Лекция 16

SQL и ВЕБ

Серверы приложений представляют собой одну из основных новых компьютерных технологий, вызванную к жизни ростом Интернета. Серверы приложений образуют ключевой уровень архитектуры большинства коммерческих веб-сайтов. Как следует из их названия, серверы приложений предоставляют платформу для работы прикладной логики, которая определяет взаимодействие с пользователем на веб-сайте. Но серверы приложений выполняют и другую важную роль — действуя как посредники между компонентами веб-сайта со стороны Интернета (веб-серверы и инструменты управления содержимым сайта) и компонентами со стороны информационных технологий, такими как унаследованные корпоративные приложения и базы данных. В этой роли серверы приложений должны тесно сотрудничать с программным обеспечением СУБД, а языком такого сотрудничества является SQL. В этой главе исследуется роль SQL в многоуровневой архитектуре веб-сайта с применением серверов приложений.

SQL и веб-сайты: ранние реализации

Серверы приложений не всегда играли ведущую роль в архитектурах веб-сайтов. Ранние веб-сайты были направлены, в первую очередь, на донесение своего содержимого пользователям в виде статических веб-страниц. Содержимое веб-сайта было структурировано в виде ряда предопределенных веб-страниц, хранящихся в виде файлов. Веб-сервер принимал запрос от пользовательского браузера (в виде сообщений HTTP (Hypertext Transfer Protocol, протокол передачи гипертекста)), находил запрошенную страницу (или страницы) и пересылал ее с использованием того же HTTP назад браузеру для вывода на экран. Содержимое веб-страниц хранилось в формате HTML (HyperText Markup Language, язык разметки гипертекста). HTML-код конкретной страницы содержал текст и графику, выводимые на экран, а также ссылки, обеспечивающие переход от одних страниц к другим.

Прошло совсем немного времени, и требования к распространяемой через World Wide Web информации "перешагнули" статические возможности предопределен-

ных веб-страниц. Компании стали использовать веб-сайты для связи со своими клиентами, и им требовалась поддержка таких возможностей, как поиск определенного товара или принятие заказов от клиентов. Первым шагом в этом направлении стала обработка информации веб-сервером, как показано на рис. 22.1. Веб-серверы стали принимать не только запросы статических страниц, но и запросы на выполнение *сценариев*, т.е. последовательностей инструкций, предназначенных для определения того, какая информация должна быть представлена на экране клиента.

Рис. 22.1. Динамические веб-страницы при отсутствии серверов приложений

Сценарии веб-серверов часто писались на специализированных языках сценариев, таких как Perl и PHP. В простейшем виде сценарий мог выполнять очень простые вычисления (такие, как получение текущих даты и времени от операционной системы) и выводить результат как часть веб-страницы. В более сложном виде сценарий мог получать введенную пользователем на странице с формой информацию, выполнять запрос к базе данных с ее использованием и возвращать результаты этого запроса. Поскольку результат работы сценария мог меняться от одного вызова к другому, получающиеся в результате веб-страницы стали *динамическими*: их содержимое могло меняться от просмотра к просмотру, в зависимости от конкретных результатов выполнения сценария.

Языки сценариев обеспечивали самую первую возможность связи между веб-сайтами и SQL-базами данных. Сценарий мог, например, передать SQL-запрос СУБД и получить результаты для вывода их в виде веб-страницы. Однако при таком решении возникала масса других проблем. Большинство языков сценариев интерпретируемые, и выполнение сложного сценария могло потребовать много процессорного времени. Сценарии работали как отдельные процессы систем на базе UNIX или Windows, что приводило к высоким накладным расходам при ежесекундном запуске десятков или сотен сценариев. Эти и другие ограничения решения с применением сценариев привели к разработке альтернативного подхода и появлению серверов приложений как части архитектуры веб-сайта.

Серверы приложений и трехуровневые архитектуры веб-сайтов

Логичной эволюцией сценариев веб-сервера стало выделение отдельной роли *сервера приложений*, что привело к трехуровневой архитектуре, показанной на рис. 22.2. Обратите внимание на то, что многие профессионалы рассматривают веб-сервер и сервер приложений как разные уровни и считают эту архитектуру четырехуровневой, или, более обобщенно, N-уровневой. Веб-сервер отвечает за поиск и обслуживание статических веб-страниц и статических частей веб-страниц. Когда требуется определить, какая информация будет выведена пользователю или когда следует обработать полученные от пользователя данные, веб-сервер вызывает отдельный сервер приложений для выполнения необходимых действий. В случае небольшого веб-сайта сервер приложений представляет собой отдельный процесс на том же физическом сервере, где работает веб-сервер. В более общем случае, на больших веб-сайтах, веб-сервер и сервер приложений работают на двух разных компьютерах, обычно соединенных высокоскоростной локальной сетью. Независимо от конфигурации, веб-сервер передает запросы в виде сообщений серверу приложений и получает ответы в виде HTML-содержимого, которое будет выводиться на странице.

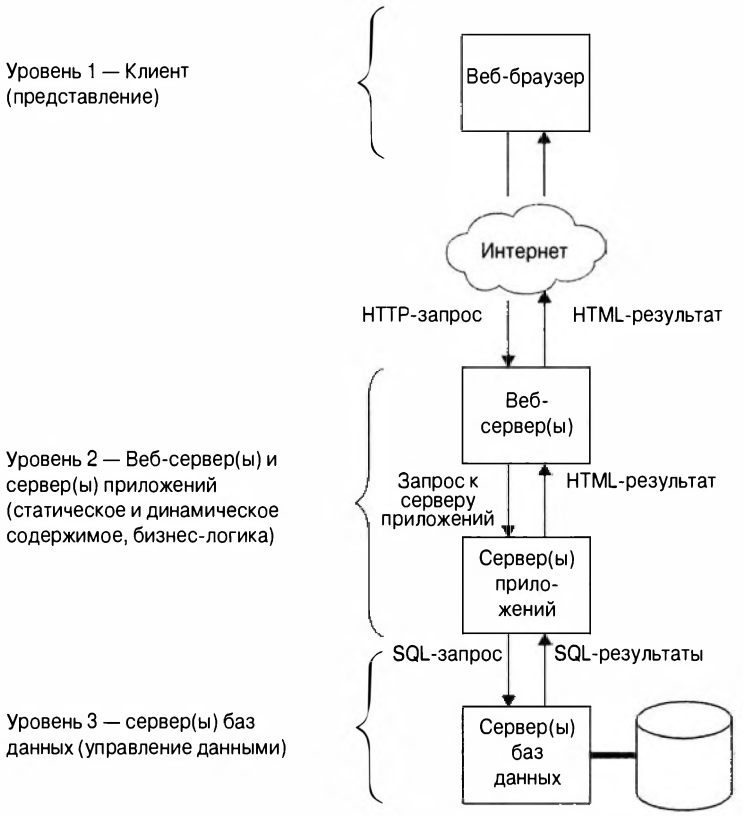

Рис. 22.2. Трехуровневая архитектура с применением сервера приложений

В первое время предлагалось большое количество разнообразных продуктов для серверов приложений. Некоторые серверы требовали, чтобы приложения были написаны на языках программирования C или C++, другие требовали использования Java. Интерфейс между сервером приложений и веб-сервером определялся интерфейсом прикладного программирования (API) двух ведущих производителей веб-серверов — Netscape и Microsoft. Но все прочие аспекты — от языка программирования до поддержки служб, предоставляемых сервером приложений и API для доступа к базам данных, — стандартизованы не были.

Разработка фирмой Sun Microsystem продукта Enterprise Java Beans (EJBs) и основанная на нем спецификация Java2 Enterprise Edition (J2EE) послужили началом стандартизации серверов приложений. Класс *java bean* представляет собой класс Java, следующий стандарту Sun Java Beans Standard, предоставляющему инфраструктуру создания объектов для использования в графическом инструментарии. EJBs основан на растущей популярности Java в качестве языка программирования. Спецификации Sun, ведущего разработчика серверов и лидера в области интернет-технологий, стали общепризнанными. Эти спецификации включают также стандартизованный API для доступа к базам данных — одной из наиболее важных функций серверов приложений — а именно: соответствующий стандарту Java Database Connectivity (JDBC).

Вскоре серверы приложений на основе спецификации J2EE вырвались в лидеры рынка. Производители, которые придерживались других подходов, дополняли свои продукты поддержкой Java и в конечном итоге переходили на стратегию J2EE. Еще через некоторое время на рынке серверов приложений началось укрупнение. Компания Sun приобрела NetDynamics, разработчика одного из первых серверов приложений J2EE. BEA Systems, ведущий производитель программного обеспечения среднего уровня для обработки транзакций, приобрела WebLogic, еще одного пионера рынка серверов приложений. (В 2007 году BEA была приобретена компанией Oracle.) Netscape, традиционно работавшая в области веб-серверов, дополнила линию своих продуктов покупкой Kiva, еще одного лидера рынка в области серверов приложений.

Позже, когда AOL приобрела Netscape, а затем организовала совместное предприятие с Sun, оба эти J2EE-сервера приложений перешли под управление Sun, в конечном итоге превратившись в сервер приложений Sun iPlanet (который позже стал сервером приложений SunONE). Hewlett-Packard последовала общему примеру, приобретя еще одного разработчика серверов приложений Bluestone. IBM же оказалась белой вороной и создала собственный сервер приложений под названием WebSphere. Oracle также пошел по пути разработки собственного продукта — Oracle Application Server, хотя со временем многие программные продукты Oracle были заменены приобретенными компонентами сторонних производителей.

После нескольких лет агрессивной конкурентной борьбы, спецификация J2EE продолжала эволюционировать и включала расширенные возможности доступа серверов приложений к базам данных. WebLogic компании BEA и WebSphere компании IBM заняли ведущее место, практически поделив между собой рынок. Продукты от Sun, Oracle и десятка более мелких производителей поделили оставшуюся часть рынка. Каждый мало-мальски значительный серверный продукт удовлетворял требованиям спецификации J2EE и обеспечивал возможность обращения к базам данных на основе JDBC.

Доступ серверов приложений к базам данных

Концентрация рынка серверов приложений вокруг спецификации J2EE привела к эффективной стандартизации, по крайней мере на время, *внешнего* интерфейса между сервером приложений и СУБД на базе JDBC. Концептуально сервер приложений может автоматически обращаться к любой базе данных, предоставляющей JDBC-совместимый API, что тем самым делает его независимым от конкретной СУБД. На практике небольшие различия между СУБД в таких областях, как, например, диалект SQL или именование, требуют определенной настройки кода и тестирования. Однако эти отличия невелики, и подгонка под конкретную базу данных относительно проста.

Настройка управления данными со стороны прикладного кода, работающего на сервере приложений, оказывается более сложной задачей. В то время как сервер приложений обеспечивает единообразность управления данными, он делает это в нескольких различных архитектурах, использующих разные типы EJBs спецификации J2EE. Разработчик приложения должен делать выбор среди этих подходов, а в ряде случаев — и комбинировать их, чтобы добиться соответствия приложения поставленным требованиям. Вот некоторые из решений, которые должны быть приняты.

- Будет ли логика приложения обращаться к базе данных непосредственно из сессии или содержимое базы данных будет представлено как entity beans с инкапсулированной в них логикой? (Что такое system beans и entity beans, будет рассказано немного позже.)
- Если используется непосредственный доступ из сессии, может ли session bean быть без запоминания состояния (что упрощает его кодирование и управление им со стороны сервера приложений) или логика обращения к базе данных требует, чтобы bean был с запоминанием состояния, сохраняющий контекст между вызовами?
- Если для представления содержимого базы данных используются entity beans, может ли приложение для управления взаимодействием с базой данных полагаться на их хранение сервером приложений или логика приложения требует, чтобы entity bean обеспечивал собственную логику обращения к базе данных?
- Если для моделирования содержимого базы данных используются entity beans, должны ли они взаимнооднозначно соответствовать таблицам базы данных или лучше, если они будут представлять более высокоуровневые объектно-ориентированные представления данных, с данными внутри каждого bean, полученными из нескольких таблиц базы данных?

Компромиссы, представленные приведенными вопросами проектирования, хорошо демонстрируют проблемы соответствия SQL и технологий реляционных баз данных требованиям веб и архитектуры без запоминания состояния, а также требованиям серверов приложений и объектно-ориентированного программирования. В нескольких следующих разделах описываются основы EJBs и варианты решений проблем обращения в различных поддерживаемых архитектурах доступа к данным.

Типы EJB

В сервере приложений, совместимом с J2EE, пользовательский прикладной код Java, который реализует некоторую бизнес-логику, упакован и выполняется как набор EJBs. Объект EJB имеет хорошо определенное множество внешних интерфейсов (методов), которые он должен предоставлять, и написан с точным множеством открытых методов класса, которые определяют внешний интерфейс объекта. Вся работа выполняется в bean-объекте, и любые закрытые переменные, которые поддерживаются им для внутреннего пользования, могут быть инкапсулированы и скрыты от других объектов и разработчиков, которым не надо знать детали внутренней реализации; более того, код, который они пишут, не должен никоим образом зависеть от них.

Объекты EJBs работают на сервере приложений в *контейнере*, который предоставляет для beans как среду времени выполнения, так и службы. Последние простираются от общих служб, таких как управление памятью или диспетчеризация работы beans, до специфичных служб типа доступа к сети или к базе данных (через JDBC). Контейнер предоставляет также услуги по хранению состояния beans между их активациями. (*Сохраняемость* (persistence) — это свойство объектно-ориентированного программирования, которое обеспечивает сохранение данных для последующего использования. Обычным средством для этого являются базы данных.)

С точки зрения управления данными, имеются два основных типа объектов EJBs, графически проиллюстрированных на рис. 22.3.

- **Session beans** представляют сессии отдельного пользователя с сервером приложений. Концептуально имеется взаимнооднозначное соответствие между каждым session bean и текущим пользователем. На рисунке пользователи Мери, Джо и Сэм представлены каждый своим собственным session bean. Если внутри такого bean имеются внутренние переменные экземпляра, то эти переменные представляют текущее состояние, связанное с данным пользователем в течение этой конкретной сессии.
- **Entity beans** представляют бизнес-объекты и логически соответствуют индивидуальным строкам таблицы базы данных. Например, для entity beans, представляющих офисы, имеется взаимнооднозначное соответствие между каждым entity bean и конкретным офисом, который, кроме того, представлен в нашей учебной базе данных одной строкой таблицы `OFFICES`. Если в bean имеются внутренние переменные экземпляра, то их значения представляют значения столбцов в соответствующей строке таблицы `OFFICES`. Это состояние не зависит ни от какой конкретной сессии пользователя.

Beans любого типа могут обращаться к базе данных, но обычно делают это разными способами.

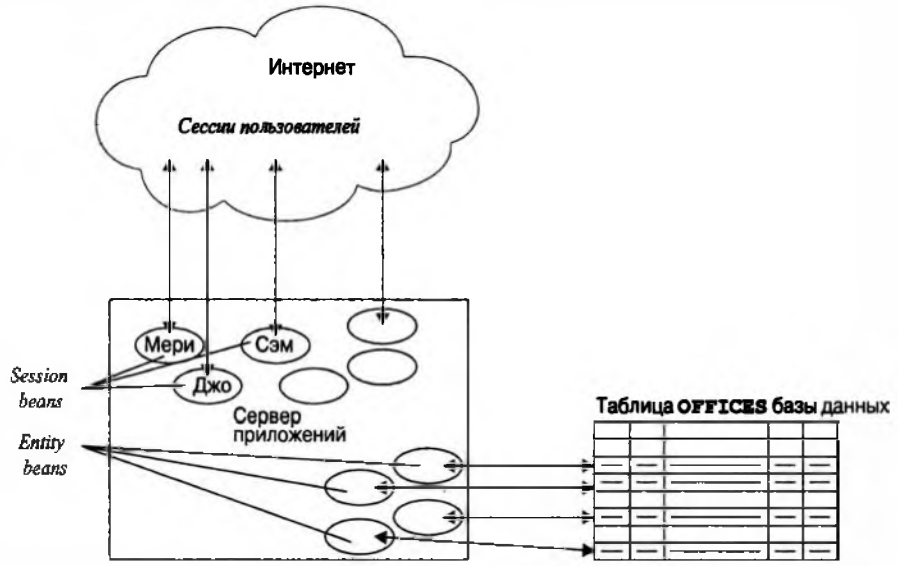

Рис. 22.3. Типы EJBs

Доступ к базе данных со стороны session bean

Обычно session bean обращается к базе данных при помощи последовательности из одного или нескольких вызовов JDBC от имени пользователя, которого представляет данный bean. Сервер приложений разделяет session beans на две категории, в зависимости от того, как они работают с состояниями.

- **Session bean без состояния**. Этот тип не хранит никакой информации о состоянии между вызовами методов. Свои действия он выполняет от имени одного пользователя и по одному запросу за раз. Каждый запрос, который выполняет bean, не зависит от предыдущих запросов. При таком ограничении каждый вызов bean должен передавать ему (в виде параметров) всю необходимую для выполнения запроса информацию.
- **Session bean с состоянием**. Этот тип поддерживает хранение информации о состоянии между вызовами. Bean должен "помнить" информацию из предыдущих вызовов (собственное состояние) для того, чтобы корректно выполнять последующие вызовы. Для хранения этой информации он использует закрытые переменные экземпляра.

В двух следующих разделах приведены примеры задач, которые проще всего решить при помощи описанных типов session bean. Вы указываете, должен ли session bean хранить состояние или нет, в специальном *дескрипторе применения* (deployment descriptor), который содержит информацию, передаваемую серверу приложений, на котором будет работать этот bean.

Сервер приложений на занятом веб-сайте может легко иметь больше session beans и прочих объектов EJBs, чем позволяет оперативная память. В таком случае сервер приложений поддерживает активными в оперативной памяти только огра-

ниченное количество экземпляров session bean. Если связанный с неактивной в настоящий момент сессией пользователь становится активен (т.е. следует обработать очередной его щелчок на странице веб-сайта), то сервер приложений выбирает другой экземпляр того же класса bean и *пассивизирует* его, т.е. сохраняет значения его переменных экземпляра, а затем использует данный bean для обслуживания активированной пользовательской сессии.

То, является ли session bean поддерживающим состояния или нет, существенно влияет на его активизацию и пассивизацию. Поскольку session bean без хранения состояния не требует сохранения информации между вызовами методов, сервер приложений не должен сохранять соответствующие переменные экземпляра при пассивизации и восстанавливать их при активизации. Однако в случае session bean с сохранением состояния сервер приложений должен копировать переменные экземпляра в постоянную память (в дисковый файл или базу данных) при пассивизации и восстанавливать их при активизации. Таким образом, session beans с сохранением состояния приводят к снижению производительности и пропускной способности сервера приложений. Beans без сохранения состояния предпочтительны в смысле производительности, но многие приложения сложно или даже невозможно реализовать без использования beans с сохранением состояния.

Использование JDBC из session bean без сохранения состояний

На рис. 22.4 показан простой пример приложения, которое легко реализовать при помощи обращений к базе данных session beans без сохранения состояния. Страница не веб-сайте выводит текущую цену товаров компании. Эта страница не может быть статической, поскольку цена может вырасти каждую минуту. Так что когда браузер пользователя запрашивает эту страницу, веб-сервер передает запрос серверу приложений, который вызывает метод session bean. Этот session bean может использовать JDBC для того, чтобы послать SQL-инструкцию SELECT базе данных и получить текущие цены на товары. Session bean переформатирует полученный результат и отправляет его веб-серверу в виде фрагмента веб-страницы, которую тот вернет пользователю.

Session beans без сохранения состояния могут выполнять и более сложные функции. Предположим, что у той же компании на том же веб-сайте имеется страница, на которой пользователь может заказать каталог продукции, заполнив небольшую форму. Когда форма заполнена и пользователь щелкает на кнопке ее отправки, браузер посылает данные из формы на веб-сервер, который передает запрос серверу приложений. В этот раз вызывается другой метод session bean, который получает в качестве параметров информацию из заполненной формы. Session bean может использовать JDBC для пересылки SQL-запроса INSERT в таблицу базы данных, хранящую заказы каталогов, полученные через веб-сайт.

В каждом из приведенных примеров вся необходимая для работы session bean информация передается методу в качестве параметров. Когда работа выполнена, хранить эту информацию больше не требуется. При очередном вызове вновь будет передана вся необходимая информация, так что хранить какие-то данные между вызовами не требуется. Что еще более важно, действия базы данных при каждом вызове никак не зависят от всех других вызовов. Ни одна транзакция базы данных не требует нескольких вызовов метода.

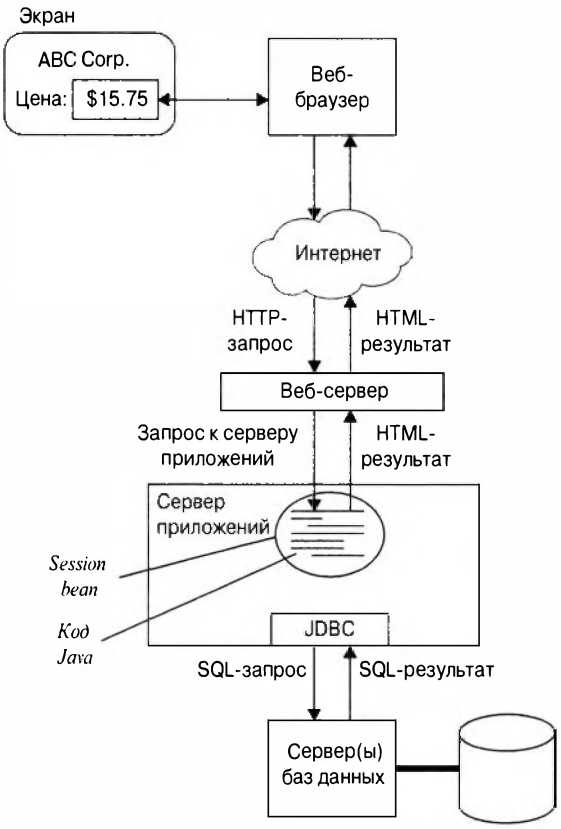

Рис. 22.4. Вызовы базы данных из session bean без сохранения состояний

Использование JDBC из session bean с сохранением состояния

Для многих задач ограничения, накладываемые session beans без сохранения состояний, оказываются чрезмерными. Рассмотрим более сложную форму, охватывающую четыре страницы. Когда пользователь заполняет каждую страницу и отправляет ее на сайт, session bean должен накапливать информацию и хранить ее до тех пор, пока не будут заполнены все четыре страницы и все данные не будут готовы для передачи в базу данных. Необходимость хранения информации между вызовами метода требует применения session bean с сохранением состояний.

Еще один пример, в котором требуется session bean с сохранением состояний, представляет собой коммерческий веб-сайт, где пользователь может выполнять покупки, складывая покупаемые товары в виртуальную "корзину", которая затем будет оплачена. После 40–50 щелчков на страницах сайта пользователь соберет в корзине пять-шесть товаров. Если после этого пользователь щелкнет на кнопке показа содержимого корзины, то проще всего будет удовлетворить его любопытство, если хранить это содержимое как состояние session bean.

В обоих примерах session bean должен поддерживать непрерывную связь с базой данных для эффективного выполнения стоящей перед ним задачи. На рис. 22.5 по-

казана соответствующая схема работы, отличающаяся от приведенной на рис. 22.4. Даже если bean можно реализовать без переменных экземпляра (например, путем хранения всей информации о состоянии в базе данных), для поддержания доступа к базе данных требуется одна непрерывная сессия. Такая сессия поддерживается посредством API со стороны клиента СУБД, а сам API должен хранить информацию о состоянии сессии между вызовами метода bean.

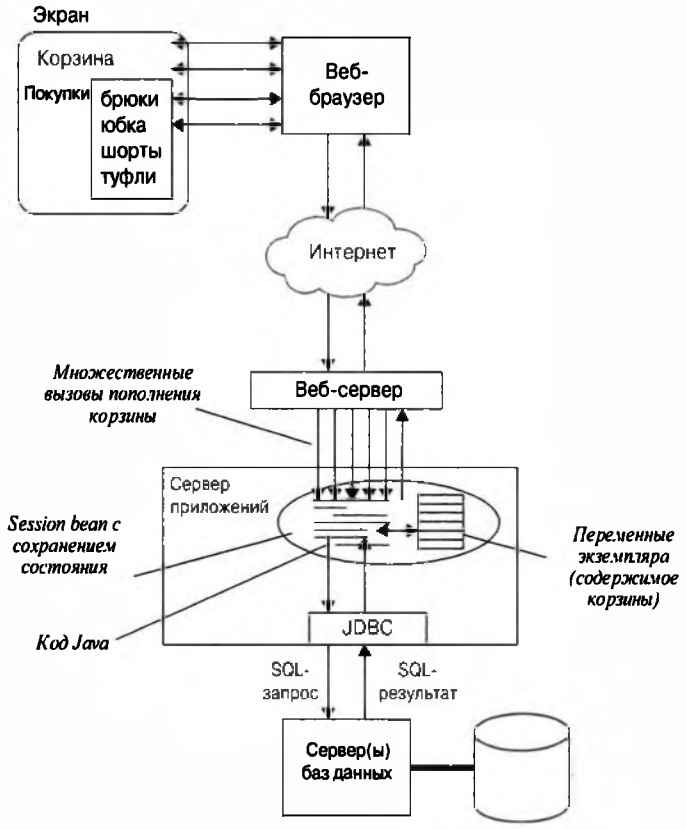

Рис. 22.5. Вызовы базы данных из session bean с сохранением состояний

Доступ к базе данных со стороны entity bean

Можно реализовать законченное, интеллектуальное приложение для веб-сайта с использованием session beans на сервере приложений на основе J2EE. Однако программирование приложений с применением session beans обычно приводит к коду в большей степени процедурному, чем объектно-ориентированному. Объектно-ориентированная философия состоит в наличии классов объектов (в данном случае, классов EJB), представляющих сущности реального мира, такие как покупатели или офисы, а также экземпляров объектов, представляющих отдельных покупателей или офисы. Но session beans не представляют никакие из указанных сущностей; они призваны представлять текущие активные пользовательские сес-

сии. Когда взаимодействие с базой данных осуществляется непосредственно при помощи session beans, представление сущностей реального мира остается в основном на долю базы данных; соответствующие объекты при этом отсутствуют.

Entity beans предоставляют объекты для сущностей реального мира и строк представляющей их таблицы реляционной базы данных. Классы entity bean олицетворяют покупателей и офисы; отдельные экземпляры представляют конкретных покупателей и офисы. Другие объекты (такие, как session beans) на сервере приложений могут взаимодействовать с покупателями и офисами с использованием объектно-ориентированных технологий, вызывая методы представляющих их entity beans.

Для поддержки этой объектно-ориентированной модели требуется очень тесное сотрудничество между представлениями сущностей при помощи entity bean и их представлениями в базе данных. Если session bean вызывает метод entity bean заказчика, который изменяет лимит кредита последнего, то это изменение должно быть отражено в базе данных, с тем чтобы приложение обработки заказов, использующее эту базу данных, работало с новым значением лимита кредита. Аналогично, если приложение управления складом добавляет некоторое количество товара в базу данных, соответствующий entity bean на сервере приложений также должен быть обновлен.

Так же как при необходимости сервер приложений пассивизирует и активизирует session beans, в случае большой загрузки пассивизируются и активизируются и entity beans. Перед тем как сервер приложений пассивизирует entity bean, состояние последнего должно быть сохранено в базе данных. Когда же сервер приложений реактивизирует entity bean, его переменные экземпляра должны получить те же значения, что были перед пассивизацией, для чего их следует загрузить из базы данных. Класс entity bean определяет методы обратного вызова, которые entity bean должен предоставить для реализации описанной синхронизации.

Имеется точное соответствие между действиями entity beans и действиями базы данных, показанное в табл. 22.1. Спецификация J2EE предоставляет два различных способа управления этой координацией.

- **Хранение на уровне beans**. Каждый entity bean сам ответственен за синхронизацию с базой данных. Прикладной программист, разрабатывающий entity bean и его методы, должен при необходимости использовать JDBC для чтения и записи информации в базу данных. Контейнер сервера приложений уведомляет bean при выполнении действий, требующих взаимодействия с базой данных.
- **Хранение на уровне контейнера**. За синхронизацию с базой данных отвечает контейнер EJB на сервере приложений. Контейнер отслеживает взаимодействия с entity bean и автоматически использует JDBC для чтения и записи информации в базу данных и обновления (при необходимости) переменных экземпляра bean. Прикладной программист, разрабатывающий entity bean и его методы, должен сосредоточиться на бизнес-логике bean и считать, что его переменные экземпляра точно представляют состояние информации в базе данных.

Таблица 22.1. Соответствие действий базы данных и EJB

Инструкция базы данных	Метод EJB	Действие EJB/базы данных
`INSERT`	`ejbCreate()`, `ejbPostCreate()`	Создает новый экземпляр entity bean; начальное состояние определяется параметрами вызова `create()`. В базу данных должна быть внесена новая строка с этими значениями
`SELECT`	`ejbLoad()`	Загружает значения переменных экземпляра, считывая их из базы данных
`UPDATE`	`ejbStore()`	Сохраняет значения переменных экземпляра в базе данных
`DELETE`	`ejbRemove()`	Удаляет экземпляр entity bean; соответствующая строка в базе данных также должна быть удалена

Обратите внимание на то, что entity beans могут быть только с сохранением состояния. Различия между рассматриваемыми типами beans по сути являются не отличиями между beans с сохранением состояния и без него, а отличиями в том, *кто* отвечает за поддержание корректности состояния. В следующих двух разделах будут рассмотрены практические вопросы, связанные с каждым типом entity beans и принятием компромиссных решений об их использовании.

Хранение на уровне контейнера

Дескриптор применения entity bean указывает, что хранение состояния должно выполняться на уровне контейнера. Дескриптор также определяет отображение переменных экземпляра bean на столбцы в базе данных. Кроме того, дескриптор определяет первичный ключ, уникальным образом идентифицирующий bean и соответствующую строку базы данных. Значение первичного ключа используется в операциях, которые сохраняют и получают значения переменных из базы данных.

В случае хранения на уровне контейнера за поддержание синхронизации между entity bean и строкой базы данных отвечает EJB-контейнер. Он использует вызовы JDBC для сохранения значений переменных экземпляра в базе данных, для восстановления этих значений, для вставки новой строки в базу данных и удаления имеющейся. Перед тем как сохранять значения в базе данных, контейнер вызывает метод обратного вызова bean `ejbStore()` для уведомления bean о том, что он должен привести свои внутренние переменные в согласованное состояние. Аналогично после загрузки значений из базы данных контейнер вызывает метод обратного вызова bean `ejbLoad()`, позволяя выполнить необходимые действия (например, вычислить значения, которые не сохраняются, а вычисляются на основе значений других переменных). Точно так же вызываются еще несколько методов обратного вызова — `ejbRemove()` перед тем, как контейнер удалит строку из базы данных, и `ejbCreate()` и `ejbPostCreate()` при вставке новой строки. У многих entity beans эти методы обратного вызова пустые, поскольку все операции с базой данных выполняет контейнер.

Хранение на уровне bean

Если дескриптор применения entity bean указывает, что хранение выполняется на уровне bean, то контейнер полагает, что каждый entity bean самостоятельно взаимодействует с базой данных. Когда создается новый entity bean, контейнер вызывает его методы `ejbCreate()` и `ejbPostCreate()`. Bean отвечает за выполнение соответствующей инструкции базы данных `INSERT`. Аналогично при удалении entity bean контейнер вызывает его метод `ejbRemove()`. Bean сам отвечает за выполнение соответствующей инструкции базы данных `DELETE`, так что после возврата из метода `ejbRemove()` контейнер может свободно удалять bean и повторно использовать выделенную ему память.

Загрузка bean точно так же осуществляется путем вызова контейнером метода `ejbLoad()`, а сохранения — путем вызова `ejbStore()`. Кроме того, контейнер уведомляет bean о пассивизации и активизации. Конечно, ничто не ограничивает взаимодействие entity bean с базой данных только этими методами. Если bean требуется доступ к базе данных в процессе выполнения некоторого из его методов, bean может выполнять любые необходимые для этого вызовы JDBC. Вызовы JDBC в методах обратного доступа предназначаются только для решения задачи сохранения состояния.

Выбор уровня хранения

Вы, конечно, можете спросить — зачем вообще нужно хранение на уровне bean, если хранение на уровне контейнера полностью устраняет необходимость заботиться о синхронизации с базой данных? Ответ заключается в наличии ряда ограничений хранения на уровне контейнера.

- **Несколько баз данных**. У большинства серверов приложений entity beans должны отображаться на один сервер баз данных. Если данные entity bean поступают от нескольких баз данных, то единственным способом синхронизации является хранение на уровне bean.
- **Несколько таблиц**. Хранение на уровне контейнера хорошо работает тогда, когда все переменные экземпляра entity bean хранятся в одной строке одной таблицы — когда имеется взаимно однозначное соответствие между экземплярами bean и строками таблицы. Если entity bean требуется моделировать более сложный объект, такой как заголовок заказа и отдельные его строки, информация о которых хранится в разных связанных таблицах, обычно требуется хранение на уровне bean, так как только код bean в состоянии обеспечить интеллектуальную связь объекта с информацией в базе данных.
- **Оптимизация производительности**. В случае хранения на уровне контейнера последний должен исходить из принципа "все или ничего" для выбора хранимых переменных экземпляра. Каждый раз при сохранении или загрузке следует обрабатывать все переменные. Во многих приложениях, однако, entity bean может, в зависимости от текущего конкретного состояния, определить, какие именно переменные следует сохранить, — и это могут быть далеко не все переменные экземпляра. Если entity bean

содержит большое количество информации, то разница в производительности может оказаться весьма значительной.

- **Оптимизация базы данных.** Если методы entity bean, реализующие его бизнес-логику, требуют большого количества обращений к базе данных (запросов и обновлений), то некоторые из операций, которые при схеме хранения на уровне контейнера выполняет последний, могут оказаться избыточными. При хранении на уровне bean последний может оказаться в состоянии определить, какие именно операции с базой данных совершенно необходимы для синхронизации и когда база данных находится в актуальном состоянии.

На практике эти ограничения часто препятствуют применению хранения на уровне контейнера. Поэтому недаром новейшие усовершенствования спецификации EJB направлены именно на их преодоление. Тем не менее хранение на уровне bean остается очень важной технологией в современных серверах приложений.

Усовершенствования EJB 2.0

Опубликованная в апреле 2001 года спецификация EJB 2.0 представляет собой существенно переработанную версию EJB. Многие усовершенствования в EJB 2.0 оказались несовместимы с соответствующими возможностями EJB 1.x. Чтобы избежать неработоспособности совместимых с EJB 1.x beans, EJB 2.0 предоставляет дополнительные возможности в соответствующих областях, которые обеспечивают параллельное существование beans версий EJB 1.x и EJB 2.0. Полное описание отличий EJB 1.x от EJB 2.0 выходит за рамки данной книги. Однако некоторые из отличий объясняются сложностями применения хранения на уровне контейнера в спецификации EJB 1.x, и эти изменения непосредственно влияют на работу с базой данных в EJB.

Одна из трудностей при работе с EJB 1.x уже упоминалась ранее — это трудность моделирования сложных объектов, которые получают свои данные из нескольких таблиц базы данных или содержат нереляционные структуры типа массивов и иерархических данных. В EJB 1.x можно моделировать сложный объект как семейство взаимосвязанных entity beans, каждый из которых связан с одной таблицей. Такой подход позволяет использовать хранение на уровне контейнера, но связи между частями комплексного объекта при этом должны реализовываться в коде приложения. В идеале внутренние детали таких сложных объектов должны быть скрыты от прикладного кода. В качестве альтернативы можно моделировать сложный объект как единый entity bean, данные внутренних переменных которого получаются из нескольких связанных таблиц. Так достигается желаемая прозрачность прикладного кода, но хранение на уровне контейнера при получении данных из нескольких таблиц при этом затруднено.

EJB 2.0 решает этот вопрос при помощи абстрактных *методов доступа* (accessor methods), которые используются для установки и получения значения каждой хранимой переменной экземпляра в entity bean. Контейнер поддерживает память для переменных и их значений. Bean явно вызывает метод доступа `get()` для получения значения переменной экземпляра и метод доступа `set()` для его уста-

новки. Аналогично имеются абстрактные методы доступа `get()` и `set()` для каждого *отношения*, связывающего строки базы данных, которые поставляют информацию для entity bean. Отношения "многие-ко-многим" легко обрабатываются путем отображения их на переменные коллекций Java.

Эти новые возможности обеспечивают контейнер полной информацией о всех переменных экземпляра, используемых bean. Последний может представлять сложный объект, который получает данные из нескольких таблиц базы данных, скрывая детали от кода приложения. Но теперь хранение на уровне контейнера становится возможным благодаря тому, что контейнер "знает" все о разных частях объекта и об отношениях между частями.

Еще одна проблема спецификации EJB 1.x заключается в том, что хотя взаимодействия с базой данных и стандартизованы, *методы поиска*, которые используются для выявления активных entity beans, таковыми не являются. Методы поиска реализуют такие возможности, как поиск определенного entity bean по первичному ключу или поиск множества beans, отвечающих определенному критерию. Без такой стандартизации невозможно обеспечить переносимость между разными серверами приложений.

EJB 2.0 преодолевает ограничения, связанные с поиском, при помощи абстрактных *методов отбора* (select methods), которые выполняют поиск entity beans. Эти методы используют введенный в EJB 2.0 язык запросов (Query Language, EJBQL). Хотя этот язык и основан на SQL, он включает такие нереляционные конструкции, как, например, выражения путей.

Наконец, EJB 2.0 спроектирован для работы со стандартом SQL и его абстрактными типами данных. Поддержка этих типов несколько упрощает взаимодействие между entity beans и базой данных, поддерживающей абстрактные типы (в настоящее время их поддерживают только некоторые СУБД).

Усовершенствования EJB 3.0

Спецификация EJB 3.0, черновой вариант которой был опубликован в 2004 году, а окончательный — в 2006 году, заставляет контейнер выполнять большее количество работы, тем самым упрощая программирование приложений. Она снижает количество кода, который должен создать разработчик, в том числе устраняет необходимость в методах обратного вызова и упрощает программирование entity bean.

EJB 3.0 упрощает интерфейс прикладного программирования разными путями.

- В качестве альтернативы сложным дескрипторам применения можно использовать аннотации с метаданными. Последние могут использоваться для определения типов beans, настроек транзакций и безопасности, отображения объектов и т.п.
- Вместо среды EJB и ссылок на ресурсы можно применять Dependency инжекции зависимостей.
- Разработчики beans могут объявлять любой метод как метод обратного вызова, что делает излишним использование специальных методов обратного вызова.

- Методы перехвата могут быть определены в session beans (с сохранением состояния или без такового) или в beans, управляемых сообщениями. *Метод перехвата* представляет собой метод, который перехватывает вызов бизнес-метода. Вместо определения метода перехвата, в классе bean может использоваться класс-перехватчик.
- Клиенты могут непосредственно вызывать метод EJB без создания экземпляра bean.

Session beans также усовершенствованы и упрощены в нескольких направлениях.

- Session beans EJB 3.0 стали проще, так как они представляют собой чистые классы Java, которые не должны реализовывать интерфейсы session bean. Session bean может иметь удаленный, локальный или и локальный, и удаленный интерфейсы.
- Для определения bean или интерфейса и свойств времени выполнения session beans могут использоваться аннотации с метаданными.
- Для session beans (с сохранением состояния или без такового) поддерживаются методы-слушатели обратного вызова.
- Для инжекции объектов среды, ресурсов или контекста EJB разработчики могут использовать метаданные или дескрипторы применения.
- Методы или классы перехвата поддерживаются как для session beans с сохранением состояния, так и без него.

Управляемые сообщениями beans усовершенствованы и упрощены в нескольких направлениях.

- Они могут реализовывать интерфейс `MessageListener` вместо интерфейса `MessageDriven`.
- Метаданные упрощают спецификацию bean или интерфейса и свойств времени выполнения.
- Поддерживаются методы-слушатели обратного вызова.
- Вместо дескрипторов применения можно использовать зависимости.
- Могут использоваться методы или классы перехвата.

Entity beans и API хранения могут быть усовершенствованы и упрощены несколькими способами. Во-первых, EJB 3.0 стандартизирует модель хранения POJO (Plain Old Java Object, старые простые объекты Java) и упрощает entity beans, которые становятся конкретными классами Java, не требующими никаких интерфейсов. Классы entity bean непосредственно поддерживают полиморфизм и наследование. Во-вторых, EJB 3.0 включает новый EntityManager API для создания, поиска, удаления и обновления объектов. В-третьих, вместо дескрипторов применения можно использовать аннотации, что существенно упрощает разработку объектов. В-четвертых, существенно усилены возможности языка запросов. В-пятых, поддержка методов-слушателей обратного вызова обеспечивается применением методов, указанных в аннотациях либо дескрипторах применения. Наконец, процессор хранения в EJB 3.0 может использоваться и вне контейнера.

Разработка приложений с открытым кодом

Развитие Интернета привело к появлению множества новинок в области серверов приложений, распространения приложений, а также их кооперации с SQL-базами данных. В этих областях активно работают и разработчики программного обеспечения с открытым кодом. Например, наиболее широко используемым веб-сервером с открытым кодом является Apache. Для создания веб-приложений разработчики часто используют комбинацию асинхронных сценариев JavaScript и XML, известную как Ajax (это слово можно рассматривать и как аббревиатуру AJAX — Asynchronous JavaScript and XML, асинхронный JavaScript и XML). Вместо JavaScript могут использоваться и другие языки сценариев, такие как PHP; применение XML не является совершенно необходимым с технической точки зрения, так что аббревиатура AJAX менее предпочтительна, чем просто имя Ajax[1]. Хотя сама аббревиатура появилась только в 2005 году, асинхронная загрузка веб-содержимого уходит корнями в средину 1990-х годов.

Веб-приложения, использующие Ajax, могут получать данные асинхронно, при помощи фонового процесса, не влияющего на вид и поведение существующей веб-страницы. Данные получаются при помощи объекта `XMLHttpRequest` или (в браузерах, не поддерживающих объекты) с использованием удаленных сценариев. Базы данных, используемые вместе с Ajax для сохранения и загрузки долговременно хранимых данных, практически всегда представляют собой реляционные продукты с открытым кодом, такие как MySQL.

Еще одной популярной платформой для распространения информации и приложений посредством веб-страниц является LAMP (Linux, Apache, MySQL и PHP/Python/Perl). Фактически, большая часть явления, известного как "Web 2.0", построена на платформе LAMP, которая обеспечивает дешевую инфраструктуру на базе SQL для веб-приложений. В настоящее время ведущую роль в Интернете играет именно LAMP и ее вариации — например, с возможными новыми компонентами (в качестве примера можно привести LAMR — LAMP с заменой языка сценариев на Ruby on Rails (ROR)). С ростом популярности служб на базе Интернета, таких как Infrastructure as a Service (IaaS), Platform-as-a-Service (PaaS) и Software as a Service (SaaS), технологии серверов приложений LAMP продолжают развиваться и эволюционировать. Все эти службы в своей массе опираются на LAMP и ее непосредственных преемников.

Серверы приложений и кеширование

Узким местом веб-сайтов большого объема, ограничивающим их производительность, может стать обращение к базам данных. Структура EJB по мере усложнения бизнес-логики приводит к росту обращений к базе данных для поддержки синхронизации между entity bean и базой данных. Если веб-сайт реализует высокую степень персонализации (т.е. большая часть страниц генерируется динамически на основе информации о конкретном пользователе, запросившем их), то ко-

---

[1] *Имеется в виду Аякс — греческий герой времен Троянской войны.* — Примеч. пер.

личество обращений к базе данных растет с еще большей скоростью. В предельном случае каждый щелчок пользователя на странице такого сайта может требовать обращения к базе данных за информацией о пользователе. Наконец, работа пользователей осуществляется в режиме реального времени, так что главную роль играет пиковая, а не средняя нагрузка на сайт. Среднее время обработки каждого щелчка пользователя менее важно, чем время обработки при пиковой нагрузке — время, из-за которого раздраженные пользователи будут называть сайт "тормознутым" или как-то похоже (и не менее неприятно).

В веб уже имеется эффективное решение проблемы с пиковой нагрузкой — кеширование веб-страниц и горизонтальное масштабирование. При кешировании копии страниц с большим количеством обращений рассылаются по сети и дублируются. В результате общая пропускная способность для данных веб-страниц повышается, а сетевой траффик, связанный с ними, снижается. При горизонтальном масштабировании содержимое веб-сайта воспроизводится на двух или большем количестве серверов (до десятков, а в тяжелых случаях — и до сотен серверов), полная мощность которых в плане обработки страниц оказывается существенно выше, чем у одного сервера.

Аналогичное кеширование и горизонтальное масштабирование используются и для повышения пропускной способности серверов приложений. Большинство нынешних коммерческих серверов приложений реализует *кеширование beans*, при котором в памяти сервера приложений содержатся копии наиболее часто используемых entity beans. Кроме того, серверы приложений часто используются в банках или кластерах, когда каждый сервер приложений обеспечивает одинаковую бизнес-логику и одинаковые возможности обработки приложений. Многие коммерческие серверы приложений используют горизонтальное масштабирование в пределах одного сервера, что обеспечивает эффективное использование симметричной многопроцессорности (SMP). Так, для восьмипроцессорного сервера совершенно типична ситуация, когда параллельно работают восемь независимых копий серверного программного обеспечения. На рис. 22.6 показана конфигурация типичного сервера приложений, состоящего их трех четырехпроцессорных серверов.

К сожалению, горизонтальное масштабирование и кеширование противоречат друг другу при работе с данными с сохранением состояния, такими как хранящиеся в entity bean или базе данных. Без специальной логики синхронизации кеша, обновления bean в кеше одного сервера не будут автоматически появляться во всех кешах, что потенциально может привести к некорректным и несогласованным результатам. Рассмотрим, например, что произойдет с данными о доступности товара на складе, если три или четыре различных кеша содержат копии entity bean для одного и того же товара и бизнес-логика сервера приложений обновляет эти значения. Кеши очень быстро станут содержать различные доступные количества товара, причем ни одно из них не будет точным. К сожалению, логика синхронизации кешей, необходимая для обнаружения и предупреждения такого рода ситуаций, сопровождается большими накладными расходами. Абсолютная синхронизация требует применения двухфазного протокола принятия изменений (который будет рассматриваться в главе 23, "Сети и распределенные базы данных").