

1 第一个 JPA 实例

Java 的面向对象模型与关系数据库模型之间有相当程度的不匹配，而一些对象与数据库数据同步、更新，也为常见的 Java 永续储存（Persistence）问题，在过去，Object/Relational Mapping（ORM）有 JBoss Hibernate、Oracle TopLink 等解决方案，而 JPA 为吸收这些方案的经验，所制订出的 Java 永续储存标准。使用 JPA，底层可以使用不同厂商的 ORM 实作，而接口则是 JPA 的标准，若您使用 NetBeans+Glassfish，则预设的底层实作为 TopLink，JBoss 的工具其底层实作为 Hibernate，若您偏好 Hibernate，则可以再参考 Hibernate Annotations、Hibernate EntityManager 内容，了解 Hibernate 如何支持 JPA。

以下先示范如何于非容器环境中使用 JPA，假设您在 demo 数据库中个 T_USER 表格，而您打算写个 User 类别来与之对应：

```
@Entity
@Table(name = "T_USER")
public class User implements Serializable {

    private static final long serialVersionUID = 1157688182550942700L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private Long age;

    // Getters & Setters

    ...
}
```

User 中的每个数据成员对应至 T_USER 中的每个字段，即 id 成员对应至 id 字段，name 成员对应至 name 字段，age 成员对应 age 字段。

为了成为 JPA 的 Entity 类别，您必须使用 @Entity 加以标注，@Table 标示这个 Entity 类别对应的数据表格，若类别名称与表格名称相同，则可以省略，预设会将类别名称对应至同名的表格，Entity 类别必须实作 Serializable 接口。

每个 Entity 类别必须有独一无二的标识属性，并与数据表格的主键相对应，您要使用 @Id 标注在数据成员或 Getter 方法上，@GeneratedValue 让您可以选择主键的产生策略，在这边利用数据库本身的自动产生策略，由底层的数据库来提供。

若成员名称与表格域名一样，则会自动对应，若不同，则可以使用 @Column 来指定域名，例如：

```

@Entity
@Table(name="T_USER")
public class User implements Serializable {
    private static final long serialVersionUID = 1157688182550942700L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name="C_NAME")
    private String name;

    @Column(name="C_AGE")
    private Long age;

    // Getters & Setters
    ...
}

```

其它对 Entity 的一些要求是：

1. 类别必须是 public
2. 不可以是 final 类别，不可以有 final 方法
3. 要有 public 或 protected 的无参构造函数
4. 数据成员不可以是 public
5. 没有 finalize 方法

为了 JPA 必须设定数据库链接与底层实作的一些细节，您要在 META-INF 下编写一个 persistence.xml：

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
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_2_0.xsd">
    <persistence-unit name="hystonePersistenceUnit"
transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>hystoneDS</jta-data-source>
        <mapping-file>orm.xml</mapping-file>
        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5Dialect" />
            <property name="hibernate.hbm2ddl.auto" value="create-drop"
/>

```

```
<property name="hibernate.format_sql" value="true" />
<property name="hibernate.show_sql" value="true" />
</properties>
</persistence-unit>
</persistence>
```

主要就是设定一些数据库 JDBC URL、用户名称、密码等信息，一个 persistence.xml 中可以设定多个 Persistence Unit，每个 Persistence Unit 可当作一个数据库链接设定，<persistence-unit>的 name 名称即作为 Persistence Unit 的识别名称。在这里所使用的是 Hibernate 实现，"hibernate.hbm2ddl.auto"用来设定当 JPA 程序 EntityManagerFactory 建立时，自动删除数据表格并重建新的数据表格，这可用在测试时期，方便您不用亲自作这些重置表格的动作。接着，您要建立 EntityManagerFactory，EntityManagerFactory 内含设定信息，负责管理 EntityManager，而这样的方式所取得的 EntityManager，称之为 Application-Managed EntityManager。

您可以如下编写一个 JPAUtil 类来测试 JPA 程序：

```
public class JPAUtil {

    private static EntityManagerFactory entityManagerFactory;

    static {
        try {
            entityManagerFactory = Persistence
                .createEntityManagerFactory("demo");
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static EntityManagerFactory getEntityManagerFactory() {
        return entityManagerFactory;
    }

    public static void shutdown() {
        getEntityManagerFactory().close();
    }
}
```

JPAUtil 方便您取得 EntityManager 对象，Entity 对象的生命周期、与数据表格的对应、数据库的存取，都与 EntityManager 息息相关，例如您可以撰写以下的程序，取得 EntityManager 进行 User 对象的储存或是查询：

```
public class JPATest {  
    public static void main(String[] args) {  
        User user = new User();  
        user.setName("Justin Lin");  
        user.setAge(new Long(30));  
        save(user);  
        user = findById(new Long(2));  
        System.out.println(user.getName());  
        JPAUtil.shutdown();  
    }  
  
    private static void save(User user) {  
        EntityManager entityManager = JPAUtil.getEntityManagerFactory()  
            .createEntityManager();  
        EntityTransaction etx = entityManager.getTransaction();  
        etx.begin();  
        entityManager.persist(user);  
        etx.commit();  
        entityManager.close();  
    }  
  
    private static User findById(Long id) {  
        EntityManager entityManager = JPAUtil.getEntityManagerFactory()  
            .createEntityManager();  
        EntityTransaction etx = entityManager.getTransaction();  
        etx.begin();  
        User user = entityManager.find(User.class, id);  
        etx.commit();  
        entityManager.close();  
        return user;  
    }  
}
```

取得 `EntityManager` 后，可透过 `getTransaction()` 取得 `EntityTransaction`，`EntityTransaction` 负责管理交易，您可以透过 `EntityManager` 的 `persist()` 方法来储存 `User` 对象，`EntityManager` 会自动将对应的成员储存至对应的数据表格字段，在这边则若透过 `EntityManager` 的 `find()` 方法，指定主键来查找数据并封装为 `User` 对象，基本上所有的 `EntityManager` 操作，要在交易中完成，但 `find()` 可以不用在交易中完成，只不过若不在交易中使用 `find()` 方法，则查找回来的 `Entity` 将立刻不在 `EntityManager` 的管理之中（也就是处于 `Detached` 状态）。若交易过程中发生错误，可以捕捉异常，执行 `EntityTransaction` 的 `rollback()` 方法来撤回交易。

2 使用实体管理器

2.1 实体管理器（EntityManager）介绍

JPA 中要对数据库进行操作前，必须先取得 `EntityManager` 实例，这有点类似 JDBC 在对数据库操作之前，必须先取得 `Connection` 实例，`EntityManager` 是 JPA 操作的基础，它不是设计为线程安全（Thread-safe）。`EntityManager` 实例基本上是从 `EntityManagerFactory` 上调用 `createEntityManager()` 方法来取得。若使用容器管理，则可以使用 `@PersistenceContext` 注入 `EntityManager`，或者可以使用 `@PersistenceUnit` 注入 `EntityManagerFactory`，再用它来建立 `EntityManager`。

`EntityManager` 主要在管理 `Entity` 实例生命周期，透过 `EntityManager`，对 `Entity` 实例的操作，可以对应至数据库进行新增、查找、修改、删除、重清等动作，以下先简介 API 的使用，以 `Application-Managed EntityManager` 为例作说明，必须搭配 `Entity` 生命周期以对 `Entity` 在 `EntityManager` 中的 `Managed`、`Detached`、`Removed` 状态获得更进一步的了解。

要新增资料，可以使用 `EntityManager` 的 `persist()` 方法，这也会让 `Entity` 实例处于 `Managed` 状态，例如：

```
User user = new User();
// 设定 user 相关属性
entityManager.persist(user);
```

若要取得数据表中的数据，使用 `EntityManager` 的 `find()` 方法，指定主键对象与 `Class` 实例来取得对应的数据并封装为对象，查找回的对象会处于 `Managed` 的状态：

```
User user = entityManager.find(User.class, id);
```

若数据库中已有对应数据，则要修改数据有几种方式：

若对象是在 `Managed` 状态，例如查找对象之后，直接更新对象，在交易确认之后，对象的更新就会反应至数据表之中：

```
User user = entityManager.find(User.class, id);
user.setName("pgao");
```

若对象属于生命周期的 Detached 状态, 可以使用 EntityManager 的 merge() 方法将对象转至生命周期的 Managed 状态, 合并对象上之变更:

```
// 若 user 状态有所变动
entityManager.merge(user);
```

也可以先使用 merge() 方法将 Detached 状态的 Entity 实例转至 Managed 的状态, 再更新对象, 在交易确认之后, 对象的更新就会反应至数据表之中:

```
User user = entityManager.merge(user);
user.setName("pgao");
```

若要删除数据表中的数据, 则对象必须是在 Managed 的状态, 例如用 EntityManager 的 find() 方法查找对象, 以查找到的对象配合 remove() 方法来移除, 或是使用 merge() 方法将 Entity 处于 Managed 状态再用 remove() 移除, 移除之后, 对象对应不到数据表格中实际的数据, 处于 Removed 状态:

```
User user = entityManager.find(User.class, id);
entityManager.remove(user);
```

若在加载某个 Entity 实例之后, 而数据表格因另一个操作而发生变动, 可以使用 EntityManager 的 refresh() 方法, 将数据表格的更动加载 Entity 实例中, 若 Entity 先前有了一些更动操作, 则会被覆盖:

```
entityManager.refresh(user);
```

可以使用 EntityManager 的 flush() 方法, 强制 EntityManager 中管理的所有 Entity 对应的数据表格与 Entity 的状态同步:

```
entityManager.flush();
```

EntityManager 的 clear() 方法, 可以将 EntityManager 所管理的 Entity 实例清除, 使 Entity 处于 Detached 状态:

```
entityManager.clear();
```

每个 EntityManager 都与一个 Persistence Context 关联, EntityManager 不直接维护 Entity, 而是将之委托给 Persistence Context, Persistence Context 中会维护一组 Entity 实例, 每个 Entity 在 Persistence Context 为 Managed 状态, Entity 实例会有 Managed、Detached、Removed 状态, 这在 Entity 生命周期 中再作进一步的说明。

2.2 Entity 生命周期

JPA 中的 Entity 对象的状态, 可以分为四种: New、Managed、Detached、Removed。以下使用 Application-Managed EntityManager 为例作说明:

- New

直接使用 new 创建出 Entity 对象，例如在之前的例子中，User 类所建立之实例，在还没有使用 persist() 之前都是 New 状态对象，这些对象还没有与数据库发生任何的关系，不对应于数据库中的任一笔数据，没有主键对映。

- Managed

当对象与数据库中的数据有对应关系，并且与 EntityManager 实例有关联而 EntityManager 实例尚未关闭 (close)，则它是在 Managed 状态，具体而言，如果将 New 状态的对象使用 EntityManager 的 persist() 或 merge() 方法加以储存、合并，或是使用 find() 从数据库加载数据并封装为对象，则该对象为 Managed 状态。

Managed 状态的 Entity 是在 PersistenceContext 的管理之中，Managed 状态的对象对应于数据库中的一笔数据，对象的 id 值与数据的主键值相同，并且 EntityManager 实例尚未失效，在这期间对对象的任何状态变动，在 EntityManager 实例关闭 (close) 或交易确认之后，数据库中对应的数据也会跟着更新。

如果将 EntityManager 实例关闭(close)，则 PersistenceContext 失效，Managed 状态的对象会成为 Detached 状态。

- Detached

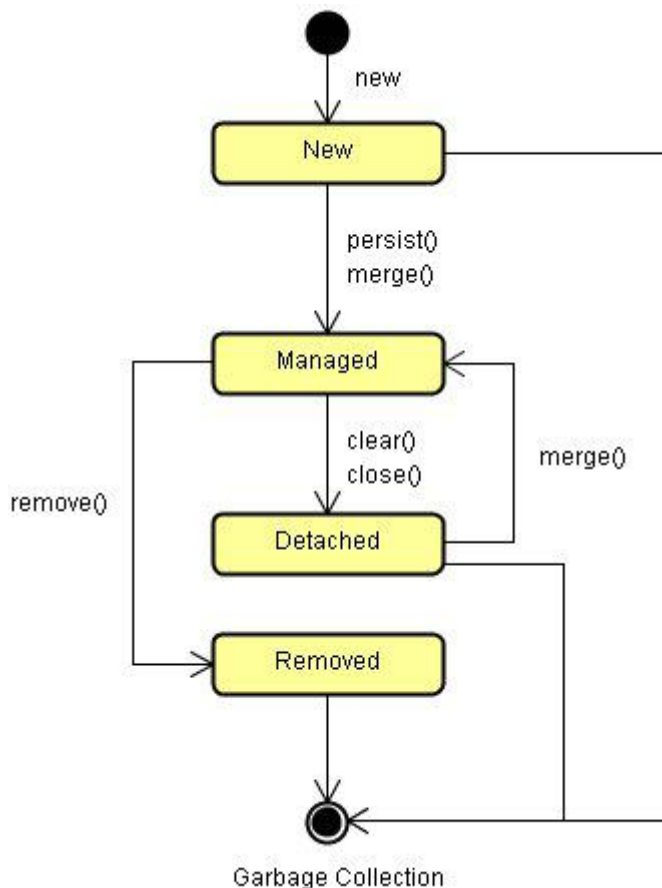
Detached 状态的对象，其 id 与数据库的主键值对应，但脱离 EntityManager 实例的管理，例如在使 find() 方法查询到数据并封装为对象之后，将 EntityManager 实例关闭，则对象由 Managed 状态变为 Detached 状态，Detached 状态的对象之任何属性变动，不会对数据库中的数据造成任何的影响。

Detached 状态的对象可以使用 merge() 方法，使之与数据库中的对应数据再度发生关联，此时 Detached 状态的对象会变为 Managed 状态，也就是被 PersistenceContext 所管理。

- Removed

如果使用 EntityManager 实例的 remove() 方法删除数据，Managed 状态的对象由于失去了对应的数据，则它会成为 Removed 状态，一个成为 Removed 状态的对象不应该被继续重用，应该释放任何引用至它的名称，让该对象在适当的时候被垃圾回收。

简单的说，New 与 Detached 状态的对象未受 EntityManager 管理，也就是不在 PersistenceContext 管理之中，对这两个状态的对象作任何属性变动，不会对数据库中的数据有任何的影响，而 Managed 状态的对象受 EntityManager 管理，也就是在 PersistenceContext 管理之中，对对象的属性变动，在 EntityManager 实例关闭（close）或交易确认之后，数据库中对应的数据也会跟着更新。



2.3 EntityManager 范围

每个 EntityManager 都与一个 Persistence Context 关联，EntityManager 不直接维护 Entity，而是将之委托给 Persistence Context，Persistence Context 中会维护一组 Entity 实例，Entity 实例在 Persistence Context 中为 Managed 状态，这在 Entity 生命周期中有说明。之前有关于使用 EntityManager 以及 Entity 生命周期，都是以 Application-Managed EntityManager 为例作说明，若是在 Container-Managed EntityManager 的情况下，可以设定由容器来为管理 Persistence Context 的范围。

在注入 PersistenceContext 至 EntitySessionBean 时，并没有使用 type 指定 PersistenceContext 的类型，则 Persistence Context 预设为 Transaction-scoped，在 EntitySessionBean 方法开始前会启始交易，结束后停止交易，Persistence Context 的存活范围在交易之间，也就是 Entity 实例在交易完成之后，将会不受

EntityManager 的管理，将不在是 Managed 状态，而处于 Detached 状态。可以在使用 @PersistenceContext 时，指定 type 属性为 PersistenceContextType.EXTENDED（预设 of PersistenceContextType.TRANSACTION），使 Persistence Context 为 Extended Persistence Context，在 EntityManager 实例存活期间，Entity 始终受到 PersistenceContext 的管理，Extended Persistence Context 只能用于 Stateful Session Bean 中，Entity 会一直受 EntityManager 的 PersistenceContext 管理，直到 Stateful Session Bean 结束而 EntityManager 关闭（close）为止。

举个例子来说，当使用预设的 Transaction-scoped Persistence Context，则要更新数据表中的对应数据时，Detached 状态的 Entity 必须先使其回到 Managed 状态，也就是也许会在 Session Bean 中设计像以下的一些方法：

```
public User updateUser(User user) { // 若 user 已被变更
    User user1 = entityManager.merge(user);
    return user1;
}

public User updateUser(User user, String name) {
    User user1 = entityManager.merge(user);
    user1.setName(name);
    return user1;
}

public User deleteUser(User user) {
    User user1 = entityManager.merge(user);
    entityManager.remove(user1);
    return user1;
}
```

可以想象指定 type 属性为 PersistenceContextType.EXTENDED 时，一旦进入 EntityManager 的管理，Entity 一直处于 Managed 的状态，若如此，则以上的程序片段中，有些不再需要：

```
public User updateUser(User user) { // 若 user 已被变更
    User user1 = entityManager.merge(user);
    return user1;
}

+

public User updateUser(User user, String name) {
    User user1 = entityManager.merge(user);
    user.setName(name);
}
```

```

        return user;
    }

    public User deleteUser(User user) {
        User user1 = entityManager.merge(user);
        entityManager.remove(user);
        return user;
    }

```

严格说来，本页标题名称应该叫作 PersistenceContext 范围，因为 type 属性设定的正是 EntityManager 的 Persistence Context 有效范围，不过一般也常称为 Transaction-scoped EntityManager 或 Extended-scoped EntityManager。

若为 Application-Managed EntityManager，在 JPA 中的 Application-Managed EntityManager，则其行为类似于以上说明的 Extended-scoped EntityManager，也就是 Persistence Context 是随着 EntityManager 的关闭而失效，也就是当 EntityManager 关闭后，Entity 就不再为 Managed 状态。

3 进阶配置

3.1 Persistence Context

PersistenceContext 是个由 EntityManager 管理的集合对象，其管理一堆 Entity 实例，Entity 在 PersistenceContext 的 Persistence scope 管理期间为 Managed 状态。

若为 Container-Managed EntityManager，可透过 @PersistenceContext 的 type 来设定 PersistenceContext 的存活范围为以交易为范围，或是为延伸范围，若为 Application-Managed EntityManager，则 PersistenceContext 存活范围在 EntityManager 建立与关闭之间，这在 EntityManager 范围有说明。PersistenceContext 管理的 Entity 为 Managed 状态，基本上此时你对 Entity 实例的任何属性变动，PersistenceContext 都会将之对应至数据库的变动，不过每次在程序中一旦变动 Entity 就作数据库 IO 会有效能议题，因此实际上对 Entity 的变动，不会马上反应出数据库中，而是会等到 EntityManager 作了 flush() 之后，在这之前对 Entity 的变动，会被收集起来，再一次进行数据库的变更。

EntityManager 的 flush() 执行时机可能在：

1. flush() 的 FlushModeType 预设是 AUTO 时：

- 1) 如果是 Transaction-scoped EntityManager，在交易确认时会 flush()。

- 2) 如果是 Extended-scoped EntityManager，或者 Application-Managed EntityManager，则是在 EntityManager 关闭时 flush()。
2. 如果查询某个实体前，该实体有变动，则会先 flush()再进行查询。
3. 主动呼叫 EntityManager 的 flush()方法。

可以使用 EntityManager 的 setFlushMode() 设定 FlushModeType，预设是 FlushModeType.AUTO，可以设定为 FlushModeType.COMMIT，则只有在主动确认一个交易，才会进行 flush()，如果设定 FlushModeType.COMMIT，在查询数据之前若 Entity 有变动，则要主动 flush()，再进行查询，才不致查询到旧数据。

由于 PersistenceContext 的 Persistence Scope 存活期间会管理 Entity，所以在大量储存对象时，PersistenceContext 中管理的 Entity 实例会不断的增加，甚至最后导致 OutOfMemoryError，可以主动每隔一段时间使用 EntityManager 的 flush() 强制同步 Entity 与数据库，再使用 clear() 清除 PersistenceContext 中管理的 Entity 实例。

3.2 Entity 生命周期监听器

先前看过的，Session Bean 与 Message-Driven Bean 有其生命周期与回调方法：

1. Stateless Session Bean 生命周期
2. Stateful Session Bean 生命周期
3. Message-Driven Bean 生命周期

Entity 类似的，也可以设定生命周期回调方法，在其储存、加载、更新、移除等适当的时机被呼叫，你可以在对应的回调方法中进行日志、效能测试、数据验证、通知改变等动作。

下表列出 Entity 相关的生命周期回调方法标注：

@PrePersist	EntityManager 储存 Entity 之前呼叫
@PostPersist	EntityManager 储存 Entity 之后呼叫
@PostLoad	EntityManager 查询 Entity 之后呼叫，像是查询、find()、refresh() 操作
@PreUpdate	EntityManager 将 Entity 与数据库同步更新发生前呼叫
@PostUpdate	EntityManager 将 Entity 与数据库同步更新发生后呼叫

@PreRemove	EntityManager 移除 Entity 前呼叫
@PostRemove	EntityManager 移除 Entity 后呼叫

这些生命周期回调标注，可以直接标注在 Entity 上，例如：

```
@Entity
@Table(name="T_USER")
public class User implements Serializable {
    ...
    @PostPersist
    @PrePersist
    @PostLoad
    @PreUpdate
    @PostUpdate
    @PreRemove
    @PostRemove
    public void monitorUser() {
        System.out.println("your action....XD");
    }
}
```

不过建议可以设计一个监听器，将生命周期回调方法定义其上，例如：

```
public class UserListener {
    @PostPersist
    @PrePersist
    @PostLoad
    @PreUpdate
    @PostUpdate
    @PreRemove
    @PostRemove
    public void monitor(Object o) {
        User user = (User) o;
        System.out.println(user.getName());
    }
}
```

监听器的回调方法接受一个对象作为参数，回调方法呼叫时会传入 Entity 实例，然后定义 Entity 时，可以使用 `@EntityListeners` 来指定监听器：

```
@Entity

@EntityListeners (com.hyland.jpatorial.UserListener.class)

@Table (name="T_USER")

public class User implements Serializable {

    ...

}
```

`@EntityListeners` 中可以定义多个监听器，例如：

```
@EntityListeners (com.hyland.jpatorial.UserListener.class,
com.hyland.jpatorial.SomeListener.class,
com.hyland.jpatorial.OtherListener.class)
```

监听器执行的顺序为定义时的先后顺序，若是在父子类关系中，父类的监听器会先执行，而后是子类监听器。若想要所有 Entity 都使用一个预设监听器，可以定义在 `persistence.xml` 中，例如：

```
<persistence-unit name="sample">
    ...
    <default-entity-listeners>com.hyland.jpatorial.SomeListener.c
lass
    </default-entity-listeners>
    ...
</persistence-unit>
```

在定义监听器时，还可以使用 `@ExcludeDefaultListeners`、`@ExcludeSuperClassListeners` 来排除预设监听器或父类监听器的执行，例如：

```
@Entity

@ExcludeDefaultListeners

@ExcludeSuperClassListeners

@EntityListeners (com.hyland.jpatorial.PowerUserListener.class)

public class PowerUser extends User {

    ...

}
```

3.3 Lock 机制

在多人使用的环境下，每个用户可能进行自己的交易，交易与交易之间，必须互不干扰，使用者不会意识到别的用户正在进行交易，就好像只有自己在进行操作一样。隔离设定与特定的资源相关，并不在 Java EE 规范之中。

乐观锁定 (Optimistic locking) 乐观的认为数据很少发生同时存取的问题，通常在数据库层级上设为 read-committed 隔离层级，并实行乐观锁定。

在 read committed 隔离层级之下，允许交易读取另一个交易已 COMMIT 的数据，但可能有 unrepeatable read 与 lost update 的问题存在，例如：

1. 交易 A 读取字段 1
2. 交易 B 读取字段 1
3. 交易 A 更新字段 1 并 COMMIT
4. 交易 B 更新字段 1 并 COMMIT

交易 B 可能基于旧的数据来更新字段，使得交易 A 的数据遗失，或者是：

1. 交易 A 读取字段 1、2
2. 交易 B 读取字段 1、2
3. 交易 A 更新字段 1、2，字段 1 是新数据，字段 2 是旧数据，并 COMMIT
4. 交易 A 更新字段 1、2，字段 1 是旧数据，字段 2 是新数据，并 COMMIT

为了维护正确的数据，乐观锁定使用应用程序上的逻辑实现版本控制的解决。

对于 lost update 的问题，可以有几种选择：

- 先更新为主 (First commit wins)

交易 A 先 COMMIT，交易 B 在 COMMIT 时会得到错误讯息，表示更新失败，交易 B 必须重新取得数据，尝试进行更新。

- 后更新的为主 (Last commit wins)

交易 A、B 都可以 COMMIT，交易 B 覆盖交易 A 的数据也无所谓。

- 合并冲突更新 (Merge conflicting update)

先更新为主的变化应用，交易 A 先 COMMIT，交易 B 要更新时会得到错误讯息，提示用户检查所有字段，选择性的更新没有冲突的字段。

JPA 中透过版本号检查来实现先更新为主，在数据库中加入一个 version 字段记录，在读取数据时连同版本号一同读取，并在更新数据时比对版本号与数据库中的版本号，如果等于数据库中的版本号则予以更新，并递增版本号，如果小于数据库中的版本号就丢出例外（OptimisticLockingException），版本号可以是数字或时间戳，通常使用数字。

若要定义 Entity 上版本号字段对应的属性，则可以使用 @Version，例如：

```
package com.hyland.jpatorial;

public class User {
    private Long id;

    @Version
    private Long version; // 增加版本属性
    ...
    public Long getVersion() {
        return version;
    }
    public void setVersion(Long version) {
        this.version = version;
    }
}
```

在 EntityManager 上有个 lock() 方法，可以让主动对 Entity 进行锁定，lock() 有两种模式： LockModeType.READ 与 LockModeType.WRITE。前者允许另一使用者读取，但不允许更新、删除，可避免 Dirty read、Non-repeatable read，后者则不允许另一使用者读取、更新、删除。使用 lock() 方法，Entity 上必须有版本属性，且必须在 Managed 状态，否则无法取得锁定，会丢出 javax.persistence.PersistenceException，EntityManager 会尝试将 lock() 转为数据库的锁定指令。

3.4 使用 XML 覆盖源数据（metadata）

在 EJB3 和 JPA 中，使用源数据的注意目标是使用注解（annotation），但是规范提供了覆盖或者代替使用注解定义的源数据的方式，那就是使用 XML 配置描述文件。XML 部署描述文件结构被设计的可以很好的映射注解，如果你对使用注解方式比较熟悉，那么使用 XML 描述文件也是比较容易理解的。

3.4.1 全局性 metadata

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings
  xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
orm_2_0.xsd"
  version="2.0">
  <persistence-unit-metadata>
    <xml-mapping-metadata-complete/>
    <persistence-unit-defaults>
      <schema>myschema</schema>
      <catalog>mycatalog</catalog>
      <cascade-persist/>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
```

xml-mapping-metadata-complete 标签意味着所有的实体，mapped-superclass 和可嵌入的源数据应该从 xml 中获取。

4 Java 持久化查询语言

JPQL 是一种中间查询语言，语法和 SQL 类似，JPA 的实现将 JPQL 转换为数据库的 SQL 查询。

4.1 使用 Query 对象

Java Persistence Query Language 简介 JPQL，是个中间的查询语言规范，根据实际所使用的数据库不同，透过设定，可以将 JPQL 转译为数据库所使用的原生语法，JPQL 提供选取、更新与删除的语法，若要透过 JPQL 来进行选取、更新与删除等操作，则可透过 EntityManager 建立 Query 对象来达成。

若要查询 T_USER 表格中的数据，可以如下：

```
EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

Query query = entityManager.createQuery("SELECT user FROM User user");
Iterator users = query.getResultList().iterator();
while(users.hasNext()) {
    User user = (User) users.next();
    System.out.printf("%d\t%s\t%d\n", user.getId(),
```



```
        user.getName(), user.getAge());  
    }  
    entityManager.close();
```

可以通过 `EntityManager` 的 `createQuery()` 搭配 JPQL 语句来建立 `Query` 对象，在这边是指定为 `select` 语句，若要取得查询结果，则是透过 `Query` 的 `getResultList()` 方法来取回，该方法传回 `List` 对象，当中每个对象皆已将表格的信息封装为 `User` 实例。

4.1.1 位置参数

在建立 JPQL 时，可以在 JPQL 中的参数处使用位置参数 (Positional Parameter)，例如，若要查询表格中用户名称为 `pgao` 的使用者之年龄，可以如下编写代码：

```
EntityManager entityManager =  
    JPAUtil.getEntityManagerFactory().createEntityManager(  
    );  
Query query = entityManager.createQuery(  
    "SELECT user.age FROM User user WHERE user.name = ?1");  
query.setParameter(1, "Justin Lin");  
Long age = (Long) query.getSingleResult();  
System.out.println("age: " + age);  
entityManager.close();
```

注意 位置参数 的写法是 `?` 后加上数字 (1 到 9)，而要指定位置参数的值，则指定位置参数的数字，如范例中的 `setParameter()` 方法，在这边也示范了 `Query` 的 `getSingleResult()` 方法，可用于取得单一个查询结果，而查询时若只想查询某个或某几个属性，则如上例中，直接接上属性名称即可，查询的结果会自动封装为对应的对象。

4.1.2 命名参数

JPQL 也可以用命名参数 (Named Parameter) 作为占位符，例如：

```
EntityManager entityManager =  
    JPAUtil.getEntityManagerFactory().createEntityManager();  
Query query = entityManager.createQuery(  
    "SELECT user.age FROM User user WHERE user.name = :userName");  
query.setParameter("userName", "pgao");  
Long age = (Long) query.getSingleResult();
```

```
System.out.println("age: " + age);

entityManager.close();
```

命名参数作为占位符时，是在“:”后跟随着名称，而使用 `setParameter()` 方法时，则可指定实际的名称。

Query 可以使用 `setMaxResults()` 指定传回的数据最大笔数，使用 `setFirstResult()` 指定数据起始位置。

若要修改或删除数据，则是透过 `executeUpdate()` 方法，例如以下示范数据的更新：

```
EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();
EntityTransaction etx = entityManager.getTransaction();
etx.begin();
Query query = entityManager.createQuery("UPDATE User user SET
    user.age = :userAge WHERE user.name = :userName");
query.setParameter("userAge", 35);
query.setParameter("userName", "Justin Lin");
query.executeUpdate();
etx.commit();
entityManager.close();
```

4.2 Named Query 和 Native Query

可以将一些静态的 JPQL 或 SQL 语句建立在同一个位置，这些静态语句会有几个参数有所不同，但基本上语句结构是相同的，避免将 JPQL 或 SQL 建立在程序中不同的位置，而造成日后修改时必须查看程序代码的麻烦。

在 JPA 中，可以使用 `@NamedQuery` 来建立 Named Query，可以在 User 类上建立 Named Query：

```
@NamedQuery(
    name="QueryUserById",
    query="SELECT user FROM User user WHERE user.id = :userId"
)

@Entity
@Table(name="T_USER")
public class User implements Serializable {
```

```
...
}
```

在建立 Query 对象时，可以使用 NamedQuery 的 name 属性取得 NamedQuery：

```
Query query = entityManager.createNamedQuery("QueryUserById");
query.setParameter("userId", id);
User user = (User) query.getSingleResult();
```

如果有多个 NamedQuery 要宣告，可以使用 @NamedQueries 来宣告，例如：

```
@NamedQueries ({
    @NamedQuery( name="QueryUserById",
        query="SELECT user FROM User user WHERE user.id = :userId"),
    @NamedQuery( name="UpdateUserById",
        query="UPDATE User user SET user.age = :userAge WHERE user.id
        = :userId")
})
@Entity
@Table(name="T_USER")
public class User implements Serializable {
    ...
}
```

可以使用 Query 对象的 createNativeQuery() 方法建立原生查询(Native Query)，也就是直接使用数据库的 SQL 语法来进行查询，对于无法使用 JPQL 查询来取得数据时可以使用，例如：

```
Query query = entityManager.createNativeQuery(
    "SELECT * FROM T_USER", User.class);
Iterator iterator = query.getResultList().iterator();
while(iterator.hasNext()) {
    User user = (User) iterator.next();
    System.out.println(user.getName());
}
```

4.3 JPQL 语法简介

JPQL 所提供的查询语法主要分为三类：

- 查询用的 SELECT 语法

- 更新用的 UPDATE 语法
- 删除用的 DELETE 语法

SELECT 语法结构由几个部份组成：

SELECT 子句 FROM 子句 [WHERE 子句] [GROUP BY 子句] [HAVING 子句] [ORDER BY 子句]

基本的 SELECT 语句如下所示：

```
SELECT u.id, u.name FROM User u WHERE u.age > 10 AND u.age < 20
```

其中 User u 是个路径表示 (path expression)，路径表示有三种：范围变量 (Range variable) 路径表示、集合成员 (Collection member) 路径表示与关联导览 (Association traversing) 表示。User u 是范围变量路径表示的一个例子，指定查询的实体为 User 与别名为 u。

一个集合成员路径表示用来指定对象中的集合成员，例如：

```
SELECT u FROM User u, IN(u.emails) e WHERE e.address LIKE '%.@openhome.cc'
```

其中 IN 中指定的，就是集合成员路径表示，而 >、<、AND、IN、LIKE 等都是 WHERE 子句中条件表示式，简单列出一些条件表示式如下：

比较语句	=、>、>=、<、<=、<>
BETWEEN 语句	[NOT BETWEEN
LIKE 语句	[NOT] LIKE
IN 语句	[NOT] IN
NULL 语句	IS [NOT] NULL
EMPTY 语句	IS [NOT] EMPTY
EXISTS 语句	[NOT] EXISTS

LIKE 中，可以用 _ 表示比对单一字符，用 % 表示比对任意数目字符。

关联导览表示则提供 SQL 语法中 JOIN 的功能，包括了 INNER JOIN、LEFT OUTER JOIN、FETCH 等，以下为 INNER JOIN 的实际例子：

```
SELECT u FROM User u INNER JOIN u.emails e WHERE e.address LIKE  
'%.@openhome.cc'
```

JOIN 关键词可以省略，上式等同于：

```
SELECT u FROM User u JOIN u.emails e WHERE e.address LIKE '%.%@openhome.cc'
```

LEFT OUTER JOIN 的 OUTER 关键词可以省略，一个例子如下：

```
SELECT u FROM User u LEFT JOIN u.emails e WHERE e.address LIKE '%.%@openhome.cc'
```

在作 INNER JOIN、LEFT OUTER JOIN 可以加上 FETCH 关键词，以预先撷取相关数据，例如：

```
SELECT u FROM User u LEFT JOIN FETCH u.emails e WHERE e.address LIKE '%.%@openhome.cc'
```

SELECT 中可以使用聚集函数，例如：

```
SELECT AVG(u.age) FROM User u
```

SELECT 中可以使用建构表示，直接将一些数据封装为指定的对象，例如：

```
SELECT NEW SomeObject(u.id, u.name, o.number) FROM User u JOIN Order o WHERE u.id = 1975
```

WHERE 子句中可以使用 LENGTH()、LOWER()、UPPER()、SUBSTRING() 等 JPQL 函式。

可以对查询结果使用 ORDER BY 进行排序：

```
SELECT u FROM User u ORDER BY u.age
```

ORDER 预设是 ASC 升序排序，可使用 DESC 降序排序：

```
SELECT u FROM User u ORDER BY u.age DESC
```

可同时指定两个以上的排序方式，例如先按照“age”降序排序，如果“age”相同，则按照“name”升序排列：

```
SELECT u FROM User u ORDER BY u.age DESC, u.name
```

可以配合 GROUP BY 子句，自动将指定的字段依相同的内容群组，例如依字段“sex”分组并作平均：

```
SELECT u.sex, AVG(u.age) FROM User u GROUP BY u.sex
```

GROUP BY 通常搭配 HAVING 来使用，例如：

```
SELECT u.sex, avg(u.age) FROM User u GROUP BY u.sex HAVING AVG(u.age) > 20
```

可以使用 UPDATE 语法来更新数据，例如：

```
UPDATE User u SET u.name='momor' WHERE u.name='bbb'
```

可以透过 DELETE 来删除数据，例如：

```
DELETE User u WHERE u.name='bush'
```

更多 JPQL 的介绍，可以引用 [The Java Persistence Query Language](#)。

5 面向对象映射

学习 JPA，大部分时间是在学习其如何实现映射，从中也可以了解到不少数据库表的设计方式。

5.1 实体映射

下面我们用循序渐进的方式来学习实体映射。

5.1.1 基本属性（property）映射

如果没有使用 `@Transient` 标记，实体的每一个非静态，非瞬时（transient）属性都是可以持久化的。在你的属性上未使用标记等同与使用 `@Basic` 标记。另外，`@Basic` 标记允许声明属性的 fetch 策略，如下：

```
@Entity
public class Employee implements Serializable {
    private static final long serialVersionUID = -812123721565782212L;
    @Id
    private int id;
    public transient int counter;           //transient property
    private String firstname;              //persistent property
    @Transient
    String getLengthInMeter() { return ""; } //transient property
    String getName() { return ""; }         // persistent property
    @Basic
    int getLength() { return 0; }           // persistent property
    @Basic(fetch = FetchType.LAZY)
    String getDetailedComment() { return null; } // persistent property
    @Temporal(TemporalType.TIME)
    java.util.Date getDepartureTime() { return null; } // persistent
property
    @Enumerated(EnumType.STRING)
    Starred getNote() { return null; } //enum persisted as String in
database
}

// Getters & Setters
...
}
```

在此例中，瞬时域 count，使用 `@Transient` 标记的方法 `getLengthInMeter()` 所对应的属性 `lengthInMeter`，在持久化是将被实体管理器忽略。`detailComment` 属性是懒加载的；通常情况下，对于简单属性是不需要设置懒加载的。

在普通的 Java API 中，时间精度是未定义的，当处理时间相关的属性的时候，时间数据有 DATE，TIME 和 TIMESTAMP 几种精度。使用@Temporal 标记可以很好的处理这个问题。

5.1.2 可嵌入（Embeddable）对象

假设设计了一个 T_USER 表，当中有 id、name、age 与 email 四个字段，而在 Java 程序的方面，原先设计了一个 User 类：

```
@Entity
@Table(name="T_USER")
public class User implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private int age;
    private String email;
    ...
}
```

现在假设基于业务上的设计需求，需要提高对象设计时的精细度，而将 email 信息封装至一个 Email 对象，让它携带更多信息或负有特定职责，而 User 类变为如下设计：

```
@Entity
@Table(name="T_USER")
public class User implements Serializable {
    private static final long serialVersionUID = 8301770461930996704L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private Long age;
    @Embedded
    private Email email;
    // Getters & Setters
    ...
}
```

其中 Email 封装了邮件地址信息，为另行设计的一个类，必须使用 @Embedded 标示，表示映射是位于 Email 之中：

```
@Embeddable

public class Email implements Serializable {

    private static final long serialVersionUID = -7584205393637684513L;

    private String email;

    public void setEmail() {

        System.out.println("Send email to" + email);

    }

    // Getters & Setters

    ...

}
```

Email 类上，则使用 @Embeddable 标示，注意在 JPA 的规格书中，目前不支持巢状的嵌入类，像 Email 这样的对象称之为值类型（value type）对象，它没有自己的标识（identity），隶属于某个实体对象，其生命周期跟随着所隶属的实体对象，值类型对象通常不会共享。

在储存时，只要 User 设定好 Email，储存 User 对象时，会自动将 Email 中的属性储存至对应的字段：

```
Email mailAddress = new Email();

mailAddress.setEmail("pgao.onlyfun@gmail.com");

User user = new User();

user.setName("pgao");

user.setAge(new Long(30));

user.setMailAddress(mailAddress);

EntityManager entityManager =

JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();

etx.begin();

entityManager.persist(user);

etx.commit();

entityManager.close();
```

而加载 User 时，也会自动将字段设定至 Email 中对应的属性：

```
EntityManager entityManager =

    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();

etx.begin();

User user = entityManager.find(User.class, id);
```



```

etx.commit();

entityManager.close();

user.getEmailAddress().sendMail();

```

@Embedded 可以搭配 @AttributeOverride 或 @AttributeOverrides 一同使用，以重新定义 @Embeddable 类中的映射，例如：

```

@Embedded

@AttributeOverride(name="email",
column=@Column(name="CUST_STREET"))

private Email Email;

...

```

5.1.3 主键映射

JPA 定义了 5 种类型的标识（ID）生成策略。

1. AUTO 依赖数据库的实现，选择 TABLE，IDENTITY 或者 SEQUENCE 方式
2. TABLE 使用专门表存储 ID
3. IDENTITY ID 列
4. SEQUENCE 序列
5. Identity copy ID 是从其他实体 copy 的

另外，不同的 JPA 实现，可能提供了其他的主键生成策略。

使用序列生成器的例子如下：

```

@Entity
@javax.persistence.SequenceGenerator(
    name="SEQ_STORE",
    sequenceName="my_sequence"
)

public class Store implements Serializable {
    private Long id;

    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="SEQ_STORE")
    public Long getId() { return id; }
}

```

Store 类使用名为 “my_sequence” 的序列，“SEQ_STORE” 序列生成器在其他类中是不可见的；

下例使用 identity 的方式生成主键：

```

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)

```

```
public Long getId() {  
    return null;  
}
```

如果想要实现可移植的应用程序（跨数据库），则 AUTO 生成器是最佳选择。

5.1.4 复合主键

基于业务需求，会需要使用两个字段来作复合主键，例如在 T_USER 数据表中，也许会使用“name”与“phone”两个字段来定义复合主键。

在设计 Java 程序时，建议为复合主键设计一个对应的对象，无论在对象语义上，或是程序编写上，都可以明确知道该对象代表主键信息，而当中的属性为必填信息。

例如，可以设计一个 UserPK 类，对应 T_USER 表格中的“name”与“phone”复合主键：

```
@Embeddable  
  
public class UserPK implements Serializable {  
    private static final long serialVersionUID = 8623239679751042846L;  
    private String name;  
    private String phone;  
  
    @Override  
    public boolean equals(Object obj) {  
        if (obj == null) {  
            return false;  
        }  
        if (getClass() != obj.getClass()) {  
            return false;  
        }  
        final UserPK other = (UserPK) obj;  
        if ((this.name == null) ? (other.name != null) : !this.name.equals(other.name)) {  
            return false;  
        }  
        if ((this.phone == null) ? (other.phone != null) : !this.phone.equals(other.phone)) {  
            return false;  
        }  
        return true;  
    }  
}
```

```

@Override
public int hashCode() {
    int hash = 5;
    hash = 73 * hash + (this.name != null ? this.name.hashCode() : 0);
    hash = 73 * hash + (this.phone != null ? this.phone.hashCode() :
0);

    return hash;
}

// Getters & Setters
...
}

```

作为复合主键的类，必须实现 `Serializable` 接口，且必须重新定义 `equals()` 与 `hashCode()` 方法，必须有无参数（预设）构造函数。而在这边使用 `@Embeddable` 标示该类，表示这个类将附属于另一个实体类，而该实体类可以这么设计：

```

@Entity
@Table(name = "T_USER")
public class User implements Serializable {
    private static final long serialVersionUID = 8301770461930996704L;

    @EmbeddedId
    private UserPK userPK;
    private Long age;

    // Getters & Setters
    ...
}

```

若要将标示为 `@Embeddable` 的类嵌入某个实体类中作为复合主键类，则使用 `@EmbeddedId` 标示。

在储存 `User` 时的一个例子如下：

```

UserPK pk = new UserPK();
pk.setName("bush");
pk.setPhone("0970123456");
User user = new User();
user.setUserPK(pk);
user.setAge(new Long(35));

```

```
EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();
EntityTransaction etx = entityManager.getTransaction();
etx.begin();
entityManager.persist(user);
etx.commit();
entityManager.close();
```

而要加载 User，则使用 UserPK 实例进行查询：

```
UserPK pk = new UserPK();

pk.setName("bush");

pk.setPhone("0970123456");

EntityManager entityManager =

    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();

etx.begin();

User user = entityManager.find(User.class, pk);

etx.commit();

entityManager.close();
```

另一个设计主键类的方式，是让主键类无需嵌入实体类，例如，可以如下设计一个主键类，无需使用任何标注：

```
public class UserPK implements Serializable {

    private static final long serialVersionUID = 8623239679751042846L;

    private String name;

    private String phone;

    @Override

    public boolean equals(Object obj) {

        if (obj == null) {

            return false;

        }

        if (getClass() != obj.getClass()) {

            return false;

        }

    }

}
```

```

        final UserPK other = (UserPK) obj;

        if ((this.name == null) ? (other.name !=
null) : !this.name.equals(other.name)) {

            return false;

        }

        if ((this.phone == null) ? (other.phone !=
null) : !this.phone.equals(other.phone)) {

            return false;

        }

        return true;

    }

    @Override
    public int hashCode() {

        int hash = 5;

        hash = 73 * hash + (this.name != null ? this.name.hashCode() : 0);

        hash = 73 * hash + (this.phone != null ? this.phone.hashCode() :
0);

        return hash;

    }

    // Getters & Setters
    ...
}

```

必须实现 `Serializable` 接口，且必须重新定义 `equals()` 与 `hashCode()` 方法，必须有无参数（预设）构造函数。

而在设计 `User` 类时，必须使用 `@IdClass` 标注主键类之信息，而 `User` 中有对应于表格字段的所有信息，而对应主键的属性，使用 `@Id` 标注：

```

@Entity
@Table(name = "T_USER")
@IdClass(com.hyland.jpa.tutorial.UserPK.class)
public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id

    private String name;

    @Id

```

```
private String phone;

private Long age;

// Getters & Setters

...

}
```

储存时例子如下：

```
User user = new User();
user.setName("bush");
user.setPhone("0970123456");
user.setAge(new Long(35));

User user = new User();
user.setUserPK(pk);
user.setAge(new Long(35));

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();
etx.begin();

entityManager.persist(user);

etx.commit();

entityManager.close();
```

只是在语义上，若无特别批注或文件说明，使用 User 实例时，可能忽略了主键字段所对应的属性必须全部填写而发生错误。

若要加载 User 数据，一样是使用 UserPK 实例：

```
UserPK pk = new UserPK();
pk.setName("bush");
pk.setPhone("0970123456");

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();
etx.begin();

User user = entityManager.find(User.class, pk);

etx.commit();

entityManager.close();
```

5.1.5 Lob 对象

在 Hibernate 中，可以直接对 Blob、Clob 作映射，例如在 T_USER 表格中，若有 BLOB 与 CLOB 字段分别为 photo 与 resume，则可以如下设计一个 User 类：

```
@Entity
@Table(name="T_USER")
public class User implements Serializable {
    private static final long serialVersionUID = 1157688182550942700L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private Long age;
    @Lob
    private byte[] photo;
    @Lob
    private char[] resume;
    // Getters & Setters
    ...
}
```

无论是映射至 BLOB 或 CLOB 的字段，在 JPA 中都是使用 @Lob 来标注，而 JPA 会自动根据属性来判断是要采用 BLOB 或 CLOB 的处理方式储存至表格中，若属性是 byte[] 型态，则以 BLOB 方式处理，若属性是 char[] 型态，则使用 CLOB 方式处理。

一个储存档案至表格的范例如下：

```
FileInputStream inputStream =
    new FileInputStream("c://workspace//pgao.jpg");
byte[] photo = new byte[inputStream.available()];
inputStream.read(photo);
inputStream.close();

User user = new User();
user.setName("pgao");
user.setAge(new Long(35));
user.setPhoto(photo);
user.setResume("Pgao's resume text.".toCharArray());

EntityManager entityManager =
```

```

        JPAUtil.getEntityManagerFactory().createEntityManager();

        EntityManager etx = entityManager.getTransaction();

        etx.begin();

        entityManager.persist(user);

        etx.commit();

        entityManager.close();

```

而一个取出档案的范例如下：

```

EntityManager entityManager =

        JPAUtil.getEntityManagerFactory().createEntityManager();

        EntityManager etx = entityManager.getTransaction();

        etx.begin();

        User user = entityManager.find(User.class, id);

        etx.commit();

        entityManager.close();

        FileOutputStream outputStream =

            new FileOutputStream("c://workspace//pgao2.jpg");

        outputStream.write(user.getPhoto());

        outputStream.close();

```

5.1.6 多表映射

有时必须将同一个对象实体中的属性，映射至两个表格，例如，有一个 User 类，其中 name 与 age 属性要映射至 T_USER 表格，而 email、blog、twitter 属性要映射至 T_CONTACT 表格，T_USER 与 T_CONTACT 表格分别拥有自己的主键 ID 为 USER_ID 与 CONTACT_ID。

可以使用 @SecondTable 来标注 User 对象所要映射的其它表格，例如：

```

@Entity
@Table(name="T_USER")
@SecondaryTable(name="T_CONTACT", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="CONTACT_ID")
})

public class User implements Serializable {

    private static final long serialVersionUID = -1157688182550942700L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    @Column(name="USER_ID")

    private Long id;

```



```

    private String name;

    private Long age;

    @Column(name="email", table="T_CONTACT")
    private String email;

    @Column(name="blog", table="T_CONTACT")
    private String blog;

    @Column(name="twitter", table="T_CONTACT")
    private String twitter;

    // Getters & Setters

    ...
}

```

在 `@SecondTable` 中，必须指定所映射的第二个表格名称，并使用内嵌的 `@PrimaryKeyJoinColumn` 来标注所要连结的主键字段，该主键域值将与 `T_USER` 的主键域值同步。而在对应于第二个表格的属性，则必须使用 `@Column` 标注出来，表明其对应哪一个表格的哪一个字段。

如果使用 `EntityManager` 来储存 `User` 对象，则会分别将属性储存至所设定的对应字段，而在查找时，则会使用 `left outer join` 从两个表格中撷取域值，在删除对象时，则会从两个表格中删除对应的数据。

如果一个实体对象所要映射的表格不只有两个，而有两个以上的话，则使用 `@SecondaryTables` 标注，而在内嵌的部份，再使用 `@SecnondTable` 分别标注出所要映射的表格，例如：

```

@Entity
@Table(name="CUSTOMER_TABLE")
@SecondaryTables({
    @SecondaryTable(name="T_CONTACT",
        pkJoinColumns={@PrimaryKeyJoinColumn (name="CONTACT_ID")}),
    @SecondaryTable(name="T_OOO",
        pkJoinColumns={@PrimaryKeyJoinColumn (name="OOO_ID")}),
    @SecondaryTable(name="T_XXX",
        pkJoinColumns={@PrimaryKeyJoinColumn (name="XXX_ID")})
})
public class User implements Serializable {
    ...
}

```

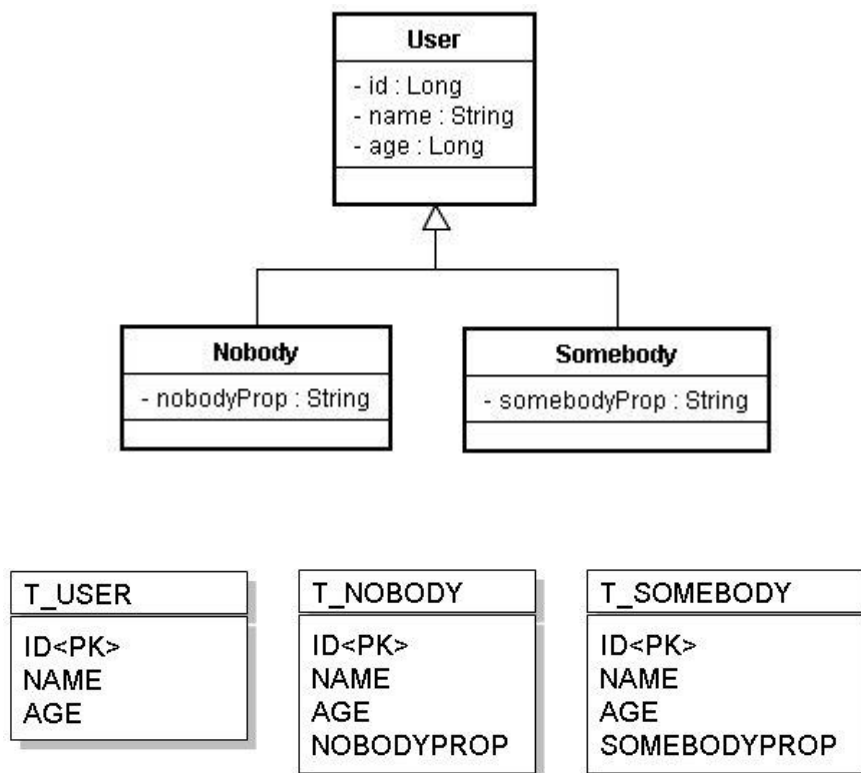
而 `User` 类中的属性，同样使用 `@Column` 来标注将映射至哪个表格。

5.2 继承映射

在面向对象设计中，继承关系是很常见的，继承关系至表格的设计上有几种方式。

5.2.1 Table per Concret Class

如果采取的是对象模型的方式来设计程序，那么继承关系可能就会在的程序设计中出现，然而关系数据库的关联模型与对象模型并不匹配，为了映射对象模型与关联模型，这里先介绍最简单的一种策略：Table per concrete class，也就是继承体系中每一个类就对应一个表格。以实例来说明，如果的程序中有以下的继承关系：



虽然这样作，表格上没有复杂的关系，除非是遗留下的系统原先表格就是这么设计，否则不建议使用这种方式。

在以对象的观点进行多型查询时，例如查询所有类型为 `User` 的数据时，必须将所有 `T_USER`、`T_NOBODY` 与 `T_SOMEBODY` 的数据都查出并加以封装，在下 SQL 语句时，必须使用 SQL `UNION`、子查询或使用多个 `SELECT` 个别查询表格，才可以达到这个目的，在效能上不好。

另外，每个表格中有一些语义相同的字段，例如 `name` 字段，当领域模型对象修

改时，这些相同语义的字段就要同时跟着修改，多个表格共享相同语义，将造成维护上的困难。

而厂商很难为这个策略进行实现或实现方式不一，因此 JPA 并没有要求厂商必须对此功能作出实现。

无论如何，若打算实现这个策略，在 JPA 下可以如下定义 User 类，必须使用 @Inheritance 标注，并设定 strategy 为 InheritanceType.TABLE_PER_CLASS：

```
Entity
@Table(name = "T_USER")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class User implements Serializable {
    private static final long serialVersionUID = 8301770461930996704L;
    @Id
    private Long id;
    private String name;
    private Long age;
    // Getters & Setters
    ...
}
```

而子类的部份，直接标注 @Entity 与 @Table 即可，例如：

```
@Entity
@Table(name="T_NOBODY")
public class Nobody extends User {
    private static final long serialVersionUID = 3837895858055507961L;
    private String nobodyProp;
    // Getters & Setters
    ...
}

@Entity
@Table(name="T_SOMEBODY")
public class Somebody extends User {
    private static final long serialVersionUID = 533199173443419320L;
    private String someBodyProp;
    // Getters & Setters
    ...
}
```

```
}
```

若储存的是 User 实例，则会储存至 T_USER 表格，若储存的是 Nobody 实例，则会储存至 T_NOBODY 表格，若储存的是 Somebody 实例，则会储存至 T_SOMEBODY 表格。

而查询时若使用 find() 方法：

```
user = entityManager.find(User.class, new Long(1));
```

则会从 T_USER 表格查询。同样地，若使用：

```
nobody = entityManager.find(Nobody.class, new Long(2));
```

则会从 T_NOBODY 表格查询。

若使用 Query 对象搭配 JPQL 来查询，可以如下进行多型查询：

```
Query query = entityManager.createQuery("SELECT user FROM User user");
Iterator users = query.getResultList().iterator();
while(users.hasNext()) {
    user = (User) users.next();
    System.out.printf("%d\t%s\t%d\n", user.getId(),
        user.getName(), user.getAge());
}
```

这会查询 T_USER、T_NOBODY 与 T_SOMEBODY 所有的数据，实际查询是使用 SQL UNION、子查询或使用多个 SELECT 个别查询表格，则依厂商实现而有所不同。

由于厂商很难为这个策略进行实现或实现方式不一，JPA 也没有要求厂商必须对此功能作出实现，所以实际要看 JPA 的底层实现如何动作，在采取这个策略时必须对程序多所测试确定行为无误。

5.2.2 Single Table per Class Hierarchy

接续 Table per Concrete Class，来看看继承关系映射至关系数据库的第二种方式：Single Table Class Hierarchy。这种方式使用一个表格储存同一个继承层的所有类，并使用额外的字段来表示所记录的是哪一个子类的数据。

具体来说，对于继承 User 类的 Nobody 及 Somebody，可以设计以下的表格来储存数据：



现在所决定的是，如果要储存的数据是来自 Nobody，则在 DISCRIMINATOR 记下一个型态说明，例如 “Nobody” 字符串，表示该笔数据为 Nobody 实体的对应数据。如果要储存的数据是来自 Somebody，则在 DISCRIMINATOR 记下一个型态说明，例如 “Somebody” 字符串，表示该笔数据为 Somebody 实体的对应数据。如果要储存的数据是来自 User，则在 DISCRIMINATOR 记下一个型态说明，例如 “User” 字符串，表示该笔数据为 User 实体的对应数据。

在实体类上，则可以使用 `InheritanceType.SINGLE_TABLE` 来设定 `@Inheritance` 的 `strategy`（事实上，`InheritanceType.SINGLE_TABLE` 是默认值），并使用 `@DiscriminatorColumn` 与 `@DiscriminatorValue` 来设定区别类型字段的名称与储存值。

例如，User 类可以如下设计：

```

@Entity
@Table(name = "T_USER")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType =
DiscriminatorType.STRING)
@DiscriminatorValue("User") // 预设类名称
public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private Long age;
    // Getters & Setters
    ...
}
  
```

其中 `@DiscriminatorValue` 预设会使用类名称，也可以改用其它的名或名称，而

Nobody 与 Somebody 类可如下设计：

```
@Entity
@DiscriminatorValue("Nobody") // 预设 of 类名称
public class Nobody extends User {
    private static final long serialVersionUID = 3837895858055507961L;
    private String nobodyProp;
    // Getters & Setters
    ...
}

@Entity
@DiscriminatorValue("Somebody") // 预设 of 类名称
public class Somebody extends User {
    private static final long serialVersionUID = 533199173443419320L;
    private String someBodyProp;
    // Getters & Setters
    ...
}
```

假设分别储存了 User、Nobody 与 Somebody 实例，则一个 MySQL 数据库中的表格状态如下所示：

```
mysql> select * from t_user;
+-----+-----+-----+-----+-----+-----+
| DISCRIMINATOR | id | age | name       | nobodyProp | someBodyProp |
+-----+-----+-----+-----+-----+-----+
| User          | 1 | 30 | caterpillar | NULL       | NULL         |
| Nobody        | 2 | 35 | Justin     | Nobody....xD | NULL         |
| Somebody      | 3 | 32 | momor      | NULL       | Somebody...xD |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

缺点就是，因子类属性的不同，对映储存时会有许多字段为 NULL，较浪费数据库空间，但查询效率较好，例如查询 User 类型的数据时，只需一次 SQL，例如这段程序代码：

```
Query query = entityManager.createQuery("SELECT user FROM User user");

Iterator users = query.getResultList().iterator();

while(users.hasNext()) {
```

```

    user = (User) users.next();

    System.out.printf("%d\t%s\t%d\n", user.getId(),
        user.getName(), user.getAge());
}

```

如果是查询个别子类型数据，则会以 WHERE 子句比对 DISCRIMINATOR 型态，例如：

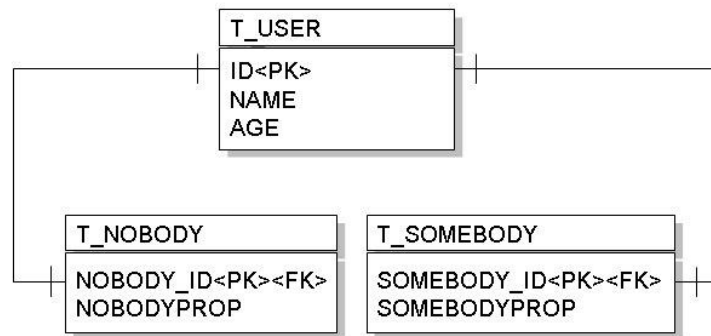
```

Query query = entityManager.createQuery("SELECT nobody FROM Nobody
nobody");

```

5.2.3 Table per Subclass (JoinedTables)

Table per subclass 的继承映射方式，给予父类与子类个分别的表格，而父类与子类对应的表格通过外键来产生关联，具体的说，User 类、Nobody 类与 Somebody 类所映射的表格如下：



其中 T_USER 表格的 id 与 Nobody 及 Somebody 的 id 一致，具体的说，在储存 Nobody 实例时，id 与 name 属性记录在 T_USER 表格中，而 nobodyProp 记录在 T_NOBODY 中，假设 T_USER 表格的 id 值为 1，则 T_NOBODY 表格对应的该笔记录其 id 值也会为 1。

而在定义时，可以标注 InheritanceType 为 JOINED，例如：

```

@Entity
@Table(name = "T_USER")
@Inheritance(strategy = InheritanceType.JOINED)
public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    private Long age;
}

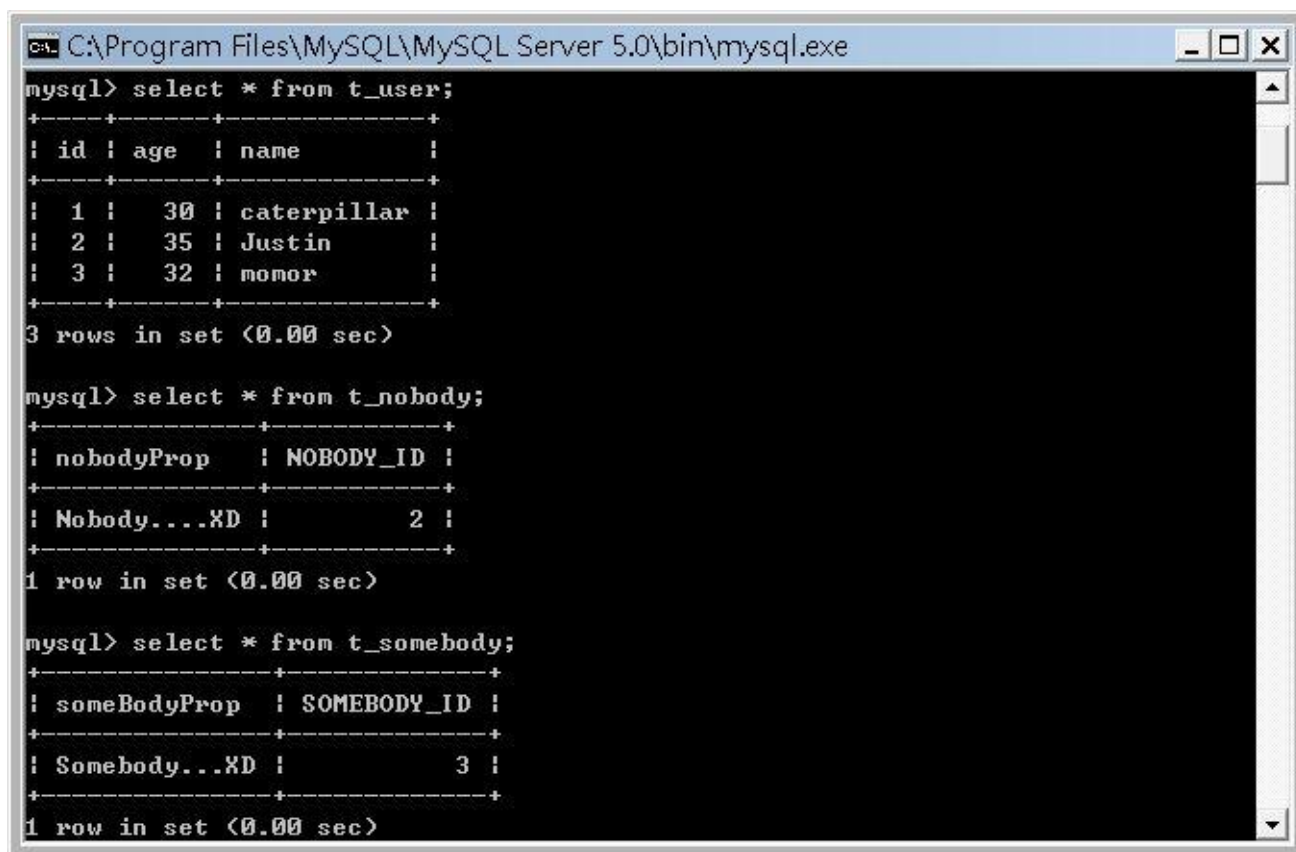
```

```
// Getters & Setters
...
}

@Entity
@Table(name="T_NOBODY")
@PrimaryKeyJoinColumn(name="NOBODY_ID")
public class Nobody extends User {
    private static final long serialVersionUID = 3837895858055507961L;
    private String nobodyProp;
    // Getters & Setters
    ...
}

@Entity
@Table(name="T_SOMEBODY")
@PrimaryKeyJoinColumn(name="SOMEBODY_ID")
public class Somebody extends User {
    private static final long serialVersionUID = 533199173443419320L;
    private String someBodyProp;
    // Getters & Setters
    ...
}
```

假设分别储存了 User、Nobody 与 Somebody 实例，则一个 MySQL 数据库中的表格状态如下所示：



```

C:\Program Files\MySQL\MySQL Server 5.0\bin>mysql.exe

mysql> select * from t_user;
+-----+-----+-----+
| id | age | name      |
+-----+-----+-----+
| 1  | 30  | caterpillar |
| 2  | 35  | Justin    |
| 3  | 32  | monor     |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> select * from t_nobody;
+-----+-----+
| nobodyProp | NOBODY_ID |
+-----+-----+
| Nobody....XD | 2 |
+-----+-----+
1 row in set (0.00 sec)

mysql> select * from t_somebody;
+-----+-----+
| someBodyProp | SOMEBODY_ID |
+-----+-----+
| Somebody...XD | 3 |
+-----+-----+
1 row in set (0.00 sec)

```

而在查询时，会使用适当的 JOIN 来结合表格进行查询，例如在 Hibernate 作为 JPA 的实现下，以下的查询：

```
Query query = entityManager.createQuery("SELECT user FROM User user");
```

而如果是查询个别数据，例如：

```
Nobody nobody = entityManager.find(Nobody.class, new Long(1));
```

性能是这个映射类型需要考虑的，在复杂的类继承下，新增数据必须对多个表格进行，而查询时，跨越多个表格的 join 也可能引发效能上的问题。如果需要多型查询，而子类相对来说有较多新增的属性，则可以使用这种映射方式。

5.2.4 Noentity Base Class

在设计类时，也许会有个父类（抽象或非抽象），当中定义了所有的子类所必须继承的属性，然而它不是实体类，无需对应至任何的表格，然而又想将一些映射的默认信息写在当中，子类继承之后，那些映射信息也跟着继承，必要时又可以重新定义映射信息。

此时可以在该父类上标注 @MappedSuperclass，例如：

```
@MappedSuperclass
```

```
public class Person {  
  
    @Id  
  
    @GeneratedValue(strategy = GenerationType.AUTO)  
  
    private Long id;  
  
    private String name;  
  
    private Long age;  
  
    ...  
}
```

当实体子类继承之时：

```
@Entity  
  
@Table(name="T_USER")  
  
@Inheritance(strategy=InheritanceType.JOINED)  
  
public class User implements Serializable {  
  
    // 被继承的 id 会映射至 T_USER.ID 字段  
  
    // 被继承的 name 会映射至 T_USER.NAME 字段  
  
    // 被继承的 age 会映射至 T_USER.AGE 字段  
  
    ...  
}
```

接着实体子类再继承：

```
@Entity  
  
@Table(name="T_NOBODY")  
  
@PrimaryKeyJoinColumn(name="NOBODY_ID")  
  
public class Nobody extends User {  
  
    private String nobodyProp;  
  
    ...  
}
```

而若想重新定义映射信息，可以使用@AttributeOverride 标注，例如：

```
@Entity  
  
@Table(name="T_USER")
```

```

@Inheritance(strategy=InheritanceType.JOINED)
@AttributeOverride(name="id", column=@Column(name="USER_ID"))

public class User implements Serializable {

    // 被继承的 id 被重新定义映射至 T_USER.USER_ID 字段

    // 被继承的 name 会映射至 T_USER.NAME 字段

    // 被继承的 age 会映射至 T_USER.AGE 字段

    ...

}

```

若有多个属性必须重新定义，则使用@AttributeOverrides，例如：

```

@AttributeOverrides({

    @AttributeOverride(name="xxx", column=@Column("USER_XXX")),

    @AttributeOverride(name="ooo", column=@Column("USER_OOO"))

})

```

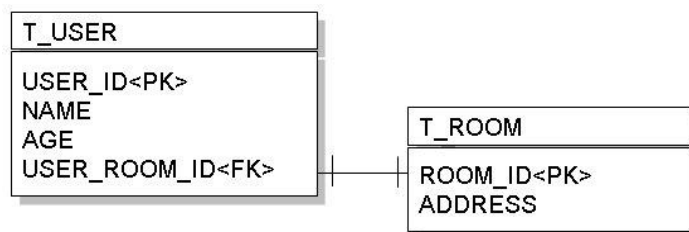
6 关联映射

本篇我们来看“多对一”、“一对多”、“一对一”、“多对多”如何在 Java 对象及表格之间进行映射。

6.1 一对一关联

6.1.1 外键关联

考虑每一个 User 配给一间 Room，形成一对一，T_USER 表格透过 USER_ROOM_ID 作为外键引用至 T_ROOM 的 ROOM_ID：



对象方面，可设计 User 的实例引用至 Room 实例，而希望储存 User 实例时，若有引用至 Room 实例，Room 实例也一并储存。

可以如下设计 User 类：

```
@Entity
```

```

@Table(name = "T_USER")

public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    @Column(name="USER_ID")

    private Long id;

    private String name;

    private Long age;

    @OneToOne(cascade=CascadeType.ALL)

    @JoinColumn(name="USER_ROOM_ID",
referencedColumnName="ROOM_ID")

    private Room room;

    // Getters & Setters

    ...

}

```

从上面可以看到，使用@OneToOne 来标注一对一实体关联，而 cascade 设定为 CascadeType.ALL，表示储存 User 实例时，若有引用至 Room 实例，Room 实例也一并储存，这个称之为联级（Cascade）操作，设定为 ALL，表示之后修改、删除等动作，也会一并更新 Room 的对应表格数据。

关于 Fetch 模式的说明，还可以引用 CascadeType 与 FetchType。

@JoinColumn 中设定 T_USER 表格透过 USER_ROOM_ID 作为外键引用至 T_ROOM 的 ROOM_ID。

Room 类的设计可以如下所示：

```

@Entity

@Table(name="T_ROOM")

public class Room implements Serializable {

    private static final long serialVersionUID = 7681100735999705373L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long id;

    private String address;

    // Getters & Setters

    ...

}

```

一个储存的例子如下所示，由于设定了联级操作为 ALL，所以只要储存 User，所引用的 Room 也会一并储存：

```
Room room = new Room();
room.setAddress("NTU-M8-419");

User user = new User();
user.setName("pgao");
user.setAge(new Long(30));
user.setRoom(room);

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();

etx.begin();

entityManager.persist(user);

etx.commit();

entityManager.close();
```

储存时会先储存 Room 的数据，取得 ROOM_ID 之后，再储存 User 的数据，储存时的数据表内容范例如下：

```
mysql> select * from t_user;
+-----+-----+-----+-----+
| USER_ID | age  | name      | USER_ROOM_ID |
+-----+-----+-----+-----+
| 1       | 30   | caterpillar | 1             |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

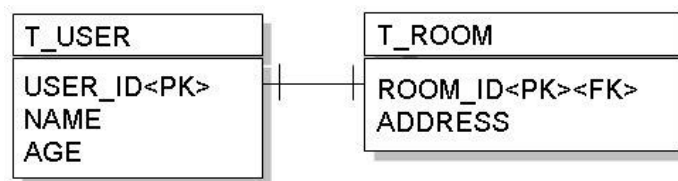
mysql> select * from t_room;
+-----+-----+
| ROOM_ID | address |
+-----+-----+
| 1       | NTU-M8-419 |
+-----+-----+
1 row in set (0.00 sec)
```

而查询时，会使用 LEFT OUTER JOIN 的方式 结合表格进行查询，例如以下的语句：

```
User user = entityManager.find(User.class, new Long(1));
```

6.1.2 共享外键

接续前一个主题一对一（外键关联），可以让 T_USER 与 T_ROOM 的主键共享来实现一对一对应，例如：



象上若要完成这样的对应信息，则可以使用@PrimaryKeyJoinColumn 标注，例如：

```

@Entity
@Table(name = "T_USER")
public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="USER_ID")
    private Long id;

    private String name;

    private Long age;

    @OneToOne(cascade=CascadeType.ALL)
    @PrimaryKeyJoinColumn
    private Room room;

    // Getters & Setters

    ...
}
  
```

Room 类无需作改变，若依一对一（外键关联）中的储存范例，则会先储存 User，取得主键值之后，再储存 Room，让 User 与 Room 的主键值相同，一个数据储存后的表格状态如下所示：

```
mysql> select * from t_user;
+-----+-----+-----+
| USER_ID | age | name |
+-----+-----+-----+
|      1 | 30 | caterpillar |
+-----+-----+-----+
1 row in set (0.00 sec)

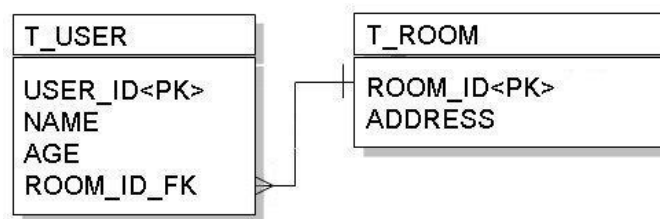
mysql> select * from t_room;
+-----+-----+
| ROOM_ID | address |
+-----+-----+
|      1 | NTU-M8-419 |
+-----+-----+
1 row in set (0.00 sec)
```

而查询时，会使用 LEFT OUTER JOIN 的方式结合表格进行查询，例如以下的语句：

```
User user = entityManager.find(User.class, new Long(1));
```

6.2 多对一

若多个使用者可共住一个房间，则使用者与房间的关系就是多对一的关系。



可以使用 @ManyToOne 于 User 类的 Room 属性上标示多对一关系，例如：

```
@Entity
@Table(name = "T_USER")
public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="USER_ID")
    private Long id;

    private String name;

    private Long age;

    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="ROOM_ID_FK")
```

```

    private Room room;

    // 以下为Getter Setter

    ...
}

```

而 Room 类可以编写代码如下：

```

@Entity
@Table(name="T_ROOM")
public class Room implements Serializable {

    private static final long serialVersionUID = 7681100735999705373L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="ROOM_ID")
    private Long id;
    private String address;

    // 以下为Getter Setter

    ...
}

```

在这样的设定下，是由 User 维持对 Room 的引用来维持多对一的关系，Room 并没有意识到 User 的存在，而在储存时，由于设定了 CascadeType.ALL，所以直接储存 User 实例时，所引用的 Room 实例也会一并被储存，例如一个范例如下所示：

```

Room room = new Room();
room.setAddress("NTU-M8-419");
User user1 = new User();
user1.setName("pgao");
user1.setAge(new Long(30));
user1.setRoom(room);
User user2 = new User();
user2.setName("Justin");
user2.setAge(new Long(35));
user2.setRoom(room);
EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();
EntityTransaction etx = entityManager.getTransaction();
etx.begin();

```



```

entityManager.persist(user1);

entityManager.persist(user2);

etx.commit();

entityManager.close();

```

这时会先储存 Room 实例,取得 ROOM_ID 之后,再储存 User 实例,并将 ROOM_ID_FK 设为与 ROOM_ID 相同,此时表格内容如下所示:

```

mysql> select * from t_user;
+-----+-----+-----+-----+
| USER_ID | age | name       | ROOM_ID_FK |
+-----+-----+-----+-----+
|      1  | 30  | caterpillar |      1      |
|      2  | 35  | Justin     |      1      |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from t_room;
+-----+-----+
| ROOM_ID | address |
+-----+-----+
|      1  | NTU-M8-419 |
+-----+-----+
1 row in set (0.00 sec)

```

而如果使用以下方式查询:

```
User user = entityManager.find(User.class, new Long(1));
```

6.3 一对多

接续多对一的内容, User 对 Room 是多对一的关系,反过来 Room 对 User 就是一对多的关系,若要由 Room 来维持 User 的引用,则可以如下设计:

```

@Entity
@Table(name="T_ROOM")
public class Room implements Serializable {

    private static final long serialVersionUID = 7681100735999705373L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String address;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="ROOM_ID_FK")
    private Set<User> users;
}

```

```

    public void addUser(User user) {
        users.add(user);
    }

    public void removeUser(User user) {
        users.remove(user);
    }

    // Getters & Setters
    ...
}

```

在这边使用 Set 来维持对多个 User 实例的引用，而的 User 可以如下设计：

```

@Entity
@Table(name = "T_USER")
public class User implements Serializable {
    private static final long serialVersionUID = 8301770461930996704L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="USER_ID")
    private Long id;
    private String name;
    private Long age;

    // Getters & Setters
    ...
}

```

一个储存的例子如下所示，由于现在是由 Room 来维持对 User 的引用，所以直接储存 Room 实例，User 实例也会一并储存：

```

User user1 = new User();
user1.setName("pgao");
user1.setAge(new Long(30));
User user2 = new User();
user2.setName("Justin");
user2.setAge(new Long(35));
Room room = new Room();
room.setUsers(new HashSet<User>());
room.setAddress("NTU-M8-419");
room.addUser(user1);

```

```

room.addUser(user2);

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

    EntityTransaction etx = entityManager.getTransaction();

etx.begin();

entityManager.persist(room);

etx.commit();

entityManager.close();

```

这个时候，有个效能上的议题可以探讨，请引用 双向关联。在查询时，要注意的是，@OneToMany 默认的 Fetch 模式是 FetchType.LAZY，若直接以下面的程序打算显示 User 信息：

```

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();

etx.begin();

room = entityManager.find(Room.class, new Long(1));

etx.commit();

entityManager.close();

System.out.println(room.getUsers());

```

FetchType.LAZY 时，除非真正要使用到该属性的值，否则不会真正将数据从表格中加载对象，所以上例中，由于 EntityManager 已经关闭，而此时若要再加载 User，就会发生异常错误，解决的方式之一是在 EntityManager 关闭前取得数据。

```

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();

EntityTransaction etx = entityManager.getTransaction();

etx.begin();

room = entityManager.find(Room.class, new Long(1));

etx.commit();

System.out.println(room.getUsers());

entityManager.close();

```

或者是在 @OneToMany 上指定 fetch 属性为 FetchType.EAGER，表示一并加载所有属性所对应的数据：

```

@OneToMany(cascade=CascadeType.ALL, mappedBy="room",
    fetch=FetchType.EAGER)

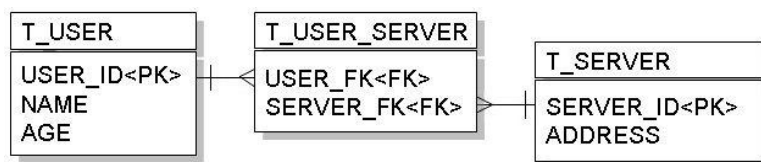
private Set<User> users;

```

关于 Fetch 模式的说明，还可以引用 CascadeType 与 FetchType。

6.4 多对多

在数据库表格上进行多对多对应，可以藉由一个中介表格来完成，也就是藉由多对一、一对多来完成多对多关联。



多对多由于使用了中介表格，在查询效率不彰，且在程序的对象模式上，多对多会使得对象与对象之间彼此依赖，并不是一个很好的设计方式，在设计上应避免使用多对多关系。若要设计多对多关系，则可以使用 @ManyToMany 的标注，例如设计 User 类如下：

```

@Entity
@Table(name = "T_USER")
public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="USER_ID")
    private Long id;

    private String name;

    private Long age;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
        name = "T_USER_SERVER",
        joinColumns = {@JoinColumn(name="USER_FK")},
        inverseJoinColumns = {@JoinColumn(name="SERVER_FK")}
    )
    private Set<Server> servers;

    // Getters & Setters

    ...
}
  
```

其中@JoinTable 中 name 设定的是中介表格的名称，并设定对应的域名。而 Server 可以如下设计：

```
@Entity
@Table(name = "T_SERVER")
public class Server implements Serializable {
    private static final long serialVersionUID = -5602359261549179662L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "SERVER_ID")
    private Long id;
    private String address;

    @ManyToMany(cascade = CascadeType.ALL, mappedBy = "servers")
    private Set<User> users;

    // Getters & Setters
    ...
}
```

这边直接透过 mappedBy 属性设定了双向关联，一个储存时的例子如下：

```
Server server1 = new Server();
server1.setAddress("PC-219");
server1.setUsers(new HashSet());
Server server2 = new Server();
server2.setAddress("PC-220");
server2.setUsers(new HashSet());
Server server3 = new Server();
server3.setAddress("PC-221");
server3.setUsers(new HashSet());
User user1 = new User();
user1.setName("pgao");
user1.setServers(new HashSet());
user1.setAge(new Long(35));
User user2 = new User();
user2.setName("momor");
user2.setServers(new HashSet());
user2.setAge(new Long(30));

// 多对多，互相引用
user1.getServers().add(server1);
user1.getServers().add(server2);
user1.getServers().add(server3);
```

```

server1.getUsers().add(user1);
server2.getUsers().add(user1);
server3.getUsers().add(user1);
user2.getServers().add(server1);
user2.getServers().add(server3);
server1.getUsers().add(user2);
server3.getUsers().add(user2);

EntityManager entityManager = JPAUtil.getEntityManagerFactory().createEntityManager();
EntityTransaction etx = entityManager.getTransaction();
etx.begin();
entityManager.persist(user1);
entityManager.persist(user2);
etx.commit();
entityManager.close();

```

```

mysql> select * from t_user;
+-----+-----+-----+
| USER_ID | age | name      |
+-----+-----+-----+
|      1 | 35 | caterpillar |
|      2 | 30 | momor      |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from t_user_server;
+-----+-----+
| USER_FK | SERVER_FK |
+-----+-----+
|      1 |      1 |
|      1 |      2 |
|      1 |      3 |
|      2 |      1 |
|      2 |      2 |
+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from t_server;
+-----+-----+
| SERVER_ID | address |
+-----+-----+
|      1 | PC-219 |
|      2 | PC-221 |
|      3 | PC-220 |
+-----+-----+
3 rows in set (0.00 sec)

```

如果使用以下方式进行查询：

```

EntityManager entityManager =
JPAUtil.getEntityManagerFactory().createEntityManager();
EntityTransaction etx = entityManager.getTransaction();

```

```

etx.begin();
user1 = entityManager.find(User.class, new Long(1));
etx.commit();
System.out.println(user1.getServers());
entityManager.close();

```

6.5 双向关联

在多对一与一对多中所实现的，分别是 User 对 Room 的单向关联，以及 Room 对 User 的单向关联。

在一对多中的储存范例，有个效能的议题可以讨论，若使用 Hibernate 作为 JPA 的实现，当中的范例在储存时，会产生以下的 SQL 语句：

Hibernate:

```
insert into T_ROOM (address) values (?)
```

Hibernate:

```
insert into T_USER (age, name) values(?, ?)
```

Hibernate:

```
insert into T_USER (age, name) values(?, ?)
```

Hibernate:

```
update T_USER set ROOM_ID_FK=? where USER_ID=?
```

Hibernate:

```
update T_USER set ROOM_ID_FK=? where USER_ID=?
```

在储存 Room 取得 ROOM_ID 之后，由于仅实现 Room 对 User 的一对多单向关联，在储存的时候，User 无法直接引用到 Room 实例，所以只得先对 User 实例分别储存，再将用 UPDATE 语句，以 ROOM_ID 更新 T_USER 表格的 ROOM_ID_FK 字段。

若能实现一对多与多对一的双向关联，也就是 User 可以引用到 Room，而 Room 也可以引用到 User，在储存时，可以将关联维持的控制权交给多的一方，这样会比较有效率，理由不难理解，就像是在公司中，老板要记住多个员工的姓名快，还是每一个员工都记得老板的姓名快。

如果要实现 User 与 Room 的双向关联，则 User 可以如下设定：

```
@Entity
```

```
@Table(name = "T_USER")

public class User implements Serializable {

    private static final long serialVersionUID = 8301770461930996704L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="USER_ID")

    private Long id;

    private String name;

    private Long age;

    @ManyToOne(cascade = CascadeType.ALL)

    @JoinColumn(name = "ROOM_ID_FK")

    private Room room;

    // Getters & Setters

    ...

}
```

而在 Room 这边，注意使用 mappedBy 属性来标示其为被控方（非主控方）：

```
@Entity

@Table(name="T_ROOM")

public class Room implements Serializable {

    private static final long serialVersionUID =
-7681100735999705373L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ROOM_ID")

    private Long id;

    private String address;

    @OneToMany(cascade=CascadeType.ALL, mappedBy = "room")

    private Set<User> users;

    public void addUser(User user) {

        users.add(user);

    }

    public void removeUser(User user) {

        users.remove(user);

    }

    // Getters & Setters

}
```



```
...  
}
```

此时可以如下以 User 为主控方进行储存：

```
Room room = new Room();  
room.setAddress("NTU-M8-419");  
User user1 = new User();  
user1.setName("pgao");  
user1.setAge(new Long(30));  
user1.setRoom(room);  
User user2 = new User();  
user2.setName("Justin");  
user2.setAge(new Long(35));  
user2.setRoom(room);  
EntityManager entityManager =  
    JPAUtil.getEntityManagerFactory().createEntityManager();  
EntityTransaction etx = entityManager.getTransaction();  
etx.begin();  
entityManager.persist(user1);  
entityManager.persist(user2);  
etx.commit();  
entityManager.close();
```

若是要储存 Room，则可以设定 User 与 Room 交互引用，真正储存时，直接储存 Room 实例：

```
Room room = new Room();  
room.setAddress("NTU-M8-419");  
room.setUsers(new HashSet<User>());  
User user1 = new User();  
user1.setName("pgao");  
user1.setAge(new Long(30));  
user1.setRoom(room);  
User user2 = new User();  
user2.setName("Justin");  
user2.setAge(new Long(35));  
user2.setRoom(room);  
room.addUser(user1);  
room.addUser(user2);
```

```

EntityManager entityManager =
    JPAUtil.getEntityManagerFactory().createEntityManager();
EntityTransaction etx = entityManager.getTransaction();
etx.begin();
entityManager.persist(room);
etx.commit();
entityManager.close();

```

此时，JPA 会将储存的主控权转为 User，若使用 Hibernate 作为 JPA 的实现，则会产生以下的 SQL 语句，也就是不再需要额外用 UPDATE 来更新 ROOM_ID_FK：

```

Hibernate:
    insert into T_ROOM (address) values (?)
Hibernate:
    insert into T_USER (age, name, ROOM_ID_FK) values (?, ?, ?)
Hibernate:
    insert into T_USER (age, name, ROOM_ID_FK) values (?, ?, ?)

```

类似的，一对一关系也可以藉由实例间互相引用设定为一对一双向关联，并于其中一方指定 mappedBy 属性来设定其为非主控方，例如在一对一（外键关联）的例子中，可以如下设定：

```

@Entity
@Table(name="T_USER")
public class User implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name="USER_ID")
    private Long id;
    private String name;
    private Long age;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="USER_ROOM_ID", referencedColumnName="ROOM_ID")
    private Room room;
    ...
}

@Entity

```

```
@Table(name="T_ROOM")

public class Room implements Serializable {

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    @Column(name="ROOM_ID")

    private Long id;

    private String address;

    @OneToOne(cascade=CascadeType.ALL, mappedBy="room")

    private User user;

    ...

}
```

6.6 基本类型和嵌入对象集合

有些情况下，不需要在实体之间建立关联关系，而是通过使用 `@ElementCollection` 注解创建基本类型和可嵌入对象的集合，即可实现所需要的功能。

6.7 CascadeType 和 FetchType

在关联映射中，如一对一、一对多、多对一等，都有设定 `cascade` 为 `CascadeType.ALL`，这表示储存其中一方实例时，若有引用另一方实例，另一方实例也一并储存，这个称之为联级（Cascade）操作。

预设是不使用联级操作，而可设定的联级操作如下所示：

<code>CascadeType.PERSIST</code>	在储存时一并储存被引用的对象。
<code>CascadeType.MERGE</code>	在合并修改时一并合并修改被引用的对象。
<code>CascadeType.REMOVE</code>	在移除时一并移除被引用的对象。
<code>CascadeType.REFRESH</code>	在更新时一并更新被引用的对象。
<code>CascadeType.ALL</code>	无论储存、合并、更新或移除，一并对被引用对象作出对应动作。

在一对多中略为介绍过 `Fetch` 模式，`FetchType.LAZY` 时，除非真正要使用到该属性的值，否则不会真正将数据从表格中加载对象，所以 `EntityManager` 后，才要加载该属性值，就会发生异常错误，解决的方式之一是在 `EntityManager` 关闭前取

得数据，另一个方式则是标示为 `FetchType.EAGER`，表示立即从表格取得数据。

一些标注的 `Fetch` 模式有其默认值，例如：

<code>@Basic</code>	<code>FetchType.EAGER</code>
<code>@OneToOne</code>	<code>FetchType.EAGER</code>
<code>@ManyToOne</code>	<code>FetchType.EAGER</code>
<code>@OneToMany</code>	<code>FetchType.LAZY</code>
<code>@ManyToMany</code>	<code>FetchType.LAZY</code>

不过，即使标注为 `FetchType.LAZY`，此一标注仅为建议，实现厂商仍可以将之实现为 `FetchType.EAGER`。