前言

本笔记的由来：首先我是 jpa 的爱好者，这个爱好导致我疏远了 Hibernate，当然没有说 Hibernate 不好的意思，Hibernate3 的新特性也支持注解。或许感觉 jpa 是规范，并且或多或少也想接触 ejb3 的缘故。当然这个笔记也是在我阅读了 ejb3 in action 后进行书写的。

笔记的特点：摘取了 ejb3 in action 里面的一些简短重要的讲解，大部分的篇幅是我一些不伦不类的想法和理解吧，因为个人水平有限，也只能用不伦不类描述了。不过请不要仅仅只看了前面一点内容就评论我很垃圾。

笔记阅读者适合人群：本人或多或少做了几个使用 jpa 的项目。所以希望阅读者，也需要或多或少对 jpa 有些初步的了解。

声明

（本人语文学的很烂，所以会有很多错别字和语误，请谅解）

搭建环境

我使用的是 Hibernate3

需要的 jar：

Hiberante 核心包(8 个文件)

hibernate-distribution-3.3.1.GA

---

hibernate3.jar

lib\bytecode\cglib\hibernate-cglib-repack-2.1_3.jar

lib\required*.jar

Hiberante 注解包(3 个文件)：hibernate-annotations-3.4.0.GA

---

hibernate-annotations.jar

lib\ejb3-persistence.jar、hibernate-commons-annotations.jar

Hibernate 针对 JPA 的实现包(3 个文件)：hibernate-entitymanager-3.4.0.GA

---

hibernate-entitymanager.jar

lib\test\log4j.jar、slf4j-log4j12.jar

JPA 规范要求在类路径的 META-INF 目录下放置 persistence.xml，文件的名称是固定的，配置模版如下：

```
<?xml version="1.0"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">
  <persistence-unit name="partner4java"
transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5Dialect"/>
        <property name="hibernate.connection.driver_class"
value="org.gjt.mm.mysql.Driver"/>
        <property name="hibernate.connection.username" value="root"/>
        <property name="hibernate.connection.password"
value="123456"/>
        <property name="hibernate.connection.url"
value="jdbc:mysql://localhost:3306/jpa?useUnicode=true&amp;characterE
ncoding=UTF-8"/>
        <property name="hibernate.max_fetch_depth" value="3"/>
        <property name="hibernate.hbm2ddl.auto" value="update"/>
        <property name="hibernate.jdbc.fetch_size" value="18"/>
        <property name="hibernate.jdbc.batch_size" value="10"/>
        <property name="hibernate.show_sql" value="true"/>
        <property name="hibernate.format_sql" value="false"/>
     </properties>
  </persistence-unit>
</persistence>
```

配置文件你可以从网上或者包里面带的例题了找或者从使用包里面的默认配置文件里面找。

并且配置文件的位置和名称是固定不允许改变的，在使用 spring 时，文件名可以改变。

每个配置项的具体含义自己 google 一下吧，很简单，我就懒得打上了。

导读：

第一部分：主要是想给您灌输一下域模型的概念，初步接触实体 bean 的实现方式，让您对实体 bean 有初步的思想理念（当然我省略了很多面向对象的思想概念，这也就最开始希望您对 JPA 有所基本的了解）

第二部分：详细说明了如何进行对象关系映射。

第一部分：实现域模型

1. 一些基本理论

域模型是和持久化是密不可分的概念。通常，开发企业项目的第一个步大都就是创建域模型，即列出域中的实体并且定义实体之间的关系。（当然这基本说的是领域建模，好多公司的项目还是采用的数据库建模的形式。）

1.1 域模型的相关概念

域模型的四个参与者：对象、关系、关系多样性和关系的可选性。

1》 对象

与 java 对象一样，域模型可以同时具有行为（在 java 在的方法）和状态（java 的实体变量）

2》 关系

从 java 角度来看，关系表现为一个对象引用另一个对象。

充血域模型与贫血域模型

域模型实际上被持久化到数据库中，数据库建模经常和域建模是同义的，域对象包含属性（映射到数据库表的列）而非行为，这种类型的建模被称为贫血模型。

充血模型封装对象的属性、行为，并且利用面向对象实用程序设计方法。（比如继承、多态和封装）。域模型充血度越高，就越难以映射到数据库。

1.2 Ejb3 java 持久化 API

Ejb3java 持久化 API 是元数据驱动的 pojo 技术。也就是说，为了把 java 对象中的数据保持到数据库中，我们的对象不必实现接口、扩展类，或采用框架模式。

（简单的 java 对象实际就是普通的 javabean 是使用 pojo 名称是为了避免和 ejb 混淆起来）

声明：如果你熟悉 sql 和 jdbc 并且喜欢通过自己动手的便利方式得到控制权和灵活性，那么 O/R 的成熟、技巧和自动化形式可能不适合你。如果情况是这样，那么你应该多了解 Spinrg JdbcTemplate（这个我不喜欢用）和 iBATIS（这个框架应该对你对对象的理解要求低很多）。

可是我要警告您，java 本身就是对象化的语言，我们要完美化，就要一切为对象。

2.使用 JPA 实现域对象

本章意在描述域对象的实现过程，下一张将详细介绍如何使用对象-关系映射把我们创建的实体和关系映射到数据库中。

（当然现在 JPA 不仅仅是 EJB3 值得骄傲的地方，其它好多公司也相继实现了 JPA 规范，做出了很多优秀的 orm 框架。如：hibernate、openjpa、toplink 等。并且规范的好处就在于，你只要完全按照规范来写代码，这些框架你可以随意更换，并是不更改代码的前提下。）

书签：下面我们主要说的是如何定义一个实体 bean，也就是如何把你的 POJO 映射为数据库对象。

2.1 @Entity

@Entity 注解，它把 POJO 转换为实体。（以后我们常叫这样的 POJO 为实体 Bean）

（jpa 主要是支持注解的形式，简单是说也就是使用注解对你对象化的 POJO 进行声明表示，生成对应的数据库和对数据库进行对象化的封装。

那么，我们学习 JPA 也就无非是学习几个注解，并且通过我们对对象的理解，标识对象的关系等。我们常用的注解无法就二三十个注解，所以 JPA 会让你感觉原来那么简单）

```
package cn.partner4java.bean.part2_1;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class User {
    @Id
    String userId;
    String username;
    String email;
}
```

2.2 持久化实体数据

？我们注解标识的位置：

持久化分为两种：基于字段与基于属性的持久化。

使用实体的字段或实体变量定义的 O/R 映射被称为基于字段的访问，也就是注解标注在字段上面。当然基于字段如果你省略了 get 和 set，也可以生产成功。

如：

```
package cn.partner4java.bean.part2_1;
import javax.persistence.Entity;
import javax.persistence.Id;
```

```
@Entity
public class User {
    @Id
    String userId;
    String username;
    String email;
}
```

基于属性，必须标注在 get 方法上面，而非 set。

如：

```
package cn.partner4java.bean.part2_1;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class User {
    String userId;
    String username;
    String email;
    @Id
    public String getUserId() {
        return userId;
    }
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

}
```

注意：同一个类中，不可同时使用两种方式。因为两种方式的内部实现方式有所区别。（当然你标注在哪里看你个人爱好了，我比较喜欢标注在字段上面，因为我查找和修改的时候方便一些，我也见很多人标注在属性上面，具体哪种性能上更优秀一些，寡人没具体研究过，

的需要查看源码了，并且每种实现产品又不一样，所以一直懒得去看）

建议：当您定义字段时，最好声明为私有的。

因为持久化提供器可能需要覆盖设置器和获取器，所以不能持久化的设置器和获取器声明为 final。也就是 get 和 set。

？如果你也随着我说的，自己打并执行了上面的代码，你会发现 username 和 email 我们并没有进行任何的标识，也生产了相应的数据库字段，但是如果我们不想生成该怎么办？定义**顺时字段**：加上@Transsient。也就是有些字段你不想映射进数据库时。

这里我们就简单的说了一下，后面我们会通过其他的解决，顺便解决了这里的详细使用。

2.3 指定实体身份

？为什么要有实体身份并且规定是必须进行指定：上面我们介绍了第一个注解@Entity，把 POJO 声明为实体 bean。这是不够的，因为实体在某个时间必须被持久化为数据库表中唯一可表示的行（或多个表中行的集合），如果没有进行标识，你就不知道在执行保存操作之后数据被保持到哪一行了。（我们还需要重写 equals 方法，是比较两个明显不同数据库记录的主键在对象中的等效方式）

身份表示注解：

a．@Id

b．@IdClass

c．@EmbeddedId

？问题：怎么会有 3 种注解，我平时用到的仅仅是@Id，我感觉应该学着带着问题和思考去学习一个知识。

？什么样的类型可标识为实体身份：EJB3 支持原始类型、原始包装器和 Serializable 类型（比如 java.lang.String、java.util.Date、java.sql.Date）作为身份，此外因为 float、Float、double 等类型不确定的特点，当选址数据类型时应该避免使用这些类型。

第一种：@Id，相信对 jpa 有一点点了解的人就会知道这是做什么的，怎么用。

？后两种注解出项的原因：你有时使用一个以上的属性或字段（复合键）唯一的标识实体。

第二种：@IdClass

如：（一般通过例子就完全可以理解的，我就不会多说了）

```
package cn.partner4java.bean.part2_3;
import java.io.Serializable;
import java.util.Date;
public class CategoryPK implements Serializable {
```

```
    private String name;
    private Date CreateDate;
    …hashCode()…equals(Object obj)
```

```
package cn.partner4java.bean.part2_3;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;
@Entity
@IdClass(CategoryPK.class)
public class Category {
    @Id
    private String name;
    @Id
    private Date CreateDate;
    private String realname;
}
```

第三种：@EmbeddedId

```
package cn.partner4java.bean.part2_3;
import java.io.Serializable;
import java.util.Date;
import javax.persistence.Embeddable;
@Embeddable
public class CategoryPK implements Serializable {
    private String name;
    private Date CreateDate;
。。。
```

```
package cn.partner4java.bean.part2_3;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
@Entity
public class Category {
    @EmbeddedId
    private CategoryPK categoryPK;
    private String realname;
}
```

有些地方说：这里的 CategoryPK 可以不实现序列化 Serializable，但事实是必须也实现。因为所有的字段都必须是序列化的。

这后两种生产复合主键的形式，都可以，看您个人爱好了

2.4 @Embeddable 注解

上面我们第三种方式用到了这个注解进行复合主键对象的声明。它主要用于嵌入对象，我们平时常用的是域对象。如实体关系，一对一。

使用场景：纯粹的面向对象的域建模，所有的域对象总都是可以独立标识的么？如果对象仅仅在其他的对象内部用作方便的数据占位器/组织方式会怎么样？一个常见的例子，是 User 对象内部使用 Address 对象，它作为优雅的面向对象的替换方式，替代列出街道地址、城市、邮编等直接作为 User 对象的字段。因为不太可能在 User 之外使用 Address 对象，所以 Adress 对象没有必要具有身份，这正好是@Embeddable 的适用场景。

如：

```
package cn.partner4java.bean.part1_2_4;
import javax.persistence.Embeddable;
@Embeddable
public class Address {
    private String city;
    private String state;
    private String country;
}
```

```
package cn.partner4java.bean.part1_2_4;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class User {
    @Id
    private Long id;
    private String username;
    @Embedded
    private Address address;
}
```

生成的表仅一个。但是我们如果想生成的是两个表，怎么办呢？是不是有什么属性表示一下？但是我们上面就说过，如果是单独表映射就需要身份表示，否则我们就不知道是对哪个字段进行操作。但是嵌入对象不需要唯一身份。那么就的选择使用实体关系的方式。

3.实体关系

域关系类型和相应注解

关系类型	注解

一对一	@OneToOne
一对多	@OneToMany
多对一	@ManyToOne
多对多	@ManyToMany

3.1 @OneToOne

单向一对一：

User 对象具有对 BillingInfo 的引用，但是没有反向引用，换句话说，次关系是单向的。

```
package cn.partner4java.bean.part1_3_1;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class BillingInfo {
    @Id
    private Long billingId;
    private String bankName;
}

package cn.partner4java.bean.part1_3_1;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class User {
    @Id
    private String userId;
    private String email;
    @OneToOne
    private BillingInfo billingInfo;
}
```

在 User 表里面生成了相应的关系引用字段。

```
@OneToOne(optional=false)
private BillingInfo billingInfo;
```

Optional 元素：通知持久化提供器相关对象是否必须存在。默认情况下为 true，不必存在。

双向一对一：

你需要从关系的任何一端访问相关实体：

```
package cn.partner4java.bean.part1_3_1;
import javax.persistence.Entity;
```

```
import javax.persistence.Id;
import javax.persistence.OneToOne;
@Entity
public class BillingInfo {
    @Id
    private Long billingId;
    private String bankName;
    @OneToOne(mappedBy="billingInfo",optional=false)
    private User user;
}
```

```
package cn.partner4java.bean.part1_3_1;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;
@Entity
public class User {
    @Id
    private String userId;
    private String email;
    @OneToOne
    private BillingInfo billingInfo;
}
```

mappedBy=”billingInfo”指定，它通知容器关系的“拥有”端在 User 类的 billingInfo 实体变量中，也就是在表 User 里生产外键引用字段。

Optional参数被设置为false，标识BillingInfo对象不能在没有相关User对象时存在。想想，你如果在User的：

```
@OneToOne(optional=false)
private BillingInfo billingInfo;
```

也设置为不能为空，会产生什么后果，保存执行 sql 语句会产生什么后果？（这个问题留在后面 EntityManager 使用时解决）

3.2 @OneToMany 和 @ManyToOne

```
package cn.partner4java.bean.part1_3_2;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToMany;
@Entity
public class Item {
    @Id
    private Long id;
```

```
    private String title;
    @OneToMany(mappedBy="item")
    private Set<Bid> bids;
}
```

```
package cn.partner4java.bean.part1_3_2;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class Bid {
    @Id
    private Long bidId;
    private Double amount;
    @ManyToOne
    private Item item;
}
```

在关系中的非维护端的实体的列上使用 mappingBy，从而标记关系的维护端，指定为 Bid 的 item 字段为维护端。并且规定必须多的一端为维护端。因为，让世界人名记住小泉的名字，总比让小泉记住世界人民的名字容易的多。

3.3 @ManyToMany

虽然多对多关系可以是单向的，但是由于这种很想特殊、相互独立的性质，所以他们经常是双向的。

```
package cn.partner4java.bean.part1_3_3;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
@Entity
public class CateGory {
    @Id
    private Long categoryId;
    private String name;
    @ManyToMany
    private Set<Item> items;
}
```

```
package cn.partner4java.bean.part1_3_3;
import java.util.Set;
import javax.persistence.Entity;
```

```
import javax.persistence.Id;
import javax.persistence.ManyToMany;
@Entity
public class Item {
    @Id
    private Long itemId;
    private String title;
    @ManyToMany(mappedBy="items")
    private Set<CateGory> categories;
}
```

双向的多对多会单独生成一个表进行关系的维护。

总结：我们上面介绍了各种关系，但是没有详细告诉大家一些注解参数的配置，如改变多对多关系表的名称等等，相信这些还是比较简单的东西，大家可以从文档里面查找。并且下面的第二部分会有详细的介绍。

第二部分：关系对象映射

ORM 是 JPA 的基础。实际上，ORM 指定如何把一组 java 对象（包括对象之间的引用）映射到数据库表的行和列。

我们从介绍 O/R 映射背后的基本动机（即所谓的阻抗失配）开始讨论，然后通过映射域对象开始分析，再转向映射关系的讨论。最后讲解映射继承的概念，你将学习 JPA 支持的继承策略。

1．1 阻抗失配

？什么是阻抗失配：这一术语是面向对象和（数据库的）关系范例之间的区别以及应用程序开发工作中因这些区别引起的问题。域模型所在的持久化层通常是阻抗失配表现的最明显的地方，问题的根源在于两种技术的基本目的之间的区别。如：JVM 提供丰富的继承和多态，而（数据库的）关系领域中不存在这些特性。数据库表固定的只有行、列和约束，并不包含业务逻辑。

1.1.1 把对象映射到数据库

仅是介绍一下映射概念。

1.一对一映射

我们上面说过嵌入对象和域对象。当对象不需要身份时，就可以考虑使用嵌入对象，这样会生成进同一张表。但是当表结构过于庞大时，会影响我们的查询效率，且我们的 ORM 框架需要挨个封装成对象的数据，也会相应的影响些效率，这时就需要考虑域对象，如果双方都需要得到对方的话，就是双向一对一，否者单方查询，就是单向一对一。

2.一对多关系

？EntityManager 执行查询时，通过一的一方获取多的一方会执行多少条 sql？性能上有什么弊端？通过多的一方获取一的一方会执行多少条 sql？性能上有什么弊端？

？当保存的时候，如何设计级联保存关系，先保持哪一端？会尽量减少 sql 执行的次数？

（请记住这些我们提到过的 EntityManager 执行相关的问题，在第三部分去学习 EntityManager）

3.多对多

可以这么说多对多是一对多关系的一个特例，因为是双方都具有一对多的关系，导致需要第三张表去维系这种双向一对多的关系。

？那么上面这种保存关系必将导致保存上的复杂性。那么当你如何设计这种关系时，才能尽量提高性能，避免 sql 执行的次数？

4.继承

继承可能是对象-关系失配最严重的我问题。因为继承需要不适合（数据库的）关系理论的解决方案。

代码如下：

```
package cn.partner4java.bean.part2_1;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class User {
    @Id
    String userId;
    String username;
    String email;
}

package cn.partner4java.bean.part2_1;
import javax.persistence.Entity;
@Entity
```

```
public class Seller extends User {
    String sellerGroup;
}

package cn.partner4java.bean.part2_1;
import javax.persistence.Entity;
@Entity
public class Bidder extends User {
    String Bidder;
}
```

问题：那么父类有或者没有进行@Entity 标识时，被继承后表结构生成方式如何？在父类是抽象类时有什么区别？

a. 当 User 和 Seller 都有@Entity 注解，三个实体 bean 仅仅生成了一个 user 表，另外两个实体 bean 的字段被包含进了 user。并且额外生成字段 btype。

b. 当 user 去掉注解，如果子类没有主键@Id 会报错误，加上后只生成了子类自己的字段，没有生成 user 的字段，生成两个字类表

c. 当 user 为抽象，没有进行@Entity 声明是生成如 b

d. 当 user 为抽象也为实体生成如 1

（注意：我的测试也许不是完全正常，因为首先的测试环境为 Hibernate3 的实现）

问题总结：这样让我感觉继承没有什么实际的作用。

那么 JPA 给我们提供了专门的注解去实现继承的三种应用场景：

- 每个类分层结构一张表
- 每个子类一张表
- 每个具体类一张表

2. 映射实体

下面我们会给出一个很有意思的例子：也许里面的有些方法你没使用过。

（本第二小节，会根据这个例子详细讲解，如何映射简单的实体—不包含关系的实体）

```
package cn.partner4java.bean.part2_2;
import javax.persistence.Embeddable;
@Embeddable
public class Address {
    private String street;
    private String city;
}
```

```
package cn.partner4java.bean.part2_2;
import java.util.Date;
import javax.persistence.Basic;
……//省略了一些导入
@Entity
@Table(name="USERS")
@SecondaryTable(name="USER_PICTURES",
        pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User {
    @Id
    @Column(name="USER_ID",nullable=false)
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long userId;
    @Column(name="USER_NAME",nullable=false)
    private String username;
    @Column(name="FIRST_NAME",nullable=false,length=1)
    private String firstName;
    @Enumerated(EnumType.ORDINAL)
    @Column(name="USER_TYPE",nullable=false)
    private UserType userType;
    @Column(name="PICTURE",table="USER_PICTURES")
    @Lob
    @Basic(fetch=FetchType.LAZY)
    private byte[] picture;
    @Column(name="CREATE_DATE",nullable=false)
    @Temporal(TemporalType.DATE)
    private Date creationDate;
    @Embedded
    private Address address;
}
```

注解与 O/R 映射中的 XML 相比较

相信看到这里，接触过 Hibernate 的朋友，心里会暗自进行了比较。在注解和 XML 部署描述文件之间做出选择非常困难。XML 描述文件很庞大和难以管理。但是注解是通过硬编码数据库设计方式。幸运的是，你可以使用 XML 描述文件覆盖 ORM 注解。

2.1 指定表

`@Table(name="USERS")`我感觉这个很简单，没什么好说明的，还有 一些不常用的参数就不再啰嗦了，当然，如果你不用Table注解指定表名，也可以这样写：`@Entity(name="USERS")`

2.2 映射列

```
@Column(name="USER_NAME",nullable=false,insertable=false,updatable=false)
```

inserttable 和 updatable 参数用户控制持久化行为，如果 insertable 参数为 false，那么当持久化提供器创建对应实体的新记录时，字段或属性就不会包含在持久化提供其生成的 insert 语句中。Updatable 一个意思。（如果不明白什么意思，记住就可以，后面讲 EntityManager 就知道了）

注意，我比较常犯的错误，就是常常忘记了 length，那么生成的字段就会为最大长度，这样会照成浪费。

2.3 使用@Enumerated

枚举就不用多说了吧，C 和 Pascal 语言里面已经有枚举类型几十年了。

```
@Enumerated(EnumType.ORDINAL)
```
以数字编码形式映射

```
@Enumerated(EnumType.STRING)
```
以字符串形式映射

2.4 映射 CLOB 和 BLOB

二进制大型对象：BLOB

字符大型对象：CLOB

```
@Lob
@Basic(fetch=FetchType.LAZY)
private byte[] picture;
```

生成：LONGBLOB

```
@Lob
@Basic(fetch=FetchType.LAZY)
private String content;
```

生成：LONGTEXT

记住如果是大文本一定要使用@Basic 标注为懒加载，因为大文本是非常耗内存资源的。Mysql 不建议把图片等附件保存在数据库里面，会严重影响数据库的性能。(不幸运的是，EJB3 规范把 LOB 类型的延迟加载作为选项留给厂商完成，因此不能确保确实进行了延迟加载)

2.5 把实体映射到多个表

在一些非常少见的情况中，这是非常有意义的策略。

2.6 生成主键

标识为主键时，在本质上要求数据库强制实现唯一性。

1》 身份列作为生成器

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```
，如果我没记错的话，

mysql 和 mssql 都是用这种生产方式，也就是自增长

在实体数据保持到数据库之前，身份字段的值可以不可用，通常在提交记录时才生成它。

```
Hibernate: insert into USERS (city, street, content, CREATE_DATE,
FIRST_NAME, USER_TYPE, USER_NAME) values (?, ?, ?, ?, ?, ?, ?)
```

看到了实现产品也不是那么傻，id 它不会插入的。

2》 数据库序列作为生成器

```
@GeneratedValue(strategy=GenerationType.SEQUENCE)
```

，如 oracle，需要定义一个序列生成主键。严格的说你还要告诉它序列生成器的名字，如：

```
@GeneratedValue(strategy=GenerationType.SEQUENCE,generator="USER_SEQUENCE_GENERATOR")
```

，如果你不给它生成器，当然我们一般的产品也不会那么傻，它会默认给你创建一个。我印象是这样的，至于名字是什么格式的，我现在也无法测试，因为本机没有装 oracle。

如果你 mysql 非这么设置，会报 not support sequences。

3.序列表作为生成器

这种方式我印象中没有见这么用的，大体含义就是搞出来一个表专门负责生产主键和 oracle 的序列生成器差不多的含义，不过这里是单独创建一个表。

```
@GeneratedValue(strategy=GenerationType.TABLE,generator="USER_GENERATOR_TABLE")
```

4.默认主键生成策略

讲过了上面主键生成策略，当然你如果想，假如数据库可能变更，如果通过这种硬编译的方式指定策略，往往会带来附加的工作。那么很庆幸，我们使用的产品并不会那么傻，hibernate 和 openjpa 都会很好的支持默认自动生成策略。不过请记住，当你使用 TopLink 时，很抱歉，它会默认都是用表生成器。（我都是使用 Hibernate，所以一般都不设置，呵呵）

2.7 映射可嵌入对象

可嵌入对象没有自己的身份，与包含它的实体共享身份。EJB3 不允许把可嵌入的对象映射到与包含它的实体不通过的表中。如果你有特殊需求，需要映射到不同表中，就可以使用上面 2.5 的生成策略：

```
@SecondaryTable(name="USER_PICTURES",pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
```

```
@Column(name="PICTURE",table="USER_PICTURES")
```

3. 映射实体关系

（我们上面说了不包含域对象关系模式的实体，现在我们就详细讲解一下最开始，我们说过的域对象—一对一、一对多、多对多关系是如何映射成实体的 ----我很多地方都说了此类的一些废话，意图就在于，很多朋友经常不知道目前自己到底在学习的具体哪部分知识，很不便于整理思路）

3.1 映射一对一关系

```
@Entity
public class BillingInfo {
    @Id
    private Long billingId;
    private String bankName;
//  @OneToOne(mappedBy="billingInfo",optional=false)
//  private User user;
}


@Entity
public class User {
    @Id
    private String userId;
    private String email;
    @OneToOne(optional=false)
    private BillingInfo billingInfo;
}
```

去掉注解为双向引用，这一点我们上面已经介绍过了，在被维护端，指定关系维护端的维护字段。（维护端：生成维护字段）

？如果你想改变维护字段的表字段生成名字该如何做？

```
@JoinColumn(name="BillingInfo_Id",referencedColumnName="billingId",updatable=false)
```

Name：生成字段的名字

referencedColumnName、updable 等都省略就可以

（你有没有发现，我上面的两端都设置为不允许为空，你要是没发现就大头了）

？有时我被维护端不想生存主键，而是引用维护端的主键。用：

```
@PrimaryKeyJoinColumn(name="userId",referencedColumnName="userId")
```

不过这种做法，我搞了很久没搞明白到底如何进行保存，所以我也就没资格多说了。

3.2 一对多和多对一

只要记住多的一端为关系的维护端，在一的一端的标记 mappedBy，在多的一端标记 @JoinColumn(name="")，因为一的一端生成相应的维护字段。

3.3 多对多

多对多是特殊的多对一，根绝业务需求在一端定义被维护

@ManyToMany(mappedBy="items")，在维护端定义中间表生成方式 @JoinTable(name="CA_IT",joinColumns=@JoinColumn(name="categoryId"),inverseJoinColumns=@JoinColumn(name="itemId"))，当然@JoinTable可以省略。

？当主键对应字段名字使用 @Column(name="category_Id") 进行了类似更改，@JoinTable 里的@JoinColumn 是否必须进行 referencedColumnName="category_Id", 标识呢？ 答案是，不是必须的，因为既然是主键，并且是唯一主键，我们的框架就不会那么傻。当然以防万一最好写上。如果你指定了，并且指定错了，那么框架不会聪明到如此程度给你自动更正。

4.映射继承

三种策略：

A. 单表：就是第一部分我们讲解的嵌入对象的使用方式，我实在没感觉对表结构生产上多么明智，仅是实体 bean 定义时，更对象化

B. 连接表：比单表查询性能还要差劲

C. 每个类一个表：性能更差劲

第三部分 使用 EntityManager 操作实体

我们前面介绍了如何创建一个实体，当然创建完之后，我们就需要对实体进行 CURD 操作，EntityManager 就是提供该操作的接口。

本部分你将了解 EntityManager 接口、实体的生命周期、持久化上下文的概念，以及如何获取 EntityManager 的实体。然后，讨论生命周期回调监听器。

1．介绍 EntityManager

1.1 EntityManager 接口

如果你用过 Hibernate，必然知道 Hibernate 的 SessionFactory，EntityManager 作用也基本如此：作为面向对象和（数据库的）关系领域之间的桥梁。它解释实体指定的 O/R 映射并且把实体保存到数据库中。

由于 JPA 实体独立于容器，因为由 EntityManager 而不是由容器管理实体，所以它们不能直接使用容器服务，比如依赖注入、定义方法级别事物和声明式安全、远程功能等等。所以，如果有人称 JPA 为实体“bean”，你可以礼貌的纠正下。当然我就喜欢这么叫

尽管 JPA 不像会话 bean 或 MDB 那样是以容器为中心的，但是实体仍然具有生命周期。

1.2 实体的生命周期

EntityManager 简言之就是对实体进行管理，进行数据库的 CURD 映射，EntityManager 可以

想象是使用它本身的容器--一个大水池对实体进行管理。所以就会有 EntityManager 和实体之间的管理关系，或者说实体与 EntityManager 的大水池之间的存在关系。如：创建的实体是否已经放进 EntityManager 的大水池进行管理等

分为：

1.新建状态：如当实体第一次被创建时，尚未拥有持久化的主键，没有和一个持久化的上下文关联起来。可以理解为这个实体还没有身份，所以无法也没有被 EntityManager 映射成对应的数据库字段。

2.托管状态：只要我们要求 EntityManager 开始管理实体，当然这个实体必须具有身份也就是主键值，EntityManager 就使数据库和实体状态同步。其次，在实体不再受管理之前，EntityManager 确保对实体书库的改动被反映到数据库中。（当然你会问我们怎么样才能要求 EntityManager 开始管理我们创建的实体？如：当你调用 EntityManager 的方法，或者说当你把实体传递给 persist、merge 活 refresh 方法时。这些方法的含义我们后面就会介绍。之所以调用 EntityManager 的这些方法会被管理，一是因为 EntityManager 主动给实体赋予了身份，或者实体本身具有身份，二是 EntityManager 主动把实体放进了自己的水池，进行了监控）

3.游离状态：和新建状态最大的区别就是，游离状态的实体有身份，只是因为一些原因被拿出了 EntityManager 的水池，或者还没有放进 EntityManager 的水池。自然而然的，游离状态的实体就无法和数据库进行同步。（如：调用 EntityManager 的 clear 方法，就会强制把水池里面的实体都拿出来，变为游离状态）

4.删除状态：最常见的也就是实体同步的数据库字段，被实体主动或被动删除了。实体就没有对应的数据库字段了。但是它还具有身份，并删除前已经放进了 EntityManager 水池。

2 获取 EntityManager

我们既然已经知道了 EntityManager 的作用，那么该如何得到 EntityManager 呢？

2.1 应用程序管理的 EntityManager

1．容器中获取

如在ｊａｖａ　ＥＥ环境中，可以使用@ＰｅｒｓｉｓｔｅｎｃｅＵｎｉｔ注解注解 EntityManager 实例，sping 中我们是`@PersistenceContext protected EntityManager em;` 当然无论是 ejb 还是 sping 或者 J2ee 环境，你都要先事先把 EntityManager 放进这里容器里。

2．容器之外的应用程序获取 EntityManager

也就是我们正常通过类获取的方式：

```
//因为我们目前使用的是hibernate作为JPA的实现产品,EntityManagerFactory背后的实现为sessionFactory,
//当我们创建EntityManagerFactory的时候,背后也会创建sessionFactory,我们学习hibernate知道,当sessionFactory被创建的时候,就会根据映射信息创建数据库表.
EntityManagerFactory factory = Persistence.createEntityManagerFactory("partner4java");//sessionFactory ---获取创建EntityManager的工厂,传入参数为我们配置<persistence-unit name,
EntityManager em = factory.createEntityManager();//session，使用工厂创建EntityManager
em.getTransaction().begin();//开启事务
。。。。CURD操作
em.getTransaction().commit();//提交事物
em.close();
factory.close();
```

3.在 Web 容器和 ThreadLocal 模式中使用 JPA

一些持久化提供器（如 Hibernate）将建议你使用 ThreadLocal 模式。它把 EntityManager 的单一实例和特定请求相关联，必须把 EntityManager 绑定到本地线程变量并且把 EntityManager 实例设置到相关的线程。

```
package cn.partner4java.service.base;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
public class EntityManagerTool {
    private static  EntityManagerFactory entityManagerFactory;
    private static final ThreadLocal<EntityManager> threadLocal = new ThreadLocal<EntityManager>();
    static{
        if(entityManagerFactory == null){
            entityManagerFactory = Persistence.createEntityManagerFactory("partner4java");
        }
    }
    /**
     * 获取当前线程对应的EntityManager，如果没有，就创建一个，并与当前线程对应
     * @return
     */
    public static EntityManager getEntityManager(){
        EntityManager entityManager = threadLocal.get();
        if(entityManager == null){
            entityManager = entityManagerFactory.createEntityManager();
            threadLocal.set(entityManager);
        }
```

```
        return entityManager;
    }
    /**
     * 关闭当前线程对应的EntityManager，并且移除他们之间的对应关系
     */
    public static void closeEntityManager(){
        EntityManager entityManager = threadLocal.get();
        if(entityManager != null){
            entityManager.close();
            threadLocal.remove();
        }
    }
}
```

3.管理持久化操作

3.1 持久化实体 persist

让实体处理托管状态，且保存实体。

（从现在开始的内容会比较繁琐，希望您能够保持清晰的思路，能够随时理清对象间的包含和被包含关系，能够清晰的知道各种状态间是如何转换的）

1.持久化实体关系

例 1.

```
EntityManager em = EntityManagerTool.getEntityManager();
em.getTransaction().begin();//开启事务

Item item = new Item();
item.setTitle("商品名称");
Bid bid = new Bid();
bid.setAmount(2D);
item.getBids().add(bid);

em.persist(item);
em.persist(bid);

em.getTransaction().commit();//提交事物
EntityManagerTool.closeEntityManager();
```

如果你对上面的事物有所生疏，那么请记住，如果你需要对数据库进行更改操作就需要开启事物。

A. 执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
```

查看表数据：我们发现 bid 并没有插入正确的 item_id 而是为空。

B. item.getBids().add(bid);更改为：bid.setItem(item);

执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
```

查看表数据：进行了正确的插入，bid 插入了正确的 item_id。

A 和 B 比较结论：需要把关系告诉维护端而不是让被维护端知道。

C. 更换两个持久化执行顺序，如

```
em.persist(bid);
em.persist(item);
```

执行 sql 如下：

```
Hibernate: insert into Bid (amount, item_id) values (?, ?)
Hibernate: insert into Item (title) values (?)
Hibernate: update Bid set amount=?, item_id=? where bidId=?
```

B 和 C 比较结论：需要先持久化被维护端，因为维护端的持久化需要被维护端的身份主键，如果顺序颠倒，会增加系统开销，多执行一次 sql 语句去更新后持久化的被维护端主键。

D. 更改代码如下：

```
        .....
        Item item = new Item();
        item.setTitle("商品名称");
        Bid bid = new Bid();
        bid.setAmount(2D);
//      item.getBids().add(bid);
        bid.setItem(item);
        em.persist(item);
        em.persist(bid);
        item.setTitle("更改标题");

        .....
```

持久化实体后加入语句：item.setTitle("更改标题");

执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
Hibernate: update Item set title=? where id=?
```

D 结论：当调用 persist 对实体进行持久化后，就会进入托管状态，实体的一切操作都会在

事务关闭时或事务提交时或调用 em.flush()或托管状态丢失之前，同步到数据库

E. item.setTitle("更改标题")前加入：em.clear();

执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
```

E 结论:当调用 clear 方法后,所有 EntityManager 池里面没托管的实体都会变为游离状态，实体的更改将不会反应到数据库中。

2.级联持久化操作

我们现在将学习一种全新的知识，级联，我感觉也是 JPA 的难点，并且也是重点，级联搞的比较清晰，有利于我们的系统优化。

域对象中存在着对应关系的对象，我们会想到，为什么对一个对象进行 CURD 操作时，不自动 CURD 我们声明的具有对应关系的对象呢？

那么先看一个例子（为多对一和一对多的例题）：

实体定义如下：

```
@Entity
public class Item {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    @OneToMany(mappedBy="item",cascade=CascadeType.PERSIST)
    private Set<Bid> bids = new HashSet<Bid>();

。。。。。。
@Entity
public class Bid {
    @Id
    @GeneratedValue
    private Long bidId;
    private Double amount;
    @ManyToOne
//  @JoinColumn(name="")
    private Item item;

。。。。。
```

操作：

```
A.      EntityManager em = EntityManagerTool.getEntityManager();
```

```
        em.getTransaction().begin();//开启事务

        Item item = new Item();
        item.setTitle("商品名称");
        Bid bid = new Bid();
        bid.setAmount(2D);
        item.getBids().add(bid);
        em.persist(item);

        em.getTransaction().commit();//提交事物
        EntityManagerTool.closeEntityManager();
```

您先仔细的看一下上面我们书写的执行代码，您感觉执行的 sql 会是怎样的呢？我们只保存了 item。

执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
```

执行结果：bid 没有报错正确的 item_id，而是为空

B. 持久化语句前加入代码：bid.setItem(item);

执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
```

执行结果：插入了正确的数据

C. 实体定义更改入下：

```
@Entity
public class Item {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    @OneToMany(mappedBy="item")
    private Set<Bid> bids = new HashSet<Bid>();
@Entity
public class Bid {
    @Id
    @GeneratedValue
    private Long bidId;
    private Double amount;
    @ManyToOne(cascade=CascadeType.PERSIST)
```

```
//  @JoinColumn(name="")
    private Item item;
```

代码更改为：

```
        Item item = new Item();
        item.setTitle("商品名称");
        Bid bid = new Bid();
        bid.setAmount(2D);
        bid.setItem(item);
        em.persist(bid);
```

执行 sql 如下：

```
Hibernate: insert into Item (title) values (?)
Hibernate: insert into Bid (amount, item_id) values (?, ?)
```

执行结果：插入正确的 sql 语句

cascade=CascadeType.*PERSIST 为级联持久化含义*

在我们的对应关系上加入对应的级联关系，就会在执行 CURD 操作时，根据级联等级进行相应的级联操作。默认情况下级联为空。级联类型为枚举，具体含义与执行 EntityManager 操作方法向对应。（自己看看就会明白）

各种级联类型值的作用

cascade=CascadeType.*PERSIST*	*EntityManager.persist 持久化-保存*
cascade=CascadeType.*MERGE*	*EntityManager.merge 更新*
cascade=CascadeType.*REFRESH*	*EntityManager.fefresh 刷新*
cascade=CascadeType.*REMOVE*	*EntityManager.remove 删除*
cascade=CascadeType.ALL	所有级联类型

A 和 B 和 C 总结：当设计级联保存时：无论级联方为维护端或为被维护端，一定要让维护端知道需要维护的实体；且只有级联设置端的持久化操作才会具有级联作用；当级联端为被维护端，被维护端还需知道维护端的实体。这种 EntityManager 才会智能的进行保存，且在最佳保存结果。（其实道理很简单，想级联，当然要知道级联谁，并且维护端肯定要知道被维护端是谁）

？您如果想问级联端到底设置为哪一段比较适合？

这就要看你的业务需求了，如上面这个例子：一个为商品、一个为商品的竞价，1.保存商品的时候不会同时保持商品的竞价，因为竞价是后来产生的 2.更改商品的时候，竞价也不大可能同时更改 3.删除商品的时候，竞价应该随同删除，因为商品都没了，竞价也就没有意义了 4.保存竞价的时候，商品肯定已经存在了，所以也不需要 5.更改竞价，商品也不可能会有什

么更改 6.删除竞价，你如果也随之删除对应的商品那你就等着被开除吧

当然我们本小节的重点在与持久化—保存，后面我们再讨论删除和更改等

我们上面看了多对一和一对多的情况，我们下面再分别讨论一对一和多对多的级联保存：

一对一例题：

A．实体定义如下：

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long userId;
    private String email;
    @OneToOne(cascade=CascadeType.PERSIST)
//
    @PrimaryKeyJoinColumn(name="userId",referencedColumnName="userId"
)
    private BillingInfo billingInfo;
    private String userName;
@Entity
public class BillingInfo {
    @Id
    @GeneratedValue
    private Long billingId;
    private String bankName;
    @OneToOne(mappedBy="billingInfo")
    private User user;
```

执行代码：

```
    EntityManager em = EntityManagerTool.getEntityManager();
        em.getTransaction().begin();//开启事务

        User user = new User();
        user.setUserName("王昌龙");
        BillingInfo billingInfo = new BillingInfo();
        billingInfo.setBankName("光大银行");
        user.setBillingInfo(billingInfo);
        em.persist(user);

        em.getTransaction().commit();//提交事物
        EntityManagerTool.closeEntityManager();
```

执行 sql：

```
Hibernate: insert into BillingInfo (bankName) values (?)
Hibernate: insert into User (billingInfo_billingId, email, userName)
values (?, ?, ?)
```

执行结果：正确插入

B．更改实体定义级联端为：BillingInfo

```
@OneToOne(cascade=CascadeType.PERSIST,mappedBy="billingInfo")
    private User user;
```

代码更改：

```
        User user = new User();
        user.setUserName("王昌龙");
        BillingInfo billingInfo = new BillingInfo();
        billingInfo.setBankName("光大银行");
//      user.setBillingInfo(billingInfo);
        billingInfo.setUser(user);
        em.persist(billingInfo);
```

执行 sql 如上：

执行结果：User 没有插入正确的 billingInfo_id

C．去掉 B 执行代码的注释语句

执行 sql 如 A

执行结果：正确插入

A 和 B 和 C 总结：和一对多的总结相同：一：无论级联方为哪方，必须让维护端知道被维护端的实体，二：如果级联端为被维护端，被维护端应该知道级联的维护端实体

？如果你够仔细，并且一直保持在清晰思路的话，你会问这么一个问题：

如果上面的 C 我们持久化的是 billingInfo，具有级联关系的 user，是否也出于托管状态了呢？

我们在持久化后加入：user.setUserName("龙哥");

执行 sql 如下：

```
Hibernate: insert into BillingInfo (bankName) values (?)
Hibernate: insert into User (billingInfo_billingId, email, userName)
values (?, ?, ?)
Hibernate: update User set billingInfo_billingId=?, email=?, userName=?
where userId=?
```

本着负责的态度，我也测试了多对多，和上面的结论一样。

3.2 通过主键检索实体 find

JPA 支持若干从数据库检索实体实例的方式，最简单的方式是使用 find 方法，同过其主键检索实体。

```
em.find(CateGory.class, 1);
```

第一个参数：实体的 java 类型

第二个参数：检索实体的身份值，主键值

Find 方法完全支持符合主键

1. 实体获取模式

   EntityManager 通常在从数据库中检索实体时加载所投实体实例数据，用 ORM 的话来说，就是预先获取或预先加载。如果你曾经处理过因过早或不当缓存照成的应用程序问题，你可能已经知道了总预先加不是好事情。如果 BLOB 字段。

   我们就可以延时加载这些数据，在第一个调用 get 方式时才进行加载，叫延时加载。

```
@Column(name="PICTURE")
 @Lob
 @Basic(fetch=FetchType.LAZY) //指定为延时加载
 private byte[] picture;
```

当然，在你调用 find 方法获取实体后，如果这个获取在事务开启中，获取到的实体就会处于托管状态。就会对数据库进行同步。

2.加载相关实体，

我们做一个测试：

实体定义如：

```
@Entity
public class BillingInfo {
    @Id
    @GeneratedValue
    private Long billingId;
    private String bankName;
    @OneToOne(mappedBy="billingInfo")
    private User user;
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long userId;
    private String email;
```

```
    @OneToOne(cascade=CascadeType.PERSIST)
//
    @PrimaryKeyJoinColumn(name="userId",referencedColumnName="userId"
)
    private BillingInfo billingInfo;
    private String userName;
```

代码：

```
User user = em.find(User.class, 1L);
```

执行 sql：

```
Hibernate: select user0_.userId as userId1_1_,
user0_.billingInfo_billingId as billingI4_1_1_, user0_.email as
email1_1_, user0_.userName as userName1_1_, billinginf1_.billingId as
billingId0_0_, billinginf1_.bankName as bankName0_0_ from User user0_
left outer join BillingInfo billinginf1_ on
user0_.billingInfo_billingId=billinginf1_.billingId where
user0_.userId=?
Hibernate: select user0_.userId as userId1_1_,
user0_.billingInfo_billingId as billingI4_1_1_, user0_.email as
email1_1_, user0_.userName as userName1_1_, billinginf1_.billingId as
billingId0_0_, billinginf1_.bankName as bankName0_0_ from User user0_
left outer join BillingInfo billinginf1_ on
user0_.billingInfo_billingId=billinginf1_.billingId where
user0_.billingInfo_billingId=?
```

此例显示，当你查询实体时，具有对应关系的实体，也一些查询了出来。并不是所有的对应关系都会同时查询出来，讨论如下

3．延迟加载与相关实体的预先加载

默认设置

关系类型	默认获取行为	检索到实体的数量
一对一	EAGER	检索到单一的实体
一对多	LAZY	检索到实体集合
多对一	EAGER	检索到单一实体
多对多	LAZY	检索到实体集合

关系的默认设置并不适用于所有环境。

如果实体包含大量一对一和多对一关系，那么预先加载所有关系就会导致数量巨大的链接 JOIN。执行数量相对大的 JOIN 可能和加载（N1+N2+ ••• +Nx）结果结合的性能一样差。

也不应该认为@ManyToMany 和@OneToMany 注解理应使用延迟加载策略。对于特别大的数

据集合而言，这样会导致生产数量巨大的 SELECT 语句对数据库进行操作。这被称为 N+1 问题，其中 1 代表用于初始实体的 SELECT 语句，N 代表检索每个相关的实体的 SELECT 语句。不幸的是，你不能完全信任这种默认的方式和指定的方式，因为数据库厂商和 JPA 实现厂商往往会有些不同的表现。

不过，当你彻底征服你使用的实现产品对加载问题的实现，你能很好的优化你的系统。

注意：有些时候懒加载会给你留下一些 BUG，因为当你试图在实体已经脱离事物的时候再去获取相应的关系对象，就会出现懒加载错误。

3.3 更新实体 merge

如果你的实体是处理托管状态，就不必考虑手工条用任何方法去更新实体。这可能是基于 ORM 的持久化最为优雅的特性。

1. 更新操作

   虽然托管状态即为有用，但是问题在于难以时刻维持实体的附属状态，通常的问题是需要在 EntityManager 范围之外的 Web 层分离和串行化实体，在这里修改实体。此外，无状态会话的 bean 不能确保同意客户端发出的调用都由相同的 bean 实例处理。

   有时我们希望把实体附属到持久化上下文，以便于使之与数据库同步。方式 merge 即可实现。不过需要注意的是，这个实体必须要具有身份，不容置疑，如果没有身份将无法对应到数据库中。

2. 更新关系

   默认情况下，与被合并的实体相关的实体不会被更新。Cascade 元素控制这一行为，如果此元素被设置为 ALL 或 MERGE。

3.4 删除实体 remove

代码：

```
EntityManager em = EntityManagerTool.getEntityManager();
em.getTransaction().begin();//开启事务

em.remove(em.getReference(User.class, 1L));

em.getTransaction().commit();//提交事物
EntityManagerTool.closeEntityManager();
```

我们上面使用了 getReference 获取实体。

getReference 方法和 find 方法类似，不同的是如果换成中没有指定的实体，EntityManager 会创建一个新的实体，但是不会立即访问数据库加载持久化状态，而是在第

一次访问某个持久属性时才家族相应的持久状态。（也就是说只会对应数据库字段，但是不会进行映射封装，只在第一次使用的时候才封装映射）这样应用在这里会提高执行性能，因为省略了封装映射的步骤。

还有一点不为人知的秘密：

```
EntityManager em = EntityManagerTool.getEntityManager();
em.getTransaction().begin();//开启事务

User user = em.getReference(User.class, 5L);
em.clear();
em.remove(user);

em.getTransaction().commit();//提交事物
EntityManagerTool.closeEntityManager();
```

如果这里是 em.find，不容置疑会报 java.lang.IllegalArgumentException 错误

因为把分离的实体传递给 remove 方法，就会抛出 IllegalArgumentException 异常。但是如果你使用的是 getReference，就会执行通过（至少在 Hibernate 的产品中是这样的）

1. 级联删除操作

   与级联保存一样，为了删除与传递给 remove 实体相关的实体，必须把关系注解中的 cascade 元素设置为 ALL 或者 REMOVE。但是设置级联删除一定要谨慎，特别在多对多关系中，因为被删除的实体可能与未知的其他实体有关，并且会照成一连串的级联删除，照成严重的后果。

级联删除提高篇：

A. 实体 bean 定义如：

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long userId;
    private String email;
    @OneToOne(cascade=CascadeType.ALL)
    private BillingInfo billingInfo;
    private String userName;
@Entity
public class BillingInfo {
    @Id
    @GeneratedValue
```

```
    private Long billingId;
    private String bankName;
    @OneToOne(mappedBy="billingInfo")//,cascade=CascadeType.ALL)
    private User user;
```

代码如：

```
em.remove(em.getReference(User.class, 11L));
```

执行 sql：

```
Hibernate: select user0_.userId as userId1_1_,
user0_.billingInfo_billingId as billingI4_1_1_, user0_.email as
email1_1_, user0_.userName as userName1_1_, billinginf1_.billingId as
billingId0_0_, billinginf1_.bankName as bankName0_0_ from User user0_
left outer join BillingInfo billinginf1_ on
user0_.billingInfo_billingId=billinginf1_.billingId where
user0_.userId=?
Hibernate: select user0_.userId as userId1_1_,
user0_.billingInfo_billingId as billingI4_1_1_, user0_.email as
email1_1_, user0_.userName as userName1_1_, billinginf1_.billingId as
billingId0_0_, billinginf1_.bankName as bankName0_0_ from User user0_
left outer join BillingInfo billinginf1_ on
user0_.billingInfo_billingId=billinginf1_.billingId where
user0_.billingInfo_billingId=?
Hibernate: delete from User where userId=?
Hibernate: delete from BillingInfo where billingId=?
```

B．实体级联删除端改为被维护端

执行代码：

```
em.remove(em.getReference(BillingInfo.class, 13L));
```

执行 sql：

```
Hibernate: select billinginf0_.billingId as billingId0_1_,
billinginf0_.bankName as bankName0_1_, user1_.userId as userId1_0_,
user1_.billingInfo_billingId as billingI4_1_0_, user1_.email as
email1_0_, user1_.userName as userName1_0_ from BillingInfo billinginf0_
left outer join User user1_ on
billinginf0_.billingId=user1_.billingInfo_billingId where
billinginf0_.billingId=?
Hibernate: delete from User where userId=?
Hibernate: delete from BillingInfo where billingId=?
```

对比 A 比 B 多执行了一条 sql 语句：为什么？

当级联设置端为维护端时，且对维护端进行删除，系统会多执行一次查询操作，查询被维护

端的存在问题。当为多对多的时候就不会存在这种问题，你想想为什么。当为一对多的时候呢？这些问题就如我们前面讨论过的级联保存一样，留给你自己去测试吧，并且这种测试是必要的，如果你在乎性能的话。还有一点，不要仅仅局限减少 sql 的执行次数，对表字段总数的预期预测，也会帮助你选择在关系的哪一端减少执行查询等有所帮助

2. 处理关系

比如当你删除一个一对多关系的实体时，你仅仅是想删除当前一个一的维护端字段，但是如果这个一的字段对应多个维护端字段。那么维护端字段就会不乐意了，会报外键约束异常。

这就需要如下我写过的这段代码这样操作：

```
public void delete(Serializable... entityids) {
        for(Serializable id : entityids){
            EntrantUser entrantUser = em.find(EntrantUser.class, id);
            if(entrantUser.getUserOpus() != null &&
entrantUser.getUserOpus().size() > 0){
                for(UserOpus userOpus:entrantUser.getUserOpus()){
                    userOpus.setEntrantUser(null);
                }
            }
            em.remove(entrantUser);
        }
    }
```

3.4 通过 flush 模式控制更新

在某些情况下，不了解 EntityManager flush 模式会是一大损失。Persist、merge 和 remove 这样的方法不会使数据库立即修改，相反，这些操作推迟到转储清除 EntityManager 时才执行。使用这种方式最大的动机在于性能优化，尽可能的批处理 sql，而不是使数据库被一批零散的请求打扰，能够节省大量通信开销。

默认情况下，数据库的 flush 模式为 AUTO，在事务范围内的 EntityManager 事务终止时，以及应用程序管理的或扩展的 EntityManager 关闭时，执行这一操作。

```
em.setFlushMode(FlushModeType.AUTO)
```

```
em.setFlushMode(FlushModeType.COMMIT)
```

如果清除模式为 COMMIT，持久化提供器就只在提交事物时才同步数据库。但是，这样做时应该谨慎，因为在执行查询之前由你负责实体状态和数据库的同步。如果你没有进行同步，导致 EntityManager 查询从数据库返回陈旧的实体，应由程序就会处理不一致的状态。

可以在需要时使用 flush 方法显示转储清除 EntityManager。

```
em.flush()
```

3.6 刷新实体 refresh

如果怀疑当前托管对象存放的并非数据库中最新的数据，可以通过 refresh 方法刷新实体对象，容器会吧数据库中的新值重写进实体对象。如果实体存在关联，并且设置了 CascadeType.REFRESH 或 ALL，刷新一个实体对象，那么与之关联的实体对象也会被刷新，刷新操作不会吧实体对象的状态同步到数据库。

- 如果是新建状态的实体，那么操作会导致 IllegalArgumentException
- 如果是托管状态的实体，那么将会刷新它的持久状态
- 如果是删除状态的实体，那么操作会导致 IllegalArgumentException
- 如果是游离（detached）状态的实体，那么操作会导致 IllegalArgumentException

3.7 分离所有正在托管的实体 clear

在处理大量实体时，如果不把已经处理过的实体从 persistence context 中分离出来，将会消耗大量的内存。或者当你不想因为误操作而更改数据库时调用。调用后所有托管状态实体会变成游离状态。

注意：在事务没有提交前调用 clear 方法，之前对实体所做的任何改变将会丢失，所有有时建议在调用 clear 方法之前先调用 flush。

3.8 检测试题是否处于托管状态 contains

Contains 方法使用一个实体作为参数，如果当前的持久化上下文包含指定的实体对象，返回结果为 true。实体处在与 persistence context 中表明实体出于托管状态。

5 实践建议

使用容器管理的实体管理器。如果构造将部署到 java ee 容器的企业应用程序，强烈建议使用容器管理的实体管理器，此外，如果会话 bean 或者 MDB 层维持实体，则应该使用声明式事务，结合使用容器管理的 EntityManager。总的涞水，这样能使你把精力集中在业务逻辑上，而不必担心管理事物、管理 EntityManager 生命周期等一般细节。

避免把实体管理器注入 web 层。如果从 web 层的组件直接使用 EntityManager API，建议避免注入实体管理器，因为 EntityManager 接口不是线程安全的。相反，应该使用 JNDI 查找获得容器管理的 EntityManager 实例。比较好的做饭是使用 Façade 模式。

使用实体访问对象模式。不要让 EntityManager API 调用导致业务逻辑混乱。

把回调分离成外部监听器。不要让横切事项“污染”域模型，相反，应该使用外部监听器。这样，就可以根绝需要切入和切出监听器。

第四部分 使用查询 API 和 JPQL 检索实体

看完上面我们的状态转化，与简单的 CURD 等操作，那么一些复杂的操作呢，如根据不同条件，不同顺序等等。还有我们既然是对象化操作，那么能不能也把 sql 转换为对象化的操作呢？

1 执行查询

三个步骤：

1》 创建 EntityManager 的实例

2》 创建查询的实例

3》 执行查询

1.1 创建查询的实例

EntityManager 接口提供使用 JPQL 或者原生 SQL 语句创建查询的若干方法。

JPQL 语句，是 JPA 提供的一种对象化 sql 的规则

创建动态查询实体：

可以使用 EntityManager.createQuery 方法创建动态查询，唯一的要求是把合法的 JPQL 语句传递给此方法。

1.2 使用 query 接口

```
Query query = em.createQuery("FROM Category c");
```

基本概念入门：

我上面的书写了一段最简单的 JPQL 语句

1》你会发现缺少了 SELECT c ，我的这种写法在 Hibernate 实现产品中是允许的，但是不是标准的 JPQL 语句，所以为了产品的可替换，一定要加上 SELECT c，标准的写法

3》 为什么还要一个 c ？这是别名的写法，并且我们规定一定需要一个别名，且是 select c，而不是 select *

4》 Category 是类名，或者说是我们实体的名称，并且是具有大小写规范的。如果我们实体声明了别名，如@Entity（name=“Category_Item”），这里是我们声明的别名。如果实体名称和一些 java 类冲突，或者实体具有重名的情况，可以写上全类名予以区分，如：cn.jnol.bean. Category

1. 设置查询参数

SELECT c FROM User c where c.userName = ?1 （c.userName，字段的大小写也是敏感的）

我们在 where 子句中使用了参数？1.指定这个参数有两种方法：按编码活按名称。当在参数中使用整数时为位置参数。

设置参数使用: query.setParameter(1, "wangcl");

当指定参数的特定名称时，叫做命名参数，名称参数提高了代码的可读性。

如：SELECT c FROM User c where c.userName = :userName

设置参数为：query.setParameter("userName", "wangcl");

当然还有第三种默认不书写的方式：

SELECT c FROM User c where c.userName = ?

设置参数以 1 开始顺序编号：

query.setParameter(1, "龙哥");

2. 检索单一实体

```
User user = (User) query.getSingleResult();
```

如果不存在会报错误：javax.persistence.NoResultException: No entity found for query

并且是肯定会报这个错误，无论你有无类型转换 Object user = query.getSingleResult();

3.检索实体集合

```
List<User> users = query.getResultList();
```

如果没有检索到任何查询的结构，它返回空列表，不会抛出异常。

4.在结构中分页

query.setFirstResult(startPosition); Set the position of the first result to retrieve. startPosition - the start position of the first result, numbered from 0 IllegalArgumentException - if argument is negative

query.setMaxResults(maxResult)；Set the maximum number of results to retrieve.检索到的实体的最大数量

这无疑也是一个很好的跨数据库的封装。最起码我们不用自己考虑是 mysql 分页形式还是 oracle 分页形式

5.控制查询的清除模式

查询的结构可能会因清除模式的不同而不同：

默认的 flush 模式是 query.setFlushMode(FlushModeType.AUTO)，代表 flush 是在查询执行前或事务提交时才发生。“查询语句”指的是通过 javax.persistence.Query 对象执行的查询，二并非 find 或 getreference 进行的查询。Flush 的另一种刷新模式 query.setFlushMode(FlushModeType.COMMIT) 它代表 flush 只有在事务提交时才发生，如果是容器管理事物，事务提交的时间发送在方法执行结束时。可以通过 em.flush 手工刷新。

2.深入介绍 JPQL

JPQL 在 java 空间内对类和对戏那个(实体)进行操作。Sql 在数据库空间内对表、列和行进行操作。对我们来说，虽然 JPQL 和 SQL 看上去类似，但是它们是在两个区别很大的领域中工作。

使用 FROM 子句

FROM Category c

Category 是我们要查询的域，其中我们制定 c 为 Category 的标示符。

如果定义实体时制定了名字，就必须使域与之相同。

Select distinct c from Category c where c.items is not empty

可以使用单一路径表达式进一步定位到其他持久化字段或关系字段。如：多对一

不能通过集合 – 值 路径表达式访问持久化或关系字段

Is empty 子句通常用于关系字段的集合-值路径表达式，因此生产的 SQL 语句将判断子查询中关系的 JOIN 操作是否为检索到了任何记录。

JPQL 函数

concat(String1 , String2)返回两个字符串活字面量连接后的值

substring(string, position ,length)返回从 position 开始长度为 length 的子字符串

trim(string)

lower(String) 返回转换为小写的字符串

upper(String) 返回转换为大写的字符串

length(String)返回字符串的长度

locate(searchString, stringToBeSearched) 返回给定字符串在另一个字符串中的位置

JPQL 时间函数

CURRENT_DATE 当前日期

CURRENT_TIME 当前时间

CURRENT_TIMESTAMP 当时时间戳

JPQL 聚合函数

AVG 返回其应用的字段的所有值的平均值 Double

COUNT 返回出阿仙奴返回的结果数量 Long

MAX 返回其应用的字段的最大值 取决于持久化字段的类型

MIN 返回最小值

SUM 返回所有值的和 可能返回 Long 或 Double

```
Select c.user ,COUNT(c.categoryId)
From Category c
Where c.createDate is between :data1 and :data2
Group by c.user
Having count(c.category) > 5
```

3 调用存储过程

最简单的，有一个传入参数，无返回值的调用：（我用的 mysql）

书写的存储过程：

```
CREATE DEFINER = 'root'@'localhost' PROCEDURE `AddUser`(IN username vARCHAR(50))
    NOT DETERMINISTIC
    CONTAINS SQL
    SQL SECURITY DEFINER
    COMMENT ''
BEGIN
```

```
    insert into user (`userName`) values (username);
END;
```

执行代码：

```
        EntityManager em = EntityManagerTool.getEntityManager();
        em.getTransaction().begin();//开启事务

        Query query = em.createNativeQuery("{call AddUser(?)}");
        query.setParameter(1, "王昌龙很伤心");
        query.executeUpdate();

        em.getTransaction().commit();//提交事物
        EntityManagerTool.closeEntityManager();
```

调用返回全部列的存储过程：

```
CREATE DEFINER = 'root'@'localhost' PROCEDURE `GetUserList`()
    NOT DETERMINISTIC
    CONTAINS SQL
    SQL SECURITY DEFINER
    COMMENT ''
BEGIN
     select * from user;
END;
```

```
        EntityManager em = EntityManagerTool.getEntityManager();
        em.getTransaction().begin();//开启事务

        Query query = em.createNativeQuery("{call
GetUserList()}",User.class);
        List<User> users = query.getResultList();
        for(User user:users){
            System.out.println(user.getUserName());
        }

        em.getTransaction().commit();//提交事物
        EntityManagerTool.closeEntityManager();
```

调用返回部分列的存储过程：

```
CREATE DEFINER = 'root'@'localhost' PROCEDURE `GetUserNameList`()
    NOT DETERMINISTIC
```

```
    CONTAINS SQL
    SQL SECURITY DEFINER
    COMMENT ''
BEGIN
     select userId,username from user;
END;
```

```
        EntityManager em = EntityManagerTool.getEntityManager();
        em.getTransaction().begin();//开启事务

        Query query = em.createNativeQuery("{call GetUserNameList()}");
        List<Object[]> users = query.getResultList();
        for(Object[] user:users){
            System.out.println(user[0] + ":" + user[1]);
        }

        em.getTransaction().commit();//提交事物
        EntityManagerTool.closeEntityManager();
```

注解	标注位置	含义	配合使用注解
@Entity	类	标注 class 为实体 bean	
@Id	字段、属性	实体 bean 的身份表示	
@IdClass	类	实体 bean 的复合主键表示	@Id
@EmbeddedId	字段、属性	标注在复合主键对象上	@Embeddable
@Embeddable	类	声明为嵌入对象	
@Embedded	字段、属性	标注在引入对象上，表示为引入对象	@Embeddable
@Table(name="U	类	为实体 bean 指定对应的生成的表	

SERS")		名，默认为类名	
@SecondaryTable(name="USER_PICTURES",pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))	类	使用 USER_ID 主键和 USER_PICTURES 表连接。USER_PICTURES 表内容为本实体 bean 的一个图片字段，进行了表分离。 这里如果设置懒加载比较特别。	@Column(name="PICTURE",table="USER_PICTURES") @Basic(fetch=FetchType.*LAZY*)
@Column(name="FIRST_NAME",nullable=**false**,length=1)	字段、属性	声明生成字段的名字，是否为空，长度。默认长度为最大长度。	
@Enumerated(EnumType.*ORDINAL*)	字段、属性	为枚举类型的字段声明对应表字段的类型	
@Lob	字段、属性	图片等，byte 数组类型	
@Temporal(TemporalType.*DATE*)	字段、属性	为 date 类型字段声明时间类型	
@GeneratedValue(strategy=GenerationType.*AUTO*)	字段、属性	设置自增长主索引。注意有写产品如果设置为自动，活出现你不想要的结果	@Id
@Transsient	字段、属性	顺时字段，即不被映射为字段	
@JoinTable(name="CA_IT",joinColumns=@JoinColumn(name="categoryId",referencedColumnName="category_Id"),inverseJoi	字段、属性	多对多维护端中间表定义	

nColumns=@JoinColumn(name="itemId"))			