



Рафеев Е.Д.

Web- программирование

Технология Java Persistence API

Содержание

- ▶ Описание сущностей и отношений между ними с помощью аннотаций.
- ▶ Управление сущностями.
- ▶ Persistence Contexts.
- ▶ Компоновка сущностей



Введение

- Объектно-реляционное преобразование (ORM) – сохранение Java объектов в реляционную базу данных. Технологии: Entity Beans 2.x, TopLink, Hibernate, JDO и др.

Экспертная группа Java EE взяла за основу эти популярные фреймворки и создала Java Persistence API.

- Java Persistence API предоставляет POJO persistence модель объектно-реляционного преобразования. Эта технология была разработана экспертной группой EJB 3.0. Ее применение не ограничивается компонентами EJB. Она может быть использована:
 - web приложениями,
 - клиентами напрямую, вне Java EE платформы, например в Java SE приложениях.



Java Persistence API

предоставляет:

- стандартный механизм для ORM,
- EntityManager API для создания, обновления и удаления объектов,
- язык запросов JPA-QL для извлечения сущностей.



Описание сущностей и отношений между ними

- В EJB3 JPA используются аннотации для определения объектов и отношений.
 - логические аннотации (позволяют описать объектную модель, связи между классами)
 - физические аннотации (описывающие физическую схему, таблицы, столбцы, индексы и т.д.).

EJB3 аннотации находятся в пакете
javax.persistence.*



Объявление сущности

- Сущность (Entity) – это простой объект (POJO), который нужно сохранить в реляционной базе данных. Сущность может быть как абстрактным, так и конкретным классом, а также может наследовать другой POJO. Для того чтобы объявить POJO сущностью, его нужно отметить аннотацией **@Entity**.
- Каждая сущность имеет первичный ключ. Указать, какое сохраняемое поле является первичным ключом, можно с помощью аннотации **@Id**. Сущность управляет состоянием, используя либо поля, либо get и set методы.



Объявление сущности

@Entity

```
public class ClassName implements Serializable{
```

```
    @Id
```

```
    Long id;
```

```
        public Long getId() {
```

```
            return id;
```

```
        }
```

```
        public void setId(Long id) {
```

```
            this.id = id;
```

```
        }
```

```
    }
```



Определение таблицы

- Аннотация **@Table** указывается на уровне класса. Она позволяет указать таблицу, каталог и схему, которые соответствуют сущности. Если эта аннотация не указана, в качестве имени таблицы воспринимается имя класса.

@Entity

@Table(name="table_name")

```
public class ClassName implements Serializable
```

- Можно также указать уникальные столбцы с помощью аннотации **@UniqueConstraint**:

@Table(name="tbl_name",

uniqueConstraints =

{ @UniqueConstraint(columnNames={ "columnName" }) }

)



Построение соответствий простых свойств

- Каждое нестатическое и не временное свойство сущности считается сохраняемым, если не указана аннотация **@Transient**.
- Объявление без аннотаций соответствует объявлению с аннотацией **@Basic**. Эта аннотация позволяет указать стратегию выборки для свойства.



```
public transient int counter; //временное свойство
private String firstname;    //сохраняемое свойство
@Transient
String getLengthInMeter() { ... } //временное свойство
String getName() {... } // сохраняемое свойство

@Basic
int getLength() { ... } // сохраняемое свойство

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... }
// сохраняемое свойство, будет извлекаться из базы при первом доступе
к полю (отложенная выборка)

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } //сохраняемое свойство
@Enumerated(STRING)
Starred getNote() { ... } //перечисление, сохраняемое в базе как строка
@LOB – свойство сохраняется в Blob или Clob
```



Определение атрибутов столбца

- Столбцы, соответствующие свойствам, могут быть определены с помощью аннотации **@Column**. Ее используют для переопределения значений по умолчанию.



Определение атрибутов столбца

```
@Column(  
    name="columnName";                (1)  
    boolean unique    default false;   (2)  
    boolean nullable   default true;    (3)  
    boolean insertable default true;     (4)  
    boolean updatable  default true;     (5)  
    String  columnDefinition default "";  (6)  
    String  table       default "";       (7)  
    int     length      default 255;      (8)  
    int     precision() default 0; // decimal precision (9)  
    int     scale()     default 0; // decimal scale  
)
```



Вложенные объекты

- Внутри сущности можно объявить вложенные компоненты, переопределить столбцы. Классы компонент должны иметь аннотацию **@Embeddable** на уровне класса.
- Переопределить маппинг столбца вложенного объекта можно с помощью аннотаций **@Embedded** и **@AttributeOverride**.

Вложенные объекты наследуют тип доступа от содержащей его сущности



@Embeddable

```
public class Address implements Serializable {  
    String city;  
    Country nationality; //no overriding here  
}
```

@Embeddable

```
public class Country implements Serializable {  
    private String iso2;  
    private String name;  
    public String getIso2() { return iso2; }  
    public void setIso2(String iso2) { this.iso2 = iso2; }  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    ...  
}
```



@Entity

```
public class Person implements Serializable {  
    // Persistent component using defaults  
    Address homeAddress;  
    @Embedded  
    @AttributeOverrides( {  
        @AttributeOverride(name="iso2",  
            column = @Column(name="bornIso2")),  
        @AttributeOverride(name="name",  
            column = @Column(name="bornCountryName") )  
    } )  
    Country bornIn;  
    ...  
}
```



Построение соответствий идентификаторов

- Аннотация `@Id` позволяет определить, какое свойство будет использовано в качестве идентификатора сущности. Стратегию генерирования идентификатора можно указать, используя аннотацию `@GeneratedValue`. Использование стратегии `IDENTITY` или `SEQUENCE` дает результаты, зависящие от СУБД:

Например, для базы данных MySQL создается столбец `AUTO_INCREMENT`

`@Id`

`@GeneratedValue(strategy = GenerationType.IDENTITY)`



Описание составных первичных ключей:

Используется вложенный компонент и аннотация **@IdClass**.

@Entity

@IdClass (StudentPk.class)

```
public class Student {  
    // part of the id key  
    @Id public String getFirstname() {  
        return firstname;  
    }  
    public void setFirstname(String firstname) {  
        this.firstname = firstname;  
    }  
    //part of the id key  
    @Id public String getLastName() {  
        return lastname;  
    }  
    public void setLastName(String lastname) {  
        this.lastname = lastname;  
    }  
    //appropriate equals() and hashCode() implementation }  
}
```




@Embeddable

```
public class StudentPk implements Serializable {  
    //same name and type as in Student  
    public String getFirstname() {  
        return firstname;  
    }  
    public void setFirstname(String firstname) {  
        this.firstname = firstname;  
    }  
    //same name and type as in Student  
    public String getLastName() {  
        return lastname;  
    }  
    public void setLastName(String lastname) {  
        this.lastname = lastname;  
    }  
    //appropriate equals() and hashCode() implementation  
}
```



Преобразование наследования

- EJB3 поддерживает три типа наследования
 - Стратегия «по таблице на каждый класс».
 - Стратегия «одна таблица на иерархию классов».
 - Стратегия «составной подкласс».



Стратегия «по таблице на каждый класс»

- Каждому классу соответствует своя таблица в базе данных. *Преимущества:* для работы не нужно соединять несколько таблиц, если подклассы также являются сущностями.

Недостатки: база становится не нормализованной, эта стратегия слабо поддерживает объединение таблиц, соответствующих классам одного уровня, а так же полиморфизм.

Реализация: перед классом-родителем нужно указать аннотацию

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS).

Эта стратегия обычно используется для верхнего уровня иерархии. Она не поддерживает стратегию IDENTITY generator, так как id является общим для нескольких таблиц.



Стратегия «одна таблица на иерархию классов»

- *Преимущества:* простота и отсутствие необходимости соединять таблицы.
Недостатки: база становится ненормализованной. При этом столбцы, соответствующие полям подклассов, должны допускать нулевые значения, большое количество столбцов в таблице.
Реализация: нужно указать аннотацию **@Inheritance(strategy = InheritanceType.SINGLE_TABLE)**, а также дискриминатор, который будет использоваться для различения классов.



Стратегия «одна таблица на иерархию классов»

```
@Entity
```

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

```
@DiscriminatorColumn(
```

```
    name = "studenttype"
```

```
    discriminatorType=DiscriminatorType.STRING
```

```
)
```

```
@DiscriminatorValue("Student")
```

```
public class Student { ... }
```

```
@Entity
```

```
@DiscriminatorValue("FullTimeStudent")
```

```
public class FullTimeStudent extends Student { ... }
```



Стратегия «составной подкласс»

Преимущества: база данных уже нормализована. При этом представления (view) базы данных имеет ту же структуру, что и объектная модель и поэтому последнюю можно легко выделить.

Недостатки: снижение производительности в глубоких иерархиях, полиморфных запросах и отношениях.

Реализация: указать аннотацию

`@Inheritance(strategy=InheritanceType.JOINED)`. По умолчанию таблицы соединяются по одноименным первичным ключам. Если первичные ключи различны, перед ключом в подклассе указывают аннотацию `@PrimaryKeyJoinColumn(name="<column_name>")`.



Преобразование связей

- Отношения между сущностями бывают следующих типов:
 - один к одному,
 - один ко многим,
 - многие к одному,
 - многие ко многим.

Для реализации этих отношений, свойства сущностей отмечаются соответствующими аннотациями.



Одно- и двунаправленные связи

- Двунаправленная связь описывается как с владеющей стороны, так и с обратной.
- Однонаправленная связь описывается только с владеющей стороны (владеющая сторона – класс, по объектам которого будет осуществляться обновление базы данных)



Правила реализации двунаправленной СВЯЗИ

- Обратная сторона отношения должна ссылаться на владеющую с помощью элемента `mappedBy`
- Сторона, соответствующая «многим» в отношениях *многие к одному* и *один ко многим* должна быть владеющей, следовательно элемент `mappedBy` не может быть указан в аннотации `@ManyToOne`
- В отношении один к одному владеющая сторона соответствует стороне, содержащей внешний ключ
- В отношении многие ко многим любая сторона может быть владеющей



Реализация связи

- сущности имеют общий первичный ключ (связь один к одному) - указывается аннотация **@PrimaryKeyJoinColumn**;
- одна из сущностей содержит внешний ключ (связи многие к одному, один ко многим) — указывается аннотация **@JoinColumn** с атрибутом *name*, содержащим имя связующего столбца;
- использование таблицы связи - указывается аннотация **@JoinTable** с атрибутами:
name (имя таблицы связи),
joinColumns (список связующих столбцов из текущей таблицы),
inverseJoinColumns (список связующих столбцов из обратной таблицы)



Уровень каскадности

В атрибуте `cascade` в аннотации отношения указывают массив следующих типов:

`CascadeType.PERSIST` – делает каскадной операцию создания при вызове метода `persist()`;

`CascadeType.MERGE` – делает каскадной операцию `merge()`;

`CascadeType.REMOVE` – делает каскадным операцию удаления;

`CascadeType.REFRESH` – делает каскадной операцию обновления;

`CascadeType.ALL` – все выше перечисленное.



Управление сущностями

- Объект `EntityManager` связан с `persistence context`.
Persistence context – это множество объектов сущностей, в котором каждому идентификатору соответствует единственный объект.
`EntityManager` управляет жизненным циклом сущностей в этом `persistence context`. Интерфейс `EntityManager` определяет методы, которые используются для взаимодействия с `persistence context`.



Операции над объектами

- Новый объект не имеет сохраненного идентификатора и еще не связан с persistence context.
- Управляемый объект – это объект с сохраненным идентификатором и связанный с persistence context.
- Отделенный объект – это объект с сохраненным идентификатором, но уже не связанный с persistence context.
- Удаленный объект – это объект с сохраненным идентификатором, связанный с persistence context, который планируется удалить из базы данных.



Операции над объектами

- **persist**(Object entity) делает новый объект *управляемым*.
- *Удаление* управляемого объекта происходит с помощью метода **remove**(Object entity), либо каскадно, если отношение определено соответствующим образом.
- Состояние управляемых объектов синхронизируется с базой данных при вызове метода **commit()** транзакции. Эта синхронизация включает в себя запись в базу данных всех изменений объектов и их отношений.
Синхронизация с базой не включает в себя *обновление* всех объектов. Для этого нужно явно вызывать метод **refresh()**. Синхронизацию можно произвести также методом **flush()**.



Операции над объектами

- Управляемый объект становится *отделенным* в результате:
 - вызова метода **commit()** или **rollback()** транзакции,
 - после очистки persistence context,
 - после закрытия EntityManager,
 - после сериализации объекта.

Такие объекты продолжают свой жизненный цикл вне persistence context, но уже не будут синхронизироваться с базой данных.

- Операция merge позволяет восстановить отделенный объект до управляемого. Эта операция каскадна.
- Метод **contains()** определяет, является ли объект управляемым.
- Загрузить объект можно с помощью метода **load()**, указав тип загружаемого объекта и первичный ключ.



Persistence Contexts

- это множество управляемых сущностей, в котором каждому уникальному идентификатору соответствует единственная сущность. В persistence context сущности и их жизненный цикл управляются объектом **EntityManager**.
- EntityManager может быть *управляемым контейнером*. Его жизненный цикл управляется контейнером Java EE.
управляемым приложением.



Получение объекта EntityManager

- Для получения объекта EntityManager, *управляемого контейнером*, используется аннотация **@PersistenceContext**, с возможным атрибутом name, указывающим имя persistence unit.
- Для получения объекта, *управляемого приложением*, нужно вызвать метод **createEntityManager** интерфейса EntityManagerFactory.
- Создание объекта EntityManagerFactory требует достаточно много ресурсов, объект создается один раз за все время работы приложения.
Получить объект EntityManagerFactory можно либо с помощью аннотации **@PersistenceUnit** (среда Java EE), либо с помощью метода Persistence.createEntityManagerFactory (среда Java SE).



Управление транзакциями

В зависимости от типа транзакции объекта EntityManager, транзакции контролируются либо JTA(Java Transaction Architecture) либо локальным EntityTransaction API.

- В первом случае объекты EntityManager используют текущую JTA транзакцию, которая начинается и фиксируется вне объекта EntityManager.
- Во втором случае объект EntityManager использует транзакцию, связанную с источником данных через persistence provider. Такие объекты EntityManager могут использовать серверные или локальные источники.

В этом случае транзакция управляется методами интерфейса **EntityTransaction**. Получить объект, реализующий этот интерфейс, можно с помощью метода `entityManager.getTransaction()`.



Компоновка сущностей

- Компоновка сущностей осуществляется с помощью persistence unit. **Persistence unit** — это логическая группировка, которая включает:
 - EntityManagerFactory и ее EntityManager вместе с их конфигурационной информацией
 - Множество управляемых классов
 - Метаданные преобразований, которые указывают соответствия с базой данных
 - Persistence unit описывается в XML файле persistence.xml, который располагается в папке META-INF проекта.



Структура файла persistence.xml

- в корневом элементе `<persistence>` находится один или несколько элементов `<persistence-unit>`. Этот элемент имеет атрибут `name` и `transaction-type` (JTA или `RESOURCE_LOCAL`). Конфигурация `persistence unit` определяется следующими элементами
 - `description` – описание `persistence-unit`;
 - `provider` – имя класса, реализующего интерфейс `javax.persistence.spi.PersistenceProvider`;
 - `jta-data-source`, `non-jta-data-source` – указывают глобальное JNDI имя соответствующего источника данных;
 - `class` – аннотированный класс сущности;
 - `properties` – специфические свойства, зависящие от реализации. Это могут быть параметры соединения с базой данных, указание уровня логгирования и т.п.



Структура файла persistence.xml

<persistence >

`<persistence-unit name="simple_factory" transaction-type="RESOURCE_LOCAL">`

`<provider>oracle.toplink.essentials.PersistenceProvider`

`</provider>`

`<class>by.red.ex1.Course</class>`

`<class>by.red.ex1.Student</class>`

`<class>by.red.ex1.Address</class>`

`<properties>`

`<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/university?characterEncoding=utf8&serverTi
mezone=UTC"/>`

`<property name="javax.persistence.jdbc.user" value="root"/>`

`<property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver"/>`

`<property name="javax.persistence.jdbc.password" value="***"/>`

`<!-- <property name="eclipselink.ddl-generation" value="create-tables"/>`

`</properties>`

`</persistence-unit>`

</persistence>



Информационные ресурсы

Организационные вопросы

