Руководство программиста Enterprise JavaBeans

VERSION 4.0

Inprise Application Server™

Inprise Corporation, 100 Enterprise Way
Scotts Valley, CA 95066-3249

Printed in the U.S.A.

ASE0040WW21000 2E3R1299
9900010203-9 8 7 6 5 4 3 2 1
PDF

Содержание

Глава 5

Написание клиентского приложения 5-1

Глава 6

Написание Session-Компонента 6-1

Глава 7

Написание Entity-Компонента 7-1

Глава 8

Управление транзакциями 8-1

Глава 9

Поставка Компонентов EJB 9-1

Глава 10

Инструментальные средства EJB 10-1

Приложение A

Анализ результатов примера cart A-1

Приложение B

Поддержка EJB 1.0 B-1

Глава

1

Предисловие

Руководство Программиста Enterprise JavaBeans Inprise Application Server содержит информацию, необходимую для разработки Компонентов EJB, написания кода клиентского приложения и поставки Компонентов.

В предисловии приведено краткое содержание Руководства. В нем также описаны соглашения, используемые в тексте документа, и приведены источники дополнительной информации, касающейся EJB и CORBA.

Структура данного руководства

- Глава 2, "Обзор" - содержит вступление в технологию EJB и описание ее архитектуры.
- Глава 3, "Первые шаги" - описывает основные приемы работы с Контейнерами EJB и программными средствами, а также рассказывает, как запустить на выполнение примеры приложений. Подробно рассматривается пример создания простейшего Session-компонента без состояния (stateless).
- Глава 4, "Разработка Компонента EJB" - подробно рассматривает все аспекты разработки Компонента.
- Глава 5, "Написание клиентского приложения" - показано, как написать код приложения, которое является клиентом Контейнера EJB.
- Глава 6, "Создание Session-компонентов" - показано, как писать Session-компоненты, как с состоянием (stateful), так и без (stateless).
- Глава 7, "Создание Entity-компонентов" - показано,как писать Entity-компоненты с сохранением состояния, управляемым как самим Компонентом (Bean-Managed Persistence, BMP), так и его Контейнером (Container-Managed Persistence, CMP).

- Глава 8, "Управление транзакциями" - описывает поддержку транзакций и их взаимодействие с JDBC.
- Глава 9, "Поставка Enterprise JavaBeans" - описывает, как использовать инструменты Inprise EJB для поставки Компонентов.
- Глава 10, "Средства работы с EJB" - описывает инструменты для управления и администрирования Компонентов EJB.
- Приложение A, "Анализ результатов примера cart" - содержит подробный разбор примера cart.
- Приложение B, "Поддержка EJB 1.0" - объясняет, как надо поступать с Дескриптором Поставки в устаревшем формате EJB 1.0.
- "Индекс"

Какие разделы предназначены для вас?

В нижеприведенной таблице описано, для какой аудитории предназначена та или иная глава.

Аудитория	Наиболее интересные разделы
Менеджеры проектов, постановщики задач	Глава 1, "Предисловие" и Глава 2, "Обзор"
Разработчики	Глава 3, "Первые шаги" и последующие главы Руководства

Типографские соглашения

В данном руководстве используются следующие соглашения:

Соглашение	Используется для
boldface	Символизирует, что синтаксическая конструкция должна быть набрана именно так, как показано в тексте. Применительно к UNIX, шрифт используется для имен баз данных, файлов и тому подобного.
italics	Информация, вводимая пользователем или выводимая приложением; также используется для новых терминов или названий книг.
`computer`	Пример кода или командной строки.
UPPERCASE	Используется для SQL-операторов и команд. Применительно к Windows, шрифт используется для имен баз данных, файлов и тому подобного.
[]	Необязательные параметры.
{}	Используются в сложных синтаксических конструкциях для обозначения обязательных параметров.
...	Говорит о том, что строка является продолжением предыдущих строк кода или указывает на возможный повтор предыдущего аргумента
|	Два взаимоисключающих варианта

Соглашения о платформах

В руководстве используются следующие символы для обозначения описаний, зависящих от выбранной платформы:

Windows Все Windows-платформы, включая Windows 3.1, Windows NT и Windows 95

WinNT только Windows NT

Win95 только Windows 95

Win98 только Windows 98

Win2000 только Windows 2000

UNIX Все виды UNIX

Solaris только Solaris

AIX только AIX

HP-UX только HP_UX

IRIX только IRIX

Digital UNIX только Digital UNIX

Где найти дополнительную информацию

Предполагается, что читатель данного руководства знаком с общими принципами построения распределенных систем вообще и со спецификацией Sun Enterprise JavaBeans (EJB) в частности.

Ниже приведен список источников дополнительной информации по EJB и CORBA.

Информация по EJB

Большое количество информации вы можете найти на web-сайте Sun (http://java.sun.com/products/ejb).

Там содержится следующее:

- Спецификация Sun Enterprise JavaBeans 1.1
- Enterprise JavaBeans CORBA Mapping
- Документ ("white paper") с названием "Enterprise JavaBeans Technology Component Model for the Java Platform"
- Наиболее часто задаваемые вопросы по EJB и ответы на них (FAQ)
- Другие документы

С темой EJB связаны также следующие документы:

- JavaBeans. См. http://java.sun.com/beans
- Java Naming and Directory Interface (JNDI). См. http://java.sun.com/products/jndi
- Java Remote Method Invocation (RMI). См. http://java.sun.com/products/jdk/rmi
- Java Security. См. http://java.sun.com/security
- Extensible Markup Language (XML). См. http://java.sun.com/xml

Информация по CORBA

Web-сайт Object Management Group (OMG) (http://www.omg.org) также содержит несколько документов, которые могут быть полезны для вас.

Обратите внимание на следующее:

- Спецификация CORBA 2.3/IIOP
- См. http://www.omg.org/
- Java to IDL Mapping
- См. http://www.omg.org/corba/cichpter.html#mijav
- OMG Object Transaction service
- См. http://www.omg.org/corba/sectrans.html
- ORB Portability Submission
- См. http://www.omg.org/library/c2indx.html

Поддержка пользователей продуктов Inprise

Inprise предлагает различные виды поддержки пользователей. Вы можете использовать Internet для доступа к нашим базам пользовательской информации и для общения с другими пользователями продуктов Inprise. В дополнение к этому, вы можете выбрать несколько видов технической поддержки по телефону - от помощи при инсталляции продуктов до платных консультаций по интересующим вас конкретным вопросам.

Дополнительную информацию о службе поддержки пользователей вы можете найти на нашем web-сайте по адресу http://www.borland.com/devsupport. Телефон службы поддержки: 800-523-7070; телефон отдела продаж: 800-632-2864. Пользователи, находящиеся вне территории США, могут найти интересующую их информацию на нашем web-сайте (http://www.borland.com/bww/intlcust.html).

Пользователи из России, стран СНГ и государств Балтии могут обращаться в Московское представительство Inprise по телефону: +7 (095) 238-3611. Сайт Московского представительства Inprise: http://www.inprise.ru или http://www.borland.ru .

Перед обращением к службе поддержки, пожалуйста, подготовьте всю необходимую информацию о вашей операционной системе, версии используемого продукта, а также детальное описание возникшей проблемы.

Дополнительную информацию по вопросам, имеющим отношения к проблеме 2000 года, вы можете найти по адресу http://www.borland.com/devsupport/y2000.

О терминологии

Терминология, используемая в данной документации, базируется на устоявшихся терминах и словосочетаниях, используемых в русскоязычной компьютерной прессе. Некоторые термины являются взаимозаменяемыми, например, *поставка* и *развертывание* - от оригинального *deployment*.

Глава

2

Обзор

Глава содержит следующие основные темы:

- "Архитектура Enterprise JavaBeans" описывает структуру технологии EJB.
- "Зачем использовать реализацию Inprise Контейнера EJB?"

Архитектура Enterprise JavaBeans

Enterprise JavaBeans - это высокоуровневая, базирующаяся на использовании компонентов технология создания распределенных приложений, которая использует низкоуровневый API для управления транзакциями. EJB существенно упрощает разработку, поставку и настройку систем уровня предприятия, написанных на языке Java. Архитектура EJB определена в спецификации, разработанной Sun Microsystems. Реализации Контейнера EJB фирмой Inprise базируется на версии EJB 1.1.

Технология Enterprise JavaBeans определяет некоторый набор универсальных и предназначенных для многократного использования компонентов, которые называются enterprise beans (в русском переводе Руководства - Компоненты EJB). При создании распределенных системы ее бизнес-логика реализована на уровне этих Компонентов.

После завершения их кодирования, наборы Компонентов EJB помещаются в специальные файлы, по одному или более Компонентов на файл, вместе со специальными параметрами Поставки (deployment). Наконец, эти наборы компонентов устанавливаются в операционной среде, в которой запускается Контейнер EJB. Клиент создает компоненты и осуществляет их поиск в Контейнере с помощью так называемого home-интерфейса Компонента. После того, как Компонент создан и/или найден, клиент выполняет обращения к его бизнес-методам с помощью так называемого remote-интерфейса.

Контейнеры EJB выполняются под управлением Сервера EJB, который выполняет роль связующего звена между Контейнерами и операционной средой. Сервер EJB обеспечивает доступ Контейнерам EJB к системным сервисам, таким, как управление доступом к базам данных или мониторам транзакций, а также к другим приложениям.

Все экземпляры Компонентов EJB выполняются под управлением Контейнера EJB. Контейнер использует системные сервисы в интересах "своих" Компонентов и управляет их жизненным циклом. Вследствие того, что Контейнер берет на себя выполнение большинства задач системного уровня, разработчик отдельного Компонента не должен включать в код бизнес-методов Компонента ничего, что предназначено для выполнения на уровне Контейнера или Сервера. В общем случае, Контейнер предназначен для решения следующих задач:

- Обеспечение безопасности - Дескриптор Поставки (deployment descriptor) определяет права доступа клиентов к бизнес-методам Компонентов. Обеспечение защиты данных обеспечивается за счет предоставления доступа только для авторизованных клиентов и только к разрешенным методам.
- Обеспечение удаленных вызовов - Контейнер берет на себя все низкоуровневые вопросы обеспечения взаимодействия и организации удаленных вызовов, полностью скрывая все детали как от разработчика Компонентов, так и от клиентов, которые пишут код точно так же, как если бы система работала в локальной конфигурации, т.е. без использования удаленных вызовов вообще.
- Управление циклом жизни - Клиент просто создает и (обычно) уничтожает экземпляры Компонентов. Тем не менее, Контейнер для оптимизации ресурсов и повышения производительности системы может самостоятельно выполнять различные действия, например, активизацию и деактивизацию этих Компонентов, создание их пулов и т.д.
- Управление транзакциями - Все параметры, необходимые для управления транзакциями, помещаются в Дескриптор Поставки. Все вопросы по обеспечению управления распределенными транзакциями в гетерогенных средах и взаимодействия с несколькими базами данных берет на себя Контейнер EJB. Контейнер обеспечивает защиту данных и гарантирует успешное подтверждение внесенных изменений; в противном случае транзакция откатывается.

Компоненты Enterprise bean

Enterprise beans являются элементами распределенных транзакционных приложений уровня предприятия. Следующие характеристики являются общими для всех Компонентов EJB:

- Компоненты EJB реализуют бизнес-логику приложения как набор операций над корпоративными данными.

- Разработчик Компонентов EJB создает Компонент так, как его видит клиент, и такой подход никак не зависит от вопросов взаимодействия Компонента с его Контейнером и с Сервером. Это позволяет осуществлять создание приложений из Компонентов EJB и их настройку без необходимости изменения и перекомпиляции исходного кода Компонентов.
- Контейнер создает экземпляры Компонентов и управляет ими во время работы приложения. Контейнер также управляет доступом клиентов к Компонентам.
- Настройка Компонентов осуществляется на этапе их поставки путем изменения их свойств, определяющих поведение компонента в конкретной среде (environment properties).
- Различные системные характеристики, такие, как атрибуты безопасности или транзакций, не являются частью класса Компонента и, следовательно, управляются с помощью специальных программных средств на этапе поставки.

Существуют два типа компонентов EJB: Session- и Entity-Компоненты.

Session-компоненты

Session-компоненты представляет собой объект, созданный для обслуживания запросов одного клиента. В ответ на удаленный запрос клиента, Контейнер создает экземпляр Компонента. Session-компонент всегда сопоставлен с одним клиентом; можно рассматривать его как представителя клиента на стороне EJB-сервера. Такие Компоненты могут "знать" о наличии транзакций - они могут отвечать за изменение информации в базах данных, но сами они непосредственно не связаны с представлением данных в БД.

Session-компоненты являются временными объектами и существуют сравнительно недолго. Обычно Session-компонент существует, пока создавший его клиент поддерживает с ним "сеанс связи". После завершения связи с клиентом компонент уже никак не сопоставлен с ним. Объект считается временным, так как в случае завершения работы (или "падения") сервера клиент должен будет создать новый Компонент.

Обычно Session-компонент содержит параметры, которые характеризуют состояние его взаимодействия с клиентом, т.е. он сохраняет некоторое состояние между вызовами удаленных методов в процессе сеанса связи с клиентом. Session-компоненты, которые поддерживают состояние, называются stateful-компонентами. Состояние существует, пока существует сеанс взаимодействия с клиентом.

Session-компонент может также не иметь состояния (stateless-компонент). Он не хранит характеристик своего взаимодействия с клиентом. Когда клиент вызывает один из его методов, разумеется, происходит изменение значений некоторых внутренних переменных, но эти значения имеют смысл только во время обработки этого

единственного вызова - до его завершения. Таким образом, все экземпляры одного stateless-компонента являются идентичными (кроме интервалов времени, когда выполняется код одного из методов). Вследствие этого, ничто не мешает одному и тому же экземпляру компонента обслуживать вызовы различных клиентов. Контейнер EJB может создать пул экземпляров таких Компонентов и выбирать любой из них для обслуживания клиентских запросов.

Entity-компоненты

Entity-компоненты представляют собой объектное представление данных из БД. Например, Entity-компонент может моделировать одну запись из таблицы реляционной базы данных. Несколько клиентов могут одновременно обращаться к одному экземпляру такого Компонента. Entity-компоненты изменяют состояние сопоставленных с ними баз данных в контексте транзакций.

Состояние Entity-компонентов в общем случае нужно сохранять, и "живут" они столько, сколько существуют в базе данных те данные, которые они представляют, а не столько, сколько существуют клиентский или серверный процессы. Остановка или "падение" Контейнера EJB не приводит к уничтожению содержащихся в нем Entity-Компонентов.

За сохранность компонента может отвечать сам Компонент (Bean Managed Persistence, BMP) или его Контейнер (Container Managed Persistence, CMP). При использовании CMP все обязанности по сохранению состояния Компонента возлагаются на Контейнер. В случае BMP, вы должны написать для Компонента нужный код, включая обращение к базам данных.

Каждый Entity-компонент характеризуется своим уникальным идентификатором - primary key. Обычно это тот же самый primary key, что и идентификатор данных в БД, например, совокупность ключевых полей записи в твблице. Primary key используется для поиска нужного компонента.

Роли EJB

Архитектура EJB позволяет упростить процесс создания сложных систем разбиением его на шесть отдельных этапов, с каждым из которых сопоставлены свои задачи и, соответственно, обязанности (роли) исполнителей. Эти роли можно разбить на следующие три группы: инфраструктура, собственно разработка приложений и их поставка и настройка.

Главная задача такой структуры - облегчение жизни разработчиков конкретных приложений. Разработчики EJB-Контейнеров и Серверов берут на себя решение многих проблем - в первую очередь, написание системных и платформенно-зависимых сервисов - элементы которых в противном случае пришлось бы писать прикладному программисту.

Роли обеспечения инфраструктуры

EJB Server Provider: обычно это разработчик или фирма, имеющие большой опыт создания инфраструктуры распределенных систем и системных сервисов. Server Provider создает как платформу для разработки распределенного приложения, так и run-time среду для его выполнения.

EJB Container Provider: как правило, это эксперт в области технологий создания распределенных систем, управления транзакциями и обеспечения безопасности. Container Provider предоставляет средства для поставки Компонентов EJB и run-time среду для установленных экземпляров Компонентов.

Контейнер поддерживает один или несколько Компонентов. Он является посредником между Компонентами и Сервером EJB, т.е. средствами управления сетевым взаимодействием, транзакциями и безопасностью. Контейнер является одновременно готовой программой и средством для генерации кода в интересах конкретного Компонента EJB. Кроме того, Контейнер обеспечивает инструменты для поставки Компонентов и средства для мониторинга и управления приложениями.

Роли разработки приложений

Enterprise Bean Provider: обычно это специалист в некоторой прикладной области; например, эксперт в области финансов или телекоммуникаций. Bean Provider определяет remote- и home-интерфейсы Компонента, реализует его бизнес-методы, а также создает Дескриптор Поставки (Deployment Descriptor) Компонента. Он не принимает во внимание такие вопросы, как обеспечение удаленного взаимодействия, управление транзакциями и безопасностью и прочие не имеющие к бизнес-логике конкретного Компонента аспекты, так как эти проблемы должен решать Container Provider.

Application Assembler: это эксперт в прикладной области, который выполняет "сборку" приложения из готовых блоков - т.е. Компонентов EJB - и добавляет некоторые другие элементы, например, апплеты или сервлеты для создания готового приложения. В процессе своей работы он имеет дело с интерфейсами Компонентов EJB (а не с кодом его бизнес-методов), а также с Дескриптором Поставки. Результатом его работы может быть как готовое приложение, так и более сложный составной Компонент EJB.

Роли поставки и настройки

Deployer: его задача - настроить приложение, собранное из Компонентов EJB, для его выполнения к конкретной операционной среде. Достигается это путем изменения свойств Компонента. Например, deployer определяет поведение транзакций или профили безопасности путем установки тех или иных значений для свойств, находящихся в Дескрипторе Поставки. Его задачей также является

интеграция приложения с существующими программными средствами управления корпоративной системой.

System Administrator работает с поставленным приложением. Его задача - задание конфигурации и администрирование информационной инфраструктурой предприятия, включая EJB-Сервера и Контейнеры. Administrator отслеживает поведение системы во время ее работы и предпринимает необходимые меры в случае, если что-то происходит не так, как надо. Обычно при этом используются корпоративные средства контроля, с которыми (посредством специальных средств на уровне Контейнера) интегрировано конкретное приложение.

В соответствие с такой схемой, "традиционный" прикладной программист - это, как правило, bean provider и, возможно, application assembler. Эти роли позволяют такому разработчику сфокусировать свое внимание на разработке и реализации бизнес-логики системы. Deployer определяет определяет значения параметров поставки в процессе установки Компонента. Сложные вопросы реализации требований поставки берет на себя специально подготовленный поставщик программных средств. Хотя процесс создания распределенных приложений остается достаточно сложным, существенно облегчается работа "обычного" прикладного программиста - за счет делегирования многих вопросов разработчикам EJB-Серверов и Контейнеров.

Спецификация EJB достигает выполнения всех поставленных задач с помощью введения некоторого числа стандартных конструкций и соглашений. Это ограничивает свободу выбора той или иной архитектуры приложения, но позволяет разработчикам Контейнеров и Серверов делать определенные предположения о дизайне программ и эффективно управлять ими.

Структурные шаблоны (design patterns) и соглашения об именах в EJB

Существуют три основных подхода для создания объектно-ориентированных многозвенных распределенных систем: с сервером без состояния, ориентированный на сеансы связи с клиентом и с использованием "долгоживущих" объектов.

Сервером без состояния называется объект, который обеспечивает нужную функциональность путем предоставления набора операций, но не сохраняющий состояния между вызовами этих операций. При обращении к такому объекту клиент не может полагаться на использование результата предыдущего вызова того же объекта.

Подход, ориентированный на использование сеансов, предполагает создание в некотором промежуточном звене системы специального объекта - "сеанса" (или "сессии"), который является представителем клиента. Обычно время существования такого объекта явно определяется клиентом или серверным процессом, в котором существует данный объект. Клиент уничтожает объект, если он ему

больше не нужен; сервер может уничтожить его, например, по тайм-ауту. Если серверный процесс завершен, имеющаяся у клиента ссылка на объект может потерять свой смысл.

Модель системы с сохраняемыми объектами предполагает создание некоей программной оболочки вокруг данных, хранящихся в некоторой БД, и создание набора операций для работы с этими данными. Созданный таким образом объект доступен одновременно для нескольких клиентов. Его время существования определяется временем существования в хранилище данных, объектным представлением которых он является.

В терминах спецификации EJB, объекты трех вышеупомянутых видов называются stateless session bean, stateful session bean и entity bean. С помощью Session-компонентов моделируется сеансо-ориентированный подход. Entity-компоненты используются в случае долгоживущих, сохраняемых серверных объектов. Для каждого из видов компонентов определены интерфейсы и соглашения, о чем подробно будет говориться в 4-ой главе.

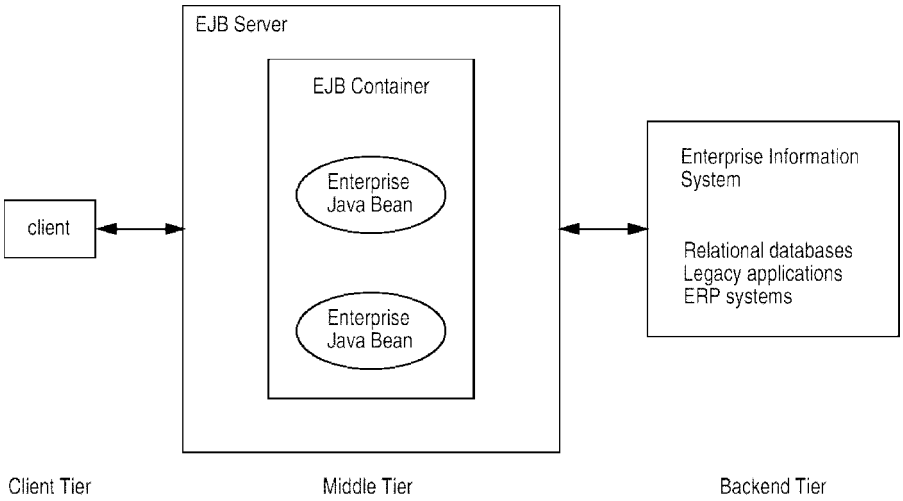
Последующие главы содержат соответствующие примеры и объяснения, как и когда использовать тот или иной вид компонентов.

Инфраструктура Enterprise JavaBean

Создатели Серверов и Контейнеров EJB реализуют инфраструктуру EJB. Инфраструктура обеспечивает удаленное взаимодействие объектов, управление транзакциями и безопасность приложения. Спецификация EJB оговаривает требования к элементам инфраструктуры и определяет Java Application Programming Interface (API); она не касается вопросов выбора платформ, протоколов и других аспектов, связанных с реализацией.

На Рис. 2.1 показаны различные элементы инфраструктуры EJB. Он должна обеспечивать канал связи с клиентом и другими Компонентами EJB. Хотя спецификация этого не требует, желательно, чтобы этот канал обеспечивал безопасность передаваемых данных, особенно при работе в Internet. Инфраструктура должна также обеспечить соблюдение прав доступа к компонентам EJB.

Рис. 2.1 Компоненты, Контейнеры и Сервера EJB.



В общем случае необходимо гарантировать сохранение состояния Компонентов в Контейнерах. Инфраструктура EJB обязана предоставить возможности для интеграции приложения с существующими системами и приложениями - без этого нельзя говорить о пригодности приложения для функционирования в корпоративной информационной сети. Все аспекты взаимодействия клиентов с серверными Компонентами должны происходить в контексте транзакций, управление которыми возлагается на инфраструктуру EJB. Для успешного выполнения процесса поставки Компонентов инфраструктура EJB должна обеспечить возможность взаимодействия со средствами управления приложениями (hooks).

Контейнер

Контейнер, возможно, является наиболее важной концепцией архитектуры EJB - именно он в наибольшей степени облегчает работу разработчика. Такие объектные технологии, как CORBA или RMI, освобождают прикладного программиста от кодирования деталей удаленного взаимодействия - поиска объектов, упаковки и пересылки данных (маршалинга) и т.п. Контейнер делает следующий шаг в направлении упрощения работы - он берет на себя такие нетривиальные аспекты создания распределенных систем, как управление транзакциями, обеспечение безопасности и сохранение состояния объектов.

После того, как компонент EJB подготовлен для поставки, он помещается в стандартный архивный файл Java - файл ejb-jar. Этот файл может содержать один или несколько Компонентов EJB. Он содержит интерфейсы, классы и Дескриптор Поставки для каждого Компонента.

Enterprise Bean Provider должен обеспечить для каждого Компонента следующее:

- Remote-интерфейс Компонента, который содержит доступные для клиента бизнес-методы.
- Home-интерфейс Компонента, который позволяет клиенту создавать и находить Компоненты EJB.
- Собственно класс Компонента, который содержит реализацию функций Компонента.
- Дескриптор Поставки. Deployer использует этот дескриптор на стадии поставки для работы с информацией, которая не является частью кода классов Компонента, но определяет его поведение во время работы. В спецификации EJB 1.0 дескриптор представляет собой результат стандартной Java-сериализации экземпляра класса `javax.ejb.deployment.SessionDescriptor` или `javax.ejb.deployment.EntityDescriptor`. Этот подход в спецификации EJB 1.1 объявлен устаревшим; вместо этого используется дескриптор в формате XML. Подробно о дескрипторе говорится несколько позже в этой главе.

Под процессом "поставки" Компонента понимается его установка его ejb-jar-файла в Контейнер EJB. Процесс поставки включает в себя:

- Проверку, что все составные части Компонента соответствуют друг другу.
- Регистрацию Компонента в Naming Service.
- Обеспечение доступа к Компоненту через коммуникационную систему Сервера EJB.
- Реализацию управления транзакциями и отслеживание профилей безопасности.

В Контейнере может быть установлено любое число Компонентов EJB. Контейнер EJB предоставляет как среду выполнения для своих Компонентов, так и инструменты для выполнения процесса поставки. Эти инструменты для Inprise EJB Container рассматриваются в 10-ой главе.

Зачем использовать реализацию Inprise Контейнера EJB?

Реализация Контейнера EJB фирмой Inprise характеризуется:

- Полной поддержкой спецификации EJB 1.1.
- Гибкостью и надежностью Контейнера как среды выполнения и управления Компонентами.
- Возможностью взаимодействия со службой имен, сервисом транзакций и встроенными Java-RDBMS в процессе разработки и поставки.

- Наличием примеров использования Компонентов и Контейнеров EJB.

Inprise EJB-Контейнер является прекрасным инструментом разработчика Компонентов EJB для их использования в корпоративных системах. Его достоинства:

Полнофункциональная и гибкая EJB run-time среда

- EJB-Контейнер реализует все требования спецификации EJB 1.1, включая необязательные.
- Каждый Компонент EJB, создаваемый с помощью предоставляемых инструментов разработки, одновременно является CORBA-объектом.
- Контейнер EJB может поставляться и как независимый (stand-alone), полностью написанный на Java сервис, и как часть интегрированной распределенной системы. Это означает, что вы можете управлять масштабируемостью и доступностью вашего приложения в соответствие с предъявляемыми к нему требованиями.
- В отличие от других EJB-Серверов, Inpise EJB Container Server не накладывает никаких ограничений на структуру системы с точки зрения выбора числа Контейнеров и распределения Компонентов между ними - на любом количестве компьютеров в сети могут выполняться любое число Контейнеров, содержащих любое число Компонентов. Это обеспечивается распределенным протоколом управления транзакциями. Такая гибкость позволяет выбрать в процессе поставки приложения наиболее оптимальную с точки зрения производительности системы в целом конфигурацию.

Контейнер EJB Inprise построен поверх VisiBroker и RMI-IIOP

- EJB-Контейнер построен на базе Inprise Visibroker. VisiBroker является лидирующим ORB на рынке аналогичных программных средств; он доказал свое превосходство как во внутренних тестах, используемых фирмами-разработчиками программного обеспечения, так и в процессе реальной работы, что еще более важно. VisiBroker предоставляет широкий спектр возможностей первостепенной важности.
- VisiBroker для достижения максимальной масштабируемости приложений использует мультиплексирование соединений с клиентами, создание и управление пулом соединений, поддержку пула потоков (threads).
- Взаимодействия между клиентами и Компонентами EJB, между различными Компонентами и между Компонентами и другими CORBA-объектами базируются на использовании протокола IIOP. VisiBroker полностью соответствует спецификации CORBA 2.3, которая требует реализации RMI-IIOP в терминах передачи объектов по значению - для достижения максимальной совместимости. Это

означает, что типы данных Java (такие, как словари, векторы и др.) могут передаваться при работе с IIOP посредством новых IDL-типов - "value", в полном соответствии со спецификацией CORBA 2.3. Разумеется, и клиенты, и сервера "знают", как передаются такие данные. Inprise EJB Container Server полностью совместим с любым сервером, поддерживающим RMI-IIOP.

- VisiBroker позволяет автоматически распространять права доступа (security credentials). Это гарантирует, что права клиента будут правильно переданы на сервер.
- VisiBroker автоматически распространяет контекст транзакций. Это означает, что когда клиент инициализирует транзакцию, а затем выполняет обращение к Компонентам EJB в Контейнере, осуществляется передача контекста транзакции, который и используется сервером при доступе к тем или иным ресурсам.
- Inprise Integrated Transaction Service (ITS) использует двухфазную схему подтверждения транзакций. Та же двухфазная схема используется, если ее поддерживает JDBC-драйвер. Если используемый вами JDBC-драйвер ее не поддерживает, используется обычное однофазное подтверждение.

Inprise EJB Container - это объект CORBA

Компилятор java2iiop, разработанный Inpise, так же, как и Контейнер EJB, полностью совместим с CORBA. Контейнер EJB "понимает" вызовы RMI, на которых построена технология EJB, но внутри себя для хранения объявлений интерфейсов он использует IDL-версии интерфейсов Java. Хотя компилятор java2iiop может генерировать стабы и скелеты CORBA непосредственно на основе Java-объявлений, вы можете сгенерировать IDL-декларации - например, для работы с другими языками программирования. CORBA-клиенты могут рассматривать EJB-Контейнер как CORBA-сервер. Средства, предоставляемые EJB-Контейнером, являются CORBA-средствами, которые могут быть использованы при работе с EJB.

Inprise-реализация Контейнера EJB базируется также на JNDI, который, в свою очередь, базируется на CORBA CosNaming, и JTS/OTS. Все вместе это обеспечивает полную совместимость с CORBA.

Поддержка для различных видов Компонентов

Важной характеристикой Inprise-реализации Контейнера EJB является поддержка всех видов Компонентов EJB, включая:

- Stateless и stateful Session-компоненты.
- Entity-Компоненты с обоими видами обеспечения хранения состояния.

Контейнер EJB может содержать один или более jar-файлов. Каждый файл может содержать несколько Компонентов. Каждый Компонент имеет Дескриптор Поставки, интерфейсы `EJBHome` и `EJBRemote`, а также код реализации методов этих интерфейсов.

Отличительной особенностью Inprise-реализации Контейнера является возможность не снижать эффективности работы при возрастании числа одновременно обслуживаемых клиентов (т.е. масштабируемость). Масштабируемость обеспечивается, главным образом, следующими двумя особенностями:

- VisiBroker обеспечивает управление соединениями с клиентами. Схема управления соединениями VisiBroker'а позволяет поддерживать одновременно больше клиентов, чем существует TCP-соединений.
- Контейнер не имеет состояния с точки зрения взаимодействия с клиентом. Это означает, что сервер не обязательно выделяет память для каждого нового Session-компонента. Таким образом, сервер может поддерживать произвольное число Session-компонентов без состояния.

Поддержка поставки

Inprise Container включает в себя компиляторы для автоматической генерации кода. Помимо этого, он предоставляет средства для проверки правильности структуры Компонента EJB перед выполнением его поставки. Полностью поддерживаются соответствующие спецификации EJB 1.1 служба JNDI и Дескриптор Поставки в XML-формате.

Менеджер транзакций

Менеджер транзакций обеспечивает управление транзакциями, включая поддержку двухфазного подтверждения для распределенных транзакций. Сервис полностью совместим с последней версией спецификации Sun Java Transaction Service (JTS), а также со спецификацией OTS OMG. Вы можете использовать более "облегченный" JTS в процессе разработки или более мощный и гибкий Transaction Service (ITS) при поставке вашего приложения.

Пул JDBC-соединений и интеграция с транзакциями

Все виды доступа к базам данных с использованием JDBC управляются с помощью Inprise-реализации объекта DataSource. Этот объект прозрачным для разработчика образом управляет пулом соединений с Базами Данных и ассоциирует глобальные транзакции JTS с JDBC-транзакциями для баз данных.

Служба имен (Naming service)

Naming Service обеспечивает поддержку произвольных имен в распределенных средах. Этот сервис полностью совместим как с Sun Java Naming and Directory Interface (JNDI), так и со спецификацией OMG CosNaming. Как и в случае с Сервисом Транзакций, вы можете использовать более "облегченный" JNS в процессе разработки или более мощный и гибкий Inprise Naming Service при поставке вашего приложения.

Поставляемый с EJB-Контейнером JNDI может быть реализован поверх другого JNDI, что позволяет использовать различные службы имен, например, LDAP.

Обеспечение безопасности

Планируется интеграция с Inprise Security Server, который базируется на соответствующей спецификации CORBA.

Базы Данных на Java

Java-БД представляют собой высокопроизводительные хранилища для долговременного хранения состояний Компонентов EJB. Это реляционные базы данных, написанные на Java.

Java-БД могут находится в адресном пространстве того же процесса, в котором исполняется Контейнер EJB, или, с целью повышения производительности, выполняется как отдельный процесс. Фактически, Java-БД являются взаимозаменяемыми, что означает, что конечный пользователь может заменить одну реализацию другой.

Container-managed persistence (CMP) для Entity-Компонентов

Inprise предоставляет встроенную Службу поддержки CMP (CMP engine), которая обеспечивает прозрачное отображение между объектами и реляционными БД и среду обеспечения сохранения состояний компонентов, базирующиеся на стандарте JDBC. Более того, возможно подключение инструментов, разработанных другими производителями, путем использования открытого API.

Интеграция с другими элементами

В реальных системах, модули EJB-Контейнеров взаимодействуют со следующими элементами:

- Клиентами, установившими связь с Компонентами EJB. Это могут быть как Java-клиенты, использующие RMI и Java-интерфейсы

Компонентов, так и CORBA-клиенты, использующие IDL-версии интерфейсов. Для Inprise EJB Container Server, каждый компонент EJB является одновременно и CORBA-объектом. CORBA-клиент может быть реализован на любом языке, поддерживающем CORBA - C++, Java, Delphi.

- Базами Данных, к которым обращаются Компоненты. Как правило, это Entity-компоненты как с CMP, так и с BMP.
- Конечными сервисами, в том числе: CORBA-серверами (реализованными на Java, C++, Delphi и др., взаимодействующими с любым CORBA-совместимым ORB), другими EJB-серверами (как от Inprise, так и от других производителей), другими сервисами (ERP-системами, серверами приложений на мэйнфреймах и т.д.).

Глава

3

Первые шаги

Глава состоит из трех основных разделов:

- Начало. Изучение примеров
- Основы Контейнера EJB и инструментальных средств
- Пример создания приложения с stateless Session-компонентом

Начало. Изучение примеров

Контейнер EJB поставляется с наборов примеров, которые иллюстрируют использование большинства из предоставляемых возможностей. Перед использованием примеров их необхолдимо построить.

Обзор примеров

В комплект поставки входят следующие примеры:

- sort - stateless Session-компонент, который реализует алгоритм "сортировки слиянием". Пример демонстрирует, каким образом нужно откомпилировать текст и запустить Компонент в Контейнере. Пример также демонстрирует работу с RMI-IIOP и взаимодействие со службой имен JNDI.
- cart - stateful Session-компонент, который реализует магазинный чек. Этот пример демонстрирует управление транзакциями с помощью Дескриптора Поставки. Этот пример также показывает, что Inprise EJB-Контейнер обеспечивает обратную совместимость с форматом дескриптора EJB 1.0. Для преобразования Дескриптора EJB 1.0 в XML-формат EJB 1.1 используется утилита dd2xml.

- pigs - два Entity-компонента с CMP, которые связаны друг с другом соотношениями один-к-одному и один-ко-многим. Используются два типа компонентов - Container и Contained. Containers могут содержать любое число Contained-компонентов. Каждый Contained-компонент может содержаться не более чем в одном Container'е.
- bank - Entity-компоненты и один stateless Session-компонент. Entity-компоненты реализуют интерфейс "счет в банке"; один из них использует CMP, другой - BMP. Session-компонент представляет собой банковского оператора, который позволяет клиенту переводить средства с одного счета на другой.
- data - Это пример с использованием Entity-компонента, который показывает все типы данных, поддерживаемых службой CMP. В примере используется компонент, который содержит по одному полю каждого из поддерживаемых CMP типов. Каждая операция выполняется в контексте отдельной транзакции; содержание операции состоит в чтении значения поля, записи в него нового значения с последующим чтением и проверкой корректности выполненного изменения данных. Для демонстрации всех возможностей, установите флаг EJBDebug и отслеживайте изменения в состоянии базы данных. Обратите внимание, что транзакция в режиме "только чтение" не изменяет данные, а другие транзакции вносят только минимально необходимые изменения; в нашем примере происходит изменение значения только одного поля при выполнении каждой операции.
- custom_cmp - Пример с использованием CMP Entity-компонента, который иллюстрирует использование CORBA-сервера для выполнения сохранения состояния компонента. В этом примере Дескриптор Поставки используется для организации взаимодействия со специальной реализацией CMP Службы Сохранения Состояния EJB-Контейнера. Идея состоит в том, что этот новый Менеджер CMP передает вызовы некоторому CORBA-объекту, который рассматривается как транзакционное хранилище данных для данных Компонента EJB.

Инфо Советуем начать с примеров sort и cart. Их проще построить и запустить, и они не требуют наличия внешних баз данных.

Инфо Мы постоянно добавляем новые примеры и улучшаем существующие. Самая свежая информация находится в файле ...\examples\index.html.

Построение примеров

Для того, чтобы построить примеры, необходимо выполнить два простых шага:

1 Установить значения переменных среды. О том, как это сделать, см. Inprise Application Server Installation Guide.

- CLASSPATH - Пользователи как Windows, так и Solaris устанавливают значение этой переменной как "." (точка). Все

необходимые jar-файлы загружаются автоматически. Для запуска примеров, которые используют базы данных, CLASSPATH должна содержать имя JDBC-драйвера.

- PATH - пользователи Solaris должны указать путь к утилите vbj.

2 Постройте примеры - при работе с Solaris, перейдите в каталог .../examples и наберите make. В случае Windows NT, наберите make_all.bat.

Makefile

Makefile автоматически компилирует код java, запускает компилятор java2iiop для нужных интерфейсов, генерирует jar-файл и проверяет его.

Ниже приведено описание того, что выполняется для примера sort:

1 Запускается vbjc и компилируется Java-код.

2 Запускается компилятор java2iiop для интерфейса SortHome. Компилятор генерирует стабы и скелеты CORBA (RMI-IIOP) для указанного интерфейса. Несмотря на то, что интерфейс Sort не был указан явно, указания SortHome достаточно для того, чтобы в результате анализа зависимостей SortHome были найдены все необходимые классы и сгенерирован весь требуемый код.

3 Запускается vbjc и компилируются два класса - SortHomePOAInvokeHandler и SortPOAInvokeHandler - вместе со всеми классамм, от которых они зависят.

4 Запускается стандартная утилита JDK jar для генерации ejb-jar-файла, необходимого для установки в Контейнер.

5 Запускается утилита VERIFY, входящая в состав Контейнера EJB, для проверки корректности ejb-jar-файла.

ejb-jar-файл содержит следующее:

- XML-Файл (или файлы) Дескрипторов Поставки.
- Файлы классов в виде байт-кода. В файл помещаются все файлы с расширением .class.

Для просмотра содержимого файла вы можете использовать стандартную утилиту jar:

```
jar tvf beans.jar
```

Более подробная информация о поставке (deployment) находится в Главе 9, "Поставка Enterprise JavaBeans".

Позднее, когда вы будете создавать свои собственные компоненты EJB, вы можете использовать данные make-файлы в качестве образца.

Запуск примеров

Для запуска программ:

1 Запустите VisiBroker Smart Agent. Это необходимо для выполнения некоторых начальных действий, например, установки взаимодействия клиента со службой имен. Smart Agent входит в комплект поставки Application Server. Для его запуска в консольном режиме из Windows, наберите

WinNT

```
prompt% osagent -C
```

При работе с UNIX, запуск VisiBroker Smart Agent'а в консольном режиме выполняется так:

UNIX

```
UNIX  % osagent &
```

2 Убедитесь, что вы находитесь в каталоге, где расположен пример, который вы хотите запустить. Например, для запуска программы sort вы должны находиться в каталоге ...\examples\sort.

3 Запустите Сервер EJB с помощью следующей команды (для запуска примера pigs):

```
prompt% vbj com.inprise.ejb.Container ejbcontainer pigs_beans.jar -jts
   -jns -jdb -jss
```

Флаги (-jts -jns -jdb) указывают на необходимость запуска соответствующих сервисов. Флаг -jdb запускает поставляемую с Контейнером Базу Данных, если в примере используются Entity-компоненты. Используйте флаг -jss для примеров с stateful Session-компонентами.

Пример bank требует обращения к внешней базе данных. Запустите Сервер для выполнения этого примера с помощью следующей команды:

```
prompt% vbj Djdbc.drivers=oracle.jdbc.driver.OracleDriver
com.inprise.ejb.Container \
   ejbcontainer bank_beans.jar -jts -jns
```

Обратитесь к файлу README в соответствующем каталоге для подробных объяснений, каким образом построить и запустить этот пример.

Примеры data и custom_cmp требуют, чтобы сервер назывался "ejbcontainer". Не меняйте этого названия без внесения соответствующих исправлений в параметр JDBC URL в Дескрипторе Поставки.

Ниже приведен список флагов, используемых для каждого примера:

Таблица 3.1 Флаги EJB-Сервера для примеров программ

Пример	Требуемые флаги
sort	-jns
cart	-jns -jts -jss
bank	-jns -its
pigs	-jns -its -jdb
data	-jns -its -jdb
custom_cmp	-jns -its

4 Подождите, пока Контейнер не запустится (появится следующее сообщение):

```
Container [ejbcontainer] is ready.
```

5 Откройте другое окно и запустите программу-клиента. К каждому примеру поставляется своя клиентская программа. Например, для примера sort клиент может быть запущен так:

```
prompt% vbj SortClient
```

Режим отладки

В первый раз, когда вы запускаете программу, вы можете захотеть включить отладочный режим.

Для запуска Контейнера/Сервера с включенным режимом отладки, наберите команду

```
prompt% vbj -DEJBDebug com.inprise.ejb.Container ejbcontainer
   sort_beans.jar -jns
```

Для запуска клиента с включенным режимом отладки, наберите

```
prompt% vbj -DEJBDebug SortClient
```

Исключение NotFound

Если при запуске клиента вы получили сообщение об исключении NotFound, проверьте следующее:

- Запущен ли Smart Agent?
- Запущен ли Контейнер?
- Запущен ли Контейнер в том же каталоге, где и клиент?
- Используется ли при запуске OSAgent, Контейнера и Клиента одно и то же значение переменной OSAGENT_PORT?
- Правильно ли заданы командные строки?

Основы Контейнера EJB и инструментальных средств

Вы можете запустить Контейнер и его сервисы как с помощью графического интерфейса пользователя (GUI), так и непосредственно из командной строки. Ниже описано, как использовать командную строку.

Запуск Контейнера EJB

Для запуска Контейнера EJB (или, если вам больше нравится, "сервера"), его сервисов и установки нужных режимов, используйте следующую команду:

```
prompt% vbj com.inprise.ejb.Container <server-name> <ejb-jar-files> [-
   jts] [-jns] [-jdb] [-jss]
```

Пример

```
prompt% vbj.com.inprise.ejb.Container ejbcontainer
   sort_beans jar -jts -jns -jdb -jss
```

Опции

Приведенная ниже таблица содержит список аргументов и опций, который вы можете использовать при запуске Контейнера, вместе с их описанием.

Опция	Описание
vbj	Это управляющая программа для VisiBroker for Java. vbj устанавливает значение порта OSAgent'а, используя текущее значение соответствующей переменной среды. Вместо нее вы можете использовать стандартную утилиту java с указанием дополнительной опции командной строки.
com.inprise.ejb.Container	Это имя Java-класса, содержащего функцию main для Контейнера.
server-name	Это имя сервера. (В настоящий момент не используется клиентом; используется при работе со средствами администрирования).
ejb-jar-files	Список, состоящий из любого количества имен jar-файлов, каждый из которых содержит один или несколько Компонентов EJB. Вы можете не указывать ни одного jar-файла.
-jts	Запускает встроенную (in-process) реализацию JTS (т.е. CORBA OTS). Если вы не указываете этот флаг, то вы должны иметь запущенный экземпляр внешней реализации Сервиса Транзакций (ITS).
-jns	Запускает встроенную реализацию CosNaming Service, которая используется для обслуживания JNDI-вызовов со стороны клиента. Если вы не указываете этот флаг, то вы должны иметь запущенный экземпляр внешней реализации Сервиса Имен.
-jdb	Запускает встроенную реализацию Java-БД (JDataStore). Она представляет собой долговременное хранилище для stateful Session-компонентов, а также позволяет

Опция	Описание
	использовать Entity-компоненты с CMP без обращения к внешним базам данных. Подробности ее использования приведены в описании примера pigs. Если вы не указываете этот флаг, то вы должны иметь запущенный экземпляр внешней базы данных.
-jss	Запускает встроенный сервис Java Stateful-Session-Storage, реализованный при помощи JDataStore. Это позволяет обеспечить долговременное хранение состояния stateful Session-компонентов.

Концепции EJB-Сервера и EJB-Контейнера

EJB-Сервер - это логическое понятие, представляющее собой набор сервисов, таких, как служба имен, сервис транзакций и т.п., которые необходимы для функционирования Компонентов EJB. Эти сервисы могут быть запущены как в одном адресном пространстве, так и в различных прстранствах.

Контейнер EJB представляет собой специализированный сервис, который непосредственно обслуживает Компоненты EJB. Передача Компонента EJB под управление этому серверу называется "поставкой" Компонента. Этот сервис берет на себя управление циклом жизни своих Компронентов, транзакциями, безопасностью, именами объектов, сохранением их состояний и др., в соответствие с требованиями и ограничениями спецификации EJB. Для решения многих из этих задач Контейнер обращается за помощью Серверу EJB.

Например, Контейнер может быть запущен с помощью следующей команды:

```
prompt% vbj com.inprise.ejb.Container test beans.jar -jns -jts -jdb
```

На самом деле, с помощью такой команды вы запускаете Контейнер, который затем запускает встроенные версии тех или иных сервисов. В принципе, вы можете считать, что запущен EJB-Сервер с четырьмя универсальными сервисами (выполняемыми в одном адресном пространстве) и с пятым, специализированным, сервисом - Контейнером EJB - который служит сервером для Компонентов EJB, находящихся в указанном jar-файле (beans.jar в нашем примере).

Вы могли бы задать следующий набор команд:

```
% vbj com.inprise.ejb.Container test_NS -jns
% vbj com.inprise.ejb.Container test_TS -jts
% vbj com.inprise.ejb.Container test_DB -jdb
% vbj com.inprise.ejb.Container test beans.jar
% vbj com.inprise.ejb.Container test_replica beans.jar
```

Сейчас вы получили точно такое же приложение с распределенной структурой. Сейчас ваш "EJB Server" представляет собой логическую совокупность универсальных сервисов и с дублированным Контейнером (предположительно, для обеспечения устойчивости к сбоям).

Флаги диагностики

Вы можете разрешить выполнение различных видов диагностики - как для EJB-клиента, так и для EJB-Контейнера. Ниже приведен список наиболее часто используемых флагов:

Флаг	Описание
EJBDebug	Обеспечивает диагностику для: EJB Container's EJB State machine Message interceptors CMP (Container-Managed Persistence)
EJBExceptions	выводит на печать сообщения при преобразовании RMI-исключений в CORBA-исключения. Во время такого преобразования происходит потеря определенной информации - трассировки стека, части сообщений. Установка флага позволяет выводить эту (иначе теряемую) информацию в стандартный поток сообщений об ошибках.
EJBTimer	Разрешает диагностику таймера, что позволяет разработчику отслеживать использование CPU Контейнером.
EJBDetailTimers	То же, что и EJBTimer, плюс информация о временных характеристиках выполнения методов. Установка этого флага позволяет пользователю отслеживать, как различные методы используют CPU. Вывод в консольном режиме может потребовать изменения параметров вашего терминала (увеличения количества столбцов) для того, чтобы избежать переноса части длинной строки на следующую строку.
EJBCopyArgs	Заставляет использовать при локальных вызовах методов передачу аргументов по значению. По умолчанию передача аргументов в этом случае происходит по ссылке. Имейте в виду, что многие Компоненты EJB в этом случае будут работать существенно медленнее. Например, пример sort будет выполняться примерно в пять раз дольше, так как процессор будет тратить много времени на копирование значений аргументов.

Инфо Флаги используются как defines при обращении к Java VM. Например, для установки флага EJBDebug в предыдущем примере, используйте команду

```
prompt% vbj -DEJBDebug com.inprise.ejb.Container test beans.jar -jns -
   jts -jdb
```

Использование флагов VisiBroker

Помимо флагов, специфических для EJB или RMI, вы можете использовать флаги для VisiBroker for Java. Два наиболее полезных из них - ORBwarn и ORBdebug:

Флаг	Описание
vbroker.orb.warn	Значение этого флага должно быть больше или равно 2 для включения диагностики ошибок ORB.
vbroker.orb.debug	Установите значение флага в True для включения подробной диагностики ORB; значение False запрещает подробную диагностику.
vbroker.agent.port	Параметр позволяет явно указать значение порта OSAgent'a.

Инструменты EJB

В настоящий момент в комплект поставки входят следующие полезные инструменты:

- Редактор Дескриптора Поставки (Deployment Descriptor Editor). Главная его задача - ввод нужной информации в удобном для пользователя виде с последующей генерацией XML-файла.
- Генератор кода EJB (EJB Code Generator) - это усовершенствованная версия компилятора RMI-IIOP. Он генерирует стабы, необходимые клиенту для выполнения удпленных вызовов, и скелеты на стороне сервера. Обратитесь к make-файлам примеров для просмотра команд вызова этого компилятора.
- Верификатор кода EJB (Verify utility) - это утилита для просмотра содержимого ejb-jar-файлов и проверке его на соответствия требованиям спецификации EJB. Например, утилита способна проверить, действительно ли для Компонента реализованы все методы его Remote-интерфейса. Функция `main` этой утилиты находится в классе `com.inprise.ejb.util.Verify`. Утилита выполняет проверку, основываясь на DTD, который генерирует конвертор dd2xml (который описан ниже). По определению, Verify проверяет XML-дескриптор, основываясь на этом DTD, даже если DTD отсутствует в явном виде. Вы можете отключить эту проверку путем присвоения значения "off" системному свойству EJBValidatingParser: "EJBValidatingParser=off".
- Утилита остановки Контейнера - Эта утилита позволяет корректно остановить сервер (Контейнер). Контейнер рассчитан и на внезапную остановку, но после этого при последующем запуске необходимо будет выполнить синхронизацию JDataStore, на что потребуется время - не менее десяти секунд. При нормальном завершении (с помощью утилиты остановки) повторный запуск Контейнера произойдет гораздо быстрее. Функция `main` для этой утилиты находится в классе `com.inprise.ejb.util.Shutdown`.
- Конвертер EJB 1.0 в EJB 1.1 - Эта утилита с именем dd2xml предназначена для преобразования Дескриптора (или Дескрипторов) Поставки в формате EJB 1.0 в XML-формат Дескрипторов EJB 1.1. Утилита также может быть использована для включения Компонентов EJB, созданных третьими сторонами, в Inprise EJB-Контейнер. dd2xml генерирует DTD, который помещается в XML-

дескриптор, что означает, что для работы с Дескриптором можно использовать любой XML-редактор. Пример cart подробно рассказывает о последовательности создания Компонента EJB 1.0 с последующим преобразованием к версии 1.1.

- Стандартная утилита jar JDK. Она может использоваться для создания ejb-jar-файлов. Подробности - в описании примеров.

Для получения описания синтаксиса каждой из команд введите ее без аргументов. Например, ввод команды

```
prompt% vbj com.inprise.ejb.util.Verify
```

приведет к появлению ее описания:

```
Usage: vbj com.inprise.ejb.util.Verify <ejb-server-name> [host-name>]
```

Инфо В комплект поставки Application Server включены графические средства управления серверами, Контейнерами и пр., которые вы можете использовать вместо командных строк.

Пример создания приложения с stateless Session-компонентом

В качестве первого знакомства с Контейнерами и Компонентами EJB, мы создадим простое клиент-серверное приложение, называемое SortClient, которое выполняет сортировку. Этот пример использует алгоритм сортировки слиянием. Задача - демонстрация основных шагов по созданию и компиляции Компонента с последующим запуском в Контейнере EJB. Пример также демонстрирует использование RMI-IIOP и взаимодействие объектов с помощью службы имен JNDI.

Сначала мы объясним, зачем нужен тот или иной фрагмент кода. Обычно написание кода, реализующего функциональность Компонента - это задача bean provider'а. Весь текст уже содержится в примере sort, поэтому ничего заново писать не придется. Затем мы покажем, как откомпилировать и построить Компонент, упаковать его, выполнить его поставку, создать приложение-клиент и, наконец, запустить его.

Для создания нашего простого приложения необходимо выполнить следующие шаги:

1 Написать код Компонента. Сюда входит написание собственно кода Компонента и его remote- и home-интерфейсов.

2 Откомпилировать и построить Компонент и его клиента.

3 Создать Дескриптор Поставки в каталоге META-INF.

4 Упаковать Компонент.

5 Поместить Компонент в JAR-файл.

6 Запустить приложение-клиент.

Написание Компонента

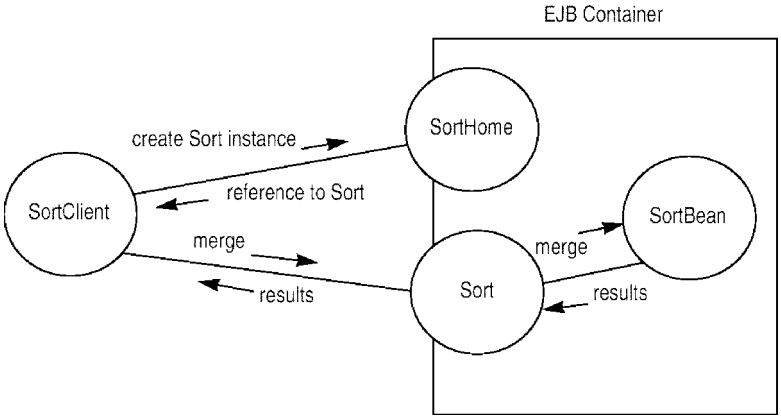
Для каждого Компонента EJB существуют три составляющих, и разработчик (bean provider) должен их написать. Это следующие части:

- Класс реализации - Это класс Компонента EJB. Он реализует бизнес-логику Компонента и функциональность, определяемую его home- и remote-интерфейсами. В нашем примере этот класс называется SortBean.java.
- Интерфейс Home - он определяет операции создания, поиска и уничтожения Компонента. Он играет роль Фабрики Объектов, если использовать принятую для распределенных систем терминологию. Имя этого интерфейса для нашего Компонента - SortHome.
- Интерфейс Remote - он определяет доступные для клиента бизнес-методы Компонента. Имя этого интерфейса в нашем примере - Sort.

Для того, чтобы клиент мог взаимодействовать с Компонентом EJB, вы должны написать все эти три составные части. Клиентское приложение использует home-интерфейс Компонента для поиска и получения ссылки на его remote-интерфейс. После получения такой ссылки, клиент может обращаться к любому методу remote-интерфейса. Клиент не знает, является этот метод локальным или удаленным. С точки зрения клиента, вызов удаленного метода Компонента EJB выглядит так же просто, как вызов любого локального метода. Контейнер EJB передает вызов клиента настоящему (и невидимому для клиента) экземпляру Компонента, используя необходимый комуникационный протокол, а затем возвращает результат вызова клиенту через remote-интерфейс.

На Рис. 3.1 показано, что происходит, когда клиентское приложение SortClient обращается к методу `merge()` Компонента SortBean. Обратите внимание, что клиент сначала вызывает метод `create()` интерфейса SortHome. Результатом вызова является ссылка на интерфейс Sort. После этого клиент вызывает метод `merge()` remote-интерфейса Sort, и этот вызов передается на выполнение соответствующему бизнес-методу Компонента SortBean.

Рис. 3.1 Флаги EJB сервера, необходимые для запуска примеров



В нашем примере Компонент является Session-Компонентом без состояния. Большинство Session-Компонентов состояние имеют - экземпляр Компонента создается в контексте сессии (или сеанса связи), и в продолжение всего сеанса экземпляр обслуживает запросы только этого клиента. По определению, stateful Session-Компонент сохраняет свое состояние, характеризующее его взамодействие с клиентом в продолжение всего сеанса.

С другой стороны, stateless Session-Компонент не предназначен для обслуживания вызовов только от одного клиента и не хранит состояние, оставшееся от обслуживания предыдущих вызовов. Когда клиент вызывает метод такого Компонента, Контейнер выбирает для обслуживания запроса любой экземпляр Компонента из пула. Компонент имеет определенное состояние, но только на время выполнения обслуживания одного вызова. После завершения обработки Компонент возвращает некоторый результат и не считается больше сопоставленным с конкретным клиентом. Контейнер сам решает, нужно или не нужно вернуть этот экземпляр Компонента в пул Компонентов.

Написание интерфейса home

Интерфейс home содержит методы для создания, поиска и уничтожения экземпляров Компонентов EJB. Home-интерфейс для Session-Компонента без состояния проще, чем для Session-Компонента с состоянием или для Entity-Компонента. Интерфейс SortHome наследует (extends) интерфейс `javax.ejb.EJBHome` и объявляет единственный метод - `create()`. Он не имеет аргументов и может возбуждать два типа исключений: `RemoteException` и `CreateException`.

Метод `create()` возвращает результат типа Sort; это и есть ссылка на remote-интерфейс нашего Компонента SortBean. Помните, что клиент не вызывает методы экземпляра Компонента непосредственно; вместо этого, клиент вызывает методы remote-интерфейса, поэтому метод `create()` и возвращает ссылку на этот интерфейс.

Session-Компоненты с состоянием и Entity-Компоненты, раз они могут поддерживать определенное состояние, могут предоставлять клиенту несколько методов для создания их экземпляров и могут требовать передачи некоторых параметров для выполнения инициализации экземпляра. Stateless Session-Компонент может иметь только один метод `create()`, причем без аргументов. Метод `remove()`, который удаляет экземпляр Компонента из пула, унаследован от интерфейса EBJHome, и от вас не требуется объявлять его в интерфейсе SortHome.

home-интерфейс для SortBean показан в Примере кода 3.1:

Пример кода 3.1 SortHome интерфейс

```
// SortHome java

public interface SortHome extends javax.ejb.EJBHome {
   Sort create() throws java.rmi.RemoteException,
   javax.ejb.CreateException;
}
```

Написание remote-интерфейса

Remote-интерфейс объявляет и делает доступным те методы Компонента, к которым может обращаться его клиент. Этот интерфейс наследует интерфейс `javax.ejb.EJBObject`.

Каждый метод, объявленный в интерфейсе Sort, должен иметь соответствующий метод в классе SortBean; методы должны иметь одинаковые имя и сигнатуры. Кроме того, методы remote-интерфейса Sort должны возбуждать исключение `java.rmi.RemoteException` и должны возвращать корректные типы RMI.

Remote-интерфейс Sort объявляет два метода: `sort()` и `merge()`. Оба могут возбуждать исключение `java.rmi.RemoteException` и оба возвращают тип Vector. Remote-интерфейс Sort для SortBean показан в Примере кода 3.2:

Пример кода 3.2 Sort интерфейс

```
// Sort java
import java.util.Vector;

public interface Sort extends javax.ejb.EJBObject {
   Vector sort(Vector v, Compare c) throws java.rmi.RemoteException;
   Vector merge(Vector a, Vector b, Compare c) throws
      java.rmi.RemoteException;
}
```

Написание реализации Компонента

Так как SortBean является Session-Компонентом, он обязан реализовать интерфейс `javax.ejb.SessionBean`. Обратите также внимание на то, что класс SortBean объявлен как public.

Реализационный класс нашего Компонента определяет бизнес-методы, к которым может обращаться клиент. В нашем примере, класс SortBean содержит реализации методов `sort()` и `merge()`. К этим двум методам клиент будет обращаться при запуске нашего приложения. Заметьте, что сигнатуры этих методов совпадают с сигнатурами методов remote-интерфейса Sort.

Компонент реализует также метод `ejbCreate()`, который соответствует методу `create()` home-интерфейса. Home-интерфейс SortHome определяет единственный метод `create()` без аргументов, возбуждающий исключение `java.rmi.RemoteException`. Класс SortBean реализует метод `ejbCreate()` без аргументов и также возбуждающий исключение java.rmi.RemoteException. Тем не менее, в отличие от метода `create()` интерфейса SortHome, который возвращает объект типа Sort, метод `ejbCreate()` возвращает void.

Класс SortBean не имеет public-конструктора, хотя это и разрешено. SortBean объявляет четыре метода, помимо бизнес-методов и методов создания (create). Вот эти методы: `ejbRemove()`, `ejbPassivate()`, `ejbActivate()` и `ejbSessionContext()`. За реализацию этих методов отвечает Контейнер; класс SortBean должен просто их объявить как public-методы, возвращающие тип void.

Пример кода 3.3 содержит фрагмент кода класса SortBean. Для ясности не приведен код методов `sort()` и `merge()`; кроме того, не показаны некоторые отладочные операторы. Полный текст содержится в файле SortBean.java, находящемся в каталоге ...\examples\sort.

Пример кода 3.3 Класс SortBean

```
// SortBean java
import java util Vector;

public class SortBean implements javax.ejb.SessionBean {

   public void setSessionContext(javax.ejb.SessionContext
      sessionContext) {}
   public void ejbCreate() throws java.rmi.RemoteException {}
   public void ejbRemove() throws java.rmi.RemoteException {}
   public void ejbActivate() throws java.rmi.RemoteException {}
   public void ejbPassivate() throws java.rmi.RemoteException {}

   public Vector sort(Vector v, Compare c) throws
      java.rmi.RemoteException {
   try (
      ...
      return result;
      }
      catch(javax.ejb.CreateException e) {
         throw new java.rmi.ServerException("Could not create
```

```
            sort instance", e);
         }
         catch(javax.ejb.RemoveException e) {
            throw new java.rmi.ServerException("Could not remove
            sort instance", e);
         }
      }
      public Vector merge(Vector a, Vector b, Compare c) throws
         java.rmi.RemoteException {
         ...
      }
   }
```

Написание кода клиента

Первое, что должно выполнить клиентское приложение - это получить доступ к home-интерфейсу для Компонента SortBean. Делается это с помощью вызова методов JNDI API. Как показано в Примере 3.4, SortClient создает контекст имен JNDI, а затем использует его метод lookup для поиска home-интерфейса для "sort":

Пример кода 3.4 Фрагмент кода SortClient

```
// SortClient java

public class SortClient {
   ...
   public static void main(String[] args) throws Exception {
      javax.naming.Context context;
      { // get a JNDI context using the Naming service
         context = new javax.naming.InitialContext();
      }
      Object objref = (SortHome) context.lookup("sort");
      SortHome home = (SortHome)
         javax.rmi.PortableRemoteObject.narrow(objref,
         SortHome.class);
      Sort sort = home.create();
      ... //do the sort and merge work
      sort.remove();
   }
}
```

После получения ссылки на интерфейс SortHome, клиент может вызвать его метод `create()` для получения ссылки на remote-интерфейс Sort. Получив эту ссылку, клиент может вызывать его методы `sort()` и `merge()` так же, как он мог бы обращаться к любым другим методам. На самом деле remote-интерфейс перенаправляет вызовы клиента собственно Компоненту EJB.

Построение Компонента и клиентского приложения

Мы компилируем и строим как Компонент, так и его клиента, используя Make-файл. Этот файл содержит все необходимые команды, а именно, обращается за выполнением требуемых действий к java2iiop, jar и verify.

Убедитесь, что ваши переменные среды - CLASSPATH и PATH - имеют правильные значения. См. раздел "Построение примеров" на стр. ??? для получения информации о том, как устанавливать значения этих переменных.

После того, как вы настроили переменные среды, откомпилируйте и постройте пример следующим образом:

Для Solaris:

```
prompt% make
```

Обратите внимание, что make-файл требует наличия UNIX-совместимой утилиты make.

Для Windows:

```
prompt% make_all
```

Make-файлы достаточно просты, и вы можете редактировать их "вручную", если необходимо.

Содержимое make-файла для примера sort показано ниже:

Пример кода 3.5 make-файл для примера sort

```
# Makefile

default: all

include  ../Makefile rules

SRCS = \
   SortClient.java \
   \
   Sort.java \
   SortHome.java \
   SortBean.java

CLASSES = $(SRCS:.java=.class)

beans jar: $(CLASSES)
   $(JAVA2IIOP) SortHome
   $(JAR) cMf beans.jar META-INF *.class
   $(VERIFY) beans.jar

all: $(CLASSES) beans.jar
```

Создание Дескриптора Поставки

Дескриптор Поставки представляет собой XML-файл, который определяет атрибуты Компонента EJB. Для создания Дескриптора можно использовать любой XML-редактор; для модификации гораздо удобнее работать с Редактором Дескрипторов. Для простоты вы можете создавать свои Дескрипторы на базе Дескрипторов из примеров. Эти XML-файлы вы найдете в подкаталогах META-INF каталогов соответствующих примеров.

Внимание! При использовании стандартного XML-редактора вы сами должны обеспечить соответствие требованиям спецификации EJB 1.1, и если вы хотите, чтобы ваш компонент мог быть помещен в EJB-Контейнер, вы должны использовать Inprise Document Type Definition (DTD).

Инфо В настоящий момент, вы должны использовать стандартный XML-редактор для создания Дескриптора Поставки. После того, как Дескриптор будет помещен в Jar-файл, вы можете для работы с ним использовать Редактор Дескриптора Поставки.

Дескриптор Поставки содержит информацию, необходимую для настройки Компонента; этим занимается Deployer.

Компоненты EJB предназначены для использования в различных приложениях без необходимости изменения их исходного кода. Но, поскольку каждое приложение использует некий универсальный Компонент по-своему, этот Компоненте необходимо настроить на взаимодействие с конкретным приложением - с помощью изменения свойств, хранящихся в Дескрипторе Поставки. Обратите внимание, что в Дескрипторе Компонента SortBean хранятся тип компонента (stateless session), имена его home-интерфейса (SortHome) и remote-интерфейса (Sort) и тип транзакции (container managed).

Подробное описание Дескриптора Поставки находится в Главе 9, "Поставка Enterprise JavaBeans".

Запуск примера Sort

Для того, чтобы выполнить пример, вы должны предварительно запустить Smart Agent для выполнения некоторых начальных действий - обеспечения возможности для клиента связаться со службой имен и др. (Smart Agent входит в комплект поставки Application Server).

Запустите EJB-Контейнер с помощью следующей команды:

```
prompt% vbj com.inprise.ejb.Container ejbcontainer sort_beans.jar -jts
   -jns
```

Для просмотра описания синтаксиса и флагов команды, обратитесь к разделу "Основы Контейнера EJB и инструментальных средств" на стр. 3-7 .

Запустите клиентское приложение с помощью команды:

```
prompt% vbj SortClient
```

Клиент выполняет поиск Компонента SortBean, а затем вызывает его методы для сортировки массивов переменной длины. Вы увидите ввод - строка, помеченная как "in" - и отсортированный выходной массив - строка, помеченная как "out" - на консоли. Результаты будут выглядеть примерно так:

```
in:  [1]
out: [1]
in:  [1, 2]
out: [1, 2]
in:  [1, 2, 7, 3, 8, 10, 4, 9, 5, 6]
out: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
in:  [-87, 91, 66, 85, -43, 29, 57, 57, 91, 22, -94, 55, 78, 50, 81,
      15, -51, 26, 13, 27, -56, -35, 84, 89, -98]
out: [-98, -94, -87, -56, -51, -43, -35, 13, 15, 22, 26, 27, 29, 50,
      55, 57, 57, 66, 78, 81, 84, 85, 89, 91, 91]
in:  [-4, 8.0, 3.0, -6, -4, 4]
out: [-6, -4, -4, 3.0, 4, 8.0]
in:  [this, is, a, test]
out: [a, is, test, this]
```

Глава

4

Разработка Компонентов EJB

В главе рассматриваются следующие основные темы:

- "Разработка Компонента: первые шаги" - краткий обзор основных этапов создания Компонента.
- "Использование JBuilder" - описывает, каким образом при этом можно использовать JBuilder.
- "Использование других средств разработки".
- "Разработка Компонента" - подробное рассмотрение процесса разработки Компонента.
- "Ограничения при программировании" - содержит список известных программных ограничений.

Разработка Компонента EJB: первые шаги

В общем случае, весь процесс разработки, поставки и сборки Компонента EJB включает в себя следующие шаги:

- Собственно разработка Компонента. Для создания Компонента необходимо построить один класс - класс Компонента, и два интерфейса - home- и remote-интерфейсы этого Компонента.
- Поставка Компонента. Сюда входит создание Дескриптора Поставки в XML-виде, который представляет собой файл, содержащий набор атрибутов Компонентов, и построение всех остальных необходимых классов, включая стабы и скелетоны, с последующим помещением всей этой информации в JAR-файлы.
- Сборка приложений. Этот процесс включает в себя установку Компонента на Сервере и проверку корректности всех связей Компонента с "внешним миром". Выполнением сборки занимается

application assembler. Задача сборки - построение на базе нескольких Компонентов и, может быть, других элементов - сервлетов, апплетов, скриптов и т.п. - готовых программ или других, более сложных Компонентов EJB.

- Управление Компонентами и другими элементами приложений.

Использование JBuilder

JBuilder очень хорошо взаимодействует с Контейнерами EJB. JBuilder и Inprise Application Server вместе предоставляют все необходимое - как инструменты, так и библиотеки - необходимые для создания и поставки Компонентов EJB:

- Гибкий и надежный Контейнер для управления Компонентами EJB.
- Службу имен.
- Сервис транзакций.
- Java-БД.
- API для создания Компонентов EJB.
- Усовершенствованный компилятор java-to-IIOP, который поддерживает передачу объектов по значению и сигнатуры RMI.
- Инструменты для генерации ejb-jar-файлов и для проверки вашего кода.
- Набор примеров, которые иллюстрируют использование Компонентов и Контейнеров EJB.

JBuilder предоставляет набор экспертов и инструментов для так называемого "быстрого создания " (RAD) Компонентов EJB. Этот эксперт проводит разработчика по всем необходимым этапам создания Компонента. Эти шаги, конечно, просты и интуитивно понятны; эксперт предлагает во многих случаях подходящие значения по умолчанию, и результатом его работы является заготовка Компонента, к которому разработчик добавляет реализации бизнес-методов.

JBuilder включает в себя эксперт интерфейсов EJB. Этот эксперт генерирует home- и remote-интерфейсы на основе public-методов Компонента. Существует также эксперт поставки (Deployment Wizard), который проводит разработчика по всем этапам создания Дескриптора Поставки в XML-формате и упаковывает сгенерированные стабы в JAR-файл.

Использование других средств разработки

Если вы используете другие средства разработки, то используйте при работе с Компонентами EJB только те средства взаимодействия с Контейнерами, которые входят в состав этого средства. Следует также

убедиться в том, что инструмент поддерживает правильные версии спецификации EJB и API EJB.

Вы должны убедиться, что вы имеете правильную версию JDK. Обратитесь к Руководству по инсталляции продукта для получения номеров версий, поддерживаемых Inprise EJB-Container'ом.

На стадии поставки вы обязаны использовать инструменты, входящие в состав Inprise Application Server, поскольку эти инструменты предусматривают выполнение всех необходимых изменений в Дескрипторе Поставки при работе с продуктами третьих фирм. Эти инструменты предусматривают выполнение проверок как Дескриптора Поставки, так и кода Компонентов.

Разработка Компонента EJB

В этом разделе внимание уделяется тем задачам, которые в процессе создания Компонента EJB должен решать bean provider. Вот они:

- Объявление и написание кода класса Компонента. Это и есть реализация бизнес-логики Компонента.
- Написание remote-интерфейса Компонента.
- Написание home-интерфейса Компонента.
- Задание класса так называемого "Primary Key" Компонента. Этот класс необходим только при создании Entity-Компонента. Имя этого класса хранится в Дескрипторе Поставки Компонента.

Разработчик Компонента (Bean Provider, по терминологии спецификации EJB) определяет home- и remote-интерфейсы и собственно класс Компонента EJB. Remote-интерфейс является интерфейсом, к методам которого обращается клиент Компонента. Частью этого интерфейса являются бизнес-методы Компонента. Методы home-интерфейса предназначены для создания, поиска и уничтожения экземпляров (instance) Компонентов.

Между home- и remote-интерфейсами и классом Компонента не существует никаких формальных отношений, подобных отношению наследования. Тем не менее, спецификация определяет правила, которым должны подчиняться, например, имена методов, объявленных в этих двух интерфейсах и классе. Например, если вы объявили некоторый метод класса Компонента, реализующий часть его функциональности (бизнес-логики), то точно такой же метод - с тем же именем и такой же сигнатурой - должен быть объявлен в remote-интерфейсе. Реализация Компонента имеет как минимум один метод с именем `ejbCreate()`; при этом она может иметь несколько методов с таким именем, но различными аргументами. Каждому из таких методов должен быть поставлен в соответствие метод с таким же списком аргументов, но с именем `create()`. Еще одно отличие состоит в том, что метод `ejbCreate()` класса возвращает другой тип результата по

сравнению с методом `create()` home-интерфейса: `create()` возвращает тип remote-интерфейса, а `ejbCreate()` - либо void (в случае использования сохранения, управляемого Контейнером, CMP), либо тип Primary Key Компонента (при использовании сохранения, управляемого Компонентом, BMP).

Обратите внимание на то, что метод `ejbCreate()` имеет свои особенности при работе с Entity-компонентами. Реализация Entity-Компонента может вообще не реализовывать метод `ejbCreate()`. Обычно это происходит, если такие Компоненты создаются не по запросу клиента, а автоматически, например, при добавлении новых записей в базу данных, представлением чего являются Entity-Компоненты. Типом возвращаемого значения для метода `ejbCreate()` является класс Primary Key Компонента. Метод `ejbCreate()` для Entity-Компонента с CMP должен возвращать void, для Entity-Компонента с BMP - класс Primary Key Компонента. Эти отличия подробно обсуждаются в главе 7, "Написание Entity-Компонента".

Разработчик Компонента (Bean Provider) определяет семантику Компонента EJB. Связь между классом Компонента и его home- и remote-интерфейсами обеспечивает Контейнер. Контейнер EJB также гарантирует (либо на этапе компиляции, либо на этапе выполнения) соответствие между классом Компонента и его remote-интерфейсом.

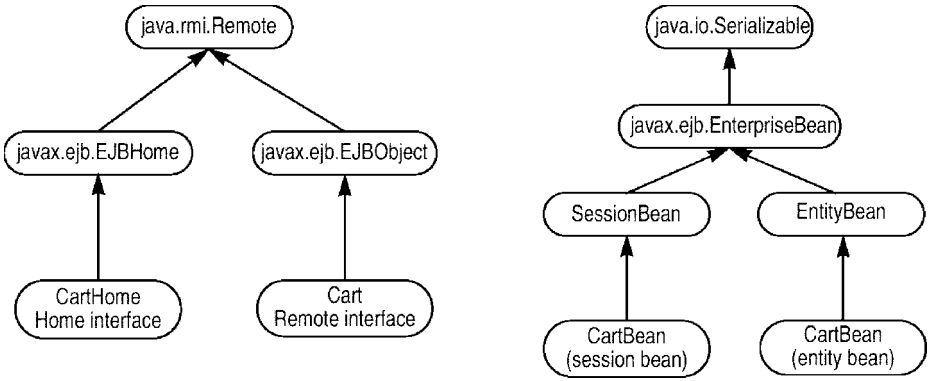
Наследование классов Компонента EJB.

Как home- и remote-интерфейсы, так и класс Компонента должны наследовать определенные классы EJB. Home-интерфейс всегда наследует (extends) интерфейс `javax.ejb.EJBHome`, Remote-интерфейс - интерфейс `javax.ejb.EJBObject`. В свою очередь, эти стандартные интерфейсы наследуют интерфейс `java.rmi.Remote`.

Session-Компонент должен реализовать его базовый класс javax.ejb.SessionBean, Entity-Компонент - javax.ejb.EntityBean. Оба этих стандартных класса являются производными от javax.ejb.EnterpriseBean, который, в свою очередь, наследует класс java.io.Serializable.

Деревья наследования показаны на рис. 4.1. Компонент CartBean на этой схеме может быть как Session-Компонентом, так и Entity-Компонентом. Его home-интерфейсом является CartHome, а remote-интерфейсом - Cart.

Рис. 4.1 Схемы наследования EJB



Интерфейс Remote

Каждый Компонент EJB должен иметь remote-интерфейс. Этот интерфейс содержит методы, определяющие бизнес-логику Компонента, к которым обращается его клиент. Класс Компонента предназначен главным образом для реализации этих методов. Имейте в виду, что клиент не может непосредственно обращаться к методам Компонента - доступ возможен только через remote-интерфейс.

Базовый класс EJBObject

Remote-интерфейс Компонента EJB является public-интерфейсом, который наследует интерфейс javax.ejb.EJBObject. Его код показан на Примере Кода 4.1.

Пример кода 4.1 Базовый интерфейс EJBObject для удаленного интерфейса (remote interface)

```
package javax ejb;
public interface EJBObject extends java.rmi.Remote {
   public EJBHome getEJBHome() throws java.rmi.RemoteException;
   public Object getPrimaryKey() throws java.rmi.RemoteException;
   public void remove() throws java.rmi.RemoteException,
      java.rmi.RemoveException;
   public Handle getHandle() throws java.rmi.RemoteException;
   boolean isIdentical (EJBObject p0) throws java.rmi.RemoteException;
}
```

Метод `getEJBHome()` позволяет вам получить соответствующий home-интерфейс. При работе с Entity-Компонентами, вы можете использовать метод `getPrimaryKey()` для получения Primary Key Компонента. Метод `remove()` уничтожает Компонент; о нем подробно говорится в теме, посвященной рассмотрению цикла жизни различных типов Компонентов. Метод `getHandle()` возвращает ссылку на экземпляр Компонента. Эта ссылка имеет смысл до тех пор, пока существует экземпляр Компонента. Метод `isIdentical ()` позволяет сравнивать Компоненты на идентичность.

Требования к методам

Все методы remote-интерфейса должны быть объявлены как public и возбуждать исключение `java.rmi.RemoteException`. Кроме того, аргументы этих методов и возвращаемые ими результаты должны иметь типы, соответствующие требованиям RMI-IIOP. Должно быть обеспечено соответствие между методами класса Компонента и методами его remote-интерфейса. Каждая пара методов должна иметь одно и то же имя, число и тип аргументов, тип результата и список возможных исключений.

Пример Кода 4.2 показывает код remote-интерфейса Atm для Session-Компонента ATM, который определяет бизнес-метод с названием transfer. Спецификация EJB требует, чтобы для каждого метода интерфейса присутствовали фрагменты, выделенные жирным шрифтом. Remote-интерфейс должен наследовать интерфейс `javax.ejb.EJBObject`. Объявляйте в remote-интерфейсе каждый бизнес-метод класса Компонента, к которому вы хотите предоставить доступ для клиента. Метод `transfer()` может возбуждать два исключения, одно из которых, `InsufficientFundsException`, является специфическим для приложения.

Пример Кода 4.2 Пример remote-интерфейса

```
public interface Atm extends javax.ejb.EJBObject {

   public void transfer(
      String source, String target, float amount)
      throws java.rmi.RemoteException, InsufficientFundsException;
}
```

Эти правила относятся как к Session-, так и к Entity-Компонентам.

Интерфейс Home

Home-интерфейс Компонента EJB служит для управления циклом жизни Компонента. Он объявляет операции для создания, поиска и уничтожения объектов, т.е. экземпляров Компонентов EJB. Циклы жизни Session- и Entity-Компонентов отличаются друг от друга. Следовательно, их home-интерфейсы содержат различные методы.

Разработчик Компонента (Bean Provider) обязан объявить home-интерфейс, но за его реализацию отвечает Контейнер EJB.

Как и в случае с remote-интерфейсом, все аргументы и типы результата методов, объявленных в home-интерфейсе, должны соответствовать требованиям RMI-IIOP. Все методы должны содержать исключение `java.rmi.RemoteException` в своем throw-списке исключений.

В home-интерфейсе должны быть объявлены один или несколько методов `create()`. Имя такого метода должно быть именно "create", а его аргументы - как их число, так и типы - должны быть такими же, как аргументы метода `ejbCreate()` класса реализации Компонента EJB.

Обратите внимание, что типы результата у соответствующих методов класса Компонента и его home-интерфейса отличаются друг от друга.

Home-интерфейс для Entity-компонента содержит также методы поиска (find-методы). Об этом будет подробно рассказано в разделе "Home-интерфейс Entity-Компонента" на стр. 4-9.

Базовый класс EJBHome

Каждый home-интерфейс наследует интерфейс `javax.ejb.EJBHome`. Пример Кода 4.3 показывает определение этого интерфейса:

Пример кода 4.3 Определение интерфейса EJBHome

```
package javax.ejb;

public interface EJBHome extends java.rmi.Remote {
   void remove(Handle handle) throws java.rmi.RemoteException,
      RemoveException;
   void remove(Object primaryKey) throws java.rmi.RemoteException,
      RemoveException;
   EJBMetaData getEJBMetaData() throws RemoteException;
   HomeHandle getHomeHandle() throws RemoteException;
}
```

Предусмотрены два варианта метода удаления экземпляра Компонента `remove()`. Первый из них удаляет экземпляр Компонента по его идентификатору (handle), второй - по значению primary key Компонента.

Handle Компонента представляет собой его уникальный идентификатор; для него может быть выполнена операция сериализации Java. Времена существования handle и сопоставленного с ним объекта совпадают. При работе с Entity-Компонентами, клиент может сохранять (с помощью Java-сериализации) значение этого идентификатора с последующим восстановлением в нужный момент. Этот идентификатор может корректно ссылаться даже на различные экземпляры Компонента; он остается корректным даже после сбоя на сервере с последующим перезапуском, а также в случае "перемещения" объектов на другие Сервера или компьютеры. Он очень похож на преобразованную к строковому виду объектную ссылку CORBA.

Второй вариант `remove()` для определения подлежащего удалению экземпляра Компонента использует значение его primary key. Его типом может быть любой тип Java, который наследует класс Object и реализует интерфейс Serializable. Primary key - главное средство для идентификации Entity-Компонентов. Как правило, он совпадает с первичным ключом в таблице базы данных, используемом для уникальной идентификации записи, объектным представлением которой является данный Компонент.

Метод `getEJBMetaData()` возвращает metadata-интерфейс для Компонента EJB. Этот интерфейс предоставляет клиенту возможность

получить информацию о структуре Компонента (его метаданные). Обычно его используют инструментальные программные средства, создающие приложения из поставленных Компонентов EJB. Интерфейс javax.ejv.EJBMetaData объявляет методы для получения ссылки на интерфейс javax.ejb.EJBHome, типов (классов) home- и remote-интерфейсов, а также типа primary key. Он также содержит метод isSession() для определения того, является ли Компонент Session- или Entity-Компонентом. Метод isStatelessSession() позволяет определить, имеет ли Session-Компонент состояние (т.е. является ли он stateless-Компонентом или нет). Пример Кода 4.4 показывает, что собой представляет интерфейс javax.ejv.EJBMetaData.

Пример Кода 4.4 Интерфейс javax.ejb.EJBMetaData

```
package javax.ejb;

public interface EJBMetaData {
   EJBHome getEJBHome();
   Class getHomeInterfaceClass();
   Class getRemoteInterfaceClass();
   Class getPrimaryKeyClass();
   boolean isSession();
   boolean isStatelessSession();
}
```

Home-интерфейс Session-Компонента EJB

Как уже говорилось в разделе "Session-Компоненты" на стр. 2-3, каждый экземпляр такого Компонента обслуживает запросы только одного клиента. Это означает, что после создания экземпляра Компонента по запросу клиента, этот экземпляр виден только для создавшего его клиента (подразумевается, что мы говорим о stateful-Компоненте, который поддерживает состояние, характеризующее его "отношения" с клиентом. Компонент без состояния, поскольку он не отслеживает историю вызовов, может быть использован для обслуживания запросов различных клиентов).

Home-интерфейс Компонента выступает в роли фабрики Компонентов, поскольку он содержит один или несколько методов create(). Спецификация EJB определяет следующие соглашения для каждого метода create():

- Он возвращает remote-интерфейс для своего Компонента.
- Имя метода - всегда "create".
- Каждый метод create() должен соответствовать методу ejbCreate в классе Компонента. Эти методы должны иметь одно и то же число аргументов одинаковых типов.
- Он должен объявлять возможность возбуждения исключения java.rmi.remoteException.

- Он должен объявлять возможность возбуждения исключения `javax.ejb.CreateException`.
- Аргументы метода `create()` используются для инициализации нового экземпляра Компонента.

Пример Кода 4.5 демонстрирует различные варианты метода create home-интерфейса Компонента. Обязательные фрагменты текста выделены жирным шрифтом.

Пример Кода 4.5 Пример методов create()

```
public interface AtmHome extends javax.ejb.EJBHome {

   Atm create()
      throws java.rmi.RemoteException, javax.ejb.CreateException;
   Atm create( Profile preferredProfile )
      throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

Обратите внимание, что home-интерфейс Session-Компонента не содержит методов для поиска объектов, поскольку такой объект может быть использован только создавшим его клиентом. Другие клиенты не могут получить к нему доступ и, следовательно, не нуждаются в средствах поиска.

Home-интерфейс Entity-Компонента EJB

Home-интерфейс Entity-Компонента содержит точно такие же методы create(), как и Session-Компонент. В дополнение к ним, интерфейс определяет find-методы для поиска нужных экземпляров Компонента с целью их последующего использования. Такие операции необходимы, так как экземпляры Entity-Компонентов являются "долгоживущими" объектами и могут использоваться многими клиентами. Для большинства приложений, используемые в них Entity-Компоненты уже существуют и клиент просто должен найти нужный экземпляр для выполнения вызова.

Home-интерфейс Entity-Компонента EJB должен обеспечивать базовый метод поиска, `findByPrimaryKey (primaryKey)`, который служит для поиска экземпляра Компонента по значению его primary key. Метод имеет единственный аргумент, а в качестве результата возвращает remote-интерфейс Компонента. В качестве primary key может использоваться любой тип Java, производный от Object. Вы помещаете тип primary key в Дескриптор Поставки. Обратите внимание, что метод `findByPrimaryKey ()` по определению возвращает только один объект, в то время как другие find-методы могут возвращать наборы объектов.

Пример Кода 4.5 метод findByPrimaryKey()

```
<entity bean's remote interface> findByPrimaryKey(
   <primary key type> key )
   throws java.rmi.RemoteException, FinderException;
```

Home-интерфейс может объявлять и другие методы поиска. Каждый такой метод должен быть реализован в классе Компонента. Каждый finder-метод home-интерфейса Компонента должен следовать следующим соглашениям:

- Типом результата является либо remote-интерфейс Компонента, либо, в случае соответствия сразу нескольких экземпляров Компонента указанному критерию поиска, набор (collection) таких интерфейсов. Корректными типами-наборами Java являются `java.util.Enumeration` (для JDK 1.1) и `java.util.Collection` (для Java 2).
- Имя метода поиска всегда начинается с префикса "find". Соответствующий ему finder-метод в классе Компонента должен начинаться с "ejbFind".
- Он должен объявлять возможность возбуждения исключения `java.rmi.remoteException`.
- Он должен объявлять возможность возбуждения исключения `javax.ejb.FinderException`.
- Списки возможных исключений методов `find()` home-интерфейса и `ejbFind()` класса Компонента должны быть одинаковыми.

В дополнение к этому, home-интерфейс Entity-Компонента может объявлять один или несколько методов `create()`. Все эти методы возвращают remote-интерфейс Компонента EJB. Список их аргументов определяется разработчиком приложения.

Методы create home-интерфейса должны соответствовать следующим правилам:

- Они должны объявлять возможность возбуждения исключения `java.rmi.remoteException`.
- Они должны объявлять возможность возбуждения исключения `javax.rmi.CreateException`.
- Они возвращают remote-интерфейс Компонента.
- Имя метода - всегда "create".
- Каждому методу `create()` интерфейса должен соответствовать метод `ejbCreate()` в классе Компонента с тем же списком своих аргументов.
- Список возможных исключений метода `create()` интерфейса должен включать в себя все исключения методов `ejbCreate()` и `ejbPostCreate()` класса Компонента, т.е. список исключений метода create() должен являться надмножестом объединения списков исключений методов `ejbCreate()` и `ejbPostCreate()`. Типом результата метода `ejbCreate()` всегда является класс Primary Key.
- Аргументы метода `create()` используются при инициализации нового экземпляра Компонента.

Пример Кода 4.7 демонстрирует различные виды find- и create-методов. Требуемые фрагменты текста выделены жирным шрифтом.

Пример кода 4.7 Методы Create и find

```
public interface AccountHome extends javax.ejb.EJBHome {

   Account create( String accountId )
      throws java.rmi.RemoteException, javax.ejb.CreateException;

   Account create( String accountId, float initialBalance )
      throws java.rmi.RemoteException, javax.ejb.CreateException;

   Account findByPrimaryKey( String key )
      throws java.rmi.RemoteException, javax.ejb.FinderException;

   Enumeration findBySocialSecurityNumber( String socialSecurityNumber )
      throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

Реализация Компонента EJB

Именно на стадии реализации Компонента происходит написание кода его бизнес-методов. Реализация Компонента - прерогатива Bean Provider'а. Для Session-Компонента также необходимо реализовать методы интерфейса `javax.ejb.SessionBean`, для Entity-Компонента - интерфейса `javax.ejb.EntityBean`. Оба этих интерфейса наследуют базовый интерфейс Компонентов EJB - `javax.ejb.EnterpriseBean`.

Реализации Session- и Entity-Компонентов отличаются друг от друга. Обратитесь к Главе 6, "Написание Session-Компонента", для получения подробной информации о реализации Session-Компонентов. Обратитесь к Главе 7, "Написание Entity-Компонента", для получения подробной информации о реализации Entity-Компонентов.

Интерфейс EnterpriseBean

Интерфейс EnterpriseBean Компонентов EJB является общим базовым интерфейсом и, в свою очередь, он наследует интерфейс `java.io.Serializable`. Этот интерфейс не содержит объявлений методов. Его код показан в Примере Кода 4.8.

Пример Кода 4.8 Интерфейс EnterpriseBean

```
package javax.ejb;

public interface EnterpriseBean extends java.io.Serializable {}
```

Идентификаторы (Handles)

Идентификатор (handle) представляет собой доступную для удаленных объектов догическую ссылку на экземпляр Компонента или его home-интерфейс. Интерфейс `javax.ejb.Handle`, который должен быть реализован для всех классов Идентификаторов Компонента, предоставляет хранимую (persistence) ссылку на экземпляр Компонента. Ниже приведен его код:

Пример Кода 4.9 Интерфейс Handle

```
public interface javax ejb Handle extends java.io.Serializable {
   public EJBObject getEJBObject() throws java.rmi.RemoteException;
}
```

Идентификатор может быть использован для долговременного хранения ссылки на экземпляр Компонента путем Java-сериализации экземпляра класса, реализующего handle-интерфейс. Это очень похоже на преобразование к строковому виду объектной ссылки CORBA. Интерфейс `javax.ejb.Handle` наследует интерфейс `java.io.Serializable`.

Существует также интерфейс javax.ejb.HomeHandle, который позволяет использовать хранимую ссылку на home-интерфейс Компонента. Этот интерфейс должен быть реализован в классах, являющихся представлением такой ссылки. Пример Кода 4.10 показывает код интерфейса:

Пример Кода 4.10 Интерфейс HomeHandle

```
public interface HomeHandle extends java.io.Serializable {
   public EJBHome getEJBHome() throws RemoteException;
}
```

Обычно реализацию handle-интерфейсов обеспечивает Контейнер.

Ограничения при программировании

Ниже приведен список программных ограничений, определенных в спецификации EJB 1.1.

- Компонентам EJB не разрешается управлять потоками (threads) или группами потоков. Им не следует создавать новые потоки или активизировать приостановленные, а так же завершать или приостанавливать выполняемые. Кроме того, Компоненты EJB не должны менять приоритеты потоков или их имена.
- Компонентам EJB не разрешено использовать static-поля, за исключением полей "только для чтения". Следовательно, все static-поля должны быть объявлены как final.
- Компонентам EJB не разрешено использовать средства синхронизации потоков для синхронизации выполнения кода нескольких экземпляров Компонента.

- Компоненты не должны использовать средства Java AWT для вывода информации на экран или воспринимать информацию, вводимую с клавиатуры.
- Компонентам не следует использовать пакет `java.io` для работы с файлами или каталогами файловой системы.
- Компонентам EJB следует избегать использования сокетов; особенно им не следует получать информацию из сокетов и отслеживать соединения. Кроме того, им не следует взаимодействовать с фабриками сокетов, используемыми `ServerSocker` и `Socket`, или фабриками, используемыми URL.
- Компоненты EJB не должны обращаться к классам или пакетам, а также пытаться получить информацию о классах, с использованием способов, не поддерживаемых языком Java, а также пытаться получить доступ к классам, недоступным для Компонента.
- Компонент EJB не должен обращаться к функциям и сервисам среды исполнения, взаимодействием с которыми занимается Контейнер EJB - создавать загрузчики классов или получать доступ к их контексту, устанавливать или создавать менеджеров безопасности, останавливать JVM, изменять стандартные потоки ввода-вывода.
- Компонент не должен получать информацию о профилях безопасности для конкретных источников кода.
- Компоненты EJB не должны загружать native-библиотеки.
- Компоненты не должны добавлять классы в пакеты, так как это (в целях обеспечения безопасности) должны делать Контейнеры.
- Компоненты EJB не должны использовать subclass и возможности object substitution Java Serialization Protocol.
- Следует проявлять осторожность при передаче `this` в качестве аргументов или при возврате результата. Безопаснее в этом случае использовать результаты вызовов методов `SessionContext.getEJBObject()` или `EntityContext.getEJBObject()`.
- Компонентам EJB не разрешается получать доступ или изменять объекты конфигурации безопасности. Например, им не разрешено изменять свой `java.security.Identity`. Любые попытки такого рода должны приводить к возникновению исключения `java.security.SecurityException`.

Глава

5

Написание клиентского приложения

В главе рассматриваются следующие основные темы:

- Компонент EJB с точки зрения клиента
- Управление транзакциями
- Получение информации о Компоненте EJB
- Поддержка JNDI
- Отображение EJB на CORBA

Компонент EJB с точки зрения клиента

Клиентом Компонента является приложение - независимое (stand-alone) приложение, сервлет или апплет - или другой Компонент. В любом случае для использования Компонента EJB клиент должен выполнить следующие действия:

- Получить доступ к home-интерфейсу Компонента. Спецификация EJB говорит, что для получения ссылки на интерфейс клиенту следует использовать JNDI (Java Naming and Directory Interface) API.
- Получить ссылку на remote-интерфейс Компонента. Для этого используются методы home-интерфейса. Вы можете либо создать Session-Компонент, либо создать или найти Entity-Компонент.
- Вызвать те или иные методы доступные методы Компонента. Клиент не может обращаться непосредственно к методам собственно Компонента. Вместо этого клиент вызывает методы его remote-интерфейса. Это те методы, которые Компонент объявляет доступными для клиента.

Инициализация клиента

Приложение SortClient импортирует все необходимые классы JNDI и home- и remote-интерфейсы Компонента SortBean (см. Пример Кода 5.1 на стр. 4-2.) Для поиска home-интерфейса используется JNDI API.

Поиск home-интерфейса

Клиент получает ссылку на home-интерфейс с помощью JNDI, как это показано в Примере Кода 5.1. Вначале клиент должен получить базовый контекст службы имен (initial naming context). В программе создается новый объект типа `javax.naming.Context`, который в нашем примере называется context. После этого клиент вызывает его метод `lookup()` для получения ссылки на home-интерфейс. Обратите внимание, что способ инициализации фабрики initial naming context зависит от реализации Контейнера/Сервера.

Метод `lookup()` контекста возвращает объект типа `java.lang.Object`. Ваша программа должна выполнить преобразование к нужному типу. Пример Кода 5.1 содержит фрагмент кода из примера sort. Функция main() начинается с использования службы имен JNDI и метода `lookup()` ее контекста для получения ссылки на home-интерфейс.

Вы передаете имя remote-интерфейса (sort в нашем случае) методу `context.lookup()`. Заметьте, что программа пробует преобразовать результат вызова метода к типу SortHome, т.е. типу home-интерфейса.

Пример Кода 5.1 Использование JNDI для поиска home-интерфейса Компонента

```
// SortClient java

import javax.naming.InitialContext;
import SortHome; // import the bean's home interface
import Sort; // import the bean's remote interface

public class SortClient {
   ...
   public static void main(String[] args) throws Exception {
      javax.naming.Context context;
      { // get a JNDI context using the Naming service
         context = new javax.naming.InitialContext();
      }
      Object objref = context.lookup("sort");
      SortHome home = (SortHome)
               javax.rmi.PortableRemoteObject.narrow(objref,
               SortHome.class);
      Sort sort = home.create();
      ... //do the sort and merge work
```

```
            sort remove();
        }
    }
```

Функция `main()` клиентской программы может возбуждать самое общее исключение Exception. При таком способе кодирования программа SortClient не должна обрабатывать любое исключение, и возбуждение любой исключительной ситуации приведет к завершению программы.

Получение remote-интерфейса

Сейчас, после того, как мы получили home-интерфейс Компонента, мы можем получить ссылку на его remote-интерфейс. Для этого мы должны использовать create- или find-методы home-интерфейса. Какой именно метод нужно использовать, зависит от вида Компонента и того, какие методы для Компонента предусмотрел его разработчик.

Например, в Примере Кода 5.1 показано как SortClient получает ссылку на remote-интерфейс Sort. После того, как клиент получил ссылку на home-интерфейс и преобразовал ее к нужному типу (SortHome), он создает экземпляр Компонента и вызывает его методы. Создание выполняется с помощью обращения к методу home-интерфейса `create()`, который и возвращает ссылку на remote-интерфейс Sort. (так ка SortBean является stateless session-Компонентом, его home-интерфейс должен содержать только один метод `create()` без аргументов.) Теперь SortClient может вызывать методы remote-интерфейса - `sort()` и `merge()` - для выполнения необходимых действий. После завершения работы клиент вызывает метод `remove()` remote-интерфейса для уничтожения экземпляра Компонента.

Session-Компоненты

Клиент получает ссылку на remote-интерфейс таких Компонентов EJB с помощью вызова одного из create-методов их home-интерфейсов.

Все Session-Компоненты должны определять как минимум один метод `create()`. Компонент без состояния (stateless) должен иметь только один метод с таким именем, причем он не может иметь аргументов. Компонент с состоянием (stateful) может иметь один метод `create()`, и еще несколько таких методов дополнительно, с различными списками аргументов. Если метод `create()` получает аргументы, их значения используются для инициализации создаваемого экземпляра Компонента.

Базовый метод `create()` не имеет аргументов. Например, в примере sort используется stateless session-Компонент. По определению, он имеет один метод `create()` без аргументов:

```
Sort sort = home.create();
```

С другой стороне, в примере cart используется stateful Session-Компонент, и его home-интерфейс, CartHome, объявляет несколько

вариантов метода create(). Один из них получает три аргумента, которые вместе идентифицируют покупателя, и возвращает ссылку на remote-интерфейс Cart. Клиентская программа CartClient устанавливает значение этих трех параметров - cardHolderName, creditCardNumber и expirationDate - и вызывает метод create(). Все это показано в Примере Кода 5.2:

Пример Кода 5.2 Вызов метода create()

```
Cart cart;
   {
      String cardHolderName = "Jack B. Quick";
      String creditCardNumber = "1234-5678-9012-3456";
      Date expirationDate = new GregorianCalendar(2001, Calendar.JULY,
              1).getTime();
      cart = home.create(cardHolderName, creditCardNumber,
              expirationDate);
   }
```

Session-Компоненты не имеют finder-методов.

Entity-Компоненты

Клиент получает ссылку на remote-интерфейс такого Компонента с помощью операций создания или поиска. Напоминаем, что Entity-Компонент является объектным представлением неких хранящихся в БД данных. Поскольку данные хранятся в общем случае в течение долгого времени, то и Компоненты являются "долгоживущими" - как правило, они существуют и после того, как завершилось создавшее их клиентское приложение. Таким образом, в большинстве случаев клиент ищет готовый экземпляр Компонента, а не создает новый, что связано с помещением новых данных в БД.

Для поиска клиент вызывает предназначенные для этого методы, подобно тому, как это происходит с поиском нужной записи в реляционной БД. Это подразумевает, что find-методы выполняют поиск экземпляров Компонентов, чьи данные уже помещены в хранилище данных. Эти данные могут быть добавлены как с помощью Entity-Компонентов, так и с помощью других инструментов, не имеющих отношения к EJB - например, непосредственно средств управления данной базой данных (Database management system, DBMS), или, в случае унаследованных (legasy) систем, данные уже существуют перед инсталляцией Контейнера EJB.

Для создания экземпляра Компонента с помещением его состояния в сопоставленную с ним БД клиент вызывает метод create(). Для инициализации внутренних переменных Компонента используются аргументы метода create(). Этот метод всегда возвращает ссылку на remote-интерфейс, хотя соответствующий ему метод ejbCreate() возвращает ключ (primary key) экземпляра Компонента.

Каждый экземпляр Entity-Компонента должен иметь primary key, который является его уникальным идентификатором. Экземпляр

Компонента может иметь и другие ключи, которые клиент может использовать при выполнении поиска.

Методы поиска и класс ключа компонента

Основным методом поиска Компонента является метод `findByPrimaryKey()`, который выполняет поиск по значению первичного ключа. Вот его сигнатура:

```
<remote interface> findByPrimaryKey( <key type> primaryKey )
```

Метод `findByPrimaryKey()` должен быть реализован для каждого Entity-Компонента. Тип единственного аргумента - `primaryKey` - представляет собой класс, объявленный в Дескрипторе Поставки. Этот тип должен соответствовать ограничениям RMI-IIOP, и может являться как стандартным классом Java, так и классом, созданным разработчиком.

Например, вы можете иметь Компонент Account, для которого определен класс его первичного ключа AccountPK. Типом AccountPK является стандартный тип String. Мы можем получить ссылку на нужный экземпляр Компонента Account, установив требуемое значение идентификатора счета и вызвав метод `findByPrimaryKey()`, как показано в Примере Кода 5.3.

Пример Кода 5.3 Поиск экземпляра Компонента по первичному ключу

```
AccountPK accountPK = new AccountPK("1234-56-789");
Account source = accountHome.findByPrimaryKey( accountPK );
```

Разработчик Компонента (bean provider) может определить дополнительные методы поиска.

Методы создания и удаления экземпляра

Клиент также может создать новый экземпляр Компонента, обратившись к одному из create-методов, объявленных в home-интерфейсе. Когда клиент вызывает метод `create()` для Entity-Компонента, состояние нового экземпляра помещается в базу данных. Новый объект обязательно должен иметь ключ, который является его идентификатором. Для установки начального состояния могут использоваться значения аргументов метода `create()`.

Всегда помните, что экземпляр Entity-Компонента существует, пока существуют сопоставленные с ним данные в БД. Время жизни такого Компонента не ограничено временем сеанса связи с клиентом. Экземпляр Компонента может быть уничтожен с помощью вызова одного из методов `remove()` - эти методы уничтожают компонент и удаляют сопоставленную с ним информацию из базы данных. Существуют и другие способы удаления Entity-Компонентов, например, с помощью средств системы управления базой данных или унаследованной системой.

Вызов методов

После того, как получена ссылка на remote-интерфейс Компонента, клиент может обращаться к методам этого интерфейса. Наиболее интересны, конечно, методы, относящиеся к реализации бизнес-логики Компонента. Кроме них, существуют методы для получения информации о Компоненте и его интерфейсах, получения идентификатора (handle) экземпляра, выполнения проверки на идентичность двух объектов и удаления экземпляров Компонента.

Пример кода 5.4 иллюстрирует использование клиентом методов Компонентов EJB, в нашем случае, Session-Компонента cart. мы приводим фрагмент текста с момента создания нового экземпляра Компонента и получения ссылки на remote-интерфейс. После этого клиент может обращаться к нужным методам.

Во-первых, клиент создает новый объект "книга" и устанавливает его параметры "заглавие" и "цена". Затем он добавляет этот объект к списку покупок с помощью вызова бизнес-метода `addItem()`. Это метод определен в классе CartBean Session-Компонента и сделан доступным для клиента его объявлением в remote-интерфейсе.

После выбора всех нужных книг (не показано в примере) клиент вызывает свой собственный метод `summarize()` для получения списка покупок в чеке. После этого вызывается метод `remove()` для уничтожения экземпляра объекта. Обратите внимание, что обращение к методам Компонента выглядит точно так же, как и обращение к локальным методам клиента, таким, как `summarize()`.

Пример Кода 5.4 Вызов методов Компонента

```
...
Cart cart;
{
   ...
   // obtain a reference to the bean's remote interface
   cart = home.create(cardHolderName, creditCardNumber,
            expirationDate);
}
// create a new book object
Book knuthBook = new Book("The Art of Computer Programming", 49 95f);
// add the new book item to the cart
cart addItem(knuthBook);
...
// list the items currently in the cart
summarize(cart);
cart removeItem(knuthBook);
...
```

Удаление экземпляров Компонента

Метод `remove()` выполняется по-разному для Session- и Entity-Компонентов. Так как Session-Компонент является временным и взаимодействует только с одним клиентом, то клиенту при завершении работы с Компонентом следует вызвать метод `remove()`. Клиент может использовать один из двух доступных методов: `javax.ejb.EJBObject.remove()` и `javax.ejb.EJBHome.remove(Handle handle)` (об использовании handle см. раздел "Использование идентификаторов компонентов" на стр. 4-7).

Хотя выполнение удаления объекта не является жестким требованием, это рассматривается как "хороший" стиль программирования. Если клиент не удалил stateful session-компонент явно, это будет сделано Контейнером по истечении определенного времени (интервала тайм-аута). Значение этого интервала хранится как свойство в Дескрипторе Поставки. Тем не менее, клиент может сохранить идентификатор этого экземпляра для дальнейшего использования.

Клиент Entity-Компонента не должен заботиться о таких проблемах, так как Entity-Компонент сопоставлен с клиентом только на время существования транзакции, а всеми вопросами цикла жизни Компонента (включая его активизацию и деактивизацию) ведает Контейнер. Клиент Entity-Компонента вызывает метод remove() только тогда, когда необходимо удалить сопоставленную с объектом информацию из базы данных.

Использование идентификаторов Компонентов

Идентификатор (handle) предоставляет другой способ получить доступ к Компоненту EJB. Идентификатор представляет собой хранимую ссылку на объект. Получить идентификатор можно, обратившись к функциям remote-интерфейса. После того, как идентификатор получен, он может быть записан в файл или сохранен другим способом. В нужный момент вы можете восстановить его из хранилища и восстановить взаимодействие с Компонентом.

Тем не менее, таким способом вы можете восстановить связь с компонентом, но не сам Компонент; если другой процесс уничтожил экземпляр компонента, произошел сбой в системе, перезапуск Сервера или что-то другое, что привело к уничтожению экземпляра Компонента, то при попытке восстановить с ним связь будет возбуждено исключение.

Если вы не уверены в том, что экземпляр Компонента все еще существует, то лучше вместо идентификатора remote-интерфейса восстановить идентификатор home-интерфейса, после чего создать экземпрляр с помощью create- или find-методов.

Для получения идентификатора объекта после его создания, вы можете использовать метод `getHandle()`, после чего он может быть записан в

файл с использованием механизма сериализации Java. Когда это станет необходимо, программа может извлечь данные из этого файла и преобразовать их к типу Handle. Затем она обращается к его методу `getEJBObject()` для получения ссылки на Компонент (с выполнением преобразования типа результата `getEJBObject()` к нужному типу).

В качестве иллюстрации рассмотрим пример, как программа CartClient могла бы сохранить идентификатор компонента CartBean:

Пример Кода 5.5 Использование идентификатора для доступа к Компоненту

```
import java.io;
import javax.ejb.Handle;
...
Cart cart;
cart = home.create(cardHolderName, creditCardNumber, expirationDate);
// call getHandle on the cart object to get its handle
cartHandle = cart.getHandle();
// write the handle to serialized file
FileOutputStream f = new FileOutputStream ("carthandle.ser");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(cartHandle);
o.flush();
o.close();
...
// read handle from file at later time
FileInputStream fi = new FileInputStream ("carthandle.ser");
ObjectInputStream oi = new ObjectInputStream(fi);
//read the object from the file and cast it to a Handle
cartHandle = (Handle)oi.readObject();
oi.close();
...
// Use the handle to reference the bean instance
try {
   Object ref = context.lookup("cart");
   Cart cart1 = (Cart) javax.rmi.PortableRemoteObject.narrow(ref,
            Cart.class);
   ...
} catch (RemoteException e) {
   ...
}
...
```

После завершения работы с идентификатором, можно удалить его, обратившись к методу `javax.ejb.EJBHome.remove(Handle handle)`.

Управление транзакциями

Клиентская программа может сама управлять своими транзакциями вместо того, чтобы полагаться в этом смысле на Компонент (или Контейнер EJB). Клиент, который управляет своими транзакциями, делает это точно так же, как и Session-Компонент, который управляет своими.

Когда клиент берет на себя управление транзакциями, естественно, он определяет их границы. Это означает, что он должен явно как начать, так и завершить (подтвердить или откатить) транзакцию.

Для управления транзакциями клиент использует интерфейс `javax.transaction.UserTransaction`. Прежде всего, он должен (с помощью JNDI) получить на него ссылку. После создания контекста `UserTransaction`, клиент начинает транзакцию, вызвав метод `UserTransaction.begin()`, за которым позднее следует вызов `UserTransaction.commit()` для подтверждения (или `UserTransaction.rollback()` для отката) и завершения транзакции. Между этими вызовами клиент выполняет запросы к БД и вносит необходимые изменения.

Пример Кода 5.6 показывает фрагмент кода, который должен быть написан в клиентской программе. Жирным шрифтом выделены операторы, непосредственно относящиеся к управлению транзакциями.

Пример Кода 5.6 Транзакции, управляемые клиентом

```
...
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
...
public class clientTransaction {
   public static void main (String[] argv) {
      UserTransaction ut = null;
      InitialContext initContext = new InitialContext();
      ...
      ut = (UserTransaction)initContext.lookup("java:comp/
                UserTransaction");
      // start a transaction
      ut.begin();
      // do some transaction work
      ...
      // commit or rollback the transaction
      ut.commit(); // or ut.rollback();
      ...
   }
}
```

Получение информации о Компоненте EJB

Информацию о самом Компоненте EJB обычно называют его метаданными. Клиент получает метаданные Компонента с помощью вызова метода home-интерфейса `getMetaData()`.

Как правило, метод `getMetaData()` используется средами разработки и другими инструментальными средствами, которые нуждаются в информации о структуре компонентов, например, для организации связи уже поставленных компонентов друг с другом. Такая информация может быть полезна также и для клиентов, использующих скрипты.

После того, как клиент получил ссылку на home-интерфейс, он может вызвать его метод `getEJBMetaData()`, после чего обращается к методам полученного интерфейса `EJBMetaData` для получения следующей информации:

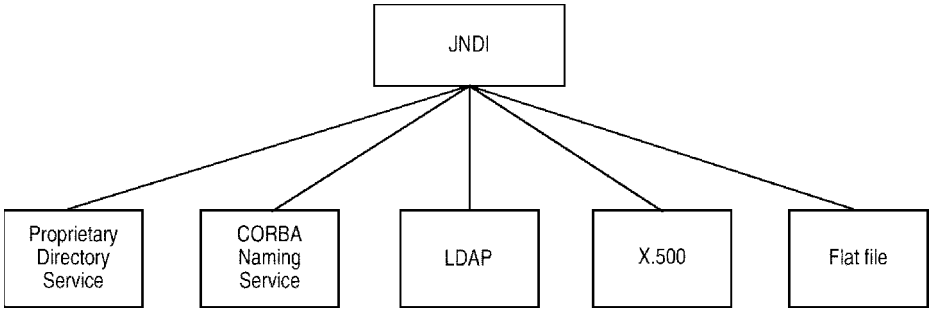
- Home-интерфейса Компонента `EJBHome` (с помощью вызова метода `EJBMetaData.getEJBHome()`).
- Объектов класса home-интерфейса, таких, как его интерфейсы, классы, поля и методы (с помощью `EJBMetaData.getHomeInterafaceClass()`).
- Объекты класса remote-интерфейса объекта, включая всю информациию о классе (с помощью `EJBMetaData.getRemoteInterafaceClass()`).
- Класса primary key Компонента (с помощью `EJBMetaData.getPrimaryKeyClass()`).
- Является ли Компонент Session- или Entity-компонентом (с помощью `EJBMetaData.isSession()`). Метод возвращает значение true, если это Session-Компонент.
- Является ли Session-Компонент компонентом с состоянием или без (с помощью `EJBMetaData.isStatelessSession()`). Метод возвращает значение true, если компонент не имеет состояния.

Определение интерфейса EJBMetaData приведено в разделе "Базовый класс EJBHome" на стр. 4-7.

Поддержка JNDI

Спецификация EJB определяет JNDI API как средство получения доступа к home-интерфейсу. JNDI реализован поверх других сервисов, включая Naming Service CORBA, LDAP/X.500, систем управления обычными файлами и специфических служб каталогов. На рис. 5.1 показаны различные варианты реализации. Как правило, разработчики Сервера EJB выбирают ту или иную реализацию JNDI.

Рис. 5.1 JNDI и примеры реализаций



Аспекты реализации JNDI не представляют интереса для клиента. Клиент использует только JNDI API.

Отображение EJB на CORBA

Существуют различные аспекты взаимоотношений между CORBA и Enterprise JavaBeans. Тремя наиболее важными из них являются следующие: взаимодействие Сервера/Контейнера EJB с ORB; интеграция унаследованных (legacy) систем в EJB; и организация доступа к Компонентам EJB для клиентов, не использующих Java. В настоящий момент в спецификации говорится только о третьем из них.

CORBA является чрезвычайно подходящей и естественной платформой, на которой может быть реализована инфраструктура EJB. Все проблемы EJB так или иначе затронуты в спецификациях ядра CORBA (CORBA Core specification) и ее сервисов (CORBA Services):

- Поддержка удаленного взаимодействия. CORBA Core и CORBA Naming service
- Поддержка транзакций. CORBA Object Transaction Service
- Обеспечение безопасности. CORBA Security specification, включая IIOP-over-SSL

Кроме того, CORBA позволяет интегрировать в приложение элементы, написанные не на Java, а других языках. Этими элементами могут быть унаследованные системы и приложения, а также различные виды клиентов. Использование различных языков, для которых существует отображение с OMG IDL, и OTS позволяет легко создавать многозвенные системы. Все это требует от Контейнера EJB поддержки OTS и IIOP API.

В спецификации EJB оговариваются вопросы взаимодействия клиентов, не использующих Java, с Компонентами EJB, и отображение EJB на CORBA. Задачами определения такого отображения являются:

- Поддержка взаимодействия между клиентами, написанными на любом языке, поддерживающем CORBA, с Компонентами EJB,

исполняемыми под управлением базирующегося на CORBA сервера EJB.

- Возможность совместного использования вызовов в стиле CORBA и RMI для обращения к Компонентам EJB в контексте одной транзакции.
- Поддержка распределенных транзакций, в контексте которых взаимодействуют несколько Компонентов EJB, исполняемых под управлением CORBA-совместимых EJB-Серверов от разных производителей.

Схема отображения основана на мэппинге Java-to-IDL. Спецификация содержит следующие части: отображение аспектов удаленного взаимодействия, отображение соглашения об именах, отображение транзакций и отображение аспектов безопасности. Все эти вопросы рассматриваются в последующих разделах. Так как схема отображения использует новые возможности IDL, связанные с поддержкой спецификации OMG Object-by-Value, обеспечение взаимодействия с другими языками программирования требует использования ORB, совместимого с CORBA 2.3.

Отображение аспектов удаленного взаимодействия

Компонент EJB имеет два интерфейса, к которым возможен удаленный доступ: remote- и home-интерфейсы. Результатом их отображении на CORBA будет набор соответствующих деклараций IDL. Таким же образом отображаются основные классы EJB.

Например, давайте рассмотрим IDL-интерфейсы для Session-Компонента EJB ATM, который содержит методы для перевода средств между счетами и возбуждает исключение в случае недопустимых состояний счетов. Результатом отображения его home- и remote-интерфейсов будут IDL-интерфейсы, показанные в Примере Кода 5.7.

Пример Кода 5.7 Интерфейсы IDL

```
module transaction {
   module ejb {

   valuetype InsufficientFundsException : ::java::lang::Exception {};

   exception InsufficientFundsEx {
      ::transaction::ejb::InsufficientFundsException value;
   };

   interface Atm : ::javax::ejb::EJBObject{
   void transfer (in string arg0, in string arg1, in float arg2)
      raises (::transaction::ejb::InsufficientFundsEx);
   };
```

```
    interface AtmHome : ::javax::ejb::EJBHome {

       ::transaction::ejb::Atm create ()
       raises (::javax::ejb::CreateEx);
    };
};};};};
```

Отображение имен

Среда исполнения EJB, которая базируется на CORBA и которая предоставляет доступ к Компонентам EJB как а CORBA-серверам, должна использовать CORBA Naming Service для обеспечения доступа к home-интерфейсам Компонентов. Эта среда может использовать Naming Service как явно, так и неявно - посредством JNDI и его стандартным отображением на CORBA Naming Service.

Имена JNDI представляют собой строки следующего вида: "directory1/directory2/.../directoryN/objectName". Служба Имен CORBA определяет имена как последовательность элементов имен.

```
typedef string Istring;

   struct NameComponent {
      Istring id;
      Istring kind;
   };
typedef sequence<NameComponent> Name;
```

Каждый фрагмент JNDI-имени между двумя "/" отображается в элемент имени CORBA; самый левый фрагмент соответствует первому элементу в последовательности составного имени CORBA Naming Service.

Имя JNDI является относительным именем относительно некоторого контекста имен, который мы называем root-контекстом JNDI. Этот контекст соответствует так называемому initial context CORBA Naming Service. CORBA-имена являются именами, относительными по отношению к CORBA initial context.

CORBA-программа получает initial CORBA Naming Service context как результат вызова метода псевдо-объекта ORB `resolve_initial_references("NameService")`. CORBA Naming Service не предусматривает наличия дерева имен с единственным корневым контекстом, поэтому термин "корневой контекст" в CORBA не используется. Какой именно контекст возвращает метод `resolve_initial_references()`, зависит от инициализации ORB.

В качестве примера рассмотрим фрагмент кода клиентской программы, написанной на C++, который обеспечивает получение ссылки на home-интерфейс Session-Компонента ATMSession, который был зарегистрирован в JNDI с именем "transaction/corbaEjb/atm". Прежде

всего, необходимо получить initial naming context.

```
Object_ptr obj = orb->resolve_initial_refernces("NameService");
NamingContext initialNamingContext= NamingContext.narrow( obj );
if( initialNamingContext == NULL ) {
   cerr << "Couldn't initial naming context" << endl;
   exit( 1 );
}
```

Следующий этап - создание имен для CORBA Naming Service в соответствие с рассмотренной ранее схемой мэппинга.

```
Name name = new Name( 1 );
name[0] id = "atm";
name[0] kind = "";
```

Теперь необходимо получить ссылку на объект, сопоставленный с именем name. Предполагается, что инициализация успешно выполнена и мы имеем контекст домена Компонентов EJB. После получения ссылки необходимо выполнить преобразование CORBA-объекта к нужному типу и проверить, успешно ли оно завершилось.

```
Object_ptr obj = initialNamingContext->resolve( name );
ATMSessionHome_ptr atmSessionHome = ATMSessionHome.narrow( obj );
if( atmSessionHome == NULL ) {
   cerr << "Couldn't narrow to ATMSessionHome" << endl;
   exit( 1 );
}
```

Отображение транзакций

Среда исполнения Компонентов EJB, которая базируется на CORBA и которая предоставляет возможность CORBA-клиентам участвовать в распределенных транзакциях вместе с Компонентами EJB, должна использовать для управления транзакциями CORBA Object Transaction Service.

В процессе поставки Компонента он должен быть установлен с теми или иными профилями транзакций (transaction policies). Значения профилей определены в Дескрипторе Поставки.

Для транзакционных Компонентов EJB определены следующие правила: CORBA-клиенты обращаются с ним при помощи стабов, сгенерированных на базе IDL-эквивалентов home- и remote-интерфейсов. Если клиент участвует в управлении транзакциями, он использует интерфейсы CORBA Object Transaction Service. Например, написанный на C++ клиент мог бы обратиться к Компоненту ATMSession из предыдущего примера следующим образом:

```
try {
   ...
   // obtain transaction current
   Object_ptr obj = orb->resolve_initial_refernces("Current");
```

```
    Current current = Current.narrow( obj );
    if( current == NULL ) {
       cerr << "Couldn't resolve current" << endl;
       exit( 1 );
    }
// execute transaction
    try {
       current->begin();
       atmSession->transfer("checking", "saving", 100 00 );
       current->commit( 0 );
    } catch( ... ) {
       current->rollback();
    }
}
catch( ... ) {
    ...
}
```

Отображение средств обеспечения безопасности

Аспекты обеспечения безопасности в спецификации EJB, которым уделено наибольшее внимание - это вопросы управления доступом к Компонентам. CORBA определяет несколько путей определения идентификации, включая следующие:

"Чистый" IIOP. Интерфейс principal CORBA признан устаревшим в начале 1998 года. Задачей интерфейса была идентификация (identity) клиента. Тем не менее, авторы Сервиса Безопасности CORBA реализовали другой подход - GIOP.

Спецификация GIOP содержит компонент, который называется service context и представляет собой массив пар значений. Идентификатор является типом CORBA::long, а значение представляет собой последовательность (sequence) октетов. Помимо других задач, контекст может быть использован для идентификации того, кто был инициатором вызова.

Безопасный (secure) IIOP. Спецификация обеспечения безопасности CORBA определяет некий "логический" тип данных для хранения информации об аутентификации. Его реальный тип зависит от выбранного конкретного механизма обеспечения безопасности, например, GSS Kerberos, SPKM или CSI-ECMA.

IIOP поверх SSL. SSL использует сертификаты X.509 для идентификации серверов и, возможно, клиентов. Когда сервер запрашивает сертификат клиента, он (сервер) может использовать этот сертификат как идентификатор клиента.

Глава

6

Написание Session-Компонента

В главе рассмотрены следующие основные темы:

- "Обзор Session-Компонентов" содержит введение в stateful и stateless session-Компоненты.
- "Цикл жизни stateful session-Компонента" описывает цикл жизни Компонента с момента создания до уничтожения.
- "Цикл жизни stateless session-Компонента" описывает цикл жизни stateless-Компонента.
- "Разработка Session-Компонента" описывает подробности реализации для всех типов Session-Компонентов.
- "Пример cart" демонстрирует код, который должен быть написан при разработке stateful session-Компонента.

Обзор Session-Компонентов

Класс Session-Компонента должен реализовать интерфейс SessionBean. Он также должен соответствовать всем требованиям и соглашениям об именах, касающихся отношений между этим классом и home- и remote-интерфейсами Компонента.

Кроме того, при создании Компонента необходимо реализовать все методы, для который объявлены соответствующие им методы в home- и remote-интерфейсах. Например, метод home-интерфейса `create()` сопоставлен с методом `ejbCreate()` в классе Компонента. Бизнес-методы, которые сделаны доступными для клиента путем их объявления в remote-интерфейсе, также имеют соответствующие им методы в классе Компонента.

Напоминаем, что существуют как stateful (с состоянием), так и stateless (не имеющие состояния) Session-Компоненты.

- Stateless session-Компонент. Это Компонент, который не сохраняет своего состояния как результата предыдущего вызова. Следствием этого является то, что один экземпляр такого Компонента может обслуживать несколько клиентов.
- Stateful session-Компонент. Такой Компонент сохраняет свое состояние между вызовами и различными транзакциями. Однажды полученная клиентом ссылка на Компонент используется в продолжение всего сеанса связи с этим клиентом. Конечно, клиент может иметь несколько различных сеансов и, следовательно, несколько экземпляров Компонента.

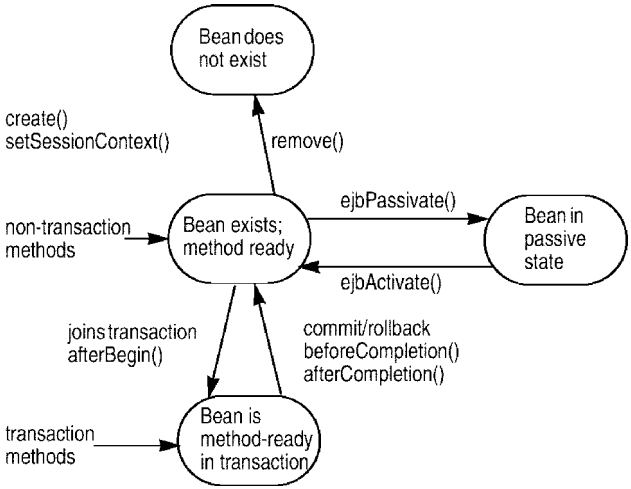
Цикл жизни stateful session-Компонента

Цикл жизни такого Компонента обычно состоит из следующих этапов и событий:

- В ответ на запрос клиента Контейнер создает новый объект.
- Экземпляр Компонента готов к выполнению запросов клиента. Такое его состояние называется "method ready state". Бизнес-методы Компонента могут выполняться как в контексте транзакции, так и вне любой транзакции - в зависимости от значения атрибутов в Дескрипторе Поставки и контекста транзакции клиента.
- Нетранзакционные методы session-Компонента выполняются экземпляром в состоянии "method ready".
- Экземпляр Компонента становится участником транзакции, когда клиент вызывает его транзакционный метод. Правильную работу с контекстом транзакции обеспечивает Контейнер. Завершение транзакции (подтверждение или откат) выполняется сервисом транзакций.
- В определенный момент Контейнер может выгрузить экземпляр Компонента из памяти. В этом случае выполняется деактивизация (passivation) Компонента с записью его состояния в хранилище. Session-Компонент не может быть выгружен, если он участвует в транзакции.
- Если клиент вызывает другой метод выгруженного Компонента, Контейнер снова загружает его в память. При этом происходит восстановление его состояния из хранилища, и Компонент готов выполнить запрос клиента.
- В ответ на вызов клиентом метода remove(), Контейнер уничтожает экземпляр Компонента. Он может сделать это и при истечении определенного интервала времени (по тайм-ауту).

На рис. 6.1 показан цикл жизни session-Компонента.

Рис. 6.1 Цикл жизни stateful session-Компонента



Обычно свое существование Компонент начинает в результате вызова клиентом метода `create()` его home-интерфейса. Реализацию home-интерфейса обеспечивает Контейнер, используя при этом класс Компонента. Контейнер создает новый экземпляр, инициализирует его и возвращает клиенту объектную ссылку. При этом Контейнер вызывает методы `setSessionContext()` и затем `ejbCreate()` класса Компонента. Разработчик Компонента может использовать эти методы для выполнения инициализации. В результате экземпляр переходит в состояние "method ready", что означает, что он готов выполнить нетранзакционный метод или присоединиться к транзакции. (Более подробные сведения о методах session-Компонентов находится в разделе "Интерфейс SessionBean" на стр. 6-5. См. раздел "Реализация session-Компонента" на стр. 6-7.)

Когда клиент вызывает метод `remove()` home- или remote-интерфейса, Контейнер вызывает соответствующий метод `ejbRemove()` объекта. Это позволяет разработчику Компонента выполнить требуемые действия непосредственно перед его удалением. После завершения вызова, объект переходит в состояние "nonexistent", т.е. "несуществующий". Если клиент попробует обратиться к объекту с таким состоянием, Контейнер возбудит исключительную ситуацию `java.rmi.NoSuchObjectException`.

Контейнер может деактивизировать экземпляр Компонента. Обычно это делается с целью оптимизации использования ресурсов, например, когда к экземпляру нет обращений в течение долгого времени или Контейнеру потребовалась дополнительная память. Когда происходит деактивизация объекта (ее также называют "passivation"), Контейнер записывает ссылку на объект и его состояние на диск, освобождая память, занимаемую объектом. Контейнер также обязан сохранить ссылки на файлы или другие ресурсы операционной системы, которые используются Компонентом. В процессе выполнения деактивизации Контейнер вызывает метод `ejbPassivation()`, а при выполнении

активации - ejbActivation(). Как правило, активизация выполняется в ответ на вызов клиентом метода Компонента по ссылке на него, находящейся у клиента. При этом Контейнер заново размещает объект в памяти.

Обратите внимание на то, что на процесс деактивизации влияет значение флага EJB passivation time-out. Если значение этого флага равно 0, то Контейнер никогда не выполняет деактивизацию объекта. Если его значение отлично от 0, то оно рассматривается как величина (в секундах) интервала времени, по истечении которого выполняется деактивизация объекта.

Когда клиент вызывает метод экземпляра Компонента в контексте транзакции, Контейнер либо начинает новую транзакцию, либо включает Компонент в уже существующую. В процессе выполнения транзакции предусмотрены специальные точки, который называются точками синхронизации транзакций (transaction synchronization points), когда экземпляр Компонента может получать сообщения о выполнении тех или иных событий и, соответственно, выполнить некоторый предварительные действия, если это необходимо. Это точки синхронизации определяются интерфейсом SessionSynchronization. (См. раздел "Session-интерфейс синхронизации" на стр. 6-6.) Session-Компонент, который хочет получать уведомление о таких событиях, обязан реализовать интерфейс SessionSynchronization. Реализовывать этот интерфейс не обязательно.

Реализация Контейнера EJB фирмой Inprise расширяет возможности архитектуры EJB в области обеспечения долговременного хранения информации. Обычно время жизни session-Компонента определяется временем жизни его Контейнера. Для большинства EJB-Контейнеров это означает, что после завершения работы Сервера (или JVM), под управлением которого выполнялся Контейнер, теряют смысл все ссылки на его экземпляры Компонентов EJB. Тем не менее, Inprise-Контейнеры позволяют выполнять деактивизацию с сохранением состояния объектов. Если это сделано, то объект может быть восстановлен в другом Контейнере. Такое поведение является предпочтительным, но поддерживается не всеми Контейнерами EJB.

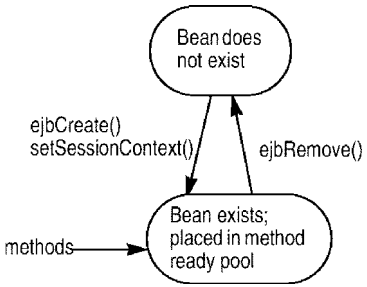
Цикл жизни stateless session-Компонента

В отличие от stateful-Компонента, всеми вопросами цикла жизни stateless session-Компонента ведает его Контейнер.

Цикл жизни такого Компонента очень прост, как видно на рис. 6.2, "Цикл жизни stateless session-Компонента". Когда Контейнер создает новый экземпляр такого Компонента, он вызывает методы setSessionContext() и ejbCreate() класса Компонента. Новый экземпляр помещается в пул таких объектов, и любой из них готов обслуживать запросы клиентов. Поскольку stateless-объекты не отслеживают своего состояния, Контейнер для выполнения запроса

клиента адресует этот запрос любому объекту из пула. При удалении Контейнером объекта из пула, он вызывает метод session-объекта `ejbRemove()`.

Рис. 6.2 "Цикл жизни stateless session-Компонента"



Инфо Действия по созданию и удалению экземпляров stateless session-Компонентов из пула не связаны с вызовом методов `create()` или `remove()` home-/remote-интерфейсов. Цикл жизни таких объектов определяется профилями Контейнера (container policies).

Разработка Session-Компонента

В этом разделе описан класс реализации Компонента и интерфейсы и методы, которые должны быть реализованы.

Интерфейс SessionBean

Интерфейс `SessionBean` session-Компонента определяет те методы, которые обязаны реализовать все session-Компоненты. Этот интерфейс наследует интерфейс `EnterpriseBean`. Методы этого интерфейса тесно связаны с циклом жизни такого объекта.

Пример Кода 6.1 Интерфейс SessionBean

```
package javax.ejb;
public interface SessionBean extends EnterpriseBean {
   void setSessionContext(SessionContext sessionContext)
      throws EJBException, RemoteException;
   void ejbRemove() throws EJBException, RemoteException;
   void ejbActivate() throws EJBException, RemoteException;
   void ejbPassivate() throws EJBException, RemoteException;
}
```

Методы выполняют следующие действия.

- `setSessionContext()` устанавливает контекст сеанса (сессии). Контейнер Компонента вызывает этот метод, чтобы ассоциировать экземпляр Компонента с контекстом сессии. Интерфейс

SessionContext объявляет методы для доступа к свойствам времени выполнения контекста, в котором выполняется сессия. Как правило, объект сохраняет этот контекст как часть своего состояния.

- `ejbRemove()` уведомляет session-объект о том, что он сейчас будет удален. По какой бы причине Контейнер не удалял экземпляр Компонента (например, вследствие вызова клиентом метода remove() home- или remote-интерфейсов), он вызывает этот метод класса Компонента.
- `void ejbActivate()` уведомляет объект о том, что он был активизирован.
- `void ejbPassivate()` уведомляет объект о том, что сейчас он будет деактивизирован Контейнером.

Уведомления о выполнении активизации и деактивизации позволяют реализовывать эффективные схемы управления ресурсами.

Session-интерфейс синхронизации

Необязательный интерфейс `SessionSynchronization` обеспечивает объекту возможность получения уведомлений о транзакционных действиях - другими словами, экземпляр получает сообщение о том, что какие-либо действия, связанные с транзакциями, либо только что произошли, либо сейчас будут выполнены. Экземпляры Компонентов могут использовать эти уведомления для выполнения некоторых действий с данными в БД. Например, реализация этого интерфейса позволяет объектам выполнить синхронизацию кешированных в памяти данных с данными в БД в процессе выполнения транзакции.

Только stateful session-Компоненты с сохранением, управляемым Контейнером (container-managed persistence, CMP), могут реализовывать интерфейс `SessionSynchronization`. Ни stateless-Компонентам, ни Компонентам с сохранением, управляемым Компонентом (bean-managed persistence, BMP) не следует пытаться реализовать этот интерфейс.

Пример Кода 6.2 Интерфейс SessionSynchronization

```
package javax ejb;
public interface SessionSynchronization {
   void afterBegin() throws RemoteException;
   void beforeCompletion() throws RemoteException;
   void afterCompletion(boolean completionStatus) throws
              RemoteException;
}
```

Контейнер использует эти три метода для уведомления экземпляра Компонента о начале транзакции или завершении ее этапов.

`afterBegin()` уведомляет объект о том, что новая транзакция началась. Контейнер вызывает этот метод перед вызовом первого бизнес-метода Компонента (не его remote-интерфейса!), но не обязательно сразу после

начала транзакции. Компонент уже включен в транзакцию, и любые действия выполняются в контексте этой транзакции.

`beforeCompletion()` уведомляет объект о том, что клиент завершил текущую транзакцию, но внесенные изменения еще не подтверждены. Если Компонент имеет кешированные в памяти значения данных из БД, то это подходящий момент для внесения изменений в базу данных. Если необходимо, объект может потребовать отката транзакции с помощью вызова метода setRollBackOnly() его контекста сессии.

`afterCompletion()` уведомляет объект о том, что текущая транзакция завершена. Аргумент completionStatus говорит о результате транзакции. Значение true говорит о том, что транзакции подтверждена (committed), false - что она откачена (rolled back).

Например, Контейнер вызывает метод `afterBegin()` после того, как клиент вызвал транзакционный бизнес-метод remote-интерфейса Компонента. Вызов бизнес-метода переводит объект в состояние "transaction ready". Через некоторое время, подтверждение транзакции приводит к вызову двух методов для объекта - сначала `beforeCompletion()`, а затем, если подтверждение успешно завершено, `afterCompletion(true)`. В случае отката транзакции Контейнер вызывает метод объекта `afterCompletion(false)`. (Обратите внимание, что в случае отката транзакции Контейнер не вызывает метод `beforeCompletion()`.) В любом случае, Компонент переходит в состояние "method ready".

Реализация session-Компонента

Класс session-Компонента (или его реализация) должен реализовать:

- Интерфейс `SessionBean`
- Методы, которые соответствуют методам home-интерфейса.
- Методы, которые соответствуют методам remote-интерфейса.

Интерфейс `SessionSynchronization` (для Session-Компонентов с CMP).

Мы уже рассматривали типичную реализацию методов интерфейса `SessionBean`. См. рис. 6.1 "Цикл жизни stateful session-Компонента" на стр.6-2 и рис.6.2 "Цикл жизни stateless session-Компонента" на стр. 6-4.

Home-интерфейс объявляет один или несколько методов `create()`. Для каждого из них в классе Компонента должен быть объявлен и реализован соответствующий метод. При этом используется определенное соглашение об именах - имена должны быть одинаковыми, за исключением префикса "ejb", добавляемого к имени метода класса. Таким образом, в home-интерфейсе объявлены методы `create()`, а в классе Компонента - соответствующие им методы `ejbCreate()`. Число и типы аргументов должны быть одинаковыми. Методы `ejbCreate()` всегда имеют результат типа void, в то время как методы `create()` home-интерфейса всегда возвращают remote-интерфейс.

Сигнатура методов ejbCreate() выглядит так:

```
public void ejbCreate( <zero or more parameters> ) {
   // implementation
}
```

Для методов ejbCreate() не требуется задавать throws-список возможных исключений, хотя в нем могут быть возбуждены как специфические для приложения исключения, так и стандартные, такие, как javax.ejb.CreateException или javax.ejb.EJBException.

В классе Компонента должны быть реализованы те бизнес-методы, для которых в remote-интерфейсе объявлены соответствующие им методы. Сигнатуры бизнес-методов в классе и в remote-интерфейсе должны полностью совпадать. Контроль соответствия типов берет на себя Контейнер. Не существует формализованных отношений между классом Компонента и его remote-интерфейсом (например, использования "implements"). Обычно Контейнер проверяет соответствие типов в процессе поставки Компонента. Возможна и проверка на этапе разработки, если ее обеспечивают используемые инструментальные средства.

Для класса session-Компонента конструктор не нужен, и вы можете не создавать его. Все необходимые действия по инициализации Компонента выполняются методом ejbCreate().

В качестве иллюстрации рассмотрим класс session-Компонента CartBean, показанный в Примере Кода 6.3. Приведенный фрагмент содержит только часть кода; это сделано для того, чтобы сфокусировать ваше внимание на том, что обязательно должно быть сделано.

Пример Кода 6.3 Пример реализации session-Компонента

```
import java util *;

public class CartBean implements javax.ejb.SessionBean {

   // variables
   private Vector _items = new Vector();
   private String _cardHolderName, _creditCardNumber;
   ...
   // required methods
   public void setSessionContext(javax.ejb.SessionContext
              sessionContext) {}

   public void ejbCreate(String cardHolderName, String
              creditCardNumber, Date expirationDate) {
      _cardHolderName = cardHolderName;
      _creditCardNumber = creditCardNumber;
      ...
   }
   public void ejbRemove() {}
```

```
    public void ejbActivate() {}
    public void ejbPassivate() {}

    // business methods
    public void addItem(Item item) {
       System.out.println("\taddItem(" + item getTitle() + "): " +
                this);
       _items addElement(item);
    }
    public void removeItem(Item item) throws ItemNotFoundException {
       ...
    }
    public float getTotalPrice() {
       ...
    }
    public java.util.Enumeration getContents() {
       ...
    }
    public void purchase()
       ...
    }
    ...
}
```

Обратите внимание на то, что класс CartBean не имеет конструктора. Он содержит один метод `ejbCreate()`, который используется для инициализации нового экземпляра Компонента cart. Класс содержит декларации других обязательных методов - `setSessionContext()`, `ejbRemove()`, `ejbActivate()` и `ejbPassivate()`. Тем не менее, все эти методы класса CartBean ничего не делают.

Пример cart

Демонстрирует использование карты покупки в электронной торговле. Вы выбираете покупку и помещаете ее в свою карточку. Вы можете на некоторое время прервать соединение с сайтом, затем вернуться и добавить новые покупки в вашу карту. В любой момент Вы можете посмотреть, что находится в списке покупок в вашей карте, а также общую стоимость. Наконец, вы окончательно проверяете список и фиксируете покупку.

Пример cart использует session statefule-Компонент.

Session stateful-Компонент

Для поддержки таких компонентов, Контейнер EJB использует высокопроизводительную архитектуру с использованием кеширования.

Создаются два основных пула объектов -"ready"-пул и пассивный пул. Если происходит тайм-аут, то компоненты могут перемещаться из первого во второй. Перемещение компонента в пассивный пул приводит к сохранению его состояния в управляемой Контейнером EJB Java-БД.

Файлы примера cart

Пример cart содержит несколько различных файлов. В этом разделе мы будем обращать внимание либо на те файлы, которые вы создаете сами, либо на те, которые иллюстрируют некоторые особенности session-компонентов. Некоторые из файлов, которые находятся в каталоге cart, сгенерированы автоматически (стабы, скелеты и другие CORBA-файлы) и не рассматриваются здесь.

Главным образом, мы будем изучать следующие файлы:

- Home-интерфейс Компонента, CartHome.java. Этот файл содержит определение Home-интерфейса для Компонента CartBean. Смотри раздел "Home-интерфейс cart" на странице 6-11.
- Remote-интерфейс Компонента, Cart.java. Этот файл содержит определение Remote-интерфейса для Компонента CartBean. Смотри раздел "Remote-интерфейс cart" на странице 6-12.
- Реализация Компонента cart, CartBean.java. Это собственно класс Компонента. Смотри "Компонент CartBean" на странице 6-13.
- Класс данных, Item.java. Этот файл используется Компонентом CartBean.
- Он содержит методы для получения стоимости и названия покупки, которую нужно поместить в карту. Смотри "Класс Item" на странице 6-18.
- Файл Дескриптора Поставки, Cart.xml. Спецификация EJB 1.1 требует, чтобы Дескриптор Поставки имел XML-формат. Смотри раздел "XML-файл Дескриптора Поставки" на странице 6-20. (Согласно спецификации EJB 1.1, должен присутствовать файл, который создает Дескриптор Поставки для CartBean в формате EJB 1.0, GenerateDescriptors.java - смотри раздел "Генерация Дескриптора Поставки" на странице B-1).
- Файлы исключений. В этих файлах содержатся определения специфических для приложения исключений, возбуждаемых в коде Компонента CartBean. Определены три класса таких исключений, каждый из которых находится в своем собственном файле: ItemNotFoundException, PurchaseProblemException и CardExpiredException. См. раздел "Исключения" на странице 6-19.
- Клиентская программа, CartClient.java. Это просто клиентское приложение.
- Make-файл для компиляции и построения Компонентов и клиентского приложения.

Home-интерфейс Компонета cart

Для Компонента EJB всегда определены два интерфейса: remote-интерфейс и home-интерфейс. Так, в нашем примере для session-Компонента, который называется CartBean, определены два public EJB-интерфеса: remote-интерфейс, который называется Cart, и home-интерфейс, который называется CartHome.

CartHome является home-интерфейсом для session stateful-Компонента CartBean. Как и все другие home-интерфейсы, он наследует интерфейс EJBHome. Хотя home-интерфейс предназначен для выполнения двух видов операций, создания экземпляра Компонента и поиска экземпляра Компонента, home-интерфейс session-Компонета может только создавать новые объекты. Для таких Компонентов не нужны средства поиска, поскольку session-Компонент является временным - по определению, экземпляр такого Компонента прекращает свое существование после завершения сеанса связи с клиентом. Только home-интерфейсы Entity-Компонентов объявляют операции поиска, поскольку такие Компоненты используются несколькими клиентами одновременно и существуют так долго, как сопоставленная с ними информация в базе данных. Пример Кода 6.4 содержит код для интерфейса CartHome.

Пример кода 6.4 Интерфейс CartHome

```
// CartHome java
public interface CartHome extends javax ejb EJBHome {
   Cart create(String cardHolderName, String creditCardNumber,
               java.util.Date expirationDate)
      throws java.rmi.RemoteException, javax.ejb.CreateException;
}
```

В нашем примере это очень простой интерфейс - он объявляет только один метод `create()`. Так как это stateful-Компонент, то могут присутствовать несколько методов `create()`, и они могут иметь список аргументов. В нашем случае, метод `create()` получает три аргумента: `cardHolderName`, `creditCardNumber` и `expirationDate`. Клиент вызывает метод `create()` для создания новой карты покупок, и Контейнер создает ее специально для этого пользователя. Клиент может работать с картой своих покупок с перерывами (кроме того, сервер может быть перезапущен в результате сбоя), но session-Компонент остается доступным для этого клиента до тех пор, пока пользователь не закончит работу и не удалит Компонент. Inprise Application Server (IAS) помещает состояние Компонента в базу данных. По умолчанию, для этой цели используется база данных JDataStore, хотя Вы можете использовать любую другую базу данных. Так как Компоненты помещаются в базу данных, они могут считаться хранимыми (persistent).

Инфо При работе с Inprise Application Server, в отличие от других продуктов EJB, stateful session-Компоненты сохраняются даже в случае сбоев сервера.

Такие Компоненты сохраняют свое состояние, оставшееся от предыдущего вызова, вне зависимости от того, выполняются они в контексте транзакции или нет. Под состоянием Компонента понимаются сопоставленные с ним данные, и они остаются сопоставленными с этим объектом в течение всего времени его жизни. Контейнер удаляет Компонент вместе с его состоянием из памяти по завершении сеанса работы с клиентом.

Метод `create()` должен соответствовать правилам, объявленным в спецификации EJB: он может возбуждать стандартное исключение RMI, `java.rmi.RemoteException`, а также исключение EJB `javax.ejb.CreateException`. Сигнатура метода create() должна соответствовать методу `ejbCreate()` в классе Компонента, то есть иметь одинаковое число и типы аргументов. Метод `create()` возвращает ссылку на remote-интерфейс Cart; это происходит потому, что интерфейс CartHome работает как фабрика Компонентов (типом возвращаемого значения метода `ejbCreate()` является void).

Remote-интерфейс Cart

Для компонента CartBean определен Remote-интерфейс Cart, который наследует интерфейс `EJBObject`, который является базовым интерфейсом для всех Remote-интерфейсов. Он позволяет Вам:

- Получать информацию о Компоненте. Вы можете проверить, является ли данный Компонент идентичным другому. Вы также можете получить значение главного ключа (primary key) для Entity-Компонента, но это не относиться к session-Компонентам.
- Получать объектную ссылку и/или идентификатор (handle) session-Компонента. Вы можете также получить ссылку на home-интерфейс Компонента. Вы можете сохранить значение идентификатора в базе данных и восстановить его оттуда позднее, а затем использовать как ссылку на экземпляр Компонента.
- Удалять экземпляры Компонента. Интерфейс `EJBObject` объявляет метод `remove()` для удаления экземпляра Компонента.

В дополнение к методам, унаследованным от интерфейса `EJBObject`, remote-интерфейс Cart содержит пять бизнес-методов. Реализация этих бизнес-методов содержится в классе Компонента CartBean; объявление их в интерфейсе Cart просто делает их доступными для клиента. Клиент может вызывать только те методы, которые объявлены в remote-интерфейсе. Вот эти методы:

- `addItem()` - позволяет добавить покупку в карту покупок.
- `removeItem()` - удаляет покупку из карты покупок.
- `getTotalePrice()` - устанавливает цену всех покупок и возвращает общую стоимость.
- `getContents()` - возвращает список всех покупок для просмотра и печати.

- `purchase ()` - выполняет попытку оформления покупки.

Код интерфейса Cart показан в Примере Кода 6.5.

Пример Кода 6.5 Remote-интерфейс Cart

```
// Cart java
public interface Cart extends javax.ejb.EJBObject {
   void addItem(Item item) throws java.rmi.RemoteException;
   void removeItem(Item item)
      throws ItemNotFoundException, java.rmi.RemoteException;
   float getTotalPrice() throws java.rmi.RemoteException;
   java.util.Enumeration getContents() throws
            java.rmi.RemoteException;
   void purchase()
      throws PurchaseProblemException, java.rmi.RemoteException;
}
```

Remote-интерфейс Cart следует всем правилам, определенным в спецификации EJB 1.1:

1 Интерфейс должен наследовать интерфейс `javax.ejb.EJBObject`.

2 Каждый метод remote-интерфейса должен соответствовать методу, реализованному в классе Компонента. Сигнатуры метода класса и метода Remote-интерфейса должны совпадать, что означает:

   - они имеют одинаковые имена.
   - они имеют одинаковое число и типы аргументов, а также тип возвращаемого значения.
   - список исключений метода класса должен быть подмножеством списка исключений метода remote-интерфейса.

3 Все методы должны возбуждать исключения `java.rmi.RemoteException`. Кроме того, они могут возбуждать другие, зависящие от приложения, исключения. Обратите внимание, как метод `removeItem()` возбуждает исключение ItemNotFoundException в добавление к `java.rmi.RemoteException`.

4 Типы всех аргументов и возвращаемых значений должны быть корректными типами данных Java RMI-IIOP.

Компонент CartBean

В этом разделе рассмотрены детали реализации Компонента CartBean. Общая структура и основные части этого класса показаны в примере Кода 6.3 на странице 6-8. Здесь мы рассмотрим класс более подробно.

Компонент EJB должен реализовать либо интерфейс `javax.ejb.SessionBean`, если он является session-Компонентом, или javax.ejb.EntityBean, если он является Entity-Компонентом. Поскольку наш объект CartBean является session-Компонентом, он реализует

интерфейс javax.ejb.SessionBean, как показано на примере:

```
public class CartBean implements javax.ejb.SessionBean {
```

Класс session-Компонента должен быть объявлен как public. Он не может быть объявлен ни как final, ни как abstract. Класс не определяет метод finalize().

Session-Компонент является обычным Компонентом языка Java, который может иметь свои собственные переменные (поля). CartBean имеет четыре такие переменные, и все они объявлены как private-переменные (См. пример Кода 6.6). Клиент не может обратиться к ним непосредственно; вместо этого чтение или изменение значения этих переменных выполняется посредством бизнес-методов Компонента и его метода create().

Пример Кода 6.6 Поля Компонента CartBean

```
private Vector _items = new Vector();
private String _cardHolderName;
private String _creditCardNumber;
private Date _expirationDate;
```

Переменная _items содержит список покупок, владельцем которых является объект cart. Он представляет собой тип "вектор" - другими словами, набор данных. Три остальные переменные - _cardHolderName, _creditCardNumber и _expirationDate - хранят значения аргументов метода create() home-интерфейса, а также метода ejbCreate() класса CartBean.

Обязательные методы

Session-Компонент должен реализовать четыре метода, которые объявлены в интерфейсе javax.ejb.SessionBean. Эти методы вызываются Контейнером EJB в определенные моменты цикла жизни экземпляра Компонента. Как минимум, разработчик Компонента должен обеспечить тривиальную (пустую) реализацию этих методов. Если необходимо, разработчик Компонента может добавить необходимые фрагменты кода в эти методы. Компонент CartBean не содержит никакого собственного кода. Методы интерфейса SessionBean показаны в Примере Кода 6.7.

Пример Кода 6.7 Методы, унаследованные от интерфейса SessionBean

```
public void setSessionContext(javax.ejb.SessionContext sessionContext) {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbRemove() {}
```

Контейнер вызывает метод setSessionContext() для того, чтобы сопоставить экземпляр Компонента с контекстом сессии. Компонент может сохранить ссылку на контекст сессии как часть своего состояния, но не обязан делать это. В нашем примере Компонент не сохраняет свой контекст сессии. Компонент может использовать контекст сессии для получения информации о самом себе, например, о параметрах среды или своем home-интерфейсе.

Контейнер вызывает метод ejbPassivate() объекта, когда он собирается перевести Компонент в пассивное состояние. Контейнер обеспечивает запись текущего состояния Компонента в хранилище перед выполнением деактивизации; он восстанавливает состояние при последующей активизации объекта. Поскольку Контейнер вызывает этот метод непосредственно перед выполнением активизации экземпляра, разработчик Компонента может добавить код в этот метод для выполнения некоторых действий, например, кеширования. В том же стиле Контейнер вызывает метод ejbActivate() непосредственно перед тем, как переводит объект из пассивного состояния в активное. Частью процесса активизации является восстановление сохраненного состояния Компонента. Если требуется, разработчик Компонента может добавить тот или иной код к методу ejbActivate(). Класс CartBean обеспечивает тривиальную реализацию обоих этих методов.

Session-Компонент не требует наличия конструктора. Вместо этого, Компонент реализует как минимум один метод ejbCreate(), который исполняет роль конструктора при создании нового экземпляра Компонента. Stateful-Компонент может содержать реализацию нескольких методов ejbCreate(); они отличаются друг от друга списком своих аргументов. В соответствие с требованием спецификации EJB, аргументы каждого метода ejbCreate() класса Компонента должны совпадать с аргументами метода create() home-интерфейса. Таким образом, home-интерфейс обязан иметь отдельный метод create() с тем же самым числом и типом аргументов для каждого метода ejbCreate(). Кроме того, не забывайте, что метод ejbCreate() всегда возвращает void, в то время как метод create home-интерфейса возвращает ссылку на remote-интерфейс. Наш класс CartBean содержит один метод ejbCreate() с тремя аргументами, как показано в Примере Кода 6.8:

Пример Кода 6.8 Метод ejbCreate() класса CartBean

```
public void ejbCreate(String cardHolderName, String creditCardNumber,
      Date expirationDate)  throws CreateException {
   _cardHolderName = cardHolderName;
   _creditCardNumber = creditCardNumber;
   _expirationDate = expirationDate;
}
```

Необходимо также объявить метод ejbRemove(). Контейнер вызывает этот метод непосредственно перед удалением экземпляра Компонента. Разработчик Компонента может добавить тот или иной фрагмент кода, который будет выполнен перед удалением, хотя это и не является обязательным. Класс CartBean реализует этот метод тривиальным образом.

Бизнес-методы

Класс session-Компонента обязан реализовать те бизнес-методы, которые объявлены в remote-интерфейсе. Существует несколько правил, которым должна следовать реализация бизнес-методов:

- имена методов не должны начинаться с "ejb" для того, чтобы избежать возможного конфликта имен с именами, зарезервированными архитектурой EJB.
- метод должен быть объявлен как public.
- метод не может быть объявлен как `final` или `static`.
- как аргументы, так и тип результата должны быть корректными типами RMI-IIOP.
- список исключений может содержать исключения `javax.ejb.EJBException`, а также произвольные исключения.

Для нашего примера cart, в remote-интерфейсе Cart объявлено пять методов. Реализация этих пяти бизнес-методов находится в классе CartBean. Сигнатуры (имя метода, число аргументов, тип аргументов и тип результата) методов класса и remote-интерфейса должны совпадать.

Класс CartBean реализует следующие методы: `addItem()`, `removeItem()`, `getTotalPrice()`, `getContents()` и `purchase()`. (Обратите внимание, что исключение java.rmi.RMIException признано устаревшим в спецификации EJB 1.1).

Для того, чтобы помочь вам отслеживать последовательность выполнения операторов программы, каждый бизнес-метод содержит строку кода, которая выводит на экран имя метода и описание того, что он делает.

Метод `addItem()` добавляет новые данные в набор данных, которые содержат список покупок карты:

Пример Кода 6.9 Метод addItem() класса CartBean

```
public void addItem(Item item) {
   System.out.println("\taddItem(" + item getTitle() + "): " + this);
   _items.addElement(item);
}
```

Метод `removeItem()` несколько более сложен. Программа выполняет цикл перебора элементов в списке для поиска того элемента, который нужно удалить, анализируя вид и название покупки. Кроме того, этот метод проверяет, действительно ли вы удаляете ту покупку, которую хотите удалить. Если покупка не найдена, происходит возбуждение исключения `ItemNotFoundException`.

Пример Кода 6.10 Метод removeItem()класса CartBean

```
public void removeItem(Item item) throws ItemNotFoundException {
   System.out.println("\tremoveItem(" + item getTitle() + "): " +
               this);
```

```
        Enumeration elements = _items.elements();
        while(elements.hasMoreElements()) {
           Item current = (Item) elements.nextElement();
           // items are equal if they have the same class and title
           if(item.getClass() equals(current.getClass()) &&
              item.getTitle() equals(current.getTitle())) {
              _items.removeElement(current);
              return;
           }
        }
        throw new ItemNotFoundException
           ("The item " + item getTitle() + " is not in your cart");
     }
```

Метод `getTotalPrice()` сначала устанавливает общую стоимость равной нулю, а затем выполняет перебор списка покупок, добавляя цену каждой покупки к общей цене. Он возвращает общую цену, округленную с точностью до цента.

Пример Кода 6.11 Метод getTotalPrice()класса CartBean

```
     public float getTotalPrice() {
        System.out.println("\tgetTotalPrice(): " + this);
        float totalPrice = 0f;
        Enumeration elements = _items.elements();
        while(elements.hasMoreElements()) {
           Item current = (Item) elements.nextElement();
           totalPrice += current.getPrice();
        }
        // round to the nearest lower penny
        return (long) (totalPrice * 100) / 100f;
     }
```

Все типы данных, передаваемых между клиентом и сервером, должны соответствовать требованиям *Java-сериализации*. Другими словами, они должны реализовывать интерфейс `java.io.Serializable`. В нашем примере, программа должна вернуть клиенту список покупок. Если бы не было ограничений сериализации, вы могли бы использовать метод `_items.elements()` для возвращения содержимого вектора покупок. Тем не менее, метод `_items.elements()` возвращает объект типа Java Enumeration, который не соответствует этим ограничениям. Для того, чтобы решить эту проблему, программа реализует класс, который называется `com.inprise.ejb.util.VectorEnumeration`. Этот класс получает вектор и возвращает обычное перечисление (enumeration), которое содержит данные этого вектора, и для которого может быть выполнена сериализация Java. Такой вектор может быть передан как от клиента, так и клиенту. Метод `getContents()`, показанный в Примере Кода 6.12, выполняет преобразование типа между `Java Enumeration` и классом `VectorEnumeration`.

Пример Кода 6.12 Метод getContents()класса CartBean

```
public java.util.Enumeration getContents() {
   System.out.println("\tgetContents(): " + this);
   return new com.inprise.ejb.util.VectorEnumeration(_items);
}
```

Метод purchase() выполняет следующие действия:

1 получает текущее время.

2 сравнивает текущее время с датой окончания действия карты. Если срок действия карты закончился, возбуждается исключение CardExpiredException.

3 метод завершает процесс покупки, включая перевод денег с карты на счет компании и инициализацию процесса доставки покупок (ни одно из этих действий на самом деле не реализовано). Если на любом из этих этапов происходит ошибка, процесс покупки не завершается и метод возбуждает исключение PurchaseProblemException.

Пример Кода 6.13 Метод purchase() класса CartBean

```
public void purchase()
   throws PurchaseProblemException {
   System.out.println("\tpurchase(): " + this);
   // make sure the credit card has not expired
   Date today = Calendar.getInstance().getTime();
   if(_expirationDate.before(today)) {
      throw new CardExpiredException("Expiration date: " +
               _expirationDate);
   }
   // complete purchasing process
   // throw PurchaseProblemException if error occurs
   }
}
```

В классе CartBean определен метод toString() для распечатки "CartBean" и имени владельца карты.

```
public String toString() {
   return "CartBean[name=" + _cardHolderName + "]";
}
```

Класс Item

Класс Item является public-классом. Он реализует интерфейс java.io.Serializable. Такие данные могут передаваться в качестве аргументов при удаленных вызовах.

Класс Item имеет два поля (title и price), два метода для чтения данных - getTitle() и getPrice() - и конструктор. Пример Кода 6.4 показывает код класса Item:

Пример Кода 6.14 Класс Item

```
// Item java
public class Item implements java.io.Serializable {

   private String _title;
   private float _price;
   public Item(String title, float price) {
      _title = title;
      _price = price;
   }
   public String getTitle() {
      return _title;
   }
   public float getPrice() {
      return _price;
   }
}
```

Исключения

В Примере cart определены три исключения. Все они являются наследниками стандартного класса `Java.Exception`. Все они показаны в Примере Кода 6.15.

Пример Кода 6.15 Исключения для примера cart

```
// ItemNotFoundException java
public class ItemNotFoundException extends Exception {
   public ItemNotFoundException(String message) {
      super(message);
   }
}
// PurchaseProblemException java
public class PurchaseProblemException extends Exception {
   public PurchaseProblemException(String message) {
      super(message);
   }
}
// CardExpiredException java
public class CardExpiredException extends PurchaseProblemException {
   public CardExpiredException(String message) {
      super(message);
   }
}
```

Метод `removeItem()` интерфейса cart возбуждает исключения `ItemNotFoundException`. Исключение `PurchaseProblemException`

возбуждается методом purchase(). На базе класса PurchaseProblemException создан производный класс CardExpiredException.

Инфо Исключения Java и производные от них исключения не поддерживались до реализации консорциумом OMG спецификацией передачи по значению (objects-by-value).

XML-файл Дескриптора Поставки

Спецификация EJB 1.1 требует, чтобы Дескриптор Поставки имел XML-формат. Дескриптор Поставки соответствует Document Type Definition (DTD), разработанному фирмой Sun Microsystems. Дескриптор Поставки содержит набор свойств, который описывает, как Контейнер будет выполнять процесс поставки Компонента или приложения.

Дескриптор Поставки включает набор тегов и атрибутов, чьи значения определяют состояние свойств Компонента. В качестве примера приведем несколько тегов для приложения cart:

- <session> - говорит о том, что Компонент является session-Компонентом (тег <entity> используется для обозначения Entity-Компонентов).
- Внутри области тега <session> могут использоваться другие теги:
  - <ejb-class> - имя класса реализации.
  - <home> - имя home-интерфейса.
  - <remote> - имя remote-интерфейса.
  - <session-type> - показывает, является ли session-Компонент stateful- или stateless-Компонентом.
  - <transaction-type> - показывает, используется ли для Компонента CMP или BMP.
- <trans-attribute> - задает значение атрибутов транзакции для каждого метода.
- <timeout> - значение тайм-аута для session-Компонента.

Пример Кода 6.16 XML-Дескриптор Поставки

```
<ejb jar>
   <enterprise beans>
      <session>
         <description>
            XML deployment descriptor created from file:
               D:\Kodiak\kodiak04\ejb_ea_0_4\examples\cart\cart.ser
         </description>
         <ejb-name>cart</ejb-name>
         <home>CartHome</home>
```

```
            <remote>Cart</remote>
            <ejb-class>CartBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
         </session>
      </enterprise-beans>
      <assembly-descriptor>
         <container-transaction>
            <method>
               <ejb-name>cart</ejb-name>
               <method-name>*</method-name>
            </method>
            <trans-attribute>NotSupported</trans-attribute>
            </container-transaction>
            <container-transaction>
               <method>
                  <ejb-name>cart</ejb-name>
                  <method-name>purchase</method-name>
               </method>
               <trans-attribute>Required</trans-attribute>
            </container-transaction>
         </assembly-descriptor>
</ejb-jar>
```

CartClient.java

Программа CartClient является клиентским приложением, которое использует Компонент Cart.

Начнем с изучения функции `main()`. Эта функция содержит элементы, которые должны быть реализованы в любом клиентском приложении. Здесь показано:

- как использовать JNDI для получения ссылки на home-интерфейс Компонента.
- как использовать метод `create()` home-интерфейса для создания нового удаленного объекта Cart.
- как обращаться к методам, объявленным для объекта Cart.

На рисунке 6.3 приведены наиболее интересные и важные фрагменты кода функции `main()` программы CartClient. Показаны только ключевые фрагменты кода. Цифры, приведенные слева от кода, обозначают номера шагов в описании, приведенном после рисунка. (Если вы хотите получить полный листинг программы, то вы можете найти его в каталоге примеров cart, который создается в процессе инсталляции).

Рис. 6.3 Функция main() программы CartClient

```
1 ———— public static void main(String[] args) throws Exception {
       // get a JNDI context using the Naming service
       javax naming Context context = new
                     javax.naming.InitialContext();

2 ———— Object objref = context.lookup("cart");
       CartHome home =
            (CartHome)javax.rmi.PortableRemoteObject.narrow(objref,
                           SortHome.class);
       Cart cart;
       {
          String cardHolderName = "Jack B. Quick";
          String creditCardNumber = "1234-5678-9012-3456";
          Date expirationDate = new GregorianCalendar(2001,
3 ————    Calendar.JULY,1).getTime();
          cart = home.create(cardHolderName, creditCardNumber,
                           expirationDate);
       }
       Book knuthBook = new Book("The Art of Computer Programming",
4 ————       49.95f);
       cart.addItem(knuthBook);
          ... // create compact disk and add it to cart, then list
                           cart contents
5 ———— summarize(cart);
6 ———— cart.removeItem(knuthBook);
       ... // add a different book and summarize cart contents
       try {
          cart.purchase();
       }
7 ————    catch(PurchaseProblemException e) {
             System.out.println("Could not purchase the
                           items:\n\t" + e);
       }
       cart.remove();
8 ————    }
       }
```

1 Функция `main()` начинается с использования контекста JNDI, необходимого для поиска объектов. Нужно создать первоначальный контекст - контекст службы имен Java. Это стандартный код JNDI.

2 Выполняется поиск объекта, реализующего home-интерфейс CartHome (этот объект называется *cart*). Поиск объекта по его JNDI-имени подразумевает обращение клиента к сервису CosNaming. Этот сервис возвращает клиенту объектную ссылку. В нашем примере, он возвращает объектную ссылку CORBA. Программа должна вызвать

операцию PortableRemoteObject.narrow() для преобразования полученной ссылки к типу CartHome с одновременным присвоением этого значения переменной с именем home. Такое преобразование является типичным для распределенных приложений. Этот вызов использует CORBA IIOP для выполнения следующих действий:

- установки связи с сервером.
- выполнения поиска на уровне службы CosNaming.
- получения объектной ссылки CORBA.
- возврата объектной ссылки клиенту.

3 Программа объявляет ссылку на удаленный объект cart, устанавливает значение трех параметров - имя пользователя, номер кредитной карты и срок действия кредитной карты - и создает новый удаленный объект cart.

4 Программа создает два объекта типа Покупка - книга и компакт-диск - и добавляет эти Покупки в карту покупок, используя вызов метода addItem().

5 Затем программа выводит список всех покупок. Для выполнения этого вызывается функция summarize(). Функция summarize() извлекает элементы или данные из карты, используя метод getContents(), который возвращает java Enumeration. Затем используются методы интерфейса Enumeration для чтения каждого элемента перечисления, и извлекаются название и цена каждой покупки. Пример Кода 6.17 показывает реализацию функции summarize():

Пример Кода 6.17 функция summarize() программы CartClient

```
static void summarize(Cart cart) throws Exception {
   System out println("======= Cart Summary ========");
   Enumeration elements = cart.getContents();
   while(elements.hasMoreElements()) {
      Item current = (Item) elements.nextElement();
      System.out.println
      ("Price: $" + current.getPrice() + "\t" +
         current.getClass().getName() + " title: " +
                        current.getTitle());
   }
   System.out.println("Total: $" + cart.getTotalPrice());
   System.out.println("============================");
}
```

6 Затем программа вызывает метод removeItem() для удаления покупки из списка покупок. Она добавляет другую покупку и опять суммирует результат.

7 Потом программа делает попытку оформить покупку. Так как эта операция не будет выполнена - она не реализована на сервере - программа возбуждает исключение PurchaseProblemException.

8 Здесь пользователь завершает сеанс работы, и программа удаляет объект cart.

Инфо Нет необходимости в явном удалении объекта cart. Здесь это сделано для ясности и для иллюстрации хорошего стиля программирования. Session-Компонент существует только для того клиента, который его создал; когда клиент завершает сеанс связи, Контейнер автоматически удаляет экземпляр Компонента. Контейнер также может удалить экземпляр Компонента по тайм-ауту, хотя это не происходит немедленно.

В программе CartClient также используется код, который создает на базе обобщенного класса Item (смотри "класс Item" на странице 6-18) два новых типа данных: "книга" и "компакт-диск". Book и CompactDisk являются конкретными классами, используемыми в нашем примере.

Пример Кода 6.18 Классы Book и CompactDisk

```
// CartClient java
import java.util.*;
class Book extends Item {
   Book(String title, float price) {
      super(title, price);
   }
}
class CompactDisc extends Item {
   CompactDisc(String title, float price) {
      super(title, price);
   }
}
```

Глава

7

Написание Entity-Компонента

В главе рассмотрены следующие основные темы:

- "Обзор Entity-Компонентов" содержит введение в Entity-Компоненты.
- "Управление сохранением состояния" объясняет два различных способа, с помощью которых Компоненты могут управлять сохранением состояния.
- "Этапы цикла жизни Entity-Компонента" описывает цикл жизни Компонента от момента создания до удаления.
- "Реализация Entity-Компонента" описывает необходимые детали его реализации.

Обзор Entity-Компонента

Entity-Компонент является объектным представлением данных, находящихся в долговременном хранилище, таком, как базы данных, или данных, реализованных с помощью существующего приложения. Они предназначены для того, чтобы несколько клиентов использовали их одновременно, в отличие от Session-Компонентов, которые всегда взаимодействуют только с одним клиентом.

Entity-Компонент соответствует записи (либо набору записей) в таблице реляционной базы данных или единичному объекту в объектно-ориентированной базе данных. Например, в случае использования реляционных баз данных, каждый столбец записи в таблице соответствует полям Entity-Компонента. Каждая запись в таблице идентифицируется с помощью главного ключа - одного или нескольких столбцов, которые уникальным образом идентифицируют каждую запись. Точно также, каждый Entity-Компонент содержит главный ключ (primary key) для идентификации конкретного экземпляра Компонента.

Хотя несколько клиентов могут взаимодействовать с одним Компонентом одновременно, Контейнер EJB управляет этим взаимодействием таким образом, чтобы сохранить целостность информации в базе данных. Контейнер может управлять конкурентным доступом к одному Компоненту, например, с помощью выстраивания запросов клиентов в очередь так, чтобы только один запрос выполнялся в каждое конкретное время. Контейнер может также делегировать управление конкурентным доступом средствам управления базами данных (DBMS) - то есть Контейнер только создает экземпляры Компонентов для каждого клиента, а затем полностью полагается на DBMS. Реализация Контейнера EJB фирмы Inprise использует этот последний подход.

Так как Entity-Компонент обычно является представлением информации в базе данных, его цикл жизни совпадает с циклом жизни самих данных - Entity-Компонент существует в течение долгого времени, обычно гораздо дольше, чем клиентское приложение, которое его создало. Такие Компоненты существуют не только после завершения сеанса связи или клиентского приложения, они существуют дольше, чем серверный процесс, в контексте которого они выполняются. Подобно данным в базе данных, Entity-Компонент не разрушается в результате сбоя сервера с последующим его перезапуском.

В отличие от Session-Компонентов, с которыми сопоставлено значение тайм-аута (и для которых Контейнер удаляет экземпляр Компонента при завершении периода тайм-аута), к Entity-Компонетам не применимо понятие тайм-аута. Вне зависимости от того, как долго они остаются неактивными, Контейнер не удаляет их из хранилища. Контейнер удаляет только экземпляры Компонента из самого Контейнера.

Единственный способ удалить Entity-Компонент - сделать это явно. Это может быть сделано с помощью вызова метода `remove()`, который удаляет как экземпляр Компонента, так и сопоставленную с ним информацию из базы данных, или с помощью средств DBMS (или других существующих приложений, не имеющих отношения к EJB).

Управление сохранением состояния

Под сохранением (Persistence) понимается протокол передачи состояния Компонента между экземпляром Компонента и сопоставленной с ним базой данных. Работая с Entity-Компонентами, разработчик может выбрать способ реализации сохранения.

Разработчик Компонента может поместить код, необходимый для сохранения состояния, непосредственно в класс Компонента (или в любой другой класс, поставляемый вместе с Компонентом). Обычно это называется bean-managed persistence (BMP).

С другой стороны, разработчик Компонента может доверить управление сохранением его состояния Контейнеру EJB. Средства, входящие в состав Контейнера, определяют способ управления сохранением на стадии поставки. Это называется container-managed persistence (CMP).

Сохранение, управляемое Компонентом

Entity-Компонент с BMP содержит код, явно управляющий доступом к базе данных. Другими словами, разработчик Компонента помещает команды обращения к базе данных непосредственно в код Компонента или другие, связанные с ним, классы. Обычно эти вызовы используют JDBC.

Операторы доступа к базе данных могут появиться как в коде бизнес-методов, так и в одном из следующих стандартных методов интерфейса - `ejbCreate()`, `ejbRemove()`, `ejbLoad()` и `ejbStore()` - или в одном из finder-методов `ejbFind<methodname>`, который объявлен в home-интерфейсе Компонента. (Методы интерфейса Entity-Компонента обсуждаются в разделе "Методы Entity-Компонента" на странице 7-9).

В общем случае Компонент с BMP реализовать значительно труднее, поскольку его разработчик должен сам писать дополнительный код. Кроме того, вследствие помещения кода доступа к базе данных непосредственно в тело методов Компонента, гораздо труднее настроить Компонент на работу с различными базами данных.

Несмотря на то, что существует много причин не использовать BMP, все же встречаются ситуации, когда это может оказаться полезным. Entity-Компоненты с BMP не требуют дополнительной поддержки со стороны Контейнеров EJB. Кроме того, они могут управлять ситуацией в стиле, который не доступен Контейнерам EJB.

Сохранение, управляемое Контейнером

В этом случае сам Компонент не содержит кода доступа к базам данных. Разработчик Компонента не пишет такой код и не помещает его в тело методов. Вместо этого, Компонент полностью полагается на Контейнер EJB.

Инструментальные средства Контейнера генерируют команды обращения к базам данных на этапе поставки Компонента, то есть тогда, когда Компонент помещается в Контейнер. Эти средства используют Дескриптор Поставки, чтобы определить имена полей объекта, для которых они должны генерировать команды взаимодействия с базами данных. Вместо того, чтобы помещать код обращения к базам данных непосредственно в Компонент (как это происходит в случае использования BMP), разработчик Компонента с CMP должен указать все необходимые поля в Дескрипторе Поставки. Контейнер содержит достаточно "разумные" средства, чтобы автоматически сопоставить поля Entity-Компонента с информацией в базе данных.

CMP имеет много преимуществ по сравнению с BMP. Ее проще использовать - разработчик Компонента сам не обращается к методам управления базами данных. Управление сохранением состояния может быть изменено без изменения и перекомпиляции исходного кода Компонента. Deployer или Application Assembler могут выполнить это

путем изменения параметров в Дескрипторе Поставки при поставке Компонента. Перенос управления взаимодействием с базами данных и сохранением состояния на Контейнер не только уменьшает сложность кода Компонента, но также локализует возможные ошибки. Разработчик Компонента может сосредоточить все свое внимание на разработке и отладке бизнес-логики Компонента и не думать о проблемах системного уровня.

Тем не менее, с CMP сопоставлены некоторые ограничения. Существует потенциальная проблема, если поля различных Entity-Компонентов с CMP отображаются на одни и те же данные в базе данных (хотя в правильно написанных программах такой ситуации следует избегать). В случае, подобном этому, различные Entity-Компоненты могут видеть недостоверные данные, если они используются в одной и той же транзакции. Кроме того, при использовании CMP Контейнер может загрузить полное состояние Компонента до того, как вызван метод `ejbLoad()`. Это может привести к снижению производительности в том случае, если с Компонентом сопоставлено большое количество данных.

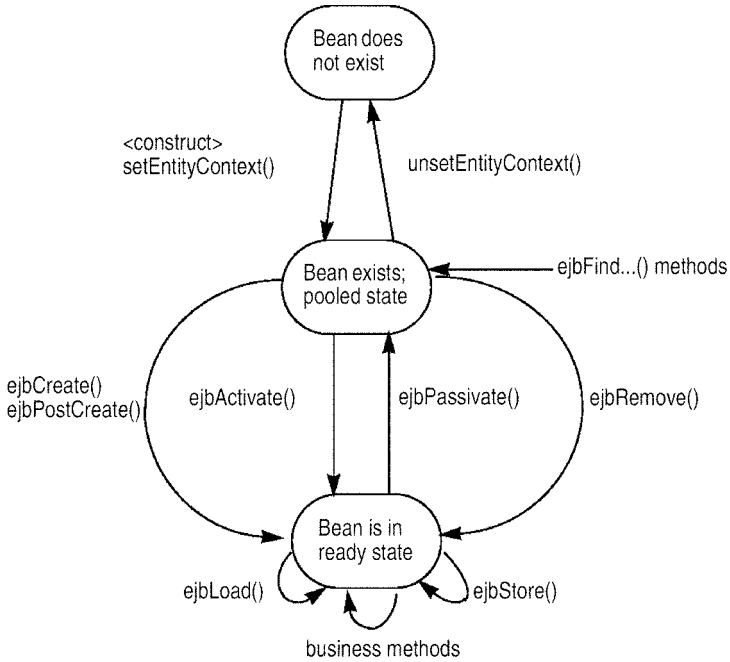
Этапы цикла жизни Entity-Компонента

В процессе существования Компонент может находиться в трех различных состояниях:

- Nonexistent. Экземпляр Entity-Компонента не существует.
- Pooled. Экземпляр Entity-Компонента существует, но не сопоставлен ни с какими конкретными данными (говорят, что он не имеет identity).
- Ready. Экземпляр Entity-Компонента сопоставлен с конкретными данными.

На рисунке 7.1 изображен цикл жизни Entity-Компонента. Каждый Компонент должен наследовать интерфейс `javax.ejb.EntityBean`. Контейнер EJB использует методы этого интерфейса для уведомления экземпляра Компонента о происходящих событиях. См. раздел "Класс Entity-Компонента" на странице 7-7 для получения дополнительной информации. Методы, приведенные на диаграмме цикла жизни, объявлены в интерфейсе EntityBean.

Рис. 7.1 Цикл жизни экземпляра Entity-Компонента



Контейнер создает экземпляр путем его конструирования. Затем он вызывает метод `setEntityContext()` для передачи экземпляру ссылки на этот контекст, то есть ссылки на интерфейс `EntityContext`. Этот интерфейс обеспечивает возможность доступа к сервисам Контейнера и позволяет экземплярам Компонента получить информацию об их клиентах.

Сейчас экземпляр Entity-Компонента находится в состоянии "Pooled". Для каждого типа Компонента создается свой собственный пул. Ни один из экземпляров, находящихся в пуле, не сопоставлен с конкретными данными. Другими словами, они не имеют identity - ни одно из их полей не имеет установленных значений - и, следовательно, все они эквивалентны. Контейнер может выбрать любой экземпляр для обслуживания клиента, который решил использовать такой Компонент. Реализация Контейнера определяет, когда именно создавать экземпляры Компонентов и как много таких экземпляров следует поместить в пул.

Когда клиентское приложение вызывает один из методов поиска, Контейнер выполняет код соответствующего метода `ejbFind()` для произвольного экземпляра Компонента с состоянием pooled. В процессе выполнения метода поиска экземпляр не переходит в состояние ready.

Когда Контейнер выбрал экземпляр для обслуживания запроса клиента, этот экземпляр переходит из состояния pooled в состояние ready. Существуют два способа перевода экземпляра Компонента из первого состояния во второе:

- с помощью вызовов методов ejbCreate() или ejbPostCreate().
- с помощью вызова метода ejbActivate().

Контейнер может выбрать экземпляр для обслуживания вызова клиентом метода create() home-интерфейса. В ответ на этот вызов, Контейнер создает объект и вызывает методы ejbCreate() и ejbPostCreate(). Контейнер вызывает метод ejbActivate() для активизации экземпляра, чтобы он мог обслуживать вызовы конкретных клиентов. Обычно Контейнер использует метод ejbActivate(), когда нет подходящих экземпляров в состоянии ready для обслуживания клиентских запросов.

Когда экземпляр находится в состоянии ready, он сопоставлен с конкретным Entity-объектом (например, с определенной записью в базе данных). Клиент может вызвать бизнес-методы этого Entity-Компонента. Контейнер использует методы ejbLoad() и ejbStore() для того, чтобы заставить Компонент считывать или записывать свои данные. Все эти методы - бизнес-методы, ejbLoad() и ejbStore() могут быть вызваны много раз, а могут и ни одного, в соответствии с логикой клиентского приложения. Методы ejbLoad() и ejbStore() позволяют экземпляру Компонента выполнять синхронизацию его состояния с информацией в базе данных.

Когда экземпляр Entity-Компонента переходит в состояние pooled, он отсоединяется от конкретных данных (от собственно Entity-Компонента). Теперь Контейнер может сопоставить этот экземпляр с любым Entity-объектом. Существуют два способа, с помощью которых Контейнер переводит экземпляр Компонента из состояния ready в состояние pooled.

- Контейнер деактивизирует экземпляр Entity-Компонента с помощью вызова метода ejbPassivate(). Он использует этот метод, чтобы отделить этот экземпляр от данных без удаления Entity-объекта, то есть собственно этих данных. Обычно Контейнер выполняет деактивизацию при завершении транзакции. Тем не менее, он может выполнить деактивизацию даже тогда, когда экземпляр Компонента сопоставлен с транзакцией; в этом случае он сначала вызывает метод ejbStore(), чтобы экземпляр мог синхронизировать свое состояние с состоянием базы данных. Когда позднее происходит активизация экземпляра, Контейнер автоматически вызывает метод ejbLoad(), чтобы опять-таки выполнить синхронизацию между экземпляром и базой данных.
- Контейнер удаляет Entity-объект путем вызова метода ejbRemove(). Вызов этого метода происходит в ответ на вызов клиентским приложением метода remove() home- или remote-интерфейса Компонента. Контейнер удаляет как экземпляр Компонента, так и сам Entity-объект (запись из базы данных).

Контейнер удаляет не сопоставленные с данными экземпляры из пула с помощью вызова метода unsetEntityContext() для этого экземпляра.

Реализация Entity-Компонента

Реализация Entity-Компонента очень похожа на реализацию Session-Компонента. Вы должны реализовать методы home-интерфейса, remote-интерфейса и создать собственно класс Компонента. Этот класс должен содержать методы, которые соответствуют методам, объявленным в home- и remote-интерфейсах.

Базовым классом для класса Entity-Компонента является класс `EntityBean`, а не `SessionBean`.

Обратитесь к разделу "Remote-интерфейс" на странице 7-16 для получения информации о разработке remote-интерфейса.

Обратитесь к разделу "Home-интерфейс Entity-Компонента " на странице 7-15 для получения информации о разработке home-интерфейса Entity-Компонента. Вам следует предварительно ознакомиться с материалом, посвященному созданию Session-Компонентов (глава 6 "Написание Session-Компонентов").

Класс Entity-Компонента должен реализовать:

- Интерфейс `EntityBean`.
- Методы, которые соответствуют методам home-интерфейса - в частности, create-методам и, для Компонентов с BMP, finder-методам.
- Методы, которые соответствуют методам remote-интерфейса - бизнес-методы и методы интерфейса `EntityBean`.

Класс EntityBean также содержит реализацию бизнес-методов, которые являются специфическими для этого Entity-объекта. Дополнительно, для Компонента с BMP, он должен включать в себя методы для доступа и изменения информации в базе данных.

Класс Entity-Компонента

Класс Entity-Компонента наследует класс `javax.ejb.EntityBean` и реализует его методы. Контейнер использует методы этого интерфейса для уведомления экземпляров Компонента о происходящих событиях. Интерфейс EntityBean является аналогом интерфейса `SessionBean`.

Когда Entity-Компонент использует CMP, методы интерфейса `EntityBean` (за исключением двух, связанных с контекстом) и метод `ejbPostCreate()` служат callback-методами. Это означает, что класс Компонента должен обеспечивать реализацию только их основных функций. Хотя это и не требуется, разработчик Компонента может добавить к этим методам код, специфический для приложения.

Entity-Компонент с BMP должен обеспечивать более полную реализацию тех же самых методов. Поскольку такой Компонент полностью берет на себя вопросы взаимодействия с базой данных, он должен корректно управлять чтением данных из БД, модификацией

данных и удалением Entity-объекта из базы данных. Это обеспечивается надлежащей реализацией методов интерфейса EntityBean - ejbLoad(), ejbStore(), ejbRemove() - и метода ejbCreate().

Определение интерфейса EntityBean приведено в Примере Кода 7.1

Пример Кода 7.1 Определение интерфейса EntityBean

```
package javax.ejb;
import java.rmi.RemoteException;

public interface EntityBean extends EnterpriseBean {
   public void setEntityContext (EntityContext ctx) throws
               EJBException, RemoteException;
   public void unsetEntityContext()throws EJBException,
               RemoteException;
   public void ejbRemove()throws RemoveException, EJBException,
               RemoteException;
   public void ejbActivate() throws EJBException, RemoteException;
   public void ejbPassivate() throws EJBException, RemoteException;
   public void ejbLoad() throws EJBException, RemoteException;
   public void ejbStore()throws EJBException, RemoteException;
}
```

Интерфейс содержит следующие методы.

- setEntityContext() - устанавливает контекст Компонента. Контейнер использует этот метод для передачи ссылки на интерфейс EntityContext экземпляру Компонента. Этот интерфейс содержит методы, которые позволяют получить доступ к свойствам среды исполнения Компонентов. Компонент, который использует этот контекст, должен хранить эту ссылку в одном из своих полей.
- unsetEntityContext() - вызывается Контейнером перед тем, как происходит уничтожение текущего экземпляра Entity-Компонента. Здесь Компонент может освободить ресурсы, которые были захвачены в процессе выполнения кода метода setEntityContext().
- ejbRemove() - удаляет запись или записи из базы данных, ассоциированные с этим Entity-Компонентом. Контейнер вызывает этот метод в ответ на вызов клиентом метода remove().
- ejbActivate() - уведомляет Компонент о том, что он только что был активизирован. Контейнер вызывает этот метод для экземпляра, выбранного из пула доступных экземпляров, и сопоставляет с ним identity конкретного Entity-объекта. В процессе выполнения активизации экземпляр Entity-Компонента имеет возможность затребовать необходимые дополнительные ресурсы.
- ejbPassivate() - уведомляет Компонент о том, что готовится его деактивизация - другими словами, экземпляр будет отсоединен от конкретного Entity-объекта (от конкретных данных в базе данных) и

возвращен в пул доступных экземпляров. В этом месте экземпляр имеет возможность освободить ресурсы, которые были захвачены в процессе выполнения метода `ejbActivate()` и которые не имеет смысла хранить при попадании объекта в пул.

- `ejbLoad()` - обновляет данные Компонента, считывая их из базы данных. Контейнер вызывает этот метод для экземпляра Компонента для выполнения синхронизации состояния полей экземпляра с состоянием информации в базе данных.
- `ejbStore()` - помещает данные из полей Компонента в базу данных. Контейнер вызывает этот метод для экземпляра Компонента, чтобы синхронизировать состояние базы данных с кешированными значениями полей Компонента.

Методы Entity-Компонента

Для Entity-Компонента обычно объявлены и реализованы create- и finder-методы.

Методы для создания (create-методы)

Для Entity-Компонента должны быть объявлены и реализованы методы `ejbCreate()` и `ejbPostCreate()`, которые соответствуют методам `create()`, объявленным в home-интерфейсе. Помните, что класс Компонента не обязан реализовывать create-методы. Результатом вызова create-метода является добавление новой информации в базу данных. Вы можете создать Entity-Компонент, не имеющий таких методов, когда новые данные в БД (т.е. новые Entity-объекты) добавляются с помощью средств DBMS или унаследованных систем.

Имена create-методов в классе реализации Компонента должны совпадать с именами соответствующих методов в home-интерфейсе, за исключением префикса "ejb". Списки аргументов (как число, так и их тип) должны быть одинаковыми.

Вызов метода `ejbCreate()` для Entity-Компонента приводит к добавлению нового Entity-объекта в базу данных. Хотя методы `create()` home-интерфейса возвращают ссылку на remote-интерфейс, соответствующие им методы `ejbCreate()` класса Компонента возвращают значение primary key (главного ключа). Для Entity-Компонента с CMP результатом метода `ejbCreate()` является `null`. Это происходит потому, что за создание объекта полностью отвечает Контейнер. Для Entity-Компонента с BMP метод `ejbCreate()` возвращает экземпляр класса primary key для вновь созданного Entity-объекта. Контейнер использует это значение для создания экземпляра Entity-Компонента.

Сигнатуры методов `ejbCreate()` для Entity-Компонентов не зависят от типа управления сохранением (т.е. BMP или CMP) и имеют следующий вид:

```
public <PrimaryKeyClass> ejbCreate( <zero or more parameters> )
// implementation
```

```
}
```

Обратите внимание, что сигнатура метода ejbCreate() для Entity-Компонента требует возврата класса primary key. Entity-Компоненты с CMP не использует при возврате значение primary key, т.к. на самом деле новый экземпляр Компонента создается Контейнером. Вместо этого, разработчик Компонента в качестве значения результата метода ejbCreate() возвращает null.

Метод ejbCreate() вызывается Контейнером в ответ на вызов клиентом метода create(), и новые данные, представлением которых является вновь созданный объект, записываются в базу данных. Затем Контейнер вызывает соответствующий метод ejbPostCreate() для того, чтобы позволить экземпляру Компонента завершить процесс инициализации. Метод ejbPostCreate() должен соответствовать методу ejbCreate() по списку аргументов, но не по типу результата (он возвращает тип void). Вот его сигнатура:

```
public void ejbPostCreate( <zero or more parameters> ) throws
                    RemoteException {
   // implementation
}
```

Методы ejbCreate() обычно используются для выполнения инициализации, поэтому они часто имеют те или иные аргументы и их реализация содержит код, который использует значения аргументов для установки начального состояния Entity-объекта. Например, пример bank для работы со счетами использует Entity-Компонент, чей метод ejbCreate() имеет два аргумента типов string и float. Значение string-аргумента используется для задания названия счета, а аргумент типа float - для задания его состояния, как показано в Примере Кода 7.2:

Пример кода 7.2 Метод ejbCreate() примера Bank

```
public AccountPK ejbCreate(String name, float balance) {
   this name = name;
   this balance = balance;
   return null;
}
```

Методы поиска (finder-методы)

Entity-Компонент с BMP должен также обеспечить реализацию методов поиска, которые соответствуют методам поиска home-интерфейса. Имена таких методов в классе реализации и в home-интерфейсе должны совпадать, за исключением префикса "ejb". Списки аргументов - как количество, так и тип - должны быть одинаковыми.

Entity-Компонент с CMP не реализует методы поиска, соответствующие методам поиска home-интерфейса. Для таких Компонентов реализацию finder-методов обеспечивает Контейнер. Тем не менее, чтобы Контейнер мог это сделать, вы должны поместить определенную информацию в Дескриптор Поставки. Реализация Inprise Контейнера EJB

предусматривает для этого специальные поля.

Home-интерфейс обязан объявить базовый метод поиска, `findByPrimaryKey(primaryKey)`, для выполнение поиска объекта по значению его главного ключа. Метод имеет единственный аргумент - значение ключа - и возвращает remote-интерфейс для найденного объекта:

```
public <entity bean's rem te interface> findByPrimaryKey(
   <primary key type> key )
   throws java.rmi.RemoteException, javax.ejb.FinderException;
```

Home-интерфейс может объявлять и дополнительные методы поиска. Для каждого метода, включая `findByPrimaryKey()`, в классе Компонента должна присутствовать соответствующая реализация (при использовании BMP). Помимо методов `ejbCreate()`, класс Компонента содержит метод `ejbFindByPrimaryKey()` и методы `ejbFind<methodname>()`, которые соответствуют методам поиска home-интерфейса.

Метод поиска может возвращать как один, так и несколько объектов. В любом случае, при использовании CMP метод возвращает тип `java.util.Collection` для Java 2 и `java.util.Enumeration` для более ранних версий JDK. При извлечении из `Collection` - или `Enumeration` каждого отдельного элемента необходимо выполнить его преобразование к типу remote-интерфейса.

Если же используется CMP и метод поиска возвращает несколько объектов, то это должна быть `Collection` - или `Enumeration` - главных ключей, который Контейнер преобразует к набору ссылок на remote-интерфейсы.

Бизнес-логика

Класс Entity-Компонента должен также обеспечить реализацию тех бизнес-методов, которые объявлены в remote-интерфейсе. Сигнатуры методов в классе и сигнатуры методов в remote-интерфейсе должны полностью совпадать.

Нет никакой необходимости определять конструктор для класса Entity-Компонента. Все необходимые действия по инициализации Компонента выполняются методами `ejbCreate()`.

Реализация Entity-Компонента с CMP должна объявлять как public те переменные, которые должны управляться Контейнером. В случае BMP переменные могут быть объявлены как public, так и private.

Синхронизация методов

Эта тема затрагивает вопросы одновременного доступа к Entity-Компонентам и создания реентерабельных (reentrant) Компонентов.

Одновременный доступ к Entity-Компонентам

Проблемами синхронизации вызовов методов при одновременном доступе к Компонентам занимается Контейнер, и разработчику Компонента (bean provider) нет необходимости беспокоиться об этом. Контейнер обычно использует одну из двух стратегий: синхронизация на уровне баз данных или синхронизация на уровне Контейнера.

При синхронизации на уровне баз данных, Контейнер просто создает несколько экземпляров одного и того же Компонента, а вопросы синхронизации решаются при вызовах к БД, выполняемых из методов `ejbCreate()`, `ejbRemove()`, `ejbLoad()` и `ejbStore()`.

При синхронизации на уровне Контейнера выполняется сериализация обращений к экземплярам Entity-Компонентов (т.е. выполнение их одного за другим). Контейнер создает только один экземпляр Компонента в каждый момент времени, и этот экземпляр получает права исключительного доступа к базе данных. Процесс создания экземпляров и вызова их методов выполняется в режиме очереди, и вызов последующего метода происходит только после завершения предыдущего.

Реентерабельные Entity-Компоненты

По умолчанию к Entity-Компонентам нельзя обращаться повторно в контексте одной транзакции - это приводит к возникновению исключения `java.rmi.RemoteException`.

Вы можете объявить, что Компонент является реентерабельным с помощью установки параметров Дескриптора Поставки; тем не менее, при этом нужно быть очень осторожным. Главная проблема состоит в том, что Контейнер в общем случае не может отличить повторный вызов в контексте одной транзакции и конкурентный вызов метода того же самого Компонента (в контексте той же транзакции).

К одному и тому же экземпляру Компонента, помеченного как реентерабельный, нельзя обращаться одновременно нескольким клиентам в контексте одной транзакции. Обеспечить выполнение этого требования - обязанность программиста.

Главные (первичные) ключи Entity-Компонента

Каждый экземпляр Entity-Компонента должен иметь сопоставленный с ним главный ключ. Главный ключ представляет собой значение (или комбинацию значений), которое уникальным образом идентифицирует данный экземпляр. Например, в таблице базы данных, которая содержит записи о сотрудниках предприятия, в качестве главного ключа может использоваться номер социального страхования. Представляется естественным, если Entity-Компонент, являющийся объектным представлением записи в таблице, также будет использовать это значение в качестве своего главного ключа.

Применительно к Компонентам EJB главный ключ реализуется с помощью Java-класса, содержащего уникальное значение. Этим классом может быть любой корректный RMI-IIOP - класс. В частности, это означает, что он реализует интерфейс `java.io.Serializable`. Этот класс должен также обеспечить реализацию методов `Object.equals(Object other)` и `Object.hashCode()`, т.е. двух методов, которые по определению наследуют все Java-классы.

Класс главного ключа является специфическим для приложения. Другими словами, каждый Entity-Компонент может иметь свой собственный класс главного ключа. С другой стороны, нескольким Компонентам разрешено использовать один и тот же класс.

Пример bank использует два различных Entity-Компонента для представления различных видов счетов. Оба вида счета используют одно и то же поле в качестве ключевого - для уникальной идентификации записи в таблице. По этой причине один и тот же класс, AccountPK, использовался в качестве класса primary key для обоих видов счетов. Определение этого класса показано в Примере Кода 7.3:

Пример Кода 7.3 Главный ключ класса Account

```
public class AccountPK implements java.io.Serializable {
   public String name;
   public AccountPK() {}
   public AccountPK(String name) {
      this name = name;
   }
}
```

Управление транзакциями с оптимистичной схемой блокировок

Inprise-реализация Контейнера EJB не выполняет блокировок Entity-Компонентов для выполнения транзакций. Это означает, что несколько различных транзакций, выполняемых параллельно в интересах различных клиентов, обращаются к "логически" одному и тому же Entity-объекту, но к различным "физическим" копиям данных. Это может вызвать проблемы при работе с большинством СУБД.

Для примера предположим, что три различные транзакции одновременно обращались к Entity-Компоненту SavingsAccount с целью модификации записи в таблице счетов Account. Если все они пробовали изменить значение одного и того же поля, то одна транзакция могла бы перезаписать результаты, только что подтвержденные другими транзакциями без анализа того, что было сделано. В общем случае это нельзя считать правильным решением. С другой стороны, если изменения вносились в разные поля одной и той же записи, то только одна транзакция была бы завершена успешно, в то время как две другие были бы откачены.

Контейнер Inprise использует форму оптимистической стратегии для уменьшения вероятности коллизий и поддержки целостности данных. Контейнер проверяет значение записи базы данных, которая должна быть изменена в процессе выполнения транзакции. Если значение записи на момент выполнения операции записи совпадает со значением в начале транзакции, то Контейнер вносит необходимые изменения. Если нет - т.е. текущее состояние записи отличается от состояния на момент начала транзакции - Контейнер запрещает выполнения второй транзакции и откатывает ее.

Контейнер делает копию состояния Компонента перед выполнением первой операции чтения. Затем, в процессе выполнения нашей транзакции, другие транзакции изменяют его состояние. При завершением нашей транзакции Контейнер определяет, значение каких полей было изменено, и выполняет запись состояния только этих полей. Контейнер позволяет транзакциям успешно завершаться, если они изменяют значения различных полей одной и той же записи. Если же транзакции пытаются одновременно изменить значение одного и того же поля - т.е. транзакции считали одно и то же значение, и одна из них успела подтвердить изменения - Контейнер откатывает вторую транзакцию при попытке ее подтверждения, поскольку данные были изменены и эта вторая транзакция (или "last to commit"-транзакция) не имеет информации о ранее внесенных изменениях.

Пример использования Entity-Компонента Bank

Пример bank иллюстрирует использование Entity-Компонента. В нем предусмотрены две реализации одного и того же remote-интерфейса Account: одна реализация использует сохранение, управляемое Контейнером (CMP), другая - сохранение, управляемое Компонентом (BMP).

Счета клиентов (saving accounts) моделируются с помощью Компонента, который называется SavingAccount. Для него используется BMP. Когда мы будем подробнее знакомиться с кодом Компонента, вы увидите, что он содержит JDBC-вызовы к базе данных.

Контрольные счета (checking accounts) моделируются с помощью Компонента, который называется CheckingAccount. Сохранение его состояния возложено на Контейнер, т.е. он иллюстрирует использование CMP.

Третий вид Компонента - Оператор (Teller), который занимается переводом средств с одного счета на другой. Он представляет из себя stateless Session-Компонент, и его назначение - показать, как группу обращений к нескольким Entity-Компонентам можно заключить в управляемую Контейнером транзакцию. Так, если операция записи на счет выполняется раньше операции списания со счета и произошел сбой во время операции списания, транзакция будет откачена, и ни одно из операций не будет выполнена.

Home-интерфейс Entity-Компонента

Несколько entity-Компонентов могут реализовывать один и тот же интерфейс, даже если они используют различный вид управления состоянием (CMP или BMP). И SavingsAccount, и CheckingAccount используют один и тот же home-интерфейс (AccountHome) и remote-интерфейс (Account).

Home-интерфейсы для entity- и session-Компонентов очень похожи. Они наследуют один и тот же интерфейс `javax.ejb.EJBHome`. Home-интерфейс для entity-Компонента обязан объявить хотя бы один метод поиска; кроме того, наличие create-методов не является обязательным.

Пример Кода 7.4 содержит определение интерфейса AccountHome.

Пример Кода 7.4 Интерфейс AccountHome

```
public interface AccountHome extends javax.ejb.EJBHome {
   Account create(String name, float balance)
      throws java.rmi.RemoteException, javax.ejb.CreateException;
   Account findByPrimaryKey(AccountPK primaryKey)
      throws java.rmi.RemoteException, javax.ejb.FinderException;
   java.util.Enumeration findAccountsLargerThan(float balance)
      throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

Интерфейс AccountHome объявляет три метода. Хотя наличие метода `create()` не является обязательным, такой метод объявлен в нашем примере. Этот метод позволяет добавить новую запись в сопоставленную с Компонентом таблицу БД. При добавлении новых данных Entity-Компонент может полагаться на систему управления этой БД (DBMS) или на другое приложение - в этом случае метод `create()` не используется.

В нашем примере метод `create()` имеет два аргумента - номер счета (string) и его начальное состояние (float). Реализация метода использует значения этих аргументов для инициализации объекта, т.е. присвоения начальных значений номера счета и его состояния для вновь созданного объекта. Список возможных исключений должен включать в себя `java.rmi.RemoteException` и `java.ejb.CreateException`; если это необходимо, в throw-список метода create() можно добавить и другие исключения.

Entity-Компонент обязан объявить метод поиска `findByPrimarykey()`; он находится в интерфейсе AccountHome. Этот метод имеет единственный аргумент - значение главного ключа типа AccountPK, и возвращает ссылку на remote-интерфейс Account. Разумеется, метод используются для поиска только одного Компонента.

В home-интерфейсе AccountHome объявлен еще один метод поиска, `findAccountLargerThan()`, хотя это и не является обязательным. Этот метод возвращает набор Компонентов (Java Enumeration), которые

удовлетворяют условию поиска (на счету средств больше, чем указанная величина).

Все методы поиска обязаны содержать java.rmi.RemoteException и `java.ejb.FinderException` в их throws-списке исключений.

Remote-интерфейс Entity-Компонента

Несколько Entity-Компонентов могут реализовывать один и тот же remote-интерфейс, даже если они используют различные способы сохранения своего состояния (BMP или CMP). Оба Entity-Компонента примера bank реализуют один и тот же интерфейс Account.

Remote-интерфейсы для Session- и Entity-Компонентов практически идентичны друг другу - оба они наследуют интерфейс `javax.ejb.EJBObject` и объявляют те бизнес-методы, который должны быть доступны для клиента.

Пример Кода 7.5 содержит реализацию remote-интерфейса:

Пример Кода 7.5 Remote-интерфейс Account

```
public interface Account extends javax.ejb.EJBObject {
   public float getBalance() throws java.rmi.RemoteException;
      public void credit(float amount) throws
                        java.rmi.RemoteException;
      public void debit(float amount) throws java.rmi.RemoteException;
}
```

Интерфейс Account объявляет три бизнес-метода: `getBalance()`, `credit(0` и `debit()`.

Entity-Компонент с CMP

Для иллюстрации основ использования Entity-Компонентов с CMP в примере bank служит Компонент CheckingAccount. Во многом его реализация очень похожа на реализацию session-Компонента. Тем не менее, для реализации Entity-Компонента с CMP характерно наличие нескольких важных особенностей:

- Entity-Компонент не содержит реализаций методов поиска. Для таких Компонентов (Компонентов с CMP) эти реализации обеспечивает Контейнер EJB. Вместо реализации этих методов в классе Компонента его разработчик помещает всю необходимую для их реализации информацию в Дескриптор Поставки.
- Поля, сохранение состояния которых берет на себя Контейнер, должны быть объявлены как `public`. В нашем примере public-полями являются поля `name` и `balance` Компонента CheckingAccount.
- Класс такого Компонента содержит реализации семи методов, объявленных в интерфейсе `EntityBean - ejbActivate()`,

ebjPassivate(), ejbLoad(), ejbStore(), ejbRemove(), setEntityContext() и unsetEntityContext(). Впрочем, обязательной является их тривиальная реализация, хотя, разумеется, разработчик может добавить в них любой код. В нашем примере для Компонента CheckingAccount выполняется сохранение контекста, возвращаемого методом setEntityContext(), и его сброс в методе unsetEntityContext(). Остальные методы не содержат никакого дополнительного кода.

- Для нашего Компонента предусмотрена реализация метода ejbCreate() (поскольку мы позволяем клиенту создавать новые счета), и для инициализации полей нового объекта используются значения аргументов этого метода. В качестве результата метод ejbCreate() возвращает значение null, так как создание нужной ссылки для возврата ее клиенту при использовании CMP выполняется Контейнером.
- Для нашего Компонента в методе ejbPostCreate() выполняется минимум операций, хотя, в принципе, этот метод мог бы выполнять значительную часть действий по инициализации Компонента. Для Компонента с CMP важно выполнить только самые необходимые действия, так как вызов метода ejbPostCreate() выполняется как уведомление (notification callback) о выполняемых действиях. Обратите внимание, что это же справедливо и для других методов, унаследованных от интерфейса EntityBean.

Пример Кода 7.6 Реализация Entity-Компонента с CMP.

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

public class CheckingAccount implements EntityBean {
   private javax.ejb.EntityContext _context;
   public String name;
   public float balance;
   public float getBalance() {
      return balance;
   }
   public void debit(float amount) {
      if(amount > balance) {
         // Пометить текущую транзакцию для последующего отката
         _context.setRollbackOnly();
      }
      else {
         balance = balance - amount;
      }
   }
   public void credit(float amount) {
      balance = balance + amount;
   }
```

```
    public AccountPK ejbCreate(String name, float balance) {
       this.name = name;
       this.balance = balance;
       return null;
    }

    public void ejbPostCreate(String name, float balance) {}
    public void ejbRemove() {}
    public void setEntityContext(EntityContext context) {
       _context = context;
    }
    public void unsetEntityContext() {
       context = null;
    }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public String toString() {
       return "CheckingAccount[name=" + name + ",balance=" + balance
                          +"]";
    }
}
```

Entity-Компонент с BMP

В примере bank присутствует также Entity-Компонент SavingsAccount. Он является хорошей иллюстрацией использования BMP.

Компонент SavingsAccount используется для работы с другими таблицами, а не с теми же, с которыми взаимодействует CheckingAccount. Несмотря на то, что эти Компоненты используют различный способ сохранения своего состояния, они оба реализуют одни и те же home- и remote-интерфейсы. Несмотря на это, их реализации во многом отличаются друг от друга, и эти отличия носят принципиальный характер.

Для реализации Entity-Компонента с CMP характерно следующее:

- Обычно поля Компонента объявляются как `private`, а не `public`. Взаимодействием с базой данных - чтением данных из БД в поля и сохранение результатов в БД - управляет сам Компонент, поэтому имеет смысл ограничить права доступа к этим полям. Это в корне отличается от использования CMP, когда вы обязаны сделать поля Компонента доступными для Контейнера, т.е. объявить их как public.
- Метод `ejbCreate()` возвращает значение типа класса главного ключа - AccountPK в нашем примере. Контейнер использует это значение для создания удаленной ссылки на экземпляр Entity-Компонента.

- Разработчик для Компонентов как с BMP, так и с CMP может поместить дополнительный код в метод `ejbPostCreate()`. В нашем примере такой код отсутствует.
- Необходимо написать реализации методов `ejbLosd()` и `ejbStore()`. Для Компонента с CMP вполне подходят тривиальные (пустые) реализации, так как взаимодействие с БД берет на себя Контейнер. Entity-Компонент с BMP должен сам выполнять чтение информации из БД в поля Компонента (метод `ejbLosd()`) и запись в нее внесенных изменений (метод `ejbStore()`).
- Должна присутствовать реализация всех методов поиска. Компонент SavingsAccount содержит два таких метода: обязательный - `ejbFindByPrimaryKey()` и произвольный - `ejbFindAccountsLargerThan()`.
- Entity-Компонент с BMP обязан реализовать метод `ejbRemove()`, объявленный в интерфейсе `EntityBean`. Поскольку Компонент является просто объектной моделью информации в БД, он обязан реализовать этот метод, чтобы позволить выполнить удаление данных (т.е. Entity-Компонента) из БД. Компоненту с CMP достаточно иметь пустую реализацию этого метода, так как все необходимые действия выполняются Контейнером.
- Каждый метод, имеющий отношение к взаимодействию с базой данных - `ejbCreate()`, `ejbRemove()`, `ejbLoad()`, `ejbStore()`, `ejbFindByPrimaryKey()` и другие методы поиска, а также бизнес-методы Компонента - должны содержать соответствующий код. В общем случае все эти методы содержат команды соединения с базой данных и команды создания и выполнения операторов SQL, которые и реализуют функциональность Компонента. После выполнения всех SQL-операторов, код метода должен "закрыть" эти операторы и выполнить отсоединение от базы данных.

Пример Кода 7.7 содержит наиболее интересные фрагменты реализации класса SavingsAccount. В примере не показаны методы с тривиальной реализацией - `ejbActivate()`, `ejbPassivate()` и др.

Давайте подробнее ознакомимся с методом `ejbLoad()` для изучения того, каким образом Компонент с BMP взаимодействует с базой данных. Обращаем ваше внимание, что все другие методы, реализованные в классе SavingsAcoounts, выполняют примерно те же действия. Сначала устанавливается соединение с базой данных, для чего используется вызов метода `getConnection`, который использует DataSource для получения JDBC-соединения с базой данных из пула JDBC-соединений. После того, как соединение установлено, метод `ejbLoad()` создает объект типа PreparedStatement и сопоставленный с ним SQL-оператор. Поскольку метод `ejbLoad()` отвечает за чтение данных из БД в поля Компонента, он формирует, а затем выполняет оператор SELECT, который читает состояние для указанного счета (по его названию). Если запрос возвращает результат, то из него извлекается количество средств на счету. Затем метод `ejbLoad()` закрывает объект `PreparedStatement` и отсоединятся от базы данных. Обратите внимание, что не происходит настоящего закрытия соединения с БД - вместо этого соединение возвращается в пул соединений.

Пример Кода 7.7 Реализация Entity-Компонента с BMP

```
import java.sql.*;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;

public class SavingsAccount implements EntityBean {
   private EntityContext _context;
   private String _name;
   private float _balance;
   public float getBalance() {
      return _balance;
   }
   public void debit(float amount) {
      if(amount > _balance) {
         // Пометить транзакцию для последующего отката
         _context setRollbackOnly();
      } else {
         _balance = _balance - amount;
      }
   }
   public void credit(float amount) {
      _balance = _balance + amount;
   }
   // Пустая реализация методов setEntityContext(),
                  unsetEntityContext(),
   // ejbActivate(), ejbPassivate(), ejbPostCreate() не показана
...
   public AccountPK ejbCreate(String name, float balance)
        throws RemoteException, CreateException {
      _name = name;
      _balance = balance;
      try {
         Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement
         ("INSERT INTO Savings_Accounts (name, balance) VALUES (?,
                  ?)");
         statement.setString(1, _name);
         statement.setFloat(2, _balance);
         if(statement.executeUpdate() != 1) {
            throw new CreateException("Could not create: " + name);
         }
         statement.close();
         connection.close();
         return new AccountPK(name);
```

```
      } catch(SQLException e) {
         throw new RemoteException("Could not create: " + name, e);
      }
   }
...
   public void ejbRemove() throws RemoteException, RemoveException {
      try {
         Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement
            ("DELETE FROM Savings_Accounts WHERE name = ?");
         statement.setString(1, _name);
         if(statement.executeUpdate() != 1) {
            throw new RemoveException("Could not remove: " + _name);
         }
         statement.close();
         connection.close();
      } catch(SQLException e) {
         throw new RemoteException("Could not remove: " + _name, e);
      }
   }
   public AccountPK ejbFindByPrimaryKey(AccountPK key) throws
                  RemoteException,
         FinderException {
      try {
         Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement
            ("SELECT name FROM Savings_Accounts WHERE name = ?");
         statement.setString(1, key.name);
         ResultSet resultSet = statement.executeQuery();
         if(!resultSet.next()) {
            throw new FinderException("Could not find: " + key);
         }
         statement.close();
         connection.close();
         return key;
      } catch(SQLException e) {
         throw new RemoteException("Could not find: " + key, e);
      }
   }

   public.java.util.Enumeration ejbFindAccountsLargerThan(float
                  balance)
         throws RemoteException, FinderException {
      try {
         Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement
```

```
            ("SELECT name FROM Savings_Accounts WHERE balance > ?");
         statement.setFloat(1, balance);
         ResultSet resultSet = statement.executeQuery();
         Vector keys = new Vector();
         while(resultSet.next()) {
            String name = resultSet.getString(1);
            keys.addElement(new AccountPK(name));
         }
         statement.close();
         connection.close();
         return keys.elements();
      } catch(SQLException e) {
         throw new RemoteException("Could not findAccountsLargerThan: "
                        + balance, e);
      }
   }

   public void ejbLoad() throws RemoteException {
      // Получение имени из primary key
      _name = ((AccountPK) _context.getPrimaryKey()).name;
      try {
         Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement
            ("SELECT balance FROM Savings_Accounts WHERE name = ?");
         statement.setString(1, _name);
         ResultSet resultSet = statement.executeQuery();
         if(!resultSet.next()) {
            throw new RemoteException("Account not found: " + _name);
         }
         _balance = resultSet.getFloat(1);
         statement.close();
         connection.close();
      } catch(SQLException e) {
         throw new RemoteException("Could not load: " + _name, e);
      }
   }

   public void ejbStore() throws RemoteException {
      try {
         Connection connection = getConnection();
         PreparedStatement statement = connection.prepareStatement
            ("UPDATE Savings_Accounts SET balance = ? WHERE name = ?");
         statement.setFloat(1, _balance);
         statement.setString(2, _name);
         statement.executeUpdate();
         statement.close();
```

```
            connection.close();
         } catch(SQLException e) {
            throw new RemoteException("Could not store: " + _name, e);
         }
      }

      private Connection getConnection() throws SQLException {
         Properties properties = _context.getEnvironment();
         String url = properties.getProperty("db.url");
         String username = properties.getProperty("db.username");
         String password = properties.getProperty("db.password");
         if(username != null) {
            return DriverManager.getConnection(url, username, password);
         } else {
            return DriverManager.getConnection(url);
         }
      }
      public String toString() {
         return "SavingsAccount[name=" + _name + ",balance=" + _balance
                     +"]";
      }
   }
```

Дескриптор Поставки Entity-Компонента

Дескриптор Поставки из примера bank содержит информацию о трех типах Компонентов - Session-Компоненте Teller, Entity-Компоненте с CMP CheckingAccount и Entity-Компоненте с BMP SavingsAccount. Подробная информация о поставке Компонентов приведена в главе 9 "Поставка Компонентов EJB". В этом разделе рассматривается то, что относится к Entity-Компонентам.

Информация в Дескрипторе Поставки (обычно используется термин свойства, properties) используется для описания интерфейсов Компонента, атрибутов транзакций и пр. точно так же, как и для Session-Компонентов, но для Entity-Компонентов появляются дополнительные свойства. Пример Кода 7.8 на стр. 7-25 демонстрирует использование типичных тегов, применяемых для описания Entity-Компонентов с CMP. Если вы сравните два Дескриптора для Entity-Компонентов с разными системами управления состоянием, то увидите, что при использовании CMP Дескриптор получается более сложным.

Дескриптор Поставки для Entity-Компонента содержит следующую информацию:

- Тип Компонента, который определяется как `<entity>`. Обратите внимание, что первый тег в секции `<enterprise-bean>` в Примерах Кода 7.8 и 7.9 говорит о том, что используется Entity-Компонент.

- Имена сопоставленных с Компонентом интерфейсов (home и remote) и имя класса реализации. Для их задания используются теги <home>, <remove> и <ejb-class>, соответственно.
- JNDI-имена, под которыми Компонент зарегистрирован в службе имен и которые использует клиент при установке связи с Компонентом.
- Атрибуты и уровни изоляции транзакций Компонента. Обычно эта информация содержится в разделе <assembly-descriptor> Дескриптора Поставки.
- Имя primary key-класса Компонента. В нашем примере, это класс AccountPK, и он описан в теге <prim-key-class>.
- Тип сохранения состояния Компонента. Entity-Компонент может использовать как BMP, так и CMP. Компонент CheckingAccount использует CMP, поэтому в теге <presistence-type> Дескриптора Поставки содержится значение Container.
- Признак, является ли Компонент реентерабельным или нет. Ни CheckingAccount, ни SavingsAccount не являются реентерабельными, поэтому для обоих в теге <reentrant> задано значение False.
- Поля, сохранением состояния которых управляет Контейнер, если используется CMP. Компонент с BMP (SavingsAccount в нашем примере) не должен указывать эту информацию. Entity-Компонент с CMP обязан указать имена полей, сохранением состояния которых должен управлять Контейнер. Для этого используется комбинация тегов <cmp-field> и <field-name>. Первый их этих тегов, <cmp-field>, говорит о том, что Контейнер управляет состоянием данного поля. Внутри этого тега, тег <field-name> определяет имя поля. Например, Дескриптор Поставки для Компонента CheckingAccount определяет, что Контейнер управляет состоянием поля balance, с помощью следущей строки:

  ```
  <cmp-field><field-name>balance</field-name></cmp-field>
  ```

- Информация, относящаяся к сопоставленной с Компонентом базой данных. В частности, эта информация идентифицирует базу данных (включая информацию о пользователе) и определяет таблицу, в которой хранятся Entity-объекты. Для этого используется специфическая для реализации фирмы Inprise секция <datasource>. Более подробно это обсуждается в главе 9 "Поставка Компонентов EJB" в разделе "Источники данных" на стр. 9-9
- Дополнительная информация оь управлемых Контейнером полях, которая необходима для генерации Контейнером методов поиска (только в случае использования CMP). Для этого также используется специфическая для Inprise-реализации секция <datasource>. См. раздел "Источники данных" на стр. 9-9 в главе 9 "Поставка Компонентов EJB".

Приведенный ниже Пример Кода 7.8 показывает основные части Дескриптора Поставки для Entity-Компонента с BMP. Поскольку

Компонент сам осуществляет обмен данных между БД и своими полями (как при чтении, так и при записи), а не полагается при этом на Контейнер, то Дескриптор Поставки не содержит перечисления таких полей. Кроме того, в Дескрипторе не содержится никакой информации для генерации методов поиска, так как в классе Компонента содержатся их готовые реализации.

Пример Кода 7.8 Дескриптор Поставки для Entity-Компонента с BMP

```
...
<enterprise-beans>
<entity>
   <description>This Entity bean is an example of Bean Managed
               Persistence</description>
   <ejb-name>savings</ejb-name>
   <home>AccountHome</home>
   <remote>Account</remote>
   <ejb-class>SavingsAccount</ejb-class>
   <persistence-type>Bean</persistence-type>
   <prim-key-class>AccountPK</prim-key-class>
   <reentrant>False</reentrant>
</entity>
   ...
</enterprise beans>
<assembly-descriptor>
   <container-transaction>
      <method>
         <ejb-name>savings</ejb-name>
         <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
   </container-transaction>
</assembly-descriptor>
```

Пример Кода 7.9 показывает основные части Дескриптора Поставки для Entity-Компонента с CMP. Поскольку Компонент полагается на Контейнер при чтении данных из БД и сохранении внесенных изменений, в Дескрипторе Поставки указаны соответствующие имена полей.

Пример Кода 7.9 Дескриптор Поставки для Entity-Компонента с CMP

```
<enterprise-beans>
<entity>
   <description>
      This Entity bean is an example of Container Managed Persistence
               </description>
   <ejb-name>checking</ejb-name>
   <home>AccountHome</home>
```

```
      <remote>Account</remote>
      <ejb-class>CheckingAccount</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>AccountPK</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field>
         <field-name>name</field-name>
      </cmp-field>
      <cmp-field>
         <field-name>balance</field-name>
      </cmp-field>
   </entity>
   </enterprise beans>
   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>checking</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
   </assembly-descriptor>
```

Использование секции datasource

Entity-Компонент с CMP для задания собственной конфигурации использует сецифическую для Inprise-реализации Дескриптора Поставки секцию datasource. Эта секция нужна для организации связи с базой данных и говорит программе, как создавать экземпляры источников данных.

Вы можете указать всю необходимую информацию обазе данных - URL, имя пользователя, пароль и т.д. - с помощью секции datasource и элемента <datasource>.

Подробное описание этой секции Дескриптора Поставки приведено в разделе "Источники данных" на стр. 9-9 в главе 9 "Поставка Компонентов EJB".

Поставка Компонентов для примера bank

Поскольку в примере для хранения состояния объектов используется внешняя база данных, то поставка Компонентов для него требует выполнения некоторых дополнительных действий. Подробное описание процесса создания таблиц и внесения необходимых изменений в Дескриптор Поставки находится в разделе "Пример использования Oracle" на стр. 7-27.

Вот шаги, которые необходимо сделать в процессе поставки:

1 Определение базы данных, с которой будет происходить взаимодействие с помощью JDBC.

   1 Создание нового пользователя и предоставление ему прав для создания новых таблиц.

   2 Создание в БД двух новых таблиц со следующей структурой полей: строковое поле переменной длины (не менее 5 символов) и поле типа float с точностью не менее 3 знака после запятой. Одна из них должна называться Savings_Accounts, вторая - Checking_Accounts.

   3 Внесение изменений в Дескриптор Поставки, хранящийся в файле EJB-JAR.XML в подкаталоге meta-inf в каталоге, где находится пример bank. Необходимо указать URL JDBC-драйвера и базы данных, а также идентификатор и пароль пользователя.

   4 Добавление к переменной среды classpath пути к JDBC-драйверу.

   5 Запуск Контейнера EJB. Для этого можно использовать следующую командную строку:

```
prompt% vbj com.inprise.ejb.Container test beans.jar -jts -jns
```

   6 Запуск приложения-клиента с помощью команды

```
prompt% vbj BankClient
```

Вы должны увидеть следующие результаты:

```
Peter's balance: 200
Paul's  balance: 100
Taking from Peter and giving to Paul
Peter's balance: 200
Paul's  balance: 100
```

Использование режима отладки

Если у вас при запуске возникли проблемы, используйте стандартный отладочный режим (EJBDebug). Для запуска сервера с включенным режимом отладки, введите следующую команду:

```
prompt% vbj -DEJBDebug com.inprise.ejb.Container test beans.jar -jts - jns
```

Пример использования Oracle

В нашем примере поставка Компонента осуществляется для работы с Oracle 7.3.3, запущенном на компьютере "gemini", номер порта 1525, алиас "GEORA733", пользователь "scott" с паролем "tiger".

Создание таблиц:

```
prompt% sqlplus scott/tiger@GEORA73
create table Savings_Accounts  (name varchar(10), balance float(10));
create table Checking_Accounts (name varchar(10), balance float(10));
```

```
exit;
prompt%
```

Все необходимые команды должны быть заданы между системными символами prompt. В нашем случае выполняется создание двух таблиц, каждая из которых содержит два поля - строковое длиной до 10 символов и поле типа float.

Применительно к этому примеру, для обоих Компонентов - и SavingsAccount, и CheckingAccount - вы могли бы использовать секцию <datasource> следующего вида:

```
<deployment-descriptor>
<datasource>
      <res-ref-name>jdbc/SavingsDataSource</res-ref-name>
      <url>jdbc:oracle:thin:@gemini:1525:GEORA733</url>
      <username>scott</username>
      <password>tiger</password>
      <driver-class-name>oracle.jdbc.driver.OracleDriver</driver-
            class-name>
   </datasource>
</deployment-descriptor>
```

Возможно, вам потребуется изменить параметры в секции datasource, например, имя хоста, номер порта или JDBC URL.

Более сложные вопросы использования CMP

В предыдущих разделах были рассмотрены основы использования CMP. Было показано, как нужно работать с полями, управление которыми берет на себя Контейнер, а именно: такие поля нужно объявить как public-поля в классе реализации Компонента, а затем перечислить их в Дескрипторе Поставки. Хотя методы поиска объявлены в home-интерфейсе, их реализация отсутствует в классе Компонента. Описание логики работы этих методов также содержится в Дескрипторе Поставки. Эта информация используется Контейнером EJB для определения полей, сохранение состояния которых он берет на себя, а также для реализации методов поиска.

В этом разделе затрагиваются более сложные вопросы, связанные с использованием CMP, особенно для Inprise-реализации Контейнера EJB. В частности, подробно рассматривается объектная (на уровне Компонентов EJB) модель взаимодействия таблиц реляционных баз данных. Обсуждаются также различные стратегии построения такой модели, которые вы можете задавать при создании Дескриптора Поставки.

Объектная модель взаимодействия реляционных таблиц

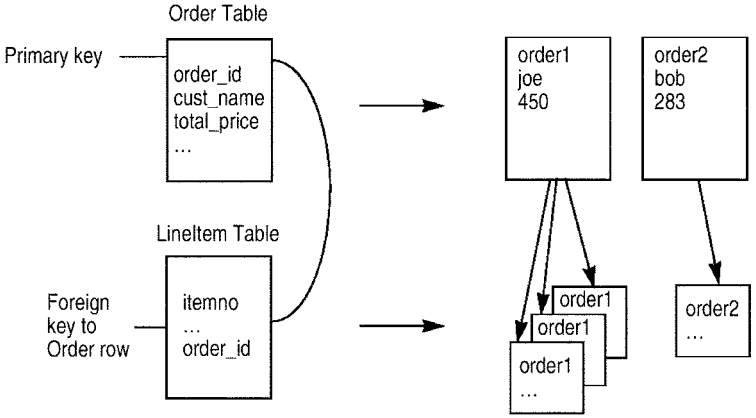
Вы уже видели, как выполняется сопоставление одного entity-Компонента с одной базой данных; другими словами, как один экземпляр entity-Компонента представляет одну запись таблицы и каждое поле представляет столбец таблицы. Экземпляр entity-Компонента идентифицируется с помощью главного ключа, который обычно соответствует главному ключу записи в таблице.

Во многих случаях таблицы базы данных связаны с другими таблицами. Как правило, таблицы связаны друг с другом отношениями один-к-одному или один-ко-многим (и, соответственно, многие-к-одному). Связь между таблицами поддерживается с помощью удаленного ключа (foreign key).

Давайте посмотрим, как это могло бы выглядеть в реляционной базе данных, которая используется для обслуживания заказов покупателей. Покупатель размещает заказ на некоторые виды товаров, и в одном заказе может содержаться несколько записей для одного и того же товара. Соответственно, база данных может содержать таблицу Order, в которой хранится информация о заказах. С каждой записью этой таблицы сопоставлен ключ, значением которого является значение колонки order_id, которое уникальным образом идентифицирует каждый заказ. Каждая запись также содержит такую информацию, как дата заказа, имя покупателя, общая стоимость и т.п. Каждая запись другой таблицы, которая называется LineItem, содержит отдельную запись на каждый заказанный товар. Конкретный заказ может иметь несколько записей о товаре. Таблица LineItem содержит удаленный ключ, который соответствует полю order_id в таблице Order, который позволяет определить, какому заказу принадлежит данный товар.

На рисунке 7.2 показаны два заказа, хранящиеся в таблице Order; значения их ключей равны `order1` и `order2`. Столбец удаленного ключа LineItem показывает, что заказ1 состоит из трех позиций, а заказ2 - из одной. Каждой записи в таблице LineItem сопоставлена одна, и только одна, запись в таблице Order.

Рис. 7.2 Отношения между таблицами и записями



Хотя entity-Компоненты могут использовать BMP для представления этих отношений между Компонентами, для этого также возможно использовать CMP. Моделирование таких отношений с использованием Entity-Компонентов с CMP - задача, намного более простая.

Реализация отношения один-к-одному

Entity-Компоненты с CMP реализуют соотношение один-к-одному (одна запись из LineItem к одной записи в Order) естественным образом. Чтобы посмотреть, как экземпляр LineItem-Компонента может найти сопоставленный с ним экземпляр Order-Компонента, давайте ознакомимся со структурой Компонентов.

Класс OrderBean вместе с классом Order (remote-интерфейс) и OrderHome (home-интерфейс) определяют представление таблицы Order. Таким же образом класс LineItemBean является моделью таблицы LineItem, и для него определены remote-интерфейс LineItem и home-интерфейс LineItemHome.

Компонент OrderBean содержит public-поля, которые соответствуют колонкам таблицы Order. Эти поля имеют имена `order_id`, `customer`, `price` и т.д. (Обратите внимание, что entity-Компонент может информировать Контейнер о том, что нужно использовать различные имена для полей и для столбцов базы данных - в Дескрипторе Поставки предусмотрены свойства, которые обеспечивают соответствия между SQL-именами столбцов и Java-именами полей). Поле `order_id` является главным ключом таблицы OrderBean. Код Компонента OrderBean содержит следующие объявления полей (см. пример кода 7.10):

Пример Кода 7.10 Объявления полей Компонента OrderBean

```
public class OrderBean implements EntityBean{
   public String order_id;
   public String customer;
```

```
public float price;
...
```

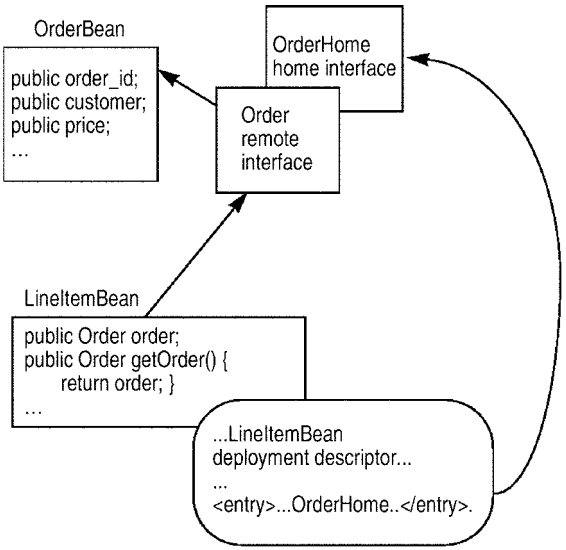
Компонент LineItemBean также содержит public-поля, которые соответствуют колонкам таблицы LineItem. Хотя поле `order_id` является удаленным ключом, оно не реализовано с помощью primary-key-класса, зато оно реализовано с помощью типа remote-интерфейса Order. Другими словами, оно представляет собой объектную ссылку на объект типа `Order`. Для Inprise-реализации EJB, любое поле entity-Компонента с CMP может использоваться как колонка удаленного ключа в таблице базы данных. Контейнер EJB трактует такие поля как ссылку на `EJBObject`. Компонент LineItemBean также объявляет метод `getOrder()`, который возвращает ассоциированный с ним order-объект. Все это показано в примере кода 7.11.

Пример Кода 7.11 Объявление полей Компонента с CMP

```
public class LineItemBean implements EntityBean {
   public itemno;
   public Order order;
   ...
   public Order getOrder() {
      return order;
   }
   ...
```

Дескриптор Поставки Компонента LineItemBean содержит свойство, которое хранит значение JNDI-имени для интерфейса OrderHome. Контейнер использует это имя для поиска home-интерфейса, как это показано на рисунке 7.3.

Рис. 7.3 Реализация отношения один-к-одному с помощью entity-Компонента с CMP.



Одним из возможных путей работы является следующий: Контейнер получает значение главного ключа для экземпляра Компонента LineItem, а затем вызывает метод `findByPrimaryKey()` home-интерфейса OrderHome для получения ссылки на экземпляр OrderBean.

Inprise-Контейнер EJB использует значение удаленного ключа - объектную ссылку на remote-интерфейс Order - в классе Компонента LineItemBean для непосредственного получения ссылки на нужный экземпляр OrderBean. Используя ссылку на OrderHome в Дескрипторе Поставки и ссылку как удаленный ключ, Inprise-Контейнер EJB позволяет обойтись без вызова метода `findByPrimaryKey()` интерфейса OrderHome. Такой способ позволяет существенно повысить производительность приложения, так как обычно этот метод обращается к базе данных.

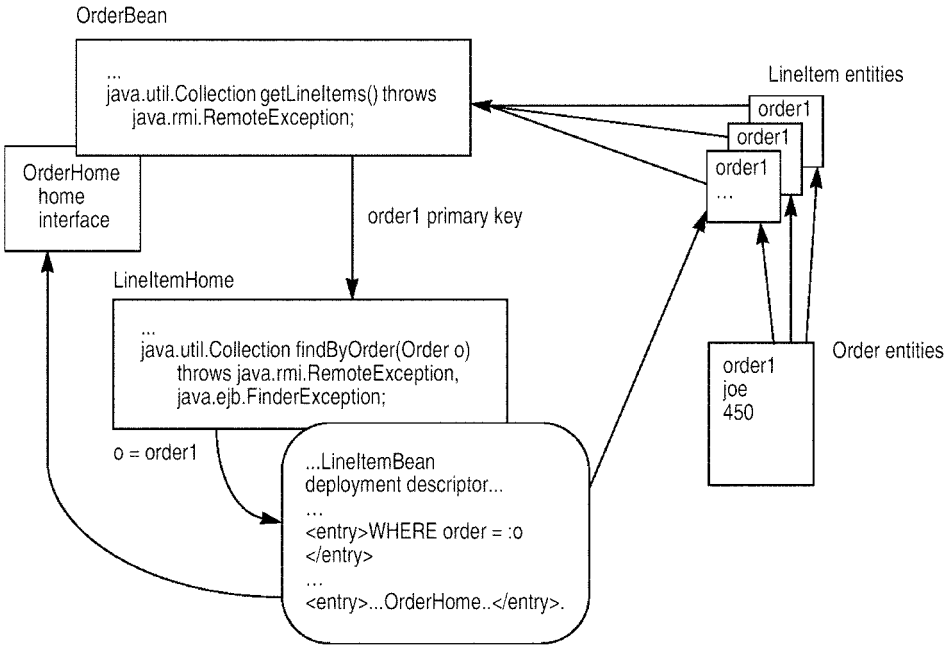
Контейнер создает объектную ссылку на remote-интерфейс Order, т.е. на конкретный экземпляр OrderBean, непосредственно. Тем не менее, Контейнер не считывает состояние экземпляра OrderBean, пока этот объект не будет использован. Другими словами, когда Контейнер считывает состояние экземпляра LineItemBean, он не считывает состояние OrderBean. Иногда это называют "lasy reference", и это позволяет существенно увеличить производительность системы.

Реализация отношения один-ко-многим

Inprise-Контейнер EJB также способен моделировать отношения один-ко-многим или (что то же самое) многие-к-одному при использовании Компонентов с CMP. Используя рассмотренные в предыдущем примере реализации Order и LineItem, Контейнер легко может найти все данные, сопоставленные с конкретным заказом.

Для выполнения этого Контейнер ищет соответствия между ссылками на `EJB-object`, которые моделируют значения удаленного ключа для entity-Компонента на стороне "многие", с полем главного ключа Компонента на стороне "один". В рассмотренном выше примере, Компонент Order содержит главный ключ, который идентифицирует каждый конкретный заказ, и он имеет значения, такие как `order1` и `order2`. Каждый экземпляр Компонента LineItemBean - т.е. каждая позиция, относящаяся к заказу - содержит удаленный ключ, чье значение идентифицирует заказ, с которым сопоставлена данная запись. Иллюстрация процесса приведена на рисунке 7.4, за которым следуют пояснения.

Рис. 7.4 Реализация отношения один-ко-многим при использовании CMP



Чтобы все это работало, реализация Компонента на стороне "один" - т.е. OrderBean - должна содержать метод, который возвращает все относящиеся к нему "многие" экземпляры - т.е. Компоненты типа LineItemBean. Этот метод, который мы назвали `getLineItems()`, базируется на использовании методов поиска home-интерфейса LineItemHome, который ищет все entity-объекты, значения удаленных ключей которых соответствуют значению главного ключа Компонента OrderBean. Поскольку мы работаем с CMP, метод поиска в home-интерфейсе представляет собой просто декларацию метода. Его логика описана в Дескрипторе Поставки. Свойство Дескриптора Поставки информирует Контейнер EJB о том, как нужно сконструировать предложение `WHERE` SQL-запроса для поиска тех объектов LineItemBean, значения удаленных ключей которых равны значению главного ключа Компонента OrderBean.

В нашем примере класс OrderBean содержит метод, который возвращает `java.util.Enumeration` или `java.util.Collection` - т.е. набор - всех нужных экземпляров LineItemBean. (Используйте интерфейс Collection при работе с Java 2. Используйте интерфейс `Enumeration`, если вы работаете с более ранними версиями JDK. Вы можете продолжать использовать интерфейс `Enumeration` даже при работе с Java 2; интерфейс `Collection` недоступен в более ранних версиях JDK.) Например, вы могли бы использовать следующее объявление метода:

```
java.util.Enumaration getLineItems() throws java.rmi.RemoteException;
```

или:

```
java.util.Collection getLineItems() throws java.rmi.RemoteException;
```

Этот метод использует контекст Компонента, который становится известен при вызове метода `setEntityContext()`, для выполнения поиска home-интерфейса LineItemHome. Пример кода 7.12 показывает наиболее важные фрагменты кода этого метода.

Пример Кода 7.12 Метод getLineItems() Компонента OrderBean

```
public class OrderBean implements EntityBean {
   private LineItemHome _lineItemHome;
   ...
   public void setEntityContext(EntityContext context) throws
               java.rmi.RemoteException {
      _context = context;
      ...
   }
   ...
   public java.util.Collection getLineItems() throws
               java.rmi.RemoteException {
      try {
         Order self = (Order) _context.getEJBObject();
         return _lineItemHome.findByOrder(self);
      catch (javax.ejb.FinderException e) {...}
   }
...
```

Класс OrderBean содержит `private`-поле `_lineItemHome` для хранения ссылки на интерфейс LineItemHome. Метод `setEntityContext()` сохраняет контекст Компонента в поле `_context`. В методе `getLineItems()` происходит обращение к методу `_context.getEJBObject()` для получения ссылки на его remote-интерфейс Order. После получения этой ссылки можно обратиться к методу `findByOrder()`, объявленному в интерфейсе LineItemHome, и передать ему в качестве аргумента ссылку на этот remote-интерфейс.

Для выполнения всех необходимых действий объявление метода `findByOrder()` могло бы выглядеть так:

```
java.util.Collection findByOrder (Order order) throws
   java.rmi.RemoteException, java.ejb.FinderException;
```

Контейнер EJB генерирует реализацию метода `findByOrder()` в соответствии с информацией в Дескрипторе Поставки. Свойства Дескриптора Поставки содержат часть SQL-оператора для выборки всех Компонентов LineItem, значение поля `order_id` которых равно значению параметра `o`, ключевому значению для OrderBean-Компонента. Этот оператор мог бы выглядеть так:

```
WHERE LineItem = :o
```

В Дескрипторе Поставки также содержится свойство, которое хранит значение JNDI-имени для интерфейса OrderHome, чтобы Контейнер EJB мог найти этот интерфейс.

Объектное представление реляционных соотношений с точки зрения клиента

Код клиентского приложения может непосредственно использовать реализацию только что рассмотренной объектной модели.

Приложение OrderClient показывает, как могут выглядеть соотношения "один-к-одному" и "один-ко-многим" (в Примере Кода 7.13 приведены наиболее интересные фрагменты). Имея определенную строку, относящуюся к заказу, клиент получает номер этого заказа. Затем, используя этот номер, клиент получает список всех позиций по заказу. В приведенном примере используется интерфейс `Collection` (приведены только наиболее важные фрагменты кода).

Пример Кода 7.13 Фрагмент кода приложения OrderClient

```
import java.util.*;
public class OrderClient {
   public static void main(String[] args) throws Exception {
      ...
      Order myOrder;
      LineItem myItem;
      // получение номера заказа по записи о товаре
      System out println (" Line item: " + myItem.getPrimaryKey() + "
            belongs to Order " + myItem.getOrder().getPrimaryKey());
      ...
      // получение списка (Collection) товаров для данного заказа
      System.out.print("The order " + myOrder getPrimaryKey() + "
            contains these items:");
      Iterator iterator = myOrder.getLineItems().iterator();
      while (iterator.hasNext()) {
         LineItem item = (LineItem)
                  javax.rmi.PortableRemoteObject.narrow
                  (iterator.next(), LineItem.class);
         System.out.print(" " + item.getPrimaryKey());
   }
   ...
```

Соотношения между типами Java и SQL

Когда вы разрабатываете Компонент EJB для работы с существующей базой данных, вы должны сопоставить типы данных Java в вашем приложении с SQL-типами данных БД.

Inprise-реализация Контейнера EJB использует стандартне отображение между SQL- и Java-типами, принятое в JDBC. JDBC определяет набор обобщенных SQL-типов, соответствующих наиболее часто используемым типам данных SQL. При разработке вашего Компонента, предназначенного для моделирования существующих баз данных,

рекомендуется использовать эти правила отображения типов (эти типы определены в классе `lava.sql.Types`.)

Таблица 7.1 содержит описание соответствия между типами SQL и Java, принятого в JDBC

Таблица 7.1 Базовое соответствие типов SQL и Java

Тип Java	тип JDBC SQL
boolean	BIT
byte	TINYINT
char	CHAR(1)
double	DOUBLE
float	REAL
int	INTEGER
long	BIGINT
short	SMALLINT
String	VARCHAR
java.math.BigDecimal	NUMERIC
byte[]	VARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.util.Date	TIMESTAMP
java.io.Serializable	VARBINARY

Хотя все реляционные базы данных используют типы SQL, все же существуют значительные отличия между различными реализациями этих типов. Даже если речь идет о типах с одинаковой семантикой, иногда для них используются различные имена. Например, типу `boolean` Java в Oracle соответствует `NUMBER(1,0)`, в Sybase - `BIT`, а в DB2 - `SMALLINT`.

Когда Inprise-Контейнер EJB создает таблицы баз данных для ваших Компонентов, он автоматически сопоставляет поля Компонента со столбцами таблиц БД. Контейнер обязан знать, какие именно типы SQL использовать при работе с конкретной базой данных. Следствием этого является различное отображение между SQL- и Java-типами для различных баз данных. В таблице 7.2 содержаться схемы отображения для Oracle, Sybase/MSSQL и DB2.

Таблица 7.2 Отображение Java в SQL для Oracle, Sybase/MSSQL и DB2

типJava	Oracle	Sybase/MSSQL	DB2
boolean	NUMBER(1,0)	BIT	SMALLINT
byte	NUMBER(3,0)	TINYINT	SMALLINT
char	CHAR(1)	CHAR(1)	CHAR(1)
double	NUMBER	FLOAT	FLOAT
float	NUMBER	REAL	REAL
int	NUMBER(10,0)	INT	INTEGER

Таблица 7.2 Отображение Java в SQL для Oracle, Sybase/MSSQL и DB2

типJava	Oracle	Sybase/MSSQL	DB2
long	NUMBER(19,0)	NUMERIC(19,0)	BIGINT
short	NUMBER(5,0)	SMALLINT	SMALLINT
String	VARCHAR(2000)	TEXT	VARCHAR(2000)
java.math.BigDecimal	NUMBER(38)	DECIMAL(28,28)	DECIMAL
byte[]	RAW(2000)	IMAGE	BLOB
java.sql.Date	DATE	DATETIME	DATE
java.sql.Time	DATE	DATETIME	TIME
java.sql.Timestamp	DATE	DATETIME	TIMESTAMP
java.util.Date	DATE	DATETIME	TIMESTAMP
java.io.Serializable	RAW(2000)	IMAGE	BLOB

В таблице 7.3 содержаться схемы отображения для JDataStore, Informix и Interbase.

Таблица 7.3 Отображение Java в SQL для JDataStore, Informix и Interbase

тип Java	JDatastore	Informix	Interbase
boolean	BOOLEAN	SMALLINT	SMALLINT
byte	SMALLINT	SMALLINT	SMALLINT
char	CHAR(1)	CHAR(1)	CHAR(1)
double	DOUBLE	FLOAT	DOUBLE PRECISION
float	FLOAT	SMALLFLOAT	FLOAT
int	INTEGER	INTEGER	INTEGER
long	LONG	DECIMAL(19,0)	NUMBER(15,0)
short	SMALLINT	SMALLINT	SMALLINT
String	VARCHAR	VARCHAR(2000)	VARCHAR(2000)
java.math.BigDecimal	NUMERIC	DECIMAL(32)	NUMBER(15,15)
byte[]	OBJECT	BYTE	BLOB
java.sql.Date	DATE	DATE	DATE
java.sql.Time	TIME	DATE	DATE
java.sql.Timestamp	TIMESTAMP	DATE	DATE
java.util.Date	TIMESTAMP	DATE	DATE
java.io.Serializable	OBJECT	BYTE	BLOB

Глава

8

Управление транзакциями

В это главе описано, как управлять транзакциями. В ней рассматриваются следующие темы:

- Что такое транзакция
- Сервисы менеджера транзакций
- Компоненты EJB и транзакции
- Использование API транзакций
- Управление исключениями
- Поддержка JDBC
- Распределенные транзакции

Что такое транзакция

Прикладные программисты пользуются многими преимуществами, разрабатывая свои приложения на платформах, подобных Java 2 Enterprise Edition (J2EE), которые поддерживают транзакции. Ориентированные на использование транзакций системы разработки существенно упрощают процесс создания приложений, так как они освобождают разработчика от необходимости заботиться об обеспечении устойчивости к сбоям и необходимости обслуживания многих клиентов одновременно. Для использования таких транзакций не обязательно работать с одной базой данных или на одном компьютере. В распределенной транзакции могут одновременно участвовать несколько баз данных, работающих под управлением разных серверов.

Обычно программист разбивает все необходимые действия в рамках своего приложения на последовательность групп действий. Каждая такая группа выполняется в контексте отдельной транзакции. При

выполнении приложения системы, с которыми оно взаимодействует, например, СУБД, гарантируют, что каждая такая группа - отдельная транзакция - завершается независимо от всех других процессов. Если произошла ошибка, СУБД откатывает транзакцию и отменяет все выполненные в ее контексте действия.

Характеристики транзакций

Как правило, говоря о транзакциях, подразумевают взаимодействие с базами данных. Любая операция с базами данных выполняется в контексте транзакции. Все транзакции характеризуются следующими свойствами:

- Атомарность (Atomicity)
- Связность (Consistency)
- Изолированность (Isolation)
- Сохраняемость (Durability)

Обычно для обозначения этих свойств используется аббревиатура ACID.

Транзакция часто содержит несколько операций. Атомарность подразумевает, что для завершения транзакции необходимо успешное выполнение всех ее операций. Если хотя бы одна из операций не выполнена, не должны быть выполнены все остальные.

Под связностью понимается связность информации в базе данных. Транзакции должны переводить базу данных из одного связного (устойчивого) состояния в другое. Транзакция должна обеспечивать семантическую и физическую связность данных в БД.

Изолированность требует, чтобы каждая транзакция рассматривала себя как единственную транзакцию, выполняемую в настоящий момент над базой данных. Параллельно с ней могут выполняться и другие транзакции, но текущая транзакция не должна видеть промежуточные изменения, внесенные другими транзакциями, до тех пор, пока они не подтвердили (commit) свои действия. Вследствие независимости вносимых изменений, транзакции могли бы видеть некорректные результаты, являющиеся только подмножеством изменений, вносимых другими транзакциями. Изолированность транзакции позволяет избежать такого рода проблем.

Понятие изолированности тесно связано с понятием совместного доступа. Существует понятие уровня изоляции, и чем он выше, тем более ограничены возможности внесения параллельных изменений. Наиболее высокий уровень изоляции транзакций подразумевает выполнение всех транзакций в порядке очереди (сериализация). Другими словами, содержимое базы данных меняется так, как если бы новая транзакция могла начаться только после завершения предыдущей. Тем не менее, многие приложения способны работать с более низким уровнем изоляции транзакций и, соответственно, с

большими возможностями параллельного доступа к данным. Как правило, такие приложения позволяют использовать большое количество одновременных транзакций, даже если некоторые из них выполняют чтение данных, которые частично изменены другими транзакциями.

И, наконец, сохраняемость означает, что изменения, внесенные подтвержденными транзакциями, остаются в базе данных независимо от последующих сбоев. Сохраняемость гарантирует, что подтвержденные изменения сохраняются в базе данных, даже если произошел сбой после завершения транзакции и что база данных может быть восстановлена после такого сбоя.

Поддержка транзакций

Контейнер поддерживает так называемые "плоские" (flat) транзакции, но не вложенные (nested) транзакции. Контейнер также неявно распространяет контекст транзакции. Последнее означает, что пользователь не должен явно использовать контекст транзакции в качестве аргумента при вызове методов, так как эту работу берет на себя Контейнер.

Программистам следует иметь в виду, что страницы JSP и сервлеты хотя и могут быть использованы в качестве клиентов, не могут рассматриваться как транзакционные компоненты. Все действия по управлению транзакциями должны быть выполнены на уровне соответствующего Компонента EJB. Все действия, связанные с управлением транзакцией, осуществляются Компонентом EJB и его Контейнером в ответ на запрос клиента.

Сервисы менеджера транзакции

Существуют два сервиса (две службы) управления транзакциями, которые входят в комплект поставки Inprise EJB Container - реализация фирмой Inprise Java Transaction Service (JTS) и Inprise Integrated Transaction Service (ITS).

JTS является службой, которая входит в состав Контейнера и используется для управления транзакцией по умолчанию. JTS предоставляет большие возможности, но содержит и существенные ограничения - она не поддерживает тайм-ауты транзакции. Транзакция может никогда не завершится вследствие некоторых причин. В отсутствие тайм-аута, некорректная транзакция может "зависнуть". Если с такой транзакцией сопоставлен session-Компонент, то Контейнер никогда не сможет удалить его, так как Контейнер не может удалять session-Компоненты, связанные с транзакцией. Очевидно, что это существенно ограничивает возможности Контейнера по эффективному управлению ресурсами.

Сервис управления транзакциями Inprise ITS поддерживает тайм-ауты транзакции. Фактически он позволяет вам определить период тайм-аута по умолчанию и его максимальную величину. (Более подробная информация содержится в документации по ITS.)

При использовании ITS в качестве сервиса транзакции Контейнер может выполнить деактивизацию session-Компонента, даже если он сопоставлен с некорректной транзакцией, которая "зависла" и никогда не будет завершена. Контейнер выполнит деактивизацию по завершении периода тайм-аута, который рано или поздно наступит. Затем, если к этому Компоненту не будет других обращений клиентов в течение времени продолжительности его собственного тайм-аута, Контейнер удалит этот Компонент. Это предотвращает ситуацию, когда Компонент начинает транзакцию и блокирует память или другие ресурсы вследствие ее "зависания". При использовании ITS Контейнер может удалить session-Компонент вне зависимости от того, сопоставлен он с транзакцией или нет, при условии, что и транзакция, и Компонент превысят соответствующие периоды тайм-аута.

Компоненты EJB и транзакции

Как Компоненты EJB, так и их Контейнеры способны управлять транзакциями. Технология EJB позволяет изменять информацию в нескольких базах данных в контексте одной транзакции. Данные могут находиться не только в различных БД, но и на разных компьютерах в сети. Технология EJB использует стиль управления транзакциями, который отличается от традиционного стиля (так называемое декларативное управление). Компонент EJB устанавливает свои атрибуты транзакции на стадии поставки. Эти атрибуты транзакции определяют, будет ли управление транзакцией осуществляться Контейнером, или это возьмет на себя сам Компонент. При использовании традиционного стиля за все аспекты управления транзакциями отвечает само приложение. Это подразумевает выполнение следующих операций:

- Создание объекта "транзакция".
- Явное начало транзакции.
- Передача и отслеживание контекста транзакции.
- Подтверждение транзакции после внесения всех изменений.

Это требует от разработчика Приложения глубоких знаний о том, как управлять транзакциями с начала и до конца - код такого приложения достаточно сложен, труден для написания и может содержать ошибки. При декларативном стиле управления транзакциями Контейнер берет на себя выполнение большинства, если не всех, необходимых операций. Контейнер начинает и завершает транзакцию, а также поддерживает контекст транзакции в течение ее существования. Это существенным образом упрощает задачу разработчика, особенно для транзакций в распределенных средах.

Основы транзакций, управляемых Компонентом или Контейнером

Говорят, что используется транзакция, управляемая Компонентом (Bean-managed transaction, BMT), когда сам Компонент программно задает начало и конец транзакции. Если же компонент доверяет управление своей транзакцией Контейнеру, и Контейнер определяет параметры транзакции на основании инструкций, находящихся в Дескрипторе Поставки, тогда говорят об использовании транзакций, управляемых Контейнером (Container-managed transaction, CMT). Как правило, эта информация помещается в Дескриптор Поставки Сборщиком Приложений (Application Assembler).

Как stateful, так и stateless Session-Компоненты могут использовать оба вида управлениями транзакциями, но не в одно и то же время. Entity-Компоненты могут использовать только CMT. Какой вид управления транзакциями использовать для session-Компонента, решает его разработчик.

Компонент EJB может захотеть самостоятельно управлять транзакцией, если нужно начать транзакцию как часть одной операции, а завершить - как часть другой. Тем не менее, такое решение может вызвать проблемы, если произошел вызов метода, начинающего транзакцию, но не вызван метод, который ее завершает.

Следует стремиться использовать транзакции, управляемые Контейнером, хотя возможен и другой способ. Их использование требует написание меньшего количества кода и уменьшает вероятность появления ошибок. Кроме того, Компонент с CMT проще настраивать и интегрировать с другими Компонентами.

Атрибуты транзакции

Session-Компонент с BMT должен определить атрибуты транзакции и сопоставить их с каждым своим методом. Эти атрибуты предоставляют Контейнеру информацию о том, как он должен управлять транзакциями, в которых участвует данный Компонент. С каждым из методов Компонента может быть сопоставлен один из шести возможных атрибутов. Это сопоставление выполняется на этапе поставки Сборщиком Приложений или Поставщиком (Deployer).

Вот эти атрибуты:

- `Reguired` - этот атрибут гарантирует, что ассоциированный с ним метод выполняется в контексте глобальной транзакции. Если клиент, вызывающий этот метод, уже создал контекст транзакции, то используется именно он. Если нет, то Контейнер автоматически начинает новую транзакцию. Этот атрибут позволяет использовать любое количество Компонентов и координировать их работу в контексте одной и той же глобальной транзакции.

- RequiresNew - этот атрибут используется, когда метод не должен быть сопоставлен с уже существующей транзакцией. Он гарантирует, что Контейнер всегда начинает новую транзакцию.
- Supports - использование этого атрибута позволяет избежать использования глобальных транзакций. Его следует использовать только тогда, когда метод объекта обращается только к одному транзакционному ресурсу или не использует их вовсе, а также не вызывает другие Компоненты EJB. Работа в таком режиме используется исключительно для повышения производительности приложения, так как позволяет избежать дополнительных расходов, связанных с поддержкой глобальной транзакции. Тем не менее, если этот атрибут установлен, но не существует глобальная транзакция, то при вызове такого метода Контейнер начнет локальную транзакцию, которая будет завершена по окончании вызова метода.
- NotSupported - этот атрибут также разрешает отказаться от использования глобальных транзакций. Когда он установлен, метод не должен быть включен в глобальную транзакцию. Вместо этого Inprise Контейнер EJB приостанавливает действие глобальной транзакции, а затем при вызове этого метода начинает локальную транзакцию, которая завершается вместе с выполнением этого метода.
- Mandatory - не рекомендуется использовать этот атрибут. Его поведение очень похоже на поведение атрибута Reguired, но клиент, вызывающий метод, обязан уже иметь контекст транзакции. Если же его не существует, Контейнер возбуждает исключительную ситуацию javax.transaction.TransactionRequiredException. Использование этого атрибута приводит к тому, что Компонент становится менее гибким с точки зрения композиции с другими Компонентами, так как делается предположение о существовании контекста транзакции клиента.
- Never - не рекомендуется использовать этот атрибут. Если же он используется, Inprise-Контейнер начинает локальную транзакцию при вызове этого метода. Как обычно, локальная транзакция завершается вместе с методом.

При нормальной работе следует использовать только два атрибута - Reguired и RequiresNew. Атрибуты Supports и NotSupported предусмотрены исключительно для достижения максимальной производительности. Атрибуты Mandatory и Never не рекомендуется использовать вследствие их влияния на возможность создавать композиции Компонентов. Кроме того, если Компонент хочет взаимодействовать с транзакцией и реализует интерфейс javax.ejb.SessionSynchronization , тогда Сборщик Приложения или Поставщик могут использовать только атрибуты Reguired, RequiresNew или Mandatory. Эти атрибуты гарантируют, что Контейнер вызывает метод только в контексте глобальной транзакции, что необходимо для выполнения синхронизации.

Локальные и глобальные транзакции

Локальная транзакция - это транзакция, которая управляется Менеджером ресурсов. Глобальной транзакцией называется транзакция, управляемая глобальным Менеджером транзакций в составе ITS Transaction Service.

Границы глобальной транзакции определяются Компонентом с BMT путем обращения к методам интерфейса `javax.transaction.UserTransaction`. Когда транзакция управляется Контейнером, он использует каждый клиентский вызов для определения границ транзакции. Управление транзакцией происходит в декларативном стиле, в соответствии со значением атрибутов транзакции из Дескриптора Поставки. Атрибуты транзакции также определяют, является транзакция локальной или глобальной.

При определении того, нужно ли использовать локальную или глобальную транзакцию при использовании CMT, Контейнер руководствуется определенными правилами. В общем случае Контейнер вызывает метод в контексте локальной транзакции после того, как он убедится в отсутствии существования глобальной транзакции. Он также убеждается в том, что не следует начинать новую глобальную транзакцию и что атрибуты транзакции задают режим CMT. Контейнер автоматически помещает вызовы метода в локальную транзакцию при выполнении одного из следующих условий:

- Если атрибут транзакции имеет значение `NotSupported` и Контейнер видит, что необходим доступ к ресурсам базы данных.
- Если атрибут транзакции установлен в значение `Supports` и Контейнер определил, что вызов метода происходит не в контексте глобальной транзакции.
- Если атрибут транзакции установлен в значение `Never` и Контейнер видит, что необходим доступ к ресурсам базы данных.

Локальные транзакции Inprise-Контейнера EJB характеризуются следующим:

- Локальные транзакции поддерживают методы `setRollbackOnly()` и `getRollbackOnly()` интерфейса `javax.ejb.EJBContext`.
- Локальные транзакции поддерживают тайм-ауты для транзакций и для соединений с базами данных.
- Локальные транзакции имеют преимущества с точки зрения производительности приложений.

Использование API транзакций

Все транзакции используют Java Transaction API (JTA). Когда для управления транзакциями используется режим CMT, границы

транзакции определяются автоматически и вызовы JTA API выполняются Контейнером; код вашего Компонента не содержит этих вызовов.

Если Компонент сам управляет своими транзакциями (BMT), то он должен использовать интерфейс `javax.transaction.UserTransaction` JTA. Этот интерфейс позволяет клиенту либо Компоненту обозначить границы транзакции. Компоненты EJB, которые используют BMT, получают этот интерфейс с помощью вызова метода `EJBContext.getUserTransaction()`.

В дополнение к этому, все транзакционные клиенты для получения ссылки на этот интерфейс могут использовать JNDI. Для этого нужно создать начальный контекст JNDI, как показано в следующей строке кода:

```
javax.namig.Context context = new javax.naming.InitialContext();
```

После того, как создан объект `InitialContext`, для него вызывается метод JNDI `lookup()`, как показано ниже.

Пример Кода 8.1 Получение интерфейса UserTransaction

```
javax.transaction.UserTransaction utx =
(javax.transaction.UserTransaction)
   context.lookup("java:comp/UserTransaction");
```

Обратите внимание, что Компонент EJB может получить ссылку на интерфейс `UserTransaction` через объект `EJBContext`. Это возможно потому, что Компонент наследует ссылку на объект `EJBContext`. Таким образом, Компонент EJB может просто использовать метод `EJBContext.getUserTransaction()` вместо того, чтобы создавать InitialContext, а затем вызывать для него метод `lookup()`. Тем не менее, транзакционный клиент, который не является Компонентом EJB, должен использовать средства поиска JNDI.

Когда Компонент или клиент получает ссылку на интерфейс `UserTransaction`, он может затем стартовать свои собственные транзакции и управлять ими. Другими словами, Вы можете использовать методы интерфейса `UserTransaction` для начала, подтверждения или отката транзакции. Вы начинаете транзакцию с помощью метода `begin()`, а затем вызываете метод `commit()` для подтверждения внесенных изменений или метод `rollbuck()` для отмены всех изменений транзакции и восстановления состояния базы данных, которое существовало до начала транзакции. Вызовы методов, реализующих логику транзакции, помещаются между вызовами `begin()` и `commit()`.

Управление исключениями

Если в процессе транзакции возникла ошибка, Компонент EJB может возбуждать как системные, так и свои собственные исключительные ситуации. Исключения, специфические для Приложения, говорят об

ошибках, связанных с бизнес-логикой и предназначены для обработки вызывающим приложением. Системные исключения, такие, как ошибки времени выполнения, имеют более общий характер и могут быть обработаны как приложением, так и Компонентом или Контейнером Компонента.

Компонент EJB объявляет возможные исключения обоих уровней в `throws`-списке методов интерфейсов `home` и `remote`. Вы должны перехватывать и обрабатывать эти исключения в блоках `try/catch` в вашей программе при вызове методов Компонента.

Системные исключения

Компонент возбуждает исключительные ситуации `java.ejb.EJBException` (или `java.rmi.RemoteException`) для сигнализации о неожиданном сбое на системном уровне. Например, такое исключение возбуждается, если невозможно установить соединение с базой данных. Исключение `java.ejb.EJBException` является runtime-исключением и не обязано присутствовать в throws-списке бизнес-метода Компонента.

Системные исключения обычно подразумевают необходимость выполнения отката транзакции. Как правило, это выполняет Контейнер. В других случаях, особенно при использовании BMT, откат транзакции должен быть выполнен клиентом.

Исключения, специфические для приложения

Такие исключения возбуждаются при возникновении ошибки, специфической для приложения - другими словами, вследствие ошибок бизнес-логики, а не возникновения системных проблем. Такими исключениями являются исключения, отличные от исключения `java.ejb.EJBException`. Эти исключения являются проверяемыми исключениями, что означает, что вы обязаны проверить их наличие при вызове метода, который потенциально способен их возбудить.

Бизнес-методы Компонента возбуждают исключения уровня приложения для сигнализации о возникновении таких условий, как недопустимое входное значение или значение вне допустимого диапазона. Например, метод, который списывает средства со счета, может возбудить такое исключение, чтобы показать, что состояние счета не позволяет выполнить такую операцию. Во многих случаях клиент обрабатывает такие ошибки без выполнения отката целой транзакции.

Приложение или вызывающая программа получает именно то исключение, которое было возбуждено, что позволяет ему точно определить вид ошибки. Когда происходит такое исключение, экземпляр Компонента не выполняет автоматически откат клиентской транзакции. Клиент имеет всю необходимую информацию, чтобы принять необходимые действия для исправления возникшей ситуации, не прибегая к откату транзакции. Конечно, клиент может явно откатить транзакцию.

Обработка исключений приложения

Вследствие того, что исключения уровня приложения говорят об ошибке в логике программы, предполагается, что вы как клиент занимаетесь их обработкой. Хотя возникновение исключений может потребовать отката транзакции, это не выполняется автоматически. Вы часто имеете возможность исправить ситуацию, хотя иногда необходимо откатить и завершить транзакцию.

Разработчик Компонента обязан гарантировать, что в случае продолжения транзакции клиентом не происходит разрушения целостности данных. Если Разработчик не может гарантировать этого, тогда Компонент помечает транзакцию как предназначенную для отката после ее завершения.

Откат транзакции

Когда клиент получает извещение о возникновении исключительной ситуации, в первую очередь ему следует проверить, не помечена ли текущая транзакция как "rollback only". Например, клиент может получить исключение `javax.transaction.TransactionRolledbackException`.

Это говорит о том, что произошла ошибка в вспомогательном классе Компонента и транзакция завершена или помечена как "rollback only".

В общем случае клиент не знает значение контекста транзакции, который сопоставлен с Компонентом. Вызываемый Компонент может выполняться в своем собственном контексте транзакции, отличающемся от контекста транзакции вызывающего приложения, или он может выполняться в том же контексте транзакции.

Если Компонент выполняется в том же контексте, что и вызывающая программа, тогда Компонент либо его Контейнер может пометить транзакцию как предназначенную для отката. Когда это выполняет Контейнер, клиенту следует прекратить выполнение всех операций в контексте этой транзакции. Обычно клиент, использующий декларативный стиль управления транзакцией, получает соответствующее исключение, такое, как `javax.transaction.TransactionRolledbackException`. Напоминаем, что декларативный стиль управления транзакциями подразумевает, что управление всеми аспектами берет на себя Контейнер.

Клиент, который представляет собой Компонент EJB, может вызвать метод `javax.ejb.EJBContext.getRollBackOnly()` для определения, помечена ли транзакция для отката или нет.

При использовании режима BMT - то есть транзакций, явно управляемых клиентом - клиенту следует откатить транзакцию с помощью вызова метода `rollback` интерфейса `java.transaction.UserTransaction`.

Возможности для продолжения транзакции

Когда транзакция не помечена как "rollback only", клиент имеет следующие три возможности:

- Откатить транзакцию.
- Передать обработку этого события дальше, возбудив новое исключение или повторно возбудив то же самое исключение.
- Внести изменения и продолжить выполнение транзакции. Это может подразумевать повторное выполнение части кода транзакции.

Когда клиент получает исключение для транзакции, не помеченной для отката, наиболее безопасным решением будет откатить транзакцию. Для этого клиент либо помечает транзакцию как "rollback only", либо, если он сам начал эту транзакцию, выполняет явный ее откат с помощью вызова метода `rollback()`.

Клиент также может возбудить свою собственную исключительную ситуацию или повторно возбудить первоначальное исключение. В любом случае, клиент откладывает вопрос принятия решения, как поступить с данной транзакцией. Тем не менее, в общем случае предпочтительным является вариант, когда решение принимается как можно раньше.

Наконец, клиент может продолжить выполнение транзакции. Он может проанализировать полученное исключение и решить, стоит ли вызвать тот же метод повторно с другими аргументами. При этом следует всегда иметь в виду, что такая повторная попытка является потенциально опасной. Вы не имеете достаточно информации или гарантий того, что Компонент EJB способен правильно работать в таком режиме.

Тем не менее, клиент, который обращается к stateless session-Компоненту, может повторно вызвать тот же метод с гораздо меньшими опасениями (конечно, если он определил причину возникновения исключительной ситуации). Это связано с тем, что Компонент не имеет состояния, и проблема некорректного состояния Компонента просто не возникает.

Поддержка JDBC

В контексте транзакции обычно выполняется доступ к одному или нескольким источникам данных, причем, как правило, с целью изменения данных в них. Для выполнения такого рода действий, участвующие в транзакции Компоненты EJB должны установить связь с базой (или базами) данных. Для установления такой связи Контейнер использует JDBC.

Inprise-Контейнер EJB реализует интерфейс `DataSource` JDBC 2.0. Это означает, что Компонент может использовать интерфейс `javax.sql.DataSource`, а не `javax.sql.DriverManager`, хотя интерфейс DriverManager все еще поддерживается для совместимости с предыдущими версиями.

Рекомендуется использовать интерфейс DataSource вследствие его более высокой производительности. Работа с этим интерфейсом не требует наличия драйверов JDBC 2.x; приложение может использовать существующие драйвера JDBC 1.x. Кроме того, при замене драйвера JDBC 1.x на JDBC 2.x, в приложение не требуется вносить никаких изменений.

Чтобы ваш Компонент использовал интерфейс DataSource (или использовал его в режиме CMP), вам необходимо указать параметры источника данных в Дескрипторе Поставки, и Компонент должен использовать средства JNDI. Взаимодействие с JNDI и конфигурирование источников данных берет на себя Контейнер. Пример Кода 8.2 показывает, как вы могли бы использовать источники данных в программе.

Пример Кода 8.2 Использование DataSource в программе

```
javax.sql.DataSource ds;
try
{
   javax.naming.Context ctx = (javax.naming.Context) new
      javax.naming.InitialContext();
   ds =  (javax.sql.DataSource)ctx.lookup(java:comp/jdbc/
            SavingsDataSource");
} catch (javax.naming.NamingException exp){
   exp.printStackTrace();
}
```

Задание DataSource

Вы помещаете информацию об источнике данных в Дескриптор Поставки. Необходимо указать три обязательных элемента:

- URL источника данных. При работе с интерфейсом DataSource, URL мог бы выглядеть так:

  ```
  jdbc:oracle:thin:@avicenna:1521:avi73a
  ```

- Jndi-имя источника данных.
- Имя класса jdbc-драйвера.

Полное описание этих элементов приведено в разделе "Источники данных" главы 9 на странице 9-9.

Пример Кода 8.3 содержит пример спецификации источника данных.

Пример Кода 8.3 Спецификация DataSource в Дескрипторе Поставки

```
<deployment-descriptor>
<datasource>
      <res-ref-name>jdbc/SavingsDataSource</res-ref-name>
      <url>jdbc:oracle:thin:@avicenna:1521:avi73a</url>
```

```
        <username>scott</username>
        <password>tiger</password>
        <driver-class-name >oracle.jdbc.driver.OracleDriver</driver-
                    class-name>
    </datasource>
</deployment-descriptor>
```

Если вы для установки соединения с базой данных используете интерфейс DriverManager, а не DataSource, то вы должны добавить к URL префикс inprise:its_direct, как показано ниже

```
jdbc:inprise:its_direct:oracle:thin:@avicenna:1521:avi73a
```

Кроме того, при использовании интерфейса DriverManager, в командной строке запуска Контейнера вы должны указать системное свойство jdbc.drivers. Например, для работы с драйвером Oracle вы могли бы задать следующую командную строку:

```
vbj -Djdbc.drivers=oracle.jdbc.driver.OracleDriver
            com.inprise.ejb.Container test bank_beans.jar -jts -jns
```

При использовании интерфейса DataSource нет необходимости указывать значение этого свойства, если вы указываете параметр driver-class-name в Дескрипторе Поставки.

Вы можете указать в Дескрипторе Поставки два необязательных дополнительных элемента, описывающих источник данных - имя пользователя и его пароль. Они не являются обязательными, поскольку могут являться частью URL. Существует и третий необязательный элемент, который называется "диалект" (dialect). Диалект - это имя базы данных, которая содержит хранимые данные приложения, и Контейнер EJB Inprise использует это имя для поддержки сохранения состояния. Например, если вы хотите указать, что для этого нужно использовать Sybase, вы могли бы включить следующую строку в ваш Дескриптор Поставки:

```
<dialect>sybase</dialect>
```

Когда этот параметр присутствует, Контейнер автоматически создает необходимые таблицы для указанной базы данных.

Всегда помните, что Контейнер использует свойства Дескриптора Поставки в момент своего запуска; в процессе работы Компонент никогда не меняет их значения.

Управление соединениями с базой данных и пулами

Для того, чтобы получить доступ к информации в базах или источниках данных, Компонент EJB должен сначала установить с ними соединение. Установление нового соединения с базой данных требует существенных затрат времени и должно выполняться только в случае необходимости.

Iprise-реализация Контейнера EJB обеспечивает механизм кеширования соединений с базами данных, что позволяет Компонентам повторно

использовать ранее установленные соединения. Эта схема существенно уменьшает время, необходимое для управления соединением и повышает производительность.

Как работает механизм кеширования соединения? После того, как Компонент устанавливает соединение с базой данных, Контейнер помещает это соединение в пул на определенное количество времени (это количество времени является настраиваемым параметром). Соединение содержится в пуле до истечения указанного интервала времени, а затем удаляется. При использовании CMP, пул соединений автоматически используется Контейнером.

Для выполнения кеширования соединений вы должны указать некоторую информацию для источника данных. Вы также указываете режимы управления тайм-аутом для неиспользуемых соединений. Тем не менее, в настоящее время вы не можете изменять другие опции, такие, как размер пула. Предусмотрены два системных свойства для управления тайм-аутом:

- Свойство `ITSJDBCidle_timeout` позволяет вам указать величину интервала тайм-аута для кешированных соединений JDBC. По определению его величина равна 10 минутам. Вы можете указать другое значение этого интервала (в секундах) непосредственно в командной строке. Например, следующая строка устанавливает интервал, равный 20 минутам:

```
Vbj -DITSJDBCidle_timeout=1200 com.inprise.ijb.Container myContainer
        beans.jar -jts -jns
```

- Свойство `ITS_timeout` управляет частотой, с которой сервис транзакции проверяет, не истек ли интервал тайм-аута для кешированных соединений. По умолчанию это 5 минут (300 секунд), и вы можете установить любое другое значение в командной строке, как показано ниже:

```
Vbj -DITS_timeout=600 -DITSJDBCidle_timeout=1200
        com.inprise.ijb.Container myContainer beans.jar -jts -jns
```

Ниже описано, как Контейнер управляет пулом соединения. В общем случае Контейнер устанавливает соединения тогда, когда это действительно необходимо, а не просто при выполнении определенных операций. Например, он не устанавливает соединение при начале транзакции, он ждет первого обращения к методу `getConnection()` интерфейса `javax.sql.DataSource`. Этот вызов сопоставляет транзакцию с соединением. Контейнер может использовать как соединение из пула, так и создать новое, если это необходимо. Когда выполняются другие вызовы метода `getConnection()` в контексте той же транзакции, Контейнер использует первоначальное соединение. Когда Компонент вызывает метод `commit()`, Контейнер проверяет, относится ли `commit()` к используемому соединению. После того, как транзакция завершена, Контейнер продолжает держать это соединение во внутреннем кеше. Если другая транзакция нуждается в соединении с тем же источником данных, Контейнер использует кешированное соединение. Контейнер поддерживает пул соединений автоматически. Код установления

соединения с базой данных мог бы выглядеть так, как показано в Примере Кода 8.4:

Пример Кода 8.4 Использование пула соединений

```
Context ctx = (Context) new InitialContext();
ds = (DataSource)ctx.lookup("java:comp/jdbc/SomeDataSource");
ds.getConnection();
// some database work
ds.getConnection();  // вы получаете то же самое соединение
```

Для выполнения действий, специфических для конкретной RDBMS, таких, как подготовка выполнения запросов, Контейнер EJB полагается на JDBC-драйвер. Контейнер игнорирует большинства JDBC-вызовов, за исключением обращения к методу `java.sql.Connection.setAutoCommit()`. Он перехватывает это обращение, так как это не разрешено в контексте глобальной транзакции. Кроме того, режим `AUTOCOMMIT` для JDBC-соединения, возвращаемого Inprise DataSource, имеет значение OFF, и Компонент не может установить его в значение ON. При установленном `AUTOCOMMIT`=OFF Контейнер Inprise подтверждает все изменения в конце транзакции, а не после того, как изменения были внесены.

Уровни изоляции транзакции

Если уровень изоляции транзакции не установлен явно, Контейнер использует уровни изоляции JDBC-драйвера и RDBMS по умолчанию. Например, Oracle поддерживает два уровня изоляции транзакций: `TRANSACTION_READ_COMMITTED` и `TRANSACTION_SERIALIZABLE`. Первый из них используется по умолчанию, и Контейнер работает в этом режиме, если в Дескрипторе Поставки не указано другое значение.

Дескриптор Поставки может содержать пять значений уровней изоляции транзакций. Эти значения должны быть набраны заглавными буквами, а уровень их поддержки зависит от конкретной RDBMS. Вот эти значения:

- `TRANSACTION_NONE`
- `TRANSACTION_READ_COMMITTED`
- `TRANSACTION_READ_UNCOMMITTED`
- `TRANSACTION_REPEATABLE_READ`
- `TRANSACTION_SERIALIZABLE`

После того, как в Дескрипторе Поставки задан определенный уровень изоляции для источника данных, этот уровень используется для всех соединений с этим источником данных. По вопросам достижения максимальной производительности обращайтесь к документации по конкретной системе. Если вы не уверены, какой уровень изоляции выбрать, рекомендуется использовать уровень изоляции по умолчанию.

Более подробная информация приведена в главе 9 "Поставка Компонентов EJB".

Распределенные транзакции

Inprise-Контейнер EJB поддерживает распределенные транзакции, то есть такие транзакции, которые пересекают границы платформ, операционных систем и Виртуальных Машин Java.

Двухфазное подтверждение

Транзакции, в которых задействовано несколько баз данных, используют двухфазный процесс подтверждения транзакции. Он гарантирует, что транзакция корректно вносит изменения во все базы данных, участвующие в ней. Если она не может изменить все базы данных, то не изменяется ни одна.

Двухфазное подтверждение выполняется за два шага. Первый шаг называется фазой подготовки. Транзакция опрашивает все участвующие базы данных, и они должны сигнализировать о том, что они готовы подтвердить изменения и завершить транзакцию. Второй шаг - шаг собственно внесения изменений - выполняется только в том случае, если все базы данных подтвердили свою готовность завершить транзакцию.

Как ITS, так и JTS поддерживают неоднородные распределенные транзакции и двухфазное подтверждение для однородных транзакций, используя встроенный координатор транзакций RDBMS.

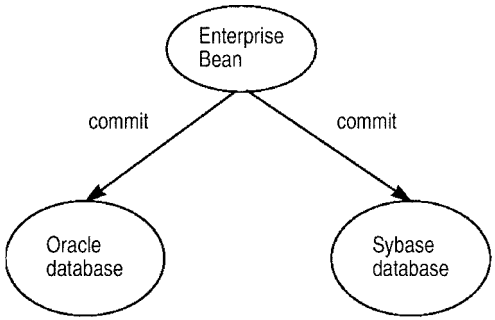
По умолчанию JTS не разрешает нескольким базам данных (ресурсам) участвовать в глобальной транзакции, хотя он и поддерживает двухфазный процесс подтверждения без возможности восстановления. Если вы хотите использовать JTS в режиме двухфазного подтверждения, вы должны установить значения флага `EJBAllowUnrecoverableCompletion`, равное True. В этом случае Контейнер в процессе подтверждения транзакции выполняет однофазное подтверждение для каждой из участвующих в ней баз данных. Работать в этом режиме следует очень осторожно, так как в случае сбоя нет возможности восстановления состояния базы данных.

Для поддержки двухфазного подтверждения гетерогенных (неоднородных) транзакций сервис ITS должен быть интегрирован с поддержкой протокола XA для конкретной базы данных (см. документацию по ITS для получения более подробной информации). В настоящий момент двухфазное подтверждение транзакций требует использования native-библиотек, то есть кода, написанного на C или C++. В будущем, когда появятся JDBC-драйвера, поддерживающие протокол XA, ITS и Контейнеры EJB позволят использовать в одной транзакции несколько баз данных.

На рисунке 8.1 показано, как Компонент и Контейнер управляют двухфазным завершением транзакции для неоднородных баз данных. Напоминаем, что для этого необходимо установить флаг

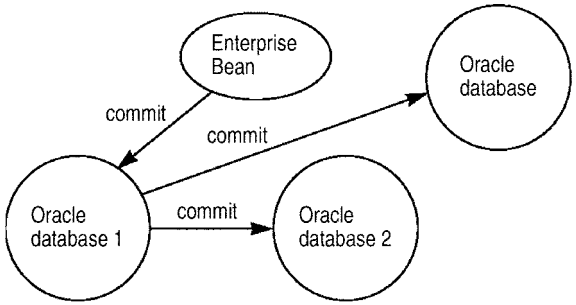
EJBAllowUnrecoverableCompletion и восстановление после сбоя невозможно.

Рис. 8.1 Двухфазное завершение транзакции для неоднородных баз данных



Двухфазное подтверждение для однородных баз данных требует выполнения настроек сервера RDBMS. В то время, как Контейнер управляет подтверждением транзакции для первой базы данных, сервер RDBMS берет на себя задачу подтверждения для всех других баз данных.

Рис. 8.2 Двухфазное подтверждение для однородных баз данных



Вам необходимо обратиться к руководству по конкретному серверу RDBMS.

Глава

9

Поставка Компонентов EJB

В главе рассмотрены следующие основные темы:

- Поставка Компонентов EJB: первые шаги
- Создание файла Дескриптора Поставки
- Задание свойств среды выполнения EJB
- Создание EJB jar-файла
- Установка вашего EJB jar-файла в Контейнер

Поставка Компонентов EJB: первые шаги

Поставка Компонентов EJB в общем случае требует выполнения следующих шагов:

1 Создание файла Дескриптора Поставки в XML-формате, совместимого со спецификацией Sun EJB 1.1.

2 Если необходимо, редактирование свойств среды, которые потребуются во время исполнения.

3 Создание EJB jar-файла, содержащего Дескриптор Поставки и все классы, необходимые для работы EJB (класс Компонента, remote-интерфейс, home-интерфейс, стабы и скелеты, класс главного ключа в случае Entity-Компонента и другие связанные с ними классы). Вы можете сделать это с помощью стандартной утилиты Java jar.

4 Поставка вашего Компонента в Контейнер EJB - либо с помощью Консоли Inprise Application Server, либо с помощью командной строки.

Создание файла Дескриптора Поставки

Для создания или изменения Дескриптора Поставки вы можете использовать Редактор Дескриптора Поставки, входящий в состав Inprise Application Server, или любой другой XML-редактор. Если вы используете стандартный XML-редактор, вы должны быть уверены, что ваши действия удовлетворяют всем требованиям спецификации EJB 1.1. Для того, чтобы ваш Компонент мог содержаться в Контейнере EJB, вы должны создать специальный файл, который называется ejb-inprise.xml.

Редактор Дескриптора Поставки Inprise Application Server создает Дескриптор Поставки, который соответствует следующим требованиям спецификации EJB 1.1:

- Он имеет XML-формат.
- Он соответствует DTD, объявленному в спецификации EJB 1.1.
- Он соответствует всем семантическим правилам DTD. Вы не должны выучить эти правила, редактор обеспечивает это самостоятельно.
- Он ссылается на DTD, используя следующий оператор:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise
JavaBeans 1.2//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_2.dtd">
```

Мы настоятельно рекомендуем использовать Редактор Дескриптора Поставки Inprise Application Server по следующим причинам:

- Он автоматически создает XML-файл Дескриптора Поставки, и от вас не требуется знание XML.
- Он соответствует семантике Sun DTD. Вы не обязаны знать эти правила, редактор использует их самостоятельно по мере ввода вами данных.
- Он автоматически добавляет в отдельный файл некоторые расширения, специфические для Inprise-реализации.
- По мере ввода вами информации он позволяет вам знать, какие данные являются обязательными.

Для получения более подробной информации о редакторе Дескриптора Поставки обратитесь к документу "*Руководство Пользователя Inprise Application Server*". В этой секции показаны фрагменты содержимого Дескриптора Поставки. Если вы хотите ознакомиться с примером полного Дескриптора Поставки, обратитесь к каталогу примеров, расположенному в каталоге инсталляций. Вам нужен подкаталог, который называется "META-INF":

```
[install_directory]/examples/bank/META-INF/ejb-jar.xml
```

Для просмотра содержимого специфических для Inprise-реализации расширений, см.

```
[install_directory]/examples/bank/META-INF/ejb-inprise.xml
```

Роль Дескриптора Поставки

Дескриптор Поставки служит для предоставления информации о каждом Компоненте EJB, который помещен в конкретный EJB jar-файл. Он предназначен для того, чтобы с ним работал пользователь EJB jar-файла. Создание Дескриптор Поставки - задача разработчика Компонента. Поставщик (Deployer) может изменять некоторые из атрибутов в процессе поставки. Вы также можете изменять Дескриптор Поставки после того, как он установлен в Контейнере.

Информация, находящаяся в Дескрипторе Поставки, используется для задания значений атрибутов Компонентов EJB. Эти атрибуты определяют поведение Компонента при его взаимодействии с конкретной средой исполнения. Например, когда Вы устанавливаете атрибуты транзакции Компонента, они определяют каким образом Компонент работает в контексте транзакции - и вообще, участвует ли он в транзакции. Дескриптор Поставки содержит следующую информацию:

- Информацию о типах - другими словами, имена классов - для home и remote интерфейсов и класса реализации.
- JNDI-имена, включая имя, под которым зарегистрирован home-интерфейс.
- Поля, сохранением значений которых управляет Контейнер.
- Профили транзакций, которые определяют поведение Компонента при выполнении транзакции.
- Атрибуты безопасности, которые определяют права доступа к Компоненту.
- Информация, специфическая для Inprise-реализации, такая, как информация об источнике данных.

Типы информации в Дескрипторе Поставки

Всю информацию в Дескрипторе Поставки можно разделить на две основные группы:

- Информация о структуре Компонента. Она описывает его структуру и объявляет все внешние связи этого Компонента. Эта информация является необходимой. В общем случае она не может быть изменена, поскольку это могло бы привести к потере функциональности Компонента.
- Информация для Сборщика Приложений. Она определяет, каким образом Компонент(ы), помещенные в файл ejb-jar, могут быть собраны вместе для формирования нового Компонента или приложения. Эта информация не является обязательной. Изменения этой информации не влияют на поведение Компонента, хотя могут повлиять на изменение поведения приложения в целом.

Информация о структуре

Разработчик Компонента EJB должен для каждого Компонента в файле ejb-jar определить структурную информацию для каждого Компонента. Часть этой информации является обязательной для всех Компонентов, другие части могут относиться только к session -Компонентам, entity-Компонентам или только к entity-Компонентам с CMP.

Для любых Компонентов должны быть заданы:

- Имя Компонента
- Класс Компонента
- Home-интерфейс Компонента
- Remote-интерфейс Компонента
- Тип Компонента
- Параметры среды
- Ссылки на фабрики ресурсов (если используются источники данных)
- Ссылки на другие Компоненты (если это необходимо)
- Ссылки на Роли безопасности

Для каждого session-Компонента EJB должны быть определены:

- Тип сохранения состояния Компонента
- Тип задания границ транзакций

Для каждого entity-Компонента EJB должны быть определены:

- Тип управления сохранением (persistence)
- Класс главного ключа Компонента

Для каждого session-Компонента EJB с CMP должны быть определены:

- Поля, управляемые Контейнером

Информация для сборки Приложений

Вы можете указать любой из перечисленных ниже видов информации. Любой из них не является обязательным на стадии сборки приложений, но должен быть задан на стадии поставки.

- Разрешение внешних ссылок Компонента
- Роли безопасности
- Права доступа к методам
- Привязки ссылок на роли безопасности
- Атрибуты транзакций

При выполнении процесса сборки Приложения или его поставки вы можете изменять следующую структурную информацию:

- Значение параметров среды. Сборщик Приложений (Application Assembler) может изменить значения существующих и/или добавить новые значения параметров.
- Поля описаний. Сборщик Приложений может изменить существующие либо создать новые элементы описаний.

Вы не можете изменить любые другие виды структурной информации; тем не менее, вы можете изменить любую информацию, относящуюся к сборке приложений, на стадии поставки.

Для получения подробного описания содержимого Дескриптора Поставки и его семантики обращайтесь к спецификации Sun EJB 1.1.

Подробное описание работы с редактором Дескриптора Поставки находится в документе "*Руководство пользователя Inprise Application Server*".

Безопасность

Разработчик - обычно Сборщик Приложений - определяет следующую информацию в Дескрипторе Поставки:

- Роли безопасности
- Права доступа к методам
- Связи между ссылками на роли безопасности и самими ролями

Роли безопасности

При работе с Дескриптором Поставки, Разработчик может определить одну или несколько ролей безопасности. Это задает рекомендуемые роли безопасности для клиента Компонента.

Права доступа к методу

Эти элементы Дескриптора Поставки позволяют Разработчику определить права доступа к методам Компонента. С методами remote- и home-интерфейсов Компонентов EJB могут быть сопоставлены те или иные роли безопасности.

Связи между ссылками на роли безопасности и самими ролями

Если определены роли безопасности, то разработчик должен определить связи между ними и ссылками на них; для этого используются специальные элементы Дескриптора Поставки.

Специфическая для Inprise-реализации информация, необходимая для поставки Компонента

Элементы, специфические для Inprise-реализации, находятся в файле с именем ejb-inprise.xml. Этот файл расположен в том же каталоге, что и файл ejb-jar.xml. В этот файл вы можете поместить следующую информацию:

- bean-home-name
- значение тайм-аута для stateful session-Компонента
- информация об источниках данных
- информация о CMP
- отображение JNDI-имен, используемых для Компонента EJB, на реальные имена

Файл содержит два главных раздела:

- <enterprise-beans>
- <datasource-definitions>

Ниже приведен DTD файла расширений Inprise.

```
<!ELEMENT inprise-specific (enterprise-beans, datasource-
                definitions?)>
   <!ELEMENT enterprise-beans (session|entity)+>
   <!ELEMENT session (ejb-name, bean-home-name,
                timeout?, ejb-ref*, resource-ref*, property*)>
   <!ELEMENT entity (ejb-name, bean-home-name,
                ejb-ref*, resource-ref*, cmp-info?, property*)>
   <!ELEMENT ejb-ref (ejb-ref-name, jndi-name)>
   <!ELEMENT resource-ref (res-ref-name, jndi-name, cmp-resource?)>
   <!ELEMENT datasource-definitions (datasource*)>
   <!ELEMENT datasource (jndi-name, url, username?, password?,
                isolation-level?, driver-class-name?, jdbc-property*,
                     property*)>
   <!ELEMENT jdbc-property (prop-name, prop-value)>
   <!ELEMENT property (prop-name, prop-type, prop-value)>
   <!ELEMENT cmp-info (description?, database-map?, finder*)>
   <!ELEMENT database-map (table, column-map*)>
   <!ELEMENT finder (method-signature, where-clause, load-state?)>
   <!ELEMENT column-map (field-name, column-name, ejb-ref-name?)>
   <!ELEMENT cmp-resource (#PCDATA)>
   <!ELEMENT method-signature (#PCDATA)>
   <!ELEMENT where-clause (#PCDATA)>
   <!ELEMENT load (#PCDATA)>
   <!ELEMENT prop-name (#PCDATA)>
```

```
<!ELEMENT prop-type (#PCDATA)>
<!ELEMENT prop-value (#PCDATA)>
<!ELEMENT field-name (#PCDATA)>
<!ELEMENT column-name (#PCDATA)>
<!ELEMENT table (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT ejb-name (#PCDATA)>
<!ELEMENT bean-home-name (#PCDATA)>
<!ELEMENT timeout (#PCDATA)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT jndi-name (#PCDATA)>
<!ELEMENT res-ref-name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT username (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!ELEMENT isolation-level (#PCDATA)>
<!ELEMENT driver-class-name (#PCDATA)>
```

Некоторые из этих элементов являются обязательными, некоторые - нет.

Вы должны создать соответствующую session- или entity-секцию для каждого Компонента EJB, который вы планируете поместить в jar-файл EJB. Например, если вы создали пять session-Компонентов, вы должны создать пять session-секций в файле ejb-inprise.xml.

Информация о Компоненте

В секции для каждого Компонента вы помещаете дополнительную информацию об этом Компоненте. Специфические для Inprise-реализации элементы описаны в таблице 9.1.

Таблица 9.1 Информация в Дескрипторе Поставки, специфическая для Inprise-реализации

Элемент	Описание
ejb-name	Этот элемент является ссылкой на Компонент и используется для доступа ко всем данным Компонента. Он обеспечивает привязку информации о Компоненте в файле ejb-jar.xml к информации в файле ejb-inprise.xml. Это имя должно быть уникальным в xml-файле. Вы можете поставлять один и тот же Компонент, но под разными именами. Эта информация является обязательной.
bean-home-name	Это имя, которое использует клиент для получения доступа к Компоненту EJB с помощью средств JNDI. Эта информация является обязательной.
timeout	Это величина тайм-аута для stateful session-Компонента. Эта информация не является обязательной и используется только для stateful session-Компонентов.
ejb-ref	Определяет соответствие между JNDI-именами, используемыми Компонентом, и реальными именами. Эта информация не является обязательной.
resource-ref	Определяет связь между JNDI-именами для источников данных и реальными именами. Эта информация не является обязательной.

Таблица 9.1 Информация в Дескрипторе Поставки, специфическая для Inprise-реализации

Элемент	Описание
property	Задает пару "имя=значение". Позволяет определить свойства Компонента EJB, используемые его Контейнером.

Секция entity-Компонентов может содержать дополнительные элементы. Если этот Компонент использует CMP, вы можете задать cmp-info.

Таблица 9.2 Элементы в cmp-info

Элемент	Описание
description	Описывает режим сохранения Компонента. Информация необязательная.
database-map	Определяет соответствие между Компонентом с CMP и конкретной таблицей в базе данных. Он отображает поля Компонента на реальные столбцы в таблице. По умолчанию используется следующее отображение: имя Компонента = имя таблицы; имя поля = имя столбца. Эта информация не является обязательной.
finders	Этот элемент влияет на операции, выполняемые методами поиска. Он задает часть WHERE SQL-оператора, используемого Контейнером для выполнения поиска и извлечения записей из базы данных. Состояние параметра load по умолчанию имеет значение true, и Контейнер извлекает значения всех управляемых им полей при выполнении поиска. Эта информация не является обязательной.
tuned-writes	Этот элемент позволяет гарантировать, что выполняются только минимально необходимые изменения в базе данных. В частности, если entity-Компонент не был изменен в данной транзакции, не выполняется его запись в базу данных. Эта информация не является обязательной.

Пример элементов session-Компонента

```
<inprise-specific>
   <enterprise-beans>
      <session>
         <ejb-name>sort</ejb-name>
         <bean-home-name>ejb/sort</bean-home-name>
         <ejb-ref>
            <ejb-ref-name>ejb/sort</ejb-ref-name>
            <jndi-name>cosnm/mySortHome</jndi-name>
         </ejb-ref>
      </session>
   </enterprise-beans>
</inprise-specific>
```

Пример entity-Компонента с CMP

```
<inprise-specific>
   <enterprise-beans>
      <entity>
         <ejb-name>checking</ejb-name>
         <bean-home-name>accounts/savings</bean-home-name>
         <cmp-info>
            <database-map>
               <table>Checking_Accounts</table>
            </database-map>
            <finders>
               <method-signature>findAccountsLargerThan(float
                              balance)</method-signature>
               <where-clause>balance > :balance</where-clause>
            </finders>
         </cmp-info>
         <resource-ref>
            <res-ref-name>jdbc/CheckingDatasource</res-ref-name>
            <jndi-name>oracleDataSource/CheckingDS</jndi-name>
            <cmp-resource>True</cmp-resource>
         </resource-ref>
      </entity>
   </enterprise-beans>
   <datasource-definitions>
      <datasource>
         <jndi-name>oracleDataSource/CheckingDS</jndi-name>
         <url>jdbc:oracle:thin:@avicenna:1521:avi73a</url>
         <username>scott</username>
         <password>tiger</password>
      </datasource>
   </datasource-definitions>
</inprise-specific>
```

Источники данных (DataSource)

Если вы используете базы данных, вы должны задать некоторые элементы в секции DataSource. Секция DataSource представляет собой набор элементов, которые позволяют вам обратиться к базам данных.

Эта секция предоставляет Контейнеру информацию, как создать экземпляр источника данных.

Она задает соответствие между источником данных, конкретным Компонентом EJB и элементами подсекции resource-ref, определенной в спецификации Sun EJB 1.1. Специфические для Inprise-реализации расширения в секции DataSource определяют соответствие между res-ref-name и реальным источником данных, что необходимо для установки соединения. Одним из элементов подсекции resource-ref

является res-ref-name. Это является JNDI-именем для конкретного источника данных. Связать JNDI-имя и конкретный источник данных Контейнеру помогают специфические для Inprise-реализации расширения.

В таблице 9.3 описаны элементы, присутствующие в секции <datasource>.

Таблица 9.3 Специфическая для Inprise-реализации информация об источнике данных в Дескрипторе Поставки

Элемент	Описание
jndi-name	Это JNDI-имя, под которым зарегистрирован источник данных. Это - обязательная информация.
url	URL - универсальный локатор ресурса - служит для организации доступа к источнику данных в сети. Его составные части: <JDBC driver><server><port><host> Информация является обязательной, если вы используете базу данных.
username	Имя пользователя для доступа к базе данных. Может отсутствовать. Может также являться частью URL. Параметр должен присутствовать, если элемент <res-auth> имеет значение "Container".
password	Пароль для доступа к базе данных. Может отсутствовать. Может также являться частью URL. Параметр должен присутствовать, если элемент <res-auth> имеет значение "Container".
isolation-level	См. раздел, приведенный после таблицы. Параметр может отсутствовать. Если вы не указали его, в качестве значения по умолчанию используется значение по умолчанию конкретного JDBC-драйвера.
driver-class-name	Это имя класса драйвера, который необходимо загрузить. Это то же самое имя, которое может быть указано в системном свойстве "jdbc.drivers". Значение может отсутствовать.
jdbc-property	Это список произвольных строковых пар имя/значение, передаваемых методу DriverManager.getConnection(). С этим свойством не сопоставлен никакой тип. Может отсутствовать.
property	Это пара вида имя=значение. Представляет собой свойства, используемые Контейнером для взаимодействия с источником данных. Может отсутствовать.

Инфо Для Компонентов EJB, которые используют CMP, для установки соединения с базой данных необходимо указать имя пользователя и пароль, следовательно, вы должны указать их либо с использованием вышеперечисленных элементов, либо как часть URL. Для Компонентов EJB со значением параметра <res-auth>, равном "Bean", имя пользователя и пароль могут отсутствовать.

Уровни изоляции транзакций

Уровни изоляции транзакций определяются в терминах решения стандартных проблем, возникающих при параллельном доступе к данным. Кроме того, уровни изоляций транзакций зависят от того, что позволяет вам ваш JDBC-драйвер. Вы должны хорошо знать эти уровни изоляции.

Допустимые проблемы - уровни изоляции транзакций определены в спецификации ANSI SQL. ODBC, JDBC и EJB следуют этому стандарту. Под уровнем изоляции транзакций понимается степень независимости одновременно выполняемых транзакций друг от друга при работе в многопользовательских системах. В идеале, нужно было бы иметь так называемые "сериализованные" транзакции. Это означает, что результат одновременного выполнения любого количества транзакций был бы тем же самым, что и результат последовательного выполнения этих транзакций. Существуют три основные проблемы, к которым сводится нарушение таких идеальных правил управления транзакциями.

- Dirty Read. Транзакция t1 изменяет запись базы данных. Затем транзакция t2 читает эту запись. Транзакция t1 откатывается, в результате t2 видит вариант записи, который никогда не существовал.
- Non-Repeatable Read. Транзакция t1 читает запись. Затем транзакция t2 изменяет запись, и t1 читает эту запись снова. Транзакция t1 считала одну и ту же запись дважды и получила два различных значения.
- Phantoms. Транзакция t1 читает набор записей, который удовлетворяет определенным условиям поиска. Затем транзакция t2 добавляет одну или несколько записей, которые удовлетворяют тому же условию. Если транзакция t1 повторно выполняет чтение, то она "увидит" записи, которые не существовали раньше. Эти записи называются фантомными записями.

Возможны следующие уровни изоляции транзакций:

- TRANSACTION_READ_COMMITTED. Не допускает Dirty Read, но не препятствует возникновению двух остальных проблем.
- TRANSACTION_READ_UNCOMMITTED. Не препятствует появлению ни одной из проблем.
- TRANSACTION_REPEATABLE_READ. Возможно появление фантомных записей, но не остальных проблем.
- TRANSACTION_SERIALAZABLE. Невозможна ни одна из проблем.
- TRANSACTION_NONE. Отсутствует поддержка транзакций.

Задание свойств среды исполнения EJB

Свойства среды могут быть изменены непосредственно перед установкой Компонента EJB (jar-файла) в Контейнер. Например, если некоторый Компонент EJB должен быть сопоставлен с конкретным пользователем, но вы не хотите устанавливать эту связь на уровне исходного кода Компонента, вы можете изменить соответствующее свойство среды в процессе поставки.

Создание EJB jar-файла

Для того, чтобы выполнить поставку Компонента, вы должны создать EJB jar-файл, который содержит все необходимые классы для функционирования этого Компонента. В один EJB jar-файл можно поместить несколько Компонентов.

Содержимое EJB jar-файла

Для создания EJB jar-файла, который содержит все необходимые для работы Компонента файлы, вы можете использовать стандартную утилиту Java, которая называется jar. Необходимо включить в него следующие файлы:

- Файлы Дескриптора Поставки в XML-формате - файл, требуемый согласно спецификации EJB 1.1, и файл специфический для Inprise-реализации. Оба эти файла должны находится в определенном каталоге и иметь определенный формат. Файлы должны называться ejb-jar.xml и ejb-inprise.xml и находиться в подкаталоге META-INF.
- Файлы откомпилированных классов (в виде байт-кода). Вот эти файлы:
  - Реализация Компонента
  - Remote
  - Home
  - Совместимые с IIOP стабы и скелеты
  - Главный ключ, если вы используете Entity-Компонент EJB
  - Дополнительные классы, которые необходимы для работы. Утилита jar помещает в jar-файл любые файлы, которые имеют расширение .class.

Установка EJB jar-файла в Контейнер

Для выполнения поставки Компонента вы должны установить содержащий его jar-файл в Контейнер EJB. Это можно сделать двумя способами:

- с помощью Эксперта Поставки, входящего в состав Консоли Inprise Application Server.
- с использованием утилиты командной строки.

В один EJB Контейнер можно установить несколько jar-файлов. В настоящее время Компонент EJB, созданный с помощью инструментов Inprise, может быть установлен только в Inprise EJB-Контейнер.

Синтаксис

Ниже приведен синтаксис командной строки:

```
prompt% vbj com.inprise.ejb.Container <server_name> <jar_file>
<jar_file> <jar_file> <options>
```

Пример

Ниже приведен пример использования командной строки для Поставки EJB jar-файла.

```
prompt% vbj com.inprise.ejb.Container test beans.jar -jts -jns -jdb
```

Глава

10

Инструментальные средства EJB

В процессе разработки и поставки Компонентов EJB вы можете использовать различные средства:

- компилятор (кодегенератор) java2iiop
- verify
- dd2xml

Их использование описано в последующих разделах.

java2iiop

Этот компилятор используется для генерации IIOP-совместимых стабов и скелетов на базе home-интерфейса. Вы должны сгенерировать стабы и скелеты для того, чтобы клиент мог обратиться к серверу.

Сгенерированные стабы и скелеты соответствуют спецификации CORBA 2.3. Это означает поддержку RMI-over-IIOP в терминах передачи объектов по значению (object-by-value). Комплексные типы данных Java (такие, как словари, векторы и другие) могут передаваться с использованием IIOP при помощи новых типов данных IDL "value", в соответствие со спецификацией CORBA 2.3. Это обеспечивает настоящее взаимодействие между продуктами разных производителей.

Когда его использовать?

Ниже приведено описание шагов, которое объяснит вам, когда нужно использовать компилятор java2iiop:

1 Напишите все классы, необходимые для работы Компонента.

2 Откомпилируйте эти классы с использованием стандартного компилятора Java (javac).

3 Запустите компилятор java2iiop, указав в качестве входа класс home-интерфейса. В результате будут сгенерированы IIOP-совместимые стабы и скелеты. Как и все другие java-файлы, эти стабы и скелеты должны быть в виде байт-кода перед их помещением в jar-файл. Для автоматической их генерации в виде байт-кода укажите опцию -compile при обращении к java2iiop.

4 Упакуйте все эти классы - включая стабы и скелеты - в jar-файл.

В качестве входных вы можете указать как один, так и несколько home-классов. Если вы ввели несколько имен файлов, убедитесь, что они разделены пробелами. Более подробная информация об использовании этой команды находится в документе "Руководство программиста" VisiBroker for Java, раздел "Определение CORBA-интерфейсов на Java".

Инфо Компилятор java2iiop не поддерживает переопределение методов в стиле Caffeine.

Синтаксис

```
java2iiop [options] {filename filename filename}
```

Пример

```
java2iiop -verbose -no_examples -compile ContainerHome ContainedHome
```

Опции

В приведенной ниже таблице находятся описания некоторых из опций, доступных при работе с компилятором java2iiop. Для просмотра синтаксиса команды и списка всех опций введите следующую команду:

```
prompt> java2iiop -help
```

После ввода этой команды появится описание синтаксиса и список всех возможных опций.

Опция	Описание
-verbose	включение режима подробного описания выполненных действий.
-no_comments	запрещает добавление комментариев в сгенерированный код.
-no_examples	запрещает генерацию example-кода.
-compile	выполняет компиляцию сгенерированных файлов.

Инфо Вы можете также использовать любую из опций командной строки команды vbj. Для получения полного списка команды vbj обратитесь к документу VisiBroker for Java Reference Guide.

Verify

Используйте эту утилиту для проверки того, соответствует ли содержимое EJB jar-файла требованиям спецификации EJB 1.1. Утилита

проверяет, правильно ли построено наследование EJB-классов, соответствует ли требованиям спецификации формат Дескриптора Поставки (он должен быть в формате XML, а не в устаревшем формате спецификации EJB 1.0), и другое. Подробное описание правил вы можете найти в спецификации EJB 1.1.

Вы должны запустить утилиту verify перед установкой вашего EJB jar-файла в Контейнер EJB. Вы можете также использовать эту утилиту в процессе разработки, чтобы убедиться в отсутствии ошибок.

В качестве входных параметров утилиты могут быть указаны несколько jar-файлов.

Когда это использовать?

Ниже приведено описание шагов, которое объяснит вам, когда нужно использовать утилиту verify:

1 Создайте классы компонента.

2 Запустите компилятор java2iiop.

3 Создайте XML-Дескриптор Поставки.

4 Создайте EJB jar-файл.

5 Запустите утилиту verify.

Синтаксис

```
vbj com.inprise.ejb.util.Verify [-verbose] <ejb-jar-file>.jar ...
```

Пример

```
vbj com.inprise.ejb.util.Verify bank753.jar storexyz.jar
```

Опции

Приведенная ниже таблица содержит список и описание опций, доступных при работе с Verify.

Опция	Описание
[-verbose] <ejb-jar-file>.jar ...	включает режим подробного описания действий при проверке jar-файла.

dd2xml

Используйте эту утилиту для преобразования Дескриптора Поставки в формате EJB 1.0 к XML-Дескриптору, соответствующего требованиям спецификации EJB 1.1.

В качестве входа укажите один или несколько файлов с расширением .ser (в таких файлах содержатся Дескрипторы Поставки EJB 1.0). Результатом работы утилиты является сгенерированный XML-файл (файл с расширением .xml), который полностью соответствует спецификации EJB 1.1. Конвертор выполняет объединение нескольких входных (устаревших) Дескрипторов Поставки и помещает результат в один XML-файл с фиксированным именем. Напоминаем, что один XML-Дескриптор может содержать информацию о нескольких Компонентах EJB. Для каждого Компонента может быть указан собственный набор атрибутов, Дескриптор может описывать и session-, и entity-Компоненты. Не существует никаких ограничений по размещению Компонентов различных типов в одном jar-файле.

Когда это использовать?

Ниже приведено описание шагов, которое объяснит вам, когда нужно использовать утилиту dd2xml, если вы выполняете перекодировку Дескриптора из формата EJB 1.0 к формату EJB 1.1:

1 Извлеките ser-файлы из EJB jar-файла.

2 Запустите конвертор dd2xml, указав в качестве входа все необходимые ser-файлы. В результате будут сгенерированы два XML-файла: ejb-jar.xml и ejb-inprise.xml.

3 Поместите сгенерированные XML-файлы в каталог META-INF/.

4 Перестройте jar-файл при помощи утилиты jar.

5 Запустите утилиту verify.

Синтаксис

```
vbj com.inprise.ejb.util.dd2xml <deployment_descriptor_file>.ser ...
```

Пример

```
vbj com.inprise.ejb.util.dd2xml bank753.ser
```

Инфо Опции отсутствуют.

Приложение

A

Анализ результатов примера cart

В данном разделе приведено подробное объяснение того, что выводится (на экран) при выполнении примера cart. И сервер, и клиент запущены с включенным режимом отладки. О том, как включить режим отладки, рассказано в разделе "Использование режима отладки" на стр. ???.

В главе рассмотрены следующие основные темы:

- Обзор информации
- Дескриптор Поставки
- Список методов Компонента EJB
- Статистика Контейнера
- Взаимодействие клиента и Контейнера
- Вывод на стороне клиента
- Выполнение закупки

Обзор информации

При запуске Контейнера вы увидите следующее:

Пример Кода A.1 Выходная информация 1 для Контейнера

```
Inprise EJB Container
=====================
     server version : 0.2.0
  server build date : December 21, 1998
       java version : JDK1.1.6_Borland
        java vendor : Sun Microsystems Inc.
   java class path : c:\kodiak\ejb_ea_0_2\lib\ejb.jar
                   : C:\Inprise\APPLIC~1\bin\vbj.exe\..\..\lib\vbjorb.jar
```

```
                : C:\Inprise\APPLIC~1\bin\vbj exe\ \ \lib\vbjapp jar
                : C:\Inprise\APPLIC~1\bin\vbj exe\ \ \lib\vbj30ssl jar
                : .
                : c:\Inprise\ApplicationServer\java\bin\..\classes
                : c:\Inprise\ApplicationServer\java\bin\..\lib\classes.zip
                : c:\Inprise\ApplicationServer\java\bin\..\lib\classes.jar
                : c:\Inprise\ApplicationServer\java\bin\..\lib\rt.jar
                : c:\Inprise\ApplicationServer\java\bin\..\lib\i18n.jar
====================
Creating POA: EJB[test]cart
```

Вы можете использовать эту информацию при возникновении тех или иных проблем.

Контейнер выводит имя Компонента EJB - в нашем случае cart, экземпляр которого создается при вызове метода `create()`.

Дескриптор Поставки

Ниже приведен фрагмент, относящийся к Дескриптору Поставки

Пример Кода A.2 Выходная информация 2 для Контейнера

```
Deployment Descriptor
--Generic--
        getEnterpriseBeanClassName:  CartBean
        getHomeInterfaceClassName:   CartHome
        getRemoteInterfaceClassName: Cart
        getBeanHomeName:             cart
        getEnvironmentProperties:    {}
        getReentrant:                false
--Session--
        getSessionTimeout:           0
        getStateManagementType:      STATEFUL_SESSION
--Access--
        getAccess getMethod:         *default*
          getAccess getAllowed:      null
--Control--
        getControl.getMethod:        *default*
          getControl.getTxAttribute: TX_NOT_SUPPORTED
          getControl.getRunAsMode:   SPECIFIED_IDENTITY
          getControl.getRunAsId:     null
        getControl.getMethod:        purchase
          getControl.getTxAttribute: TX_REQUIRED
          getControl.getRunAsMode:   SPECIFIED_IDENTITY
          getControl.getRunAsId:     null
```

Дескриптор Поставки содержит атрибуты, сопоставленные с

Компонентом EJB. Эти атрибуты говорят Контейнеру, как управлять Компонентом EJB. Выходная информация показывает, что:

- Именем класса реализации является CartBean.
- Именем класса home-интерфейса является CartHome.
- Именем класса remote-интерфейса является Cart.
- JNDI-именем является cart. JNDI на самом деле не использует строки как имена. В нашем примере используется составное имя (javax.naming.CompositeName) "cart".

Свойства среды не используются, о чем говорит наличие пустого списка.

Наш Компонент EJB не является реентерабельным. Так как getReentrant() имеет значение false в нашем примере, то Контейнер понимает, что только один поток может взаимодействовать с Компонентом в любой момент времени. Если запрос клиента приходит в тот момент, когда Компонент обслуживает другой запрос, Контейнер для второго запроса возбуждает исключение java.rmi.RemoteException. Обратите внимание, что session-Компонент предназначен для обслуживания только одного клиента. Одним из следствий этого является невозможность для приложения выполнения loopback-обращений к такому Компоненту.

Информация, относящаяся к Session-Компоненту, содержит информацию о величине тайм-аута. Значение 0 говорит о том, что значение тайм-фута не установлено и Контейнер никогда не удаляет Компонент самостоятельно. Если бы значение было равно 10 или 100, то Контейнер мог бы удалить Компонент по истечении 10 или 100 секунд. Завершает секцию Session сообщение о том, что тип управления состоянием определяется значением "stateful session bean", что означает, что для Компонента необходимо сохранять его состояние и что экземпляр Компонента EJB может быть деактивизирован, а затем активизирован заново. Короче, наш Компонент cart является stateful Session-Компонентом.

Информация о правах доступа говорит о том, что используется управление доступом по умолчанию - контроль прав доступа не выполняется.

Control-дескриптор установлен для использования режима по умолчанию: для всех методов, которые не перечислены здесь, транзакции не поддерживаются. Далее сказано, что для метода purchase() необходимо использовать транзакции в режиме Required. Таким образом, вы можете определить одинаковые параметры для всех методов, а затем указать другие режимы для каждого метода отдельно. Дополнительно секция Control содержит информацию об идентификации (identity) пользователя.

Список методов Компонента EJB

После информации, описанной в предыдущих разделах, выводится список всех методов Компонента EJB. Это 5 бизнес-методов и методы ejbCreate() и ejbRemove().

Пример Кода А.3 Выходная информация 3 для Контейнера

```
--Methods (bean)--
        0: addItem
        1: removeItem
        2: getTotalPrice
        3: getContents
        4: purchase
--Methods (create)--
        0: ejbCreate
--Methods (miscellaneous)--
        0: ejbRemove
*sc* user   sending request: CartHome._is_a
*sc* user  received request: CORBA::Object._is_a
```

Статистика Контейнера

Контейнер готов к взаимодействию с клиентом и выводит сообщение об этом.

Пример Кода А.4 Выходная информация 4 для Контейнера

```
Container [test] is ready
```

Обычно выводится дополнительная статистическая информация:

Выходная информация 5 для Контейнера

```
EJB Container Statistics
                =======================
                Time                    Fri Mar 19 10:27:11 PST 1999
                Memory (used)           514 Kb  (max 514 Kb)
                Memory (total)          1023 Kb (max 1023 Kb)
                Memory (free)           49.0%
                -----------------------
                Home                    cart
                Total created           0
                Total active            0
                =======================
```

Частота, с которой выводится эта информация, может меняться. По умолчанию это происходит примерно через каждые пять секунд. Вы можете явно указать, через какой интервал времени нужно выводить эту информацию. В нашем примере Контейнер использует 500 K

памяти. Может показаться, что это довольно много, но Контейнер реализует значительную часть функциональности системы.

Идеальной ситуацией (которая может быть отражена с помощью выводимой статистической информации) является следующая: клиент начинает взаимодействовать с Контейнером, выполняет необходимые действия, и отсоединяется. В этом случае количество используемой памяти остается постоянным. Контейнер не должен использовать все больше и больше памяти с течением времени.

Что касается объектов cart, то ни один из них еще не был создан, что и отражено при выводе статистики. Вообще, информация в этой секции относится к состоянию Компонента и говорит о том, что он в настоящий момент делает. Если бы он был активен, он мог бы находиться в различных состояниях. Он мог бы находиться в пассивном состоянии, если он был бы деактивизирован и записан во временное хранилище, или он мог бы находиться в состоянии "transactional ready". Вывод содержит "снимок" текущего состояния Компонента.

Взаимодействие клиента и Контейнера

После того, как Контейнер и клиент начинают взаимодействовать друг с другом, при выводе появляется больше информации.

Пример Кода A.5 Выходная информация 6 для Контейнера

```
*sc* user  received request: CORBA::Object.root_context
*sc* user  received request: CORBA::Object.resolve
*sc* user  received request: CORBA::Object._is_a
*sc* user  received request: CORBA::Object._is_a
*sc* user  received request: CORBA::Object._is_a
*sc* user  received request: CORBA::Object._get_EJBMetaData
*sc* user  received request: CORBA::Object._is_a
*sc* user  received request: CORBA::Object.create
*st* prepare context NOT_EXIST --[setContext:setSessionContext]-->
SET_CONTEXT
Invoking method         void
CartBean.setSessionContext(javax.ejb.SessionContext=SessionContext[id=1,status=NoTransaction,caller=null])
  caller identity       null
  run as identity       null
  completed             CartBean.setSessionContext()
*st* commit context NOT_EXIST --[setContext:setSessionContext]-->
SET_CONTEXT
*st* prepare context SET_CONTEXT --[create:ejbCreate]--> READY
Invoking method         void CartBean.ejbCreate(java.lang.String=Jack
B. Quick,
Java.lang.String=1234-5678-9012-3456, java.util.Date=Sun Jul 01
```

```
00:00:00 PDT 2001)
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
  completed             CartBean.ejbCreate()
*st* commit context SET_CONTEXT --[create:ejbCreate]--> READY
*sc* user  received request: CORBA::Object.addItem
*st* prepare context READY --[method:addItem]--> READY
Invoking method         void CartBean.addItem(Item=Book@314860)
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        addItem(The Art of Computer Programming): CartBean[name=Jack
B. Quick]
  completed             CartBean.addItem()
*st* commit context READY --[method:addItem]--> READY
*sc* user  received request: CORBA::Object.addItem
*st* prepare context READY --[method:addItem]--> READY
Invoking method         void CartBean.addItem(Item=CompactDisc@317725)
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        addItem(Kind of Blue): CartBean[name=Jack B. Quick]
  completed             CartBean.addItem()
*st* commit context READY --[method:addItem]--> READY
*sc* user  received request: CORBA::Object._get_contents
*st* prepare context READY --[method:getContents]--> READY
Invoking method         java.util.Enumeration.CartBean.getContents()
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        getContents(): CartBean[name=Jack B. Quick]
  result
java.util.Enumeration=com.inprise.ejb.util.VectorEnumeration@317933
CartBean.getContents()
*st* commit context READY --[method:getContents]--> READY
*sc* user  received request: CORBA::Object._get_totalPrice
*st* prepare context READY --[method:getTotalPrice]--> READY
Invoking method         float CartBean.getTotalPrice()
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
```

```
 transaction status   StatusNoTransaction
       getTotalPrice(): CartBean[name=Jack B. Quick]
  result                float=61.92 CartBean.getTotalPrice()
*st* commit context READY --[method:getTotalPrice]--> READY
```

Включение режима отладки задействует использование интерсепторов, и, если вы посмотрите на выводимую информацию, вы увидите отслеживание RPC-вызовов. Режим отладки включает также диагностику состояния машины. Видно, как изменяются состояния объектов и какие методы вызываются для каждого объекта.

Вывод на стороне клиента

Теперь посмотрим, какую информацию может получить клиент.

Пример Кода A.6 Выходная информация 1 для клиента

```
*sc* user   sending request: CORBA::Object.root_context
*sc* user   sending request: CosNaming::NamingContext.resolve
*sc* user   sending request: CORBA::Object._is_a
*sc* user   sending request: CORBA::Object._is_a
*sc* user   sending request: CORBA::Object._is_a
*sc* user   sending request: CORBA::Object._get_EJBMetaData
*sc* user   sending request: CORBA::Object._is_a
*sc* user   sending request: CORBA::Object.create
```

Выыполнено несколько обращений к службе имен CORBA CosNaming. Первое из них необходимо для получения доступа к CosNaming, второе - для получения объектной ссылки на объект при помощи JMDI-имени "cart". В процессе поиска происходит обращение к сервису транзакций, а затем выполняется вызов метода create(), который переводит Компонент EJB из состояния DOES NOT EXIST в состояние METHOD READY.

Кроме того, по выводимой информации для Контейнера видно, что метод ejbCreate() вызывается с несколькими аргументами - двумя строками, содержащими "Jack B. Quick" и номер кредитной карточки, а также со значением даты. Фиксируется переход объекта в состояние READY.

Продолжим анализ информации, выводимой для клиента.

Пример Кода A.7 Выходная информация 2 для клиента

```
*sc* user   sending request: Cart.addItem
*sc* user   sending request: Cart.addItem
======= Cart Summary ========
*sc* user   sending request: Cart._get_contents
Price: $49.95   Book title: The Art of Computer Programming
Price: $11.97   CompactDisc title: Kind of Blue
*sc* user   sending request: Cart._get_totalPrice
Total: $61.92
```

Клиент начинает работу с посылки двух запросов Cart.addItem(), а затем вызывает метод summarize(). При выполнении summarize() сначала выводится информационное сообщение, а затем происходит вызов метода Cart._get_contents(). Результатом этого метода является набор данных (enumeration), который возвращается клиенту. Затем метод summarize() выполняет перебор полученных данных, вызывает метод Cart._get_totalPrice() и выводит на печать список покупок с их ценами и общую стоимость заказа.

Инфо Вызов метода create() не приводит к выводу какой-либо информации на стороне сервера (Контейнера) - так уж этот метод написан, зато каждый вызов метода addItem() приводит к выводу одной строки. Вывод на стороне сервера содержит эти строки - сообщение о вызове этого метода и название покупки - например, книга "Art of Computing Programming". Появление этих строк - результат действий самого Компонента; Контейнер не выполняет вызовов, приводящих к выводу какой-либо информации на печать. Такие команды содержит сам Компонент, что позволяет отслеживать выполняемые действия. Когда вы создаете свой Компонент EJB и хотите получить какую-либо информацию о выполнении программы среди отладочной информации для Контейнера, вы должны в код Компонента поместить соответствующие операторы.

Выполнение закупки

Пример Кода A.8 Выходная информация 3 для клиента

```
*sc* user   sending request: Cart.removeItem
*sc* user   sending request: Cart.addItem
======= Cart Summary ========
*sc* user   sending request: Cart._get_contents
Price: $11.97   CompactDisc title: Kind of Blue
Price: $44.99   Book title: Programming with VisiBroker
*sc* user   sending request: Cart._get_totalPrice
Total: $56.96
============================
*sc* user   sending request: Cart.purchase
Could not purchase the items:
        PurchaseProblemException: Purchasing not implemented yet!
```

На следующем этапе выполнения программы происходит обращение к методу removeItem(). Клиент удалает книгу "Art of Computing Programming" из списка покупок, а вместо этого покупает другую книгу - "Programming with VisiBroker". Программа опять вызывает метод summarize(), что опять приводит к вызовам методов Cart._get_contents() и Cart._get_totalPrice(). Наконец, мы видим, что была сделана попытка вызвать метод purchase(). Это привело к возникновению исключительной ситуации, которая информирует, что этот метод еще не реализован.

Пример Кода A.9 Выходная информация 7 для Контейнера

```
*sc* user  received request: CORBA::Object.removeItem
*st* prepare context READY --[method:removeItem]--> READY
Invoking method         void CartBean.removeItem(Item=Book@317ed1)
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        removeItem(The Art of Computer Programming):
CartBean[name=Jack B. Quick]
completed             CartBean.removeItem()
*st* commit context READY --[method:removeItem]--> READY
*sc* user  received request: CORBA::Object.addItem
*st* prepare context READY --[method:addItem]--> READY
Invoking method         void CartBean.addItem(Item=Book@318040)
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        addItem(Programming with VisiBroker): CartBean[name=Jack B.
Quick]
  completed             CartBean.addItem()
*st* commit context READY --[method:addItem]--> READY
*sc* user  received request: CORBA::Object._get_contents
*st* prepare context READY --[method:getContents]--> READY
Invoking method         java.util.Enumeration CartBean.getContents()
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        getContents(): CartBean[name=Jack B. Quick]
  result
java.util.Enumeration=com.inprise.ejb.util.VectorEnumeration@318243
CartBean.getContents()
*st* commit context READY --[method:getContents]--> READY
*sc* user  received request: CORBA::Object._get_totalPrice
*st* prepare context READY --[method:getTotalPrice]--> READY
Invoking method         float CartBean.getTotalPrice()
  caller identity       null
  run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
        getTotalPrice(): CartBean[name=Jack B. Quick]
  result                float=56.96 CartBean.getTotalPrice()
*st* commit context READY --[method:getTotalPrice]--> READY
```

```
*sc* user  received request: CORBA::Object.purchase
*st* prepare context READY --[method:purchase]--> READY
*tx* Dispatcher.invoke: tx.begin(serverTransaction)
*sc* user   sending request: CosTransactions::TransactionFactory._is_a
*sc* user  received request: CORBA::Object._is_a
*tx* Dispatcher.invoke: tx.register_synchronization()
*st* commit context READY --[afterBegin:afterBegin]--> TX_READY
Invoking method         void CartBean.purchase()
  caller identity       null
  run as identity       null
  transaction attribute TX_REQUIRED
  transaction status    StatusActive
        purchase(): CartBean[name=Jack B. Quick]
  call threw exception  CartBean.purchase()
*tx* Dispatcher.invoke: tx.rollback(serverTransaction)
*st* commit context TX_READY --
[afterCompletionRollback:afterCompletion]--> READY
*st* commit context READY --[method:purchase]--> READY
*sc* user  received request: CORBA::Object.remove
*st* prepare context READY --[remove:ejbRemove]--> BEFORE_FINALIZE
Invoking method         void CartBean.ejbRemove()
  caller identity       null
run as identity       null
  transaction attribute TX_NOT_SUPPORTED
  transaction status    StatusNoTransaction
  completed             CartBean.ejbRemove()
*st* commit context READY --[remove:ejbRemove]--> BEFORE_FINALIZE
              EJB Container Statistics
              ========================
              Time                    Fri Mar 19 10:29:46 PST 1999
              Memory (used)           656 Kb  (max 656 Kb)
              Memory (total)          1023 Kb (max 1023 Kb)
              Memory (free)           35.0%
              ------------------------
              Home                    cart
              BEFORE_FINALIZE         1
              Total created           1
              Total active            1
              ========================
```

Вывод для Контейнера показывет, что клиент выполнил обращение к методу purchase(). Напоминаем, что атрибуты в Дескрипторе Поставки требуют выполнения этого метода в контексте транзакции. Клиент явно не начал транзакцию. Тогда транзакцию начал Контейнер, и он же обратился к regiter_synchronization() после того, как он вызвал метод afterBegin(). Затем он вызвал метод purchase() Компонента CartBean. В соответствие с требованиями спецификации Sun EJB, транзакция

откатывается, если Компонент возбудил исключительную ситуацию в коде транзакционного метода. Транзакция откатывается, после чего вызывается метод afterCompletion(). Выполнение метода purchase() завершено.

Посмотрим теперь, что при удалении происходит на стороне клиента.

Пример Кода А.10 Выходная информация 4 для клиента

```
*sc* user   sending request: Cart.remove
```

Приложение

B

Поддержка EJB 1.0

Генерация Дескриптора Поставки

Класс GenerateDescriptors генерирует Дескриптор Поставки EJB 1.0. Дескрипторы Поставки - это объекты языка Java, свойства которых описывают, как нужно поставить (deploy) конкретную программу, написанную на Java. Например, свойство может описывать, как нужно запустить программу на выполнение или как зарегистрировать ее имя в службе имен (naming service). В EJB 1.0 Дескриптор Поставки хранится в файле в сериализованном виде. Процесс построения приложения включает в себя помещение файла, содержащего сериализованный Дескриптор Поставки, в jar-файл.

Этот класс создает выходной файл, устанавливает свойства Дескриптора, а затем выполняет его запись в файл как стандартную процедуру сериализации Java. Давайте сначала ознакомимся с функцией `main()` этого класса (Пример Кода 11.1). Функция создает файл с именем cart.ser, для работы с которым создается объект типа "поток вывода". Для вывода класса Java в поток используется стандартный класс `ObjectOutputStream`. Затем происходит обращение к методу `createDescriptor()`, который создает Дескриптор Поставки и записывает его в файл cart.ser. См. Пример Кода 11.2.

Пример Кода B.1 Функция main() класса GenerateDescriptors

```
public static void main(String[] args) throws Exception {
   ObjectOutputStream objectOutput =
      new ObjectOutputStream(new FileOutputStream("cart.ser"));
   objectOutput.writeObject(createDescriptor());
}
```

При создании Дескриптора Поставки метод `createDescriptor()` инициализирует session-Дескриптор (так как CartBean является Session-

Компонентом; для Entity-Компонента была бы выполнена инициализация entity-Дескриптора) и устанавливает следующие его свойства:

- CartBean как имя класса реализации Компонента.
- CartHome как имя home-интерфейса.
- Cart как имя remote-интерфейса.
- Cart как JNDI-имя. JNDI не использует строки в качестве имен. Он использует так называемые "naming names" Java. Простейшим способом преобразования строки в naming name Java является создание объекта типа `javax.naming.CompositeName`, что и сделано в нашем примере.
- STATEFUL_SESSION как тип состояния Компонента CartBean. Это выполняется с помощью вызова функции `setStateManagementType()`. (Для stateless Session-Компонента использовалось бы значение STATELESS_SESSION). Это также устанавливает режим транзакций для Компонента EJB.
- Атрибуты транзакций. С CartBean сопоставлен атрибут TX_NOT_SUPPORTED. Метод `purchase()` получает атрибут TX_REQUIRES.
- 0 как величину тайм-аута сеанса связи. Эта величина является настраиваемым значением, которая влияет на цикл жизни Компонента. Существуют два фактора, которые управляют циклом жизни Session-Компонента:
  - Явные команды клиента. Как правило, клиент создает Компонент, использует его, а затем уничтожает.
  - Величина тайм-аута, если она задана. Если она задана, то Контейнер может самостоятельно уничтожить Компонент, если в течение определенного периода времени клиент к нему не обращался. В нашем примере величина тайм-аута в Дескрипторе Поставки установлена равной 0. Нулевое значение означает, что интервал тайм-аута не установлен и Контейнер не удалит объект, если клиент забыл это сделать. Если же установить величину тайм-аута равной 100, то если в течение 100 секунд клиент не будет обращаться к Компоненту, Контейнер его уничтожит.

Пример Кода B.2 Установка свойств Дескриптора

```
public class GenerateDescriptors {
   static DeploymentDescriptor createDescriptor() throws Exception {
   SessionDescriptor d = new SessionDescriptor();
   d.setEnterpriseBeanClassName("CartBean");
   d.setHomeInterfaceClassName("CartHome");
   d.setRemoteInterfaceClassName("Cart");
   d.setBeanHomeName(new javax.naming.CompositeName("cart"));
   d.setSessionTimeout(0);
```

```
    d.setStateManagementType(d.STATEFUL_SESSION);
    { // set the default control descriptor
       ControlDescriptor defaultControl = new ControlDescriptor(null);
       DefaultControl.setTransactionAttribute
                   (ControlDescriptor.TX_NOT_SUPPORTED);
       // set the transaction mode on purchase to TX_REQUIRES
       ControlDescriptor purchaseControl =
          new ControlDescriptor(CartBean.class.getMethod("purchase",
                   null));
          purchaseControl.setTransactionAttribute
                   (ControlDescriptor.TX_REQUIRED);
          ControlDescriptor[] controls = { defaultControl,
                   purchaseControl };
          d.setControlDescriptors(controls);
    }
    return d;
}
```