Объектно-реляционное преобразование в технологии EJB

Entity beans (компоненты сущности)

EB – это EJB-компоненты специального вида, которые предназначены для организации компонентного интерфейса доступа к данным из того или иного источника. При этом имеют место следующие особенности:

1. Природа источника данных может быть различна, в реальных приложениях, как правило роль источника данных выполняется реляционными БД, в силу того, что реляционные СУБД занимают доминирующее положение на рынке. Тем временем может быть использовано и хранилище другой природы, например файловая система, объектно-ориентированная или древовидная, объектная БД, также XML-хранилище. Ни спецификация EJB, ни J2EE не ограничивают ни разработчика приложения, ни разработчика контейнера в выборе источника данных.
2. EB представляют работу с данными как с системой объектов, при этом состояние объектов, моделируемое их полями, отражает состояние данных в соответствующем источнике. Вызывая методы объектов, клиент изменяет их состояние. Изменение состояния объектов синхронизируется с источником данных автоматически под управлением контейнера.
3. Многие системы хранения информации, которые могут выступать как источники данных, позволяют определить отношение между различными элементами данных, а также задавать правила, определяющие целостность этих отношений. Эти правила определяются предметной областью, к которой относятся эти данные, а система контролирует соблюдение этих правил при выполнении операций с данными. Соответственно технология EB позволяет на компонентном уровне моделировать отношения, существующие между элементами данных, при этом контейнер обеспечивает соблюдение правил целостности.
4. При работе с тем или иным источником данных используется соответствующий язык запросов, который специфичен для источников различных типов. Поскольку EB должны отображаться потенциально в источнике различных типов, потребовалось создать язык запросов, работающих на компонентном уровне и независимых от типа источника данных. Существует язык запросов EJB QL, который оперирует компонентами EB подобно тому, как язык SQL оперирует таблицами.
5. Общие принципы разработки и использования EB похожи на те, что имеют место для сессионных компонентов, т.е. клиент работает с компонентом через представление, а логика компонента реализуется классом компонента. При этом ЖЦ, назначение и детальная реализация EB существенно отличаются от SB.

Entities (сущности) в EJB 3.0

Спецификация Java Persistence API определяет набор прикладных интерфейсов языка Java для управления персистентностью и объектно-реляционного преобразования на платформах Java SE и Java EE.

Сущность – это легкий (lightweight) устойчивый объект предметной области. Основным программным артефактом является класс сущности, который может использовать другие вспомогательные классы, в том числе для представления состояния сущности. При этом имеют место следующие особенности:

1. В качестве источника данных может выступать только реляционная база данных
2. Сущности представляют работу с данными как с системой объектов, при этом состояние объектов, моделируемое их полями, отражает состояние данных в соответствующем источнике. Вызывая методы объектов, клиент изменяет их

состояние. Изменение состояния объектов синхронизируется с источником данных автоматически под управлением контейнера или по требованию приложения.
3. Сущности позволяют моделировать отношения, существующие между элементами данных, при этом контейнер не обеспечивает соблюдение правил целостности.
4. Существует язык запросов (JP QL), который оперирует сущностями подобно тому, как язык SQL оперирует таблицами. Данный язык обладает большими возможностями, чем EJB QL.
5. Общие принципы разработки и использования сущностей значительно отличаются от тех, что имеют место для сессионных компонентов. Клиент работает с экземпляром сущности напрямую, логика реализуется классом сущности, для управления жизненным циклом сущности используется не домашний интерфейс, а менеджер сущностей.

Сущности

Требования к классу сущности.

1. Класс сущности должен быть отмечен аннотацией `Entity` или указан как сущность в XML-дескрипторе.
2. У класса сущности должен быть открытый или защищенный конструктор без параметров. У класса сущности могут быть и другие конструкторы.
3. Класс сущности должен быть классом верхнего уровня (top-level class). Не следует указывать перечислимый тип или интерфейс в качестве сущности.
4. Класс сущности, его методы и постоянные поля не должны быть `final`.
5. Если экземпляр сущности должен передаваться по значению как отсоединенный объект (detached object), например, через удаленный интерфейс, класс сущности должен реализовывать интерфейс `Serializable`.
6. Постоянное состояние экземпляра сущности составляют значения полей экземпляра класса сущности, которые могут соответствовать свойствам JavaBeans. Доступ к полям сущности напрямую должен вестись исключительно из методов этой сущности. Состояние сущности доступно для клиентов только через методы доступа к полям (get-/set-методы) или другие бизнес-методы. Поля класса сущности не должны быть открытыми.

Постоянные поля и свойства

Провайдер персистентности осуществляет доступ к постоянному состоянию сущности одним из двух способов: через свойства (с помощью get-/set-методов в стиле JavaBeans) или через поля. Выбор способа осуществляется указанием аннотаций объектно-реляционного преобразования (ОРП) для get-методов или для полей, соответственно.
Если используется доступ через свойства, то все свойства, не помеченные аннотацией `Transient`, являются постоянными. Get-методы для свойств сущности должны быть открытыми (`public`) или защищенными (`protected`).
Если используется доступ через поля, то все поля, не имеющие модификатор `transient` и не помеченные аннотацией `Transient`, являются постоянными.
Не рекомендуется использовать в одном классе сущности оба указанных способа, так как в этом случае поведение провайдера персистентности не определено.
В качестве типов полей и свойств, значение которых представляет собой коллекцию объектов, разрешено использовать только следующие интерфейсы: `java.util.Collection`, `java.util.Set`, `java.util.List`, `java.util.Map`, а также их

параметризованные варианты. Конкретный тип коллекции можно использовать только при инициализации поля или свойства перед сохранением сущности.
Get-/set-методы могут содержать дополнительную бизнес-логику, например, для проверки допустимости значения свойства. Провайдер персистентности выполняет эту логику, если используется доступ через свойства. Однако порядок вызова этих методов при загрузке или сохранении состояния сущности не определен, что следует учитывать при разработке данной бизнес-логики.
Если get-/set-метод выбрасывает исключение, то текущая транзакция откатывается. Если get-/set-метод был вызван при загрузке или сохранении состояния сущности, то выброшенное им исключение заворачивается в исключение `PersistenceException`.
Допускаются следующие типы постоянных полей и свойств:

1) примитивные типы;
2) `java.lang.String`;
3) сериализуемые типы (в том числе оболочки примитивных типов, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, и любые сериализуемые типы, определенные пользователем);
4) массивы `byte[]`, `Byte[]`, `char[]` и `Character[]`;
5) перечислимые типы;
6) типы сущностей и коллекции типов сущностей;
7) вложимые классы (см. далее).

Создание экземпляра сущности

Экземпляры сущностей создаются с помощью оператора `new`. Непосредственно после создания экземпляр сущности не является постоянным. Экземпляр становится постоянным с помощью менеджера сущностей.

Первичные ключи и идентичность сущности

У каждой сущности должен быть первичный ключ.
Первичный ключ должен быть определен ровно один раз в иерархии классов сущностей – в классе сущности, являющемся корнем иерархии сущностей, или в отображенном родительском классе иерархии сущностей.
Простому (несоставному) первичному ключу должно соответствовать единственное постоянное поле или свойство класса сущности, которое отмечается аннотацией `Id`.
Составному первичному ключу может соответствовать одно или несколько постоянных полей (свойств) класса сущности. Для представления составного первичного ключа должен быть определен класс первичного ключа. Для определения составных первичных ключей используются аннотации `EmbeddedId` и `IdClass`.
Допускаются следующие типы простых первичных ключей и полей (свойств) составных первичных ключей:

1) примитивный тип;
2) класс-оболочка примитивного типа;
3) `java.lang.String`;
4) `java.util.Date`, `java.sql.Date`.

Не следует использовать для первичных ключей приблизительные числовые типы (например, `float`). Если поле (свойство) первичного ключа имеет тип `java.util.Date`, то для него следует указать временной тип `DATE`, то есть в качестве части первичного ключа нельзя использовать временную отметку.

Способ доступа к классу первичного ключа должен совпадать со способом доступа к классу соответствующей сущности.
Для составных первичных ключей действуют следующие правила:
- Класс первичного ключа должен быть открытым и должен иметь открытый конструктор без параметров.
- Класс первичного ключа должен быть сериализуемым.
- В классе первичного ключа должны быть определены методы `equals()` и `hashCode()`.
- Составной первичный ключ должен быть представлен как вложимый класс (в этом случае используется аннотация `EmbeddedId`) или должен быть отображен на множество полей или свойств класса сущности (с помощью аннотации `IdClass`). В последнем случае каждому полю (свойству) класса первичного ключа по имени и типу должно соответствовать поле (свойство) класса сущности.

Приложение не должно изменять значение первичного ключа после сохранения сущности, в противном случае его поведение не определено.
Первичный ключ допускается генерировать автоматически. В этом случае допускается использовать только целочисленные типы. Способ генерации указывается с помощью элемента `strategy` аннотации `GeneratedValue`. Возможные способы генерации определены в перечислении `GenerationType`:
1. `AUTO` – провайдер персистентности самостоятельно выбирает наиболее подходящий способ генерации для конкретной базы данных (данный способ действует по умолчанию);
2. `SEQUENCE` – уникальные значения берутся из последовательности значений (особого объекта базы данных), имя которой указывается в элементе `generator` аннотации `GeneratedValue`;
3. `IDENTITY` – уникальное значение формируется СУБД в столбце особого типа (т.н. identity column) при вставке строки в таблицу, имя столбца указывается в элементе `generator` аннотации `GeneratedValue`;
4. `TABLE` – провайдер персистентности при создании схемы базы данных создает также дополнительную таблицу, содержащую пары значений «имя сущности – последнее значение первичного ключа». Значение из этой таблицы используется для генерации первичного ключа экземпляра сущности, после чего в таблицу записывается новое значение.

Вложимые классы

Для представления состояния сущности можно использовать другие классы, называемые вложимыми. Экземпляры вложимых классов не имеют собственной идентичности и существуют только как вложенные объекты той сущности, к которой они принадлежат. В силу этого они, как правило, отображаются на таблицы вместе с владеющей ими сущностью.
Вложимый класс должен отвечать требованиям к классу сущности, но вместо аннотации `Entity` должен иметь аннотацию `Embeddable`. Способ доступа к вложенному объекту совпадает со способом доступа к классу владеющей им сущности.
В настоящее время поддерживается только один уровень вложенности.

Модуль персистентности

Модуль персистентности (persistence unit) представляет собой средство упаковки и установки классов сущностей и состоит из следующих элементов:
- Конфигурационная информация для СОРП (в т.ч. источник данных, тип СОРП, способ управления транзакциями), которая определяет поведение фабрики менеджеров сущностей и создаваемых с ее помощью менеджеров сущностей.

- Множество классов сущностей, управляемых с помощью менеджеров сущностей данного модуля.
- Метаданные (в форме аннотаций или XML-дескриптора) об отображении классов сущностей на базу данных.

Модуль персистентности позволяет отобразить управляемые с его помощью классы сущностей на единственную реляционную базу данных, определяя, таким образом, контекст для запросов и связей между сущностями. Это позволяет представить реляционную БД сложной структуры в виде композиции проблемно-ориентированных модулей персистентности.

Модуль персистентности может быть определен в EJB-модуле, веб-модуле, модуле клиентского приложения или в Java EE-приложении с помощью файла `persistence.xml`. Данный файл может содержать определения нескольких модулей персистентности. Каждый модуль персистентности должен обладать уникальным именем в пределах родительского модуля (или приложения).

Контекст персистентности

Контекст персистентности (persistence context) представляет собой множество экземпляров сущностей одного модуля персистентности, в котором любому постоянному идентификатору сущности соответствует единственный экземпляр сущности. В контексте хранятся экземпляры сущностей, которые были прочитаны из БД и/или которые нужно сохранить в БД.

С контекстом персистентности всегда связан экземпляр менеджера сущностей (объект типа `javax.persistence.EntityManager`), который управляет жизненным циклом экземпляров сущностей, находящихся в рамках данного контекста.

В интерфейсе `EntityManager` определены следующие группы методов:

- управление жизненным циклом сущностей (`persist()`, `refresh()`, `remove()`, `merge()`);
- поиск экземпляра сущности по первичному ключу (`find()`, `getReference()`);
- создание объектов типа `Query`, предназначенных для выполнения запросов (`createNamedQuery()`, `createQuery()`, `createNativeQuery()`);
- управление контекстом персистентности (`flush()`, `clear()`, `close()`, `getTransaction()` и т.д.).

Контекст персистентности существует только во время выполнения приложения (пока существует связанный с ним менеджер сущностей) и имеет ограниченное время жизни: в контексте транзакции или расширенное, охватывающее несколько транзакций. По окончании времени жизни контекст очищается.

Контекст персистентности может управляться контейнером или приложением.

Контекст персистентности, управляемый контейнером:

- обеспечивает простоту использования в среде Java EE;
- передается между компонентами вместе с JTA-транзакцией;
- получается через инъекцию зависимости (аннотация `PersistenceContext`) или поиск в окружении компонента;
- может иметь как расширенное время жизни (только при использовании в сессионных компонентах с состоянием), так и время жизни в контексте транзакции.

Контекст персистентности, управляемый приложением:

- используется в средах Java SE и Java EE;
- получается с помощью фабрики менеджера сущностей;
- имеет расширенное время жизни.

Получить менеджер сущностей для контекста, управляемого контейнером, можно с помощью инъекции зависимости или поиском в окружении EJB-компонента. Первый вариант реализуется с помощью аннотации `PersistenceContext`, элементы которой

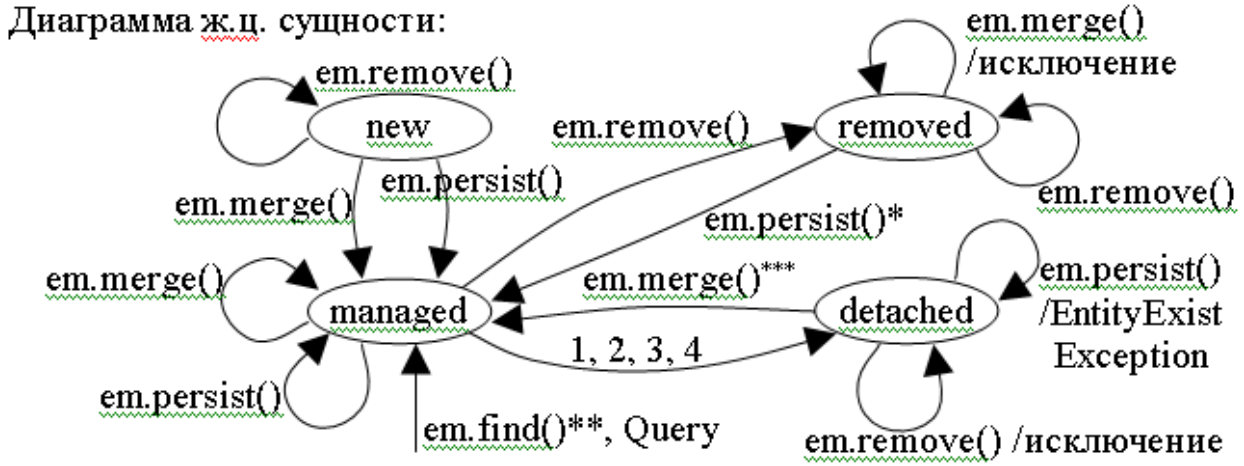
позволяют указать имя модуля персистентности (`unitName`) и время жизни контекста (`type`), например:

```
@PersistenceContext(type=PersistenceContextType.TRANSACTION,
unitName="orders")
EntityManager orderEM;
```

Жизненный цикл экземпляра сущности

Выделяют 4 состояния экземпляра сущности:

- новый экземпляр сущности (new) не имеет постоянного идентификатора и не связан с контекстом персистентности;
- управляемый экземпляр сущности (managed) имеет постоянный идентификатор, связанный с контекстом персистентности;
- отсоединенный экземпляр сущности (detached) имеет постоянный идентификатор, который (более) не связан с контекстом персистентности;
- удаленный экземпляр сущности (removed) имеет постоянный идентификатор, связанный с контекстом персистентности, и запланирован на удаление из базы данных.



* – до того как контекст ещё существует.

** – находит значение экземпляра сущности по первичному ключу.

*** – позволяет сохранить изменения в экземпляре сущности.

1 – передача объекта по значению.

2 – подтверждение или откат транзакции.

3 – em.close() – закрытие контекста персистентности.

4 – em.clear() – очистка контекста персистентности.

Для распространения действия операции над текущей сущностью на связанные с ней сущности предназначен элемент аннотации `cascade`. Каскадирование часто применяется в отношениях «часть-целое».

Обработчики событий ЖЦ сущности

Существует возможность обработки событий ЖЦ сущности с помощью методов класса сущности или методов отдельного класса. Эти методы могут выбрасывать неконтролируемые исключения, что приводит к откату транзакции. В методах обработки событий ЖЦ можно вызывать интерфейсы JNDI, JDBC, JMS и обращаться к сессионным компонентам, но не рекомендуется вызывать методы интерфейсов `EntityManager` и `Query`, работать с другими экземплярами сущностей и изменять связи между сущностями.

Метод обработки события ЖЦ сущности, определенный в классе сущности или в отображенном суперклассе, должен иметь сигнатуру:

```
void <METHOD>()
```

, а обработчик, определенный в классе обработки событий ЖЦ сущности, должен иметь сигнатуру

```
void <METHOD>(Object)
```

, где в качестве параметра передается экземпляр сущности, с которым произошло событие.

Класс обработки событий ЖЦ сущности назначается для класса сущности с помощью аннотации `EntityListeners`.

Различают следующие события ЖЦ и соответствующие им обработчики:

- `PrePersist` – вызывается прежде других операций в рамках метода `persist()`;
- `PostPersist` – вызывается после выполнения операции вставки в БД для экземпляра сущности (сразу после вызова метода `persist()` или при синхронизации контекста персистентности с БД), в обработчике доступно сгенерированное автоматически значение первичного ключа;
- `PreRemove` – вызывается прежде других операций в рамках метода `remove()`;
- `PostRemove` – вызывается после выполнения операции удаления из БД экземпляра сущности (сразу после вызова метода `remove()` или при синхронизации контекста персистентности с БД);
- `PreUpdate` – вызывается перед обновлением экземпляра сущности в БД;
- `PostUpdate` – вызывается после обновления экземпляра сущности в БД;
- `PostLoad` – вызывается после загрузки экземпляра сущности из БД в текущий контекст персистентности либо после выполнения метода `refresh()` и перед любыми операциями приложения с этим экземпляром (перед возвратом результатов запроса и перед навигацией ассоциации).

Каскадирование операций с сущностями приводит к вызову обработчиков событий ЖЦ для связанных сущностей.

Обработчики событий ЖЦ сущности позволяют выполнять проверку правильности состояния сущности непосредственно перед или после синхронизации состояния сущности с БД, то есть в моменты, когда постоянное состояние сущности полностью сформировано. Это позволяет обойти ограничения, свойственные проверке правильности с помощью set-методов, и обойтись доступом СОРП к состоянию сущности через поля, а не через свойства класса сущности.