

J2EE与中间件技术

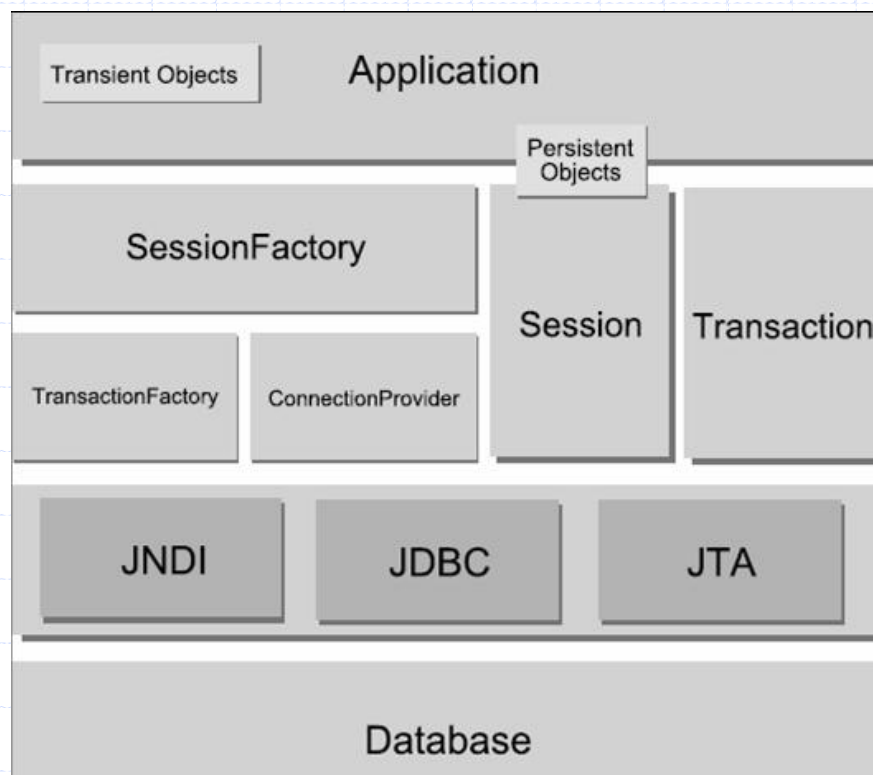
——Hibernate

Hibernate与EJB3.0的关系

- ◆ 作为最成功的ORM解决方案之一，Hibernate在很大程度上影响了EJB 3.0的设计，Hibernate团队参与了EJB标准的制定
- ◆ 可以使用Hibernate作为EJB容器的EntityManager组件实现
- ◆ Hibernate的HQL（Hibernate Query Language）也与新的EJB QL（EJB Query Language）有密切的联系
- ◆ 允许使用注解来描述实体/POJO与数据库之间的关系
- ◆ Hibernate 3（及以上）提供的特性超过了EJB 3.0标准的要求
 - 不要求使用应用服务器，适合Swing应用程序、其他客户端应用程序和轻型Web应用程序（例如，在Tomcat上运行的应用程序）
- ◆ Hibernate 6, 最新

Hibernate体系结构

- ◆ 将应用层从底层的JDBC/JTA API中抽象出来，而让Hibernate来处理这些细节



Eclipse+Hibernate

◆ <http://www.hibernate.org/>

◆ 下载Hibernate包

设置开发环境

◆ 解压

- lib: 库文件

◆ 以Tomcat Web项目为例

- 把lib/required下的文件复制到WEB-INF/lib中

Hibernate API

◆ org.hibernate包

◆ Configuration

- 定位映射文件（hibernate.cfg.xml）的位置，读取这些配置

◆ SessionFactory

- 针对**单个数据库**映射关系经过编译后的内存镜像，是**线程安全**的
- 从Configuration取得，是生成Session的工厂，用到ConnectionProvider
- 用户程序从SessionFactory中取得Session的实例

◆ ConnectionProvider

- 生成JDBC连接的工厂（有连接池的作用）
- 通过抽象将应用从底层的DataSource或DriverManager隔离开，仅供开发者扩展/实现用，并不暴露给应用程序使用

Hibernate API

◆ Session

- 一个持久层管理器，是一个轻量级的类
 - ◆ 包含一些持久层相关的操作，如存储持久对象至数据库、从数据库中获得持久对象等
- 表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短，隐藏了JDBC连接
- 与JPA EntityManager类似

◆ TransactionFactory

- 生成Transaction对象实例的工厂
- 仅供开发者扩展/实现用，并不暴露给应用程序使用

◆ Transaction

- 对实际事务实现的一个抽象
- 应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短

Hibernate API

◆ Query和Criteria

- 对数据库及持久对象进行查询的接口

◆ Callback

- 允许用户程序能对一些事件的发生做出相应的操作

建立实体Bean

- ◆ 实体Bean用来映射数据库中的表
- ◆ 一般一个实体Bean对应于一个数据表
- ◆ 表中的每个字段也可以对应于实体Bean中的某种属性（字段不一定都有对应的属性）
 - 如果在应用程序中采用了Struts2, Hibernate的实体Bean正好与**Struts2**的**模型类**相对应

定义映射关系

◆方法1：编写映射文件来定义映射关系

- . hbm. xml 文件，位于WEB-INF/classes目录中

◆方法2：通过注解映射

User.hbm.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="edu.nju.hbn.User" table="tbluser">
    <id name="userID" column="userID" type="java.lang.String"/>
    <property name="userName" column="UserName" type="java.lang.String"/>
    <property name="userMail" column="userMail" type="java.lang.String"/>
    <property name="userPassword" column="UserPassword" type="java.lang.String"/>
    <property name="userCreated" column="userCreated" type="java.util.Date"/>
    <property name="userType" column="userType" type="java.lang.Integer"/>
  </class>
</hibernate-mapping>
```

.hbm.xml

- ◆ `<class>` 标签: `name` 属性为映射的对象, `table` 属性为映射的表
- ◆ `<id>` 标签: 代表主键, `column` 属性指定表中字段, `type` 属性指定 User 实例中 user ID 的类型
- ◆ `<property>` 标签: `column` 属性指定表中字段, `type` 属性指定对象中属性的类型

通过注解映射

- ◆ Hibernate 框架使用元数据来管理数据库记录和类之间的映射
 - Hibernate 2.x中映射数据大多数基于XML的映射文件
 - Hibernate **3及以上**中加入了注解的支持
 - ◆ 使用JavaSE5.0及以上的JDK版本

通过注解映射

◆使用@Entity注解实体Bean

- 必须是public且不能是abstract

◆使用@Table注解实体Bean

- Catalog 数据库名； name表名 ； schema 所有者名

◆使用@Id注解主键

- 可以注解属性或get方法

配置Hibernate

◆ 指定连接数据库的信息

- JDBC连接
- 或JNDI DataSource

◆ Hibernate中可以使用JDBC或JNDI DataSource来连接数据库

◆ 可使用XML文件进行配置

- 默认Hibernate配置文件名是
hibernate.cfg.xml

hibernate.cfg.xml

- ◆ 使用XML文件对Hibernate框架进行配置是最常用的方式，也是官方建议的配置方式，默认的XML配置文件是hibernate.cfg.xml
- ◆ Hibernate配置文件一般放到CLASSPATH环境指定的路径的根目录，但事实上可以放到任何路径下
- ◆ 为了找到并装载Hibernate配置文件，需要使用org.hibernate.cfg.Configuration.Configuration类的configure方法，此方法有多种重载形式

hibernate.cfg.xml (以 Hibernate3, JDBC连接为例)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD
    3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-
    configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property
            name="connection.driver_class">com.mysql.jdbc.Driver</proper
            ty>
        <property
            name="connection.url">jdbc:mysql://localhost:3306/abcdef</pr
            operty>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
```

以*. hbm. xml映射文件为例

```
<!-- SQL dialect -->
<property
name="dialect">org.hibernate.dialect.MySQLDialect
</property>
<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>
<!-- 根据需要自动创建数据表 -->
<property name="hbm2ddl.auto">update</property>
<!-- 把实体类与属性映射成数据库中的表与列 -->
<mapping resource="User.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

或使用class注解映射

.....

<!-- 把实体类与属性映射成数据库中的表与列 -->

<mapping class="edu.nju.hbn.User"/>

</session-factory>

.....

SQL方言 (Dialect)

- ◆ 数据库方言实际上就是Hibernate提供的一系列Java类
 - 解决不同数据库之间的差异性，使得在Hibernate框架中操作数据库是透明的
 - 如，在查询数据是可以对其分页显示
 - ◆ 对于查询记录的分页显示，不同的数据库的处理方式不同，需要对每种数据库单独来处理

Hibernate 4以上

.....

```
<!-- SQL dialect -->
```

```
<property  
name="dialect">org.hibernate.dialect  
. MySQL5Dialect</property>
```

.....

Hibernate 5

.....

<! - 不支持mapping, 使用编程方式配置-->

<!-- mapping class="edu.nju.hbn.User"/-->

.....

连接池配置

- ◆ c3p0连接池是Hibernate推荐使用的连接池
- ◆ 将lib\optional\c3p0下的*.jar文件复制到WEB-INF/lib中

hibernate.cfg.xml (以c3p0连接池连接为例)

```
.....  
<property  
name="connection.provider_class">org.hibernate.c3p0.inte  
rnal.C3P0ConnectionProvider</property>  
<property name="hibernate.c3p0.max_size">20</property>  
<property name="hibernate.c3p0.min_size">1</property>  
<property name="hibernate.c3p0.timeout">5000</property>  
<property  
name="hibernate.c3p0.max_statements">100</property>  
<property  
name="hibernate.c3p0.idle_test_period">3000</property>  
<property  
name="hibernate.c3p0.acquire_increment">2</property>  
<property name="hibernate.c3p0.validate">true</property>  
.....
```


Hibernate的其他配置方法

◆配置Hibernate方法

- XML文件
- 属性文件
- 编程方式

用属性文件配置Hibernate

- ◆ 可以使用属性文件key-value对来配置Hibernate
- ◆ 配置Hibernate的属性文件名为hibernate.properties

属性名	描述
hibernate.connection.username	用户名
hibernate.connection.password	密码
hibernate.connection.url	JDBC连接字符串
hibernate.dialect	Hibernate方言类
hibernate.connection.driver_class	JDBC驱动类

用编程的方式配置Hibernate

- ◆ 下面的代码通过编程的方式设置了数据库的连接信息

```
Configuration configuration =new Configuration();  
configuration.addResoure( "mapping.xml" ).setProperty(  
    "connection.username", "root" );  
configuration.addResoure( "mapping.xml" )..setProperty(  
    "connection.password", "jameszx" );  
configuration.addResoure( "mapping.xml" )..setProperty(  
    "dialect", "org.hibernate.dialect.MySQLDialect" );  
configuration.addResoure( "mapping.xml" )..setProperty(  
    "connection.url", "jdbc:mysql://localhost/test/?chara  
cterEncoding=UTF8" );  
configuration.addResoure( "mapping.xml" )..setProperty(  
    "connection.driver_class", "com.mysql.jdbc.Driver" );
```

向数据库表添加记录

◆ 基本步骤如下：

- (0) 建立SessionFactory
- (1) 使用SessionFactory类的openSession方法来获得一个Session对象
- (2) 使用Session接口的beginTransaction方法开始一个新事务
- (3) 使用Session接口的save方法保存实例
- (4) 如果成功，使用Transaction接口的commit方法提交事务，否则使用rollback方法回滚事务
- (5) 关闭Session对象

(0) 建立会话工厂

- ◆ 这一步不是必须的，但建立一个SessionFactory来获得Hibernate Session对象是一个好习惯
- ◆ Hibernate3 示例：

```
//读hibernate.cfg.xml配置文件  
Configuration config = new Configuration().configure();  
//建立SessionFactory  
SessionFactory sessionFactory =  
config.buildSessionFactory();
```

Hibernate4

◆ 新增ServiceRegistry接口

- 将Hibernate的配置或者服务等统一向ServiceRegistry注册后，才能生效
 - ◆ 方便统一管理
- 建立SessionFactory步骤
 - ◆ 构建ServiceRegistry对象
 - ◆ 将配置信息向它注册
 - ◆ Configuration对象根据从ServiceRegistry对象中获取的配置信息生成SessionFactory

Hibernate4 示例

```
Configuration config = new  
Configuration().configure();  
ServiceRegistry serviceRegistry = new  
StandardServiceRegistryBuilder().applySettings(con  
fig.getProperties()).build();  
SessionFactory sessionFactory =  
config.buildSessionFactory(serviceRegistry);
```

Hibernate5 示例

```
Configuration config = new Configuration().configure();  
config.addAnnotatedClass(User.class);
```

//编程配置映射

//否则org.hibernate.MappingException: Unknown entity:

```
ServiceRegistry serviceRegistry = .....
```


(1) 获得一个Session对象... (5)

.....

```
session=sessionFactory.openSession();
```

```
Transaction tx =
```

```
session.beginTransaction();
```

```
session.save(user); //保存Entity到数据库  
库中
```

```
tx.commit();
```

```
session.close();
```

.....

Session对象

- ◆ Session对象，不是线程安全的
- ◆ Web服务端可以并发来处理多个用户请求，共享同一个Hibernate Session可能会造成冲突

解决方案1——编写HibernateUtil类（3以下版本）

- 通过ThreadLocal类可以很容易地达到这个目的

```
public class HibernateUtil {  
    private static SessionFactory factory;  
    public static final ThreadLocal threadLocal = new ThreadLocal();  
    static {  
        Configuration con = new Configuration().configure();  
        .....  
        factory = con.buildSessionFactory();  
    }  
    public static Session currentSession() {  
        Session currentSession = (Session) threadLocal.get();  
        if (currentSession == null) {  
            currentSession = sessionFactory.openSession();  
            threadLocal.set(currentSession);  
        }  
        return currentSession;  
    }  
    public static void closeSession() {  
        Session currentSession = (Session) threadLocal.get();  
        if (currentSession == null) {  
            currentSession.close();  
        }  
        threadLocal.set(null);  
    }  
    .....  
}
```

(1) 获得... (5) 关闭Session对象

◆ 使用HibernateUtil类

.....

```
session = HibernateUtil.currentSession();  
Transaction tx =  
session.beginTransaction();  
session.save(user); //保存Entity到数据库中  
tx.commit();  
HibernateUtil.closeSession();
```

解决方案2——编写 HibernateUtil类（3以上版本）

```
public class HibernateUtil {  
    private static final SessionFactory sessionFactory;  
    public static SessionFactory getSessionFactory() {  
        .....  
        Configuration config;  
        ServiceRegistry serviceRegistry;  
        .....  
        sessionFactory=config.buildSessionFactory(serviceRegistry);  
        return sessionFactory;  
        .....  
    }  
    public static Session getSession() {  
        return sessionFactory().getCurrentSession();  
    }  
}
```

getCurrentSession()方法

- ◆ 与openSession()的区别:
- ◆ 1、getCurrentSession()创建的session会绑定到**当前线程**, 而openSession()不会
- ◆ 2、getCurrentSession()创建的session会在事务回滚或事物提交后**自动关闭**, 而openSession()必须手动关闭
- ◆ 3、**事务配置**
 - 每数据库事务对应一个session
 - 在配置文件(hibernate.cfg.xml)里需进行如下设置
 - 如果使用的是本地**事务**
 - ◆ `<property name="hibernate.current_session_context_class">thread</property>`
 - 如果使用的是全局事务(jta事务)
 - ◆ `<property name="hibernate.current_session_context_class">jta</property>`

BaseDAO设计

```
public interface BaseDao {  
    public void save(Object bean);  
    public void delete(Object bean) ;  
    public Object load(Class c, String id) ;  
    .....  
}
```

BaseDAOImpl 类——使用 HibernateUtil

```
public class BaseDAOImpl implements BaseDao {  
    public void save(Object bean) {  
        Session session = HibernateUtil.getSession() ;  
        Transaction tx=session.beginTransaction();  
        session.save(bean);  
        tx.commit();  
    }  
    .....  
}
```


简单查询 Criteria Query

.....

//查询所有user

```
Criteria criteria = session.createCriteria(User.class);
```

```
List users=criteria.list();
```

.....

.....

//查询符合条件的user

```
criteria.add(Expression.eq("userName", "测试用户2"));
```

```
users=criteria.list();
```

.....

Criteria Query

- ◆ 通过面向对象的设计，将数据查询条件封装为一个对象，可以看作是传统SQL的对象化表示，如
 - `Criteria criteria = session.createCriteria(User.class);`
 - `criteria.add(Expression.eq("userName", "测试用户2"));`
- ◆ 实际上是SQL——`select * from tbluser where username=“测试用户2”` 的封装，Criteria 是一个查询容器，Expression对象描述了查询条件，add方法将查询条件添加到Criteria实例中

Hibernate Query Language

- ◆ 面向对象的查询语言，查询以对象形式存在的数据，推荐的查询模式

HQL

.....

```
Session session=HibernateUtil.getSession();  
String hql = "from edu.nju.hbn.User as us where  
us.userName like '测试用户2%'";  
Query query=session.createQuery(hql);  
List users=query.list();  
.....
```

Eclipse+Hibernate

◆ <http://www.hibernate.org/>

◆ 下载Hibernate包

设置开发环境(以Hibernate5为例)

◆ 解压

- lib: 库文件

◆ 以Tomcat Web项目为例

- 把lib/required下的文件复制到WEB-INF/lib中

◆ 注: 如果选择jre9, 则缺少以下

- jaxb-impl-2.1.13.jar
- activation-1.1.jar
- javax.xml.bind.jar
- 需要添加, 或选择jre8

- ▼ UserStockWebH
 - > Deployment Descriptor: UserStockWebH
 - > JAX-WS Web Services
 - ▼ Java Resources
 - ▼ src
 - > edu.nju.onlinestock.dao
 - > edu.nju.onlinestock.factory
 - > edu.nju.onlinestock.model
 - > edu.nju.onlinestock.service
 - > edu.nju.onlinestock.servlets
 - > edu.nju.onlinestock.utils
 - > **hibernate.cfg.xml**
 - > Libraries
 - > JavaScript Resources
 - > build
 - ▼ WebContent
 - > image
 - > META-INF
 - > user
 - ▼ WEB-INF
 - ▼ lib
 - antlr-2.7.7.jar
 - classmate-1.3.0.jar
 - dom4j-1.6.1.jar
 - geronimo-jta_1.1_spec-1.1.1.jar
 - hibernate-commons-annotations-5.0.1.Final.jar
 - hibernate-core-5.2.7.Final.jar
 - hibernate-jpa-2.1-api-1.0.0.Final.jar
 - jandex-2.0.3.Final.jar
 - javassist-3.20.0-GA.jar
 - jboss-logging-3.3.0.Final.jar

作业7

◆修改作业4中数据访问层和Model的设计

- edu.nju.onlinestock.model

 - ◆ Hibernate Entity Beans

- edu.nju.onlinestock.dao

 - ◆ Hibernate Session

- Service层

 - ◆ 不使用EJB技术

Hibernate缓存

- ◆ 缓存，存放数据库数据的拷贝
- ◆ SessionFactory缓存（又称作**应用缓存**），二级缓存
 - 存放元数据和预定义SQL
 - 被**应用**范围内的所有session共享
- ◆ **Session缓存**（又称作事务缓存），第一级缓存
 - 1，减少数据库的访问频率，**提高访问性能**
 - 2，保证缓存中的对象与数据库同步，位于缓存中的对象称为**持久化对象**
 - 3，当持久化对象之间存在**关联**时，Session 保证不出现对象图的死锁
 - 问题：只能被当前Session对象访问，多线程（不同的session，不能共享）查询性能较低
 - ◆ 解决方案：**配置**使用二级缓存

查询：session.get()

- ◆ 出于性能考虑，避免无谓的数据库访问，Session在调用数据库查询功能之前，会先在缓存中进行查询
 - 首先在第一级缓存（session）中，通过实体类型和id进行查找：
 - 如果第一级缓存查找命中，且数据状态合法，则直接返回；
 - 否则，查询二级缓存，再没有查询到，就查找数据库，返回的对象为实体对象，如果没有找到，则返回null

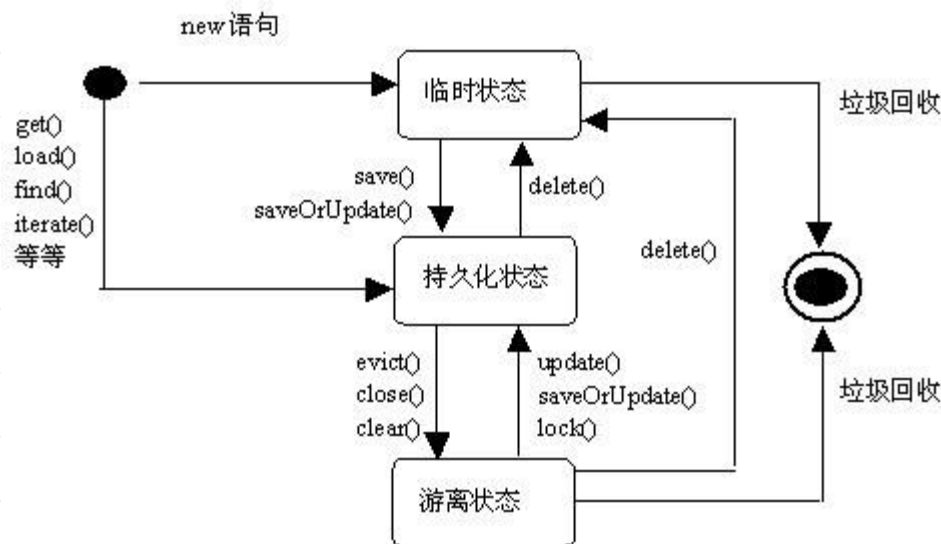
查询：session.load()

◆ 与get()方法区别：延迟加载

- 1. 先查一下session缓存，看看该id对应的对象是否存在
- 2. 如果缓存中没有这个对象，则创建代理
 - ◆ 因为延迟加载需要代理来执行，并没有去查询数据库；
 - ◆ 当使用这个对象时，如user.getName()或get()，才会触发sql语句；
 - ◆ 这时，才会查询二级缓存和数据库，如果数据库没有这条数据，抛出异常ObjectNotFoundException

Hibernate中Entity的三种状态

- ◆ 瞬时/临时状态 (Transient), 持久状态 (Persistent), 脱管/游离状态 (Detached)
- ◆ 三种状态主要取决于对象是否在session缓存中
- ◆ 三种状态转化的方法都是通过session方法来调用



Transient

- ◆ 当通过new生成一个实体对象时，这个实体对象就处于自由状态，如：

```
User user=new User ( “001” , “xyz” , ..... );
```

- ◆ 此时，user通过JVM获得了一块内存空间，并没有通过Session对象的save()方法保存进数据库，还没有纳入Hibernate的缓存管理中，也就是说user对象现在还自由的游荡于Hibernate**缓存管理之外**
- ◆ 在数据库中不存在一条与它对应的记录

Persistent

◆瞬时对象转为持久对象：

- (1) 通过 Session 的 `save()` 和 `saveOrUpdate()` 方法把一个瞬时对象与数据库相关联，这个瞬时对象就成为持久化对象
- (2) 使用 `find()`, `get()`, `load()` 和 `iterator()` 等方法查询到的数据对象，将成为持久化对象

Persistent

- ◆ 持久化对象, 已经被保存进数据库的实体对象, 处于Hibernate的**缓存管理之中**
 - 持久的实例在数据库中有对应的记录, 并拥有一个**持久化标识** (identifier)
 - 对该实体对象的任何**修改**, 都会在**清理缓存时同步**到数据库中
 - ◆ 持久对象与 Session 和 Transaction 相关联, 在一个 Session 中, 对持久对象的**改变不会马上**对数据库进行变更, 而必须在 Transaction 终止, 也就是执行 commit() 之后, 才在数据库中真正运行 SQL 进行变更, 持久对象的状态才会与数据库进行同步。在同步之前的持久对象称为脏 (dirty) 对象。

Persistent

- ◆ 示例: user对象通过save方法保存进数据库后, 成为持久化对象, 然后通过load方法再次加载它(仍然是持久化对象, 处于Hibernate缓存管理之中), 当执行tx.commit()方法时, Hibernate会自动清理缓存, 并且自动将持久化对象的属性变化同步到数据库中

```
User user=new User( "001", "xyz", .....);
```

```
tx=session.beginTransaction();
```

```
session.save(user); //立即执行Sql insert
```

```
user=(User) session.load(User.class, "001"); //延迟加载
```

```
user.setName( "mary" ); //不会立即执行Sql update
```

```
tx.commit(); //立即执行Sql update
```


Detached

◆持久对象转为游离对象：

- 当执行 `close()` 或 `clear()`, `evict()` 之后，持久对象会变为游离对象

◆游离对象转为持久对象：

- 通过 Session 的 `update()`, `saveOrUpdate()` 和 `lock()` 等方法，把游离对象变为持久对象

Detached

◆ 游离对象和自由对象的区别

- 游离对象在数据库中**可能还存在**一条与它对应的记录，只是现在这个游离对象脱离了Hibernate的缓存管理
- 自由对象不会在数据库中出现与它对应的数据记录

Detached

.....

```
session.close();
```

- ◆ 当session关闭后，user对象就不处于Hibernate的缓存管理之中了，但是此时在数据库中还存在一条与user对象对应的数据记录，处于游离态

M:N关系（一）

- ◆ 如果中间表仅仅是做关联用的，仅有2个外键做联合主键，使用 **ManyToMany**
 - 不用写中间表的实体类，只需要写出两张主表的实体类即可
 - 并且不需要手动创建中间表，Hibernate会根据配置自动创建
 - ◆ 中间表字段为两个外键，且作为联合主键
 - ◆ 注意：两个实体类的id属性类型，必须与数据库表的主键类型匹配

示例：User和Stock

◆ 表：users和stocks

◆ 需要中间表trade存放两者关系

- 可自动生成trade表, 2字段：stockid和userid

◆ 自定义关系的维护端与被维护端

- 可选择**Stock**作为关系的**维护端**
- 通过Stock来维护两者的关系，如级联为ALL
 - ◆ 调用session.save(stock)，会自动向users和中间表trade插入数据，即级联新增
 - ◆ 调用session.delete(stock)，自动删除相应的user（与其他stock无关）和trade记录——注：业务逻辑不合理！
- 外键，中间表等信息在这个类里定义

实体

◆ User: inverse side, 关系被维护端

- 使用ManyToMany
- 定义mappedBy属性

◆ Stock: owning side, 关系维护端

- 使用ManyToMany
- 定义JoinTable和外键

User

.....

@Entity

@Table(name= "users")

public class User implements Serializable {

private String id;

private String userName;

.....

private Set<Stock> stocks = new HashSet<Stock>();

@Id

public String getId() {

return id;

}

public void setId(String id) {

this.id=id;

}

@ManyToMany (mappedBy="users")

public Set<Stock> getStocks () {

return stocks;

}

public void setStocks (Set<Stock> stocks) {

this.stocks=stocks;

}

.....

}

Stock

.....

@Entity

@Table(name= "stocks")

```
public class Stock implements Serializable {
```

```
    private String id;
```

```
    private String companyName;
```

.....

```
    private Set<User> users = new HashSet<User>();
```

@Id

```
    public String getId() {
```

```
        return id;
```

```
}
```

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

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

```
}
```

//根据如下内容, 可自动生成中间表

```
@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
```

```
@JoinTable(
```

```
name="trade",
```

```
joinColumns=
```

```
@JoinColumn(name= "stockid", referencedColumnName="id"),
```

```
inverseJoinColumns=
```

```
@JoinColumn(name= "userid", referencedColumnName="id")
```

```
)
```

```
public Set<User> getUsers() {
```

```
    return users;
```

```
}
```

```
public void setUsers(Set<User> users) {
```

```
    this.users=users;
```

```
}
```

```
.....
```

```
}
```

StockDAO

.....

```
public interface StockDAO extends BaseDao {  
    public void saveStock(Stock stock);  
    public void deleteStock(Stock stock);  
    public Stock getStockByID(String id);  
    .....  
}
```

StockDAOImpl

```
public class StockDaoImpl extends  
BaseDaoImpl implements StockDao {  
    public void saveStock (Stock stock) {  
        super.save (stock) ;  
    }  
    .....  
}
```

StockManageService

.....

```
public interface StockManageService {  
    public void InsertStock(Stock stock);  
    public void deleteStock(Stock stock);  
    public Stock getStockById(String id);  
    .....  
}
```

Test1TradeServlet

```
.....  
private static StockManageService  
stockService=ServiceFactory.getStockManageService();  
.....  
Stock s1=new Stock();  
Stock s2=new Stock();  
s1.setId("1");  
.....  
s2.setId("2");  
.....  
User u1 = new User();  
User u2 = new User();  
u1.setId("1");  
.....  
u2.setId("2");  
.....  
//设置二者之间的关系  
s1.getUsers().add(u1);  
s1.getUsers().add(u2);
```

.....

//stock为主控方，因此：

//stocks表新增1行记录，users表和trade表同时各新增2行记录

//如果没有trade表，则自动建表（仅2个字段）

//注意：表字段类型须与实体属性一致

```
stockService.insertStock(s1);
```

◆ 控制台：

```
Hibernate: create table trade (stockid varchar(255) not null, userid  
varchar(255) not null, primary key (stockid, userid)) engine=MyISAM
```

```
Hibernate: alter table trade add constraint ... foreign key (userid)  
references users (id)
```

```
Hibernate: alter table trade add constraint ... foreign key  
(stockid) references stock (id)
```

```
Hibernate: select stock_.id, ..... from stock stock_ where  
stock_.id=?
```

```
Hibernate: select user_.id, ..... from users user_ where user_.id=?
```

```
Hibernate: select user_.id, ..... from users user_ where user_.id=?
```

◆ 控制台:

Hibernate: **insert into stock** (companyName, date, price, type, id)
values (?, ?, ?, ?, ?)

Hibernate: **insert into users** (account, bankid, birthday, email, name,
password, phone, userid, id) values (?, ?, ?, ?, ?, ?, ?, ?, ?)

Hibernate: **insert into users** (account, bankid, birthday, email, name,
password, phone, userid, id) values (?, ?, ?, ?, ?, ?, ?, ?, ?)

Hibernate: **insert into trade** (stockid, userid) values (?, ?)

Hibernate: **insert into trade** (stockid, userid) values (?, ?)

//同时删除trade表、stocks表和users表中的记录

//注：此user必须与其他stock无关，否则出错

stockService.DeleteStock(stock1);

◆ 控制台：

Hibernate: alter table trade add constraint ... foreign key (userid)
references users (id)

Hibernate: alter table trade add constraint ... foreign key (stockid)
references stock (id)

Hibernate: select stock_.id, from stock stock_ where stock_.id=?

Hibernate: select user_.id, from users user_ where user_.id=?

Hibernate: select user_.id, from users user_ where user_.id=?

Hibernate: delete from trade where stockid=?

Hibernate: delete from users where id=?

Hibernate: delete from users where id=?

Hibernate: delete from stock where id=?

```
s2. getUsers().add(u1);
```

```
//stock表新增1行记录, trade表和users表同时各新增1行记录
```

```
stockService.insertStock(s2);
```

```
.....
```

◆ 控制台:

```
Hibernate: alter table trade add constraint FKap7cs6u41to59ke5f926mktc2  
foreign key (userid) references users (id)
```

```
Hibernate: alter table trade add constraint FKd65trceb14hfo9dfhni9irw  
foreign key (stockid) references stock (id)
```

```
Hibernate: select stock_.id, ..... from stock stock_ where stock_.id=?
```

```
Hibernate: select user_.id, ..... from users user_ where user_.id=?
```

```
Hibernate: select user_.id, ..... from users user_ where user_.id=?
```

```
Hibernate: insert into stock (companyName, ..., id) values (?, ?, ?, ?, ?)
```

```
Hibernate: insert into users (account, ....., id) values (?, ?, ?, ?, ?,  
?, ?, ?, ?)
```

```
Hibernate: insert into trade (stockid, userid) values (?, ?)
```

session.merge() 方法

◆ 如果新增的一个对象，存在关联对象，建议使用merge()方法

■ 如果使用session.save()方法

```
s1.getUsers().add(u1);
```

```
s1.getUsers().add(u2);
```

```
stockService.insertStock(s1);
```

```
//s1和u1在一个session缓存中
```

```
//异常! s1和u2在另一个session缓存中
```

■ hibernate不允许出现同一主键对象有两个不同session同时关联的情况

public Object merge(Object object)

◆ 先执行sql select

- 如果在数据库中没有找到该id, 则执行sql insert
- 如果存在, 则与object比较, 不同就执行sql update

◆ 注: 调用merge()方法, object的状态并没有被持久化, 即不在session中, 但数据库中的记录被更新了

M:N关系（二）

- ◆ 如果中间表不仅仅是做关联用的，还包含了其他字段信息
 - 需要写三个实体类
 - 解决方案1：多对多的关系拆分为两个一对多
 - ◆ 参考user-files的例子
 - 解决方案2：两张主表实体类之间的关系为多对多；中间表实体类与两张主表的关系为多对一。

Userstock表——中间表

```
◆ CREATE TABLE `userstock` ( `id`  
  int(11) NOT NULL AUTO_INCREMENT,  
  `stockid` varchar(3) DEFAULT NULL,  
  `userid` varchar(32) DEFAULT NULL,  
  `number` int(11) DEFAULT NULL, `date`  
  datetime DEFAULT NULL, PRIMARY KEY  
  (`id`), FOREIGN KEY(`stockid`)  
  REFERENCES stock(`id`), FOREIGN  
  KEY(`userid`) REFERENCES users(`id`)  
  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

解决方案2示例

- ◆ 表: users, stocks和userstock
- ◆ 中间表userstock存放了两者关系和其他字段
 - 建议手工建表
 - 仍可自动生成userstock表
 - ◆ 需设置主键生成策略
- ◆ 实体类
 - User
 - Stock
 - ◆ 修改 @JoinTable(name= “userstock”, ...)
 - 新增Trade

解决方案2示例

◆ 关系的维护端与被维护端

- 仍**选择Stock**作为关系的维护端
- **Trade**也是关系的维护端
 - ◆ **ManyToOne**
 - ◆ 设置外键和级联

Trade

.....

@Entity

@Table(name= "userstock")

public class Trade implements Serializable {

private int id;

private int number;

private Date date;

.....

private User user;

private Stock stock;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY) //使用自增主键生成策略

public int getId() {

return id;

}

public void setId(int id) {

this.id=id;

}

```
@ManyToOne(cascade=CascadeType. ALL)
@JoinColumn(name= "userid" ) //外键
```

```
public User getUser() {
    return user;
}
public void setUser(User user) {
    this.user=user;
}
```

```
@ManyToOne(cascade=CascadeType. ALL)
@JoinColumn(name= "stockid" ) //外键
```

```
public Stock getStock() {
    return stock;
}
public void setStock(Stock stock) {
    this.stock=stock;
}
.....
```

```
}
```

TradeDAO

.....

```
public interface TradeDAO extends BaseDao {  
    public void saveTrade(Trade trade);  
    .....  
}
```

TradeDAOImpl

```
public class TradeDaoImpl extends
BaseDaoImpl implements TradeDao {
    .....
    public void saveTrade(Trade trade) {
        super.save(trade)
    }
    .....
}
```

TradeManageService

.....

```
public interface TradeManageService {  
    public void InsertTrade(Trade Trade);  
    .....  
}
```

Test2TradeServlet

.....

```
Stock stock1 =stockService.getStockById("1"); //查找stock("1")
```

/*控制台:

```
Hibernate: select stock0_.id..... from stock stock0_ where  
stock0_.id=? */
```

```
User user1 =userService.getUserById( "1" ); //查找user( "1")
```


/*控制台:

```
Hibernate: select user0_.id..... from users user0_ where  
user0_.id=? */
```

.....

```
Stock stock2= stockService.getStockById("2"); //查找stock("2")
```

.....



```
Trade trade1=new Trade();
```

```
Trade trade2=new Trade();
```

```
.....
```

```
trade1.setStock(stock1);
```

```
trade1.setUser(user1);
```

```
.....
```

```
trade2.setStock(stock2);
```

```
trade2.setUser(user1);
```

```
.....
```

//trade也是**主控方**，因此：在trade表中增加1行

```
tradeService.InsertTrade(trade1);
```

```
/* Hibernate: select stock0_.id..... from stock stock0_ left  
outer join userstock users1_ on stock0_.id=users1_.stockid  
left outer join users user2_ on users1_.userid=user2_.id where  
stock0_.id=?
```

```
Hibernate: insert into userstock (date, number, stockid,  
userid) values (?, ?, ?, ?)  
*/
```

//在trade表中增加1行

```
tradeService.InsertTrade(trade2);
```