE=15>
        <INPUT TYPE="submit" VALUE="Enter name");
      </FORM>
  </BODY>
</HTML>
```

Пояснения к `HelloBean.jsp`:

1. Тэг `<jsp:useBean>` объявляет, что JSP использует экземпляр класса зерна `mybeans.UserNameBean`, и связывает его с идентификатором `nameBean`. Поскольку область действия не указана, предполагается, что применяется область действия по умолчанию — `page`.
2. Этот оператор устанавливает для свойства `userName` экземпляра `nameBean` значение соответствующего параметра запроса. Заметим, что для описания свойства зерна, соответствующего параметру HTTP-запроса, применяется сокращение "*".
3. В приветствии "Hello" значение свойства `userName` считывается из зерна с помощью метода доступа `getUserName()`.
4. В последней части JSP указана HTML-форма для ввода имени пользователем, после чего вызывается та же JSP (действие формы по умолчанию). Заметим, что в текстовом поле ввода формы указано имя `userName`. Это имя намеренно совпадает с именем свойства зерна, "связывая" параметры HTTP-запроса с атрибутами зерна. Другими словами, значение, вводимое в этом текстовом поле, становится частью HTTP-запроса, который затем используется для установки соответствующего свойства зерна посредством тэга `<jsp:setProperty>`.

Выполнение HelloBean.jsp

Сначала установим исходный файл `HelloBean.jsp` на web-сервере, следуя инструкциям, приведенным в приложении E. Кроме того, необходимо скомпилировать класс `UserNameBean` и установить его на web-сервере. В приложении E эти операции описаны для разных web-серверов. Например, для сервера Tomcat, функционирующего в системе UNIX, можно воспользоваться сценарием оболочки, представленным в листинге 11.1, и скомпилировать класс зерна, поместив его в подкаталог `webapps/examples/WEB-INF/classes` корневого каталога Tomcat.

Листинг 11.1. Сценарий оболочки `buildbeans.sh` **для компиляции класса** `UserNameBean`

```
#!/bin/sh
# buildbeans.sh
TOMCAT_HOME=/private/jakarta-tomcat
JDK_HOME=/usr/local/packages/jdk1.2.2
APP_HOME=${TOMCAT_HOME}/webapps/examples
exec ${JDK_HOME}/bin/javac -d ${APP_HOME}/WEB-INF/classes \
     ${APP_HOME}/jsp/chapter11/mybeans/*.java
```

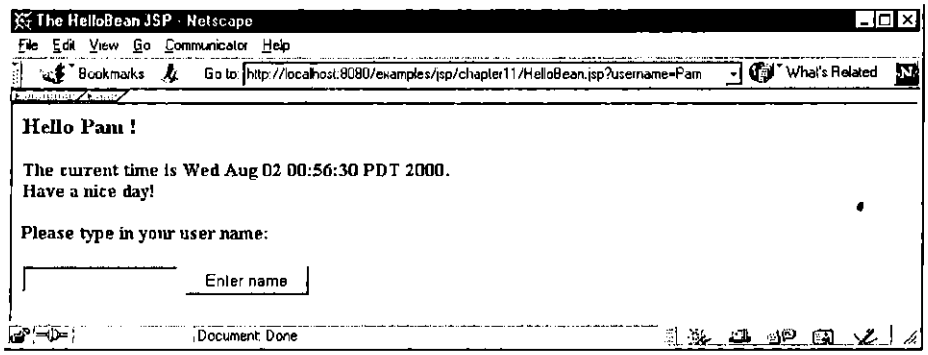
После компиляции зерна можно выполнить JSP при помощи браузера. При вызове страницы без параметров отображается приветственное сообщение без имени пользователя, текущего времени и формы для ввода имени пользователя.

Теперь допустим, что мы ввели в текстовом окне экрана имя **Pam** и щелкнули мышью на кнопке "Enter name". В результате HTTP-запрос посылается на ту же JSP-страницу (как стандартное действие HTTP-формы), а для свойства `userName` экземпляра `UserNameBean`, связанного с JSP- страницей (это стандартная область действия экземпляра зерна), устанавливается указанное значение. Впоследствии это значение можно считать и отобразить с помощью метода `nameBean.getUserName()`. Результат, получаемый в браузере, приведен на рис. 11.2.

Использование зерен JavaBean с разными областями действия

Теперь создадим JSP-приложение, использующее компоненты JavaBean с разными областями действия. В примере демонстрируется не только применение

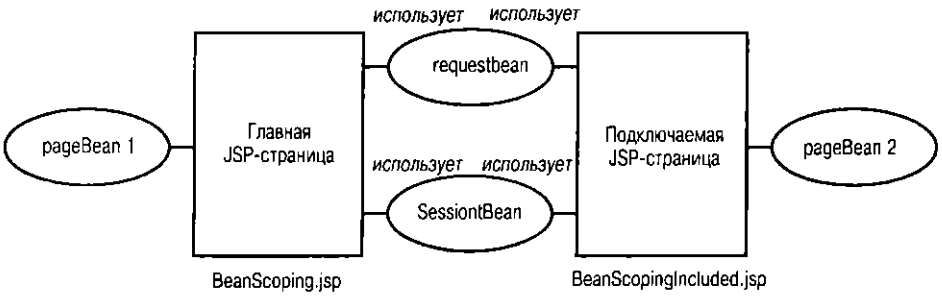
Рис. 11.2. Выходные данные HelloBean.jsp после ввода значения Pam



модульных зерен и их свойств, но и изменение продолжительности их существования путем изменения области действия. Приложение состоит из двух JSP-страниц: BeanScoping.jsp и BeanScopingIncluded.jsp. Вторая JSP вызывается с первой с помощью тэга <jsp:include> (см. главу 10), поэтому обе они исполняются как часть одного HTTP-запроса. Цель примера — показать, что к экземплярам зерна с областями действия request и session можно обращаться с подключаемой (или новой управляющей) JSP-страницы, в то время как зерно с областью действия page локально для JSP-страницы.

Структура этого приложения представлена на рис. 11.3. Она включает в себя две JSP-страницы и используемые ими зерна-компоненты. Зерно запроса и сеансовое зерно совместно используются двумя страницами (посредством соответственно HTTP-запроса и HTTP-сеанса), в то время как экземпляры страничного зерна для каждой JSP-страницы свои.

Рис. 11.3. Компоненты приложения BeanScoping



Ниже следует текст программы BeanScoping.jsp, в которой используются три экземпляра одного класса UserNameBean с областями действия page, request и session. JSP можно разделить на четыре части:

- В первой части создаются и устанавливаются свойства трех экземпляров зерна.
- Во второй части значения свойств экземпляров зерна считываются и распечатываются.
- В третьей части подключается другая JSP как элемент того же самого HTTP-запроса.
- В четвертой, заключительной, части содержатся формы для ввода данных, где указывается значение свойства userName зерен.

```
<!-- Имя программы:  BeanScoping.jsp
  -- Назначение:     Применить компоненты JavaBean с областями
  --                 действия page, request и session.
  --!>
<!-- (см. пояснение 1) -->
<!-- Воспользуемся экземпляром зерна на текущей странице -->
<Jsp:useBean  id="pageBean"
              class="mybeans.UserNameBean"
```

```
                scope="page" />
<jsp:setProperty name="pageBean"
                 property="*" />

<!-- Воспользуемся экземпляром зерна в текущем запросе -->
<Jsp:useBean  id="requestBean"
              class="mybeans.UserNameBean"
              scope="request" />
<jsp:setProperty name="requestBean"
                 property="*" />

<!-- Воспользуемся экземпляром зерна в текущем HTTP-сеансе -->
<Jsp:useBean  id="sessionBean"
              class="mybeans.UserNameBean"
              scope="session" />
<jsp:setProperty name= "sessionBean"
                 property="*" />

<HTML>
<HEAD> <TITLE> The BeanScoping JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<H2> Welcome to the BeanScoping JSP! </H2>
<B> Printing bean properties...
<!-- (см. пояснение 2) -->
<% if (pageBean.noUser())
    {  /* имя пользователя не установлено в зерне с областью действия page */ %>
      <BR> No user name in page-scoped bean!
<% } else {
     /* извлечем свойство имени пользователя из pageBean */ %>
      <BR> User name in pageBean: <%= pageBean.getUserName() %>
<%  } %>

<% if (requestBean.noUser())
    {  /* имя пользователя не установлено в зерне с областью действия request */ %>
      <BR> No user name in request-scoped bean!
<% } else {
      /* извлечем свойство имени пользователя из requestBean */ %>
      <BR> User name in requestBean: <%= requestBean.getUserName()%>
<% } %>

<% if (sessionBean.noUser())
    {  /* имя пользователя не установлено в зерне с областью действия session */ %>
      <BR> No user name in session-scoped bean!
<% } else {
      /* извлечем свойство имени пользователя из sessionBean */ %>
      <BR> User name in sessionBean: <%=sessionBean.getUserName()%>
<% } %>

<P>Now printing output from an included page..
<BR><HR>
<!--  (см. пояснение 3) -->
  <jsp:include page="BeanScopingIncluded.jsp" flush="true" />
<BR><BR><HR><BR>
<!--  (см. пояснение 4) -->
<P><H3>Please enter your user name:</H3>
  <FORM METHOD="GET">
    <INPUT TYPE=TEXT name="userName" size = 20>
    <INPUT TYPE=SUBMIT VALUE="Submit new user name">
  </FORM>
  <FORM METHOD="GET">
    <INPUT TYPE=SUBMIT VALUE="Use previous name"> <BR><BR>
  </FORM>
 </BODY>
</HTML>
```

Пояснения к BeanScoping.jsp:

1. В первой части JSP содержатся три тэга <jsp:useBean>, использующих разные экземпляры одного класса зерна с разными областями действия: page, request и session. За каждым из трех операторов <jsp:useBean> следует тэг <jsp:setProperty>, устанавливающий свойства экземпляра зерна на основании соответствующего параметра HTTP-запроса (userName).
2. В этой части JSP считывается и отображается значение свойства каждого зерна. Если значением является null или пустая строка, что определяет вызывающий метод noUser() зерен, выводится сообщение об этом. В противном случае отображается значение свойства, извлеченное вызывающим методом доступа getUserName().
3. Этот оператор подключает JSP-страницу BeanScopingIncluded.jsp (см. ниже).
4. В последней части JSP содержатся две HTML-формы:
   - Сегмент первой формы позволяет ввести имя пользователя, которое становится параметром HTTP-запроса GET после щелчка мыши на кнопке "Submit new user name". Вовсе не случайно, что имя текстового поля ввода совпадает с атрибутом userName класса UserNameBean; совпадение имен – это механизм, "связывающий" параметр запроса со свойствами зерна на этой JSP.
   - Функция передачи второй HTML-формы вызывает ту же самую JSP, но с параметром userName. Задача этой функции – считать и повторно использовать значение userName, сохраненное в зерне с областью действия session из предыдущего вызова JSP.

Ниже приводится текст программы BeanScopingIncluded.jsp, которая вызывается главной JSP-страницей BeanScoping.jsp. Страница BeanScopingIncluded.jsp похожа на BeanScoping.jsp; главное их отличие в том, что подключаемая JSP-страница не устанавливает никаких свойств зерна посредством тэгов <jsp:setProperty> и не содержит никаких HTML-форм для ввода данных. Она просто использует три соответствующим образом названных экземпляра с областями действия page, request и session так, что зерна с областями действия request и session разделяются между данной страницей и главной страницей BeanScoping.jsp. Как показано на рис. 11.3, экземпляр зерна с областью действия page – это собственный локальный компонент страницы, который не разделяется между страницами.

```
<!-- Имя программы:      BeanScopingIncluded.jsp
  -- Назначение:         Применить зерна, созданные на главной странице
  --                     BeanScoping.jsp.
  --!>

<!-- Воспользуемся экземпляром зерна на текущей странице -->
<jsp:useBean id="pageBean"
class="mybeans.UserNameBean"
scope="page" />

<!-- Воспользуемся экземпляром зерна в текущем запросе -->
<jsp:useBean id="requestBean"
class="mybeans.UserNameBean"
scope="request" />

<!-- Воспользуемся экземпляром зерна в текущем HTTP-сеансе -->
<jsp:useBean id="sessionBean"
class="mybeans.UserNameBean"
scope="session" />

<% if (pageBean.noUser())
    { /* имя пользователя не установлено в зерне с областью действия page */ %>
      <BR> BeanScopingIncluded.jsp: No user name in page-scoped bean!
```

```
<% } else {
      /* извлечем свойство имени пользователя из pageBean */ %>
      <BR> BeanScopingIncluded.jsp: User name in pageBean:
                        <%= pageBean.getUserName() %>
<% } %>

<% if (requestBean.noUser())
    {  /* имя пользователя не установлено в зерне с областью действия request */ %>
      <BR> BeanScopingIncluded.jsp: No user name in
                                    request-scoped bean!
<% } else {
      /* извлечем свойство имени пользователя из requestBean */ %>
      <BR> BeanScopingIncluded.jsp: User name in requestBean:
                        <%= requestBean.getUserName()%>
<% } %>

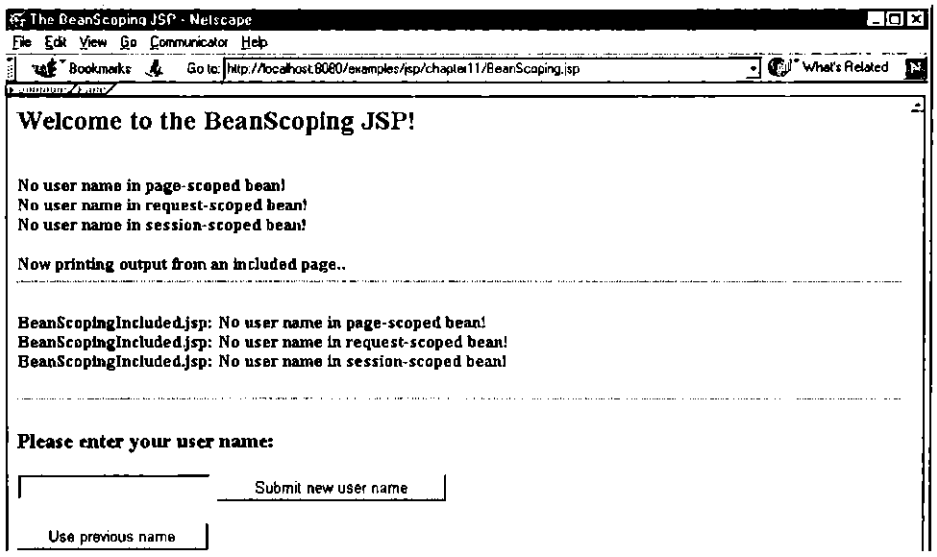
<% if (sessionBean.noUser())
    {  /* имя пользователя не установлено в зерне с областью действия session */ %>
      <BR> BeanScopingIncluded.jsp: No user name in
                                    session-scoped bean!
<% } else {
      /* извлечем свойство имени пользователя из sessionBean */ %>
      <BR> BeanScopingIncluded.jsp: User name in sessionBean:
                          <%=sessionBean.getUserName()%>
<% } %>
```

Выполнение JSP-страницы BeanScoping

Посмотрим, как работает JSP-страница BeanScoping. На рис. 11.4 показан результат, выдаваемый браузером при вызове данной JSP в первый раз и без параметров запроса. В этом случае тэги <jsp:useBean> создают новые экземпляры трех зерен с надлежащими областями действия. Однако соответствующего HTTP-параметра для их свойства userName нет, поэтому первоначально методы noUser() возвращают true. Таким образом, JSP распечатывает сообщения "no user name".

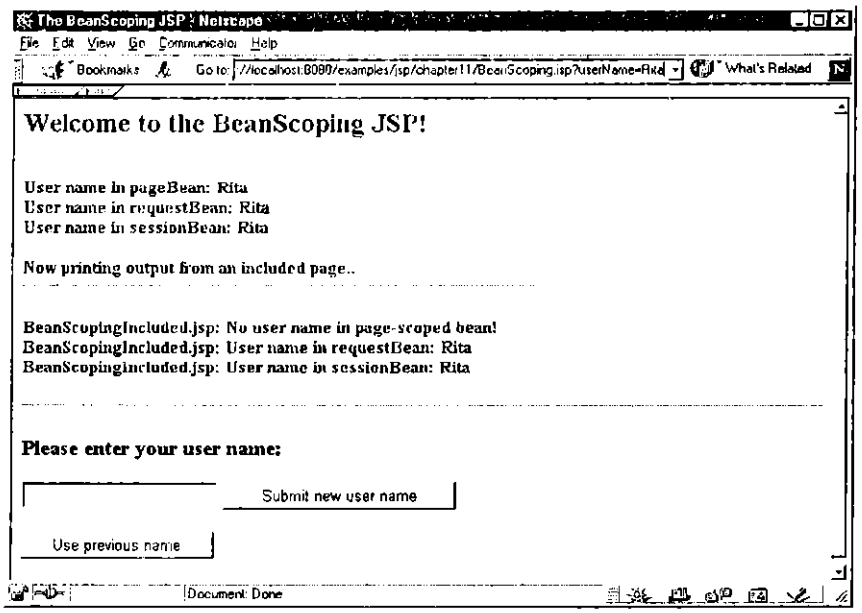
Теперь предположим, что мы ввели в HTML-форме имя **Rita** и щелкнули мышью на кнопке "Submit new user name". Будет сформирован новый HTTP-запрос и созданы новые экземпляры страничного зерна и зерна запроса на главной странице BeanScoping.jsp. Если срок действия HTTP-сеанса не истек, используется тот же самый экземпляр сеансового зерна. Для свойства userName новых экземпляров страничного зерна, зерна запроса и того же самого сеансового зерна устанавливается входное значение Rita. Затем все

Рис. 11.4. Начальные выходные данные BeanScoping.jsp



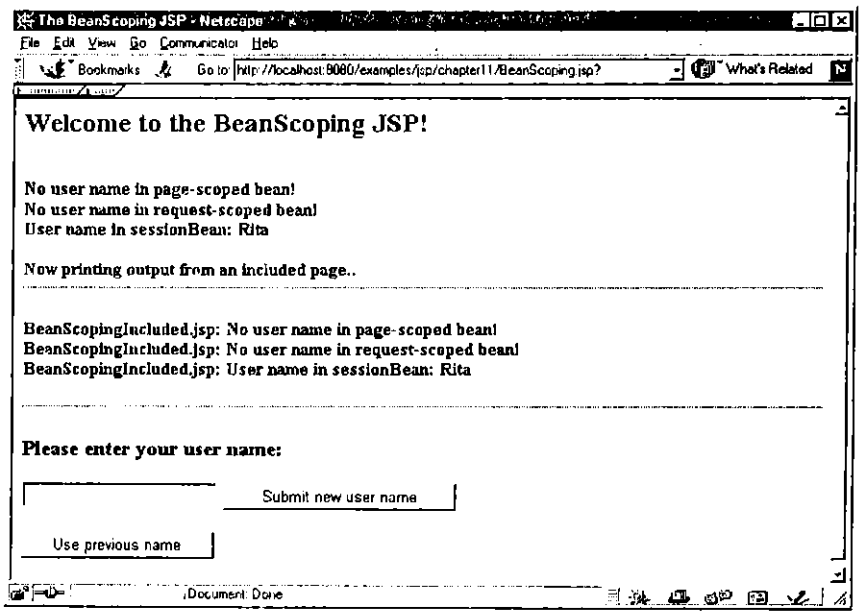
зерна главной страницы распечатывают имя пользователя, как показано в первой части рис. 11.5. В следующей части рисунка изображены выходные данные, генерируемые подключаемой страницей `BeanScopingIncluded.jsp`. Эта страница не устанавливает никаких свойств зерен, поэтому в этом страничном зерне имя пользователя отсутствует. Свойства же зерна запроса и сеансового зерна устанавливаются главной страницей, и значение имени пользователя надлежащим образом считывается в разделяемых экземплярах зерна и распечатывается подключаемой страницей.

Рис. 11.5. Выходные данные `BeanScoping.jsp` после ввода имени пользователя Rita



Чтобы увидеть, каким образом сеансовое зерно сохраняет установки своих свойств во время вызова, нужно еще раз обратиться к той же JSP без параметров запроса, щелкнув мышью на кнопке "Use previous name" (использовать предыдущее имя) до истечения срока действия HTTP-сеанса. Как показано на рис. 11.6, зерно с областью действия `session` сохранило значение своего свойства `userName` из предыдущего вызова, а в новых экземплярах страничного зерна и зерна запроса имя пользователя не установлено.

Рис. 11.6. Выходные данные `BeanScoping.jsp` после повторного вызова без параметров



Использование зерен JavaBean с областью действия application аналогично приведенному выше примеру. О кэшировании соединений с базой данных с помощью зерна с областью действия application рассказывается ниже в разделе "Пулинг соединений".

Разработка JSP-приложения с контролем за сеансами

Теперь рассмотрим более сложный пример применения JavaBean в JSP, когда действия пользователя отслеживаются в конкретном HTTP-сеансе. Этот пример написан по образцу распространенного web-приложения – интерактивной системы ввода заказов. Предположим, что наша прикладная программа должна позволять:

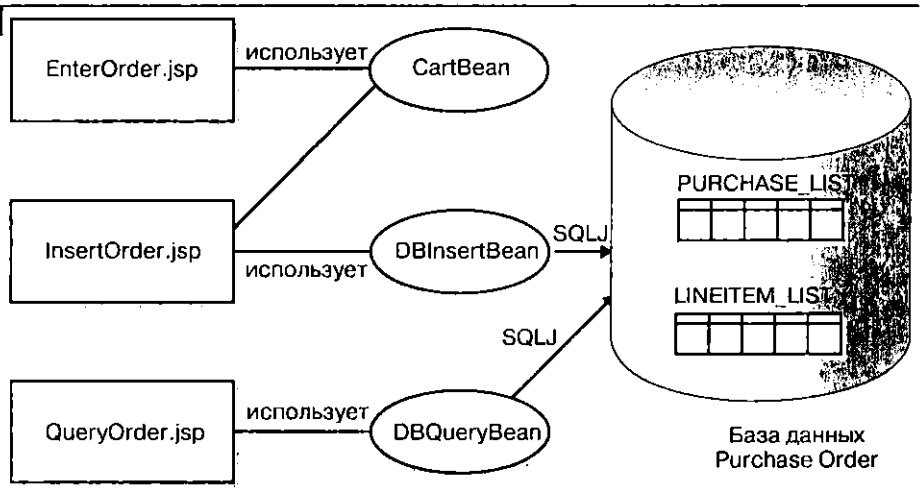
- Выбирать один или несколько товаров (пунктов ассортимента) в группе.
- Помещать выбранные товары в корзину.
- Добавлять и удалять товары из корзины, а также очищать корзину по мере выбора.
- В конце процесса выбора товаров передавать заказ системе. При этом информация о заказе будет вводиться в таблицы базы данных.
- Запрашивать характеристики заказа в базе данных.

Рассмотрим этапы процесса разработки этого приложения с помощью JSP-страниц и зерен JavaBean.

Архитектура интерактивного приложения заказов

Первый шаг – создание проекта компонентов JSP-приложения. Модульная архитектура интерактивного JSP-приложения заказов (Order Online) представлена на рис. 11.7. Три JSP-страницы – EnterOrder.jsp, InsertOrder.jsp и QueryOrder.jsp – будут служить пользовательским интерфейсом для интерактивных процессов выбора товаров, передачи заказов и изучения заказов соответственно. Корзина представлена компонентом cartBean. В нем будет содержаться текущий список выбранных товаров, и его будут совместно использовать страницы EnterOrder.jsp и InsertOrder.jsp. Страница InsertOrder.jsp имеет дополнительный компонент DBInsertBean, который будет исполнять SQL-операцию INSERT для записи переданного заказа в базу данных заказов на покупку. У страницы QueryOrder.jsp будет свой собственный компонент JavaBean – DBQueryBean. Этот JavaBean будет обращаться с запросами к базе данных заказов на покупку и считывать информацию о заказах.

Рис. 11.7. Архитектура приложения Order Online



Создание класса CartBean

На первом этапе программирования опишем зерно JavaBean с именем CartBean, чтобы смоделировать корзину. Оно будет управлять списком товаров, выбираемых пользователем, и обеспечивать выполнение операций добавления, удаления и очистки этого списка. Таким образом, текущее содержимое корзины будет меняться в зависимости от параметров HTTP-запросов, передаваемых пользователем в границах одного HTTP-сеанса.

Ниже приводится программный текст класса CartBean. Для представления корзины используется экземпляр класса java.util.ArrayList — нового служебного класса, описанного в JDK 1.2. В среде JDK 1.1 нужно модифицировать программу, воспользовавшись другим типом данных, например java.util.Vector.

```
/** Имя программы:   CartBean.java
 ** Назначение:      Реализовать компонент для корзины, в которую
 **                  можно добавлять товары и из которой их можно
 **                  удалять; кроме того, корзину можно очищать.
 ** /
package mybeans;

import java.util.ArrayList; // требует JDK 1.2

public class CartBean {
  // (см. пояснение 1)
  private String[] itemsSelected = null;    // выбираемые товары
  private String   action = null;           // выбираемое действие
  // (см. пояснение 2)
private ArrayList itemsList = new ArrayList();  // содержимое корзины

  /* метод для установки действия: добавить, удалить или очистить */
  public synchronized void setAction (String newAction) {
    // (см. пояснение 3)
    action = newAction;
    if (action.startsWith("Clear")) {  // очистить корзину
        itemsList.clear();
    }
  }
  /* метод для обработки нового списка выбранных товаров */
  public synchronized void setItemsSelected (String[] newItems) {
    // (см. пояснение 4)
    if (newItems != null) {  // какие-то товары выбраны
        for (int i = 0; i < newItems.length; i++) {
            if (action.startsWith("Add")){  // добавить в корзину
                itemsList.add(newItems [i]);
            } else if (action.startsWith("Remove")) {  // удалить
               // найти последний экземпляр этого товара в корзине
               int index = itemsList.lastIndexOf(newItems[i]);
               if (index != -1) {
                   itemsList.remove(index);// удалить из списка
               }
            }
         }
        }
    }
  // (см. пояснение 5)
  /* метод для получения текущего номера заказанных товаров */
  public int countItems() {
    return itemsList.size();
  }

  /* метод для получения конкретного товара */
  public Object getItem(int index) {
    return itemsList.get(index);
  }
```

```
  /* метод для получения всего списка заказанных товаров */
  public ArrayList getOrder() {
    return itemsList;
  }
} // конец CartBean
```

Пояснения к CartBean.java:

1. Зерно обладает двумя свойствами: `itemsSelected` и `action`. Свойство `itemsSelected` указывает на список товаров, выбранных пользователем на экране, а свойство `action` — на природу операции: добавление товаров, удаление товаров или очистка корзины. В зерне присутствуют методы set для обоих его свойств. Значения этих свойств только записываются, но не считываются извне, поэтому методы get отсутствуют.
2. Переменная `itemsList` указывает на текущее содержимое корзины.
3. Метод `setAction()` устанавливает значение атрибута `action`. Помимо этого, если действием является очистка корзины ("Clear the cart"), вызывается метод `itemsList.clear()`.
4. Метод `setItemsSelected()` исполняет логику установки содержимого корзины в зависимости от выбранного действия:
   - Если какие-то товары выбраны, а действием является добавление в корзину ("Add to cart"), в `itemsList` вносятся выбранные товары, для чего вызывается метод `itemsList.add()`.
   - Если какие-то товары выбраны, а действием является удаление из корзины ("Remove from cart"), из массива `itemsList` удаляется последний экземпляр каждого выбранного товара.
5. В зерне присутствуют три общих метода: `countItems()`, `getItem()` и `getOrder()`. Эти методы применяются соответственно для получения общего числа товаров, находящихся в тот момент в корзине, для получения товара в указанной позиции в списке и для получения списка товаров, накопленных в корзине. Эти методы используются в программе `EnterOrder.jsp` (см. ниже).

Создание программы EnterOrder.jsp

Теперь создадим страницу `EnterOrder.jsp`, использующую описанный выше класс `CartBean`. Эта JSP будет интегрироваться в зерне с HTML-текстом представления и с логикой вычислений, управляя процессом размещения заказов. Ввод заказа в базу данных заказов на покупку управляется другой JSP с именем `InsertOrder.jsp`, которая приводится в следующем разделе.

Программный текст `EnterOrder.jsp`:

```
<!-- Имя программы:  EnterOrder.jsp
  -- Назначение:     Страница для выбора заказываемых товаров.
  -->
<HTML>
  <HEAD> <TITLE> The EnterOrder JSP </TITLE> </HEAD>
  <BODY BGCOLOR="white">
    <H2> Welcome to Order Online! </H2>
    <!-- Воспользуемся зерном корзины с областью действия session -->
    <!-- (см. пояснение 1) -->
    <jsp:useBean id="cart" class="mybeans.CartBean" scope="session"/>
    <jsp:setProperty name="cart" property="action" />
    <jsp:setProperty name="cart" property="itemsSelected"
                                 parameter="item"/>

    <HR><BR>
    <BLOCKQUOTE> <!--  (см. пояснение 2) -->
        <H3>Please choose from the following items:</H3>
        <FORM METHOD="GET">
            <INPUT TYPE="Checkbox" NAME="item" VALUE="Book Shelf">
               Book Shelf <BR>
```

```
            <INPUT TYPE="Checkbox" NAME="item" VALUE="File Cabinet">
               Filing Cabinet <BR>
            <INPUT TYPE="Checkbox" NAME="item" VALUE="Office Desk">
               Office Desk <BR>
            <INPUT TYPE="Checkbox" NAME="item" VALUE="PC">
               Personal Computer <BR>
            <INPUT TYPE="Checkbox" NAME="item" VALUE="Wastebasket">
               Wastebasket <BR> <BR>
            <P> <INPUT TYPE=SUBMIT NAME="action" VALUE="Add to cart">
                <INPUT TYPE=SUBMIT NAME="action" VALUE="Remove from cart ">
                <INPUT TYPE=SUBMIT NAME="action" VALUE="Clear cart">
          </FORM>
        </BLOCKQUOTE>
        <CENTER>
          <!-- (см. пояснение 3) -->
          <FORM ACTION="InsertOrder.jsp" METHOD="GET">
            <INPUT TYPE="SUBMIT" NAME="insert" VALUE="Submit my order">
          </FORM>
        </CENTER>
        <!-- (см. пояснение 4) -->
        <HR><BR>
        <% int numItems = cart.countItems(); %>
        <H3>Your cart now contains
          <%= numItems %> item<%= (numItems > 1) ? "s" : "" %>.
        </H3>
        <UL>
        <% for (int i = 0; i < numItems; i ++ ) { %>
             <LI> <%= cart.getItem(i) %>
        <% }   %>
    </UL>
  </BODY>
</HTML>
```

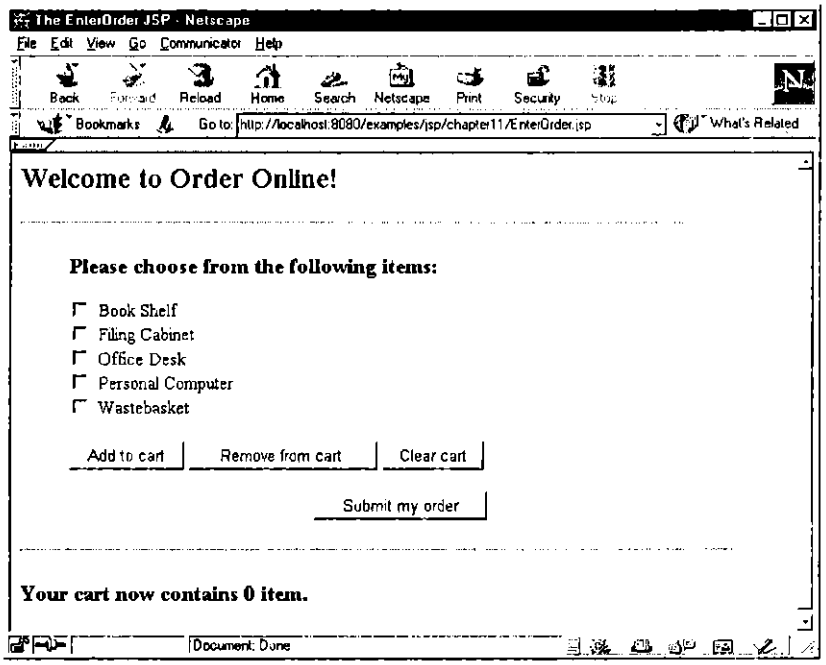
Пояснения к `EnterOrder.jsp`:

1. JSP использует ранее описанный класс `CartBean` с областью действия `session`. Два свойства этого зерна, `action` и `itemsSelected`, устанавливаются на основании параметров HTTP-запроса с именами `action` и `item` соответственно.
2. Страница состоит из двух сегментов HTML-форм. Первая форма содержит список товаров с флажками выбора, а также три функциональные кнопки "Add to cart", "Remove from cart" и "Clear cart". Щелчок мыши на этих кнопках устанавливает параметр запроса с именем `action` в соответствующее значение, и происходит обращение к той же самой JSP-программе (т.е. к `EnterOrder.jsp`) при помощи HTTP-метода `GET`.
3. Вторая форма состоит из одной кнопки передачи с именем "Submit my order". Если щелкнуть на этой кнопке мышью, форма направит HTTP-запрос на страницу `InsertOrder.jsp`. Предполагается, что этот запрос является частью того же HTTP-сеанса, в котором работает `EnterOrder.jsp`, поэтому другая JSP может обращаться в области действия `session` к тому же самому экземпляру `CartBean`. Значимость этого станет очевидной при написании программы `InsertOrder.jsp`.
4. В последней части страницы `EnterOrder.jsp` отображается список товаров, находящихся в корзине в текущий момент, для чего вызываются соответствующие методы экземпляра `CartBean`.

Выполнение приложения Order Online

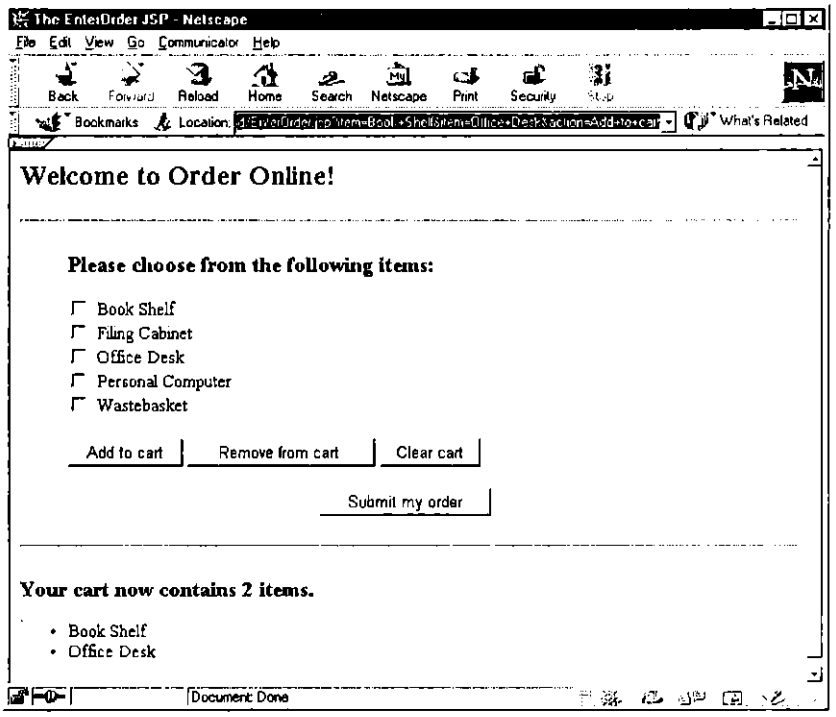
После установки на web-сервере файла `EnterOrder.jsp` и скомпилированного класса для `CartBean.java` (см. выше сценарий `buildbeans.sh` и операции установки в приложении А) можно вызвать программу `EnterOrder.jsp` при помощи соответствующего URL. На рис. 11.8 показан результат первого вызова этой страницы.

Рис. 11.8. Начальные выходные данные EnterOrder.jsp



Теперь выберем с помощью флажков на экране товары "Book Shelf" и "Office Desk", а затем щелкнем мышью на кнопке "Add to cart". На рис. 11.9 показаны выходные данные браузера после этих действий.

Рис. 11.9. Выходные данные EnterOrder.jsp после выбора товаров



Аналогично можно удалить товары из корзины, выбрав нужные флажки, а затем щелкнув мышью на кнопке "Remove from cart". Если щелкнуть мышью на кнопке "Clear cart", то из корзины будут удалены все товары. Чередовать действия "Add to cart", "Remove from cart" и "Clear cart" можно сколь угодно часто, при условии, что срок действия HTTP-сеанса не истек между этими операциями.

После того как выбор сделан, все готово к размещению заказа! Теперь можно щелкнуть мышью на кнопке "Submit my order", после чего начнет действовать вторая HTML-форма, которая перейдет к странице InsertOrder.jsp. Ее мы создадим позже.

Выполнение операций над базой данных с помощью зерен JavaBean

Выше рассказывалось о программировании процесса ввода заказа при помощи JSP и JavaBean корзины. В этом разделе рассматривается выполнение в JSP-приложении операций над базой данных. Будет реализован последний этап процесса ввода заказа – ввод нужной информации в базу данных Purchase Order.

Программные конструкции JDBC или SQLJ можно вводить непосредственно в JSP, чтобы обращаться к базе данных. Однако наличие большого объема логических схем Java на JSP-странице не слишком удобно при программировании серьезного JSP-приложения. Основой компонентного программирования является модульность (см. главу 2). С этой точки зрения лучшим стилем программирования является размещение программных конструкций Java в отдельном компоненте JavaBean с последующим вызовом их с JSP-страницы. При этом обеспечивается четкое отделение логики Java для динамической обработки информации от статического HTML-текста. Теперь посмотрим, как обращаться к базе данных Oracle при помощи JavaBean и как ввести зерно в JSP-приложение.

Ввод информации в базу данных

В JSP-приложении Order Online запрос на передачу заказа, производимый операцией "Submit my order" HTML-формы, направляется на страницу InsertOrder.jsp, которая вводит характеристики заказа в базу данных Purchase Order. Создадим эту JSP, а затем JavaBean с именем DBInsertBean, которое исполняет SQL-операцию INSERT для данных, собранных программой EnterOrder.jsp.

Создание InsertOrder.jsp

Ниже приводится текст программы InsertOrder.jsp и даются пояснения. Главное, что стоит отметить,– для обработки информации JSP использует два зерна JavaBean:

- Экземпляр cartBean с областью действия session, созданный JSP-страницей EnterOrder. В этом зерне содержится список товаров, выбранных пользователем приложения Order Online.
- Экземпляр DBInsertBean с областью действия page, который является локальным компонентом для данной JSP. Это зерно будет подключаться к базе данных Oracle и исполнять необходимые SQL-операции.

```
<%-- Имя программы:  InsertOrder.jsp
  -- Назначение:     Страница для ввода заказа, созданного при помощи
  --                 JSP-страницы EnterOrder, в базу данных Purchase Order.
  --%>
<HTML>
<!-- Воспользуемся зерном корзины с областью действия session -->
  <jsp:useBean id="cart" class="mybeans.CartBean"
                         scope="session" /> <!-- (см. пояснение 1) -->
  <HEAD> <TITLE> The InsertOrder JSP </TITLE> </HEAD>
  <BODY BGCOLOR="white">
    <H2> Ready to Order! </H2>       <!-- (см. пояснение 2) -->
    <B> You have selected: </B>      <%= cart.getOrder() %>
<%    if ((request.getParameter("DBInsert")) != null) {
%>
    <!--  (см. пояснение 3) -->
```

```
    <jsp:useBean id="dbInsert" class="mybeans.DBInsertBean"
                 scope="page" />
    <jsp:setProperty name="dbInsert" property="*" />
<H3> Result of processing order: </H3>
<PRE>    <!-- (см. пояснение 4) -->
<%=  dbInsert.processOrder(cart.getOrder()) %>
</PRE>
<!--   Очистим содержимое корзины ->
<!--   (см. пояснение 5) ->
<%   cart.setAction("Clear"); }
     else {
  %>
  <HR><BR>
  <BLOCKQUOTE> <!-   (см. пояснение 6) ->
  <P><H3>Please fill in the following information: </H3>
  <FORM METHOD="POST" >
    <P> <b>Employee Number:</b> <INPUT TYPE="TEXT" NAME="empNum" SIZE=20>
        <b>Project Number:</b> <INPUT TYPE="TEXT" NAME="projNum" SIZE=20>
    <P>   <INPUT TYPE="SUBMIT" NAME="DBInsert" VALUE="Submit Order">
  </FORM>
<% } %>
<BR><BR>
<FORM METHOD="GET" ACTION="EnterOrder.jsp">
        <P>     <INPUT TYPE="SUBMIT" VALUE="Back to Order Entry">
    </FORM>
    </BLOCKQUOTE>
  </BODY>
</HTML>
```

Пояснения к `InsertOrder.jsp`:

1. Предполагается, что JSP-страница `InsertOrder` вызывалась с JSP- страницы `EnterOrder` после завершения процесса выбора товаров, так что экземпляр `CartBean` уже создан JSP-страницей `EnterOrder`, и для него установлена область действия `session`. Поэтому первый JSP-тэг `<jsp:useBean>` JSP-страницы `InsertOrder` считывает в области действия `session` этот экземпляр `CartBean` и делает его доступным для остальной части программы.
2. Этот оператор распечатывает список товаров, находящихся в корзине в текущий момент. Для этого вызывается метод `getOrder()` экземпляра `CartBean`.
3. Второй тэг `<jsp:useBean>` исполняется по условию, только когда в HTTP-запросе присутствует параметр с именем `DBInsert`. Этот тэг создает экземпляр класса зерна `mybeans.DBInsertBean` (см. ниже) с идентификатором `dbInsert`. Зерно создается с областью действия `page`, поскольку операция, выполняемая над базой данных, не имеет смысла вне области действия этой JSP. Свойства экземпляра зерна `dbInsert` устанавливаются на основании параметров HTTP-запроса.
4. Следующее JSP-выражение вызывает метод `processOrder()` зерна `dbInsert` с заказом, считываемым в экземпляре `CartBean`, в качестве аргумента. Результат вызова этого метода отображается на экране.
5. Этот оператор очищает корзину, сбрасывая список покупок.
6. В последней части JSP-страницы используются сегменты двух HTML-форм:
   - В сегменте первой формы присутствуют два текстовых поля для ввода номера служащего (`empNum`) и номера проекта (`projNum`) заказа. Указав в этих полях необходимые данные, можно щелкнуть мышью на кнопке "Submit my order", чтобы еще раз вызвать ту же JSP, но с параметром `DBInsert`, установленным в HTTP-запросе. Как отмечалось выше, при наличии этого параметра заказ передается зерну `dbInsert` для обработки.

- В сегменте второй формы полей ввода нет, но есть кнопка "Back to Order Entry", возвращающая пользователя к начальному экрану ввода заказа, для чего в форме вызывается EnterOrder.jsp.

Создание DBInsertBean.sqlj

Назначением класса DBInsertBean является ввод списка выбранных товаров в базу данных Purchase Order. Следовательно, он должен выполнять необходимые операции над таблицами. Схема для этих таблиц описана во введении к книге. Для удобства воспроизведем таблицы PURCHASE_LIST и LINEITEM_LIST, имеющие отношение к этому классу:

```
CREATE TABLE PURCHASE_LIST (
   requestno       NUMBER(10),
   employeeno      NUMBER(7),
   vendorno        NUMBER(6),
   purchasetype    VARCHAR2(20),
   checkno         NUMBER(11),
   whenpurchased   DATE);

CREATE TABLE LINEITEM_LIST (
   requestno       NUMBER(10),
   lineno          NUMBER(5),
   projectno       NUMBER(5),
   quantity        NUMBER(5),
   unit            VARCHAR2(2),
   estimatedcost   NUMBER(8,2),
   actualcost      NUMBER(8,2),
   description     VARCHAR2(30));
```

Для приема данных, вводимых в форме JSP-страницы InsertOrder, в JavaBean должны существовать два свойства с именами empNum и projNum. Текст данного JavaBean приведен ниже. Оно написано с использованием программных конструкций SQLJ, что упрощает исполнение операций над базой данных.

```
/** Имя программы:   DBInsertBean.sqlj
 ** Назначение:      Ввести заказ в таблицы базы данных заказов на покупку.
 **/

package mybeans;

import java.util.ArrayList;
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext ;

public class DBInsertBean {
  // (см. пояснение 1)
  private String empNum = null; // номер служащего
  private String projNum= null; // номер проекта

  // Метод для установки свойства empNum
  public synchronized void setEmpNum(String empNum) {
    this.empNum = empNum;
  }

  // Метод для установки свойства projNum
  public synchronized void setProjNum(String projNum) {
    this.projNum = projNum;
  }

  private DefaultContext dctx = null;    // контекст соединения с базой данных
  private int requestno;                 // номер запроса для заказа

  // Метод для обработки переданного заказа
```

```
    // (см. пояснение 2)
    public synchronized String processOrder (ArrayList order) {
      if ((order == null) || (order.size() == 0)) {
          return ("No order items to insert!");
      }
      try {
          // Подключимся к базе данных
          // (см. пояснение 3)
          dctx = Oracle.getConnection("jdbc:oracle:oci8:@",
                                      "jspuser", "jsp");
          // Введем заказ в базу данных
          DBInsert(order);
          // Ошибок нет!
          return ("Purchase order processed successfully! <BR> " +
                  "Your request number is: " + requestno);
      } catch (SQLException e) {
            return ("SQL Error:" + e.getMessage());
      } finally {
          // Отключимся от базы данных
          try {
            dctx.close(); // (см. пояснение 4)
          } catch (Exception e) {}
      }
    }

  private void DBInsert(ArrayList order) throws SQLException {
      // (см. пояснение 5)
      // Сначала узнаем следующий порядковый номер запроса
      #sql [dctx] { SELECT requestno_seq.NEXTVAL
                     INTO :requestno
                       FROM DUAL };

      // Затем введем запрос в таблицу PURCHASE_LIST
      #sql [dctx] { INSERT INTO
                    PURCHASE_LIST( requestno,
                                   employeeno,
                                   whenpurchased)
                    VALUES( :requestno,
                            :empNum,
                             SYSDATE
                    )
      };

    // Теперь введем все пункты ассортимента из переданного заказа
    for (int i = 0; i < order.size(); i++) {
        #sql [dctx] { INSERT INTO
                     LINEITEM_LIST ( requestno,
                                          lineno,
                                          projectno,
                                     quantity,
                                     description
                           )
                VALUES (:requestno,

                              lineno_seq.NEXTVAL,
                             :projNum.
                              1,
                           :(order.get (i))
                )
        };
        #sql [dctx] { COMMIT }; // завершим ввод строк
    }
  }
}
```

Пояснения к `DBInsertBean.sqlj`:

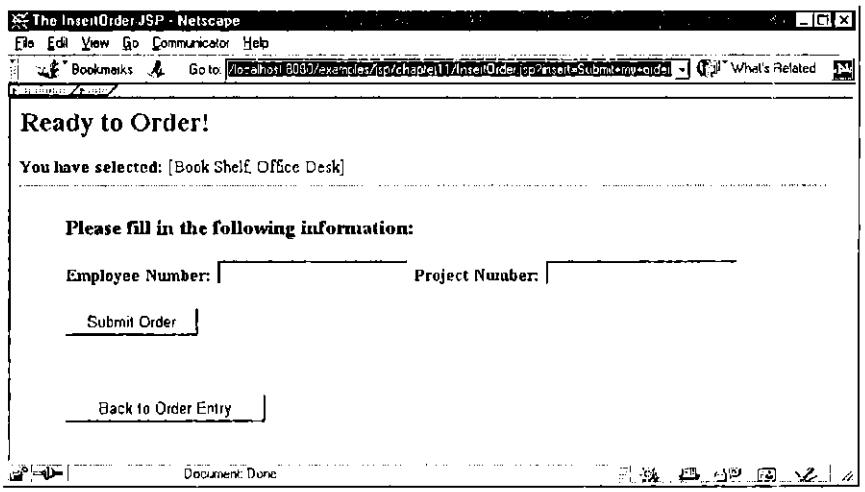
1. У `DBInsertBean` два свойства: `empNum` и `projNum`. Предлагаются методы для установки значений этих свойств. Значения этих свойств только записываются, но не считываются, поэтому методы `get` отсутствуют.
2. Главный метод зерна — `processOrder()`, который принимает заказ типа `java.util.ArrayList` в качестве аргумента. Заказ обрабатывается следующим образом:
   - Если товаров в заказе нет, метод немедленно возвращает соответствующее сообщение.
   - В противном случае он подключается к базе данных с помощью метода `Oracle.getConnection()`, вызывает метод `DBInsert()` для ввода заказа в базу данных и возвращает номер только что введенного заказа. Затем он отключается от базы данных.
3. Стоит сказать несколько слов по поводу установления соединения с базой данных. Метод `getConnection()` класса `oracle.sqlj.runtime.DefaultContext` вызывается для того, чтобы вернуть SQLJ-контекст соединения `dctx`. Этот контекст соединения явным образом используется в каждом последующем SQL-операторе при помощи синтаксиса `#sql [dctx]`. Причина применения явного контекста соединения в каждой SQL-операции весьма интересна и неочевидна. Вспомним, что одновременно с JSP-приложением могут работать несколько пользователей, применяющих одну и ту же виртуальную машину Java web-сервера, и для ввода заказов они могут подключаться к базе данных одновременно. Ясно, что в такой ситуации устанавливать и использовать стандартный контекст соединения SQLJ (который реализуется исполняющей системой SQLJ через статическую переменную), нельзя, так как пользователи будут мешать друг другу подключаться. *Поэтому в SQLJ-операторах нужно всегда применять явный контекст соединения, чтобы избежать проблем при параллельной работе пользователей.*
4. И последнее замечание относительно соединений с базой данных. В этом зерне соединение с базой данных открывается и закрывается для каждого передаваемого заказа. На практике открытие и закрытие соединений — довольно ресурсоемкие операции, и число их следует сводить к минимуму. Более того, подключение каждого HTTP-сеанса может привести к слишком большому числу открытых соединений, особенно если велико число пользователей, одновременно работающих с приложением Order Online. Во избежание этого можно применять *пулинг соединений* (connection pooling) для совместного использования соединений с базой данных (см. ниже раздел "Оптимизация SQL-операций").
5. Метод `DBInsert()` использует SQLJ для выполнения следующих операций над базой данных:
   - Ввод одной записи в таблицу `PURCHASE_LIST` для запроса на покупку. Номер запроса извлекается из последовательности запросов при помощи SQL-оператора `SELECT INTO`, считывающего следующий порядковый номер `requestno_seq.NEXTVAL`.
   - Ввод записей в таблицу `LINEITEM_LIST` для каждого товара из списка заказанных, для чего используется номер запроса, ранее выбранный в последовательности запросов. Номера в ассортименте генерируются в последовательности `lineno_seq`.

Выполнение `InsertOrder.jsp`

Сначала необходимо установить программу `InsertOrder.jsp` на web-сервере, следуя инструкциям приложения Е. Не забудьте скомпилировать программный текст `DBInsertBean.sqlj` и сделать скомпилированный класс доступным web-серверу. После этого можно будет вызывать JSP-страницу `InsertOrder` при помощи кнопки "Submit my order" на странице `EnterOrder.jsp` приложения Order Online.

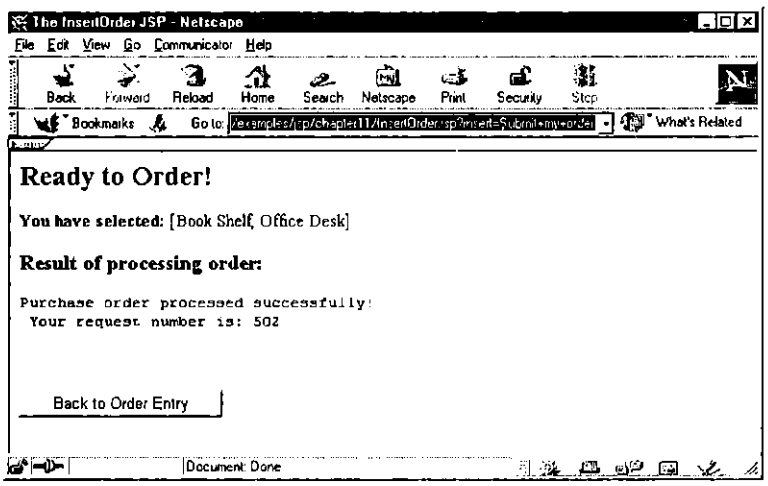
На рис. 11.10 показан начальный экран страницы InsertOrder.jsp, выводимый после выбора товаров "Book Shelf" и "Office Desk" при помощи EnterOrder.jsp.

Рис. 11.10. Начальный экран JSP-страницы InsertOrder



Если теперь ввести номер служащего и номер проекта в форму, отображаемую JSP-страницей InsertOrder, а затем щелкнуть мышью на "Submit Order", то этими двумя параметрами запроса будет создан и инициализирован экземпляр класса DBInsertBean. Вызов метода processOrder() приводит к тому, что зерно исполняет над базой данных необходимые операции INSERT, и для успешного ввода заказа генерируются выходные данные, изображенные на рис. 11.11. В этот момент можно вернуться к JSP-странице EnterOrder, щелкнув мышью на кнопке "Back to Order Entry".

Рис. 11.11. Конечный экран JSP-страницы InsertOrder после успешной передачи заказа



Исполнение статических SQL-запросов

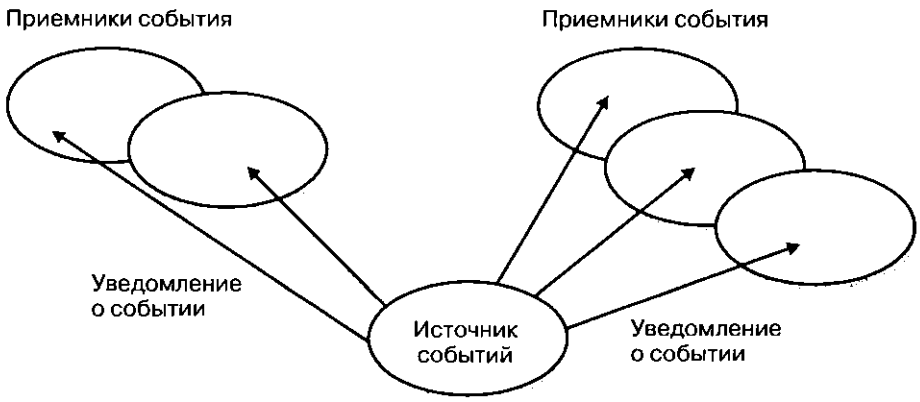
Итак, мы ввели информацию в базу данных заказов на покупку. Теперь посмотрим, как исполнять статические SQL-запросы и отображать результаты, выдаваемые JSP-страницей. В частности, для выбора характеристик заказа на покупку в базе данных воспользуемся SQLJ-итератором. Но перед созданием программы обсудим базовые концепции, касающиеся событий и интерфейсов приемников событий. Подобные интерфейсы реализуются при помощи зерен JavaBean, что позволяет эффективно применять такие общие ресурсы, как соединения с базой данных.

События и интерфейсы приемников событий

Событие (event) – это общий термин Java для механизма уведомления, в котором задействованы различные компоненты Java-приложения. Так, щелчок кнопкой мыши – это событие. С помощью механизма уведомления о событиях любой объект Java может регистрировать свою заинтересованность в одном или нескольких событиях, генерируемых другими объектами. Потом он выполняет работу, называемую *прослушиванием* (listening) событий. Объекты, генерирующие события, называются *источниками событий* (event source), а объекты, уведомляемые о событиях,– *приемниками событий* (event listener).

Основой механизма уведомления о событиях являются методы реализации, содержащиеся в *интерфейсах приемников событий* (event listener interface). Каждое событие соответствует конкретному методу интерфейса, который можно реализовать при помощи класса Java. Объект приемника событий регистрирует себя в источнике, разрешая уведомление о событиях. При возникновении того или иного события объект источника событий вызывает соответствующий метод для всех объектов, зарегистрировавшихся в качестве приемников данного события. Само событие представляется объектом события, в котором содержится ссылка на источник событий, и передается в качестве параметра вызываемому методу. На рис. 11.12 приведена диаграмма модели уведомления о событиях.

Рис. 11.12. Модель событий с источниками и приемниками событий



Заметим, что модель событий не специфична для зерен JavaBean, а является общим свойством языка программирования Java. Несколько классов и интерфейсов, поддерживающих модель событий, описаны в Java-пакете `java.util`.

В качестве конкретного примера рассмотрим класс JavaBean с областью действия `session`. Это зерно может быть заинтересовано в уведомлении о своем сохранении (привязке) в HTTP-объекте `session` или удалении (отмене привязки) из него, чтобы иметь возможность управлять своими ресурсами, такими, как курсоры и соединения базы данных. Для этой цели в API сервлетов описывается интерфейс приемника типа `javax.servlet.http.HttpSessionBindingListener`, содержащий два метода: `valueBound()` и `valueUnbound()` – со следующими сигнатурами:

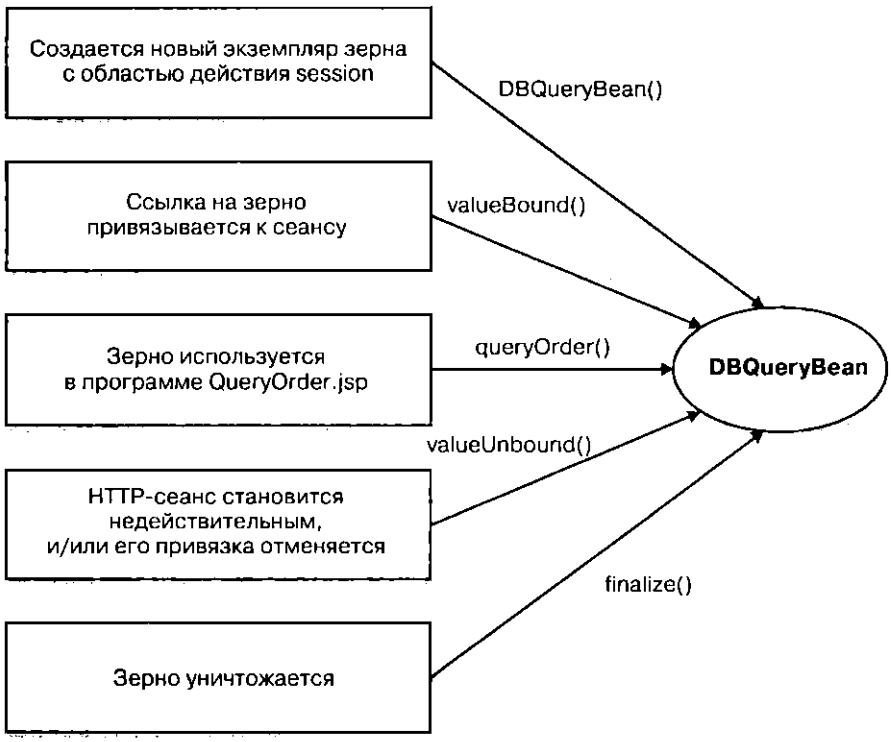
```
public interface HttpSessionBindingListener
                 extends java.util.EventListener {
    public void valueBound(HttpSessionBindingEvent event);
    public void valueUnbound(HttpSessionBindingEvent event);
}
```

Этот интерфейс приемника событий можно реализовать при помощи JavaBean с областью действия `session`. Метод `valueBound()` вызывается, когда зерно сохраняется в HTTP-объекте `session`, в составе операций, выполняемых тэгом `<jsp:useBean>` с областью действия `session`. Метод `valueUnbound()` вызывается, когда зерно удаляется из объекта `session`.

Вызовы методов, выполняемые во время действия зерна `DBQueryBean` (см. ниже), показаны на рис. 11.13. Это зерно для своего уведомления о событиях,

имеющих отношение к сеансу, реализует интерфейс HttpSessionBindingListener. Предположим, что с помощью тэга <jsp:useBean> создается экземпляр зерна с областью действия session. Он автоматически привязывается к HTTP-объекту session (вызовом метода session.setAttribute()). Операция привязки приводит к вызову метода valueBound() зерна через интерфейс приемника событий. После этого JSP-страница может обращаться к экземпляру зерна и вызывать такие его методы, как queryOrder(). При завершении HTTP-сеанса (например, методом session.invalidate()) или при явном удалении ссылки на зерно из сеанса (методом session.remove()) для зерна вызывается метод valueUnbound(). Кроме того, в зерне может присутствовать метод finalize(), вызываемый после уничтожения экземпляра зерна.

Рис. 11.13. JavaBean с областью действия session, прослушивающее события сеанса



Для чего нужно реализовывать в зерне специальные сеансовые методы приемника? Почему бы ни закрыть соединение с базой данных методом finalize() зерна, который бы вызывался во время "уборки"? Обычно считается, что это проще программировать. Но это не так. Действие метода finalize() не вполне совпадает с действием методов приемника сеанса. Если быть более точным, то действие интерфейса приемника гораздо лучше определено по сравнению с методом finalize(). Естественно, после завершения HTTP-сеанса ссылка на зерно с областью действия session становится недоступной. Скорее всего, после этого будет проводиться очистка ("уборка") зерна, для чего вызывается метод finalize(). Однако частота очистки зависит от расходования оперативной памяти во всей виртуальной машине Java, где исполняется одно или несколько приложений. Таким образом, неясно, когда зерно будет очищаться на самом деле, и до этого момента соединение с базой данных будет оставаться открытым. С точки зрения эффективности работы, удерживать соединение в течение неизвестного времени недопустимо. Именно с этой целью в интерфейсах приемников HTTP-сеансов хорошо описаны средства обратного вызова для уведомления о событиях.

Создание DBQueryBean.sqlj

Теперь создадим зерно DBQueryBean, которое с помощью статического SQL-запроса считывает характеристики конкретного заказа в базе данных заказов

на покупку. Это зерно реализует интерфейс приемника событий HTTP-сеанса, освобождая соединение с базой данных по завершении сеанса или по окончании срока его действия. SQL-запрос в DBQueryBean выполняется на основании свойства requestNum зерна, после чего результаты запроса форматируются для отображения в браузере. Следует сделать два важных замечания относительно логики этого зерна:

- Оно реализует интерфейс javax.servlet.http.HttpSessionBindingListener и предназначено для использования в области действия session.
- Оно помнит контекст SQLJ-соединения в течение HTTP-сеанса, применяя для этого переменную экземпляра dctx. Соединение с базой данных, представляемое переменной dctx, закрывается при вызове метода valueUnbound() приемника событий.

Приведем текст программы DBQueryBean.sqlj:

```
/** Имя программы:   DBQueryBean.sqlj
 ** Назначение:      Считать информацию о заказе из базы данных
 **                  Purchase Order с помощью SQLJ.
 **/

package mybeans;

import java.sql.*;
import javax.servlet.http.*;
import oracle.sqlj.runtime.Oracle;
import sqlj.runtime.ref.DefaultContext;

public class DBQueryBean
             implements HttpSessionBindingListener { // (см. пояснение 1)

  String requestNum = null; // (см. пояснение 2)
  // Метод для установки свойства requestNum
  public synchronized void setRequestNum(String requestNum) {
    this.requestNum = requestNum;
    result = null;
  }

  private StringBuffer result = null;    // результат запроса
  private DefaultContext dctx = null;    // контекст соединения с базой данных

  // Метод для считывания характеристик заказа
  public synchronized String queryOrder() {  // (см. пояснение 3)

    if (result != null) // такой запрос уже выполнялся, поэтому
                        // воспользуемся его результатом!
        return result.toString();

    try {
        // При необходимости подключимся к базе данных
        if (dctx == null)
          dctx = Oracle.getConnection("jdbc:oracle:oci8:@",
                                      "jspuser", "jsp");// (см. пояснение 4)
    // Запросим заказ в таблицах базы данных
    DBQuery();

    return result.toString() ;
    // Но не будем закрывать соединение с базой данных!

  } catch (SQLException e) {
         return("Error in querying: " + e.getMessage());
    }
  }

  // Объявление SQLJ-итератора для пунктов ассортимента
```

```
// (см. пояснение 5)
#sql iterator LineItems (int lineno, int projectno,
                         int quantity, String description);
private LineItems li = null;

// Метод для выбора заказа на покупку и данных о пунктах
// ассортимента для нового заказа
private String DBQuery() {
  int empNum;
  String empName;

  try {    // (см. пояснение 6)
      #sql [dctx] { SELECT PURCHASE_LIST.employeeno,
                          firstname || ' ' || lastName
                    INTO :empNum,
                         :empName
                    FROM PURCHASE_LIST, EMPLOYEE_LIST
                     WHERE requestno = :requestNum
                           AND EMPLOYEE_LIST.employeeno =
                             PURCHASE_LIST.employeeno
  };

      result = new StringBuffer();
      result.append("Employee: " + empName +
                    " , number: " + empNum + "<BR><BR>");

      // Теперь запросим пункты ассортимента для заказа
      // (см. пояснение 7)
      #sql [dctx] li = {
                      SELECT   lineno,
                               projectno,
                               quantity,
                               description
                        FROM LINEITEM_LIST
                        WHERE requestno = :requestNum
            };
  formatHTML() ;
  li.close(); // закроем итератор
      return result.toString();
  } catch (SQLException e) {
        return("Error on DB Query: " + e.getMessage());
  }
}

private void formatHTML() throws SQLException {  // (см. пояснение 8)
  if (li == null // !li.next()) {
      result.append("<P> No matching rows.<P>\n");
  return;
}

  // Распечатаем заголовок таблицы с характеристиками заказа
  result.append("<TABLE BORDER>\n");
  result.append("<TH><I> Line # </I></TH>");
  result.append("<TH><I> Project # </I></TH>");
  result.append("<TH><I> Quantity </I></TH>");
  result.append("<TH><I> Description </I></TH>");

      do { // Распечатаем данные для каждого заказанного товара
        result.append("<TR>\n");
        result.append("<TD>" + li.lineno() + " </TD>");
        result.append("<TD>" + li.projectno() + " </TD>");
        result.append("<TD>" + li.quantity() + " </TD>");
        result.append("<TD>" + li.description() + " </TD>");
        result.append("</TR>");
```

```
        } while (li.next());
        result.append("</TABLE>");
    }

  // (см. пояснение 9)
  // Метод, исполняемый при запуске HTTP-сеанса
  public void valueBound(HttpSessionBindingEvent event) {
    // Здесь ничего не делаем!
    // Соединение с базой данных будет открыто только при передаче запроса.
  }

  // Метод, исполняемый при выходе из HTTP-сеанса
  public synchronized void valueUnbound(HttpSessionBindingEvent event) {
    if (dctx != null) {
        try { dctx.close();
      } catch (SQLException e) {}
    }
  }
}
```

Пояснения к `DBQueryBean.sqlj`:

1. `DBQueryBean` реализует интерфейс `HttpSessionBindingListener` и должно вызываться посредством тэга `<jsp:useBean>` с областью действия `session`. JSP-страница `QueryOrder` (см. ниже) использует экземпляр этого зерна с областью действия `session`.
2. `DBQueryBean` имеет одно свойство с именем `requestNum`, которое представляет номер запрашиваемого заказа. Метод `setRequestNum()` служит для установки значения этого свойства. Кроме того, данный метод сбрасывает результат любого ранее исполнявшегося запроса.
3. JSP-страница `QueryOrder` будет вызывать метод `queryOrder()` зерна, если пользователь укажет параметр поиска (т.е. номер заказа) в HTML-форме. Поэтому для минимизации продолжительности соединения зерно не подключается к базе данных с помощью своего метода `valueBound()`, который вызывается при создании зерна и его привязке к HTTP-объекту `session`. Вместо этого соединение с базой данных открывается методом `queryOrder()` при его первом вызове с номером запроса. Это соединение продолжает быть открытым в течение оставшегося срока действия зерна, что определяется длительностью HTTP-сеанса. Другой возможный подход — поместить тэг `<jsp:useBean>` в условную логическую конструкцию, исполняемую только при первом вводе номера запроса, и открыть соединение с базой данных при создании экземпляра зерна.
4. Этот оператор устанавливает контекст SQLJ-соединения с помощью драйвера JDBC-OCI.
5. Этот оператор объявляет указанный SQLJ-итератор для считывания информации о пунктах ассортимента.
6. SQLJ-оператор `SELECT INTO` используется для считывания номера и имени служащего, соответствующего заказу.
7. Экземпляр итератора `LineItems` заполняется соответствующим SQL-запросом с помощью значения свойства `requestNum`.
8. Метод `formatHTML()` форматирует результаты запроса в виде HTML-таблицы с соответствующими заголовками столбцов.
9. Обратите внимание на то, что зерно реализует два метода интерфейса `HttpSessionBindingListener`: `valueBound()` и `valueUnbound()`. Метод `valueBound()` вызывается при создании зерна и его связывании с HTTP-сеансом. Однако совсем не обязательно, чтобы этот метод выполнял какие-то действия, так как соединение с базой данных открывается только при передаче запроса. Метод `valueUnbound()` закрывает соединение с базой данных, если значение не равно `null`.

Создание QueryOrder.jsp

Теперь можно написать программу QueryOrder.jsp, в которой используется описанный выше класс DBQueryBean. Область действия этого зерна — session, так как оно реализует интерфейс HttpSessionBindingListener. Программный текст JSP-страницы приводится ниже, причем важные его участки выделены жирным шрифтом. Вам должны быть понятны общие закономерности JSP-логики. HTML-форма применяется для получения от пользователя номера запроса. Если номер не равен null, вызывается метод queryOrder() экземпляра DBQueryBean, отображая характеристики конкретного заказа.

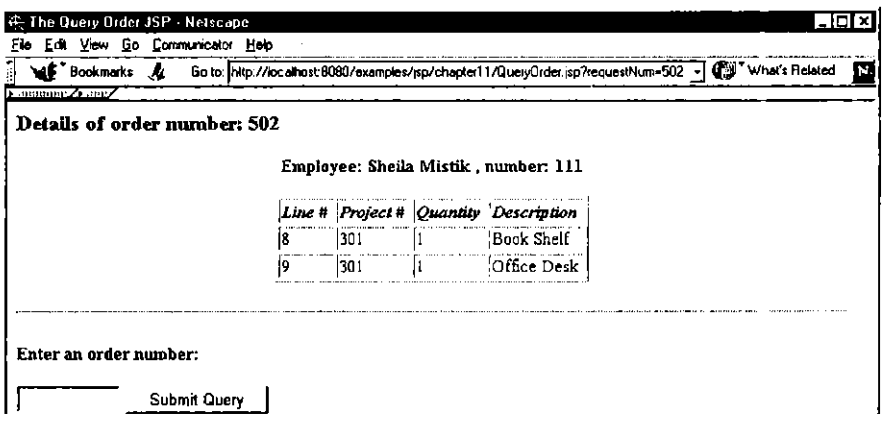
```
<%-- Имя программы:  QueryOrder.jsp
  -- Назначение:     Ввести номер заказа и запросить его
  --                 характеристики в базе данных заказов на покупку.
  --%>
<jsp:useBean id="queryBean" class="mybeans.DBQueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="requestNum" />
<HTML>
<HEAD> <TITLE> The Query Order JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<%  String requestNum = request.getParameter("requestNum");
if (requestNum != null) { %>
        <H3>Details of order number: <%= requestNum %> </H3>
        <CENTER><B>
<%  = queryBean.queryOrder() %>
        </CENTER><B><BR>
        <HR>
<%  }    %>

<P><B>Enter an order number:</B></P>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="requestNum" SIZE=10>
<INPUT TYPE="submit" VALUE="Submit Query">
</FORM>
</BODY>
</HTML>
```

Выполнение QueryOrder.jsp

Необходимо установить на web-сервере JSP-программу QueryOrder и скомпилированный класс DBQueryBean (см. приложение Е). После этого можно будет вызывать JSP, указывая соответствующий URL. На рис. 11.14 представлены выходные данные запроса для заказа №502, полученные с помощью JSP-страницы QueryOrder.

Рис. 11.14. Результат считывания заказа посредством QueryOrder.jsp



Исполнение динамических SQL-запросов

В примере с QueryOrder.jsp SQL-запрос статичен. Другими словами, во время разработки программы известно, какие столбцы каких таблиц нужно считывать и какими условиями поиска пользоваться в SQL-операторе SELECT. В некоторых приложениях до этапа выполнения программы доступна не вся информация. В этих случаях необходимо применять API JDBC, так как в SQLJ исполняются только статические SQL-запросы и операции обновления. Напомним, что конструкции SQLJ и JDBC способны взаимодействовать друг с другом, поэтому в приложении их можно смешивать в любых комбинациях.

В этом разделе мы напишем JavaBean, использующее JDBC для выполнения динамического SQL-запроса. Это зерно, создаваемое как класс DynamicQueryBean, будет применяться в качестве компонента в программе AnyQuery.jsp для динамического исполнения любого SQL-запроса, указываемого в HTML-форме. Сначала разработаем JavaBean, а затем напишем страницу AnyQuery.jsp, вызывающую это зерно.

Создание DynamicQueryBean.java

Функцией этого зерна запросов является исполнение любого запроса, обращенного к базе данных Purchase Order. Текст программы приводится ниже, причем важные участки выделены жирным шрифтом. Заметим, что это зерно, как и созданная в предыдущем разделе программа DBQueryBean.sqlj, реализует интерфейс HttpSessionBindingListener. Таким образом, данное зерно будет использоваться в области действия session на JSP-странице и закрывать свое соединение с базой данных при завершении HTTP-сеанса.

Главное, на что стоит обратить внимание в программе DynamicQuerybean,— это общий способ форматирования результата SQL- запроса в HTML-таблице. Данную функцию выполняет метод formatResult(), принимающий результирующее множество JDBC и генерирующий HTML-таблицу с соответствующими заголовками и строками. Более детально логика этого метода представлена в тексте программы.

```
/** Имя программы: DynamicQueryBean.java
 ** Назначение: Выполнить динамический JDBC-запрос к базе
 ** данных Purchase Order.
 **/

package mybeans;

import java.sql.*;
import javax.servlet.http.HttpSessionBindingListener;
import javax.servlet.http.HttpSessionBindingEvent;

public class DynamicQueryBean implements HttpSessionBindingListener{

  public void DynamicQueryBean() {
  }
  private String query = "";          // (см. пояснение 1)
  private String result = null;       // (см. пояснение 2)
  public synchronized void setQuery(String newQuery) {
       result = null;
       query = newQuery;
  }

  // Главный метод для получения результата запроса
  public String getResult() throws SQLException { // (см. пояснение 3)
    if (result != null) return result;
    else return (runQuery());
  }

  private Connection conn = null;
  private synchronized String runQuery() { // (см. пояснение 4)
    try {
```

```
            if (conn == null) {
              DriverManager.registerDriver(
                           new oracle.jdbc.driver.OracleDriver());
              conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                 "jspuser", "jsp");
            }
            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery (query);
            result = formatResult(rset));
            if (rset!= null) rset.close();
            if (stmt!= null) stmt.close();
            return result;
          } catch (SQLException e) {
            return ("<P> SQL error for query: <PRE> " +
                    query + "<BR>" + e + " </PRE> </P>\n");
          }
        }

        private String formatResult(ResultSet rs) // (см. пояснение 5)
                       throws SQLException {
          StringBuffer sb = new StringBuffer();
          if (rs == null || !rs.next()) {
            sb.append("<P> No matching rows found.<P>\n");
            return sb.toString();
          }
          sb.append("<TABLE BORDER>\n");
          ResultSetMetaData md = rs.getMetaData();
          int numCols = md.getColumnCount();
          for (int i=1; i<= numCols; i++) {
            sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
          }

      do {
          sb.append("<TR>\n");
          for (int i = 1; i <= numCols; i++) {
            sb.append("<TD>");
            Object obj = rs.getObject(i);
            if (obj != null) sb.append(obj.toString());
                    sb.append("</TD>");
          }
          sb.append("</TR>");
        } while (rs.next());
        sb.append("</TABLE>");
        return sb.toString();
        }

        // Метод, исполняемый при запуске сеанса
        public void valueBound(HttpSessionBindingEvent event) {
          // Здесь ничего не делается. Зерно создается только при передаче запроса.
        }

        // Метод, исполняемый при выходе из сеанса
        public synchronized void valueUnbound(HttpSessionBindingEvent event)
        {
          try {
              if (conn != null) conn.close(); // (см. пояснение 6)
           }  catch (SQLException e) { }
        }
      }
```

Пояснения к DynamicQueryBean.java:

1. Зерно DynamicQueryBean имеет одно свойство с именем query. Это свойство представляет текст исполняемого SQL-запроса и устанавливается пользователем в HTML-форме.
2. Локальная переменная result содержит результат, возвращаемый предыдущим запросом (если запрос исполнялся). Это значение кэшируется зерном, чтобы его можно было повторно использовать при исполнении такого же запроса. Следовательно, метод setQuery() делает кэшированную копию результата недостоверной, устанавливая для нее значение null.
3. Метод getResult() — это общий метод, который вызывается для исполнения запроса и выдачи результата, форматированного в виде HTML-таблицы. Метод возвращает кэшированный результат запроса, если тот доступен (остался от предыдущего исполнения с тем же параметром запроса), или вызывает метод runQuery() зерна.
4. Метод runQuery() — частный метод, который содержит реальный текст программы исполнения запроса. С помощью JDBC он открывает соединение с базой данных (если оно еще не открыто), а затем, после получения описателя оператора для соединения, исполняет запрос, создавая результирующее множество данных. Результирующее множество форматируется методом formatResult().
5. Метод formatResult() интересен с точки зрения программирования. Он принимает результирующее множество JDBC в качестве аргумента и форматирует строки в виде HTML-таблицы. Сначала он проверяет, не пусто ли результирующее множество. Если это так, он возвращает сообщение о том, что ни одной строки не найдено; в противном случае он, вызывая метод getColumnLabel() объекта ResultSetMetadata JDBC-аргумента ResultSet, определяет имена столбцов, возвращаемых запросом. Используя эти имена, он создает заголовок HTML-таблицы. Затем он обрабатывает каждую строку результирующего множества и форматирует их в виде HTML-таблицы, применяя соответствующие HTML-тэги и вызывая метод toString() для каждого выбираемого объекта.
6. Наконец, после завершения HTTP-сеанса метод valueUnbound() интерфейса HttpSessionBindingListener закрывает описатель JDBC-соединения.

Создание AnyQuery.jsp

Программный текст этой JSP-страницы приводится ниже. Отметим, что в программе используется класс DynamicQueryBean с областью действия session и что его свойство query устанавливается на основании текста SQL-запроса, вводимого в текстовой области HTML-формы. JSP проверяет, указан ли в HTTP-запросе параметр с именем query, и, если это так, вызывает метод getResult() зерна DynamicQueryBean для исполнения запроса и выводит отформатированный результат. В текстовой области содержится запрос по умолчанию, с которого можно начать работу. Если пользователь вводит новый запрос, запрос по умолчанию заменяется текущим.

```
<%-- Имя программы:  AnyQuery.jsp
  -- Назначение:     Принять и исполнить любой пользовательский
  --                 запрос, обращенный к базе данных заказов на
  --                 покупку, отобразив результаты в виде
  --                 HTML-таблицы.
  --%>

<jsp:useBean id="queryBean" class="mybeans.DynamicQueryBean"
scope="session" />
<jsp:setProperty name="queryBean" property="*" />

<HTML>
<HEAD> <TITLE> The AnyQuery JSP </TITLE> </HEAD>
```

```
<BODY BGCOLOR="white">

<%   String query = request.getParameter("query");
     if (query != null) { %>
         <H3>Result of your query: </H3>
         <CENTER><B>
         <%= queryBean.getResult() %>
         </CENTER><B><BR>
         <HR>
<%   }   %>

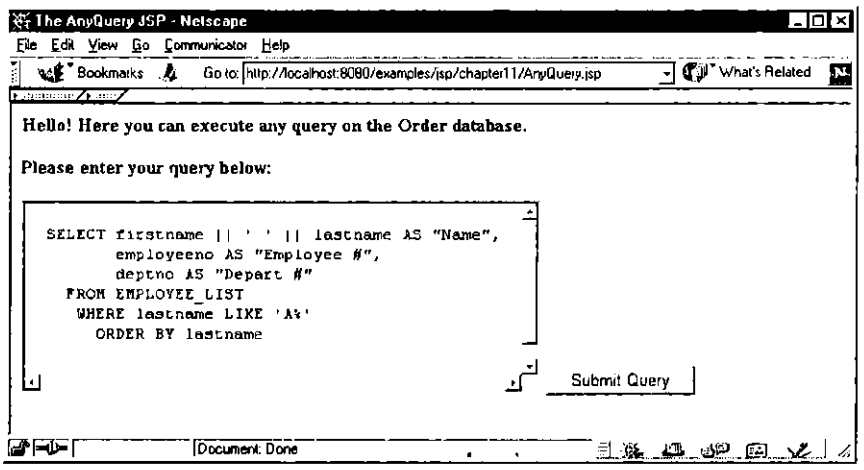
<H4> Hello! Here you can execute any query on the
           Order database. </H4>
<H4>Please enter your query below:</H4>

<FORM METHOD=post>
<TEXTAREA NAME="query" rows="8" cols="50">
<% if (query != null ) { %>
<%= query %>
<% } else { %>
   SELECT  firstname || ' ' || lastname AS "Name",
           employeeno AS "Employee #",
           deptno AS "Depart #"
      FROM EMPLOYEE_LIST
       WHERE lastname LIKE 'A%'
         ORDER BY lastname
   <% } %>
</TEXTAREA>
<INPUT TYPE="submit" VALUE="Submit Query">
</FORM>
</BODY>
</HTML>
```

Выполнение AnyQuery.jsp

На рис. 11.15 показан результат первого вызова страницы AnyQuery.jsp. На экране присутствуют текстовое окно для ввода запроса, а также запрос по умолчанию, обращенный к таблице EMPLOYEE_LIST базы данных заказов на покупку.

Рис. 11.15. Начальные выходные данные AnyQuery.jsp



Модифицируем запрос по умолчанию: изменим условие поиска в конструкции WHERE оператора SELECT на lastname LIKE M%. После щелчка мыши на кнопке "Submit Query" вызывается метод getResult() зерна DynamicQueryBean. Этот метод при помощи JDBC исполняет текст запроса, введенный во входном окне. Результат форматируется в виде HTML-таблицы и посылается обратно браузеру (см. рис. 11.6). Заметим, что запрос по умолчанию в текстовом окне заменен модифицированным запросом, который был передан.

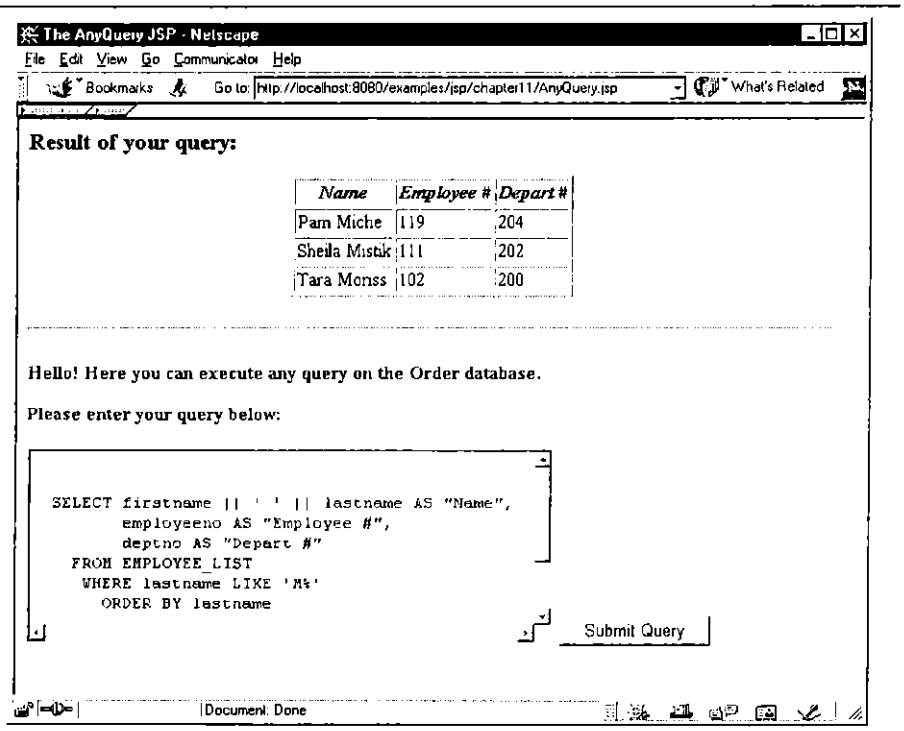


Рис. 11.16. Выходные данные `AnyQuery.jsp` после исполнения запроса

Зерна доступа к базам данных в Oracle JSP 1.1

В Oracle JSP 1.1 предлагается набор зерен доступа к базам данных – заранее построенных компонентов, исполняющих SQL-операции в web-приложении. Ниже описываются эти зерна и создается JSP-страница, использующая подобное зерно для исполнения SQL-запроса и генерации результатов в XML.

Библиотека зерен Oracle JSP

В библиотеке зерен Oracle JSP содержатся четыре зерна JavaBean. Их можно создавать и применять в разных областях действия JSP-страницы при помощи тэга `<jsp:useBean>`. Для соединения с базой данных и исполнения SQL-операции они используют JDBC, однако многие детали SQL-исполнения скрываются зернами, и в результате формируется интерфейс связи с базой данных, более простой, чем JDBC. Например, зерно для управления соединениями может автоматически закрывать соединение с базой данных при выходе за пределы области своего действия (`page`, `request`, `session` или `application`). Таким образом, соединение всегда закрывается надлежащим образом, и удается избежать ошибок со стороны программиста. Одно и то же зерно соединений можно использовать в разных областях действия, поскольку класс зерна реализует интерфейс `oracle.jsp.event.JspScopeListener` – общий механизм приемников событий, поддерживаемый в Oracle JSP для разных областей действия.

В пакете `oracle.jsp.dbutil`, входящем в состав библиотек Oracle JSP 1.1 (`ojsputil.jar`), доступны следующие зерна:

- `ConnBean` Используется для установления простого соединения с базой данных без пулинга соединений.
- `ConnCacheBean` Использует средства кэширования соединений Oracle для эффективного разделения соединений с базой данных среди нескольких пользователей.
- `DBBean` Применяется для исполнения запроса, обращенного к базе данных, и для вывода результатов или в виде HTML-таблицы, или с помощью XML.

- CursorBean Может вызывать хранимые процедуры и исполнять SQL-операторы DML (UPDATE, INSERT и DELETE), а также SQL-запросы.

В следующем разделе говорится о применении DBBean. Это зерно может подключаться к базе данных и исполнять SQL-операции. Оно эффективно управляет соединением, закрывая его автоматически при выходе из области своего действия. Другим полезным свойством DBBean является возможность получения результата SQL-запроса в формате XML. Ниже предлагается обзор XML, а затем создается JSP-страница, использующая DBBean для генерации XML-текста.

Затем будет приведен пример стандартного зерна ConnCacheBean, которое используется для управления разделяемым пулом соединений.

Генерация XML-страниц с помощью JSP-страниц и зерен JavaBean

Спецификация JSP поддерживает создание XML-страниц при помощи страниц JSP. Для каждого JSP-тэга существует обозначение, совместимое с XML (см. главу 10), поэтому JSP-страницы можно создавать, применяя удобные инструментальные средства XML. Выходные данные, генерируемые JSP-страницей, также могут быть XML-текстом. Что же такое XML и для чего его используют?

Общее представление о XML

XML расшифровывается как *Extensible Markup Language* (расширяемый язык разметки). Он основан на стандартном обобщенном языке разметки (SGML, Standard Generalized Markup Language), который применялся на протяжении ряда лет для создания документов общего характера. XML – упрощенный вариант SGML, в котором устранены некоторые сложные свойства. Тем не менее XML во многом сохранил мощь SGML, будучи расширяемым (в отличие от HTML, который имеет предопределенный набор тэгов). Это значит, что в документе XML можно описать свои собственные тэги (называемые также *элементами* (element)) и синтаксические правила для них, воспользовавшись *описанием типа документа* (DTD, document type definition). DTD можно применять для проверки XML-документа на "организованность" и на достоверность при помощи *синтаксического анализатора* (parser) XML – программы, которая считывает XML-документ и применяет синтаксические правила по отношению к конструкции элементов.

Тэги в XML используются для описания иерархической структуры документа, которая может иметь произвольную глубину. Например, заказ на покупку можно представить в XML следующей структурой с вложениями:

```
<order>
  <employeeInfo>
    <employeeName> name </employeeName>
    <employeeNumber> 1234 </employeeNumber>
  </employeeInfo>
  <itemList>
    <item>
      <itemNumber> 1 </itemNumber>
      <itemName> XXX </itemName>
      <quantity> 123 </quantity>
      <description> This item is for XXX purpose. </description>
    </item>
    <item>
      <itemNumber> 2 </itemNumber>
      <itemName> YYY </itemName>
      <quantity> 345 </quantity>
      <description> This item is for YYY purpose. </description>
    </item>
  </itemList>
</order>
```

Как видим, XML-документ имеет иерархическую, или древовидную, структуру. Тэг "верхнего" уровня <order> представляет элемент заказа на покупку. Он завершается соответствующим конечным тэгом </order>. В элементе заказа присутствуют два вложенных элемента, employeeInfo и itemList, которые в свою очередь содержат другие субэлементы. Отметим, что элемент item повторяется в тэге itemList для каждого пункта (товара) в заказе на покупку. Элемент типа quantity – это простое целое число без субэлементов. Такие элементы называются *листовыми* (leaf), они хранят реальные данные в XML-документе. *Нелистовые* (non-leaf) элементы описывают структуру с вложениями и могут содержать внутри себя другие листовые и нелистовые элементы.

В отличие от HTML, XML-документ не всегда содержит информацию форматирования, т.е. в XML нет описания шрифтов или цветов, с помощью которых следует отображать элемент. Это позволяет отделить содержимое от средств представления. Возникает вопрос: как же отображается XML-документ браузером? Для этого служит приложение *таблиц стилей* (style sheets) XSL. XSL расшифровывается как *Extensible Stylesheet Language* (расширяемый язык таблиц стилей), и его спецификация дополняет стандарт XML. XSL – мощный инструмент преобразования и форматирования древовидной структуры с правилами соответствия шаблонам, применяемый для обработки XML-элементов и для описания формата их визуального представления в браузере.

Благодаря своей гибкости и расширяемости, XML быстро завоевал огромную популярность. Теперь это стандарт W3C (World Wide Web Consortium – ассоциация World Wide Web; домашний сайт www.w3.org). Технология XML применяется все большим и большим числом компаний для самых разных целей: от представления структурированных данных до протоколов передачи сообщений. Вне всякого сомнения, в будущем ее использование будет расширяться.

Создание XMLQuery.jsp

Интересным аспектом приложений баз данных является способ генерации структурированного XML-представления результатов SQL-запроса. В Oracle предлагается несколько утилит для подобной XML-генерации и для работы с XML-документами. XML-утилиты Oracle скомпонованы в виде инструментария разработчика Oracle XML Developer's Kit (XDK), который можно загрузить с web-сайта сети Oracle Technology Network (technet.oracle.com/tech/xml). В состав XDK входят синтаксический анализатор XML, XSL-процессор и утилита Oracle XML-SQL для генерации XML-данных из результирующих множеств JDBC. Утилита Oracle XML-SQL используется зернами доступа к базам данных (например, DBBean) для прозрачного преобразования SQL-результатов в XML, когда скрываются многие детали XML-генерации. Рассмотрим пример JSP-страницы, генерирующей XML-результаты для SQL-запроса при помощи DBBean.

Приводимая ниже JSP-программа XMLQuery.jsp использует экземпляр класса DBBean (см. выше) с областью действия page для исполнения запроса, вводимого в текстовом окне в HTML-форме. Важные участки программы выделены жирным шрифтом.

```
<%-- Имя программы: XMLQuery.jsp
  -- Назначение: Сгенерировать XML-результаты для SQL-запроса.
  --%>
%@ page import="java.sql.*" %>
<jsp:useBean id="dbbean" class="oracle.jsp.dbutil.DBBean"
scope="page">
<jsp:setProperty name="dbbean" property="user" value="jspuser"/>
<jsp:setProperty name="dbbean" property="password" value="jsp"/>
<jsp:setProperty name="dbbean" property="URL"
value="jdbc:oracle:oci8:@"/>
</jsp:useBean>

<HTML>
```

```
  <HEAD> <TITLE> XMLQuery JSP </TITLE> </HEAD>
 <BODY BGCOLOR ="white">
<%   String query = request.getParameter("query");
     if (query != null) { %>
       <H3>Result of your query in XML: </H3>
 <P>
<%   dbbean.connect();   /* подключимся к базе данных */ %>
<PRE> <%= dbbean.getResultAsXMLString(query) %> </PRE>
<%   dbbean.close();     /* закроем соединение с базой данных */
     } %>
<H4>Please enter your query below:</H4>
  <FORM METHOD=post>
   <TEXTAREA NAME="query" rows="8" cols="50">
   <% if (query != null ) { %>
   <%= query %>
   <% } else { %>
     SELECT firstname ||    ' || lastname AS "Name",
            employeeno AS "Employee #",
            deptno AS "Depart #"
       FROM EMPLOYEE_LIST
      WHERE lastname LIKE 'A%'
      ORDER BY lastname
     <% } %>
   </TEXTAREA>
   <INPUT TYPE="submit" VALUE="Submit Query">
   </FORM>
 </BODY>
</HTML>
```

Зерно DBBean имеет три свойства: user, password и URL, которые, как правило, обязательны для установления соединения с базой данных. Эти свойства задаются обычно вызовами <jsp:setProperty> или, что реже, методами set напрямую. Если в параметрах запроса (request) есть SQL-запрос (query), соединение с базой данных устанавливается при помощи метода dbbean.connect(). Для исполнения SQL-запросов применяются два метода этого зерна с сигнатурами:

- ```
  String getResultAsHTMLTable(String queryText);
  String getResultAsXMLString(String queryText);
  ```

Первый метод исполняет запрос и форматирует строки в виде HTML-таблицы (используя алгоритм, схожий с алгоритмом описанного выше зерна DynamicQueryBean). Имена или псевдонимы SQL-столбцов становятся заголовками столбцов HTML-таблицы. Второй метод также исполняет запрос, но форматирует строки в XML-виде. Имена или псевдонимы SQL-столбцов становятся тэгами XML.

В программе XMLQuery.jsp для исполнения запроса пользователя над базой данных заказов на покупку вызывается второй метод — getResultAsXMLString(), генерирующий выходные данные XML. В заключение соединение с базой данных закрывается методом dbbean.close(). Кроме того, экземпляр DBBean закрывает соединение автоматически (если оно еще не закрыто), когда исполнение JSP-страницы завершается и он выходит из области своего действия.

Выполнение XMLQuery.jsp

Установите JSP-страницу на web-сервере и не забудьте добавить библиотеку ojsputil.jar из Oracle JSP 1.1, а также xsu12.jar (для JDK 1.2) или xsu111.jar (для JDK 1.1) в CLASSPATH. Последние библиотеки входят в состав Oracle XDK (который можно загрузить с сайта technet.oracle.com сети Oracle Technology Network), Oracle 8.1.7 и Oracle Internet Application Server.

При вызове XMLQuery.jsp создается HTML-страница с встроенным в нее XML-блоком. Исходный текст страницы, генерируемой при исполнении XMLQuery.jsp, приведен в листинге 11.2, участки с XML выделены жирным шрифтом.
  ```

Листинг 11.2. HTML/XML-страница, генерируемая программой XMLQuery.jsp

```
<HTML>
  <HEAD> <TITLE> XMLQuery JSP </TITLE> </HEAD>
 <BODY BGCOLOR="white">
       <H3>Result of your query in XML: </H3>
  <P>
 <?xml versions="1.0" ?>
 <ROWSET>
 <ROW num="1">
  <Name>Pam Miche</Name>
  <Employee #>119</Employee #>
  <Depart #>204</Depart #>
 </ROW>
 <ROW num="2">
  <Name>Sheila Mistik</Name>
  <Employee #>111</Employee #>
  <Depart #>202</Depart #>
 </ROW>
 <ROW num="3">
  <Name>Tara Moriss</Name>
  <Employee #>102</Employee #>
  <Depart #>200</Depart #>
 </ROW>
 </ROWSET>
 <H4>Please enter your query below:</H4>
 <FORM METHOD=post>
  <TEXTAREA NAME="query" rows="8" cols="50">

    SELECT firstname ||     || lastname AS "Name",
          employeeno AS "Employee #",
          deptno AS "Depart #"
       FROM EMPLOYEE_LIST
        WHERE lastname LIKE 'M%'
          ORDER BY lastname
    </TEXTAREA>
  <INPUT TYPE="submit" VALUE="  ubmit Query">
  </FORM>
 </BODY>
</HTML>
```

На этой странице, генерируемой программой XMLQuery.jsp, оператор <?xml version="1.0" ?> отмечает начало XML-данных. Элемент "верхнего" уровня, формируемый для результата, по умолчанию называется ROWSET, а каждая вложенная строка — ROW. Строки нумеруются по порядку, для чего используется атрибут num элемента ROW. Это режим работы по умолчанию служебных подпрограмм Oracle XML-SQL.

Браузер (например, Netscape 4.6), возможно, не отобразит какие-то из XML-тэгов на генерируемой странице, так как не понимает обозначения XML. Для форматирования XML-результата в виде HTML-таблицы для браузера используются таблицы стилей XSL. Таблицу стилей можно применить несколькими способами. Одним из способов, причем весьма удобным, является использование специального тэга <jml:transform> из Oracle JSP 1.1 (см. главу 12).

Оптимизация SQL-операций

Выше говорилось об эффективном использовании в web-приложении зерен JavaBean для программирования операций, выполняемых над базой данных. В настоящем разделе рассматриваются вопросы, имеющие отношение к производительности исполнения SQL-операций в web-среде.

Web-приложения уникальны тем, что нередко одновременно обслуживают тысячи запросов в реальном времени. Время реакции web-приложения крайне важно, поскольку оно взаимодействует с человеком, который с нетерпением ожидает ответа. Это уникальное требование предполагает, что для построения web-приложения должны использоваться особые методы проектирования. Обсудим ряд важных вопросов, связанных с производительностью, и методы проектирования, взяв за основу исполнение SQL-операций в web-среде.

Использование готовых JDBC-операторов

Если при каждом вызове приложение исполняет одну и ту же SQL-операцию с разными параметрами, постоянное повторение анализа и оптимизации SQL-оператора неэффективно. План исполнения SQL-оператора, как правило, не зависит от параметров привязки, которые для SQL-операции являются обычными аргументами. Можно сэкономить время, подвергнув SQL-оператор анализу лишь однажды, а затем исполнять его снова и снова с разными параметрами привязки. Именно такой алгоритм действия обеспечивает JDBC-тип `PreparedStatement`. Тип `java.sql.PreparedStatement` – это подтип типа `java.sql.Statement`, предоставляющий методы привязки параметров для повторного исполнения.

Исполняющая система SQLJ тоже пользуется готовыми операторами JDBC. В Oracle 8.1.6 и более старших версий они кэшируются и повторно применяются для исполнения в цикле. Итак, всякий раз *при использовании SQLJ автоматически применяются готовые операторы JDBC*, что весьма полезно.

В качестве примера рассмотрим SQL-оператор, вводящий 100 пунктов ассортимента в таблицу `LINEITEM_LIST` базы данных заказов на покупку. Один и тот же оператор `INSERT` необходимо выполнить в цикле 100 раз с собственными данными для каждой строки. Как в этом случае используются готовые JDBC-операторы, показывает программный фрагмент, приведенный в листинге 11.3.

Листинг 11.3. Использование готовых JDBC-операторов

```
public void insertLineItems(java.sql.Connection conn)
  throws java.sql.SQLException {
    java.sql.PreparedStatement pstmt =
        conn.prepareStatement( // подготовим оператор ввода один раз
        "INSERT INTO LINEITEM_LIST (requestno, lineno, projectno, " +
                                   " quantity, description) VALUES( " +
                                   " ?, ?, ?, ?, ?)" );
    for (int i=1; i<100; i++) { // введем 100 строк

    // Привяжем значение параметров
    pstmt.setInt(1,requestno);
    pstmt.setInt(2, i);
    pstmt.setInt(3, projectno);
    pstmt.setInt(4, quantity[i]);
    pstmt.setString(5, description[i]);
    pstmt.executeUpdate();          // введем строку
  }
  pstmt.close();
}
```

Отметим, что объект готового оператора создается только один раз методом `prepareStatement()` JDBC-соединения с указанием SQL-команды в качестве аргумента. Символ "?" в тексте SQL-оператора позиционно представляет каждый из параметров привязки. SQL-операция `INSERT` выполняется в цикле для ввода 100 строк в таблицу `LINEITEM_LIST`. В каждой операции ввода используются собственные значения для столбцов `lineno`, `quantity` и `description`. Соответствующие значения привязываются к заполнителям "?" готового оператора с помощью методов `setInt()` и `setString()`. Тип `PreparedStatement` предоставляет вызовы `set` для всех типов данных, поддерживаемых в JDBC.

После привязки параметров готовый оператор исполняется. В последующих итерациях цикла повторный анализ SQL-команды выполняться не будет.

Пакетное обновление

В Oracle 8.1.6 и более старших версий SQL-операции `INSERT`, `DELETE` и `UPDATE` можно объединять в *пакеты* (batches) JDBC 2.0 и SQLJ. Пакетирование оптимизирует работу следующим образом: при вводе или обновлении данных циклически одним оператором DML операции привязки параметров можно группировать локально на стороне клиента, пока пакет не будет послан на сервер методом `executeBatch()`. При этом оператор DML исполняется один раз с привязкой массива скомпонованных значений параметров. Следовательно, пакетирование сокращает число обращений к базе данных. В API JDBC 2.0 существуют явные методы объекта `java.sql.Statement`, позволяющие добавлять строки в пакет и исполнять его.

Пакетирование поддерживается и в SQLJ за счет использования *контекстов исполнения* (execution contexts). Определенный контекст исполнения указывается в операторе `#sql` и описывает среду: характеристики пакетирования и другие параметры исполнения SQL-операции. Для установки параметров исполнения, в частности для разрешения пакетирования и для задания временного лимита пакета, вызываются методы экземпляра контекста исполнения. Ожидающий пакет можно послать базе данных явным образом, вызвав метод `executeBatch()` контекста исполнения, либо он посылается неявно исполняющей системой SQLJ при достижении указанного размера пакета или при исполнении программой другого SQLJ-оператора. Таким образом, в программах SQLJ можно пользоваться всеми преимуществами средств пакетирования JDBC 2.0.

Кэширование описателей операторов

В Oracle SQLJ (Oracle 8.1.6 и выше) обеспечивается автоматическое кэширование подготовленных в JDBC описателей операторов. Другими словами, объекты готовых операторов для SQLJ-операторов программы SQLJ сохраняются и многократно используются, пока открыто соединение с базой данных.

JDBC-драйверы Oracle 8.1.7 также обеспечивают прозрачное кэширование операторов, основой которого является схема "least recently used" (использовавшийся наиболее давно). Когда в JDBC разрешено кэширование операторов, закрытие объекта `Statement` не всегда влечет за собой закрытие основного курсора базы данных. Вместо этого он кэшируется, что позволяет использовать его впоследствии при исполнении такого же SQL-оператора. Альтернативой этой схеме является применение SQL-зерна с областью действия `session`, которое явно применяет описатель готового оператора, не закрывая его в течение HTTP-сеанса.

Пулинг соединений

Пулинг (pooling) соединений – важная технология, позволяющая оптимизировать обращение к базе данных со стороны нескольких сеансов web-приложения. Идея состоит в совместном использовании физических соединений из общего кэша, что сводит к минимуму затраты на создание и закрытие соединений с базой данных, т.е. на операции, которые, как правило, довольно ресурсоемки. Закрытие "логического" соединения с базой данных на самом деле не завершает его, а возвращает физическое соединение в разделяемый пул. Пулинг соединений распространен в среде прикладных серверов, функционирующих на среднем уровне, и служит для оптимизации доступа к базе данных.

В API JDBC 2.0 введено понятие основанных на JNDI *источников данных* (data sources) для подключения к базам данных (в качестве альтернативы JDBC-схеме `DriverManager` в JDBC 1.0). Кроме того, в JDBC 2.0 описана стандартная схема пулинга соединений посредством `javax.sql.ConnectionPoolDataSource` и связанных с ним интерфейсов. Эти интерфейсы реализуются различными

JDBC-драйверами, обеспечивая эффективный пулинг соединений. Драйверы JDBC 2.0 в Oracle также поддерживают интерфейсы пулинга соединений. За дополнительными сведениями об источниках данных и пулинге соединений обращайтесь к руководству "Oracle8*i* JDBC Developer's Guide and Reference, Release 8.1.6" корпорации Oracle.

Использование пулинга соединений в JSP

Средствами пулинга соединений JDBC 2.0 можно воспользоваться и в JSP-приложении. Разделяемый пул соединений с базой данных (кэш соединений) связывается с объектом `application` и применяется всеми HTTP-сеансами одной и той же виртуальной машины Java. Такой подход особенно выгоден, если разные HTTP-сеансы пользуются одной и той же схемой базы данных, как, например, в рассмотренном выше приложении Order Online. Кроме того, в кэше могут присутствовать соединения и для других схем. В этом случае объект соединения с нужной схемой возвращается на основе имени, пароля пользователя и информации URL базы данных.

Библиотека зерен в Oracle JSP 1.1 содержит стандартный компонент JavaBean с именем `ConnCacheBean`, предоставляющий возможности кэширования соединений в JSP. Это зерно можно применять в области действия `application` web-приложения, эффективно управляя разделяемым пулом физических соединений с базой данных. Зерно обладает следующими свойствами:

- `user` Имя схемы в базе данных.
- `password` Пароль схемы.
- `URL` URL базы данных для установления JSP-соединения.
- `maxLimit` Максимальное число открытых соединений с базой данных в кэше.
- `minLimit` Минимальное число открытых соединений с базой данных в кэше.
- `stmtCacheSize` Размер кэша JDBC-операторов Oracle (по умолчанию операторы не кэшируются).
- `cacheScheme` Политика для управления кэшированием. Это свойство может иметь одно из следующих значений: `DYNAMIC_SCHEME`, `FIXED_WAIT_SCHEME` или `FIXED_RETURN_NULL_SCHEME`.

Эти схемы кэша реализуют различные режимы работы в ситуации, когда превышается максимально допустимое число открытых соединений кэша. В схеме `DYNAMIC_SCHEME` разрешается, чтобы число соединений пула было больше максимального числа, но все лишние физические соединения закрываются, если свободно логическое. В схеме `FIXED_WAIT_SCHEME` при достижении максимального числа открытых соединений запрос на новое соединение не устанавливает его, а ждет, пока не освободится существующее. В схеме `FIXED_RETURN_NULL_SCHEME` при достижении максимального числа возвращается `null`-соединение, пока не освободятся какие-либо соединения.

Класс `oracle.jsp.dbutil.ConnCacheBean` расширяет класс Oracle JDBC с именем `oracle.jdbc.pool.OracleConnectionCacheImpl`, который, в свою очередь, расширяет класс Oracle JDBC с именем `oracle.jdbc.pool.OracleDataSource`. Класс `ConnCacheBean` наследует ряд свойств и их методы доступа из класса `oracle.jdbc.pool.OracleDataSource`: `databaseName`, `dataSourceName`, `description`, `networkProtocol`, `portNumber`, `serverName` и `driverType`. За дополнительными сведениями об этих свойствах и семантике их использования обращайтесь к руководству "Oracle8*i* JDBC Developer's Guide and Reference, Release 8.1.6" корпорации Oracle.

Применение `ConnCacheBean` иллюстрируется программой `ConnectionCaching.jsp`, приведенной в листинге 11.4. Сначала JSP устанавливает свойства `MaxLimit`, `CacheScheme`, `user`, `password` и `URL` в нужные значения. В данной программе это постоянные строки символов, но они могут устанавливаться и на основании параметров HTTP-запроса. Метод `connCacheBean.getConnection()` вызывается для получения логического соединения в кэше. После этого создается

оператор и исполняется запрос. После печати результатов запроса результирующее множество и объекты оператора закрываются. Соединение закрывается вызовом метода ConnCacheBean.close(), но фактически этот метод возвращает в кэш лишь описатель соединения.

Листинг 11.4. Использование ConnCacheBean в JSP-программе

```
<!-- Имя программы:  ConnectionCaching.jsp
  -- Назначение:     Использовать ConnCacheBean из библиотеки зерен
  -- OracleJSP,      чтобы подключиться к базе данных и выполнить
  --                 SQL-запрос.
  -->
<%@ page import="java.sql.*" %>
<%@ page import="javax.sql.*" %>
<%@ page import="oracle.jsp.dbutil.ConnCacheBean" %>

<jsp:useBean id="ConnCacheBean"  class="oracle.jsp.dbutil.ConnCacheBean"
                                 scope="application">
   <jsp:setProperty name="ConnCacheBean" property="MaxLimit"
                                         value="50" />
   <jsp.setProperty name="ConnCacheBean" property="CacheScheme"
         value="<%= ConnCacheBean.FIXED_RETURN_NULL_SCHEME %>" />
   <jsp:setProperty name="ConnCacheBean" property="user"
         value="jspuser"/>
   <jsp:setProperty name="ConnCacheBean" property="password"
         value="jsp"/>
   <jsp:setProperty name="ConnCacheBean" property="URL"
                    value="jdbc:oracle:oci8:@" />
</jsp:useBean>
<HTML>
   <HEAD> <TITLE> The Connection Caching JSP </TITLE> </HEAD>
   <BODY BGCOLOR ="white">
<%
         // получим соединение в кэше
         Connection conn = ConnCacheBean.getConnection();
         // создадим оператор
         Statement stmt = conn.createStatement();
         // выполним запрос
         ResultSet rset = stmt.executeQuery (
                           "SELECT * FROM EMPLOYEE_LIST" );
         if (rset.next()) {
%>
           <!-- распечатаем результаты запроса -->
           Result of your query:
           ... ...
<%       } else { %>
           <P> No rows found that match your query. </P>
<%       }
         rset.close(); // закроем результирующее множество
         stmt.close(); // закроем оператор
         conn.close(); // вернем соединение в пул
     %>
   </BODY>
</HTML>
```

Кэширование результатов запроса

В ряде случаев можно многократно применять результаты SQL-запросов в разных HTTP-сеансах или распределять их среди разных HTTP-сеансов. В качестве примера рассмотрим web-сайт, предлагающий некие товары в течение недели. Запрос, выясняющий, какие товары продаются, можно распределить между разными пользователями и сеансами. Кэширование, аналогичное механизму пулинга соединений, можно организовать при помощи зерен SQL-запросов

с соответствующими областями действия (session или application) и тэга <jsp:useBean>. Зерно будет кэшировать результаты запроса после его обработки. Клиенты экземпляра зерна смогут все вместе пользоваться этим результатом, не исполняя запроса.

Процесс кэширования результатов запроса предполагает также проверку их достоверности и их обслуживание. Другими словами, если изменяются базовые данные или истекает допустимый срок действия результата, его нужно обновить, выполнив запрос еще раз. Автоматическая отмена или регенерация кэшируемых результатов запроса после обновления данных — сложный вопрос, требующий применения триггеров базы данных. Реализация подобной функции каким-то скрытым образом, без явного программирования, может оказаться неэффективной. Однако можно предоставить программисту API, к которому тот будет обращаться, указывая свои требования в отношении кэширования. Во время издания этой книги в Oracle работали над подобным "программируемым" функциональным средством web-кэширования для собственных сред сервлетов и JSP. Возможно, что в будущей редакции Oracle JSP такое средство будет присутствовать.

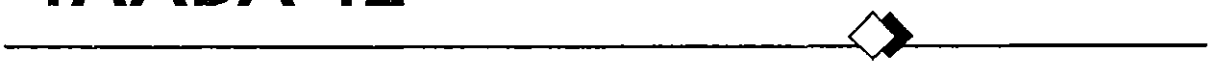
Итоги

В этой главе рассматривалось использование компонентов JavaBean в JSP-программах. JavaBean — универсальный класс Java, который соответствует особой структурной схеме, позволяющей без труда манипулировать ее свойствами и интегрироваться с JSP. JavaBean можно применять в качестве модульного Java-компонента для исполнения динамических логических конструкций. В спецификации JSP определены специальные встроенные тэги, упрощающие работу с зернами JavaBean и не только четко отделяющие HTML-текст от программных конструкций Java, но и предоставляющие средства для их интеграции. Таким образом, web-дизайнеры могут сосредоточивать все свои усилия на написании удобных для пользователя JSP-страниц с помощью HTML или XML, а Java-программисты — на разработке зерен JavaBean. Впоследствии динамическую логику можно "вставить" в JSP, используя для этого простые, но эффективные тэги зерен.

Мы создали программы с использование JSP-страниц и зерен JavaBean. Рассмотрели разнообразные вопросы, связанные с написанием компонентов JavaBean, причем особое внимание уделялось приложениям баз данных. В частности, были созданы классы JavaBean, к их свойствам применялись методы set и get, после чего классы связывались с различными областями действия JSP. Было разработано JSP-приложение с контролем за сеансами для системы ввода заказов и со специальной "корзиной". Это приложение вводит передаваемый заказ в базу данных заказов на покупку, используя в компоненте JavaBean SQLJ-конструкции. Помимо этого, мы создали зерно JavaBean, выполняющее статический запрос при помощи SQLJ и отображающее его результаты в виде HTML-таблицы. Для динамических SQL-запросов, неизвестных до этапа исполнения программы, было написано JavaBean с применением JDBC, причем результаты форматировались в виде HTML-таблицы с помощью SQLJ-объекта ResultSetMetaData. Кроме того, в этой главе рассказывалось о генерации XML-текста из SQL-таблиц базы данных с помощью компонентов JavaBean из библиотеки зерен Oracle JSP. В заключение мы узнали о том, как оптимизировать производительность доступа к базе данных из web-приложения.

В главе 12 говорится об использовании компонентов EJB и CORBA для модульной разработки JSP-приложений. Зерна JavaBean можно применять для повышения качества доступа к объектам EJB и CORBA, сводя объем Java-конструкций на JSP-странице к минимуму. Кроме того, описывается использование библиотек тэгов, содержащих специальные тэги для расширения JSP. Эти отличные друг от друга технологии можно объединять, создавая эффективные и универсальные web-приложения.

ГЛАВА 12



Использование JSP вместе с EJB, CORBA и библиотеками тэгов

Обычно web-приложение состоит из самых различных ресурсов: из статических элементов, например HTML-страниц и GIF-изображений, и из динамических модулей, таких как JSP-страницы и сервлеты, зерна JavaBean, зерна Enterprise JavaBean и объекты CORBA. Динамические компоненты комбинируются друг с другом, что позволяет создавать мощные приложения уровня предприятия.

В двух предыдущих главах говорилось об использовании JSP вместе с другими JSP-страницами и сервлетами, а также о применении в JSP-программе компонентов JavaBean. В части II рассказывалось о зернах EJB, о том, как внедрять их в Oracle8*i* и вызывать из клиентской программы. В части III показано, как описывать и внедрять компоненты CORBA в Oracle8*i*. В этой главе рассматривается применение всех этих методов компонентного программирования в основанном на JSP web-приложении.

В спецификацию JSP 1.1 введено понятие расширения набора JSP-тэгов при помощи *библиотек тэгов* (tag libraries). Схема расширения тэгов является модульной и переносимой, позволяющей описывать и применять специальные тэги, удобно упакованные в библиотеку тэгов. Основываясь на этой схеме, можно описать свой собственный тэг, характеризующий некоторое действие в компоненте. Можно также воспользоваться библиотеками тэгов, предлагаемыми различными поставщиками. Библиотеки тэгов вместе с другими компонентами Java, например с зернами JavaBean и EJB, используются на JSP-страницах для построения сложных модульных web-приложений.

В подробных примерах этой главы показано, как:

- Вызывать EJB из JSP-приложения
- Вызывать компонент CORBA с JSP
- Исполнять SQL-запросы с помощью библиотеки JSP-тэгов Oracle
- Используя библиотеку JSP-тэгов Oracle, генерировать XML-результаты для SQL-запроса и преобразовывать их при помощи таблиц стилей XSL
- Создавать свой собственный специальный JSP-тэг для вывода текущих даты и времени
- Создавать свой собственный специальный JSP-тэг для исполнения SQL-запроса и создания HTML-меню элементов

Таким образом, вы научитесь в полной мере пользоваться всей мощью и всеми возможностями модели компонентного программирования в приложениях JSP.

Разработка web-приложений с помощью EJB-компонентов

Enterprise JavaBean — важная составляющая компонентной модели в архитектуре программирования Java. В части II этой книги говорилось о создании и внедрении зерен EJB в базе данных Oracle8*i*. Теперь посмотрим, как вызывать EJB-компоненты с JSP-страниц и из сервлетов. В зерне EJB концентрируется бизнес-логика, в то время как JSP и сервлет содержат "фронтальную", внешнюю логику для интерактивного ввода данных, организуют вызов EJB и формируют логику представления результата. Сначала реализуем EJB-компонент, а потом создадим использующие его JSP-страницы и зерна JavaBean.

Сеансовое EJB для заказов

Создадим простое сеансовое EJB-зерно для обработки заказов, обращенных к базе данных Purchase Order (см. введение). Напомним, что в главе 11 было написано зерно JavaBean — `OBQueryBean.sqlj` — для запроса информации о заказах в базе данных. Теперь создадим программу со схожими функциями,

применив не JavaBean, а EJB. О сеансовых зернах EJB говорилось в части II. По сравнению с JavaBean, зерно EJB гораздо мощнее и гибче с точки зрения управления транзакциями и организации безопасности в бизнес-логике, хотя нередко его сложнее программировать и внедрять. Использовать JavaBean, исполняющее SQL-запросы, удобнее тогда, когда выполняемые над данными операции достаточно просты; более сложная бизнес-логика и рабочие сценарии требуют применения EJB.

Ниже приводятся операции, необходимые для разработки сеансового зерна `OrderEJB` и внедрения его в Oracle8*i*. Если вы еще не знакомы с моделью EJB, обратитесь к части II книги, где рассказано о структуре зерен EJB и о способах их программирования.

1. Опишем домашний интерфейс `OrderHome` для `OrderEJB`.
2. Опишем удаленный интерфейс `Order` для `OrderEJB`.
3. Создадим Java-класс `OrderHeader` для получения информации "высшего" уровня о заголовках заказа.
4. Создадим Java-класс `LineItem` для получения сведений о пунктах заказа.
5. Создадим класс реализации для `OrderEJB`.
6. Опишем дескриптор внедрения EJB.
7. Внедрим EJB в Oracle8*i*.

Эти операции рассматриваются в последующих разделах. После создания и внедрения EJB поговорим о том, как вызывать его из web-приложения.

Создание домашнего интерфейса

Для сеансового зерна `OrderEJB` нужно описать домашний интерфейс `OrderHome`. Ниже приводится его программный текст. Он расширяет стандартный интерфейс `EJBHome` и содержит метод `create()`, вызываемый для создания экземпляра EJB.

```
/** Имя программы:   OrderHome.java
  * Назначение:      Домашний интерфейс для OrderEJB
  **/

package order;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface OrderHome extends EJBHome {
    public Order create()
        throws CreateException, RemoteException;
}
```

Создание удаленного интерфейса

Теперь опишем удаленный интерфейс `Order` для `OrderEJB`. Он состоит из двух методов: `getOrder()` и `getItems()`. Метод `getOrder()` возвращает заголовочную информацию: имена и число служащих, разместивших заказ, и общее число пунктов заказа. Эта информация концентрируется в Java-классе с именем `OrderHeader` (см. ниже). Метод `getItems()` возвращает характеристики заказываемого товара; выдается значение, представляемое массивом объектов `LineItem`. Класс `LineItem` будет создан ниже.

Метод `getOrder()` вызывается первым, в результате чего номер заказа записывается в EJB (с сохранением состояния). Затем вызывается метод `getItems()` для получения характеристик пункта ассортимента. Программный текст интерфейса `Order`:

```
/** Имя программы:   Order.java
  * Назначение:      Удаленный интерфейс для OrderEJB
  **/
```

```
package order;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Order extends EJBObject {

    public OrderHeader getOrder (int orderNumber)
        throws java.sql.SQLException, RemoteException;

    public LineItem [] getItems()
        throws java.sql.SQLException, RemoteException;
}
```

Создание класса Java для заголовочной информации заказа

Теперь опишем Java-класс OrderHeader. Он предназначен для хранения информации высшего уровня, связанной с заказом. Экземпляр этого класса будет использоваться сеансовым зерном OrderEJB для возвращения клиенту результатов запроса заказа. Поля empName и empNum описывают имя и номер служащего, размещающего заказ, а itemCount – общее число заказанных товаров. Программный текст OrderHeader.java:

```
/** Имя программы:   OrderHeader.java
  * Назначение:      Хранение информации высшего уровня о заказе на покупку.
  **/

package order;

public class OrderHeader implements java.io.Serializable {
   public String empName;     // имя служащего
   public int empNum;         // номер служащего
   public int itemCount;      // общее число пунктов
   // конструктор

public OrderHeader (String empName,
                    int empNum,
                    int itemCount) {
    this.empName = empName;
    this.empNum = empNum;
    this.itemCount = itemCount;
    }
}
```

Создание класса Java для информации о пункте ассортимента

Опишем Java-класс LineItem. Этот класс предназначен для хранения информации о пункте заказа: номер в ассортименте, номер проекта, количество и описание пункта в заказе на покупку. Программный текст LineItem.java:

```
/** Имя программы:   LineItem.java
  * Назначение:      Хранение информации о пунктах ассортимента для заказа.
  **/ package order;

public class LineItem implements java.io.Serializable {
   public int lineno;
   public int projectno;
   public int quantity;
   public String description;
   // конструктор
   public LineItem(int lineno, int projectno, int quantity,
                   String description){
     this.lineno = lineno;
     this.projectno = projectno;
     this.quantity = quantity;
```

```
    this.description = description;
  }
}
```

Создание класса реализации EJB

Наконец, опишем класс реализации OrderEJBImpl.sqlj, который исполняет SQLJ-запросы, возвращая информацию о заголовках и товарах. Класс реализует два метода: getOrder() и getItems(). Эти методы считывают информацию о заголовках и ассортименте заказа при помощи SQL-запросов, обращенных к таблицам PURCHASE_LIST, EMPLOYEE_LIST и LINEITEM_LIST базы данных Purchase Order (см. введение). После текста программы даются пояснения.

```
/** Имя программы:   OrderEJBImpl.sqlj
  * Назначение:      Класс реализации для OrderEJB. Запрашивает в базе
  *                  данных информацию о заказе и ассортименте.
  **/

package order;

import order.OrderHeader;
import order.LineItem;
import java.sql.*;
import java.rmi.RemoteException;
import javax.ejb.*;

public class OrderEJBImpl implements SessionBean {
  SessionContext ctx;
  public void ejbCreate() throws CreateException, RemoteException {
  }

  public void ejbActivate() { }
  public void ejbPassivate() { }
  public void ejbRemove() { }

  public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
  }
  private int orderNumber = 0;  // Сохраним номер заказа
  private int itemCount = 0;    // и счетчик пунктов для getOrder().
                                // Они, возможно, будут использованы
                                // методом getItems().

  // Метод для получения заголовочной информации о заказе
  public OrderHeader getOrder (int orderNumber)
    throws SQLException, RemoteException    // (см. пояснение 1)
  {
    String empName;
    int empNum;
    this.orderNumber = orderNumber;         // (см. пояснение 2)
    // Считываем имя и номер служащего, разместившего заказ
    #sql { SELECT PURCHASE_LIST.employeeno,
                  firstname || ' ' || lastName
             INTO :empNum,
                  :empName
             FROM PURCHASE_LIST, EMPLOYEE_LIST
            WHERE requestno = :orderNumber
              AND EMPLOYEE_LIST.employeeno =
                  PURCHASE_LIST.employeeno
        }; // (см. пояснение 3)
    return new OrderHeader (empName, empNum, getItemCount());
  }
  // Подсчитаем число заказанных товаров
  private int getItemCount () throws SQLException { // (см. пояснение 4)
```

```
        #sql { SELECT COUNT(*)
                 INTO :itemCount
                 FROM LINEITEM_LIST
                WHERE requestno = :orderNumber };
        return itemCount;
    }
    // SQLJ-итератор для считывания характеристик пунктов ассортимента
    #sql iterator LineItemsIter (int lineno,
                                 int projectno,
                                 int quantity,
                                 String description); // (см. пояснение 5)

    // Метод для считывания характеристик пунктов ассортимента
    public LineItem [] getItems() throws SQLException {
                                                        // (см. пояснение 6)
      // Объявление SQLJ-итератора для пунктов ассортимента
      LineItemsIter li = null;
      // Считываем информацию об ассортименте при помощи SQL-запроса
         #sql li = { SELECT lineno,
                            projectno,
                            quantity,
                            description
                       FROM LINEITEM_LIST
                      WHERE requestno = :orderNumber
                      ORDER BY lineno
         };  // (см. пояснение 7)
      LineItem[] items = new LineItem [itemCount];     //(см. пояснение 8)
      for (int i = 0; i < itemCount; i++) {
         li.next();
         items[i] = new LineItem(li.lineno(),
                                 li.projectno(),
                                 li.quantity(),
                                 Li.description());   //(см. пояснение 9)
      }
      li.close(); // Закрываем итератор                // (см. пояснение 10)
      return items;
    }
}
```

Пояснения к `OrderEJBImpl.sqlj`:

1. Этот метод реализует метод `getOrder()` для удаленного интерфейса `Order`.
2. Данный оператор сохраняет аргумент `orderNumber` метода `getOrder()` в переменной экземпляра EJB-класса. Это EJB будет применяться в качестве зерна с сохранением состояния, а сохраненные номер заказа и число пунктов впоследствии будут повторно использоваться при вызове метода `getItems()`.
3. Этот SQL-оператор соединяет таблицы `PURCHASE_LIST` и `EMPLOYEE_LIST`, считывая полное имя и номер служащего, разместившего заказ. В конструкции `WHERE` оператора `SELECT` используется аргумент `orderNumber`.
4. Этот метод подсчитывает общее число пунктов в заказе, выполняя SQL-запрос к таблице `LINEITEM_LIST`. Для выбора верного множества строк в конструкции `WHERE` SQL-оператора `SELECT` используется аргумент `orderNumber`.
5. Этот оператор объявляет класс именованного SQLJ-итератора `LineItemsIter` для считывания характеристик пунктов ассортимента.
6. Этот метод реализует метод `getItems()`, объявленный на шаге 2 для удаленного интерфейса `Order`.

7. Данный SQLJ-оператор инициализирует экземпляр SQLJ-итератора `li`, используя результаты SQL-запроса. Этот запрос считывает характеристики пунктов конкретного заказа в таблице `LINEITEM_LIST`.
8. Этот оператор создает массив объектов `LineItem`, заполняемый с помощью SQLJ-итератора и возвращаемый как результат вызова метода `getItems()`.
9. Этот оператор создает новый экземпляр класса `LineItem` из данных строки в экземпляре итератора `li`.
10. Этот оператор закрывает экземпляр итератора `li`, освобождая курсор для SQL-запроса.

Создание дескриптора внедрения EJB

Для внедрения EJB необходимо создать дескриптор внедрения. Дескриптор внедрения `order.ejb` для компонента `OrderEJB` представлен в листинге 12.1. Зерно `OrderEJBImpl` объявляется как сеансовое с домашним именем `/test/OrderEJB` и с соответствующими домашним и удаленным интерфейсами.

Листинг 12.1. Дескриптор внедрения `OrderEJB`

```
SessionBean order.OrderEJB_Impl {
  BeanHomeName = "/test/OrderEJB";
  RemoteInterfaceClassName = order.Order;
  HomeInterfaceClassName = order.OrderHome;
  AllowedIdentities = {PUBLIC};
  StateManagementType = STATEFUL_SESSION;
  RunAsMode = CLIENT_IDENTITY;
  TransactionAttribute = TX_REQUIRED;
}
```

Внедрение EJB

Итак, программа `OrderEJB` создана. Теперь необходимо скомпилировать исходные файлы (формирующий файл (Makefile) прилагается к тексту программы) и создать jar-файл `OrderEJB.jar`. Затем нужно загрузить и внедрить `OrderEJB` в Oracle8*i*, например:

```
deployejb -republish -temp temp \
  -u jspuser -p jsp -s sess_iiop://data-i:2481:ORCL \
  -descriptor order.ejb orderEJB.jar
```

Вызов `OrderEJB` из web-приложения

О разных архитектурных вариантах написания web-приложений говорилось в разделе "Архитектура web-приложения" главы 2. Действительно, вызвать `OrderEJB` из текста JSP или сервлета можно несколькими способами. Среди них следует отметить:

1. Вызов EJB непосредственно из JSP-программы. Для этого нужно, чтобы на JSP-странице присутствовали программные конструкции Java для поиска, инициализации и вызова методов EJB. Пользоваться таким подходом, как правило, не рекомендуют, поскольку JSP-страницы предназначены в первую очередь для логики представления, а не для вычислений. Большой объем конструкций Java на JSP усложняет обслуживание программы.
2. Создание JavaBean-оболочки для EJB и вызов ее методов с JSP. Этот подход скрывает детали работы с EJB от автора JSP и позволяет использовать JavaBean при помощи стандартного тэга `<jsp:useBean>`. В этом разделе для вызова `OrderEJB` применяется именно такой подход. Его можно расширить, создав специальные тэги для EJB-компонента.
3. Вызов EJB из сервлета. Можно вызывать EJB непосредственно из текста сервлета или создать для EJB JavaBean-оболочку, как в п.2, и вызывать зерно из сервлета. Вызов зерен JavaBean не ограничивается

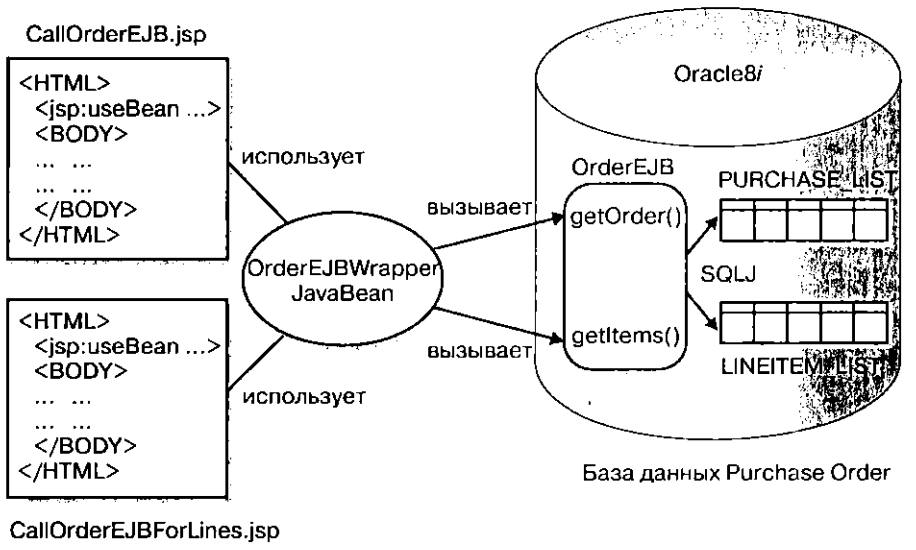
JSP-страницами – ничто не мешает многократно пользоваться программой JavaBean из сервлета. В JSP для установки свойств и области действия JavaBean применяется ряд удобных сокращений, что приходится программировать явным образом в сервлете. Достоинством использования сервлета является то, что он может содержать более сложные программные конструкции Java (если это нужно для web-приложения) и что его может создавать и обслуживать Java-программист.

4. Использование сервлета для поиска и инициализации EJB, а JSP – для логики представления данных. Такой подход называется структурной схемой "модель-представление-контроллер" (MVC, model-view-controller) (см. главу 2). EJB или JavaBean – это модель компонентов, а сервлет – контроллер, принимающий HTTP-запрос, подтверждающий правильность его параметров, инициализирующий EJB и, наконец, переходящий к JSP, где пользователю представляются обработанные данные. Программный текст сервлета создается разработчиком Java, а не автором HTML-текста, поэтому в нем присутствуют детали манипулирования EJB и обработки исключительных ситуаций. Предлагаем читателю создать web-приложение сервлет-EJB-JSP для вызова EJB с именем `ObservationBean` (см. главу 3).

Вызов `OrderEJB` с помощью JavaBean-оболочки

Создадим основанное на JSP приложение, считывающее информацию о заказах при помощи `OrderEJB`. Компоненты этого web-приложения представлены на рис. 12.1. В качестве программы-оболочки, через которую происходит обращение к `OrderEJB`, в приложении используется JavaBean-класс `OrderEJBWrapper`. При применении для этой цели JavaBean намного снижается объем программных конструкций Java, которые нужно писать собственно в JSP-программе. Здесь две JSP-страницы: `CallOrderEJB.jsp` и `CallOrderEJBForLines.jsp`, которые служат точками входа в приложение. Эти JSP-страницы будут пользоваться одним и тем же экземпляром JavaBean при помощи HTTP-объекта `session`.

Рис. 12.1. Архитектура web-приложения, использующего компонент `OrderEJB`



Создание JavaBean-зерна OrderEJBWrapper Создадим JavaBean-зерно `OrderEJBWrapper` для вызова компонента `OrderEJB`. JavaBean должно использоваться в области действия `session`. `OrderEJBWrapper` выполняет следующие функции:

- Поиск `OrderEJB` в JNDI-пространстве имен Oracle8*i* и создание экземпляра компонента. Экземпляр EJB следует уничтожать при выходе JavaBean из области действия `session`.
- Предоставление методов оболочки для методов `getOrder()` и `getItems()` компонента `OrderEJB`.

- Вызов методов `getOrder()` и `getItems()` в любом порядке. Заметим, что `OrderEJB` требует, чтобы `getOrder()` вызывался до `getItems()`, поскольку при считывании пунктов ассортимента должен быть доступен номер заказа. При использовании JavaBean-оболочки это требование можно обойти, вызывая `getOrder()` первым, если `getItems()` вызывается напрямую.
- Некоторое повышение производительности путем кэширования данных, считываемых для заказа. Например, если в новом заказе HTTP-сеанса запрашивается то же, что и в предыдущем, то не повторяется вызов метода `getOrder()` или `getItems()` компонента `OrderEJB`, а возвращается кэшированный результат. Конечно, кэширование подобного типа полезно только тогда, когда одновременно не обновляется информация заказов в базовых SQL-таблицах.

Ниже приводится программный текст зерна JavaBean. Важные участки программы выделены жирным шрифтом. Зерно имеет одно свойство с именем `orderNumber` и методы `set` и `get` этого атрибута. После создания методом `getOrder()` экземпляр компонента `OrderEJB` сохраняется локально и многократно используется в течение существования JavaBean-зерна `OrderEJBWrapper`. JavaBean реализует интерфейс `HttpSessionBindingListener` (см. раздел "События и интерфейсы приемников событий" главы 11), так как оно предназначено для применения JSP-страницей с областью действия `session`. Экземпляр EJB становится недействительным, когда истекает срок действия HTTP-объекта `session`.

Отметим, что установка номера заказа в значение, отличное от предыдущего, вызывает отбраковку кэшированной информации о заказе. Если номер заказа тот же, что и раньше, возвращается кэшированное значение. Если нужно не то значение, которое содержится в кэше, можно вызвать ту же JSP в другом HTTP-сеансе; будет создан другой экземпляр JavaBean (а следовательно, и `OrderEJB`), и можно будет считать информацию о последнем заказе. Еще одна альтернатива – указать иной номер заказа, отбросив кэшируемое значение.

```
/** Имя программы:   OrderEJBWrapper.java
  * Назначение:      JavaBean для обращения к OrderEJB.
  **/

package mybeans ;

import order.Order;
import order.OrderHome;
import order.OrderHeader;
import order.LineItem;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.http.HttpSessionBindingListener;
import javax.servlet.http.HttpSessionBindingEvent;
import java.util.Hashtable;

public class OrderEJBWrapper
             implements HttpSessionBindingListener
{
   public OrderEJBWrapper() {}           // конструктор зерна
   private int orderNumber = 0;          // свойство зерна

   // Кэшируем экземпляр OrderEJB и считаем характеристики заказа и ассортимента
   private Order orderEJB = null;
   private OrderHeader orderHeader = null;
   private LineItem[] itemsArray = null;

        // Метод для установки свойства orderNumber
        public synchronized void setOrderNumber (int orderNum) {
```

```
  if (this.orderNumber != orderNum) {
      orderHeader = null;      // отбросим кэшированное значение
      itemsArray = null;       // отбросим кэшированное значение
      this.orderNumber = orderNum;
  }
}

public int getOrderNumber () {
  return orderNumber;
}

public synchronized OrderHeader getOrder () throws Exception {
  if (orderHeader != null)
  return orderHeader;      // Вновь воспользуемся кэшированным значением,
                           // если оно доступно
      if (orderEJB == null) { // создадим экземпляр EJB
      String serviceURL = "sess_iiop://data-i:2481:ORCL";
      String objectName = "/test/OrderEJB";
      String user = "jspuser";
      String password = "jsp";

      // Клиентская программная конструкция для поиска EJB и
      // создания экземпляра
      Hashtable env = new Hashtable();
      env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
      env.put(Context.SECURITY_PRINCIPAL, user);
      env.put(Context.SECURITY_CREDENTIALS, password);
      env.put(Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN);
      Context ic = new InitialContext(env);

      OrderHome home = (OrderHome)ic.lookup (serviceURL + objectName);
      orderEJB = home.create(); // создадим экземпляр EJB
    }
  orderHeader = orderEJB.getOrder (orderNumber);  // получим информацию о заказе
  return orderHeader;
}

public synchronized LineItem[] getItems() throws Exception {
  if (itemsArray != null)
      return itemsArray; // Вновь воспользуемся кэшированным значением
  if (orderEJB == null)
      orderHeader = getOrder(); // создадим экземпляр EJB
  itemsArray = orderEJB.getItems(); // затем считаем характеристики
  return (itemsArray);
}
// Метод, исполняемый при помещении зерна в HTTP-сеанс
public void valueBound(HttpSessionBindingEvent event) {
  // Здесь ничего не делаем!
  // EJB будет создано только при передаче запроса.
}
// Метод, исполняемый при удалении зерна из HTTP-сеанса
public synchronized void valueUnbound(
                    HttpSessionBindingEvent event) {
  if (orderEJB != null) {
      try {
        orderEJB.remove(); // уничтожим экземпляр зерна
      } catch (Exception ignore) {}
  }
}
```

После создания JavaBean-класса OrderEJBWrapper необходимо скомпилировать его и установить скомпилированный класс на web-сервере (см. приложение E). Затем его методы можно вызывать с одной или нескольких JSP-страниц. Теперь создадим страницы CallOrderEJB.jsp и CallOrderEJBForLines.jsp, совместно использующие один экземпляр этого JavaBean при помощи HTTP-объекта session.

Создание CallOrderEJB.jsp Ниже приводится программный текст страницы CallOrderEJB.jsp, используемой в качестве начального экрана приложения. Важные участки программы выделены жирным шрифтом. Здесь применяется экземпляр зерна OrderEJBWrapper с областью действия session. На JSP-странице присутствует HTML-форма для ввода номера заказа, который ищется в базе данных. После ввода номера заказа в текстовом окне и щелчка мыши на кнопке Submit Query генерируется новый HTTP-запрос, обращенный к той же JSP, с введенным номером заказа. Этот номер используется для установки соответствующего свойства JavaBean-зерна OrderEJBWrapper, после чего вызывается его метод getOrder(). Другая HTML-форма JSP отображает кнопку Show Items для считывания пунктов заказа. При щелчке мыши на этой кнопке вызывается программа CallOrderEJBForLines.jsp.

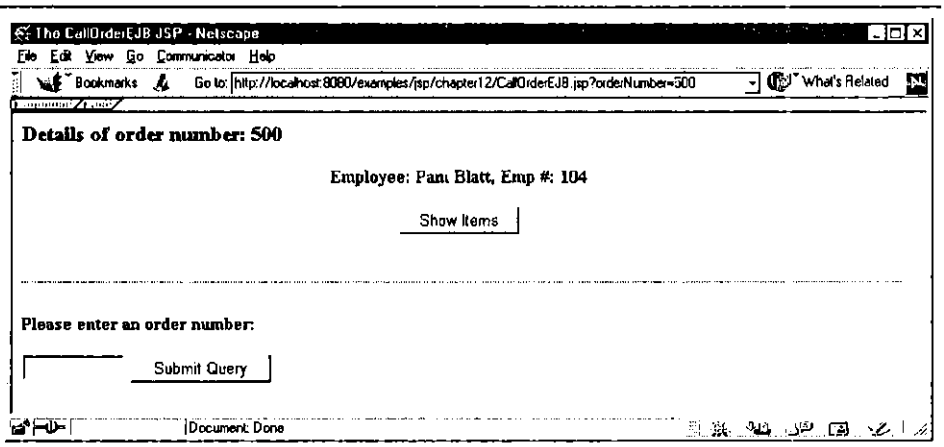
```
<-- Имя программы:   CallOrderEJB.jsp
 -- Назначение:      Принять номер заказа от пользователя и запросить
 --                  информацию о заказе.
 -->
<%@ page import="order.*" %>
<jsp:useBean id="orderBean" class="mybeans.OrderEJBWrapper"
             scope="session" />

<HTML>
<HEAD> <TITLE> The CallOrderEJB JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<%   String orderNumber = request.getParameter("orderNumber");
     if (orderNumber != null) { %>
          <jsp: setProperty name="orderBean" property="orderNumber" />
          <H3>Details of order number: <%= orderNumber %> </H3>
          <CENTER><B>
          <% OrderHeader oh = orderBean.getOrder(); %>
          <P>Employee: <%= oh.empName %>, Emp #: <%= oh.empNum %>
          </B><BR><BR>
          <FORM METHOD=get ACTION="CallOrderEJBForLines.jsp">
            <INPUT TYPE="submit" VALUE="Show Items">
          </FORM>
          </CENTER><BR>
          <HR>
<%   }    %>

<P><B>Please enter an order number: </B></P>
  <FORM METHOD=get ACTION="CallOrderEJB.jsp">
    <INPUT TYPE="text" NAME="orderNumber" SIZE=10>
    <INPUT TYPE="submit" VALUE="Submit Query">
  </FORM>
 </BODY>
</HTML>
```

Чтобы понять принцип действия web-приложения, посмотрим его в работе. Результат выполнения CallOrderEJB.jsp показан на рис. 12.2. На этой странице отображаются информация о каждом служащем, разместившем заказ, и номер служащего. Кроме того, присутствует кнопка Show Items, с помощью которой можно вызывать CallOrderEJBForLines.jsp (см. ниже) и считывать характеристики пунктов ассортимента. Весь смысл заключается не в автоматическом отображении пунктов ассортимента, а в их выводе только после того, как пользователь щелкнет мышью на кнопке Show Items.

Рис. 12.2. Выходные данные CallOrderEJB.jsp



Создание CallOrderEJBForLines.jsp Страница CallOrderEJB.jsp работает совместно с CallOrderEJBForLines.jsp. Эту JSP можно вызывать напрямую или щелчком мыши на кнопке Show Items на CallOrderEJB.jsp. На странице выводятся все пункты ассортимента для конкретного заказа на покупку.

Отметим, что в тэге <jsp:useBean> этой JSP для зерна OrderEJBWrapper используется то же имя orderBean с областью действия session. Напомним, что JavaBean с областью действия session размещается в HTTP-объекте session, и к нему может обращаться любая программа, участвующая в том же HTTP-сеансе (см. главу 11). Следовательно, тэг <jsp:useBean> в тексте программы CallOrderEJBForLines.jsp ищет объект с именем orderBean, ранее созданный программой CallOrderEJB.jsp. Другими словами, эти две JSP-страницы могут совместно пользоваться информацией о заказах, считываемой в продолжение одного и того же HTTP-сеанса через один и тот же экземпляр JavaBean-зерна OrderEJBWrapper.

```
<!-- Имя программы: CallOrderEJBForLines.jsp
  -- Назначение: Использовать JavaBean-зерно orderBean и
  -- отобразить информацию об ассортименте для заказа.
  -->
<%@ page import="order.*" %>
<jsp:useBean id="orderBean" class="mybeans.OrderEJBWrapper"
             scope="session" />
<HTML>
  <HEAD> <TITLE> The CallOrderEJBForLines JSP   </TITLE> </HEAD>
<BODY BGCOLOR="white">
    <CENTER><B>

    <% OrderHeader oh = orderBean.getOrder(); %>
    <P>Employee: <%= oh.empName %>,
        Emp #: <%= oh.empNum %>,
        Order #: <%= orderBean.getOrderNumber() %>
    </B><BR><BR>
  <%

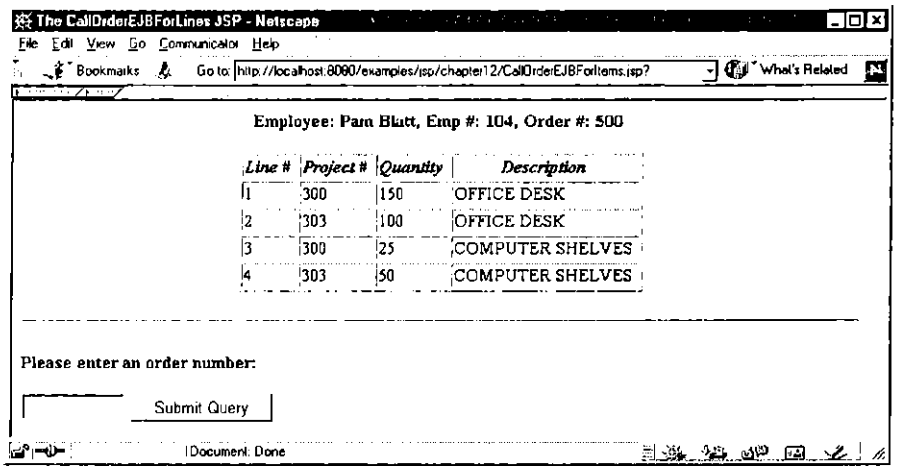
    LineItem[] li = orderBean.getItems();
    if (oh.itemCount > 0) { %>
        <TABLE BORDER>
        <TH><I> Line # </I></TH>
        <TH><I> Project # </I></TH>
        <TH><I> Quantity </I></TH>
        <TH><I> Description </I></TH>
<%      for (int i=0; i < oh.itemCount; i++) { %>
          <TR>
            <TD> <%= li[i].lineno %> </TD>
            <TD> <%= li[i].projectno %> </TD>
            <TD> <%= li[i].quantity %> </TD>
```

```
            <TD> <%= li[i].description %> </TD>
          </TR>
        <% } %>
          </TABLE>
      <% }
          </CENTER><B><BR>
          <HR>

  <P><B>Please enter an order number:</B></P>
    <FORM METHOD=get ACTION="CallOrderEJB.jsp">
        <INPUT TYPE="text" NAME="orderNumber" SIZE=10>
        <INPUT TYPE="submit" VALUE="Submit Query">
    </FORM>
  </BODY>
</HTML>
```

Результат передачи запроса Show Items изображен на рис. 12.3. На этой странице присутствует кнопка Submit Query, при щелчке мыши на которой возвращается страница `CallOrderEJB.jsp`.

Рис. 12.3. Выходные данные `CallOrderEJBForLines.jsp`



Вызов зерна `OrderEJBWrapper` из сервлета

Существует также возможность вызывать JavaBean-оболочку для `OrderEJB` из сервлета, а с помощью JSP отображать данные. Такой подход соответствует архитектуре "модель-представление-контроллер", которая удобна для работы со сложными web-приложениями (см. раздел "Архитектура web-приложения" главы 2). Сервлет-"контроллер" выполняет следующие задачи:

- Инициализация экземпляра JavaBean-зерна `OrderEJBWrapper` или непосредственно компонента `OrderEJB`
- Вызов метода `getOrder()` для считывания информации о конкретном заказе
- Связывание считанного объекта `OrderHeader` или зерна с HTTP-сеансом с помощью метода `session.setAttribute()`
- Передача управления исполнением программы JSP-странице для представления информации

JSP может считывать объект `OrderHeader` из совместно используемого HTTP-сеанса и отображать его данные надлежащим образом; то же самое можно сказать и о вызове `getItems()` для представления пунктов заказа. Создать приложение, использующее подобную архитектуру компонентов, предлагаем читателю.

Разработка web-приложений с помощью компонентов CORBA

В части III этой книги говорилось о разработке и внедрении компонентов CORBA в базе данных Oracle8*i*. Применим полученные знания и создадим компонент CORBA, который будет считывать информацию о заказе, хранимую в базе данных Purchase Order, а также вызовем экземпляр объекта CORBA из web-приложения. Функциональные возможности этого компонента схожи с теми, которыми обладает JSP-страница `OrderEJB` (см. выше), но для его программирования и внедрения применяется технология CORBA. Сначала создадим компонент CORBA для получения информации о заказе в базе данных, а затем построим web-приложение, использующее этот компонент.

Объект CORBA для заказов

Процесс описания и внедрения объекта CORBA состоит из трех шагов:

1. Создание и компиляция IDL-файла для модуля CORBA.
2. Описание серверного класса реализации.
3. Внедрение объекта CORBA в Oracle8*i*.

Рассмотрим выполнение этих операций для модуля CORBA заказов.

Создание IDL-файла CORBA

Сначала необходимо создать файл `orderCorba.idl`. Он состоит из модуля `orderCorba`, описывающего структуры информации о заголовках заказа и пунктах ассортимента. Используем две структуры: `LineItem` и `OrderInfo`, причем в `OrderInfo` будет содержаться последовательность объектов `LineItem`. В интерфейсе `Order` определяется один метод `getOrder()`, предназначенный для считывания всей информации, связанной с заказом. Эта схема намеренно отличается от реализации `OrderEJB` (см. выше), в которой имеются два метода: `getOrder()` и `getItems()`. Здесь преследуется цель — показать иной способ выбора информации о заказах.

```
// Имя программы:    orderCorba.idl
// Назначение:       Описать структуру объекта CORBA с помощью IDL.
//
module orderCorba {
   struct LineItem {
     long lineno;
     long projectno;
     long quantity;
     wstring description;
   };

   struct OrderInfo {
     wstring empName;
     long empNum;
     long itemCount;
     sequence <LineItem> items;
   };

   exception SQLError {
     wstring message;
   };

   interface Order {
     OrderInfo getOrder (in long orderNumber) raises (SQLError);
   };
};
```

Этот IDL-файл можно скомпилировать с помощью утилиты idl2java, сгенерировав при этом нужные классы Java. Среди генерируемых файлов будут присутствовать описания классов Java для orderCorba.OrderInfo и orderCorba.LineItem для структур OrderInfo и LineItem соответственно, а также другие helper- и holder-классы, необходимые для организации взаимодействия между клиентом и сервером. В генерируемом классе orderCorba.OrderInfo будет содержаться Java-массив LineItem[], представляющий последовательность пунктов ассортимента.

Создание серверного класса реализации

Теперь опишем серверный класс реализации OrderCorbaImpl. Этот класс должен реализовывать общий метод getOrder() с сигнатурой, описанной в IDL-файле orderCorba. Статические SQL-запросы для считывания информации о заказах удобно выполнять с помощью SQLJ. Ниже показан файл OrderCorbaImpl.sqlj и даны пояснения. Вы уже знакомы с объявлением SQLJ-итератора и запросами. Схема базы данных Purchase Order приведена во введении к книге.

```
/** Имя программы: OrderCorbaImpl.sqlj
  * Назначение: Серверный класс реализации для модуля orderCorba.
  **/
package orderCorbaServer;

import orderCorba.*;
import oracle.aurora.AuroraServices.ActivatableObject;
import java.sql.*;

public class OrderCorbaImpl
      extends _OrderImplBase
      implements ActivatableObject {

  public OrderInfo getOrder (int orderNumber)
                  throws SQLError {    // (см. пояснение 1)
    try {
        String empName = null;
        int empNum = 0;

        // Выберем информацию о служащем для заказа
        #sql { SELECT PURCHASE_LIST.employeeno,
                      firstname || ' ' || lastName
                 INTO :empNum,
                      :empName
                 FROM PURCHASE_LIST, EMPLOYEE_LIST
                WHERE requestno = :orderNumber
                  AND EMPLOYEE_LIST.employeeno =
                      PURCHASE_LIST.employeeno
        };

    return new OrderInfo(empName, empNum, getItemCount(orderNumber),
                         GetItems(orderNumber)); //(см. пояснение 2)
    } catch (SQLException e) {
      throw new SQLError (e.getMessage());
    }
  }

  private int itemCount = 0;
  // Метод для выбора общего числа пунктов заказа
  private int getItemCount (int orderNumber) throws SQLException {
        #sql { SELECT COUNT(*)
             INTO :itemCount
             FROM LINEITEM_LIST
            WHERE requestno = :orderNumber
    };
    return itemCount;
```

```
    }

    // Объявление SQLJ-итератора для пунктов ассортимента
    #sql iterator LineItemsIter (int lineno, int projectno,
                                 int quantity, String description);

    // Метод для выбора пунктов, соответствующих определенному заказу
    private LineItem[] getItems(int orderNumber) throws SQLException {
      LineItemsIter li = null;
      // (см. пояснение 3)
      #sql li = {
                      SELECT lineno,
                             projectno,
                             quantity,
                             description
                        FROM  LINEITEM_LIST
                        WHERE requestno = :orderNumber
                        ORDER BY lineno
            };

      LineItem[] items = new LineItem [itemCount];
      for (int i = 0; i < itemCount; i++) {
          li.next();
          items[i] = new LineItem(li.lineno(),
                                  li.projectno(),
                                  li.quantity(),
                                  li.description());
      }
      li.close();
      return items;
    }

    public org.omg.CORBA.Object _initializeAuroraObject() {
      return this;
    }
}
```

Пояснения к `OrderCorbaImpl.sqlj`:

1. Это реализация метода `getOrder()`, описанного в модуле `orderCorba`. Из таблиц `PURCHASE_LIST` и `EMPLOYEE_LIST` выбирается информация о заголовках заказа, а затем вызывается метод `getItems()` для считывания информации об ассортименте.
2. Этот оператор строит новый объект `OrderInfo` для указанного заказа и возвращает его.
3. Этот метод считывает пункты указанного заказа из таблицы `LINEITEM_LIST` с помощью именованного SQLJ-итератора.

Компиляция и внедрение компонента CORBA

Классы компилируются и загружаются в Oracle8*i* с помощью утилиты `loadjava` (Makefile прилагается к программному тексту). После этого необходимо опубликовать объект CORBA в пространстве имен JNDI с помощью инструмента оболочки сеансов Oracle8*i*. Приведем пример:

```
publish -republish -u jspuser -p jsp -schema JSPUSER \
    -s  sess_iiop://data-i:2481:ORCL \
      /test/OrderCorba orderCorbaServer.OrderCorbaImpl \
       orderCorba.OrderHelper
```

Вызов объекта CORBA из web-приложения

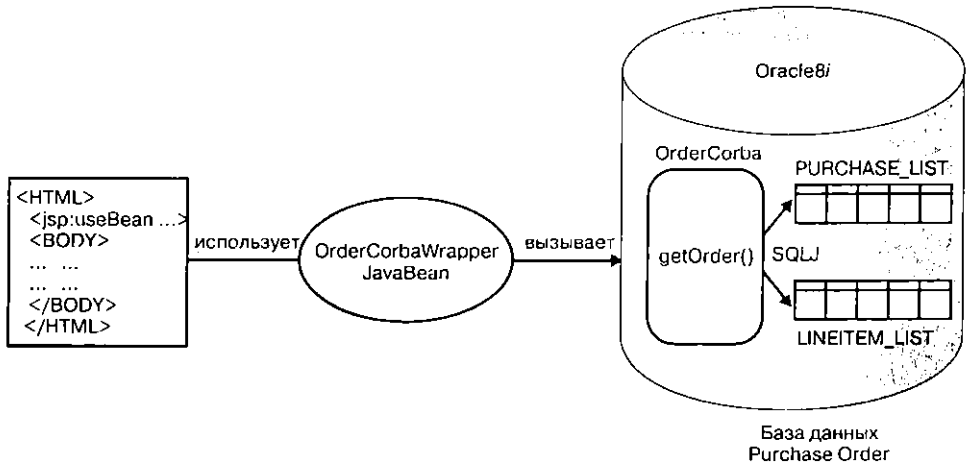
Как и при работе с зернами EJB, для web-приложения, использующего компонент CORBA, существует несколько архитектурных вариантов. Самым

очевидным, но не обязательно лучшим, подходом является вызов его из JSP-программы. Это предполагает написание в JSP-программе большого объема программного текста Java, что усложняет процесс сопровождения. Лучше вызывать объект CORBA при помощи JavaBean или сервлета и применять JSP только для ввода данных и для отображения результатов. Именно таким подходом мы и воспользуемся. Опишем для объекта `OrderCorba` JavaBean-оболочку `OrderCorbaWrapper` и применим это зерно на JSP-странице.

Вызов объекта `OrderCorba` при помощи JavaBean-оболочки

Создадим приложение на основе JSP, использующее `OrderCorba` для считывания информации о заказе. Компоненты этого web-приложения представлены на рис. 12.4. Для обращения к компоненту `OrderCorba` в приложении применяется JavaBean-класс `OrderCorbaWrapper`. При использовании компонента JavaBean снижается объем программных конструкций Java, которые нужно писать собственно в JSP-программе. JSP-страница `CallOrderCorba.jsp` будет точкой входа в это приложение.

Рис. 12.4. Архитектура web-приложения, использующего компонент `OrderCorba`



Создание JavaBean-зерна `OrderCorbaWrapper` Ниже приводится программный текст JavaBean-зерна `OrderCorbaWrapper`, которое ищет и активизирует объект CORBA, а также предоставляет метод оболочки для его метода `getOrder()`. Дополнительной функцией метода `getOrder()` является попытка повторного использования кэшированной информации о заказе в том случае, если номер заказа не изменился с момента последнего вызова этого метода. Логика предлагаемого JavaBean довольно похожа на логику `OrderEJBWrapper` (см. выше). Важные участки текста программы выделены жирным шрифтом.

```
/** Имя программы:   OrderCorbaWrapper.java
  * Назначение:      Обратиться к объекту OrderCorba и считать информацию о заказе.
  */
package mybeans;

import orderCorba.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.servlet.http.*;

public class OrderCorbaWrapper
             implements HttpSessionBindingListener
{

  public OrderCorbaWrapper() {}

  private Order order = null;           // объект CORBA
```

```
  private OrderInfo orderInfo = null;   // считываемая информация о заказе

  private int orderNumber;
  public void setOrderNumber (int orderNum) {
    if (this.orderNumber != orderNum) {
        orderInfo = null; // отбросим ранее кэшированное значение
        this.orderNumber = orderNum;
        }
          }

  public OrderInfo getOrder() throws Exception {
    if (orderInfo != null)
        return orderInfo;   // повторно воспользуемся ранее кэшированным значением

    if (order == null) { // найдем объект
        String serviceURL = "sess_iiop://data-i:2481:ORCL";
        String objectName = "/test/OrderCorba";
        String user = "jspuser";
        String password = "jsp";

        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, user);
        env.put(Context.SECURITY_CREDENTIALS, password);
        env.put(Context.SECURITY_AUTHENTICATION,
          ServiceCtx.NON_SSL_LOGIN);

        Context ic = new InitialContext(env);

        order = (Order)ic.lookup (serviceURL + objectName);
    }
    OrderInfo orderInfo = order.getOrder(orderNumber);
    return orderInfo;
  }
  // Метод, исполняемый при размещении зерна в HTTP-сеансе
  public void valueBound(HttpSessionBindingEvent event) {
    // Здесь ничего не делаем!
    // Объект CORBA будет создан только при передаче запроса
  }
  // Метод, исполняемый при удалении зерна из сеанса
  public synchronized void valueUnbound(
                    HttpSessionBindingEvent event) {
    if (order != null) {
        try {
          order.deactivate();
        } catch (Exception e) {}
    }
  }
}
```

Создание `CallOrderCorba.jsp` В этой JSP-программе для вызова объекта `OrderCorba` используется описанное выше JavaBean-зерно `OrderCorbaWrapper`. Текст программы приводится ниже. Важные участки текста выделены жирным шрифтом. На JSP-странице присутствуют HTML-форма для ввода номера заказа и кнопка Submit Query для считывания характеристик заказа. Эта кнопка инициирует вызов компонента `OrderCorba` через JavaBean. Данные форматируются и выдаются пользователю в виде HTML-таблицы.

```
<!-- Имя программы:  CallOrderCorba.jsp
  -- Назначение:     Считать информацию о заказе, используя JavaBean-зерно
  --                 OrderCorbaWrapper для обращения к объекту OrderCorba.
  -->

<%@ page import="orderCorba.*" %>
```

```
<jsp:useBean id="orderWrapper"
             class="mybeans.OrderCorbaWrapper"
             scope="session" />
<jsp:setProperty name="orderWrapper"
                 property="orderNumber" />

<HTML>
<HEAD> <TITLE> The CallOrderCorba JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<%    String orderNumber = request.getParameter("orderNumber");
      if (orderNumber != null) { %>
        <H3>Details of order number: <%= orderNumber %> </H3>
        <CENTER><B>
        <%  OrderInfo oi = orderWrapper.getOrder(); %>
        Employee: <%= oi.empName %>,  Emp #:  <%= oi.empNum %>
        <BR><BR>
        <% LineItem[] li = oi.items;
        if (oi.itemCount > 0) { %>
           <TABLE BORDER>
           <TH><I> Line # </I></TH>
           <TH><I> Project # </I></TH>
           <TH><I> Quantity </I></TH>
           <TH><I> Description </I></TH>
      <%   for (int i=0; i < oi.itemCount; i++) { %>
           <TR>
              <TD> <%= li[i].lineno %> </TD>
              <TD> <%= li[i].projectno %> </TD>
              <TD> <%= li[i].quantity %> </TD>
              <TD> <%= li[i].description %> </TD>
           </TR>
      <% } %>
        </TABLE>
     <% } %>
       </CENTER><B><BR> <HR>
<%  }  %>

<P><B>Please enter an order number: </B></P>
  <FORM METHOD=get>
  <INPUT TYPE="text" NAME="orderNumber" SIZE=10>
  <INPUT TYPE="submit" VALUE="Submit Query">
  </FORM>
</BODY>
</HTML>
```

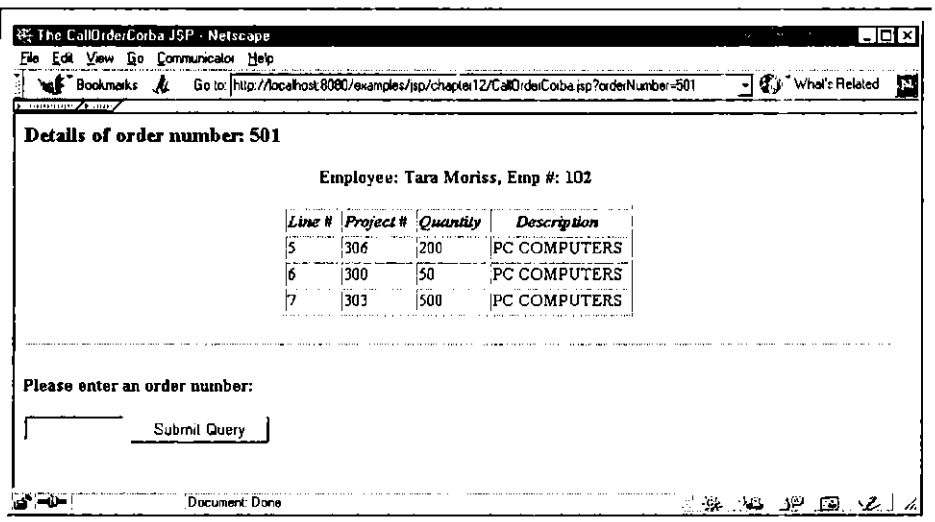
Результат вызова этой JSP показан на рис. 12.5. Отметим, что при вводе в форму номера заказа, равного предыдущему, будет использоваться значение, локально кэшируемое в JavaBean с областью действия session, а не вызываться метод getOrder() удаленного объекта CORBA.

Вызов зерна OrderCorbaWrapper из сервлета

Можно также вызывать JavaBean-оболочку из сервлета, а с помощью JSP отображать данные. Такой подход соответствует архитектуре "модель-представление-контроллер", которая удобна для работы со сложными web-приложениями. Сервлет- "контроллер" выполняет следующие задачи:

- Инициализация экземпляра JavaBean-зерна OrderCorbaWrapper или непосредственно объекта OrderCorba
- Вызов метода getOrder() для считывания информации о конкретном заказе

Рис. 12.5. Выходные данные CallOrderCorba.jsp



- Связывание считанного объекта `OrderInfo` с HTTP-сеансом при помощи метода `session.setAttribute()`
- Передача управления исполнением программы JSP-странице для представления информации

JSP может считывать объект `OrderInfo` из (общего) HTTP-объекта `session` и отображать его данные. Предлагаем читателю написать программу с использованием этой модели.

Использование библиотек тэгов JSP

В спецификации JSP 1.1 определена стандартная переносимая схема расширения тэгов для описания и применения библиотек тэгов. *Библиотека тэгов* (tag library) – это совокупность специальных тэгов, каждый из которых выполняет определенную функцию. Например, можно создать компонент Java и раскрыть его функциональные возможности при помощи удобного синтаксиса JSP-тэгов. Это предотвратит загромождение JSP программными конструкциями Java и скроет сложность программы. В состав описания специального тэга входят его имя, атрибуты и тело. Набор специальных тэгов, доступных в библиотеке, описывается в файле *дескриптора библиотеки тэгов* (TLD, tag library descriptor) в XML-обозначении. Библиотеку тэгов можно импортировать в JSP-программу с помощью директивы `taglib`, а затем применить тэги, определенные в библиотеке.

Библиотеки тэгов реализуются, как правило, поставщиками JSP, но ничто не мешает описать свои собственные тэги и упаковать их в одну или несколько библиотек. Схема расширения тэгов позволяет увеличить число стандартных функциональных тэгов, подобных `<jsp:useBean>`, модулями, переносимыми из одной JSP-реализации в другую.

В реализации JSP 1.1 в Oracle содержатся две библиотеки тэгов:

- Библиотека SQL-тэгов, используемых для исполнения SQL-запросов и форматирования результатов
- Библиотека JML-тэгов (JML расшифровывается как *JSP Markup Language* – язык разметки JSP), в которой содержатся тэги общего характера, облегчающие написание сценариев, включая приложение таблиц стилей XSL для XML-документов

С помощью этих тэгов удобно, к примеру, исполнять и форматировать SQL-запросы, чему способствует компактность синтаксиса, а также преобразовывать динамически генерируемые XML-конструкции в HTML-таблицу, используя таблицы стилей. Ниже описывается ряд полезных тэгов из этих двух библиотек (полностью характеристики всех тэгов представлены в руководстве "Oracle

JSP Developer's Guide and Reference, Release 8.1.7" корпорации Oracle). Сначала поговорим об импорте библиотек тэгов при помощи директивы `taglib`. Затем создадим JSP- страницы, которые используют библиотеки тэгов, предлагаемые в реализации Oracle JSP 1.1. В заключение напишем простой специальный тэг.

Директива `taglib`

Директива `taglib` используется для того, чтобы указать на применение библиотеки специальных тэгов на JSP-странице. Назначение этой директивы двояко: задать местонахождение файла TLD (дескриптор библиотеки тэгов) и описать локальный префикс для этих тэгов. Синтаксис директивы `taglib`:

```
<%@ taglib uri="tldURI" prefix="префикс" %>
```

Директива `taglib` имеет два атрибута: `uri` и `prefix`. Префикс используется для выделения тэгов данной библиотеки среди других тэгов. Атрибут `uri` определяет имя и местоположение TLD-файла (URI означает Universal Resource Indicator – *универсальный индикатор ресурса*; это более общий термин, чем URL). Существуют два различных способа указания TLD, что зависит от способов создания библиотеки тэгов и web-приложения. Например, можно указать полностью определенный путь к TLD-файлу:

```
<%@ taglib uri="/WEB-INF/tlds/sqltaglib.tld" prefix="ora" %>
```

Здесь местоположение задается маршрутом относительно приложения (начинается с символа "/"), и в этой области TLD-файл должен быть доступен. В качестве альтернативы можно указать не `tld`-, а `jar`-файл, содержащий TLD-файл в виде `META-INF/taglib.tld`. В этом случае директива `taglib` будет выглядеть так:

```
<%@ taglib uri="/WEB-INF/tlds/sqltags.jar" prefix="ora" %>
```

Кроме того, в директиве `taglib` можно использовать сокращенный URI, описав сокращение в дескрипторе внедрения `web.xml` (о дескрипторе внедрения `web.xml` и его использовании вместе с TLD-файлами подробно рассказывается в спецификациях Servlet 2.2 и JSP 1.1 компании Sun Microsystems). Этот файл применяется JSP-страницами для указания местоположения и кратких имен TLD-файлов; впоследствии эти сокращения можно использовать в директиве `taglib`.

В импортируемой библиотеке тэгов директива `taglib` должна быть указана раньше каких бы то ни было специальных тэгов.

SQL-тэги

Для удобства исполнения SQL-операций в состав Oracle JSP 1.1 включена библиотека с тэгами. Основой тэгов являются подпрограммы JDBC, исполняющие SQL-операции. В этой библиотеке содержатся следующие тэги:

- `dbOpen`
- `dbClose`
- `dbQuery`
- `dbCloseQuery`
- `dbNextRow`
- `dbExecute`

Ниже приводятся описание тэгов и примеры их использования.

Тэг `dbOpen`

Этот тэг используется для открытия соединения с базой данных. Общий синтаксис тэга:

```
<ora:dbOpen [ connId="идентификатор_соединения" ]
             user="имя_пользователя"
             password="пароль"
             URL="URL_базы данных" >
        ...необязательное вложенное тело, использующее данное соединение...
</ora:dbOpen>
```

Префикс ora соответствует префиксу, указанному в директиве taglib для JSP. Атрибут connId необязателен и описывает идентификатор соединения. Если идентификатор соединения отсутствует, соединение с базой данных автоматически закрывается при достижении конечного тэга </ora:dbOpen>. Описатель соединения неявно доступен для таких тэгов, как dbQuery, вложенных в тело. Если идентификатор соединения указан, на описатель соединения можно ссылаться по его имени из любого места JSP-страницы, например из тэга dbQuery. Совсем не обязательно, чтобы SQL-операции, где используется идентификатор соединения, были вложены между начальным и конечным тэгами dbOpen. В случае их произвольного размещения по окончании выполнения SQL-операций соединение необходимо закрывать явным образом с помощью тэга dbClose.

Атрибуты user, password и URL базы данных обязательны. Их значениями на JSP-странице могут быть постоянные строки символов или, что более распространено, выражения этапа запроса, в которых фигурируют, как правило, HTTP-объекты request или session. Приведем пример использования вычисляемых выражений для атрибутов user и password:

```
<ora:dbOpen connId="conn"
            user="<%=request.getParameter(\"username\") %>"
            password="<%=request.getParameter(\"mypassword\") %>"
            URL="jdbc:oracle:oci8:@" />
```

В этом примере у тэга dbOpen нет тела, и поэтому конечным тэгом является />. Тэг открывает соединение с базой данных с указанными параметрами username, password и URL и присваивает описателю этого соединения имя conn. Впоследствии другие операторы JSP-страницы смогут ссылаться на соединение conn до окончания действия JSP-страницы.

Возникает вопрос: каков же тип объекта conn? Вспомните зерна доступа к базе данных, о которых говорилось в главе 11. Это компоненты, заранее созданные в служебной библиотеке зерен Oracle JSP и помогающие исполнять SQL-операции на JSP-странице. Эти зерна являются основой SQL-тэгов. Так, в тэге dbOpen объект ConnBean используется для представления соединения. При необходимости на JSP можно установить дополнительные свойства ConnBean, например stmtCacheSize, preFetch или batchSize, чтобы применить эти средства Oracle JDBC (см. раздел "Библиотека зерен Oracle JSP " главы 11).

Тэг dbClose

У тэга dbClose один обязательный атрибут – connId, который соответствует идентификатору соединения, описанному ранее в тэге dbOpen. Тэг dbClose закрывает соединение с базой данных, представленное идентификатором. Если идентификатор соединения не указан в тэге dbOpen, соединение с базой данных закрывается автоматически при обнаружении конечного тэга для dbOpen. В этом случае тэг dbClose необязателен. В следующем примере закрывается соединение, представленное идентификатором conn:

```
<ora:dbClose connId="conn" />
```

Тэг dbQuery

Тэг dbQuery используется для исполнения SQL-запроса. Выходными данными может быть результирующее множество JDBC, HTML-таблица или строка символов XML. Оператор SELECT размещается в теле этого тэга между начальным и конечным тэгами dbQuery. Синтаксис таков:

```
<ora:dbQuery
    [ connId="имя_соединения" ]
    [ output="HTML"|"XML"|"JDBC" ]
    [ queryId="идентификатор_запроса" ] >
    ... оператор SELECT ...
    </ora:dbQuery>
```

Тэг имеет три необязательных атрибута: connId, queryId и output. Тэг можно вкладывать в dbOpen, при этом атрибут connId не применяется, и для исполнения запроса неявно используется соединение, установленное внешним тэгом dbOpen.

Атрибут output служит для указания вида результата запроса:

- **HTML-таблица** В самом простом случае можно распечатать результаты в виде HTML-таблицы с именами или псевдонимами SQL-столбцов в качестве заголовков. Это режим по умолчанию.
- **Строка символов XML** Можно получать результат в виде строки символов XML, генерируемой утилитой Oracle XML-SQL (см. главу 11).
- **Результирующее множество JDBC** Наконец, можно получить результирующее множество JDBC и обработать его строки с помощью тэга dbNextRow (см. ниже). В этом случае необходимо указать идентификатор для генерируемого результирующего множества с помощью атрибута queryId.

При использовании атрибута queryId после обработки строк необходимо явно закрывать базовый курсор тэгом dbCloseQuery (см. ниже). В случае отсутствия атрибута queryId курсор закрывается автоматически по достижении конечного тэга </ora:dbQuery>.

На самом деле объектом, представляемым идентификатором queryId, является объект CursorBean – стандартный компонент-зерно из библиотеки зерен Oracle (см. раздел "Библиотека зерен Oracle JSP " главы 11).

Тэг dbCloseQuery

Тэг dbCloseQuery закрывает курсор, который соответствует атрибуту queryId, указанному в тэге dbQuery (см. выше). Если атрибут queryId не используется в тэге dbQuery, тэг dbCloseQuery необязателен; в этом случае курсор закрывается автоматически при достижении конечного тэга для dbQuery. Синтаксис тэга dbCloseQuery:

```
<ora:dbCloseQuery queryId="идентификатор_запроса" />
```

Тэг dbNextRow

Тэг dbNextRow используется совместно с тэгом dbQuery (см. выше) для обработки каждой строки результирующего множества данных. На это множество ссылается queryId, указанный в тэге dbQuery, для атрибутов которого необходимо установить output="JDBC". Программная конструкция для обработки строк результирующего множества вкладывается между начальным и конечным тэгами dbNextRow и исполняется для каждой строки множества. Общий синтаксис этого тэга:

```
<ora:dbNextRow queryId="идентификатор_запроса" >
... Программная конструкция для обработки каждой строки
результирующего множества запроса ...
</ora:dbNextRow >
```

Например, в следующем программном фрагменте в каждой строке распечатываются значения второго и четвертого столбцов результирующего множества JDBC с именем query1 (заданным в тэге dbQuery):

```
... ...
<br>
<jml:dbNextRow queryId="query1">
      <%= query1.getString(2) %>
```

```
    <%= query1.getString(4) %> <br>
</jml:dbNextRow>
```

Тэг dbExecute

Тэг dbExecute используется для исполнения любого SQL-оператора DDL или DML, в том числе хранимой программы. SQL-оператор вкладывается в тело тэга между начальным и конечным тэгами dbExecute. Общий формат этого тэга:

```
<ora:dbExecute
     [connId="идентификатор_соединения"]
     [output="yes"|"no"] >
  ... вложенный SQL-оператор DDL или DML ...
</ora:dbExecute >
```

Тэг имеет два необязательных атрибута: connId и output. Если тэг вкладывается в dbOpen, неявно используется соединение с базой данных, открытое этим тэгом. Иначе необходимо указывать идентификатор, который ссылается на соединение, открытое ранее тэгом dbOpen. Атрибут output применяется для описания дальнейших действий по отношению к результату: посылать ли его браузеру как часть генерируемой web-страницы или нет. По умолчанию выходные данные этим тэгом не генерируются.

Создание DBQuery.jsp

Теперь создадим JSP-страницу DBQuery.jsp, которая использует описанные выше SQL-тэги dbOpen и dbQuery для исполнения SQL-запроса и автоматического отображения результата в виде HTML-таблицы. Архитектура этого приложения представлена на рис. 12.6. Классы обработчиков тэгов выступают в роли модульных компонентов, концентрирующих в себе конкретные функциональные возможности, например средства обращения к базе данных.

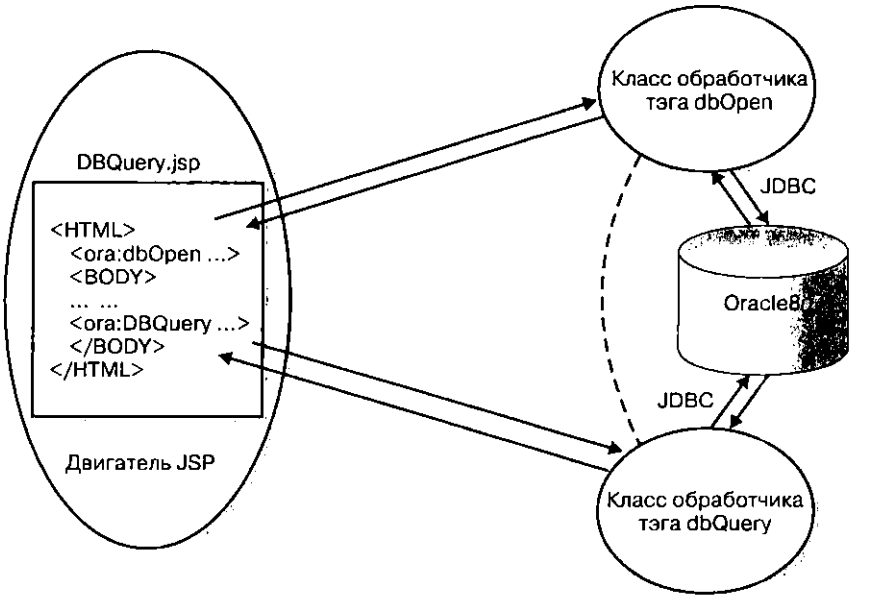


Рис. 12.6. Архитектура JSP с SQL-тэгами

Приведем программный текст DBQuery.jsp, выделив важные его участки жирным шрифтом:

```
<!-- Имя программы: DBQuery.jsp
  -- Назначение: С помощью тэгов dbOpen и dbQuery обратиться к
  -- таблице с запросом.
  -->
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="ora" %>
```

```
<HTML>
<HEAD> <TITLE> The DBQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<H3>You can query any table using this page.</H3>
<%
   String tableName = request.getParameter("tableName");
   String condition = request.getParameter("searchCond");
   if ((condition != null) && (condition.length() != 0))
        condition = "WHERE " + condition; else condition="";
   if (tableName != null) { %>
     <ora:dbOpen URL="jdbc:oracle:oci8:@"
                 user="<%= request.getParameter(\"userName\") %>"
                 password="<%= request.getParameter(\"password\") %>" >
          <H3> Search results for table: <%= tableName %>,
                                 condition: <%= condition %>
          <ora:dbQuery>
            SELECT * FROM <%= tableName %>
                     <%= condition %>
          </ora:dbQuery>
          </ora:dbOpen>
<% } %>
<BR><HR>
   <B>Enter the following information:</B>
   <FORM METHOD="POST">
      <TABLE>
         <TR> <TD> User Name </TD>
              <TD> <INPUT TYPE="text" NAME="userName" SIZE=30> </TD>
         </TR>
         <TR> <TD> Password </TD>
              <TD> <INPUT TYPE="password" NAME="password" SIZE=30> </TD>
         </TR>
         <TR> <TD> Table Name </TD>
              <TD> <INPUT TYPE="text" NAME="tableName" SIZE=30> </TD>
         </TR>
         <TR> <TD> Search Condition </TD>
              <TD> <INPUT TYPE="text" NAME="searchCond" SIZE=30> </TD>
         </TR>
      </TABLE>
         <INPUT TYPE="submit" VALUE="Lookup rows">
    </FORM>
  </BODY>
</HTML>
```

В программе DBQuery.jsp HTML-форма применяется для приема имени и пароля пользователя, регистрирующегося в базе данных, а также имени таблицы и условий поиска для выполнения запроса. Соединение с базой данных открывается с помощью тэга dbOpen. Запрос исполняется тэгом dbQuery, который вложен в тэг dbOpen, чтобы соединение, открытое тэгом dbOpen, можно было использовать неявно для исполнения SQL-запроса. Как видим, применение SQL-тэгов значительно упрощает программу обращения к базе данных.

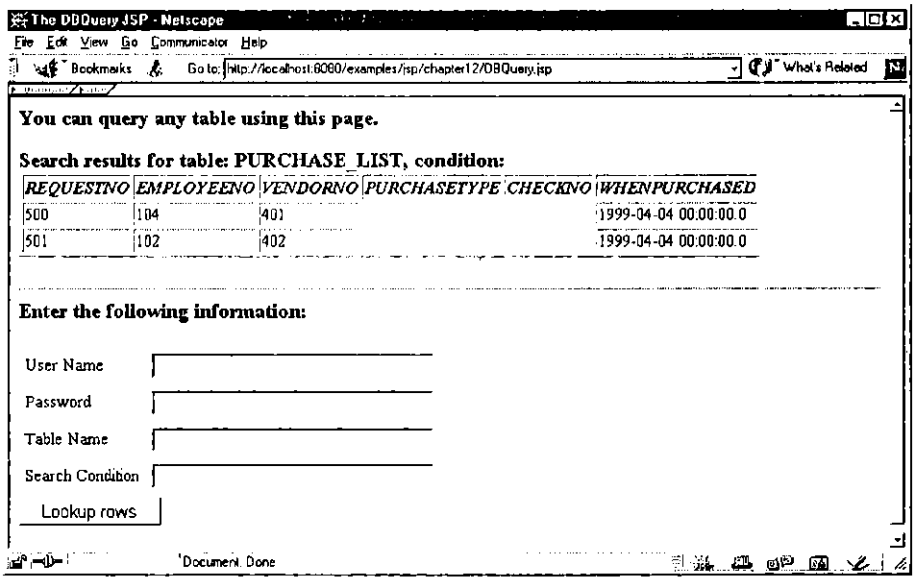
Выполнение DBQuery.jsp Перед выполнением DBQuery.jsp нужно убедиться в том, что установлены следующие библиотеки:

- ojsputil.jar Эта библиотека поставляется в дистрибутиве Oracle JSP и содержит в своем составе библиотеки зерен и тэгов. Она должна быть доступна web-серверу и двигателю сервлетов — например, можно включить ее в переменную среды CLASSPATH.
- sqltaglib.tld Этот TLD-файл является частью дистрибутива Oracle JSP и содержит в своем составе описание SQL- и других тэгов. Поместите этот файл в область, указанную директивой taglib JSP-страницы; для страницы DBQuery.jsp это /WEB-INF/sqltaglib.tld.

- `oraclexmlsql.jar` Этот jar-файл необходим при использовании выходных данных XML в тэге `dbQuery`. Эта библиотека является частью дистрибутива Oracle XDK (его можно загрузить с web-сайта technet.oracle.com сети Oracle Technology Network) и содержит в своем составе служебные классы XML SQL для форматирования строк результирующего множества JDBC с помощью XML. Сделайте эту библиотеку доступной web-серверу при помощи CLASSPATH.

Кроме того, необходимо установить и сконфигурировать Oracle JSP для web-сервера (см. приложение А). Результат выполнения `DBQuery.jsp` после правильной установки изображен на рис. 12.7.

Рис. 12.7. Результат выполнения `DBQuery.jsp`



Тэги JML

Помимо SQL-тэгов, Oracle JSP предоставляет и другие служебные тэги для удобства написания сценариев и для повторения операций (итерации). Эти тэги называются *тэгами JML* (JSP Markup Language – *язык разметки JSP*). Рассмотрим один из тэгов JML – `transform`.

Тэг `transform`

Этот тэг используется при преобразовании XML-документов с помощью таблиц стилей XSL (по XML и XSL вышло немало книг, например "Professional XML" Марка Бирбека и др., издательство Wrox Press). XSL (Extensible Stylesheet Language – расширяемый язык таблиц стилей) является спецификацией, дополняющей стандарт XML. Сама по себе таблица стилей XSL – это XML-документ, применяемый для форматирования или изменения структуры другого XML- документа, обычно для его визуального представления в браузере. Для управления структурой XML-документа используются правила соответствия древовидным шаблонам. Например, XML-данные, генерируемые как результат исполнения SQL-запроса тэгом `dbQuery` (см. выше), можно сформатировать при помощи таблицы стилей XSL в виде HTML-таблицы со строками и столбцами для отображения в браузере.

Тэг `transform` обеспечивает применение таблицы стилей XSL к части JSP-страницы, заключенной между начальным и конечным тэгами. Синтаксис этого тэга:

```
<jml:transform href="ссылка_на_таблицу_стилей_XSL" >
... Вложенное тело, содержащее или генерирующее XML ...
</jml:transform >
```

Тэг имеет один обязательный атрибут — href. Этот атрибут ссылается на таблицу стилей XSL, которая может быть статическим или динамически генерируемым (например, при помощи другой JSP или сервлета) файлом. Таблица стилей указывается маршрутом относительно приложения или страницы либо полностью определенным URL. Имя таблицы стилей может быть также выражением этапа запроса, динамически вычисляемым с помощью синтаксиса JSP-выражений. Здесь используется префикс jml, при условии, что он был указан в директиве taglib JSP-страницы.

На JSP-странице может находиться несколько XML-блоков, и таблицы стилей с помощью тэга transform можно применять по отношению к одному или нескольким таким блокам. Иногда строка символов XML задается как константа, но обычно она генерируется динамически, например, при выполнении SQL- запроса с помощью тэга dbQuery с атрибутом output="XML". Пример такого использования тэга transform приведен в листинге 12.2.

Листинг 12.2. Применение таблицы стилей XSL с тэгом transform

```
<!-- JSP-страница ApplyXSL
  -- Таблицы стилей применяются по отношению к XML-блокам JSP-
  -- страницы с использованием тэга transform из библиотеки тэгов
  -- Oracle JSP.
  -->

<?xml version="1.0"?>
<%@ taglib uri="/WEB-INF/jml.tld" prefix="jml" %>
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="ora" %>
  <ora:dbOpen URL="jdbc:oracle:oci8:@"
              user="jspuser"
              password="jsp" >
   <jml:transform href="mystyle1.xsl" >
       <ora:dbQuery output="XML">
            SELECT * FROM PURCHASE_LIST
       </ora:dbQuery>
   </jml:transform>
   ... ...
   <jml:transform href="/styles/mystyle2.xsl" >
       <!-- другой статический или динамический XML-блок ->
       ... ...
   </jml:transform>
  </ora:dbOpen>
```

В этой программе используются тэги как из библиотеки SQL-тэгов, так и из библиотеки JML-тэгов, что объявляется на странице директивой taglib. Эта JSP содержит два XML-блока. Первый генерируется тэгом dbQuery и обрабатывается с помощью страницы стилей mystyle1.xsl из того же каталога, что и JSP. Второй XML-блок обрабатывается с применением файла mystyle2.xsl, находящегося в каталоге styles относительно корня web-приложения.

Реализация специального тэга

Как видно из приведенных примеров, использование специального тэга — хороший способ создания модульных приложений. Можно описать компонент Java, выполняющий определенную функцию, и разрешить его вызов при помощи специального JSP-тэга. Обычно специальные тэги компонуются в виде библиотеки, переносимой в среде различных реализаций JSP. В этом разделе рассказывается о том, как действует специальный тэг, и создаются простые специальные тэги.

Схема библиотеки тэгов

Описанная в спецификации JSP 1.1. схема расширения тэгов действует следующим образом. Каждому специальному тэгу соответствует Java-класс *обработчика тэга* (tag handler), который исполняет функцию тэга. Текст класса

обработчика тэга меняется в зависимости от наличия тела у тэга. Примером тэга только с атрибутами и без тела является тэг dbClose (см. выше). У тэга dbQuery тело есть — SQL-запрос, и действием тэга является исполнение этого запроса. В зависимости от необходимости обработки тела, класс обработчика тэга реализует один из следующих двух стандартных интерфейсов:

- Интерфейс javax.servlet.jsp.tagext.Tag, если тело тэга обрабатывать не нужно.
- Интерфейс javax.servlet.jsp.tagext.BodyTag, если тело тэга между начальным и конечным тэгами обрабатывать нужно. Этот класс расширяет интерфейс Tag.

Интерфейс Tag содержит такие методы, как doStartTag() и doEndTag(), которые вызываются двигателем JSP при обнаружении начального и конечного тэгов. В интерфейсе BodyTag есть дополнительные методы, например doInitBody() и doAfterBody(), вызываемые в соответствующие моменты во время обработки тела. Кроме того, существуют и такие методы, как setPageContext() и setParent(), вызываемые JSP-программой для установки соответствующих свойств экземпляра обработчика тэга для текущего объекта PageContext JSP-страницы и экземпляра обработчика родительского тэга (для внешнего тэга, если таковой имеется).

После создания класса обработчика тэга его необходимо внедрить при помощи файла *дескриптора библиотеки тэгов* (TLD, tag library descriptor), который указывается в JSP-программе директивой taglib. Этот файл пишется на XML и по определению имеет расширение .tld. В нем содержатся сведения о внедряемой библиотеке тэгов и о каждом тэге библиотеки. Эти сведения используются JSP-транслятором для определения действия, которое необходимо предпринять при обнаружении на JSP-странице тэга из библиотеки. Каждому тэгу в TLD-файле соответствует запись с информацией о его применении:

- Имя тэга
- Имя соответствующего класса обработчика тэга
- Имена атрибутов и их характеристики (например, необязательный или обязательный)
- Информация о способе обработки тела тэга (если оно есть)
- Класс TagExtraInfo (TEI) для всех переменных, установленных тэгом

Пример TLD-файла приводится в следующем разделе.

Во время запроса экземпляр обработчика тэга создается при обнаружении специального тэга. Двигатель JSP устанавливает свойства этого экземпляра, в том числе объект PageContext для JSP-страницы, на которой указан специальный тэг, и объект обработчика родительского тэга, если данный специальный тэг вложен во внешний специальный тэг.

Теперь реализуем и внедрим два простых тэга: один без тела, а другой с телом.

Реализация тэга без тела Для тэга без тела класс обработчика реализует интерфейс Tag. Этот интерфейс содержит два метода: doStartTag() и doEndTag(), вызываемых JSP-программой при обнаружении начального и конечного тэгов. Сигнатуры этих методов:

```
public int doStartTag()
    throws javax.servlet.jsp.tagext.JspTagException;
public int doEndTag()
    throws javax.servlet.jsp.tagext.JspTagException;
```

Помимо этих, в интерфейсе Tag присутствуют и другие методы, обеспечивающие взаимодействие класса обработчика тэга с JSP. Это методы setPageContext(), setParent(), getParent() и release(). Метод setPageContext() устанавливает текущий объект PageContext в обработчике тэга и вызывается JSP-программой до метода doStartTag(). Методы setParent() и getParent() используются для установки и получения экземпляра родительского (внешнего) тэга для

обработчика. Метод release() вызывается после метода doEndTag() для очистки состояния класса обработчика тэга. Сигнатуры этих методов приведены в приложении D.

Класс обработчика тэга, который создается программистом тэгов, должен реализовывать эти методы надлежащим образом. Функция тэга реализуется методом doStartTag (), а все операции, выполняемые после обработки и во время очистки,— методом doEndTag(). Метод doStartTag() возвращает целочисленное значение — одну из следующих трех констант:

- SKIP_BODY, если у тэга нет тела или если обработку тела следует пропустить. Это значение возвращается стандартной реализацией метода doStartTag() в классе TagSupport.
- EVAL_BODY_INCLUDE, если у тэга есть тело, но его не должен обслуживать обработчик тэга. Это возвращаемое значение говорит о том, что текст тела следует включить в выходной поток JSP.
- EVAL_BODY_TAG, если тело тэга должен обслужить сам обработчик. Эта константа описана в интерфейсе BodyTag (см. ниже). Возврат данного значения методом doStartTag() запрещается для тэга без тела.

Метод doEndTag() возвращает одну из двух целочисленных констант:

- SKIP_PAGE, если остальную часть страницы следует пропустить.
- EVAL_PAGE, если обработку страницы следует продолжить после тэга. Это значение возвращается стандартной реализацией метода doEndTag() класса TagSupport.

Класс обработчика тэга обычно расширяет стандартный служебный класс javax.servlet.jsp.tagext.TagSupport, который описан в спецификации JSP. Об использовании этого класса рассказывается ниже.

Создание специального тэга dateTime Спроектируем, реализуем, внедрим и выполним специальный тэг, отображающий при вызове дату или время. Общий синтаксис этого тэга можно определить следующим образом:

```
<mytag:dateTime [ output="DATE"|"TIME"|"DATETIME" ]
                [ format="SHORT"|"MEDIUM"|"FULL"|"LONG" ] />
```

Тэг dateTime обладает двумя атрибутами: output и format. Атрибут output указывает, что нужно печатать: дату, время или и то и другое. По умолчанию печатается только дата. Атрибут format описывает стиль отображения даты и времени. Возможными значениями format являются SHORT, MEDIUM, FULL и LONG, что соответствует константам, описанным в классе java.text.DateFormat. В режиме SHORT выходные данные форматируются в виде числового значения, например 09.15.00 для даты. В режиме MEDIUM используется более подробный формат: Aug 15, 2000. Формат LONG длиннее, чем MEDIUM: August 15, 2000, а FULL — это полное описание даты или времени, включающее часовой пояс. По умолчанию стилем форматирования является MEDIUM.

Программный тест класса обработчика тэга DateTimeTag.java:

```
/* Имя программы:    DateTimeTag.java
 * Назначение:       Распечатать дату и/или время в указанном формате.
 */
package chapter12;

import javax.servlet.jsp.*;          // стандартная jsp-библиотека
import javax.servlet.jsp.tagext.*;   // стандартная библиотека расширения jsp-тэгов
import java.util.Date;
import java.text.DateFormat;

public class DateTimeTag
  extends TagSupport

{
```

```
  private String output = "DATE";                // по умолчанию DATE
  private int format = DateFormat.MEDIUM;        // по умолчанию стиль MEDIUM

  public void setOutput(String output)
  { // установим выходные данные: TIME, DATE или DATETIME
    this.output = output;
  }

  public void setFormat(String format)
  { // установим формат: SHORT, MEDIUM, FULL или LONG
    if (format.equalsIgnoreCase("MEDIUM"))
        this.format = DateFormat.MEDIUM;
    else if (format.equalsIgnoreCase("SHORT"))
        this.format = DateFormat.SHORT;
    else if (format.equalsIgnoreCase("LONG"))
        this.format = DateFormat.LONG;
    else if (format.equalsIgnoreCase("FULL"))
        this.format = DateFormat.FULL;
  }

  public int doStartTag() throws JspTagException
  { // этот тэг не имеет тела
  return SKIP_BODY;
  }

  public int doEndTag() throws JspTagException
  {
    try {
        // сначала установим формат
        DateFormat df = null;
        if (output.equalsIgnoreCase("DATE"))
            df = DateFormat.getDateInstance(format);
        else if (output.equalsIgnoreCase("TIME"))
          df = DateFormat.getTimeInstance(format);
        else if (output.equalsIgnoreCase("DATETIME"))
          df = DateFormat.getDateTimeInstance(format, format);

        // выведем дату или время в выходной поток
        pageContext.getOut().write(df.format(new Date()));
    }
    catch (java.io.IOException e) {
        throw new JspTagException ("Error in datetime tag:" +
                              e.getMessage());
    }
    return EVAL_PAGE; // продолжим обработку страницы
  }
}
```

Класс `DateTimeTag` расширяет служебный класс `TagSupport` и реализует методы `setOutput()` и `setFormat()` для установки двух своих атрибутов, а также методы `doStartTag()` и `doEndTag()` для выполнения функции тэга. Метод `doStartTag()`возвращает константу `SKIP_BODY`, так как у этого тэга тела нет. Метод `doEndTag()` создает объект `DateFormat` в соответствии с указанным стилем и выводит текущие дату и время в выходной поток объекта `PageContext`.

Компиляция, внедрение и тестирование тэга `dateTime` Для выполнения тэга `dateTime` необходимо скомпилировать его, разместив генерируемый класс в нужной области web-сервера. Затем необходимо создать для тэга файл дескриптора библиотеки тэгов (TLD). В заключение можно написать JSP для вызова тэга. Ниже эти операции показаны применительно к серверу TOMCAT:

1. Скомпилируем `DateTimeTag.java`.

Файл DateTimeTag.java можно скомпилировать с помощью приведенного ниже сценария оболочки. В этом сценарии предполагается, что используется сервер Tomcat и что генерируемый файл класса размещается в каталоге $TOMCAT_HOME/webapps/examples/WEB_INF/classes (об установке сервера Tomcat рассказывается в приложении E).

```
#!/bin/sh
TOMCAT_HOME=/private/jakarta-tomcat
JDK_HOME=/usr/local/packages/jdk1.2.2
APP_HOME=${TOMCAT_HOME}/webapps/examples
CLASSPATH=${TOMCAT_HOME}/lib/servlet.jar:${CLASSPATH}
exec ${JDK_HOME}/bin/javac -d ${APP_HOME}/WEB-INF/classes \
     -classpath ${CLASSPATH} \
     ${APP_HOME}/jsp/chapter12/DateTimeTag.java
```

2. Опишем тэг dateTime в файле дескриптора библиотеки тэгов.

Ниже показан XML-файл mytaglib.tld, описывающий тэг dateTime и его атрибуты. Важные участки выделены жирным шрифтом. Первая часть TLD-файла соответствует стандартному формату спецификации JSP, а во второй содержатся сведения о тэге: его имя, имена его атрибутов и вид каждого атрибута (обязательный или необязательный). Этот файл необходимо поместить в область web-сервера, указанную директивой taglib JSP-страницы, вызывающей этот тэг. Мы напишем JSP-страницу на этапе тестирования тэга dateTime.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
     PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
     "web-jsptaglib_1_1.dtd">
<!-- дескриптор библиотеки тэгов для тэга dateTime -->
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>mytags</shortname>
  <info>
    My simple tab library
  </info>
  <!-- Запись для тэга dateTime -->
  <tag>
    <name>dateTime</name>
    <tagclass>chapter12.DateTimeTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>
      Prints the current date and/or the time in different formats.
    </info>
    <attribute>
      <name>output</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>format</name>
      <required>false</required>
    </attribute>
  </tag>
</taglib>
```

3. Протестируем тэг dateTime.

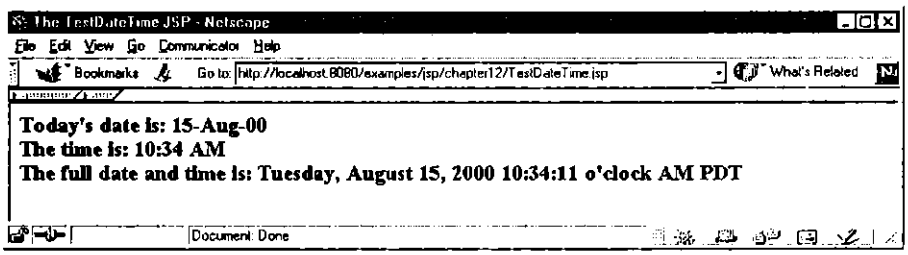
Эта простая JSP с именем TestDateTime.jsp сначала импортирует файл mytaglib.tld для тэга dateTime при помощи директивы taglib, а затем вызывает тэг в трех местах с разными параметрами:

```
<!-- Имя программы:    TestDateTime.jsp
  -- Назначение:       Протестировать специальный тэг dateTime.
  -->
```

```
<%@ taglib uri="/WEB-INF/mytaglib.tld" prefix="mytag" %>
<HTML>
  <HEAD><TITLE> The TestDateTime JSP </TITLE></HEAD>
  <BODY>
    <H3>
      Today's date is: <mytag:datetime /> <BR>
      The time is: <mytag:datetime output="TIME" format="SHORT" /><BR>
      The full date and time is:
          <mytag:datetime output="DATETIME" format="FULL" />
    </H3>
  </BODY>
</HTML>
```

Установите JSP на web-сервере, следуя инструкциям, изложенным в приложении Е. Результат выполнения TestDateTime.jsp представлен на рис. 12.8. Как видим, специальные тэги могут с успехом скрывать большую часть логики вычислений от JSP, используя класс обработчика в качестве компонента.

Рис. 12.8. Выходные данные TestDateTime.jsp



Создание тэга с телом Рассмотрим специальный тэг, тело которого должен обслуживать обработчик. Класс обработчика тэга реализует стандартный интерфейс javax.servlet.jsp.tagext.BodyTag, который расширяет интерфейс javax.servlet.jsp.tagext.Tag. В дополнение к методам интерфейса Tag в интерфейсе BodyTag определены методы doInitBody() и doAfterBody(). Метод doInitBody() вызывается JSP-страницей перед вычислением тела тэга, а метод doAfterBody () — каждый раз при вычислении тела тэга (на самом деле тело может вычисляться несколько раз, например в итеративном тэге). Сигнатуры этих методов:

```
public void doInitBody()
          throws javax.servlet.jsp.tagext.JspTagException;
public int doAfterBody()
          throws javax.servlet.jsp.tagext.JspTagException;
```

Специальные тэги, которые должны обрабатывать тело, обычно расширяют стандартный класс поддержки javax.servlet.jsp.tagext.BodyTagSupport. Как и в случае с тэгами без тел, метод doStartTag() описывается в классе обработчика тэга для исполнения соответствующего начального действия тэга, а метод doEndTag() переопределяется, предоставляя тэгу логику операций, выполняемых после обработки. Для класса обработчика тэга, обслуживающего тело (т.е. реализующего интерфейс BodyTag), метод doStartTag() не может возвращать значение EVAL_BODY_INCLUDE. Методы doInitBody() и doAfterBody() тоже реализуются классом обработчика тэга и используются для обработки тела. Методы вызываются в следующем порядке:

- Исполняется метод doStartTag(). Он может возвращать значение SKIP_BODY или EVAL_BODY_TAG.
- Если doStartTag() возвращает EVAL_BODY_TAG, вызываются методы doInitBody() и doAfterBody(). В случае выдачи SKIP_BODY эти методы не вызываются (например, для специального тэга, который подключает тело по некоторому условию, а условие в методе doStartTag() определяется как ложное (false)). Вместо них вызывается метод doEndTag().

- Если метод doAfterBody() возвращает значение EVAL_BODY_TAG, он вызывается еще раз для новой итерации процесса обработки тела. Таким образом, для итеративных тэгов метод doAfterBody() вызывается несколько раз до тех пор, пока не будет возвращено значение SKIP_BODY.
- Когда после обработки тела обнаруживается конечный тэг, исполняется метод doEndTag(). Этот метод возвращает или SKIP_PAGE, или EVAL_PAGE в зависимости от того, что нужно сделать: завершить или продолжить обработку страницы.
- Метод release() всегда вызывается для очистки состояния экземпляра обработчика тэга независимо от того, какое значение возвращает метод doEndTag(): SKIP_PAGE или EVAL_PAGE.

В интерфейсе BodyTag присутствует дополнительный метод setBodyContent(), который в качестве аргумента принимает объект BodyContent, используемый при обработке тела тэга (интерфейс BodyTag и класс BodyContent описаны в приложении D). Этот метод может применяться двигателем JSP для передачи описателя BodyContent экземпляру обработчика тэга. Соответствующим get-методом является getBodyContent(), который возвращает объект BodyContent. Этот объект будет использоваться для обработки тела специального тэга makeMenu (см. ниже).

Создание тела специального тэга makeMenu Спроектируем и реализуем специальный тэг makeMenu, который создает HTML-меню выбора из столбца SQL-запроса, указанного в теле тэга. Это меню используется в HTML-форме для выбора определенного значения, например номера заказа в таблице PURCHASE_LIST схемы Purchase Order. Тэг makeMenu будет выполнять следующие операции:

- Считывать текст SQL-запроса, вложенный между начальным и конечным тэгами.
- Устанавливать соединение с базой данных и исполнять SQL-запрос.
- Формировать список значений пунктов HTML-меню из первого столбца запроса.
- Выводить этот список в выходной поток.

Текст программы MenuTag.java приводится ниже. Стоит отметить, что данный класс расширяет описанный выше класс BodyTagSupport и наследует из него стандартные реализации большинства методов, за исключением метода doAfterBody(). Этот метод используется для обработки текста тела и выполнения перечисленных операций, специфичных для тэга.

```
/* Имя программы:    MenuTag.java
 * Назначение:       Создать HTML-меню выбора из запроса базы данных
 */

package chapter12;

import javax.servlet.jsp.*;          // стандартная jsp-библиотека
import javax.servlet.jsp.tagext.*;   // стандартная библиотека расширения jsp-тэгов
import java.sql.*;

public class MenuTag
  extends BodyTagSupport
{
  int maxRows = 100;        // (см. пояснение 1)
  public void setMaxRows(int maxRows) {
    this.maxRows = maxRows;
  }

  /** Извлечем String из вложенного тела, установим соединение,
    * выполним запрос и напечатаем HTML-меню
   */
```

```
  public int doAfterBody() throws JspTagException
  { // Получим тело тэга в виде строки символов
    BodyContent body = getBodyContent();     // (см. пояснение 2)
    String query = body.getString().trim();
    try {
        DriverManager.registerDriver(
             new oracle.jdbc.driver.OracleDriver());
        Connection conn = DriverManager.getConnection(   // (см. пояснение 3)
                          "jdbc:oracle:oci8:@", // URL
                          "jspuser", "jsp");   // Пользователь/пароль
        Statement stmt = conn.createStatement();
        ResultSet rset = stmt.executeQuery(query);
        if (rset.next()) { // Результирующее множество не пустое
          body.getEnclosingWriter().println(makeMenu(rset));
                                  // (см. пояснение 4)
        }
    } catch (Exception e) {
      throw new JspTagException("Error in Menu tag: " +
                          e.getMessage());
    }
    finally {try {conn.close();} catch (Exception e) {}}
    return EVAL_PAGE; // Продолжим обработку страницы
  }

  /* Создадим список пунктов меню */
  private String makeMenu(ResultSet rset) throws Exception {
    int num = 0;
    StringBuffer sb = new StringBuffer();
    do {  // (см. пояснение 5)
        sb.append("<OPTION> " + rset.getString(1) + " </OPTION>\n");
        num++;
    } while (rset.next() && (num <= maxRows)); rset.close();
    return sb.toString();
  }
}
```

Пояснения к MenuTag.java:

1. Тэг makeMenu имеет один необязательный атрибут maxRows, который определяет число строк, считываемых в результирующем множестве для запроса. Для установки значения этого свойства из JSP-программы используется метод setMaxRows().
2. Этот оператор получает объект BodyContent для тэга, как ранее было установлено исполняющей системой JSP. Этот объект используется для получения текста тела с помощью метода getString(). В тексте тела удаляются конечные пробелы, чтобы получился текст SQL-запроса.
3. Этот оператор создает соединение с базой данных. Вероятнее всего, что на практике соединение не будет открываться непосредственно в классе MenuTag, а будет получено при помощи зерна соединений в HTTP-объекте session или зерна диспетчера соединений в области действия application. Для сохранения простоты логики оно программируется так же, как в предыдущем примере.
4. Этот оператор извлекает объект Writer из объекта BodyContent с помощью метода getEnclosingWriter(). Объект Writer используется для вывода результата обработки тела тэга.
5. Это цикл, обрабатывающий каждую строку результирующего множества и помещающий значение первого столбца между HTML-тэгами <OPTION> и </OPTION> для печати меню в HTML-операторе SELECT (см. ниже программу OrderMenu.jsp).

Компиляция, внедрение и тестирование специального тэга makeMenu
Перед вызовом тэга makeMenu из JSP-программы его необходимо скомпилировать и внедрить. Нужно поместить скомпилированный класс в соответствующую область web-сервера и создать для тэга запись в дескрипторе библиотеки тэгов (TLD). Эти операции выполняются для сервера Tomcat:

1. Скомпилируем MenuTag.java.

   Файл MenuTag.java можно скомпилировать в среде Tomcat с помощью приведенного ниже сценария оболочки (об установке сервера Tomcat рассказывается в приложении E).

```
#!/bin/sh
TOMCAT_HOME=/private/jakarta-tomcat
JDK_HOME=/usr/local/packages/jdk1.2.2
APP_HOME=${TOMCAT_HOME}/webapps/examples
CLASSPATH=${ORACLE_HOME}/jdbc/lib/classes12.zip
CLASSPATH=${TOMCAT_HOME}/lib/servlet.jar:${CLASSPATH}
exec ${JDK_HOME}/bin/javac -d ${APP_HOME}/WEB-INF/classes \
     -classpath ${CLASSPATH} \
     ${APP_HOME}/jsp/chapter12/MenuTag.java
```

2. Опишем тэг makeMenu в файле mytaglib.tld.

   В файл mytaglib.tld, созданный ранее для тэга dateTime, можно добавить приведенную ниже запись. Она указывается между внешними элементами <taglib> и </taglib> файла. Главным отличием этой записи от записи для тэга dateTime является значение элемента <bodycontent>. Здесь для него устанавливается значение JSP, а не empty, как в тэге dateTime. Значение JSP говорит о том, что в теле будет содержаться программный текст JSP, возможно, с включением скриптлетов и выражений, которые нужно будет вычислять. Таким образом, SQL-запрос в теле тэга makeMenu может быть сформирован во время выполнения запроса.

```
<taglib>


<!- тэг makeMenu ->
  <tag>
    <name>makeMenu</name>
    <tagclass>chapter12.MenuTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
      Makes an HTML menu out of the first column of a
      SQL query in the tag body
    </info>
    <attribute>
      <name>maxRows</name>
      <required>false</required>
    </attribute>
  </tag>
</taglib>
```

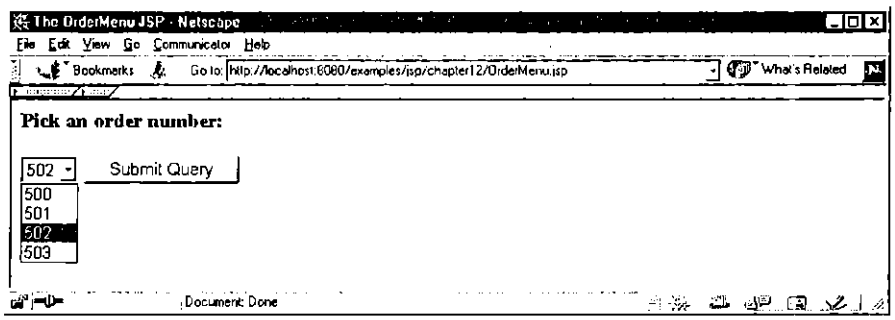
3. Создадим JSP для вызова тэга makeMenu.

Третий и последний этап — создание JSP, которая будет вызывать новый специальный тэг makeMenu. Приведенная ниже JSP-страница OrderMenu.jsp использует этот тэг для создания меню из значений столбца requestno таблицы PURCHASE_LIST. Это меню размещается между HTML-тэгами <SELECT> и </SELECT> в границах HTML-формы и служит для выбора значения в отображаемом списке. Выбранному значению дается имя orderNumber, оно обрабатывается в HTML-форме и направляется на JSP-страницу CallOrderEJB.jsp (см. выше) для запроса характеристик заказа с выбранным номером. Здесь можно выполнить любое другое действие; страница CallOrderEJB.jsp приводится лишь в качестве иллюстрации. Заметим, что SQL-запрос формируется во время выполнения запроса, что делает тэг по-настоящему гибким средством.

```
<!-- Имя программы: OrderMenu.jsp
  -- Назначение: С помощью специального тэга makeMenu выбрать
  -- заказ.
  -->
 <%@ taglib uri="/WEB-INF/mytaglib.tld" prefix="mytag" %>
<HTML>
  <HEAD><TITLE> The OrderMenu JSP </TITLE></HEAD>
  <BODY>
    <% String columnName = "requestno"; %>
    <H3> Pick an order number: </H3>
    <FORM ACTION="CallOrderEJB.jsp">
        <SELECT Name="orderNumber" >
          <mytag:makeMenu>
            SELECT <%= columnName %> FROM PURCHASE_LIST
          </mytag:makeMenu>
    </SELECT>
    <INPUT TYPE="submit" Name="Query order" >
    </FORM>
  </BODY>
</HTML>
```

Результат выполнения OrderMenu.jsp приведен на рис. 12.9.

Рис. 12.9. Выходные данные OrderMenu.jsp



Создание переменных сценариев в специальных тэгах Для обработчиков специальных тэгов можно создавать *переменные сценариев* (scripting variables), используемые этим или другими тэгами, а также другими элементами сценариев страницы. Например, в описанном выше тэге dbOpen переменная сценариев представлена через его атрибут connId. Впоследствии эту переменную можно применять в тэге dbQuery для получения описателя соединения.

Все переменные сценариев, описанные в специальном тэге, нужно указать в классе *дополнительной информации тэга* (TEI, tag extra information), который является подклассом стандартного абстрактного класса javax.servlet.jsp.tagext.TagExtraInfo. TEI-класс, применяемый специальным тэгом, указывается в TLD-файле тэга, как определено в директиве taglib на JSP-странице. Двигатель JSP пользуется экземплярами TEI во время трансляции. В TEI-классе есть метод getVariableInfo(), считывающий имена и типы переменных сценариев, которые будут определены во время запроса. При компиляции JSP этот метод вызывается JSP-транслятором для получения имени, типа Java и области действия переменной сценариев, а также для определения, первый ли раз она объявляется. За дополнительными сведениями о переменных сценариев и об их применении в специальных тэгах обращайтесь к "JavaServer Pages 1.1 Specification" компании Sun Microsystems.

В качестве простого примера рассмотрим усовершенствованный вариант тэга dateTime, который сохраняет вычисленные дату и время в переменной сценариев, чтобы другие тэги впоследствии могли пользоваться этими значениями:

```
<mytag:dateTime theTime="имя_переменной"
                [ output="DATE"|"TIME"|"DATETIME" ]
                [ format="SHORT"|"MEDIUM"|"FULL"|"LONG" ] />
```

Предлагаем читателю создать этот специальный тэг и его TEI-класс. Кроме того, можно создать специальные JSP-тэги для упрощения работы с компонентами EJB и CORBA. Другими примерами усовершенствованных специальных тэгов являются тэги с вложениями и объединяющие тэги. За дополнительными сведениями и примерами обращайтесь к учебным материалам Sun Microsystems: http://java.sun.com/products/jsp/tutorial/TagLibrariesTOC.html.

Итоги

В этой главе рассказывалось о построении основанных на JSP web-приложений, использующих различные программные компоненты, такие как зерна Enterprise JavaBean и объекты CORBA, внедряемые в Oracle8*i*. Были созданы JSP-страницы, которые принимают параметры, вводимые на экране браузера, с их помощью вызывают объекты EJB и CORBA для исполнения бизнес-логики и выдают результаты пользователю. В бизнес-компонентах концентрируется логика обработки, например исполнение операций в базе данных, а JSP-страница представляет и применяет эти объекты.

Кроме того, говорилось об использовании библиотек JSP-тэгов. Модель библиотек позволяет расширить набор JSP-тэгов собственными специальными тэгами или стандартным образом подключить библиотеки специальных тэгов других фирм. С помощью такой схемы можно создавать модульные Java-компоненты, а затем раскрывать их функциональные возможности посредством удобного синтаксиса тэгов. В JSP-реализации Oracle предлагается набор служебных тэгов, которые позволяют открывать соединение с базой данных, исполнять SQL-запросы, форматировать генерируемые XML- результаты при помощи таблиц стилей XSL и т.д. Был рассмотрен процесс импорта и применения этих тэгов в JSP-программе. В конце главы изложены принципы действия схемы расширения тэгов и созданы специальные тэги, один из которых отображает дату и время в указанном формате, а второй формирует HTML-меню из столбца SQL-запроса.

Таким образом, эффективные web-приложения можно строить, "вставляя" программные компоненты в JSP-страницы и сервлеты. В web-приложении можно без труда объединять различные технологии компонентного программирования, тем самым в полной мере используя мощь и универсальность платформы Java.

ЧАСТЬ V

Приложения

ПРИЛОЖЕНИЕ А

Краткий справочник по сервлетам и общие сведения об API сервлетов

В этом приложении содержится краткий справочник по сервлетам и приводятся общие сведения о Servlet 2.2 API.

Краткий справочник по сервлетам

Для реализации и выполнения сервлета Java необходимо:

- Написать программный текст сервлета
- Скомпилировать программный текст сервлета в виде байтового кода Java
- Внедрить скомпилированный класс сервлета на web-сервере
- Вызвать внедренный сервлет при помощи браузера

Создадим сервлеты `HelloServlet` и `JDBCQueryServlet`. Для этих двух программ произведем перечисленные выше операции и затем выполним программы. Напомним, что внедрение программы `HelloServlet` рассматривалось в разделе "Написание простого сервлета" главы 2. Сервлет `JDBCQueryServlet` обращается с SQL-запросом к базе данных Oracle с помощью JDBC.

Поскольку этапы установки сервлета для разных web-сред различны, для иллюстрации этапов внедрения воспользуемся сервером Tomcat. Этапы установки и конфигурирования Tomcat и других web-серверов описаны в приложении Е.

Написание сервлетов

Сервлет, обрабатывающий HTTP-запросы, обычно расширяет абстрактный класс `javax.servlet.http.HttpServlet`, который, в свою очередь, расширяет абстрактный класс `javax.servlet.GenericServlet` (см. ниже). В классе сервлета описываются, как правило, следующие методы:

- Метод `init()` для инициализации экземпляра сервлета. Если отсутствует, стандартная (тривиальная) реализация этого метода наследуется из эталонного класса `javax.servlet.GenericServlet`.
- Метод `destroy()` для уничтожения экземпляра сервлета. Если отсутствует, стандартная (тривиальная) реализация этого метода наследуется из эталонного класса `javax.servlet.GenericServlet`.
- Методы `doGet()` и/или `doPost()` для обработки HTTP-запросов `GET` и `PUT`. Менее распространены методы `doPut()` и `doHead()` для HTTP-запросов PUT и HEAD соответственно. В зависимости от HTTP-запроса нужный метод `do...()` выбирается методом `service()` родительского класса `javax.servlet.http.HttpServlet`. Аргументами всех этих методов являются HTTP-объекты `request` и `response`.

Написание HelloServlet

`HelloServlet` — простой сервлет, распечатывающий приветствие и время. По умолчанию применяется HTTP-метод `GET`. В качестве параметра запроса в URL указывается имя пользователя, которое можно считывать методом `request.getParameter()`. Этот метод принимает в качестве аргумента имя параметра и возвращает его значение. Для сервлета `HelloServlet` не нужно выполнять никаких специальных операций во время инициализации или остановки. Поэтому методы `init()` и `destroy()` необязательны; необходим лишь метод `doGet()`. Программный текст `HelloServlet`:

```
/** Имя программы: HelloServlet.java
 ** Назначение: Вывод приветствия пользователю и текущего времени.
 **/
package appendixA;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;

public class HelloServlet extends HttpServlet {
  public void doGet ( HttpServletRequest request,
                      HttpServletResponse response)
              throws ServletException, IOException {
   // Получим объект записи выходных данных из ответа
   PrintWriter out= response.getWriter();
   // Установим тип содержимого ответа как HTML-текст
   response.setContentType("text/html");
   // Сгенерируем содержимое ответа
   out.println("<HTML>");
   out.println("<BODY>");
   out.println("<B><P>Hello " + request.getParameter("user") +
               ", how are you? ");
   out.println("<P>The current time is " + new java.util.Date());
   out.println("<P>Have a nice day! </B>");
   out.println("</BODY>");
   out.println("</HTML>");
   out.close();
  }
}
```

Написание JDBCQueryServlet

Создадим сервлет JDBCQueryServlet. Он иллюстрирует два момента:

- Использование HTML-форм для ввода параметров запроса
- Использование параметров HTTP-запроса для выполнения JDBC-запроса, обращенного к базе данных Oracle

Непосредственная функция этого сервлета – принятие условия поиска от пользователя и поиск имени и номера телефона в таблице EMPLOYEE_LIST схемы Purchase Order. Эта схема описана во введении к книге и воспроизводится здесь для удобства работы:

```
CREATE TABLE EMPLOYEE_LIST(
  employeeno     NUMBER(7),
  deptno         NUMBER(5),
  type           VARCHAR2(30),
  lastname       VARCHAR2(30),
  firstname      VARCHAR2(30),
  phone          VARCHAR2(10));
```

Разделим программный текст сервлета на несколько методов:

- init() Этот метод устанавливает соединение с базой данных.
- destroy() Этот метод отключает соединение с базой данных.
- doPost() Этот метод обрабатывает HTTP-запрос POST.
- doQuery() Этот частный метод исполняет SQL-запрос с заданным условием.
- printResults() Этот частный метод распечатывает результаты запроса в виде HTML-таблицы.
- printForm() Этот частный метод распечатывает HTML-форму для ввода условия поиска.

Кроме того, в JDBCQueryServlet определяется метод doGet() так, чтобы можно было вызывать его через HTTP-запрос GET. Запрос такого типа генерируется при вводе URL сервлета в браузере. Метод doGet() вызывает метод doPost() с объектами request и response в качестве параметров. Приведем программный текст сервлета JDBCQueryServlet, который ищет номер телефона:

```
/** Имя программы: JDBCQueryServlet.java
 ** Назначение: Выполнение JDBC-запроса с вводом параметров запроса
 ** в HTML-форме.
 **/

package appendixA;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.sql.*;

public class JDBCQueryServlet extends HttpServlet {
  // Переменная-экземпляр для соединения с базой данных
  Connection conn = null;

  // Метод для инициализации сервлета - установка соединения с базой данных
  public void init()
    throws ServletException {
        try { // Подключимся к схеме jspuser/jsp с помощью драйвера JDBC-OCI
            DriverManager.registerDriver(new
                   oracle.jdbc.driver.OracleDriver());
            conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                               "jspuser", "jsp");
          } catch (SQLException e) {
            conn = null;
            throw new ServletException(e.getMessage());
          }
    }

  // Метод для уничтожения сервлета - отключение от базы данных
  public void destroy() {
    try {
          if (conn!= null) conn.close();
        } catch (SQLException ignore) {}
    }

  // Метод для обработки HTTP-запросов POST
  public void doPost (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // Получим объект записи выходных данных из ответа
        PrintWriter out= response.getWriter();

        // Установим тип содержимого ответа как HTML-текст
        response.setContentType("text/html");

        // Теперь распечатаем HTML-содержимое
        out.println("<HTML>");
        out.println("<HEAD> <TITLE> The JDBCQueryServlet" +
                    " </TITLE> </HEAD>");
        out.println("<BODY>");
        String condition = request.getParameter("cond");
        if (condition != null) {
          out.println("<H3> Search results for <I>" +
                      condition + " </I> </H3><B>");

          // Выполним запрос и распечатаем результаты поиска
          doQuery(condition, out);

          out.println(" </B> <HR><BR>");
        }
```

```
        // Распечатаем HTML-форму для ввода условия поиска
        printForm(out);

        // Завершив работу, закроем объект записи выходных данных
        out.close();
    }

    // Метод для обработки HTTP-запросов GET
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException
    { // Вызовем метод POST
        doPost(request, response);
    }

// Метод для исполнения SQL-запроса
// Он синхронизирован для управления доступом к соединению с базой
// данных - в любой момент времени может использоваться только один поток.
private synchronized void doQuery(String condition,
                                  PrintWriter out) {
    if (conn == null) {
        out.println("No database connection!");
        return;
    }
    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        rs = stmt.executeQuery ("SELECT firstname, lastname, phone" +
                                " FROM employee_list " +
                                (condition.length() == 0?
                                "" : ("WHERE " + condition)) +
                                " ORDER BY firstname" );

          // Отформатируем строки результата в виде HTML-таблицы
          printResult(rs, out);
        } catch (SQLException e) {
          out.println ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
        } finally { // Выполним очистку
          try {
             if (rs!= null) rs.close();      // Закроем результирующее множество
             if (stmt!= null) stmt.close(); // Закроем оператор
          } catch (SQLException ignore) { }
        }
    }

    // Метод для печати результатов запроса в виде HTML-таблицы
    private void printResult (ResultSet rs, PrintWriter out)
            throws SQLException {
        If (!rs.next()) { // Строки не найдены
            out.println("<P> No such employee!<P>\n");
            return;
        }

        // Отформатируем строки в виде HTML-таблицы
        out.println("<TABLE BORDER>");
        // Сначала распечатаем заголовки
        out.println("<TH>First Name</TH>");
        out.println("<TH>Last Name</TH>");
        out.println("<TH>Phone</TH>");

        // Затем распечатаем строки таблицы, организовав цикл для обработки
        // результирующего множества
```

```
            do { out.println("<TR>" + "<TD>" + rs.getString(1) + "</TD>"
                                    + "<TD>" + rs.getString(2) + "</TD>"
                                    + "<TD>" + rs.getString(3) + "</TD>"
                               + "</TR>");
            } while (rs.next());
            out.println("</TABLE>");
        }

        // Метод для печати HTML-формы, служащей для ввода условия поиска
        private void printForm(PrintWriter out) {
            out.println("<B>Search condition:</B>");
            out.println("<FORM METHOD=POST> ");
            out.println("<INPUT TYPE=\"text\" NAME=\"cond\" SIZE=30" +
                        " VALUE=\"firstname like 'A%'\">");
            out.println("<INPUT TYPE=\"submit\" VALUE=\"Lookup Phone\">");
            out.println(" </FORM>");
            out.println(" </BODY>");
            out.println("</HTML>");
        }
}
```

Метод doPost() проверяет, пет ли в HTTP-объекте request параметра cond. Если есть, он вызывает метод doQuery() для исполнения запроса на поиск номера телефона по указанному условию. Метод doQuery() добавляет данное условие к SQL-оператору SELECT и генерирует результирующее множество запроса. При успешном выполнении запроса он вызывает метод printResult(), форматирующий строки результирующего множества в виде HTML-таблицы. Метод printResult() циклически проходит по всем строкам, распечатывая данные. Наконец, методом doPost() вызывается метод printForm(), предоставляющий текстовое окно для ввода пользователем условия поиска номера телефона. В этом текстовом окне указано условие по умолчанию firstname like 'A%'.

Обратите внимание на то, что метод doQuery() *синхронизирован* (synchronized). Это сделано для того, чтобы доступ к разделяемому соединению с базой данных в любой момент времени имел только тот, кто вызывает сервлет (напомним, что одновременные HTTP-запросы для сервлета обрабатываются, как правило, порождением нескольких потоков (см. главу 2)). Переменная conn, представляющая разделяемое соединение с базой данных, инициализируется методом init() сервлета, а закрывается методом destroy(). Во время обработки запроса обращение к базе данных осуществляется последовательно. На практике лучшей производительности можно достичь при помощи разделяемого пула соединений с базой данных.

Условие поиска SQL-запроса вводится пользователем в HTML-форме, которая распечатывается методом printForm() сервлета. Эта форма имеет одно текстовое окно с именем cond для ввода условия и функциональную кнопку для передачи запроса. По умолчанию вызывается та же самая программа, в данном случае JDBCQueryServlet. Когда пользователь щелкает на кнопке мышью, значение, введенное в поле cond, включается в новый HTTP-запрос для вызова программы JDBCQueryServlet HTTP-методом POST. За дополнительными сведениями о работе HTML-форм обратитесь к любой книге по HTML и HTTP.

Компиляция сервлетов

Ниже описываются операции, выполняемые при компиляции программы сервлета в среде UNIX. Операции в системе Windows NT аналогичны. Для иллюстрации используется сервер Tomcat.

Компиляция HelloServlet.java

Сначала необходимо инсталлировать JDK и сервер Tomcat (см. приложение E). Установите переменную среды JDK_HOME так, чтобы она указывала на корень JDK, а переменную среды TOMCAT_HOME – чтобы она указывала на

корень Tomcat. Затем можно воспользоваться командами, приведенными в листинге А.1, для компиляции HelloServlet.java в системе UNIX.

Листинг А.1. Команды UNIX для компиляции HelloServlet

```
# В приглашении Unix введем:
${JDK_HOME}/bin/javac -g \
     -d ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes \
     -classpath ${TOMCAT_HOME}/lib/servlet.jar \
     HelloServlet.java
```

Компиляция JDBCQueryServlet.java

Этот сервлет соединяется с базой данных при помощи JDBC, и поэтому необходимы JDBC-библиотеки. Их можно найти в установленной системе Oracle. Воспользуемся командами, приведенными в листинге А.2, для компиляции JDBCQueryServlet.java в системе UNIX, где применяется JDK 1.2.

Листинг А.2. Команды UNIX для компиляции JDBCQueryServlet **с использованием JDK 1.2**

```
# В приглашении Unix введем:
${JDK_HOME}/bin/javac -g \
    -d ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes \
    -classpath ${ORACLE_HOME}/jdbc/lib/classes12.zip:\
${TOMCAT_HOME}/lib/servlet.jar \
  JDBCQueryServlet.java
```

Если в системе применяется JDK 1.1, нужно воспользоваться JDBC- библиотеками ${ORACLE_HOME}/jdbc/lib/classes111.zip, а не classes12.zip.

Внедрение сервлетов в Tomcat

Для внедрения сервлетов необходимо разместить скомпилированные классы в соответствующем каталоге web-сервера. Каталог определяется корнем web-приложения. Например, для Tomcat в качестве корневого каталога приложения examples заранее устанавливается каталог ${TOMCAT_HOME}/webapps/examples. Следовательно, в Tomcat файлы скомпилированных классов HelloServlet.class и JDBCQueryServlet.class можно разместить в каталоге $TOMCAT_HOME/webapps/examples/WEB-INF/classes. Это удобно делать на этапе компиляции при помощи параметра -d компилятора javac (см. листинги А.1 и А.2).

Вызов сервлетов

После внедрения можно вызывать сервлеты из браузера. Если сервер Tomcat работает на порту 8080 локальной машины, сервлеты можно вызвать следующими указателями URL: http://localhost:8080/examples/servlet/appendixA.HelloServlet?user=Pat и http://localhost:8080/examples/servlet/appendixA.JDBCQueryServlet.

Отметим, что в указателях URL для сервлетов задается полностью определенное имя класса, например appendixA.HelloServlet. Результат, получаемый в браузере после исполнения HelloServlet, показан на рис. А.1.

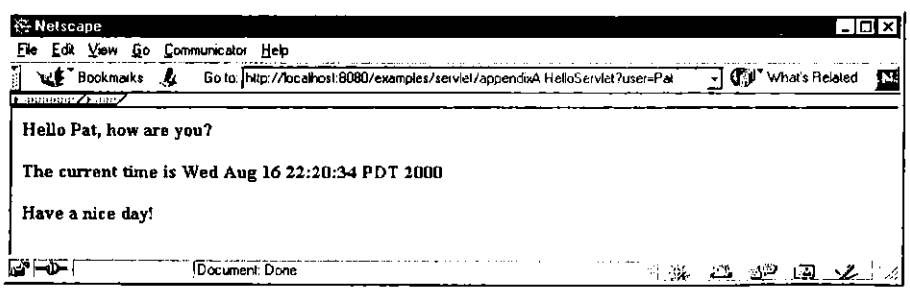


Рис. А.1. Результат исполнения HelloServlet

HTML-текст сгенерированной страницы можно вывести с помощью средства просмотра исходного текста в браузере. Например, в браузере Netscape нужно нажать кнопку View в главном меню и выбрать Page Source. HTML-текст приводится в листинге А.3.

Листинг А.3. HTML-страница, генерируемая сервлетом `HelloServlet`

```
<HTML>
<BODY>
<B><P>Hello Pat, how are you?
<P>The current time is Wed Aug 16 22:20:34 PDT 2000
<P>Have a nice day! </B>
</BODY>
</HTML>
```

Результат исполнения `JDBCQueryServlet` в браузере показан на рис. А.2. Можно изменить условие поиска в HTML-форме и выполнить другой запрос. Текст получаемой на выходе HTML-страницы приведен в листинге А.4.

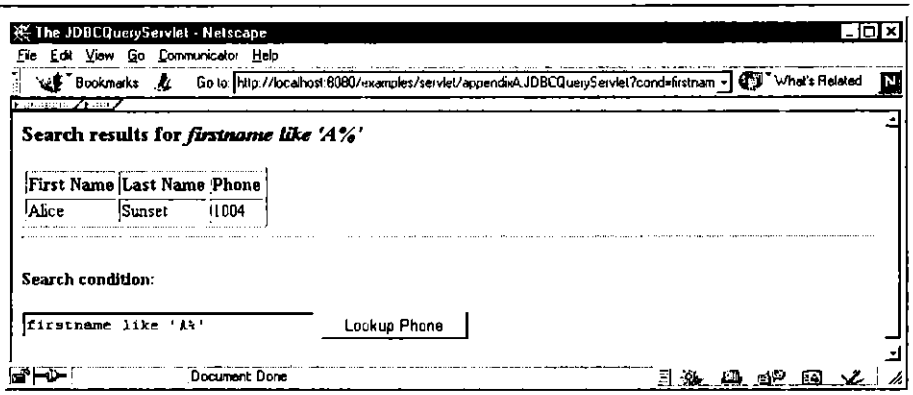


Рис. А.2. Результат исполнения запроса по умолчанию в `JDBCQueryServlet`

Листинг А.4. HTML-страница, генерируемая сервлетом `JDBCQueryServlet`

```
<HTML>
<HEAD> <TITLE> The JDBCQueryServlet </TITLE> </HEAD>
<BODY>
<H3> Search results for <I>firstname like 'A%' </I> </H3><B>
<TABLE BORDER>
<TH>First Name</TH>
<TH>Last Name</TH>
<TH>Phone</TH>
<TR><TD>Alice</TD><TD>Sunset</TD><TD>1004</TD></TR>
</TABLE>
  </B> <HR><BR>
<B>Search condition:</B>
<FORM METHOD=POST>
<INPUT TYPE="text" NAME="cond" SIZE=30 VALUE="firstname like 'A%'">
<INPUT TYPE="submit" VALUE="Lookup Phone">
    </FORM>
  </BODY>
</HTML>
```

Servlet API 2.2

В последующих разделах кратко описываются классы и интерфейсы Servlet 2.2 API. API сервлетов состоит из двух пакетов: `javax.servlet` и `javax.servlet.http`. Пакет `javax.servlet` — это общий, не зависящий от протоколов API для сервлетов, а пакет `javax.servlet.http` — API для сервлетов, основанных на HTTP. Сначала рассмотрим иерархическую структуру классов и интерфейсов в этих пакетах. Затем познакомимся с отдельными классами и интерфейсами каждого пакета.

Иерархическая структура классов

Иерархическая структура классов в пакетах javax.servlet и javax.servlet.http имеет вид:

```
class java.lang.Object
  class javax.servlet.http.Cookie (implements java.lang.Cloneable)
  class java.util.EventObject (implements java.io.Serializable)
    class javax.servlet.http.HttpSessionBindingEvent
  class javax.servlet.GenericServlet (implements java.io.Serializable,
            javax.servlet.Servlet, javax.servlet.ServletConfig)
    class javax.servlet.http.HttpServlet (implements
                                    java.io.Serializable)
  class javax.servlet.http.HttpUtils
  class java.io.InputStream
    class javax.servlet.ServletInputStream
  class javax.servlet.jsp.JspEngineInfo
  class javax.servlet.jsp.JspFactory
  class java.io.OutputStream
    class javax.servlet.ServletOutputStream
  class javax.servlet.jsp.PageContext
  class javax.servlet.jsp.tagext.TagAttributeInfo
  class javax.servlet.jsp.tagext.TagData (implements
                                    java.lang.Cloneable)
  class javax.servlet.jsp.tagext.TagExtraInfo
  class javax.servlet.jsp.tagext.TagInfo
  class javax.servlet.jsp.tagext.TagLibraryInfo
  class javax.servlet.jsp.tagext.TagSupport (implements
            java.io.Serializable, javax.servlet.jsp.tagext.Tag)
    class javax.servlet.jsp.tagext.BodyTagSupport (implements
                                    javax.servlet.jsp.tagext.BodyTag)
  class java.lang.Throwable (implements java.io.Serializable)
    class java.lang.Exception
        class javax.servlet.jsp.JspException
          class javax.servlet.jsp.JspTagException
        class javax.servlet.ServletException
          class javax.servlet.UnavailableException
  class javax.servlet.jsp.tagext.VariableInfo
  class java.io.Writer
    class javax.servlet.jsp.JspWriter
        class javax.servlet.jsp.tagext.BodyContent
```

Иерархическая структура интерфейсов

Иерархическая структура интерфейсов в пакетах javax.servlet и javax.servlet.http имеет вид:

```
interface java.util.EventListener
  interface javax.servlet.http.HttpSessionBindingListener
interface javax.servlet.http.HttpSession
interface javax.servlet.http.HttpSessionContext
interface javax.servlet.RequestDispatcher
interface javax.servlet.Servlet
  interface javax.servlet.jsp.JspPage
    interface javax.servlet.jsp.HttpJspPage
interface javax.servlet.ServletConfig
interface javax.servlet.ServletContext
interface javax.servlet.ServletRequest
  interface javax.servlet.http.HttpServletRequest
interface javax.servlet.ServletResponse
  interface javax.servlet.http.HttpServletResponse
interface javax.servlet.SingleThreadModel
interface javax.servlet.jsp.tagext.Tag
  interface javax.servlet.jsp.tagext.BodyTag
```

Классы и интерфейсы

Ниже приводится перечень интерфейсов, классов и исключительных ситуаций, составляющих Servlet API. Сигнатуры методов этих классов и интерфейсов представлены в таблице А.1. За детальными описаниями типов и их методов обращайтесь к Java Servlet Specification, v2.2 компании Sun Microsystems.

Таблица А.1. Краткие сведения о пакетах Servlet API

Пакет `javax.servlet`	Пакет `javax.servlet.http`
RequestDispatcher	HttpServletRequest
Servlet	HttpServletResponse
ServletConfig	HttpSession
ServletContext	HttpSessionBindingListener
ServletRequest	HttpSessionContext
ServletResponse	Cookie
SingleThreadModel	HttpServlet
GenericServlet	HttpSessionBindingEvent
ServletInputStream	HttpUtils
ServletOutputStream	
ServletException	
UnavailableException	

Пакет `javax.servlet`

Пакет `javax.servlet` используется в программировании сервлетов, не зависящих от применяемого протокола. Он состоит из следующих интерфейсов и классов:

```
interface RequestDispatcher
interface Servlet
interface ServletConfig
interface ServletContext
interface ServletRequest
interface ServletResponse
interface SingleThreadModel

class GenericServlet
class ServletInputStream
class ServletOutputStream
class ServletException
class UnavailableException
```

Познакомимся с каждым из этих интерфейсов и классов.

RequestDispatcher Интерфейс `RequestDispatcher` используется для подключения и перехода к таким ресурсам, как сервлеты, HTML-страницы, JSP-страницы и др. Описание интерфейса `RequestDispatcher`:

```
public interface RequestDispatcher

public void forward(ServletRequest req, ServletResponse res);
public void include(ServletRequest req, ServletResponse res);
```

Servlet Интерфейс `Servlet` – это базовый интерфейс, реализуемый всеми сервлетами и JSP-программами. Он содержит методы для инициализации и уничтожения экземпляра сервлета и для обслуживания запросов. Описание интерфейса `Servlet`:

```
public interface Servlet

public void init(ServletConfig config) throws ServletException;
public ServletConfig getServletConfig();
public void service(ServletRequest req, ServletResponse res)
    throws IOException, ServletException;
public String getServletInfo();
public void destroy();
```

ServletConfig Параметры инициализации передаются сервлету при помощи объекта ServletConfig. Описание интерфейса ServletConfig:

```
public interface ServletConfig

public ServletContext getServletContext();
public String getInitParameter(String name);
public Enumeration get InitParameterNames();
public String getServletName();
```

ServletContext Интерфейс ServletContext используется для получения информации о среде web-сервера, в которой исполняется сервлет или JSP. Описание интерфейса ServletContext:

```
public interface ServletContext

public String getMimeType(String filename);
public URL getResource(String path) throws MalformedURLException;
public InputStream getResourceAsStream(String path);
public RequestDispatcher getRequestDispatcher(String path);
public RequestDispatcher getNamedDispatcher(String name);
public String getRealPath(String path);
public ServletContext getcontext(String uripath);
public String getServerInfo();
public String getInitParameter(String name);
public Enumeration getInitParameterNames;
public Object getAttribute(String name);
public Enumeration getAttributeNames();
public void setAttribute (String name, Object attribute);
public void removeAttribute(String name);
public int getMajorVersion();
public int getMinorVersion();
public void log(String message);
public void log(String message, Throwable cause);

// Методы, использовать которые не рекомендуется
public Servlet getServlet(String Name) throws ServletException;
public Enumeration getServlets();
public Enumeration getServletNames();
public void log(Exception exception, String messages);
```

ServletRequest В объекте ServletRequest содержатся такие сведения о запросе клиента, как имена и значения параметров. Описание интерфейса ServletRequest:

```
public interface ServletRequest

public Object getAttribute(String name);
public Object setAttribute(String name,Object attribute);
public Enumeration getAttributeNames();
public void removeAttribute(String name);
public Locale getLocale();
public Enumeration getLocales();
public String getCharacterEncoding();
public int getContentLength();
public String getContentType();
```

```
public ServletInputStream getInputStream() throws IOException;
public String getParameter(String name);
public String getParameterNames();
public String getParameterValues();
public String getProtocol();
public String getScheme();
public String getServerName();
public int getServerport();
public BufferedReader getReader() throws IOException;
public String getRemoteAddr();
public String getRemoteHost();
public boolean isSecure();
public RequestDispatcher getRequestDispatcher(String path);

// Методы, использовать которые не рекомендуется
public String getRealPath();
```

ServletResponse Объект ServletResponse представляет ответ на запрос клиента. Описание интерфейса ServletResponse:

- `public interface ServletResponse`

```
public String getCharacterEncoding();
public ServletOutputStream getServletOutputStream()
        throws IOException;
public PrintWriter getWriter throws IOException;
public void setContentLength (int length);
public void setContentType(String type);
public void setBufferSize(int size);
public int getBuffersize();
public void reset();
public boolean isCommitted();
public void flushBuffer() throws IOException;
public void setLocale(Locale locale);
public Locale getLocale();
```

SingleThreadModel Интерфейс SingleThreadModel реализуется сервлетом для указания того, что только один поток может исполнять его метод service() в любой момент времени (см. главу 2). У этого интерфейса нет методов; он служит лишь индикатором однопоточной схемы исполнения. Описание интерфейса SingleThreadModel:

- `public interface SingleThreadModel`

```
// Методов нет
```

GenericServlet Абстрактный класс GenericServlet представляет сервлет, не зависящий от протокола. Он реализует интерфейс Servlet. Описание класса GenericServlet:

- `public abstract class GenericServlet implements Servlet`

```
public GenericServlet();
public String getInitParameter();
public Enumeration getInitParameterNames();
public ServletConfig getServletConfig();
public ServletContext getServletContext();
public String getServletInfo();
public void init();
public void init(ServletConfig config) throws ServletException;
public void log(String message);
public void log(String message, Throwable cause);
public abstract void service (ServletRequest req,
                              ServletResponse res)
      throws ServletException, IOException;
public void destroy();
```

ServletInputStream Объект ServletInputStream используется для чтения двоичных данных в запросе клиента. Описание абстрактного класса ServletInputStream:

```
public abstract class ServletInputStream extends InputStream

public ServletInputStream();.
public int readLine(byte[] buffer, int offset, int length)
    throws IOException;
```

ServletOutputStream Объект ServletOutputStream используется для посылки двоичных данных клиенту. Класс ServletOutputStream расширяет класс java.io.OutputStream и описывает переопределяемые методы для печати данных скалярных типов и типа String. Описание абстрактного класса ServletOutputStream:

```
public abstract class ServletOutputStream extends OutputStream

public ServletOutputStream();
public void print(String s) throws IOException;
public void print(boolean b) throws IOException;
public void print(char c) throws IOException;
public void print(int i ) throws IOException;
public void print(long l) throws IOException;
public void print(float f) throws IOException;
public void print(double d) throws IOException;
public void println() throws IOException;
public void println(String s) throws IOException;
public void println(boolean b) throws IOException;
public void println(char c) throws IOException;
public void println(int i) throws IOException;
public void println(long l) throws IOException;
public void println(float f) throws IOException;
public void println(double d) throws IOException;
```

ServletException Объект ServletException представляет исключительную ситуацию общего характера, которую устанавливает сервлет или JSP. Описание класса ServletException:

```
public class ServletException extends Exception;

public ServletException();
public ServletException(String message);
public ServletException(String message, Throwable cause);
public ServletException(Throwable cause);
public Throwable getRootCause();
```

UnavailableException Класс UnavailableException расширяет класс ServletException. Он используется для указания того, что сервлет временно или постоянно недоступен. Описание класса UnavailableException:

```
public class UnavailableException extends ServletException

public UnavailableException(String message);
public UnavailableException(String message, int sec);
public int getUnavailableException();
public boolean isPermanent();

// Методы, использовать которые не рекомендуется
public UnavailableException(Servlet servlet, String message);
public UnavailableException(int sec, Servlet servlet, String msg);
public Servlet getServlet();
```

Пакет javax.servlet.http

Пакет javax.servlet.http используется в программировании сервлетов с применением протокола HTTP. Он состоит из следующих интерфейсов и классов:

```
interface HttpServletRequest
interface HttpServletResponse
interface HttpSession
interface HttpSessionBindingListener
interface HttpSessionContext

class Cookie
class HttpServlet
class HttpSessionBindingEvent
class HttpUtils
```

Кратко опишем каждый из этих интерфейсов и классов.

HttpServletRequest Объект HttpServletRequest представляет запрос сервлета, посылаемый посредством HTTP. Описание интерфейса HttpServletRequest:

```
public interface HttpServletRequest extends ServletRequest;

public String getAuthType();
public Cookie[] getCookies();
public long getDateHeader(String name);
public String getHeader(String name);
public Enumeration getHeaders(String name);
public Enumeration getHeaderNames();
public int getIntHeader(String name);
public String getMethod();
public String getContextPath();
public String getPathInfo();
public String getPathTranslated();
public String getQueryString();
public String getRemoteUser();
public boolean isUserInRole(String role);
public java.security.Principal getUserPrincipal();
public String getRequestedSessionId();
public boolean isRequestedSessionIdValid();
public boolean isRequestedSessionIdFromCookie();
public boolean isRequestedSessionIdFromURL();
public String getRequestURI();
public String getServletPath();
public HttpSession getSession();
public HttpSession getSession(boolean create);

// Методы, использовать которые не рекомендуется
public boolean isRequestSessionIdFromUrl();
```

HttpServletResponse Интерфейс HttpServletResponse предоставляет методы для таких специфичных для HTTP средств, как заголовки и домашние данные. Описание этого интерфейса:

```
public interface HttpServletResponse extends ServletResponse
<<< STATUS CODES 416 AND 417 REPORTED MISSING>>>

public static final int SC_CONTINUE;
public static final int SC_SWITCHING_PROTOCOLS;
public static final int SC_OK;
public static final int SC_CREATED;
public static final int SC_ACCEPTED;
public static final int SC_NON_AUTHORITATIVE_INFORMATION;
public static final int SC_NO_CONTENT;
public static final int SC_RESET_CONTENT;
public static final int SC_PARTIAL_CONTENT;
```

```
public static final int SC_MULTIPLE_CHOICES;
public static final int SC_MOVED_PERMANENTLY;
public static final int SC_MOVED_TEMPORARILY;
public static final int SC_SEE_OTHER;
public static final int SC_NOT_MODIFIED;
public static final int SC_USE_PROXY;
public static final int SC_BAD_REQUEST;
public static final int SC_UNAUTHORIZED;
public static final int SC_PAYMENT_REQUIRED;
public static final int SC_FORBIDDEN;
public static final int SC_NOT_FOUND;
public static final int SC_METHOD_NOT_ALLOWED;
public static final int SC_NOT_ACCEPTABLE;
public static final int SC_PROXY_AUTHENTICATION_REQUIRED;
public static final int SC_REQUEST_TIMEOUT;
public static final int SC_CONFLICT;
public static final int SC_GONE;
public static final int SC_LENGTH_REQUIRED;
public static final int SC_PRECONDITION_FAILED;
public static final int SC_REQUEST_ENTITY_TOO_LARGE;
public static final int SC_REQUEST_URI_TOO_LONG;
public static final int SC_UNSUPPORTED_MEDIA_TYPE;
public static final int SC_REQUESTED_RANGE_NOT_SATISFIABLE;
public static final int SC_EXPECTATION_FAILED;
public static final int SC_INTERNAL_SERVER_ERROR;
public static final int SC_NOT_IMPLEMENTED;
public static final int SC_BAD_GATEWAY;
public static final int SC_SERVICE_UNAVAILABLE;
public static final int SC_GATEWAY_TIMEOUT;
public static final int SC_VERSION_NOT_SUPPORTED;
public void addCookie(Cookie cookie);
public boolean containsHeader(String name);
public String encodeURL(String url);
public String encodeRedirectURL(String url);
public void sendError(int status) throws IOException;
public void sendError(int status, String message)
    throws IOException;
public void sendRedirect(String location) throws IOException;
public void setDateHeader(String headername, long date);
public void setHeader(String headername, String value);
public void addHeader(String headername, String value);
public void addDateHeader(String headername, long date);
public void addIntHeader(String headername, int value);
public void setIntHeader(String headername, int value);
public void setStatus(int statuscode);

// Методы, использовать которые не рекомендуется
public void setStatus(int statuscode, String message);
public String encodeUrl(String url);
public String encodeRedirectUrl(String url);
```

HttpSession В интерфейсе HttpSession описаны методы для сеансового взаимодействия клиента и пользователя, например, в интерактивном приложении ввода заказов. Описание интерфейса HttpSession:

```
public interface HttpSession

public long getCreationTime();
public String getId();
public long getLastAccessedTime();
public boolean isNew();
public int getMaxInactiveInterval();
public void setMaxInactiveInterval(int interval);
public Object getAttribute(String name);
```

```
public Enumeration getAttributeNames();
public void setAttribute(String name, Object attribute);
public void removeAttribute(String name);
public void invalidate();

// Методы, использовать которые не рекомендуется
public Object getValue(String name);
public String[] getValueNames();
public void putValue(String name, Object value);
public void removeValue(String name);
public HttpSessionContext getSessionContext();
```

HttpSessionBindingListener Это интерфейс приемника событий (см. раздел "События и интерфейсы приемников событий" главы 2). Методы этого интерфейса вызываются при привязке или при отмене привязки объекта к HTTP-сеансу. Описание интерфейса HttpSessionBindingListener:

- ```
  public interface HttpSessionBindingListener extends EventListener

  public void valueBound(HttpSessionBindingEvent event);
  public void valueUnbound(HttpSessionBindingEvent event);
  ```

HttpSessionContext Интерфейс HttpSessionContext использовать не рекомендуется, что связано с вопросами безопасности. Описание этого интерфейса:

- ```
  // Использовать не рекомендуется
  public abstract interface HttpSessionContext

  // Методы, использовать которые не рекомендуется
  public void Enumeration getIds();
  public HttpSession getSession(String id);
  ```

Cookie Объект Cookie предоставляет информацию о состоянии клиенту. Она может быть послана сервлетом браузеру и использована для контроля за HTTP-сеансами. Описание класса Cookie:

- ```
  public class Cookie implements Cloneable

  public Cookie(String name, String value);
  public void setComment(String comment);
  public String getComment();
  public void setDomain(String domain);
  public String getDomain();
  public void setMaxAge(int expiry);
  public int getMaxAge();
  public void setPath(String uriPath);
  public String getPath();
  public void setSecure();
  public boolean getSecure();
  public String getName();
  public void setValue(String value);
  public String getValue();
  public int getVersion();
  public void setVersion(int version);
  public Object clone();
  ```

HttpServlet Абстрактный класс HttpServlet предоставляет сервлетам функциональные средства, специфичные для HTTP. Например, в нем описаны методы doGet() и doPost() для обслуживания HTTP-запросов GET и POST. Описание абстрактного класса HttpServlet:

- ```
  public abstract class HttpServlet extends GenericServlet
        implements Serializable
  ```

```
public HttpServlet();
protected void doGet(HttpServletRequest req,
                     HttpServletResponse res)
    throws ServletException, IOException;
protected void doPost(HttpServletRequest req,
                      HttpServletResponse res)
    throws ServletException, IOException;
protected void doPut(HttpServletRequest req,
                     HttpServletResponse res)
    throws ServletException, IOException;
protected void doDelete(HttpServletRequest req,
                        HttpServletResponse res)
    throws ServletException, IOException;
protected void doOptions(HttpServletRequest req,
                         HttpServletResponse res)
    throws ServletException, IOException;
protected void doTrace(HttpServletRequest req,
                       HttpServletResponse res)
    throws ServletException, IOException;
protected void service(HttpServletRequest req,
                       HttpServletResponse res)
    throws ServletException, IOException;
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;
protected long getLastModifed(HttpServletRequest req);
```

HttpSessionBindingEvent Объект HttpSessionBindingEvent представляет события привязки и отмены привязки объекта к HTTP-сеансу. Он передается в качестве аргумента методам интерфейса HttpSessionBindingListener. Описание класса HttpSessionBindingEvent:

- `public class HttpSessionBindingEvent extends EventObject`

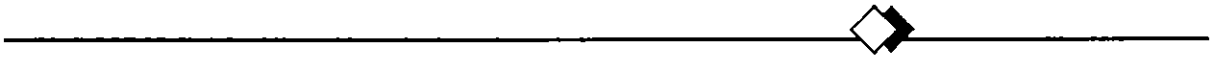
```
public HttpSessionBindingEvent(HttpSession session, S' ing name);
public String getName();
public HttpSession getSession();
```

HttpUtils В классе HttpUtils описываются статические служебные методы для программирования сервлетов. Описание класса HttpUtils:

- `public class HttpUtils`

```
public HttpUtils();
public static Hashtable parseQueryString(String queryString);
public static Hashtable parsePostData(int length,
                                      ServletInputStream in);
public static StringBuffer getRequestURL(HttpServletRequest req);
```

ПРИЛОЖЕНИЕ В



Краткое описание API Enterprise JavaBeans

В этом приложении показано, как создавать, компилировать и внедрять зерна Enterprise JavaBean, а также дан обзор Enterprise JavaBeans API (версий 1.0 и 1.1).

Краткий справочник по Enterprise JavaBeans

Для реализации приложения клиент/сервер, использующего зерно уровня предприятия, необходимо:

- Написать зерно уровня предприятия
- Скомпилировать компоненты зерна уровня предприятия
- Упаковать зерно уровня предприятия
- Написать файл дескриптора внедрения
- Внедрить зерно уровня предприятия
- Написать клиентское приложение
- Скомпилировать клиентское приложение
- Выполнить клиентское приложение

Для выполнения всех этих задач в качестве примера используем сеансовое зерно `My8iEJB` (см. раздел "Создание первого Enterprise JavaBean" главы 3).

Создание зерна уровня предприятия (сеансового зерна)

Каждому зерну уровня предприятия необходимы следующие компоненты:

- Удаленный интерфейс
- Домашний интерфейс
- Класс зерна уровня предприятия (Bean-класс)

Написание удаленного интерфейса

В удаленном интерфейсе EJB описываются бизнес-методы, которые могут вызываться клиентом. В этом интерфейсе разработчик EJB предоставляет список сигнатур методов, реализации которых находятся в Bean-классе. Все удаленные интерфейсы EJB должны расширять интерфейс `javax.ejb.EJBObject`, а в конструкцию `throws` сигнатур методов должна быть включена исключительная ситуация `java rmi.RemoteException`. Кроме того, в описаниях методов могут определяться любые исключительные ситуации, специфичные для приложения. Удаленный интерфейс `My8iEJB`:

```
// Имя программы: My8iEJB.java

package server;

// Обязательные пакеты Java и ejb для удаленных интерфейсов EJB
import java.rmi.*;
import javax.ejb.*;

// Опишем общий интерфейс, который расширяет интерфейс
// javax.ejb.EJBObject
public interface My8iEJB extends EJBObject {

  // Опишем общий метод, который в точности соответствует методу,
  // определенному в Bean-классе.
  // Сигнатура бизнес-метода:
  public String getFirst8iEJB() throws RemoteException;
}
```

Написание домашнего интерфейса

Домашние интерфейсы EJB должны расширять интерфейс javax.ejb.EJBHome. В домашнем интерфейсе описываются методы, позволяющие клиенту создавать, обнаруживать и удалять зерно уровня предприятия. Отметим, что в домашнем интерфейсе сеансового зерна можно описать только один метод create(), а в домашнем интерфейсе сущностного зерна – один или несколько методов create() и один метод find(). Метод create() домашнего интерфейса возвращает объект типа удаленного интерфейса и должен содержать в своей конструкции throws исключительные ситуации java.rmi.RemoteException и javax.ejb.CreateException. Домашний интерфейс My8iEJBHome:

```
// Имя программы:    My8iEJBHome.java

package server;

import java.rmi.*;
import javax.ejb.*;

// Этот интерфейс должен быть общим и расширять интерфейс java.ejb.EJBHome
public interface My8iEJBHome extends EJBHome {

// Описание домашнего интерфейса разработчика должно содержать один
// или несколько методов create(). Каждому указанному здесь методу
// create() должен соответствовать метод ejbCreate(), указанный в Bean-классе.

  My8iEJB create() throws CreateException, RemoteException;
}
```

Написание Bean-класса

Все сеансовые Bean-классы должны реализовывать интерфейс javax.ejb.SessionBean, а все сущностные – интерфейс javax.ejb.EntityBean. Здесь используется класс My8iEJBClass, созданный в главе 3; он внедряется как сеансовое зерно типа STATELESS. Это зерно реализует один бизнес-метод – getFirst8iEJB(), описанный в удаленном интерфейсе My8iEJB. Напомним, что бизнес-методы, реализуемые в Bean- классе, должны в точности соответствовать сигнатурам методов удаленного интерфейса. Когда клиент вызывает метод getFirst8iEJB(), тот возвращает вызывающему Java-строку символов (String). Помимо бизнес-методов, все классы зерен уровня предприятия должны содержать следующие методы: ejbCreate(), ejbActivate(), ejbPassivate(), ejbRemove() и setSessionContext(SessionContext ctx). Bean-класс My8iEJBClass:

```
// Имя программы:    My8iEJBClass.java

package server;

import java.rmi.RemoteException;
import javax.ejb.*;

public class My8iEJBClass implements SessionBean {
  public String getFirst8iEJB() throws RemoteException {
    String aString =
        "First EJB application stored in the "
      + " Oracle8i database on data-i.com server "
      + " from the Oracle8i Java Component Programming "
      + " with EJB, CORBA, and JSP book.";
    return aString;

  // Обратите внимание на метод ejbCreate(), который соответствует
  // методу create() в описании домашнего интерфейса.
  public void ejbCreate()
    throws RemoteException, CreateException {
  } // Конец ejbCreate()

  public void ejbActivate()
```

```
    throws RemoteException {
  } // Конец ejbActivate()

  public void ejbPassivate()
    throws RemoteException {
  } // Конец ejbPassivate()

  public void ejbRemove()
    throws RemoteException {
  } // Конец ejbRemove()

  public void setSessionContext(SessionContext ctx)
    throws RemoteException {
  } // Конец setSessionContext()

} // Конец My8iEJBClass
```

Компиляция исходного программного текста зерна

Для установки CLASSPATH и для компиляции исходного программного текста в среде UNIX следует воспользоваться MAKEFILE, а в Windows – SetupJDK1EnvMy8iBean.bat или SetupJDK2EnvMy8iBean.bat. В листинге В.1 показана установка среды для компиляции исходного программного текста Java:

Листинг В.1. Компиляция исходного программного текста зерна в DOS

```
// В приглашении DOS введем:
set ORACLE_HOME=G:\Oracle\Ora81
set ORACLE_SERVICE=sess_iiop://data-i.com:2481:ORCL
REM Using JDK1.1.x
set JDK_CLASSPATH=J:\JDeveloper30\java\lib\classes.zip
REM Using JDK1.2.x: set JDK_CLASSPATH=C:\jdk1.2.1\lib\dt.jar
set CLASSPATH=.;I:\donnai;%ORACLE_HOME%\lib\aurora_client.jar;
%ORACLE_HOME%\jdbc\lib\classes12.zip;
%ORACLE_HOME%\sqlj\lib\translator.zip;
%ORACLE_HOME%\lib\vbjorb.jar;
%ORACLE_HOME%\lib\vbjapp.jar;%JDK_CLASSPATH%
```

Компиляция исходного программного текста Для компиляции исходного программного текста зерна воспользуемся следующим программным фрагментом:

```
javac -g My8iEJB.java
javac -g My8iEJBHome.java
javac -g My8iEJBClass.java
```

Упаковка зерна уровня предприятия

Для упаковки зерна уровня предприятия используется команда jar:

```
jar cvf0 server/My8iEJB.jar server/*.class
```

Написание файла дескриптора внедрения

Файл дескриптора внедрения для зерен, согласующихся с EJB 1.0, и XML-текст для зерен, согласующихся с EJB 1.1, можно создать в любом текстовом редакторе. Однако в новой версии Oracle8*i* – 8.1.7 – для файла дескриптора внедрения будут поддерживаться и текстовый, и XML- формат. Приведем пример текстового формата:

```
// Имя программы:    My8iEJB.ejb
SessionBean server.My8iEJBClass {
    BeanHomeName = "test/My8iEJB";
    HomeInterfaceClassName = server.My8iEJBHome;
    RemoteInterfaceClassName = server.My8iEJB;
    AllowedIdentities = {PUBLIC};
    SessionTimeout = 0;
```

```
    StateManagementType = STATELESS_SESSION;
    RunAsMode = CLIENT_IDENTITY;
}
```

Внедрение зерна уровня предприятия

Для внедрения сеансового зерна My8iEJB воспользуемся Oracle-инструментом deployejb. При его применении генерируются stub-классы, которые позволяют клиенту взаимодействовать с экземпляром зерна уровня предприятия, функционирующим в EJB-контейнере. В данном примере этот Oracle-инструмент генерирует stub-классы и сохраняет их в файле My8iEJBClient.jar:

```
deployejb -republish -keep -temp temp -u scott -p tiger
-s %ORACLE_SERVICE% -verbose -descriptor My8iEJB.ejb
-generated My8iEJBClient.jar My8iEJB.jar
```

Написание клиентского приложения

Программа MyFirstEJBClient — это автономное приложение Java. На примере MyFirstEJBClient демонстрируются основные задачи, решаемые клиентом зерна уровня предприятия:

- Поиск домашнего интерфейса
- Создание экземпляра зерна уровня предприятия
- Вызов бизнес-метода

Поиск домашнего интерфейса Для обнаружения домашнего интерфейса зерна нужно выполнить следующие операции:

1. Установить переменные среды и создать JNDI-контекст именования:

```
// Создадим объект Hashtable для хранения переменных среды
   Hashtable env = new Hashtable();

   // Обязательная установка свойств JNDI-среды безопасности,
   // подлежащих передаче объекту Context
   env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
   env.put(Context.SECURITY_PRINCIPAL, 'SCOTT');
   env.put(Context.SECURITY_CREDENTIALS, 'tiger');
   env.put(Context.SECURITY_AUTHENTICATION,
        ServiceCtx.NON_SSL_LOGIN);

// Создадим экземпляр класса Context
Context ic = new InitialContext (env);
```

2. С помощью метода lookup() интерфейса Context получить ссылку на домашний интерфейс зерна. Метод lookup() принимает один входной параметр — объединение адреса URL и имени опубликованного объекта:

```
My8iEJBHome my8iEJBHome =
   (My8iEJBHome)ic.lookup ("sess_sh://data-i.com:2481:orcl" `
                           +"/test/My8iEJB");
```

Создание экземпляра зерна уровня предприятия С помощью метода create() домашнего интерфейса создадим экземпляр зерна My8iEJBBean. Метод create() возвращает объект типа My8iEJB. В удаленном интерфейсе My8iEJB описан один метод в классе My8iEJBClass, который может быть вызван клиентом. Когда клиент вызывает метод create(), EJB-контейнер создает экземпляр My8iEJBClass, а затем вызывает метод My8iEJBClass.ejbCreate():

```
my8iEJBBean = my8iEJBHome.create();
```

Вызов бизнес-метода Вызовем метод объекта My8iEJB. Когда клиент вызывает метод зерна, EJB-контейнер вызывает соответствующий метод экземпляра My8iEJBClass, функционирующего на сервере. В следующем программном фрагменте показано, как клиент вызывает метод getFirst8iEJB():

```
String outputFromBean = my8iEJBBean.getFirst8iEJB();
```

Приведем полный исходный текст программы MyFirstEJBClient:

```
/*  Имя программы: MyFirstEJBClient.java
*/

package client;

// Импортируем удаленный интерфейс зерна.
import server.My8iEJB;

// Импортируем домашний интерфейс зерна.
import server.My8iEJBHome;

// Импортируем класс ServiceCtx из пакета Oracle.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем классы Java, обязательные для использования JNDI.
import javax.naming.Context;
import javax.naming.InitialContext;

// Импортируем классы, специфичные для приложения.
import java.util.Hashtable;
import java.sql.*;

public class MyFirtsEJBClient {
   public static void main(String[] args)
     throws Exception {
   // Проверим входной параметр.
   if (args.length != 4) {
         System.out.println("usage: Client "
           +"serviceURL objectName user password");
         System.exit(1);
     } // Конец if

     // Создадим String-объекты для хранения входных параметров,
     // содержащих:
     // args [0] = sess_sh://data-i.com:2481:orcl;
     // args [1] = /test/My8iEJB;
     // args [2] = scott;
     // args [3] = tiger;
     String serviceURL = args [0];
     String objectName = args [1];
     String user = args [2];
     String password = args [3];

     // Создадим объект Hashtable для хранения переменных среды.
     Hashtable env = new Hashtable();

     // Обязательная установка свойств JNDI-среды безопасности,
     // подлежащих передаче объекту Context.
     env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
     env.put(Context.SECURITY_PRINCIPAL, user);
     env.put(Context.SECURITY_CREDENTIALS, password);
     env.put(Context.SECURITY_AUTHENTICATION,
         ServiceCtx.NON_SSL_LOGIN);

     // Создадим экземпляр класса Context.
     Context ic = new InitialContext (env);

     // Создадим экземпляр домашнего интерфейса объекта EJB.
     My8iEJBHome myEJBHome = null;

     // Создадим удаленный объект.
     My8iEJB my8iEJBBean = null;

     System.out.println("Variable Created Properly");
```

```
        System.out.println("serviceURL : " + serviceURL);
        System.out.println("objectName : " + objectName);

        try {
            // Воспользуемся домашним интерфейсом для нахождения
            // объекта EJB.
            my8iEJBHome =
              (My8iEJBHome)ic.lookup (serviceURL + objectName);
            System.out.println("Home Lookup Properly");

            // Создадим экземпляр зерна.
            my8iEJBBean = myEJBHome.create();

            System.out.println("Bean Created Properly");

        } // Конец try
        catch ( Exception ex ) {
            System.out.println("Cannot locate"
              +" or create My8iEJB object");
            System.err.println("error : " + ex);
            System.exit(1);
        } // Конец catch

        // Создадим Java-объект String для хранения результата
        // вызова метода зерна.
        String outputFromBean = null;

        // Вызовем удаленный метод EJB: getFirst8iEJB().
        // Вызовем метод зерна с помощью удаленного экземпляра
        // my8iEJBBean и сохраним результат в Java-переменной
        // outputFromBean.
        try {
            outputFromBean = My8iEJBBean.getFirst8iEJB();
            System.out.println("Method Invocation went well: "
                       + outputFromBean);
        } // Конец try
        catch (Exception ex) {
        System.err.println(" Unable to invoke "
        + " the getFirst8iEJB() method : " + ex);
                  System.exit(1);
        } // Конец catch
      }
} // Конец класса MyFirstEJBClient
```

Компиляция клиентского приложения Для компиляции клиента воспользуемся листингом B.1 и добавим файлы `My8iEJB.jar` в `My8iEJBClient.jar` в переменную `CLASSPATH` (см. выше разделы "Упаковка зерна уровня предприятия" и "Внедрение зерна уровня предприятия"). Например:

```
...
set CLASSPATH=.;I:\donnai;%ORACLE_HOME%\lib\aurora_client.jar;
%ORACLE_HOME%\jdbc\lib\classes12.zip;
%ORACLE_HOME%\sqlj\lib\translator.zip;
%ORACLE_HOME%\lib\vbjorb.jar;
%ORACLE_HOME%\lib\vbjapp.jar;%JDK_CLASSPATH%;
My8iEJB.jar;My8iEJBClient.jar
// Скомпилируем клиента
javac -g MyFirtsEJBClient.java
```

Выполнение клиентского приложения Для выполнения клиента воспользуемся командой из раздела "Компиляция клиентского приложения" этого приложения:

```
java MyFirtsEJBClient %ORACLE_SERVICE% /test/My8iEJB scott tiger
```

Enterprise JavaBeans API

В этом разделе дается краткий обзор EJB API (1.0 и 1.1). За дополнительными сведениями об Enterprise JavaBeans API обращайтесь к www.java.sun.com/products/ejb/javadoc-1.1/.

Иерархическая структура пакета javax.ejb

Enterprise JavaBeans API состоит из нескольких интерфейсов и классов. Сначала рассмотрим иерархическую структуру классов и интерфейсов, а затем опишем классы и интерфейсы Java, составляющие EJB API.

Иерархическая структура классов

Иерархическая структура классов имеет вид:

```
class java.lang.Object
  class java.lang.Throwable (implements java.io.Serializable)
    class java.lang.Exception
        class javax.ejb.CreateException
          class javax.ejb.DuplicateKeyException
        class javax.ejb.FinderException
          class javax.ejb.ObjectNotFoundException
        class javax.ejb.RemoveException
        class java.lang.RuntimeException
          class javax.ejb.EJBException
            class javax.ejb.NoSuchEntityException
```

Иерархическая структура интерфейсов

Иерархическая структура интерфейсов имеет вид:

```
interface javax.ejb.EJBContext
  interface javax.ejb.EntityContext
  interface javax.ejb.SessionContext
interface javax.ejb.EJBMetaData
interface java.rmi.Remote
  interface javax.ejb.EJBHome
  interface javax.ejb.EJBObject
interface java.io.Serializable
  interface javax.ejb.EnterpriseBean
    interface javax.ejb.EntityBean
    interface javax.ejb.SessionBean
  interface javax.ejb.Handle
  interface javax.ejb.HomeHandle
interface javax.ejb.SessionSynchronization
```

Пакет javax.ejb

Этот пакет состоит по большей части из интерфейсов, многие из которых реализуются поставщиком EJB, а ряд исключительных ситуаций устанавливается зернами уровня предприятия.

CreateException

В конструкции throws всех методов create(), описанных в домашнем интерфейсе, должна присутствовать исключительная ситуация CreateException:

```
public class javax.ejb.CreateException
    extends java.lang.Exception{
  public CreateException();
  public CreateException(String message);
}
```

DuplicateKeyException

В конструкции throws всех методов create() домашнего интерфейса сущностных зерен должна присутствовать исключительная ситуация DuplicateKeyException:

```
public class javax.ejb.DuplicateKey
    extends javax.ejb.CreateException{
  public DuplicateKeyException();
  public DublicateKeyException(String message);
}
```

EJBContext

Это родительский интерфейс как для EntityContext, так и для SessionContext. EJBContext предоставляет информацию об идентификационных характеристиках безопасности, состоянии транзакций и о доступе к переменным среды и к домашнему объекту EJB зерна:

```
public interface javax.ejb.EJBContext {
  // getCallerPrincipal() - новый метод в EJB 1.1
  public abstract Principal getCallerPrincipal();
  public abstract EJBHome getEJBHome();
  public abstract boolean getRollbackOnly();
  public abstract UserTransaction getUserTransaction();
  // Использовать getEnvironment() в EJB 1.1 не рекомендуется
  public abstract Properties getEnvironment();
  // Использовать getCallerIdentity() в EJB 1.1 не рекомендуется
  public abstract Identity getCallerIdentity();
  // Использовать isCallerInRole в EJB 1.1 не рекомендуется
  public abstract boolean isCallerInRole(Identity role);
  // Новый метод в EJB 1.1
  public abstract boolean isCallerInRole(String roleNames);
  public abstract void setRollbackOnly();
}
```

EJBException (версия 1.1)

Это класс EJBException, описанный в EJB Specification версии 1.1:

```
public class javax.ejb.EJBException
    extends java.lang.RuntimeException{
  public EJBException();
  public EJBException(String message);
  public EJBException(Exception exception);
  public Exception getCausedException();
}
```

EJBHome

Домашние интерфейсы всех зерен должны расширять этот интерфейс:

```
public interface javax.ejb.EJBHome
    extends java.rmi.Remote{
  // Метод getHomeHandle() - новый метод в EJB 1.1
  public abstract HomeHandle getHomeHandle();
  public abstract EJBMetaData getEJBMetaData();
  public abstract void remove(Handle handle);
  public abstract void remove(Object primaryKey);
}
```

EJBMetaData

Этот интерфейс реализуется поставщиком контейнера. EJB-контейнер предоставляет класс Serializable с информацией о зернах уровня предприятия:

```
public interface javax.ejb.EJBHome{
  public abstract EJBHome();
```

```
    public abstract Class getHomeInterfaceClass();
    public abstract Class getPrimaryKeyclass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
    // isStatelessSession() - новый метод в EJB 1.1
    public abstract boolean isStatelessSession();
}
```

EJBObject

Все удаленные интерфейсы должны расширять интерфейс EJBObject:

```
public interface javax.ejb.EJBObject
    extends java.rmi.Remote {
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
    public abstract boolean isIdentical(EJBObject obj);
    public abstract void remove();
}
```

EnterpriseBean

Этот интерфейс расширяют все интерфейсы EntityBean и SessionBean:

```
public interface javax.ejb.EnterpriseBean
    extends java.io.Serializable{
}
```

EntityBean

Все сущностные зерна должны реализовывать этот интерфейс:

```
public interface javax.ejb.Entitybean
    extends javax.ejb.EnterpriseBean{
    public abstract void ejbActivate();
    public abstract void ejbLoad();
    public abstract void ejbPassivate();
    public abstract void ejbRemove();
    public abstract void ejbStore();
    public abstract void setEntityContext(EntityContext ctx);
    public abstract void unsetEntityContext();
}
```

EntityContext

EntityContext предоставляет экземпляру зерна интерфейс связи с контейнером:

```
public interface javax.ejb.EntityContext
    extends javax.ejb.EJBcontext{
    public abstract EJBObject getEJBObject();
    public abstract Object getPrimaryKey();
}
```

FinderException

Эта стандартная исключительная ситуация устанавливается методами обнаружения, описанными в домашнем интерфейсе сущностного зерна:

```
public class javax.ejb.FinderException
    extends java.lang.Exception{
    public FinderException();
    public FinderException(String message);
}
```

Handle

Этот интерфейс предоставляет клиенту сериализуемый объект, используемый для сохранения удаленного интерфейса зерна и при необходимости для повторного получения ссылки:

```
public interface javax.ejb.Handle
    extends java.io.Serializable{
  public abstract EJBHome getEJBHome();
}
```

HomeHandle (версия 1.1)

Это класс HomeHandle, описанный в EJB Specification (версии 1.1):

```
public class javax.ejb.NoSuchEntityException
    extends javax.ejb.EJBException{
  public NoSuchEntityException();
  public NoSuchEntityException(String message);
  public NoSuchEntityException(Exception exception);
}
```

ObjectNotFoundException

Описание класса ObjectNotFoundException:

```
public class javax.ejb.ObjectNotFoundException
    extends javax.ejb.FinderException{
  public ObjectNotFoundException();
  public ObjectNotFoundException(String message);
}
```

RemoveException

Это класс RemoveException:

```
public class javax.ejb.RemoveException
    extends java.lang.Exception{
  public RemoveException();
  public RemoveException(String message);
}
```

SessionBean

Все сеансовые зерна должны реализовывать этот интерфейс:

```
public interface javax.ejb.SessionBean
    extends javax.ejb.EnterpriseBean{
  public abstract void ejbActivate();
  public abstract void ejbPassivate();
  public abstract void ejbRemove();
  public abstract void setSessionContext(SessionContext ctx);
}
```

SessionContext

Методы этого интерфейса служат для получения ссылки на EJB-объект SessionBean. SessionContext предоставляет экземпляру зерна интерфейс связи с контейнером:

```
public interface javax. ejb.SessionContext
    extends javax.ejb.EJBContext{
  public abstract EJBObject getEJBObject();
}
```

SessionSynchronization

Описание интерфейса SessionSynchronization:

```
public interface javax.ejb.SessionSynchronization{
   public abstract void afterBegin();
   public abstract void afterCompletion(boolean commited);
   public abstract void beforeCompletion();
}
```

Пакет javax.ejb.deployment (только EJB 1.0)

В пакете javax.ebj.deployment содержится ряд классов, используемых для внедрения зерен уровня предприятия в контейнере. В EJB 1.1 нет этого пакета и его классов.

AccessControlEntry

Описание класса AccessControlEntry:

```
public class javax.ejb.deployment.AccessControlEntry
     extends java.lang.Object
     implements java.io.Serializable{
   public AccessControlEntry();
   public AccessControlEntry(Method method);
   public AccessControlEntry(Method method,
                    Identity identities);
   public Identity[] getAllowedIdentities();
   public Identity getAllowedIdentities(int index);
   public Method getMethod();
   public void setAllowedIdentities(Identity values[]);
   public void setAllowedIdentities (int index, Identity values);
   public void setMethod(Method value);
}
```

ControlDescriptor

Объект этого класса используется для указания транзакционного атрибута и атрибута runAs, связанных с методами зерна:

```
public class javax.ejb.deployment.ControlDescriptor
     extends java.lang.Object implements java.io.Serializable{
   public final static int CLIENT_IDENTITY;
   public final static int SPECIFIED_IDENTITY;
   public final static int SYSTEM_IDENTITY;
   public final static int TRANSACTION_READ_COMMITTED;
   public final static int TRANSACTION_READ_UNCOMMITTED;
   public final static int TRANSACTION_REPEATABLE_READ;
   public final static int TRANSACTION_SERIALIZABLE;
   public final static int TX_BEAN_MANAGED;
   public final static int TX_MANDATORY;
   public final static int TX_NOT_SUPPORTED;
   public final static int TX_REQUIRED;
   public final static int TX_REQUIRES_NEW;
   public final static int TX_SUPPORTS;
   public ControlDescriptor();
   public ControlDescriptor(method method);
   public int getIsolationLevel();
   public Method getMethod();
   public Identity getRunAsIdentity();
   public int get RunAsMode();
   public int get TransactionAttribute();
   public void setIsolationLevel();
   public void setMethod(Method method);
   public void setRunAsIdentity(Identity value);
```

```
    public void setRunAsMode(int value);
    public void setTransactionAttribute(int value);
}
```

DeploymentDescriptor

При внедрении объект этого класса применяется для описания зерна для контейнера. Класс DeploymentDescriptor – базовый для классов EntityDescriptor и SessionDescriptor, которые используются типами EntityBean и SessionBean соответственно. В объекте класса DeploymentDescriptor содержатся все объекты AccessControlEntry и ControlDescriptor для зерна, описания удаленного интерфейса, домашнего интерфейса, имен Bean-классов и привязки имени (имени опубликованного объекта) зерна:

```
public class javax.ejb.deployment.DeploymentDescriptor
    extends java.lang.Object implements java.io.Serializable {
  protected int versionNumber;
  public DeploymentDescriptor();
  public AccessControlEntry[] getAccessControlEntries();
  public AccessControlEntry getAccessControlEntries(int index);
  publicName getBeanHomeName();
  public ControlDescriptor[] getControlDescriptors();
  public ControlDescriptor getControlDescriptors(int index);
  public String getEnterpriseBeanClassName();
  public Properties getEnvironmentProperites();
  public String getHomeInterfaceClassName();
  public boolean getReentrant();
  public String getRemoteInterfaceClassName();
  public boolean isReentrant();
  public void setAccessControlEntries(AccessControlEntry value[]);
  public void setAccessControlEntries(int i, AccessControlEntry v);
  public void setBeanHomeName(Name value);
  public void setControlDescriptors(ControlDescriptor values[]);
  public void setControlDescriptors(int index,
                                    ControlDescriptor value);
  public void setEnterpriseBeanClassName(String value);
  public void setEnvironmentProperties(Properties value);
  public void setHomeInterfaceClassName(String value);
  public void setReentrant(boolean value);
  public void setRemoteInterfaceClassName(String value);
}
```

EntityDescriptor

Этот класс расширяет класс DeploymentDescriptor и предоставляет методы, специфичные для сущностных зерен:

```
public class javax.ejb.deployment.EntityDescriptor
    extends javax.ejb.deployment.DeploymentDescriptor {
  public EntityDescriptor();
  public Field[] getContainerManagedFields(int index);
  public Field getContainerManagedFields(int, index);
  public String getPrimaryKeyClassName();
  public void setContainerManagedFields(Fields values[]);
  public void setContainerManagedFields(int index, Field value);
  public void setPrimaryKeyClassName(String value);
}
```

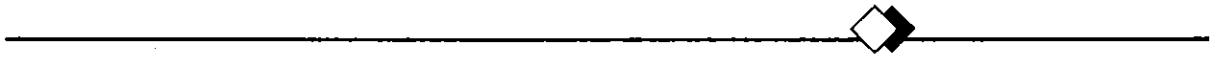
SessionDescriptor

Этот класс расширяет класс DeploymentDescriptor и предоставляет методы, специфичные для сеансовых зерен:

```
public class javax.ejb.deployment.SessionDescriptor
    extends javax.ejb.deployment.DeploymentDescriptor {
```

```
    public final static int STATEFUL_SESSION;
    public final static int STATELESS_SESSION;
    public SessionDescriptor();
    public int getSessionTimeout();
    public int getStateManagementType();
    public void setSessionTimeout(int value);
    public void setStateManagementType(int value);
}
```

ПРИЛОЖЕНИЕ C



Краткий справочник по Oracle8i CORBA для Java

Ниже дается краткий обзор этапов внедрения и использования серверного объекта на примере серверного объекта Account (см. главу 6).

На этих этапах выполняются следующие действия:

- Программирование IDL-модуля
- Генерация исходных файлов Java из IDL-модуля с помощью idl2java
- Реализация серверных объектов CORBA с помощью Java
- Компиляция файлов Java и загрузка сгенерированных классов Java в JServer
- Публикация имени для каждого серверного объекта
- Программирование и компиляция клиента Java, вызывающего методы серверного объекта CORBA
- Выполнение клиента

Программирование модуля IDL

Цель этого серверного объекта — получить из таблицы ACCOUNT_LIST номер отдела и номер проекта для указанного номера счета. Начнем с программирования IDL-модуля:

❒
```
/* Имя файла: account.idl */

module account {
   struct AccountInfo { long departmentno; long projectno; };
   exception NoAccountError { wstring mess; };
   interface Account {
     AccountInfo getAccountInfo( in long accountno )
         raises( NoAccountError );
   };
};
```

Метод getAccount возвращает структуру Account, содержащую значения departmentno и projectno для записи ACCOUNT_LIST с указанным значением accountno. Если такой записи в ACCOUNT_LIST не существует, устанавливается исключительная ситуация NoAccountError.

Генерация исходных файлов Java из файла IDL с помощью idl2java

Для компиляции account.idl и создания конструкции Java нужно ввести следующее:

❒
```
idl2java _no_tie account.idl
```

Параметр _no_tie указывает на то, что механизм привязки не применяется, поэтому генерироваться будет меньшее число исходных файлов Java (см. раздел "Механизм привязки CORBA" главы 6). С параметром _no_tie программа idl2java генерирует для модуля 12 исходных файлов Java, причем все они будут содержаться в каталоге account:

- Интерфейс Java, helper-, holder-, stub- и скелетный файлы для интерфейса Account
- Исходный файл Java с описанием класса, а также helper- и holder-файлы Java для *структуры* AccountInfo и для *исключительной ситуации* NoAccountError
- Файл _example_Account, содержащий неполный пример Java-реализации серверного объекта Account

Программирование Java-реализации серверного объекта

Ниже приводится SQLJ-реализация серверного объекта. В разделе "Реализация серверного объекта CORBA с помощью JDBC" главы 6 представлен JDBC-вариант данной программы.

```
/* Имя программы: AccountImpl.sqlj
**
** Назначение: Получить информацию для заданного номера счета.
**
*/
package accountServer;
import account.*;
import java.sql.*;
public class AccountImpl extends _AccountImplBase {
   public AccountInfo getAccountInfo( int accountno )
     throws NoAccountError {
     try {
         // Необходимо инициализировать переменные для размещения в них информации.
         int departmentno = 0;
         int projectno = 0;

         #sql {
           SELECT departmentno, projectno
              INTO :departmentno, :projectno
                FROM ACCOUNT_LIST
                  WHERE accountno = :accountno
         };
         return new AccountInfo( departmentno, projectno );
     }
     catch( SQLException e ) {
         throw new NoAccountError( e.getMessage() );
     }
   }
}
```

Компиляция файлов Java и загрузка сгенерированных файлов классов в JServer

Сначала скомпилируем исходные файлы Java:

```
javac account/Account.java\
  account/AccountHolder.java\
  account/AccountHelper.java
javac account/AccountInfo.java\
  account/AccountInfoHolder.java\
  account/AccountInfoHelper.java
javac account/NoAccountError.java\
  account/NoAccountErrorHolder.java\
  account/NoAccountErrorHelper.java
javac account/_EmployeeImplBase.java\
  account/_st_Account.java
sqlj -J-classpath\
  .:$(ORACLE_HOME)/lib/aurora_client.jar:\
  $(ORACLE_HOME)/jdbc/lib/classes111.zip:\
  $(ORACLE_HOME)/sqlj/lib/translator.zip$(ORACLE_HOME)/lib/vbjorb.jar:\
  $(ORACLE_HOME)/lib/vbjapp.jar$(JDK_HOME)/lib/classes.zip -ser2class\
  AccountServer/AccountImpl.sqlj
```

Обратите внимание на то, что команда sqlj генерирует два дополнительных файла классов: AccountImpl_SJProfile0 и AccountImpl_SJProfilekeys, которые также необходимо загружать на сервер. После этого можно создать файл Java-архива (.jar), содержащий все классы, которые нужно загрузить на сервер. Рекомендуется загружать каждый класс по очереди.

```
jar -cf0 accountjar.jar\
  account/Account*.class\
  account/NoAccountError*.class\
  account/_AccountImplBase.class\
  _st_Account.class\
  accountServer/AccountImpl*.class
```

Отметим, что трафаретный символ (*) используется при условии, что в каталоге нет других файлов с аналогичными именами. Наконец, загрузим все классы из файла .jar в JServer:

```
loadjava -oracleresolver -resolve -user scott/tiger accountjar.jar
```

Публикация имени серверного объекта CORBA

Последний этап подготовки серверного объекта — публикация его имени, которое будет искать клиентская программа. Для этого применяется команда publish:

```
publish -republish -user scott -password tiger -schema scott -service\
   sess_iiop://localhost:2481:ORCL\
   /test/Account AccountImpl account.AccountHelper
```

Используются следующие параметры и аргументы:

- -republish — перезаписать опубликованный объект с тем же именем.
- -user scott — имя пользователя публикующей схемы.
- -password tiger — пароль для scott.
- -schema scott — имя публикующей схемы.
- -service sess_iiop://localhost:2481:ORCL — этот параметр указывает URL, идентифицирующий базу данных, пространство имен сеансов которой нужно открыть. Хост-имя (здесь localhost) идентифицирует компьютер, на котором находится база данных (localhost возвращает хост-имя текущего компьютера), 2481 — порт прослушивающего процесса для iiop, а ORCL — системный идентификатор (SID) базы данных.
- /test/Account — имя опубликованного объекта. Публикуемое имя серверного объекта не обязано совпадать с именем интерфейса. Публикующий контекст /test в пространстве имен — это изначально существующий публикующий контекст, открытый для публикации.
- AccountImpl — имя Java-класса, реализующего серверный объект.
- AccountHelper — имя helper-класса для Account.

Программирование и компиляция клиента Java

Для обращения к серверному объекту по его опубликованному имени клиентская программа должна:

- Создать экземпляр JNDI-объекта initialContext и заполнить его необходимыми свойствами соединения.
- Вызвать метод lookup() объекта InitialContext, передав параметр URL с указанием искомых имени службы IIOP и имени серверного объекта. Метод lookup() возвращает ссылку на нужный серверный объект.
- Вызвать нужный метод серверного объекта по ссылке на объект.

Следующая клиентская программа AccountUser.java выполняет все эти действия, чтобы вызвать метод getAccount() по ссылке на серверный объект Account:

```
/* Имя программы:    AccountUser.java
**
** Назначение:       Получить сведения о счете 1056 с помощью серверного
**                   объекта Account.
**
*/
import account.*;
import accountServer.*;

// Импортируем константы свойств JNDI.
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

// Импортируем JNDI-интерфейс Context.
import javax.naming.Context;

// Импортируем класс InitialContext, реализующий интерфейс Context.
import javax.naming.InitialContext;

// Импортируем класс хэш-таблицы для хранения свойств среды начального контекста.
import java.util.Hashtable;

public class AccountUser {

  // main устанавливает, а не перехватывает исключительные ситуации.
  public static void main( String[] args ) throws Exception {

    // Создадим экземпляр и заполним объект InitialContext.
    Hashtable env = new Hashtable();
    env.put( Context.URL_PKG_PREFIXES, "oracle.aurora.jndi" );
    env.put( Context.SECURITY_PRINCIPAL, "scott" );
    env.put( Context.SECURITY_CREDENTIALS, "tiger" );
    env.put( Context.SECURITY_AUTHENTICATION,
             ServiceCtx.NON_SSL_LOGIN );
    Context ic = new InitialContext( env );

    // Найдем имя объекта, получив ссылку на объект.
    Account account = ( Account )
    ic.lookup( "sess_iiop://localhost:2481:ORCL/test/Account" );

    // Вызовем метод getAccountInfo().
    AccountInfo ai = account.getAccountInfo( 1056 );

    // Распечатаем сведения о счете.
    System.out.println( "Department number = " + ai.departmentno +
                        " Project number = " + ai.projectno );
  }
}
```

Создав программу для клиента Java, можно скомпилировать ее с помощью javac:

```
javac AccountUser.java
```

Выполнение клиента

Для выполнения клиентского класса AccountUser воспользуемся командой java. Необходимо установить для этой команды classpath, указав classes.zip (стандартный библиотечный архив Java), классы, подобные тем, что находятся в vbjapp.jar и vbjorb.jar и используются клиентом ORB, а также поставляемый вместе с Oracle8*i* jar-файл aurora_client.jar. В следующей командной строке соответствующим образом выполняется нужный класс Java:

- ```
  % java -classpath\
    .:$(ORACLE_HOME)/lib/aurora_client.jar:\
    $(ORACLE_HOME)/jdbc/lib/classes111.zip:\
    $(ORACLE_HOME)/sqlj/lib/translator.zip:\
    $(ORACLE_HOME)/lib/vbjorb.jar:\
    $(ORACLE_HOME)/lib/vbjapp.jar:$(JDK_HOME)/lib/classes.zipAccountUser
  ```

  Здесь JDK_HOME — место установки инструментария разработки Java Development Kit (JDK), а ORACLE_HOME — домашний каталог Oracle.
  ```

ПРИЛОЖЕНИЕ D

Краткий справочник по JavaServer Pages

В этом приложении дается краткий обзор синтаксиса JavaServer Pages (JSP) Specification 1.1 и JSP API. За дополнительными сведениями обращайтесь к JSP Specifications 1.0 и 1.1, а также к синтаксическим PDF-картам по адресу java.sun.com/products/jsp.

- Справочник по синтаксису JavaServer Pages (JSP)
- В состав JSP API входят пакеты `javax.servlet.jsp` и `javax.servlet.jsp.tagext`

Справочник по синтаксису JSP

В этом разделе предлагается краткий справочник по синтаксису JSP.

Директива page

Директива page описывает атрибуты страницы. Все они необязательны:

```
<%@ page список_атрибутов_директивы_page %>
список_атрибутов_директивы_page ::= { language=" язык_сценариев"}
{ extends=" имя_класса"}
{ import=" список_импорта"}
{ session="true | false" }
{ buffer="none | размер_в_kb" }
{ autoFlush="true | false" }
{ isThreadSafe="true | false" }
{ info=" информационный_текст" }
{ errorPage=" url_ошибок" }
{ content_type="text/html;charset=IANA-имя_набора_символов" }
{ isErrorPage="true | false" }
```

Директива `taglib`

Эта директива описывает пространство имен библиотеки тэгов для страницы:

```
<%@ taglib uri=" URI_библиотеки_тэгов" prefix=" префикс_тэгов" %>
...
<префикс_тэга:имя_тэга имя_атрибута="значение_атрибута" >
  JSP-содержимое
</префикс_тэга:имя_тэга>
< префикс_тэга:имя_тэга имя_атрибута="значение_атрибута" >
```

Дескриптор библиотеки тэгов

Дескриптор библиотеки тэгов (TLD, tag library descriptor) — это XML-документ, описывающий библиотеку тэгов. В нем содержатся документация на библиотеку в целом и на ее отдельные тэги, данные о версии JSP-контейнера и библиотеки тэгов, а также сведения о каждой из функций, описанной в библиотеке. DTD для LTD определено в java.sun.com/j2ee/dtds/web-jsptaglibrary_1_.dtd.

Тэги `include`

Существуют два тэга include: директива include и jsp:include. Директива include подключает статический файл, указанный с помощью URL. Ее синтаксис:

```
<%@ include file=" относительный_URL" %>
```

Тэг jsp:include подключает статический файл или файл, на который производится динамическая ссылка. Его синтаксис:

```
<jsp:include page="относительный_URL" flush="true"/>
```

Тэги JavaBean

Существуют три тэга для обработки зерен JavaBean страницы:

- jsp:useBean
- jsp:setProperty
- jsp:getProperty

Тэг jsp:useBean

Этот тэг проверяет наличие экземпляра зерна указанного класса и с указанной областью действия. Его синтаксис:

```
<jsp:useBean id= "имя_зерна"
   class="пакет.имя_класса"
   scope="page | request | session | application"
/>
```

Реже используются атрибуты type="имя_типа" и beanName="некоторое_зерно".

Тэг jsp:setProperty

Этот тэг устанавливает свойство зерна, ссылка на которое производится по имени:

```
<jsp:setProperty name="имя_другого_зерна"
    property="*|имя_свойства"
    value="новое_значение"
/>
```

Тэг jsp:getProperty

Этот тэг принимает указанное свойство и выводит его значение, которое включается в страницу как String:

```
<jsp:getProperty name="имя_другого_зерна"
    property="имя_свойства"
/>
```

Комментарии

Помимо комментариев программного текста Java, в исходный текст можно вводить и комментарии JSP. Существуют комментарии двух типов — JSP и HTML:

```
<%-- JSP-комментарии --%>
<!-- HTML-комментарии -->
```

Объявление переменных и функций

В JSP можно объявлять для страницы переменные и функции:

```
<%!
  int k = 0;
  int calculateSum (int a, int b) {
     // Производим вычисления
  }
%>
```

Скриптлеты

Скриптлеты применяются для встраивания программных конструкций Java в JSP-программы. Например:

```
<%
    // Здесь введем программные конструкции Java
%>
```

Выражения

В JSP-программах можно использовать выражения, возвращающие на страницу значение из программы Java:

```
<br> Hello, <%=username %> <br>
```

Тэг jsp:plugin

Этот тэг разрешает JSP-программам включать зерна и апплеты в состав клиентской страницы:

```
<jsp:plugin type="bean | applet"
code=" код_объекта"
codebase=" база_кода_объекта"
{ align=" выравнивание" }
{ archive=" список_архивов" }
{ height=" высота" }
{ hspace=" интервал_по_горизонтали" }
{ jreversion=" версия_jre" }
{ name=" имя_компонента" }
{ vspace=" интервал_по_вертикали" }
```

Тэг jsp:forward

Этот тэг позволяет передавать запрос клиента по другому URL, который может быть HTML-, jsp-файлом или файлом сервлета:

```
<jsp:forward page="относительный_URL" />

<jsp:forward page="относительный_URL" >
  <jsp:param имя="имя_параметра" значение="значение_параметра" />
</jsp:forward>
```

Неявные объекты

Неявным называется серверный объект, описанный в JSP-контейнере. Неявные объекты представлены в таблице D.1.

Таблица D.1. Неявные объекты JSP

Неявный объект	Тип	Область действия
request	javax.servlet.ServletRequest	Request
response	javax.servlet.ServletResponse	Page
pageContext	javax.servlet.jsp.PageContext	Page
session	javax.servlet.HttpSession	Session
application	javax.servlet.ServletContext	Application
out	javax.servlet.jsp.JspWriter	Page
config	javax.servlet.ServletConfig	Page
exception	java.lang.Throwable	Page
page	java.lang.Object	Page

Справочник по JSP API

В этом разделе дается общий обзор JSP API. За дополнительными сведениями о различных пакетах, интерфейсах и классах обращайтесь по адресам java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/package-summary.html и java.sun.com/j2ee/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/tag ext/package-summary.html.

JSP API состоит из двух пакетов:

- javax.servlet.jsp
- javax.servlet.jsp.tagext

Пакет javax.servlet.jsp

Пакет javax.servlet.jsp состоит из нескольких интерфейсов и классов. JSP-интерфейсы и их методы представлены в листинге D.1.

Листинг D.1. JSP-интерфейсы

```
// Интерфейс HttpJspPage
public interface HttpJspPage extends JspPage {
   public void _jspService (HttpServletRequest request,
                            HttpServletResponse response)
     throws ServletException, IOException;
} // Конец интерфейса HttpJspPage

// Интерфейс JspPage
public interface JspPage extends Servlet {
    public void jspDestroy();
    public void jspInit();
} // Конец интерфейса JspPage
```

JSP-классы и их методы представлены в листинге D.2.

Листинг D.2. JSP-классы

```
// Класс JspEngineInfo
public abstract class JspEngineInfo extends Object {
   // Конструктор
   public JspEngineInfo();
   // Метод
   public abstract String getSpecificationVersion();
} // Конец класса JspEngineInfo

// Класс JspFactory
public abstract class JspFactory extends Object {
   // Конструктор
   public JspFactory();

   // Методы
   public static getDefaultFactory();
   public static void setDefaultFactory (JspFactory aJspFactory);
   public abstract JspEngineInfo getEngineInfo();

   // Методы PageContext
   public abstract PageContext getPageContext
           (Servlet requestServlet,
            ServletRequest request,
            ServletResponse response,
            String errorPageUrl,
            boolean needsSession, int buffer,
            boolean autoFlush);
   public abstract void releasePageContext (PageContext pc);
} // Конец класса JspFactory

// Класс JspWriter
public abstract class JspWriter extends Writer {
   // Конструктор
   protected JspWriter (int bufferSize, boolean autoFlush);

   // Методы
   public abstract void clear() throws IOException;
```

```
    public abstract void clearBuffer() throws IOException;
    public abstract void close() throws IOException;
    public abstract void flush() throws IOException;
    public int getBufferSize();
    public abstract int getRemaining();
    public boolean isAutoFlush();
    public abstract void newLine()throws IOException;

    // Методы print
    public abstract void print (boolean b) throws IOException;
    public abstract void print (char c) throws IOException;
    public abstract void print (char[] charArray) throws IOException;
    public abstract void print (double d) throws IOException;
    public abstract void print (float f) throws IOException;
    public abstract void print (int i) throws IOException;
    public abstract void print (long l) throws IOException;
    public abstract void print (Object obj) throws IOException;
    public abstract void print (String str) throws IOException;

    // Методы println
    public abstract void println () throws IOException;
    public abstract void println (boolean b) throws IOException;
    public abstract void println (char c) throws IOException;
    public abstract void println (char[] charArray) throws IOException;
    public abstract void println (double d) throws IOException;
    public abstract void println (float f) throws IOException;
    public abstract void println (int i) throws IOException;
    public abstract void println (long l) throws IOException;
    public abstract void println (Object obj) throws IOException;
    public abstract void println (String str) throws IOException;
} // Конец класса JspWriter

// Класс PageContext
public abstract class PageContext extends Object {
    // Конструктор
    public PageContext();

    // Константы области действия
    public static final int APPLICATION_SCOPE;
    public static final int PAGE_SCOPE;
    public static final int REQUEST_SCOPE;
    public static final int SESSION_SCOPE;

    // Методы атрибутов
    public abstract Object findAttribute(String name);
    public abstract Object getAttribute(String name);
    public abstract Object getAttribute(String name, int scope);
    public abstract Enumeration
        getAttributeNamesInScope(int scope);
    public abstract int getAttributeScope(String name);
    public abstract void removeAttribute(String name);
    public abstract void removeAttribute (String name, int scope);
    public abstract void setAttribute (String name, Object obj);
    public abstract void setAttribute (String name,
                                       Object obj,
                                       int scope);

    // Методы исключительных ситуаций
    public abstract Exception getException();
    public abstract void handlePageException(Exception e )
          throws ServletException, IOException;

    // Метод forward
    public abstract void forward(String relativePath)
          throws ServletException, IOException;
```

```
    // Метод getOut
  public abstract JspWriter getOut();

    // Метод include
    public abstract void include(String relativePath)
      throws ServletException, IOException;

    // Метод initialize
    public abstract void initialize
             (Servlet requestServlet,
              ServletRequest request,
              ServletResponse response,
              String errorPageUrl,
              boolean needsSession, int buffer,
              boolean autoFlush)
      throws IOException, IllegalStateException,
             IllegalArgumentException;

    // Метод popBody
    public JspWriter popBody();

    // Метод pushBody
    public BodyContent pushBody();

    // Метод release
    public abstract void release();

    // Методы для получения компонентов сервлета
    public abstract Object getPage();
    public abstract ServletRequest getRequest();
    public abstract ServletResponse getResponse();
    public abstract ServletConfig getServletConfig();
    public abstract ServletContext getServletContext();
    public abstract HttpSession getSession();
} // Конец класса PageContext
```

Пакет javax.servlet.jsp.tagext

Этот пакет состоит из нескольких интерфейсов и классов. В листинге D.3 представлены интерфейсы и их методы, а в листинге D.4 — классы и их методы.

Листинг D.3. Интерфейсы тэгов

```
// Интерфейс BodyTag
public interface BodyTag extends Tag {
  // Константа
  public static final int EVAL_BODY_TAG;

  // Методы
  public int doAfterBody() throws JspTagException;
  public int doInitBody() throws JspTagException;
  public void setBodyContent(BodyContent bc);
} // Конец интерфейса BodyTag

// Интерфейс Tag
public interface Tag {
  // Константы
  public static final int SKIP_BODY;
  public static final int EVAL_BODY_INCLUDE;
  public static final int SKIP_PAGE;
  public static final int EVAL_PAGE;

  // Методы
  public int doEndTag() throws JspTagException;
  public int doStartTag() throws JspTagException;
```

```
   public Tag getParent();
   public void release();
   public void setPageContext(PageContext pc);
   public void setParent(Tag tag);
} // Конец интерфейса Tag
```

Листинг D.4. Классы тэгов

```
// Класс BodyContent
public abstract class BodyContent extends JspWriter {
   // Конструктор
   protected BodyContent();

   // Методы
   public void clearBody();
   public void flush() throws IOException;
   public JspWriter getEnclosingWriter();
   public abstract Reader getReader();
   public abstract String getString();
   public abstract void writeOut(Writer out)
     throws IOException;

} // Конец класса BodyContent

// Класс BodyTagSupport
public class BodyTagSupport extends TagSupport
    implements BodyTag {
   // Конструктор
   public BodyTagSupport();

   // Методы
   public int doAfterBody() throws JspTagException;
   public int doEndTag() throws JspTagException;
   public int doInitBody();
   public BodyContent getBodyContent();
   public JspWriter getPreviousOut();
   public void release();
   public void setBodyContent(BodyContent bc);

} // Конец класса BodyTagSupport

// Класс TagAttributeInfo
public class TagAttributeInfo extends Object {
   // Конструктор
   public TagAttributeInfo(String name,
                           boolean required,
                           boolean rtexprvalue,
                           String type,
                           boolean reqTime);

   // Методы
   public boolean canBeRequestTime();
   public static TagAttributeInfo
   getIdAttribute(TagAttributeInfo[] lists);
   public String getName();
   public String getTypeName();
   public boolean isRequired();
   public String toString();

} // Конец класса TagAttributeInfo

// Класс TagData
public class TagData extends Object
    implements Cloneable {
   // Конструктор
```

```
    public TagData(Object [][] attrs);
    public TagData(Hashtable attrs);

    // Методы
    public Object getAttribute(String name);
    public String getAttributeString(String name);
    public String getId();
    public void setAttribute(String name, Object value);

} // Конец класса TagData

// Класс TagExtraInfo
public abstract class TagExtraInfo extends Object {
    // Конструктор
    public TagExtraInfo();

    // Методы
    public TagInfo getTagInfo();
    public VariableInfo[] getVariableInfo(TagData td);
    public boolean isValid(TagData td);
    public void setTagInfo(TagInfo ti);

} // Конец класса TagExtraInfo

// Класс TagInfo
public class TagInfo extends Object

    public static final int BODY_CONTENT_JSP;
    public static final int BODY_CONTENT_TAG_DEPENDENT;
    public static final int BODY_CONTENT_EMPTY;
    // Конструктор
    public TagInfo(String name, String class,
                   String content, String infoString,
                   TagLibraryInfo lib, TagExtraInfo tsi,
                   TagAttributeInfo[] attr);

    // Методы
    public TagAttributeInfo[] getAttributes();
    public String getBodyContent();
    public String getInfoString();
    public String getTagClassName();
    public TagExtraInfo getTagExtraInfo();
    public TagLibraryInfo getTagLibraryInfo();
    public String getTagName();
    public VariableInfo[] getVariableInfo(TagData td);
    public boolean isValid(TagData td);
    public String toString();

} // Конец класса TagInfo

// Класс TagLibraryInfo
public class TagInfo extends Object {
    //Конструктор
    public TagLibraryInfo(String LibraryPrefix,
                          String libraryURI) ;

    // Методы
    public String getInfoString();
    public String getPrefixString();
    public String getReliableURN();
    public String getRequiredVersion();
    public String getShortName();
    public TagInfo getTag(String tagName);
    public TagInfo [] getTags();
    public String getURI();
```

```
} // Конец TagLibraryInfo

// Класс TagSupport
public class TagSupport extends Object
   implements Tag, Serializable {
   // Конструктор
   public TagSupport();

   // Методы
   public int doEndTag() throws JspTagException;
   public int doStartTag() throws JspTagException;
   public String getTagId();
   public void setTagId(String id);
   public Tag getParent();
   public void setParent(Tag t);
   public void release();
   public void setPageContext(PageContext pc);
   public Object getValue(String name);
   public Enumeration getValue();
   public void removeValue(String name);
   public void setValue(String name, Object value);

} // Конец класса TagSupport

// Класс VariableInfo
public class VariableInfo extends Object {
   // Конструктор
   public VariableInfo(String varName, String classname
                       boolean declare, int scope;

   // Методы и константы
   public static final int NESTED;
   public static final int AT_BEGIN;
   public static final int AT_END;
   public String getVarName();
   public String getClassName();
   public boolean getDeclare();
   public int getScope();

} // Конец класса TagSupport
```

За дополнительными сведениями о JSP обращайтесь к главам 2, 10 – 12, java.sun.com/products/jsp/ и к библиографическому справочнику в конце книги.

ПРИЛОЖЕНИЕ E

Установка и конфигурирование web-серверов для работы с Oracle JSP

В этом приложении говорится о том, как устанавливать и настраивать для работы с Oracle JSP следующие web-серверы:

- Tomcat (Servlet 2.2 API)
- Java Web Server (Servlet 2.1 API)
- Apache и JServ (Servlet 2.0 API)

В Oracle Internet Application Server (iAS) версии 1.0 используется среда Apache/JServ. Двигатель Oracle JSP устанавливается в процессе инсталляции iAS. В Oracle версии 8.1.7 также предусмотрена возможность выполнения сервлетов и JSP-программ в базе данных. В этом случае двигатель Oracle JSP устанавливается при инсталляции базы данных. Дополнительно в Oracle 8.1.7 включен Oracle HTTP Server, который функционирует вне базы данных и усилен комбинацией Apache/JServ и двигателем Oracle JSP. Кроме того, JSP-программы можно редактировать и отлаживать при помощи Oracle JDeveloper (см. приложение F).

Установка Oracle JSP в Tomcat

Tomcat – это эталонная реализация Servlet 2.2 API и JSP 1.1 с бесплатно распространяемым доступным исходным текстом, разработанная совместно фондом Apache Software Foundation, Sun Microsystems и др. Tomcat довольно просто настраивается для работы с Oracle JSP. Ниже описываются этапы установки и конфигурирования Tomcat и Oracle JSP.

Этапы установки

Tomcat может функционировать вместе с web-сервером Apache или автономно в качестве упрощенного web-сервера. Рассмотрим этапы установки и выполнения Tomcat в автономном режиме. За информацией об установке Tomcat вместе с Apache в другой конфигурации обращайтесь к документации на Tomcat.

Для выполнения JSP-программ с помощью Oracle JSP и Tomcat необходимо:

- Установить JDK
- Установить Tomcat
- Установить Oracle JSP
- Установить базу данных Oracle с JDBC и SQLJ
- Установить утилиты Oracle XML (необязательно)

Рассмотрим подробнее эти операции.

Установка JDK

Загрузите JDK 1.1 или выше с java.sun.com и установите его. Рекомендуется JDK 1.2 (Java 2), так как он содержит средства пулинга соединений JDBC 2.0 и другие новые полезные библиотеки. Предположим, что инструментарий установлен в каталоге `${JDK_HOME}` системы. Укажите в переменной среды CLASSPATH компилятор javac – он понадобится для компиляции исходных программ Java, генерируемых для JSP- файлов. Этот этап зависит от установленной версии JDK:

- Для JDK 1.2 укажите `${JDK_HOME}/lib/tools.jar` в CLASSPATH.
- Для JDK 1.1.* укажите `${JDK_HOME}/lib/classes.zip` в CLASSPATH.

Для NT используйте в маршрутах к каталогам соответствующие символы разделителей файлов (\) и маршрутов (;).

Установка Tomcat

Загрузите последнюю версию Tomcat для своей платформы с web-сайта jakarta.apache.org и установите сервер. Задайте переменную среды `${TOMCAT_HOME}` так, чтобы она указывала на корневой каталог инсталляции. Эта переменная используется сценариями запуска и остановки Tomcat (по умолчанию ее значением является текущий или родительский каталог относительно того, откуда вызываются сценарии).

Для применения Tomcat не в качестве автономного web-сервера, а вместе с Apache обратитесь к инструкциям, приведенным в файле "Tomcat+Apache-HOWTO" в каталоге `${TOMCAT_HOME}/docs`.

Установка Oracle JSP

Загрузите последнюю версию (1.1 или выше) Oracle JSP по сети Oracle Technology Network (technet.oracle.com/tech/servlets). Установите переменную среды `${OJSP_HOME}` надлежащим образом. Компонентом системы Oracle 8.1.7 являются еще и библиотеки Oracle JSP 1.1.

Установка Oracle с JDBC и SQLJ

Если сервлеты и JSP-программы взаимодействуют с базой данных, нужно также обращаться к экземпляру базы данных. Пробные варианты Oracle8*i* для NT и Linux можно загрузить с сайта сети Oracle Technology Network (technet.oracle.com/). Для установки базы данных следуйте предлагаемым инструкциям. Задайте переменную среды `${ORACLE_HOME}` так, чтобы она указывала на корневой каталог инсталляции. Библиотеки JDBC и SQLJ являются компонентами системы Oracle. Кроме того, можно загрузить их по отдельности с сайта сети Oracle Technology Network (technet.oracle.com/tech/java/sqlj_jdbc).

Установка утилит Oracle XML

При использовании XML в JSP-программах необходимо установить библиотеки Oracle XML. К ним относятся синтаксический анализатор Oracle XML (версии 2) и утилита Oracle XML-SQL (XSU). Они входят в состав Oracle 8.1.7, и, кроме того, их можно загрузить по отдельности с сайта сети Oracle Technology Network (technet.oracle.com/tech/xml) как часть инструментария разработчика Oracle XML Developer's Kit (XDK). Установите переменную среды `${XML_HOME}` надлежащим образом.

Этапы конфигурирования

Tomcat обладает своим собственным двигателем JSP, но ничто не мешает настроить Tomcat для работы с Oracle JSP, установив параметры Oracle JSP в соответствии с потребностями применяемой среды. Можно также модифицировать стандартную конфигурацию Tomcat (например, заменить порт прослушивающего процесса по умолчанию 8080 каким-либо иным). Для настройки Tomcat используются файлы двух типов: `server.xml` и `web.xml`.

Ниже описываются этапы конфигурирования Tomcat:

- Конфигурирование сервера Tomcat при помощи `server.xml`
- Конфигурирование Tomcat с Oracle JSP при помощи `web.xml`

Конфигурирование сервера Tomcat

Работу двигателя Tomcat можно настроить с помощью XML-файла `${TOMCAT_HOME}/conf/server.xml`. Например, чтобы изменить номер порта по умолчанию 8080 на номер 8085, нужно модифицировать этот файл следующим образом (строка выделена жирным шрифтом):

```
<Connector className="org.apache.tomcat.service.SimpleTcpConnector">
  <Parameter name="handler"
    value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
      <Parameter name="port" value="8085" />
</Connector>
```

Использование Oracle JSP вместе с Tomcat

Tomcat поставляется с собственным двигателем JSP (с именем Jasper). Однако конфигурацию сервера Tomcat можно изменить так, чтобы использовать в нем Oracle JSP, поставляемый вместе с дополнительными зернами и библиотеками для обращения к базам данных, а также с другими ценными средствами. Для указания Tomcat на необходимость использования двигателя Oracle JSP нужно:

- Добавить `oracle.jsp.JspServlet` к набору сервлетов. Этот сервлет является точкой входа для двигателя Oracle JSP.
- Отобразить JSP-расширения в URL на этот сервлет. Это называется *отображением расширений* (extension mapping).
- Добавить необходимые библиотеки в CLASSPATH.

Первые два шага выполняются при помощи файла `web.xml`. В нем содержатся имена сервлетов и их URL-отображения, а также сведения о конфигурации сеансов: таймаут, отображения библиотек тэгов и т.д. Глобальный файл `web.xml` находится в каталоге `${TOMCAT_HOME}/conf`. Если требуется применить для устанавливаемого web-приложения другие параметры, обновите файл `web.xml`, расположенный в `WEB-INF` ниже корня данного приложения. Например, для выполнения Oracle JSP в Tomcat можно модифицировать файл `${TOMCAT_HOME}/webapps/examples/WEB-INF/web.xml`.

Добавление oracle.jsp.JspServlet В раздел `<web-app>` файла `web.xml` добавьте следующие записи для сервлета Oracle JSP:

```
<servlet>
 <servlet-name>
   ojsp
 </servlet-name>
 <servlet-class>
   oracle.jsp.JspServlet
 </servlet-class>
       <init-param>
           <param-name>
             external_resource
           </param-name>
           <param-value>
             true
           </param-value>
           <param-name>
             unsafe_reload
           </param-name>
           <param-value>
             true
           </param-value>
       </init-param>
</servlet>
```

Этим объявлением сервлет с именем `oracle.jsp.JspServlet` добавляется к набору сервлетов данного приложения. Ему присваивается краткое имя `ojsp`, используемое в качестве псевдонима (см. следующий шаг). Здесь указаны два параметра инициализации: `external_resource` и `unsafe_reload`; они описываются для сервлета Oracle JSP при помощи элемента `<init-param>`. Имя и значение параметра задаются элементами `<param-name>` и `<param- value>`, которые включаются в блок `<init-param>`. Описание параметров конфигурации Oracle JSP приводится ниже.

Отображение JSP-расширений Необходимо отобразить JSP-сервлет на указатели URL, ссылающиеся на JSP-файлы. Для этого отобразите `oracle.jsp.JspServlet` на четыре различных шаблона URL: `*.jsp`, `*.JSP`, `*.sqljsp` и `*.SQLJSP`. Это значит, что данный сервлет должен вызываться web-сервером

для всех указателей URL запросов, заканчивающихся расширением .jsp, .JSP, .sqljsp или .SQLJSP. С этой целью добавьте в раздел отображения сервлетов файла web.xml следующие строки:

```
servlet-mapping>
 <servlet-name>
    ojsp
 </servlet-name>
 <url-pattern>
    *.jsp
 </url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>
    ojsp
 </servlet-name>
 <url-pattern>
    *.JSP
 </url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>
    ojsp
 </servlet-name>
 <url-pattern>
    *.sqljsp
 </url-pattern>
</servlet-mapping>
<servlet-mapping>
 <servlet-name>
    ojsp
 </servlet-name>
 <url-pattern>
    *.SQLJSP
 </url-pattern>
</servlet-mapping>
```

В данном отображении для класса oracle.jsp.JspServlet используется краткое имя ojsp (этот псевдоним был создан на шаге 2) и указывается, что сервлет должен вызываться для четырех суффиксов файлов.

Добавление библиотек Рассмотрим библиотеки трех различных типов:

- Oracle JSP
- JDBC и SQLJ
- XML

Опишем этапы добавления этих библиотек в среду Tomcat.

- **Добавление библиотек Oracle JSP** В системах, основанных на UNIX, скопируйте (или символически свяжите) библиотеки Oracle JSP ${OJSP_HOME}/lib/ojsp.jar и ${OJSP_HOME}/lib/ojsputil.jar с каталогом ${TOMCAT_HOME}/lib. Файлы jar этого каталога автоматически включаются в CLASSPATH сервера Tomcat. В системах, основанных на Windows, отредактируйте файл tomcat.bat в каталоге ${TOMCAT_HOME}\bin, включив эти два файла jar в переменную среды CLASSPATH.

В качестве альтернативы можно добавить эти библиотеки в CLASSPATH до запуска сервера Tomcat.

- **Добавление библиотек JDBC и SQLJ** Если в сервлетах и JSP-программах выполняются операции над базой данных, необходимо, помимо всего прочего, добавить в CLASSPATH библиотеки JDBC и SQLJ. Обычно они помещаются в каталог ${TOMCAT_HOME}/lib. Можно непосредственно

скопировать эти библиотеки либо указать на них символической ссылкой в системе Oracle. Библиотеки JDBC находятся в `${ORACLE_HOME}/lib/classes111.zip` (для JDK 1.1) и `${ORACLE_HOME}/lib/classes12.zip` (для JDK 1.2). Библиотеки транслятора SQLJ размещаются в `${ORACLE_HOME}/lib/translator.zip`. Кроме того, нужно добавить надлежащие классы исполняющей системы SQLJ в одну из следующих библиотек:

- `runtime12.zip` (для использования с JDK 1.2 и Oracle JDBC 8.1.7)
- `runtime12ee.zip` (для JDK 1.2.* Enterprise Edition с Oracle JDBC 8.1.7)
- `runtime11.zip` (для JDK 1.1.* с Oracle JDBC 8.1.7)
- `runtime.zip` (общие: для JDK 1.2 или 1.1 с любой версией Oracle JDBC)

В версии JDK 1.2 Enterprise Edition (ee) библиотек SQLJ поддерживаются *источники данных* (datasources) JDBC 2.0, согласующиеся с ISO-спецификацией SQLJ.

- **Добавление библиотек Oracle XML** При использовании в JSP-программах XML-конструкций добавьте библиотеки ORACLE XML — `${XML_HOME}/lib/xmlparserv2.jar` и `${XML_HOME}/lib/oraclexmlsql.jar` — в `${TOMCAT_HOME}/lib`. В качестве альтернативы можно добавить их в CLASSPATH до запуска сервера Tomcat.

Запуск сервера Tomcat

После установки перейдите в каталог `${TOMCAT_HOME}/bin` и запустите сервер Tomcat. В системе UNIX для запуска сервера используется сценарий `startup.sh`, а для остановки — `shutdown.sh`. В системах Windows для запуска и остановки сервера применяются пакетные (командные) файлы `startup.bat` и `shutdown.bat`. После запуска сервера начальная страница выводится при указании нужного URL в браузере, например http://localhost:8085/.

Выполнение JSP-программ в Tomcat

В комплект поставки Tomcat входит ряд примеров использования JSP. Ссылки на эти примеры находятся на начальной странице Tomcat, и мы рекомендуем выполнить их, прежде чем приступить к работе с собственными JSP-программами. Ниже показано, как выполнять в Tomcat программу `Hello.jsp` (см. главу 10) и приложение Order Online (см. главу 11).

Выполнение `Hello.jsp`

Разместите файл `Hello.jsp`, созданный в главе 10, в каталоге `${TOMCAT_HOME}/webapps/examples/jsp/chapter10/Hello.jsp`. Затем можно выполнить JSP, указав в браузере URL http://localhost:8085/examples/jsp/chapter10/Hello.jsp.

Можно также установить в Tomcat собственное web-приложение и файл `Hello.jsp`. В этом случае выполняются те же действия, что и при инсталляции приложения Order Online (см. ниже).

Выполнение приложения Order Online

Приложение Order Online было создано в главе 11. Оно состоит из следующих файлов:

- JSP-страницы `EnterOrder.jsp`, `InsertOrder.jsp`, `QueryOrder.jsp`
- Зерна `CartBean.java`, `DBInsertBean.sqlj` и `DBQueryBean.sqlj`, которые входят в состав Java-пакета `mybeans`

Внедрить приложение в Tomcat можно двумя способами:

- Как часть существующего приложения examples
- Как новое приложение OrderOnline

В первом случае нужно разместить JSP-файлы в каталоге ${TOMCAT_HOME}/webapps/examples/jsp/chapter11/, а классы JavaBean после компиляции — в ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes. Для компиляции зерен в системе UNIX и среде JDK 1.2 применяются следующие команды:

```
${JDK_HOME}/bin/javac -g \
     -d ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes \
     -classpath ${TOMCAT_HOME}/lib/servlet.jar \
      CartBean.java

${ORACLE_HOME}/bin/sqlj -g \
    -d ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes \
    -classpath ${ORACLE_HOME}/jdbc/lib/classes12.zip:\
${ORACLE_HOME}/sqlj/lib/translator.zip:\
${ORACLE_HOME}/sqlj/lib/runtime.zip:${TOMCAT_HOME}/lib/servlet.jar \
       DBInsertBean.sqlj DBQueryBean.sqlj
```

В системе Windows NT используются аналогичные команды. Компилируемые классы зерен помещаются в каталог ${TOMCAT_HOME}/webapps/examples/WEB-INF/classes, для чего служит параметр -d в javac. Этот каталог включается в CLASSPATH web-сервера Tomcat для приложения examples.

После этого можно вызвать первый экран приложения, введя нужный URL в браузере, например http://localhost:8085/examples/jsp/chapter11/EnterOrder.jsp.

Во втором случае, когда создается отдельное web-приложение с именем OrderOnline, необходимо ввести запись контекста в файл server.xml (см. выше):

```
<Context path="/OrderOnline"
         docbase="webapps/OrderOnline"
         reloadable="true" >
</Context>
```

В этом объявлении URL-путь к приложению указывается как /OrderOnline, а корень приложения — как каталог ${TOMCAT_HOME}/webapps/OrderOnline. В подкаталоге jsp можно разместить три JSP-файла. Параметр reloadable говорит о том, что зерна и другие классы следует перезагружать автоматически (без перезапуска сервера Tomcat), что ускоряет время разработки программ. Скомпилированные классы JavaBean размещаются, как правило, в каталоге ${TOMCAT_HOME}/webapps/OrderOnline/WEB-INF/classes. К каталогу WEB-INF нельзя получить доступ с помощью URL браузера, этот каталог по умолчанию включается в CLASSPATH сервера Tomcat. Другие вспомогательные классы (если они есть) нужно размещать в каталоге ${TOMCAT_HOME}/webapps/OrderOnline/WEB- INF/lib. Все jar- и zip-файлы этого каталога автоматически добавляются в CLASSPATH сервера.

После этого можно вызвать первый экран приложения Order Online, введя нужный URL, например

```
http://localhost:8085/OrderOnline/jsp/EnterOrder.jsp.
```

Установка и конфигурирование Java Web Server

Web-сервер Java (JWS, Java Web Server) входит в состав инструментария разработки Java Server Web Development Kit (JSWDK) и служит эталонной реализацией для Servlet 2.1 API и спецификации JSP 1.0. Сегодня JWS по большей части вытеснен сервером Tomcat, но некоторые пользователи продолжают применять эту среду.

Этапы установки

1. **Установка JDK.** См. процесс установки Tomcat, описанный выше. Не забудьте включить библиотеки компилятора javac в CLASSPATH.

2. **Установка JSWDK.** Загрузите JSWDK с web-сайта java.sun.com/products/servlet/index.html и установите его. Предположим, что он установлен в каталоге `${JWS_HOME}`.
3. **Установка Oracle JSP.** См. процесс установки Tomcat, описанный выше.
4. **Установка Oracle с JDBC и SQLJ.** См. процесс установки Tomcat, описанный выше.
5. **Установка Oracle XML.** См. процесс установки Tomcat, описанный выше.

Этапы конфигурирования

В этом разделе описывается функционирование Oracle JSP на web-сервере Java Web Server (JWS).

Использование Oracle JSP вместе с JWS

Как и Tomcat, JWS поставляется с собственным двигателем JSP, но его без труда можно настроить для выполнения Oracle JSP. Эта процедура аналогична той, что использовалась для Tomcat, и состоит из трех этапов:

- Добавление сервлета Oracle JSP к JWS
- Отображение JSP-расширений
- Добавление необходимых библиотек в среду JWS

Рассмотрим подробнее эти этапы.

Добавление сервлета Oracle JSP Модифицируйте файл `servlets.properties` в каталоге `WEB-INF` для каждого контекста сервлета, указав `oracle.jsp.JspServlet` в качестве сервлета, обрабатывающего JSP. Кроме того, пометьте символами комментариев отображение, ранее описанное для эталонной реализации JSP. Например, измените файл `${JWS_HOME}/examples/WEB-INF/servlets.properties` следующим образом:

```
#удалим: jsp.code=com.sun.jsp.runtime.JspServlet
#удалим: jsp.initparams=keepgenerated=true
jsp.code=oracle.jsp.JspServlet
jsp.initparams=external_resource=true,unsafe_reload=true
```

Режим работы двигателя Oracle JSP определяет строка `jsp.initparams`. Здесь указаны два параметра инициализации: `external_resource` и `unsafe_reload`. Описание этих и других параметров конфигурации Oracle JSP можно найти в конце приложения.

Отображение расширений JSP Отредактируйте файл `mappings.properties` в каталоге `WEB-INF` для каждого контекста сервлета, отобразив JSP- расширения следующим образом:

```
# Отобразим JSP-расширения
.jsp=jsp
.JSP=jsp
.sqljsp=jsp
.SQLJSP=jsp
```

Добавление библиотек Вам понадобятся библиотеки Oracle JSP, библиотеки Oracle JDBC и SQLJ и (необязательно) библиотеки Oracle XML:

- **Добавление библиотек Oracle JSP** Поместите библиотеки Oracle JSP — `${OJSP_HOME}/lib/ojsp.jar` и `${OJSP_HOME}/lib/ojsputil.jar` — в каталог `${JWS_HOME}/lib`. Кроме того, необходимо модифицировать сценарий `startserver` в корневом каталоге `${JWS_HOME}`, добавив библиотеки `ojsp.jar` и `ojsputil.jar` в переменную среды `jspJars`:

```
set jspJars=./lib/ojsp.jar:./lib/ojsputil.jar
```

- **Добавление библиотек Oracle JDBC и SQLJ** Поместите библиотеки JDBC и SQLJ (набор необходимых библиотек рассматривался выше при конфигурировании Tomcat) в каталог `${JWS_HOME}/lib`. Помимо этого, нужно модифицировать сценарий startserver в корневом каталоге `${JWS_HOME}`, добавив эти библиотеки в переменную среды `miscJars`:

  ```
  miscJars=./lib/translator.zip;./lib/runtime.zip;./lib/classes12.zip
  ```

- **Добавление библиотек Oracle XML** Скопируйте записанные библиотеки Oracle XML – `${XML_HOME}/xmlparserv2.jar` и `${XML_HOME}/lib/oraclexmlsql.jar` – в каталог `${JWS_HOME}/lib` . Кроме того, необходимо модифицировать сценарий startserver в корневом каталоге `${JWS_HOME}`, добавив эти библиотеки в переменную среды `miscJars`.

Выполнение JSP-программ в JWS

В состав JWS входит несколько примеров JSP, которые можно выполнить при помощи браузера. Ссылки на эти примеры размещены на начальной странице JWS, и мы рекомендуем запустить их, прежде чем приступить к работе с собственными JSP-программами. Убедившись в нормальной работе web-сервера, выполните в JWS программу `Hello.jsp` и приложение Order Online.

Выполнение `Hello.jsp`

Поместите файл `Hello.jsp` (см. главу 10) в каталог `${JWS_HOME}/examples/jsp/chapter10/Hello.jsp`. Теперь можно выполнить JSP, указав в браузере нужный URL, например http://localhost:8080/examples/jsp/chapter10/Hello.jsp.

Выполнение приложения Order Online

Три JSP-файла можно поместить в каталог `${JWS_HOME}/webapps/examples/jsp/chapter11/`. Скомпилированные классы JavaBean размещаются, как правило, в каталоге `${JWS_HOME}/examples/jsp/beans/classes`, который включается в CLASSPATH для JWS. После этого можно вызвать первый экран приложения при помощи соответствующего URL, например http://localhost:8080/examples/jsp/chapter11/EnterOrder.jsp.

В качестве альтернативы можно установить приложение Order Online как самостоятельное web-приложение (см. документацию на JSWDK).

Установка и конфигурирование Apache и JServ

Apache – популярный web-сервер с открытым программным текстом, распространяемый бесплатно. JServ – двигатель сервлетов, который реализует Servlet 2.0 API и для работы с web-сервером Apache применяет модуль `mod_jserv`. Apache и JServ используются в Oracle Internet Application Server (iAS) версии 1.0, в состав которого входит еще и двигатель Oracle JSP для исполнения страниц JavaServer Page. В этом разделе рассказывается об установке Apache и JServ, причем предполагается, что отдельные программные модули уже загружены. При использовании Oracle Internet Application Server установка и конфигурирование web-сервера Apache, двигателя сервлетов JServ и Oracle JSP являются частью процесса инсталляции iAS. Рекомендуем применять последний подход.

Этапы установки

Рассмотрим этапы установки Apache, JServ и Oracle JSP:

1. **Установка JDK.** См. процесс установки Tomcat, описанный выше.
2. **Установка Apache.** Загрузите с сайта Apache (www.apache.org/dist) последнюю версию web-сервера Apache для своей платформы. Выполните программу инсталляции. Предположим, что web-сервер установлен в каталоге `${APACHE_HOME}`. Отредактируйте файл конфигурации `${APACHE_HOME}/conf/httpd.conf` и удалите символ комментариев в строке

```
#ServerName localhost
```

Это позволит ссылаться на сервер в web-браузере при помощи URL http://localhost.

3. **Установка JServ и mod_jserv.** Загрузите исполнитель сервлетов JServ и модуль mod_jserv с web-сайта Apache (java.apache.org/jserv/dist) и установите их. Надлежащим образом задайте переменную среды ${JSERV_HOME}.
4. **Установка Oracle JSP.** См. процесс установки Tomcat, описанный выше.
5. **Установка Oracle JDBC и SQLJ.** См. процесс установки Tomcat, описанный выше.
6. **Установка библиотек Servlet 2.0 и 2.2.** Исполнитель сервлетов JServ реализует Servlet 2.0 API, упакованный как библиотека jsdk.jar, которая входит в состав JSDK 2.0 (www.javasoft.com/products/servlet/index.html). Хотя JServ использует только Servlet 2.0, двигателю Oracle JSP 1.1 для своих внутренних операций необходимы библиотеки Servlet 2.2. Загрузите файл servlet.jar с web-сайта www.javasoft.com/servlet/index.html, а затем выполните предлагаемые инструкции по настройке.

Этапы конфигурирования

1. **Конфигурирование JServ с Apache.** Чтобы двигатель сервлетов JServ мог работать вместе с web-сервером Apache, выполните инструкции, приведенные в документации на JServ.
2. **Отображение JSP-расширений на сервлет Oracle JSP.** Обновите файл ${JSERV_HOME}/conf/jserv.conf или файл ${JSERV_HOME}/conf/mod_jserv.conf (в зависимости от того, какой из них включен в файл ${APACHE_HOME}/conf/httpd.conf), добавив команды ApJServAction для отображения JSP-расширений:

```
# Отображение JSP-расширений
ApJServAction .jsp /servlets/oracle.jsp.JspServlet
ApJServAction .JSP /servlets/oracle.jsp.JspServlet
ApJServAction .sqljsp /servlets/oracle.jsp.JspServlet
ApJServAction .SQLJSP /servlets/oracle.jsp.JspServlet
```

3. **Добавление библиотек.** Необходимо добавить соответствующие команды wrapper.classpath в файл ${JSERV_HOME}/conf/jserv.properties. Файлы для среды JDK должны уже присутствовать в CLASSPATH. В следующем примере (где используются маршруты каталогов UNIX) подключаются файлы для библиотек Servlet 2.0 и 2.2, Oracle JSP, JDBC, SQLJ и XML:

```
# servlet 2.0 APIs (необходимы для Apache/JServ; из Sun JSDK 2.0):
wrapper.classpath=${JSDK_HOME}/lib/jsdk.jar
# servlet 2.2 APIs (загружаются в составе Oracle JSP):
wrapper.classpath=${OJSP_HOME}/lib/servlet.jar
# Пакеты OracleJSP:
wrapper.classpath=${OJSP_HOME}/lib/ojsp.jar
wrapper.classpath=${OJSP_HOME}/lib/ojsputil.jar
# Библиотеки JDBC 2.0 (JDK 1.2)
wrapper.classpath=${ORACLE_HOME}/lib/classes12.zip
# SQLJ-транслятор и классы исполняющей системы
wrapper.classpath=${ORACLE_HOME}/lib/translator.zip
wrapper.classpath=${ORACLE_HOME}/lib/runtime12.zip
# Библиотеки Oracle XML (применяются для XML/XSL)
wrapper.classpath=${XML_HOME}/lib/xmlparserv2.jar
wrapper.classpath=${XML_HOME}lib/oraclexmlsql.jar
```

Выполнение JSP-программ в Apache/JServ

Apache/JServ – это среда Servlet 2.0, поэтому понятие web-приложения в ней не сформулировано (в отличие от Tomcat, который является реализацией

Servlet 2.2). Однако в JServ принята концепция зон сервлетов. Концептуально *зона* (zone) сервлетов аналогична их контексту. Страницы Oracle JSP можно настроить так, чтобы они выполнялись в отдельной зоне. Об установке зон сервлетов рассказывается в документации на JServ.

Выполнение Hello.jsp

Для выполнения Hello.jsp нужно поместить этот JSP-файл в каталог ${APACHE_HOME}/htdocs/examples/jsp/. После этого можно вызывать JSP, указывая нужный URL в браузере, например URL http://localhost:8080/examples/jsp/Hello.jsp.

Выполнение приложения Order Online

Поместите три файла – EnterOrder.jsp, InsertOrder.jsp и QueryOrder.jsp – в каталог ${APACHE_HOME}/htdocs/examples/jsp/OrderOnline. В файле EnterOrder.jsp этого приложения присутствует следующая команда useBean:

```
<jsp:useBean id="cartBean"
             class="mybeans.CartBean"
             scope="session" />
```

В файл jserv.properties добавьте запись wrapper.classpath:

```
wrapper.classpath=${APACHE_HOME}/beans/
```

После этого скомпилируйте класс зерна и поместите его в каталог ${APACHE_HOME}/beans/mybeans/.

Теперь можно вызывать первую страницу EnterOrder.jsp этого приложения, используя соответствующий URL, например http://localhost:8080/examples/jsp/OrderOnline/EnterOrder.jsp.

Параметры конфигурации Oracle JSP

В этом разделе представлены параметры конфигурации Oracle JSP версии 1.1.0.0.0. Двигатель Oracle JSP вызывается как сервлет oracle.jsp.JspServlet, который имеет несколько параметров конфигурации, определяющих его поведение. Эти параметры устанавливаются в качестве инициализационных параметров сервлета oracle.jsp.JspServlet в соответствии с правилами для web-сервера и двигателя сервлетов (см. выше этапы конфигурирования web-сервера). Список параметров Oracle JSP и их краткое описание приводятся ниже. Некоторые из них имеют отношение только к определенным средам, например к Apache/JServ. Эти параметры указываются отдельно.

- alias_translation

Этот параметр специфичен для Apache/Jserv. Он содержит логическое значение, которое позволяет Oracle JSP обходить ограничения, налагаемые на способ обработки псевдонимов каталогов в Apache/JServ. Например, в файле ${APACHE_HOME}/conf/httpd.conf можно указать следующую команду, назначающую псевдоним каталогу:

- alias /images/ "/home/apache/images/"

Для того чтобы система Oracle JSP правильно обрабатывала ссылки на подобные маршруты с псевдонимами, необходимо установить параметр конфигурации alias_translation в значение true. По умолчанию значением этого параметра является false.

- bypass_source

Этот параметр имеет логическое значение и определяет, будет ли Oracle JSP проверять наличие исходного jsp-файла. Такой режим удобен, когда разработчик хочет внедрить только скомпилированные JSP-классы, а не JSP-источник. При установке этого параметра в значение true двигатель Oracle JSP

будет загружать и исполнять скомпилированный JSP-класс, даже если исходный JSP-файл отсутствует. Если источник доступен, его временная метка будет проверяться для определения, нужна ли трансляция (при условии, что параметр `developer_mode` (см. ниже) установлен в `true`). По умолчанию значением этого параметра является `false`.

- `classpath`

  Этот параметр служит для добавления записей в CLASSPATH, используемый по умолчанию Oracle JSP. Двигатель Oracle JSP загружает классы из своего собственного CLASSPATH (включая записи из данного параметра CLASSPATH), CLASSPATH системы, CLASSPATH web-сервера, из репозитория страниц и из других областей, относительных для корневых каталогов JSP-приложения. Отметим, что классы, указанные этим параметром, загружаются средством загрузки JSP-классов, а не загрузчиком классов системы (JDK). Смысл добавления записи двояк:

  - Эти классы (загружаемые средством загрузки JSP-классов) не могут ссылаться на классы, загружаемые системным или другими загрузчиками классов, и наоборот.
  - После модификации jar-файла в этом CLASSPATH в Oracle JSP выполняется автоматическая перезагрузка классов.

  По умолчанию значением данного параметра является `null`.

- `developer_mode`

  Этот параметр имеет логическое значение и определяет, нужно ли сопоставлять временную метку исходного JSP-файла с метками компилируемых файлов JSP-классов при вызове JSP-страницы. Если значением флага является `false`, Oracle JSP не проверяет временную метку, что ускоряет обработку JSP-запросов. Поэтому, если JSP-страницы не меняются, например в среде внедрения, устанавливайте флаг в значение `false`. В этом случае Oracle JSP будет проверять временные метки только первого запроса страницы и в последующих вызовах будет по-новому исполнять скомпилированный JSP-класс. По умолчанию значением параметра является `true`.

- `emit_debuginfo`

  Этот параметр имеет логическое значение и используется для указания транслятору Oracle JSP на необходимость генерации карты строк исходного JSP-файла для отладки. Параметр разрешается для отладки JSP-программ уровня источника в JDeveloper. По умолчанию значением этого параметра является `false`.

- `external_resource`

  Этот параметр с логическим значением управляет программным текстом, генерируемым транслятором Oracle JSP. Если значением этого флага является `true`, статическое содержимое JSP-страницы (т.е. HTML- или XML-текст) помещается в файл ресурса Java, а не в метод `_jspService()` генерируемого JSP-класса. Имя этого файла устанавливается по имени JSP-файла, но имеет расширение `.res`. Файл ресурса помещается в тот же каталог, что и файлы классов. Использование подобных файлов внешних ресурсов ускоряет трансляцию и исполнение JSP, особенно при наличии большого объема статического программного текста. Кроме того, это иногда помогает избежать превышения методом `_jspService()` размера в 64 Кбайт, что является ограничением Java. По умолчанию значение параметра — `false`.

- `javaccmd`

  Этот параметр указывает команду для компиляции класса Java, генерируемого для JSP. Он полезен при использовании компилятора, отличного от стандартного `javac` из JDK (этот компилятор применяется в Oracle JSP по умолчанию), или при указании для `javac` таких параметров командной строки, как:

- `javac -verbose -O`

  Использование этого параметра обязывает Oracle JSP вызывать не стандартный компилятор в той же виртуальной машине Java, где исполняется Oracle JSP, а другой компилятор в отдельном процессе. Для исполняемой программы компилятора можно указывать полностью определенный или неопределенный путь; в последнем случае Oracle JSP ищет исполняемую программу по системному маршруту. По умолчанию этот параметр не устанавливается.

- `page_repository_root`

  Этот параметр указывает полностью определенный корневой каталог для JSP-страниц. По умолчанию корневым каталогом является корневой каталог документов среды web-сервера Apache/JServ, а в средах Servlet 2.1 и 2.2 это корень контекстов сервлетов приложения, которому принадлежит JSP-страница. Параметр используется для указания другого корневого каталога. JSP-страницы должны находиться в этом корневом каталоге или в каком-либо подкаталоге. По умолчанию этот параметр не устанавливается.

- `session_sharing`

  Этот параметр специфичен для Apache/JServ. Он действует при использовании файла `globals.jsa` в качестве маркера приложения в средах Servlet 2.0, подобных Apache/JServ (см. документацию на Oracle JSP). Если параметр установлен в значение `true` (значение по умолчанию), для каждого вызова JSP двигатель Oracle JSP создает объект-оболочку для объекта `session` сервлетов, предоставляемого двигателем сервлетов. В этом случае данные JSP-сеанса (например, значение, связываемое с JSP-сеансом методом `session.putValue()`) передаются базовому сеансу сервлетов. Это позволяет JSP-программам одного приложения совместно пользоваться данными сеансов. Если же значением параметра `session_sharing` является `false`, данные JSP-сеанса не передаются сеансу сервлетов, и сервлеты не могут к ним обращаться. Этот параметр не действует, если файл `globals.jsa` не применяется для JSP-приложения.

- `sqljcmd`

  Этот параметр определяет команду для вызова SQLJ-транслятора. Он полезен при использовании SQLJ-транслятора, отличного от стандартного, или при указании параметров командной строки SQLJ, например, для проверки семантики:

- `sqljcmd= sqlj -ser2class -user scott -password tiger`

  Кроме того, в некоторых средах (JDK 1.1) при использовании этого параметра сообщения об ошибках становятся более информативными. Oracle JSP вызывает SQLJ-транслятор не в той же самой виртуальной машине Java, а в отдельном процессе (т.е. в отдельной виртуальной машине Java). Можно указать для исполняемой программы SQLJ полностью определенный путь либо разрешить ее обнаружение по системному маршруту. По умолчанию этот параметр не устанавливается.

- `translate_params`

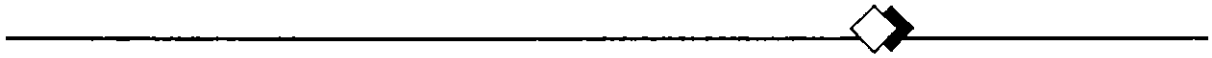
  Этот параметр с логическим значением определяет кодировку многобайтовых (NLS) параметров в HTTP-объекте `request`. В большинстве двигателей сервлетов NLS-кодировка параметров запроса не учитывается, поэтому верную кодировку во время трансляции должен обеспечивать JSP- программист. Если параметр установлен в значение `true`, данная задача автоматически решается двигателем Oracle JSP, т.е. он транслирует параметры HTTP-запроса и установки свойств зерен при помощи кодировки HTTP-объекта `request`. По умолчанию значением параметра является `false`, при этом Oracle JSP возвращает параметры HTTP-запроса в точности такими, какими они были получены от базового двигателя сервлетов (без какой бы то ни было дополнительной NLS-обработки).

- `unsafe_reload`

  Этот параметр определяет поведение двигателя Oracle JSP, когда JSP-страница автоматически меняется в процессе работы JSP-приложения. В этом случае стандартным режимом работы двигателя Oracle JSP является перезапуск приложения и сеансов сервлетов, что делает существующие сеансы недействительными. Такой режим не всегда желателен. Если установить параметр в значение `true`, Oracle JSP не будет перезапускать приложение после динамической трансляции JSP-страниц. Это защищает существующие сеансы и не позволяет делать их недостоверными. Заметим, однако, что данный режим небезопасен тем, что внесение в JSP-класс некорректных изменений может привести к установлению в классе исключительных ситуаций и к другим ошибкам этапа исполнения. Таким образом, этот параметр следует применять только в режиме разработки программ, но не в средах внедрения. По умолчанию значением параметра является `false`.

  За дополнительными сведениями о параметрах конфигурации Oracle JSP обращайтесь к руководству "OracleJSP Developer's Guide and Reference, Release 8.1.7" корпорации Oracle.

ПРИЛОЖЕНИЕ F



Инструментальные средства Oracle8i EJB и CORBA и поддержка JSP в JDeveloper

Это приложение состоит из двух частей: инструментальные средства Oracle8*i* EJB и CORBA и поддержка JSP в JDeveloper. Инструментальные средства для EJB и CORBA рассматриваются вместе потому, что их наборы во многом совпадают.

Инструментальные средства Oracle8*i* EJB и CORBA

Это часть состоит из трех разделов:

- Инструментальные средства Oracle8*i*, используемые для EJB
- Инструментальные средства Oracle8*i*, используемые как для EJB, так и для CORBA
- Инструментальные средства Oracle8*i*, используемые для CORBA

Инструментальные средства Oracle8*i*, используемые для EJB

В этом разделе описываются:

- `deployejb`
- `ejbdescriptor`

deployejb

Инструментальное средство `deployejb` готовит компонент EJB к использованию его клиентами. Оно работает с дескриптором внедрения зерна и с jar-файлом, содержащим интерфейсы, классы и подчиненные им классы EJB. Инструмент превращает создаваемый для зерна файл дескриптора внедрения в сериализованный объект Java, а также генерирует и компилирует классы, обеспечивающие связь между клиентами и опубликованными объектами. Напомним, что для публикации объектов в базе данных нужно иметь специальные системные привилегии Oracle. Скорее всего, эту задачу будет выполнять администратор базы данных Oracle. Отличным примером использования этого инструмента является шаг 9, описанный в разделе "Создание первого Enterprise JavaBean" главы 3. В Oracle версии 8.1.7 инструмент `deployejb` принимает дескриптор внедрения XML, требуемый новой спецификацией EJB 1.1.

Синтаксис инструмента `deployejb`:

```
deployejb -user <username> -password <password>
-service <serviceURL> -descriptor <file>
-temp <dir> <beanjar>
  [-addclasspath <dirlist>]
  [-describe]
  [-generated <clientjar>]
  [-help]
  [-iiop]
  [-keep]
  [-republish]
  [-role <role>]
  [-ssl]
  [-useServiceName]
  [-verbose]
  [-version]
```

Приведем пример использования `deployejb`:

```
deployejb -temp temp -u scott -p tiger
-s %ORACLE_SERVICE%
-descriptor My8iEJB.ejb -generated My8iEJBClient.jar My8iEJB.jar
```

Аргументы deployejb описаны в таблице F.1 ("Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide, Release 8.1.5" [52, стр. 6-37 – 6-38, таблица 6-18]).

Таблица F.1. *Краткие сведения об аргументах* `deployejb`

Аргумент	Описание и значения
`-user`	Указывает схему, в которую будут загружены EJB-классы.
`-password`	Указывает пароль для `<username>`.

`-service`	URL, идентифицирующий базу данных, в пространстве имен сеансов которой нужно опубликовать EJB. `serviceURL` имеет вид:
	`sess_iiop://<host>:<lport>:<sid>`
	`<host>` — компьютер, на котором расположена целевая база данных; `<lport>` — порт прослушивающего процесса, настроенный для прослушивания сеансового IIOP; `<sid>` — идентификатор экземпляра базы данных. Пример:
	`sess_iiop://localhost:2481:ORCL`
	что соответствует установке по умолчанию на машине вызывающего.

`-descriptor`	Указывает текстовый файл, содержащий дескриптор внедрения EJB.
`-temp`	Задает временный каталог для хранения промежуточных файлов, создаваемых утилитой `deployejb`. Если не указан `-keep`, `deployejb` по завершении работы удаляет файлы и каталог.
`<beanjar>`	Указывает имя `jar`-файла, содержащего интерфейс и классы реализации зерна.
`-addclasspath`	Указывает каталоги, в которых находятся интерфейс и/или зависящие от реализации классы, не содержащиеся в `<beanjar>`. Формат `<dirlist>` такой же, что и у аргумента `classpath` программы `javac`. Обязателен для `-beanonly`.
`-beanonly`	Пропускает этап генерации файлов интерфейсов. Полезен при изменении только реализации зерна.
`-describe`	Кратко описывает действие инструмента, а затем осуществляет выход из программы.
`-generated`	Задает имя выходного (генерируемого) `jar`-файла, в котором содержатся коммуникационные файлы, необходимые клиенту. Если не указан, именем выходного `jar`-файла становится имя входного `jar`-файла с добавлением `-generated`.
`-help`	Кратко описывает синтаксис инструмента, а затем осуществляет выход из программы.
`-iiop`	Устанавливает соединение с целевой базой данных с помощью IIOP вместо заданного по умолчанию ссансового IIOP. Используется при внедрении на сервере базы данных, сконфигурированном без сеансового IIOP.
`-keep`	Сохраняет временные файлы, генерируемые инструментом. Может быть полезен при отладке, поскольку обеспечивает доступ к исходным файлам, генерируемым утилитой `deployejb`.
`-republish`	Заменяет опубликованные атрибуты `BeanHomeName`, если `BeanHomeName` уже опубликовано; в противном случае публикует его.
`-role`	Указывает роль, принимаемую при подключении к базе данных; значения по умолчанию нет.
`-ssl`	Устанавливает соединение с базой данных посредством аутентификации и шифрования SSL.
`-verbose`	Выдает сообщения с подробным описанием состояния во время работы.
`-version`	Показывает версию инструмента, а затем осуществляет выход из программы.

ejbdescriptor

Инструментальное средство ejbdescriptor позволяет разработчику преобразовывать читаемый формат ASCII (.txt) файла дескриптора внедрения в нечитаемый (.ejb). Синтаксис команды:

❒ `ejbdescriptor {-parse | -dump} <infile> <outfile>`

Параметр -parse используется для создания сериализованного выходного файла <outfile> дескриптора внедрения из входного <infile>:

❒ `ejbdescriptor -parse My8iEJB.txt My8iEJB.ejb`

Параметр -dump служит для создания текстового файла <outfile> из сериализованного <infile> дескриптора внедрения:

❒ `ejbdescriptor -dump My8iEJB.ejb My8iEJB.txt`

В Oracle версии 8.1.7 инструмент ejbdescriptor преобразует собственный текстовый дескриптор Oracle, совместимый с предыдущими версиями Oracle8*i*, в XML-дескриптор, предписанный к применению спецификацией EJB 1.1.

Инструментальные средства Oracle8*i*, используемые как для EJB, так и для CORBA

В этом разделе описываются:

- sqlj
- dropjava
- remove
- sess_sh

sqlj

Oracle-инструмент sqlj используется для преобразования исходного программного текста SQLJ и для вызова компилятора Java. Синтаксис, применяемый для исполнения sqlj из командной строки:

❒ `sqlj <option_list> file_list`

option_list — список значений параметров SQLJ, разделенных пробелами. file_list — список файлов .sqlj, .java, .ser или .jar, разделенных пробелами. В именах файлов можно указывать трафаретный символ *. SQLJ-параметры описаны в таблице F.2 ("Oracle8*i* SQLJ Programming" [37, стр. 520 – 521]).

Таблица F.2. *Список параметров sqlj*

Параметр	Описание	Значение по умолчанию
-C<option>	Передает -<option> компилятору javac	
-classpath (только в командной строке)	Указывает classpath для Java VM и компилятора Java (передается в javac)	Нет
-compile=false	Не компилировать генерируемые файлы Java	true
-d=<directory>	Установить выходной каталог для генерируемых двоичных файлов .ser и .class	Отсутствует
-dir	Служит для установки выходного каталога для генерируемых в SQLJ .java-файлов	Отсутствует
-driver	Служит для указания драйвера JDBC для регистрации	Oracle.jdbc.driver.OracleDriver
-P<option>	Префикс, помечающий параметры для передачи в средство настройки профилей SQLJ	

Таблица F.2. *Список параметров sqlj (продолжение)*

Параметр	Описание	Значение по умолчанию
`Password=<password>`	Служит для установки пароля пользователя для соединения с базой данных при оперативной проверке семантики	Нет
`-profile=false`	Не настраивать генерируемые профильные файлы `*.ser`	`true`
`-props`	Служит для указания файла свойств	Нет
`-ser2class`	Преобразует генерируемые файлы `*.ser` в файлы `*.class`	`false`
`-status, -v`	Выводит состояние во время трансляции	`false`
`-url=<url>`	Указывает URL для оперативной проверки	`jdbc:oracle:oci8:@`
`-user, -u`	Разрешает оперативную проверку	Нет (никакой оперативной проверки семантики)
`-verbose` (только в командной строке)	Передается в `javac`; разрешает состояние	
`-warn`	Список разделенных запятыми флагов, разрешающих или запрещающих различные предупреждения; отдельные флаги: `precision/noprecision`, `nulls/nonulls`, `portable/noportable`, `strict/nostrict` и `verbose/noverbose`; глобальный флаг `all/none`	`precision` `nulls` `noportable` `strict` `noverbose`

В следующем примере специальный профиль не выполняется, файлы `.ser` преобразуются в файлы `.class`, выходной каталог для файлов `.class` устанавливается в значение `dist`:

```
sqlj -profile=false -ser2class -d=dist PIManager.sqlj SqljInJavaApplet.java
```

dropjava

Утилита `dropjava` используется для удаления схемных объектов Java с сервера. Команда `dropjava` противоположна команде `loadjava`. Синтаксис исполнения `dropjava` из командной строки (см. "Oracle8*i* SQLJ Programming" [37, стр. 527]):

```
dropjava {-u | -user} <user>/<password>[@<database>] [options]
{<file>.java | <file>.class | <file>.sqlj | <file>.jar | <file>.zip |
<resourcefile>} ...
```

Приведенные в таблице F.3 параметры описаны в "Oracle8*i* SQLJ Programming" [37, стр. 528].

Таблица F.3. *Краткие сведения об аргументах `dropjava`*

Параметр	Описание
`-user`	Указывает пользователя, пароль и необязательную строку соединения с базой данных.
`<filenames>`	Можно указать любое число и любую комбинацию имен файлов `.java`, `.class`, `.sqlj`, `.jar`, `.zip` и ресурсов в любом порядке. Файлы `.jar` и `.zip` должны быть распакованы.
`-oci8`	Использует JDBC-драйвер OCI. `-oci8` и `-thin` взаимно исключают друг друга. Если ничего не указано, по умолчанию применяется `-oci8`.

Таблица F.3. *Краткие сведения об аргументах dropjava (продолжение)*

Параметр	Описание
-schema	Определяет схему, из которой удаляются схемные объекты. Если схема не указана, используется схема регистрации (входа в систему). Для удаления схемного объекта из чужой схемы необходимо иметь системную привилегию DROP ANY PROCEDURE.
-thin	Использует тонкий JDBC-драйвер. -oci8 и -thin взаимно исключают друг друга. Если ничего не указано, по умолчанию применяется -oci8.
-verbose	Заставляет dropjava выдавать во время работы сообщения с подробным описанием состояния.

remove

Утилита remove удаляет PublishedObject или PublishingContext из пространства имен сеансов. Обратите внимание: соответствующий объект Java с сервера не удаляется; для этого служит утилита dropjava. Синтаксис команды remove представлен в "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-21]:

```
remove <name> -user <username> -password <password>
-service <serviceURL> [options]
  [{-d | -describe}]
  [{-h | -help}]
  [-iiop]
  [{-r | -recurse}]
  [-role <role>]
  [-ssl]
  [-version]
```

Аргументы remove описаны в таблице F.4 ("Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide, Release 8.1.5" [52, стр. 6-21 – 6-22, таблица 6-7]).

Таблица F.4. *Краткие сведения об аргументах remove*

Параметр	Описание
<name>	Имя удаляемого PublishingContext или PublishedObject.
-user	Указывает идентификационные характеристики для регистрации в экземпляре, заданном в -service.
-password	Указывает аутентифицирующий пароль для <username>, заданного в -user.
-service	URL, идентифицирующий базу данных, пространство имен сеансов которой "открывается" с помощью sess_sh. ServiceURL имеет вид: sess_iiop://<*host*>:<*lport*>:<*sid*> <host> — это компьютер, на котором расположена целевая база данных; <lport> — порт прослушивающего процесса, установленный для прослушивания сеансового IIOP; <sid> — идентификатор экземпляра базы данных. Пример: sess_iiop://localhost:2481:ORCL что соответствует установке по умолчанию на машине вызывающего.
-describe	Кратко описывает действие инструмента, а затем осуществляет выход из программы.
-help	Кратко описывает синтаксис инструмента, а затем осуществляет выход из программы.

Таблица F.4. *Краткие сведения об аргументах remove (продолжение)*

Параметр	Описание
-iiop	Устанавливает соединение с целевой базой данных с помощью IIOP вместо заданного по умолчанию сеансового IIOP. Используется при удалении с сервера базы данных, сконфигурированного без сеансового IIOP.
-recurse	Рекурсивно удаляет <name> и все подчиненные контексты PublishingContext; обязателен для удаления PublishingContext.
-role	Роль, принимаемая для выполнения remove; значения по умолчанию нет.
-ssl	Устанавливает соединение с базой данных посредством серверной SSL-аутентификации. Для использования этого параметра необходимо настроить базу данных на работу с SSL.
-version	Показывает версию инструмента, а затем осуществляет выход из программы.

Приведем пример вызова remove:

```
remove /test/Account -user scott -password tiger\
  -service sess_iiop://localhost:2481:ORCL
```

Эта команда удалит из пространства имен сеансов PublishedObject с именем /test/Account.

sess_sh

Инструментальное средство sess_sh (session shell – оболочка сеансов) – это интерактивный интерфейсный инструмент, позволяющий передавать команды, сходные с командами оболочки UNIX, для работы в пространстве имен сеансов экземпляра базы данных, т.е. для обработки контекстов PublishingContext и объектов PublishedObject. Например, команда

```
ls /test
```

выдает список всех объектов PublishedObject и контекстов PublishingContext в PublishingContext с именем test.

Аргументы соединения с базой данных указываются при исполнении команды sess_sh. Синтаксис команды sess_sh:

```
sess_sh [options] -user <user>
-password <password> -service <serviceURL>
[-d | -describe]
[-h | -help]
[-iiop] [-role <rolename>]
[-ssl] [-useServiceName] [-version]
```

Приведем пример использования команды sess_sh:

```
sess_sh -user scott -password tiger
 -service sess_iiop://data-i.com:2481:ORCL
```

Аргументы команды sess_sh представлены в таблице F.5 (см. "CORBA Developer's Guide, Release 2 (8.1.7)" [32, стр. 7-9, таблица 7-5]).

Таблица F.5. *Краткие сведения об аргументах sess_sh*

Параметр	Описание
-user	Указывает имя пользователя для подключения к базе данных.
-password	Указывает пароль пользователя для подключения к базе данных.

Таблица F.5. *Краткие сведения об аргументах sess_sh (продолжение)*

Параметр	Описание
-service	URL, идентифицирующий базу данных, пространство имен сеансов которой "открывается" с помощью sess_sh. ServiceURL имеет вид: sess_iiop://<host>:<lport>:<sid> <host> — это компьютер, на котором расположена целевая база данных; <lport> — порт прослушивающего процесса, установленный для прослушивания сеансового IIOP; <sid> — идентификатор экземпляра базы данных. Пример: sess_iiop://localhost:2481:ORCL что соответствует установке базы данных по умолчанию на машине вызывающего.
-describe	Кратко описывает действие инструмента, а затем осуществляет выход из программы.
-help	Кратко описывает синтаксис инструмента, а затем осуществляет выход из программы.
-iiop	Устанавливает соединение с целевой базой данных с помощью обычного IIOP вместо заданного по умолчанию сеансового IIOP. Этот параметр используется на сервере базы данных, сконфигурированном без сеансового IIOP.
-role	Роль, передаваемая в базу данных; значения по умолчанию нет.
-ssl	Устанавливает соединение с базой данных посредством серверной SSL-аутентификации. Для использования этого параметра необходимо настроить базу данных на работу с SSL и указать порт SSL для этого режима.
-useServiceName	Этот флаг необходимо указывать при использовании в URL имени службы вместо SID. В противном случае инструмент предполагает, что последней строкой символов в URL является SID.
-version	Показывает версию команды, а затем осуществляет выход из программы.

Таблица F.6. *Команды, передаваемые в sess_sh*

Команда	Назначение
cd	Аналогична команде cd оболочки UNIX; изменяет рабочий PublishingContext.
chmod	Аналогична команде chmod оболочки UNIX; изменяет пользователей или роли, имеющие права на PublishingContext или PublishedObject.
chown	Аналогична команде chown в UNIX; изменяет право владения PublishingContext или PublishedObject. Владельцем вновь создаваемого PublishingContext или PublishedObject является пользователь, который его публикует.
exit	Завершает работу sess_sh.
help	Выводит краткий синтаксис команд оболочки сеансов.
java	Аналогична команде java в JDK; вызывает статический метод main() класса. Класс до этого должен быть загружен с помощью loadjava.
lcd	Команда lcd (local cd) изменяет локальный рабочий каталог точно так же, как и при исполнении cd вне оболочки сеансов.
lls	Команда lls (local ls) выводит список содержимого рабочего каталога точно так же, как и при исполнении ls вне оболочки сеансов.

Таблица F.6. *Команды, передаваемые в sess_sh (продолжение)*

Команда	Назначение
`ln`	Команда `ln` (link — связь) аналогична команде `ln` в UNIX. Связь является синонимом `PublishingContext` или `PublishedObject`. Она помогает сохранить достоверность ссылки на `PublishingContext` или `PublishedObject` при перенесении `PublishingContext` или `PublishedObject` (см. Команду `mv`); создание связи со старым именем делает объект доступным и по старому, и по новому имени.
`lpwd`	Команда `lpwd` (local print working directory — локальный рабочий каталог печати) отображает имя рабочего каталога точно так же, как и при исполнении `pwd` вне оболочки сеансов.
`ls`	Команда `ls` (list — список) показывает содержимое контекстов `PublishingContext` так же, как в UNIX команда `ls` показывает содержимое каталогов.
`mkdir`	Аналогична команде `mkdir` оболочки UNIX; создает `PublishingContext`. Для использования `mkdir` в целевом `PublishingContext` необходимо иметь право записи в него информации.
`mv`	Команда `mv` (move — перенос) аналогична команде `mv` оболочки UNIX.

Инструментальные средства Oracle8i, используемые для CORBA

В этом разделе описываются инструменты `loadjava` и `publish`.

loadjava

Утилита `loadjava` используется для преобразования class-файлов в библиотечные единицы базы данных, называемые *схемными объектами Java-классов* (Java class schema objects) и хранимые на сервере, для преобразования ser-файлов в схожие единицы, называемые *схемными объектами ресурсов* (resource schema objects) и тоже хранимые на сервере, и для преобразования исходных файлов в библиотечные единицы, называемые *схемными объектами источников* (source schema objects) и хранимые на сервере. Для каждого class-файла, каждого ser-файла и каждого java-файла источника создается отличный от других схемный объект.

В командной строке `loadjava` можно указать каждый class- или ser-файл по отдельности либо общий jar-файл (архив Java), объединив предварительно все файлы. Для объединения файлов в jar-файл применяется утилита `jar`. В jar-файле можно объединить файлы классов и ресурсов или файлы источников и ресурсов, но только не файлы источников и классов. Синтаксис инструмента `loadjava` описан в "Oracle8*i* SQLJ Programming" [37, стр. 525]:

```
loadjava {-user | -u} <user>/<password>[@<database>] [options] {<file>.java |
<file>.class | <file>.jar | <file>.zip | <file>.sqlj | <resourcefile>} ...
```

Параметры команды приведены в таблице F.7 ("Oracle8i SQLJ Programming" [37, стр. 526 – 527]):

Таблица F.7. *Краткие сведения об аргументах loadjava*

Параметр	Описание
`<filenames>`	Можно указать любое число и любую комбинацию имен файлов `.java`, `.class`, `.sqlj`, `.jar`, `.zip` и ресурсов в любом порядке.
`-andresolve`	Заставляет `loadjava` компилировать источники, если те были загружены, и определять внешние ссылки в каждом классе по мере его загрузки. Параметры `-andresolve` и `-resolve` являются взаимоисключающими.
`-debug`	Заставляет компилятор Java генерировать отладочную информацию.

Таблица F.7. *Краткие сведения об аргументах* `loadjava` *(продолжение)*

Параметр	Описание
`-definer`	По умолчанию схемные объекты классов действуют с привилегиями их вызывающего. Этот параметр предоставляет классам привилегии определяющего (описывающего).
`-grant <grants>`	Предоставляет привилегию EXECUTE (исполнение) на загруженные классы указанным пользователям и ролям. Можно задавать любое число пользователей и ролей в любой комбинации, отделяя их друг от друга запятыми, но не пробелами (`-grant Bob,Betty`, но не `-grant Bob, Betty`).
`-oci8`	Использует JDBC-драйвер OCI; `-oci8` и `-thin` взаимно исключают друг друга; если ничего не указано, по умолчанию используется `-oci8`.
`-oraclere-solver`	Использует определитель, требующий, чтобы были найдены все ссылки на классы.
`-resolve`	Компилирует (при необходимости) и определяет внешние ссылки в классах после загрузки всех классов из командной строки. Параметры `-andresolve` и `-resolve` являются взаимоисключающими.
`-Resolver <resolver>`	Использует определитель, требующий, чтобы были найдены все ссылки на классы.
`-schema`	Определяет схему, в которой создаются схемные объекты. Если схема не указана, используется схема регистрации (входа в систему). Для создания схемного объекта не в собственной схеме необходимо иметь привилегию CREATE PROCEDURE или CREATE ANY PROCEDURE.
`-synonym`	Создает общий (PUBLIC) синоним для загруженных классов, делая их доступными извне схемы, в которую они загружаются. Необходимо иметь привилегию CREATE PUBLIC SYNONYM.
`-thin`	Использует тонкий JDBC-драйвер. `-oci8` и `-thin` взаимно исключают друг друга. Если ничего не указано, по умолчанию применяется `-oci8`.
`-user, -u`	Указывает пользователя, пароль и строку соединения с базой данных. Вид аргумента: `<username>/<password>[@<database>]`
`-verbose`	Заставляет `loadjava` выдавать во время работы сообщения с подробным описанием состояния.

publish

У каждого экземпляра базы данных, в котором функционирует программное обеспечение JServer, имеется пространство имен сеансов, используемое Oracle8*i* ORB для активизации серверных объектов CORBA. *Пространство имен сеансов* (session namespace) – это иерархическая совокупность объектов, называемых опубликованными объектами (`PublishedObjects`) и публикующими контекстами (`PublishingContexts`).

`PublishedObjects` – нижний уровень иерархии, а `PublishingContexts` – узлы верхнего уровня, что аналогично файлам и каталогам в системе UNIX. Каждый `PublishedObject` соответствует схемному объекту класса, представляющему CORBA- или EJB-реализацию. Для активизации серверного объекта CORBA клиент ссылается на имя соответствующего опубликованного объекта.

Процесс создания `PublishedObject` называется *публикацией* (publishing) и выполняется с помощью утилиты `publish` или интерактивной оболочки сеансов (см. выше). Опубликованные объекты CORBA создаются этими утилитами после загрузки на сервер реализаций соответствующих объектов с помощью утилиты `loadjava`.

Синтаксис команды `publish` описан в "Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide, Release 8.1.5" [52, стр. 6-19]:

```
publish <name> <class> [<helper>] -user <username> -password <password>
-service <serviceURL> [options]
```

```
[-describe]
[{-g | -grant} {<user> | <role>}[,{<user> | <role>}]...]
[{-h | -help}]
[-iiop]
[-role <role>]
[-republish]
[-schema <schema>]
[-ssl]
[-version]
```

Аргументы командной строки publish приведены в таблице F.8 ("Oracle8*i* Enterprise JavaBeans and CORBA Developer's Guide Release 8.1.5" [52, стр. 6-19 — 6-21, таблица 6-6]).

Таблица F.8. *Краткие сведения об аргументах publish*

Параметр	Описание
<name>	Имя создаваемого или повторно публикуемого PublishedObject; при необходимости создаются PublishingContexts.
<class>	Имя схемного объекта класса, соответствующего <name>.
<helper>	Имя схемного объекта Java-класса, реализующего метод narrow() для <class>.
-user	Указывает идентификационные характеристики для регистрации в экземпляре базы данных, указанном в -service.
-password	Указывает аутентифицирующий пароль для имени пользователя, заданного в -user.
-service	URL, идентифицирующий базу данных, пространство имен сеансов которой "открывается" с помощью sess_sh. ServiceURL имеет вид: sess_iiop://<host>:<lport>:<sid> <host> — это компьютер, на котором расположена целевая база данных; <lport> — порт прослушивающего процесса, установленный для прослушивания сеансового IIOP; <sid> — идентификатор экземпляра базы данных. Пример: sess_iiop://localhost:2481:ORCL что соответствует установке по умолчанию на машине вызывающего.
-describe	Кратко описывает действие инструмента, а затем осуществляет выход из программы.
-grant	После создания или повторной публикации объекта PublishedObject предоставляет права read и execute последовательности имен <user> и <role>. При повторной публикации заменяет существующих пользователей/роли с правами read/execute на имена <user> и <role>. Для селективного изменения прав PublishedObject используется команда chmod утилиты sess_sh. Обратите внимание: для активизации объекта CORBA или зерна EJB пользователь должен иметь право execute как на PublishedObject, так и на соответствующий схемный объект класса.
-help	Кратко описывает синтаксис инструмента, а затем осуществляет выход из программы.
-iiop	Устанавливает соединение с целевой базой данных с помощью IIOP вместо заданного по умолчанию сеансового IIOP. Этот параметр используется при публикации на сервере базы данных, сконфигурированном без сеансового IIOP.
-role	Роль, принимаемая для выполнения publish; значения по умолчанию нет.
-republish	Обязывает publish заменять имеющийся PublishedObject. При отсутствии этого параметра инструмент publish отказывается от попытки публикации существующего имени. Если PublishedObject не существует, publish создает его. При повторной публикации права, не принадлежащие владельцу, удаляются; для добавления прав read/execute при повторной публикации используется параметр -grant.

Таблица F.8. *Краткие сведения об аргументах `publish` (продолжение)*

Параметр	Описание
`-schema`	Схема, содержащая схемный объект `<class>` Java. Если не указана, инструмент `publish` использует схему вызывающего.
`-ssl`	Устанавливает соединение с базой данных посредством серверной SSL-аутентификации. Для использования этого параметра необходимо настроить базу данных на работу с SSL и указать порт прослушивающего процесса SSL в `-service`.
`-version`	Показывает версию инструмента, а затем осуществляет выход из программы.

Поддержка JSP в JDeveloper

Oracle предлагает целое семейство инструментальных средств прикладной разработки и сбора информации для построения Интернет-приложений под общим названием *Oracle Internet Developer Suite*. Этот набор инструментов основан на таких Интернет-стандартах, как Java, XML, EJB, Servlet, JSP и CORBA. Компонентами Oracle Internet Developer Suite являются:

- **Oracle Designer**, предоставляющий ориентированную на выполнение задач среду для моделирования и создания серверных DDL-приложений, приложений клиент/сервер и web-приложений, использующих мощь и переносимость пользовательских интерфейсов Java и HTML.
- **Oracle Forms Developer**, предоставляющий среду для построения Интернет-приложений класса предприятия, основой которых является база данных. Oracle Forms Developer – фундамент Oracle Applications 11*i*.
- **Oracle JDeveloper** – инструмент разработки приложений, обеспечивающий интегрированную поддержку построения полнофункциональных приложений электронной коммерции для Интернета. В JDeveloper предлагается интегрированная среда для разработки, отладки, внедрения, многократного применения и настройки многоуровневых компонентных приложений Java, Servlet, JSP и XML.
- **Oracle Reports Developer**, позволяющий составлять и публиковать динамически генерируемые web-отчеты. Отчеты могут публиковаться по всему предприятию при помощи стандартного web-браузера в любом формате, в том числе в HTML, HTMLCSS, PDF, в виде текста с ограничителями, RTF, PostScript, PCL и XML.
- **Oracle Discoverer** – специальный инструмент для создания запросов, отчетов, для анализа и публикации в Интернете, позволяющий пользователям на всех уровнях организации обращаться к информации хранилищ реляционных данных и систем оперативной обработки транзакций.
- **Oracle Portal** – web-приложение для построения и внедрения производственных порталов, предлагающее удобную среду для предоставления сетевых программных услуг и информационных ресурсов.

За дополнительными сведениями об инструментальных средствах разработки Oracle обращайтесь на technet.oracle.com.

Ниже кратко описываются этапы создания и выполнения JSP-приложения при помощи инструмента Oracle JDeveloper. Более подробная информация по разработке приложений JavaServer Pages (JSP) приведена в главе 2 и в части IV (главы 10 – 12).

В разделе "Написание Hello.jsp" главы 10 была разработана наша первая JSP-программа. Здесь с помощью инструмента Oracle JDeveloper версии 3.1 создается простая JSP-программа `Hello.jsp`. Она будет выводить приветствие клиенту, текущие дату и время.

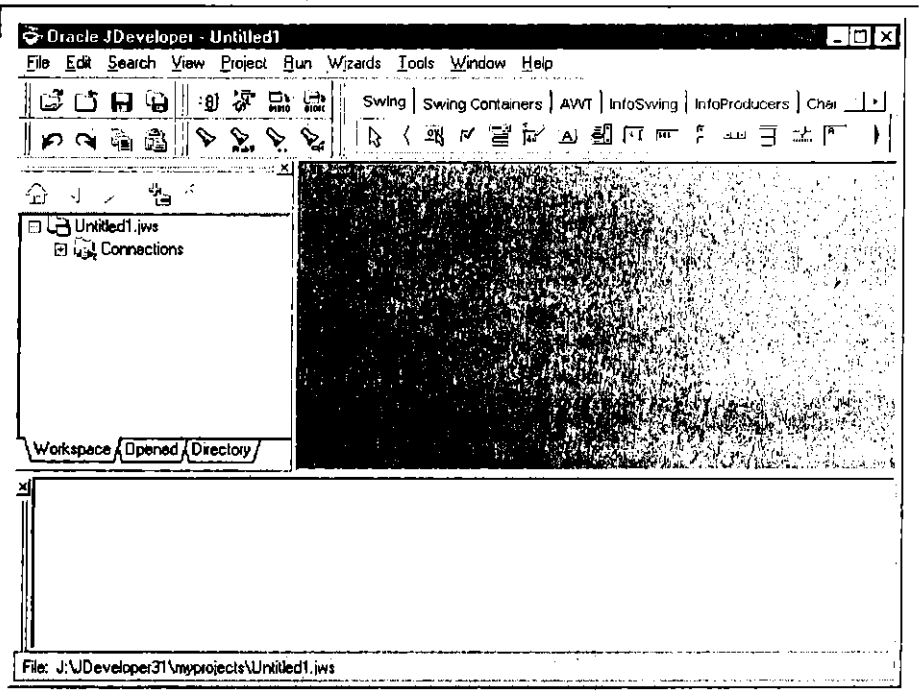
Процесс разработки нового JSP-приложения с помощью JDeveloper довольно прост. Он включает в себя следующие этапы:

1. Создание проекта.
2. Добавление JSP-файла в проект для создания файла с исходным программным текстом.
3. Выполнение программы.

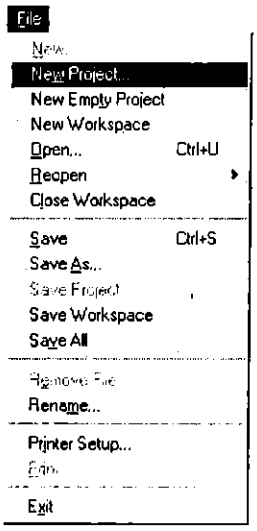
Создание нового приложения JavaServer Pages

Запустите JDeveloper (см. рис. F.1). Если JDeveloper использовался ранее, будет автоматически открыто рабочее пространство (Workspace) приложения, созданного последним. Для нового проекта необходимо создать новое рабочее пространство. Следовательно, нужно сначала закрыть рабочее пространство другого приложения. Для этого щелкните мышью на File, Close Workspace.

Рис. F.1. Oracle JDeveloper — Untitled

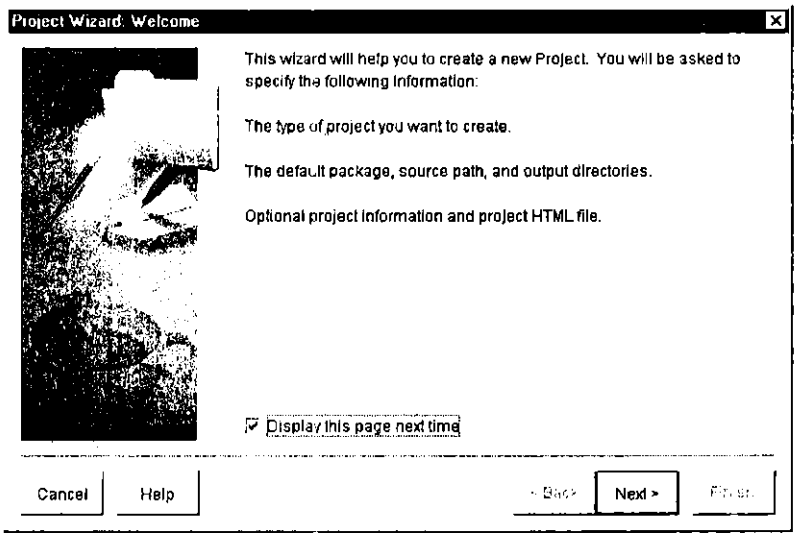


1. Создайте новый проект: щелкните мышью на File, а затем на New Project, как показано ниже. При создании нового проекта JDeveloper автоматически создает для него новое рабочее пространство.



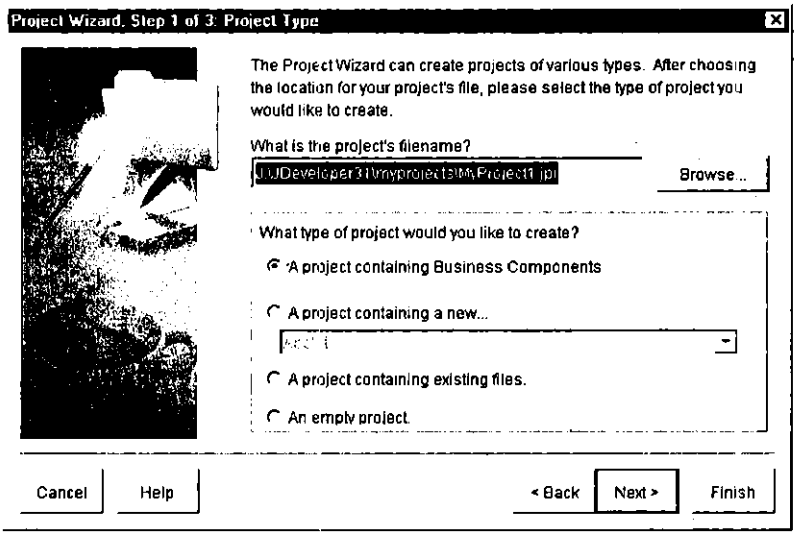
2. С помощью программы-мастера Project Wizard создайте новое web-приложение (см. рис. F.2).

Рис. F.2.
Экран Welcome в Project Wizard



3. В окне Welcome мастера Project Wizard щелкните мышью на кнопке Next. В появившемся окне (см. рис. F.3) можно либо принять значение, предлагаемое по умолчанию, либо указать путь, имя файла или их комбинацию. Значение по умолчанию -- `J:\JDeveloper31\myprojects\MyProject1.jpr`.

Рис. F.3.
Project Wizard, этап 1



4. Воспользовавшись кнопкой `Browse`, выберите путь и укажите нужное имя (см. рис. F.4). Измените `MyProject1` на `MyFirstJspApp` и щелкните мышью на Open.

5. После щелчка мыши на Open в окне Select Project вновь выводится окно Project Wizard. Щелкните мышью на Empty Project, а затем на Next (см. рис. F.5).

6. В этот момент можно щелкнуть мышью на кнопке Finish, приняв остальные значения по умолчанию, предлагаемые инструментом JDeveloper, либо щелкнуть мышью на кнопке Next и продолжить установку значений. Щелкните мышью на кнопке Next. На этапе 2 мастер позволяет ввести имя нового пакета и выбрать каталоги для хранения исходного программного текста Java и JSP, а также файлов `.class` (см. рис. F.6).

Рис. F.4. Выбор проекта с помощью кнопки Browse

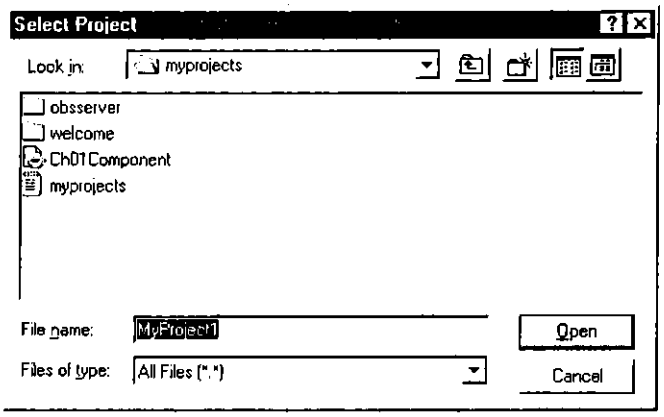


Рис. F.5. Выбор имени файла проекта

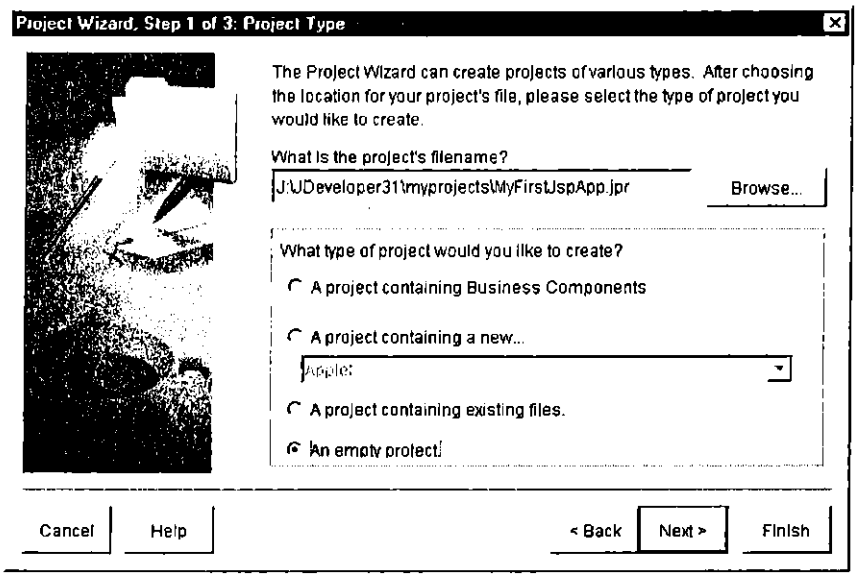
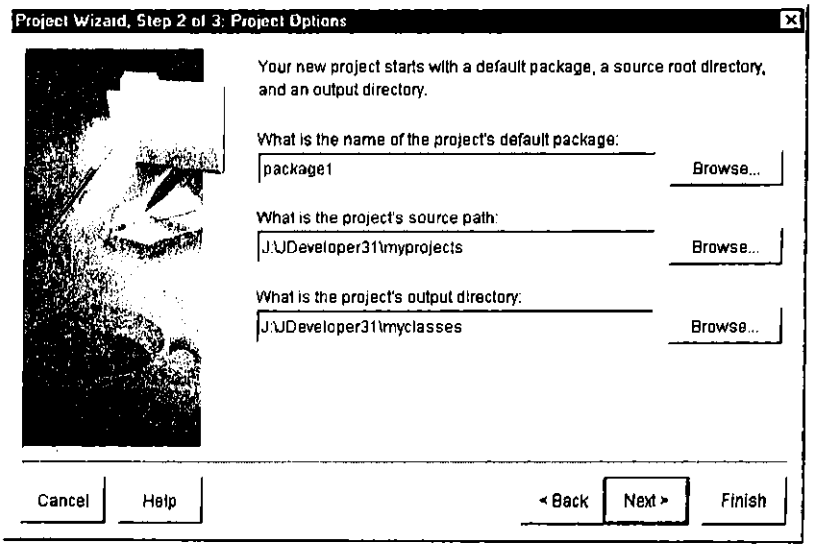


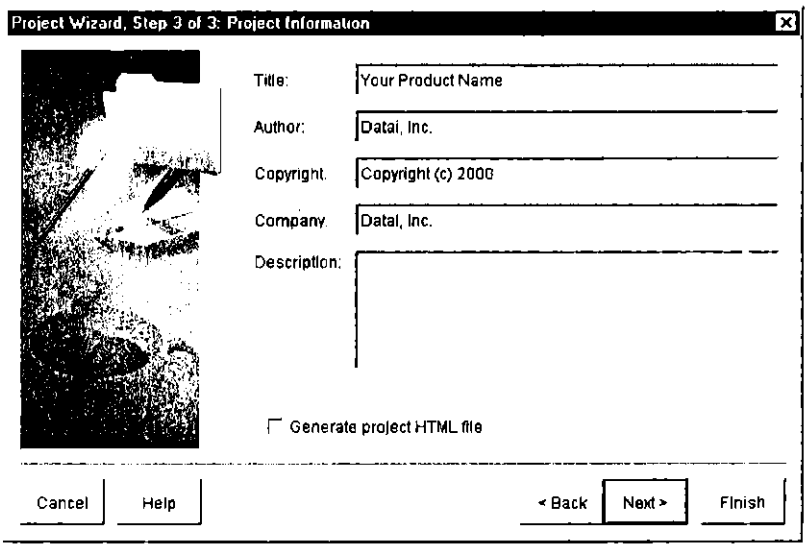
Рис. F.6. Выбор имени пакета и маршрута для файлов



7. Согласитесь со значением по умолчанию, показанным на рис. F.6, и щелкните мышью на кнопке Next. На этапе 3 в окне Project Wizard можно ввести свое имя, название своей компании и другую информацию

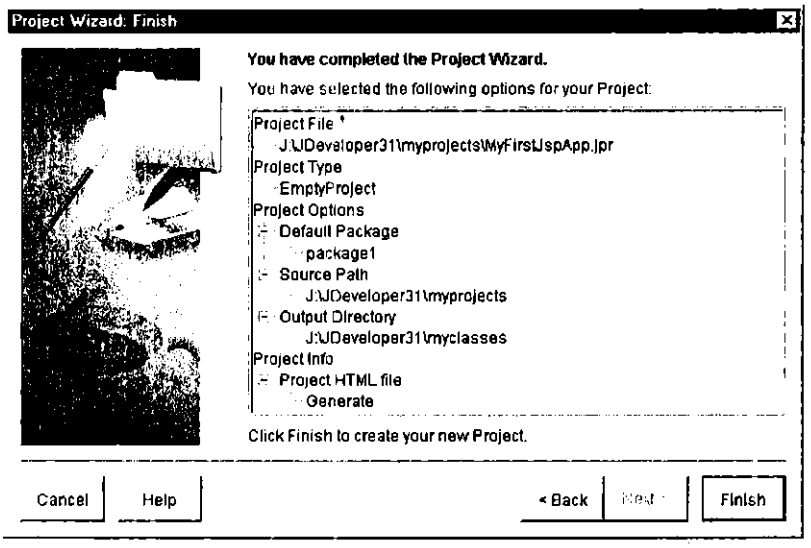
(см. рис. F.7). Можно запросить у JDeveloper генерацию HTML-файла для проекта. Для этого щелкните мышью на Generate project HTML file, а затем на Next.

Рис. F.7. Информация о проекте



8. После щелчка мыши на кнопке Next инструмент JDeveloper выводит список всех параметров, выбранных для проекта (см. рис. F.8).

Рис. F.8. Экран Finish



9. После щелчка мыши на кнопке Finish инструмент JDeveloper возвращается к панели Navigator. Как видим, на рис. F.9 показано то же окно, что и на рис. F.1, за исключением того, что в нем содержится информация, зафиксированная в окне Project Wizard.
10. Переименуйте проект, изменив `Untitled1` на `MyFirstJspApp`. Для этого выделите имя `Untitled1.jws`, щелкните мышью на File в панели инструментов и переименуйте проект. Введите `MyFirstJspApp` и щелкните мышью на Save. JDeveloper автоматически обновит панель Navigator (см. рис. F.10).

Рис. F.9.
Oracle JDeveloper — Untitled1

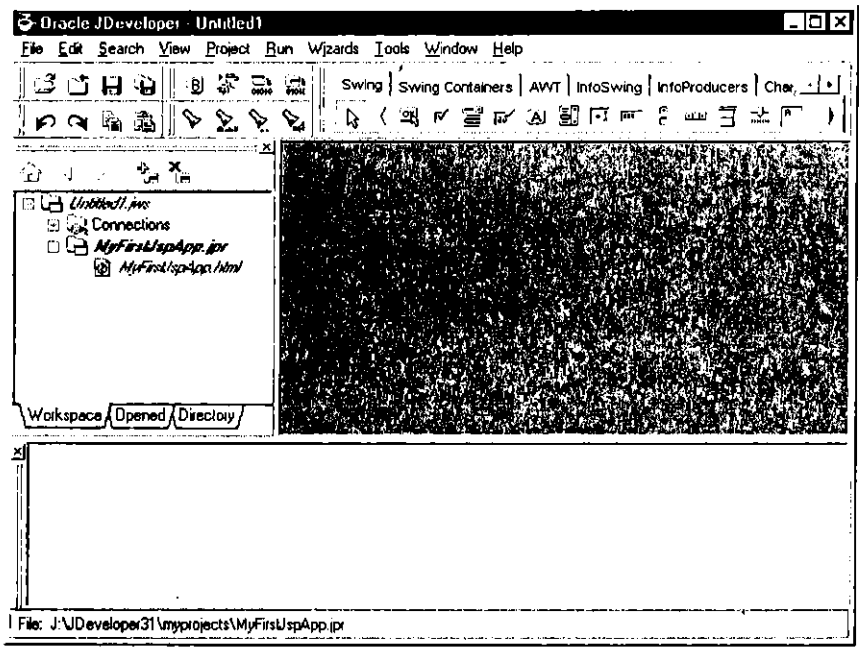
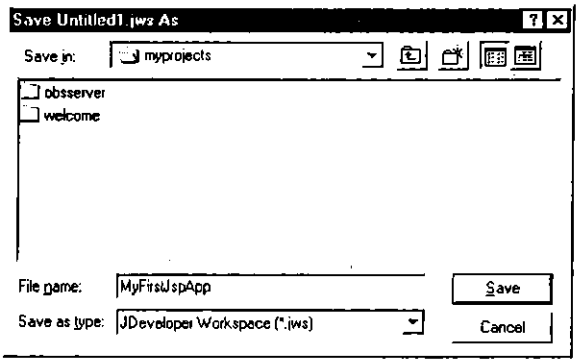


Рис. F.10.
Сохранение Untitled.jws как MyFirstJspApp



11. Теперь все готово к созданию нашего первого JSP-приложения. Щелкните мышью на File, New. Перейдите на вкладку Web Objects. Этот мастер позволит выбрать тип создаваемого web-приложения. В JDeveloper можно создавать приложения, используя "чистый" HTML, JSP, HTTP Servlet, DBServlet, Web Bean и бизнес-компоненты JSP. Создадим простое JSP-приложение. Для этого выделите JSP и щелкните мышью на OK. Появится обновленная панель Navigator. Обратите внимание на то, что инструмент создал файл `untitled1.jsp`, на котором нужно дважды щелкнуть клавишей мыши (см. рис. F.11).

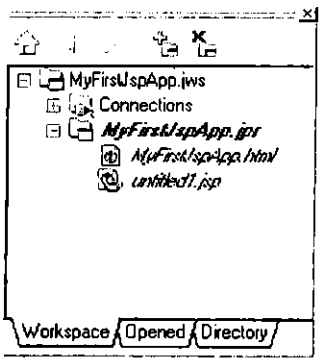
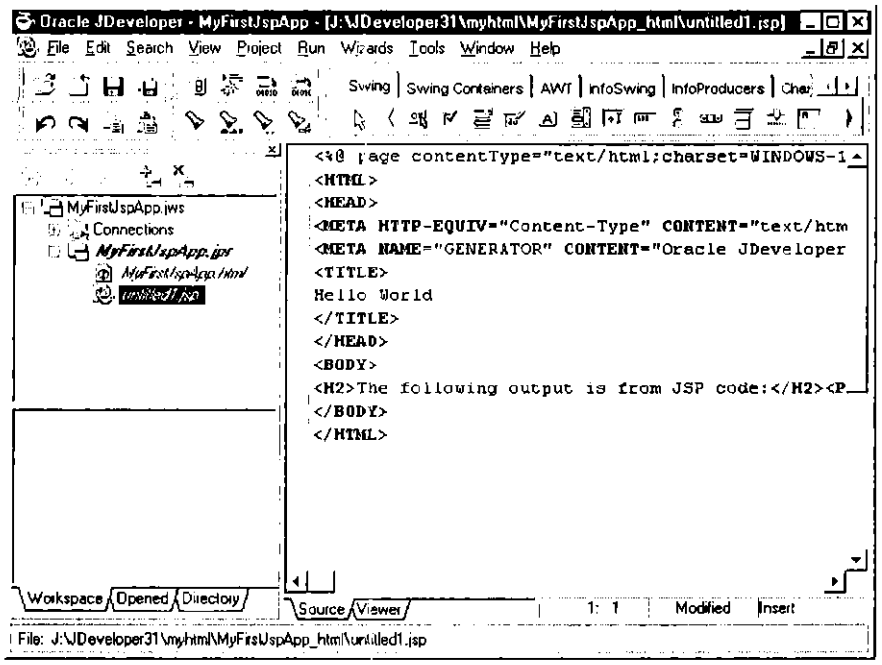


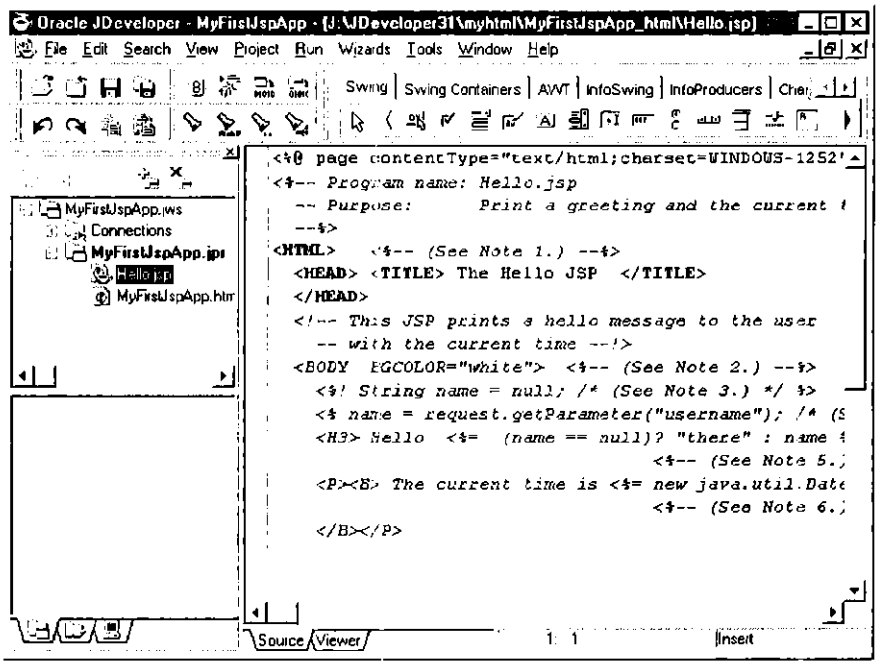
Рис. F.11.
Новый .jsp-файл, создаваемый в Jdeveloper



12. Переименуйте файл `untitled1.jsp`. Для этого выделите `untitled1.jsp`, щелкните мышью на File, а затем переименуйте файл.

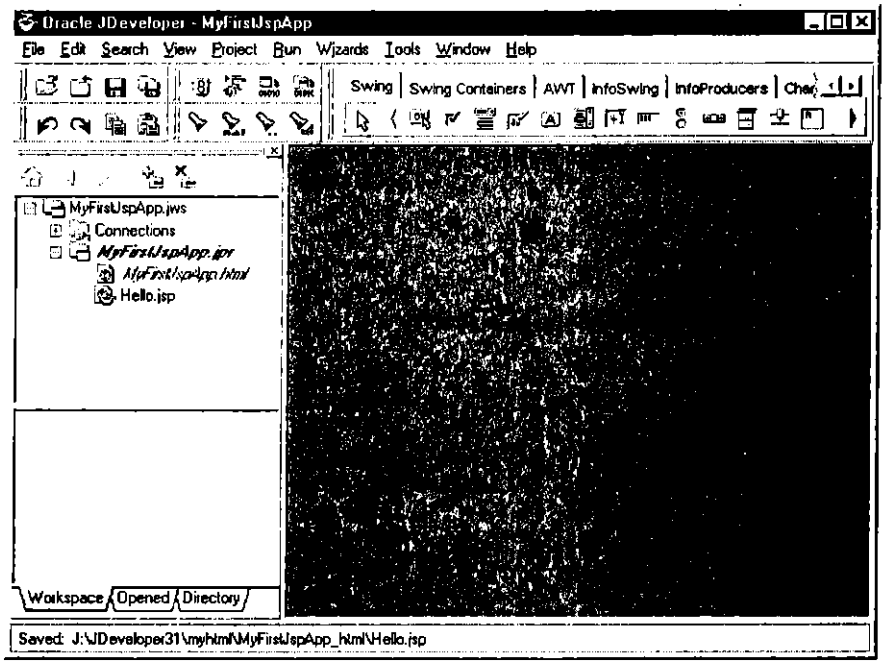
13. Замените программу, сгенерированную инструментом JDeveloper, программой `Hello.jsp` (см. раздел "Написание `Hello.jsp`" главы 10). Обновленный JSP-файл показан на рис. F.12.

Рис. F.12.
Замена JSP-программы, предлагаемой в Jdeveloper по умолчанию



14. Замените `untitled1.jsp` на `Hello.jsp` и щелкните мышью на кнопке Save. Изменения, внесенные инструментом, представлены на рис. F.13.

Рис. F.13.
Панель Navigator с файлом Hello.jsp



15. Выполните JSP-приложение Hello. Щелкните правой кнопкой мыши на программе Hello.jsp и выберите Run:

16. Результат исполнения программы Hello.jsp показан на рис. F.14.

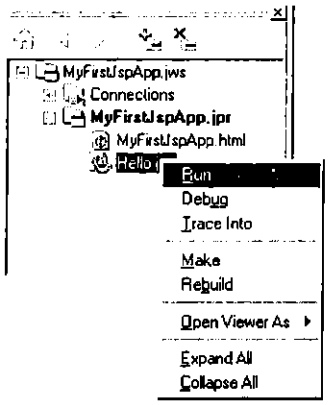
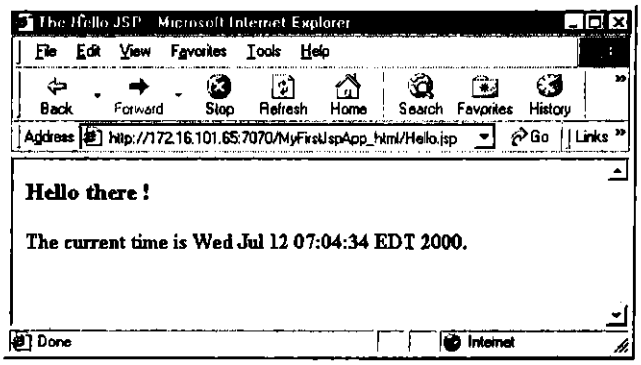


Рис. F.14.
Выходные данные программы Hello.jsp



Библиография

1. Al-Ashmad, W. and Steegmans, E., Inheritance in *Object-Oriented Languages: Requirements and Supporting Mechanisms*, Journal of Object-Oriented Programming (JOOP), January 2000, pp. 15 – 24
2. Ambler, Scott, *Architectural Modeling, Software* Development Online, www.sdmagazine.com/uml/articles/s9910dc1.shtml, October 1999
3. Anderson, Richard, *Professional XML*, Wrox Press, January 2000
4. Avedal, Karl et al, *Professional JSP: Using JavaServer Pages, Servlets, EJB, JNDI, JDBC, XML, XSLT, and WML to Create Dynamic and Customizable Web Content*, Wrox, May 2000
5. Bayram, Zeki, *Business Object-Oriented Analysis and Design(BOOAD) Methodology*, Journal of Object-Oriented Programming (JOOP), March – April 1999, pp. 59 – 67
6. *The Business Benefits of EJB and J2EE Technologies Over COM+ and Windows DNA. A white paper by the Middleware Company, java.sun.com/products/ejb/ejbvscom.html*
7. *CORBA 2.2 Specification*, www.omg.org/corba/cichpter.html#idls&s, February 1998
8. *The Developer's Guide to Understanding Enterprise JavaBeans Applications*, Nova Laboratories, www.nova-labs.com/ejbguide/EJB_Guide_1_1aweb.pdf
9. Douglas, B., *Components: Logical, Physical Models*, Software Development Online, www.sdmagazine.com/uml/beyondobjects/s9912bo1.shtml, December 1999
10. Englander, Robert, *Developing Java Beans*, O'Reilly, June 1997
11. *Enterprise JavaBeans 1.0 Documentation*, java.sun.com/products/ejb/docs10.html
12. *Enterprise JavaBeans 1.1 Documentation*, java.sun.com/products/ejb/docs.html
13. The Esperanto Group, *MTS vs. EJB? Really?*, java.sun.com/products/ejb/pdf/zona.pdf
14. Gosling, James, Joy, Bill, and Steele, Guy, *The Java Language Specification*, Addison-Wesley Publishing Co., September 1996
15. Horton, Ivor, *Beginning Java 2*, Wrox Press, April 1999
16. Horstmann, Cay S. and Cornell, Gary, Core Java 2, Volume 2: Advanced Features, Prentice Hall, December 1999
17. Hunter, Jason and Crawford, William, *Java Servlet Programming*, O'Reilly, November 1998
18. *Java Naming and Directory Interface Application Programming (JNDI API)*, Sun Microsystems, Inc., July 1999, java.sun.com/products/jndi/
19. Java Transaction API (JTA), Sun Microsystems, java.sun.com/products/jta/index.html
20. Java Transaction Service API (JTS), Sun Microsystems, java.sun.com/products/jts/index.html
21. Jörelid, Lennart, *Use JDBC for Industrial-Strength Performance, Part 1*, January 2000, www-4.ibm.com/software/developer/library/jw-jdbc1/index.html
22. Jörelid, Lennart, *JDBC Usage for Industrial-Strength Performance, Part 2*, February 2000, www-4.ibm.com/software/developer/library/jw-jdbc2/index.html
23. JSP FAQ, java.sun.com/products/jsp/faq.html
24. JSP FAQ, www.esperanto.org.nz/jsp/jspfaq.html

25. JSP FAQ, www.jguru.com/jguru/faq/faqpage.jsp?name=JSP
26. JSP FAQ, www.jguru.com/jguru/faq/faqpage.jsp?name=Servlets
27. Kim, Won, et al, *Component-Based Knowledge Engineering Architecture*, Journal of Object-Oriented Programming (JOOP), October 1999, pp. 40 – 48
28. Ladd, Eric and O'Donnel, Jim, *Platinum Edition Using HTML 4*, XML, and Java 1.2, Macmillan USA, December 1998
29. Laitkorpi, Markku and Jaaksi, Ari, *Extending the Object-Oriented Software Process with Component-Oriented Design*, Journal of Object-Oriented Programming (JOOP), March – April 1999, pp. 41 – 50
30. Lemay, Laura and Cadenhead, Rogers, *Sams Teach Yourself Java 2 in 21 Days*, Sams Publishing, 1999
31. Lorentz, Diana and Denise Oertel, *Oracle8i SQL Reference, Release 8.1.5 and 8.1.6*, Oracle Corporation, 2000
32. Maring, Sheryl, *CORBA Developer's Guide, Release 2 (8.1.7)*, Part No. A83755-01
33. Marshall, Chris, *Enterprise Modeling with UML*, Addison-Wesley Publishing Co., October 1999
34. Marshall, Martin, *EJB Flashpoint*, Informationweek Online, March 9, 1999, www.informationweek.com/732/32iuejb2.htm
35. MDE Enterprise Java APIs Team, Deployathon, HYPERLINK developer.java.sun.com/developer/ developer.java.sun.com/developer/ technicalArticles/Programming/deployathon/
36. Monson-Haefel, Richard, *Create forward-compatible beans in EJB, Part 1*, Java World, December 1999, www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-ejb1.html
37. Monson-Haefel, Richard, *Enterprise JavaBeans*, O'Reilly, June 1999
38. Monson-Haefel, Richard, *The Java Naming and Directory Interface (JNDI): A More Open and Flexible Model*, Java Report Online, 1998, www.javareport.com/html/features/archive/9802/haefel.shtml
39. Morgenthal, JP, Understanding *Enterprise Java APIs Why Corporate Java Developers Need to Know About JNDI, JTS, and JMS*, www.componentmag.com/html/from_pages/feature.shtml
40. Morisseau-Leroy, N., Solomon, M., and Momplaisir, G., *Oracle8i SQLJ Programming*, Osborne/McGraw-Hill, November 1999
41. Myers, Tom and Nakhimovsky, Alexander, *Professional Java XML Programming with Servlets and JSP*, Wrox Press, December 1999
42. Patricia Seybold Group, *Enterprise JavaBeans Technology Server Component Model for the Java Platform*, December 1998, java.sun.com/products/ejb/white_paper.html
43. Patzer, Andrew, et al, *Professional Java Server Programming: with Servlets, JavaServer Pages (JSP), XML, Enterprise JavaBeans (EJB), JNDI, CORBA, Jini and Javaspaces*, Wrox, 1999
44. Pawlan, Monica, *Using Session and Entity Beans to Write a Multitiered Application*, java.sun.com/products/ejb/articles/multitier.html
45. Perrone, Paul J. and Chaganti, Krishna, *Java Naming and Directory Service Interfaces*, www.distributedcomputing.com/fullarticle.asp?ID=1119925700PM
46. Pfaeffle, Thomas, *Oracle8i JDBC Developer's Guide and Reference, Release 8.1.5 & 8.1.6*, Oracle Corporation, 1999-2000

47. Portfolio, Tom, *Oracle8i Java Stored Procedures and Developer's Guide, Release 8.1.5 & 8.1.6*, Oracle Corporation, 1999-2000

48. *The Remote Method Invocation Specification*, Sun Microsystems, java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

49. Scallan, T., *Assuring Reliability of Enterprise JavaBean Applications*, Java Developer Journal, January 2000, p. 30

50. Schmid, H.A., *Business Entity Components and Business Process* Components, Journal of Object-Oriented Programming (JOOP), October 1999, pp.6 – 15

51. Seshadri, Govind, *Understanding JavaServer Pages Model 2 Architecture*, December 1999, www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

52. Smith, Tim and Bill Courington, *Enterprise JavaBeans and CORBA Developer's Guide, Release 8.1.5 & 8.1.6*, Oracle Corporation, 1999 – 2000.

53. Sundsted, Todd, *JNDI Overview, Part 1: Introduction to Naming Services*, JavaWorld, January 2000, www.javaworld.com/javaworld/jw-01-2000/jw-01-howto.html

54. Sundsted, Todd, *JNDI Overview, Part 2: Introduction to Directory Services*, JavaWorld, February 2000, www.javaworld.com/javaworld/jw-02-2000/jw-02-howto.html

55. Sundsted, Todd, *JNDI Overview, Part 3: Advanced JNDI*, JavaWorld, March 2000, www.javaworld.com/javaworld/jw-03-2000/jw-03-howto.html

56. Szyperski, Clemens, *Component Software Beyond Object-Oriented Programming*, Addison-Wesley Publishing Co., 1999

57. Tanrikorur, Tulu, *Concepts for Simpler Design*, Software Development Online, www.sdmagazine.com/breakrm/features/s996f2.shtml, June 1999

58. *The Technical Benefits of EJB and J2EE Technologies Over COM+ and Windows DNA*, A white paper by the Middleware Company, java.sun.com/products/ejb/ejbvscom.html

59. Thomas, Anne, *Comparing MTS and EJB*, Patricia Seybold Group, java.sun.com/products/ejb/ejbvscom.html

60. Thomas, Anne, *Enterprise JavaBeans Technology Server Component Model for the Java*, java.sun.com/products/ejb/white_paper.html#ejbcompmod

61. Valesky, Thomas C., *Enterprise Javabeans: Developing Component- Based Distributed Applications*, Addison-Wesley Publishing Co., May 1999

62. Vogel, Andreas and Duddy, Keith, *Java Programming with CORBA, Second Edition*, Wiley Computer Publishing, New York, NY, 1998

63. Wright, Brian, *Oracle8i JavaServer Pages Developer's Guide and Reference Release 8.1.7*, Oracle Corporation, August 2000

64. Wright, Brian, *Oracle8i SQLJ Developer's Guide and Reference Release 8.1.5 & 8.1.6*, A64684-01, Oracle Corporation, 2000