OpenDoc Series’

Hibernate 开发指南

V1.0

作者：夏昕 xiaxin(at)gmail.com

So many open source projects. Why not **Open** your **Doc**uments? **J**

文档说明

参与人员：

作者	联络
夏昕	xiaxin(at)gmail.com

(at) 为 email @ 符号

发布记录

版本	日期	作者	说明
0.9	2004.5.1	夏昕	第一版
1.0	2004.9.1	夏昕	错误修订 增加 Hibernate in Spring

OpenDoc 版权说明

如果本文对您有些许帮助，表达谢意的最好方式，是将您发现的问题和文档改进意见及时反馈给作者。当然，倘若有时间和能力，能为技术群体无偿贡献自己的所学为最好的回馈。

另外，笔者近来试图就日本、印度的软件开发模式进行一些调研。如果诸位可以赠阅日本、印度软件研发过程中的需求、设计文档以供研究，感激不尽！

Hibernate 开发指南

本文是由笔者 2003 年底一个咨询项目中，为客户做的持久层设计培训文案整理而来。

其中的内容涉及 Hibernate 的使用，以及一部分笔者实际咨询项目中的经验积累，另一方面，大部分是笔者在 Hibernate 的官方论坛中与众多技术专家交流所得。

既来于斯，则归于斯。希望能聊有所用。

本文并非试图替代 Hibernate Reference，相对而言，Hibernate Reference 的编写目的是为开发者提供更简便的条目索引，而本文目标则在于为开发人员提供一个入门和掌握 Hibernate 的途径。

本文需结合 Hibernate Reference 使用。

笔者好友曹晓钢义务组织了 Hibernate 文档的汉化工作，在此对其辛勤劳作致敬。中文版 Hibernate Reference 将被包含在 Hibernate 下个官方 Release 中，目前可通过 http://www.redsaga.com 获取中文版 Hibernate Reference 的最新版本。

Hibernate 初识

首先来看个 Quick Start 教程。

准备工作

1. 下载 Ant 软件包，解压缩（如 C:\ant\）。并将其 bin 目录（如 c:\ant\bin）添加到系统 PATH 中。

2. 下载 Hibernate、Hibernate-Extension 和 Middlegen-Hibernate 软件包的最新版本。
   http://prdownloads.sourceforge.net/hibernate/

构建 Hibernate 基础代码

Hibernate 基础代码包括：

1. POJO

POJO 在 Hibernate 语义中理解为数据库表所对应的 Domain Object。这里的 POJO 就是所谓的“**P**lain Ordinary **J**ava **O**bject”，字面上来讲就是无格式普通 Java 对象，简单的可以理解为一个不包含逻辑代码的值对象（Value Object 简称 VO）。

一个典型的 POJO：

```
public class TUser implements Serializable {

    private String name;

    public User(String name) {
        this.name = name;
    }

    /** default constructor */
    public User() {
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

2. Hibernate 映射文件

Hibernate 从本质上来讲是一种“对象－关系型数据映射”（**O**bject **R**elational **M**apping 简称 ORM）。前面的 POJO 在这里体现的就是 ORM 中 Object 层的语义，而映射（Mapping）文件则是将对象（Object）与关系型数据（**R**elational）相关联

的纽带，在 Hibernate 中，映射文件通常以“.hbm.xml”作为后缀。

构建 Hibernate 基础代码通常有以下途径：

1. 手工编写

2. 直接从数据库中导出表结构，并生成对应的 ORM 文件和 Java 代码。
   这是实际开发中最常用的方式，也是这里所推荐的方式。
   通过直接从目标数据库中导出数据结构，最小化了手工编码和调整的可能性，从而最大程度上保证了 ORM 文件和 Java 代码与实际数据库结构相一致。

3. 根据现有的 Java 代码生成对应的映射文件，将 Java 代码与数据库表相绑定。
   通过预先编写好的 POJO 生成映射文件，这种方式在实际开发中也经常使用，特别是结合了 xdoclet 之后显得尤为灵活，其潜在问题就是与实际数据库结构之间可能出现的同步上的障碍，由于需要手工调整代码，往往调整的过程中由于手工操作的疏漏，导致最后生成的配置文件错误，这点需要在开发中特别注意。
   结合 xdoclet，由 POJO 生成映射文件的技术我们将在“高级特性”章节中进行探讨。

由数据库产生基础代码

通过 Hibernate 官方提供的 MiddleGen for Hibernate 和 Hibernate_Extension 工具包，我们可以很方便的根据现有数据库，导出数据库表结构，生成 ORM 和 POJO。

1) 首先，将 Middlegen-Hibernate 软件包解压缩（ 如解压缩到 C:\Middlegen\ ）。

2) 配置目标数据库参数

进入 MiddleGen 目录下的\config\database 子目录，根据我们实际采用的数据库打开对应的配置文件。如这里我们用的是 mysql 数据库，对应的就是 mysql.xml 文件。

```
<property name="database.script.file"
          value="${src.dir}/sql/${name}-mysql.sql"/>

<property name="database.driver.file"
          value="${lib.dir}/mysql.jar"/>

<property name="database.driver.classpath"
          value="${database.driver.file}"/>

<property name="database.driver"
          value="org.gjt.mm.mysql.Driver"/>

<property name="database.url"
          value="jdbc:mysql://localhost/sample"/>

<property name="database.userid"
          value="user"/>
```

```
<property name="database.password"
          value="mypass"/>

<property name="database.schema"
          value=""/>

<property name="database.catalog"
          value=""/>

<property name="jboss.datasource.mapping"
          value="mySQL"/>
```

其中下划线标准的部分是我们进行配置的内容，分别是数据 url 以及数据库用户名和密码。

3) 修改 Build.xml

修改 MiddleGen 根目录下的 build.xml 文件，此文件是 Middlegen-Hibernate 的 Ant 构建配置。Middlegen-Hibernate 将根据 build.xml 文件中的具体参数生成数据库表映射文件。可配置的项目包括：

a) 目标数据库配置文件地址

查找关键字 "!ENTITY"，得到：

```
<!DOCTYPE project [
       <!ENTITY database SYSTEM
   "file:./config/database/hsqldb.xml">
]>
```

默认情况下，MiddleGen 采用的是 hsqldb.xml，将其修改为我们所用的数据库配置文件（mysql.xml）：

```
<!DOCTYPE project [
       <!ENTITY database SYSTEM
   "file:./config/database/mysql.xml">
]>
```

b) Application name

查找：

```
<property name="name" value="airline"/>
```

"aireline"是 MiddleGen 原始配置中默认的 Application Name，将其修改为我们所希望的名称，如"HibernateSample"：

```
<property name="name" value="HibernateSample"/>
```

c)输出目录

查找关键字"name="build.gen-src.dir""，得到：

```
<property name="build.gen-src.dir"
          value="${build.dir}/gen-src"/>
```

修改 value="${build.dir}/gen-src"使其指向我们所期望的输出目录，这里我们修改为：

```
<property name="build.gen-src.dir"
          value="C:\sample"/>
```

d)对应代码的 Package name

查找关键字"destination"，得到：

```
<hibernate
    destination="${build.gen-src.dir}"
    package="${name}.hibernate"
   genXDocletTags="false"
   genIntergratedCompositeKeys="false"
   javaTypeMapper=
"middlegen.plugins.hibernate.HibernateJavaTypeMapper"
/>
```

可以看到，hibernate 节点 package 属性的默认设置实际上是由前面的 Application Name （${name}）和".hibernate"组合而成，根据我们的需要，将其改为：

```
<hibernate
     destination="${build.gen-src.dir}"
       package="org.hibernate.sample"
     genXDocletTags="true"
     genIntergratedCompositeKeys="false"
     javaTypeMapper=
"middlegen.plugins.hibernate.HibernateJavaTypeMapper"
/>
```

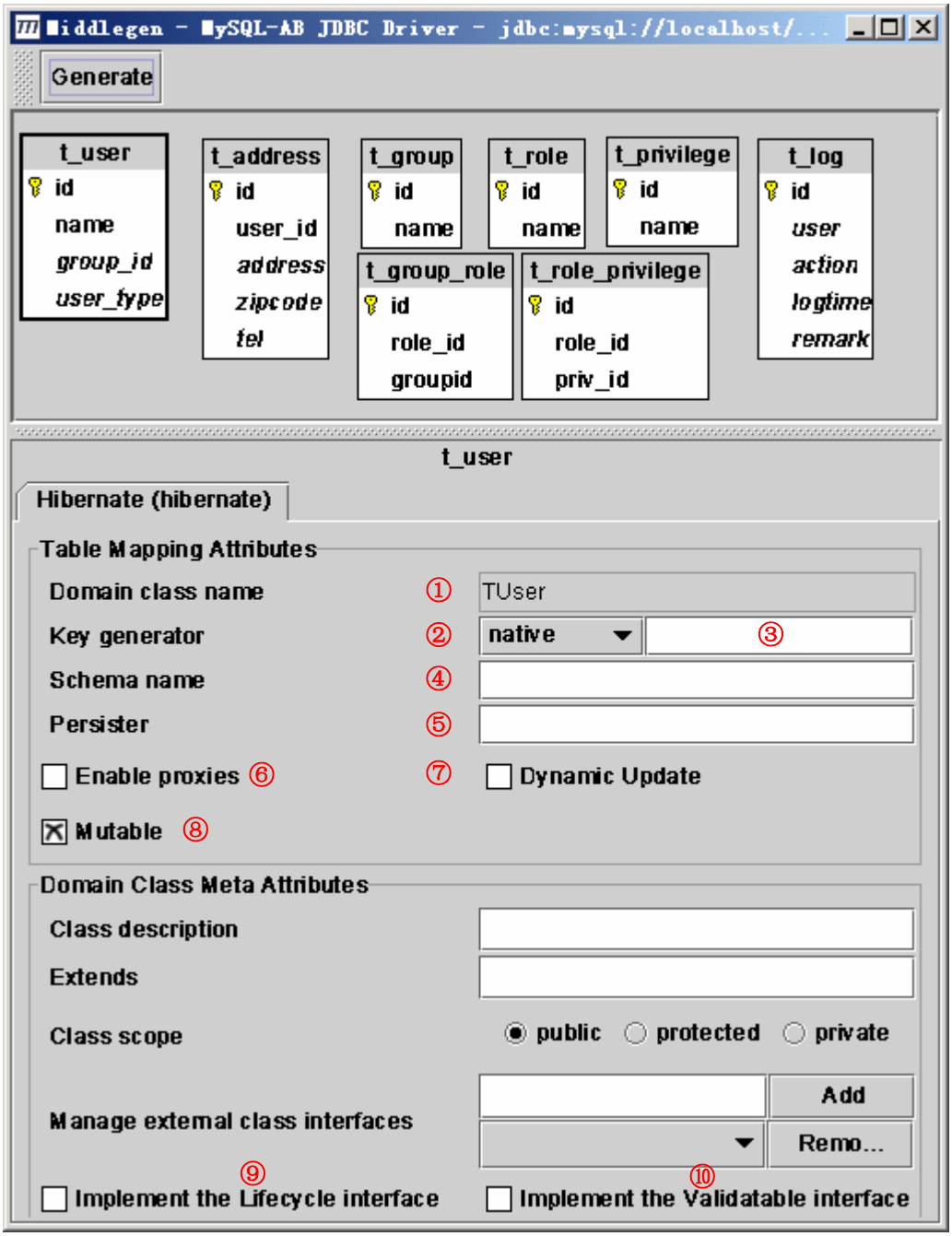
这里还有一个属性 genXDocletTags，如果设置为 true，则生成的代码将包含

xdoclet tag，这为以后在开发过程中借助 xdoclet 进行映射调整提供了帮助。关于 Hibernate 的 xdoclet 使用，请参见“高级特性”中的相关内容。

注意，如果使用的数据库为 SQLServer，需要将 build.xml 中如下部分（下划线部分）删除，否则 Middlegen 会报出找不到表的错误。

```
<middlegen
  appname="${name}"
  prefsdir="${src.dir}"
  gui="${gui}"
  databaseurl="${database.url}"
  initialContextFactory="${java.naming.factory.initial}"
  providerURL="${java.naming.provider.url}"
  datasourceJNDIName="${datasource.jndi.name}"
  driver="${database.driver}"
  username="${database.userid}"
  password="${database.password}"
  schema="${database.schema}"
  catalog="${database.catalog}"
>
```

至此为止，MiddleGen 已经配置完毕，在 MiddleGen 根目录下运行 ant，就将出现 MiddleGen 的界面：



可以看到，数据库中的表结构已经导入到 MiddleGen 的操作界面中，选定数据库表视图中的表元素，我们即可调整各个数据库表的属性。

1 Domain Class Name

对应 POJO 的类名

2 Key Generator

主键产生器

可选项说明：

1) Assigned

主键由外部程序负责生成，无需 Hibernate 参与。

2) hilo
通过 hi/lo 算法实现的主键生成机制，需要额外的数据库表保存主键生成历史状态。

3) seqhilo
与 hilo 类似，通过 hi/lo 算法实现的主键生成机制，只是主键历史状态保存在 Sequence 中，适用于支持 Sequence 的数据库，如 Oracle。

4) increment
主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存着当前的最大值，之后每次需要生成主键的时候将此值加 1 作为主键。
这种方式可能产生的问题是：如果当前有多个实例访问同一个数据库，那么由于各个实例各自维护主键状态，不同实例可能生成同样的主键，从而造成主键重复异常。因此，如果同一数据库有多个实例访问，此方式必须避免使用。

5) identity
采用数据库提供的主键生成机制。如 DB2、SQL Server、MySQL 中的主键生成机制。

6) sequence
采用数据库提供的 sequence 机制生成主键。如 Oralce 中的 Sequence。

7) native
由 Hibernate 根据底层数据库自行判断采用 identity、hilo、sequence 其中一种作为主键生成方式。

8) uuid.hex
由 Hibernate 基于 128 位唯一值产生算法生成 16 进制数值（编码后以长度 32 的字符串表示）作为主键。

9) uuid.string
与 uuid.hex 类似，只是生成的主键未进行编码（长度 16）。在某些数据库中可能出现问题（如 PostgreSQL）。

10) **foreign**
使用外部表的字段作为主键。

一般而言，利用 uuid.hex 方式生成主键将提供最好的性能和数据库平台适应性。

另外由于常用的数据库，如 Oracle、DB2、SQLServer、MySql 等，都提供了易用的主键生成机制（Auto-Increase 字段或者 Sequence）。我们可以在数据库提供的主键生成机制上，采用 generator-class=native 的主键生成方式。

不过值得注意的是，一些数据库提供的主键生成机制在效率上未必最佳，大量并发 insert 数据时可能会引起表之间的互锁。

数据库提供的主键生成机制，往往是通过在一个内部表中保存当前主键状态（如对于自增型主键而言，此内部表中就维护着当前的最大值和递增量），之后每次插入数据会读取这个最大值，然后加上递增量作为新记录的主键，之后再把这个新的最大值更新回内部表中，这样，一次 Insert 操作可能导致数据库内部多次表读写操作，同时伴随的还有数据的加锁解锁操作，这对性能产生了较大影响。

因此，对于并发 Insert 要求较高的系统，推荐采用 uuid.hex 作为主键生成机制。

3 如果需要采用定制的主键产生算法，则在此处配置主键生成器，主键生成器必须实现 net.sf.hibernate.id.IdentifierGenerator 接口。

4 Schema Name

数据库 Schema Name。

5 Persister

自定义持久类实现类类名。如果系统中还需要 Hibernate 之外的持久层实现机制，如通过存储过程得到目标数据集，甚至从 LDAP 中获取数据来填充我们的 POJO。

6 Enable proxies

是否使用代理（用于延迟加载[Lazy Loading]）。

7 Dynamic Update

如果选定，则生成 Update SQL 时不包含未发生变动的字段属性，这样可以在一定程度上提升 SQL 执行效能。

8 Mutable

类是否可变，默认为选定状态（可变）。如果不希望应用程序对此类对应的数据记录进行修改（如对于数据库视图），则可将取消其选定状态，之后对此类的 Delete 和 Update 操作都将失效。

9 Implement the Lifecyle interface

是否实现 Lifecyle 接口。Lifecyle 接口提供了数据固化过程中的控制机制，

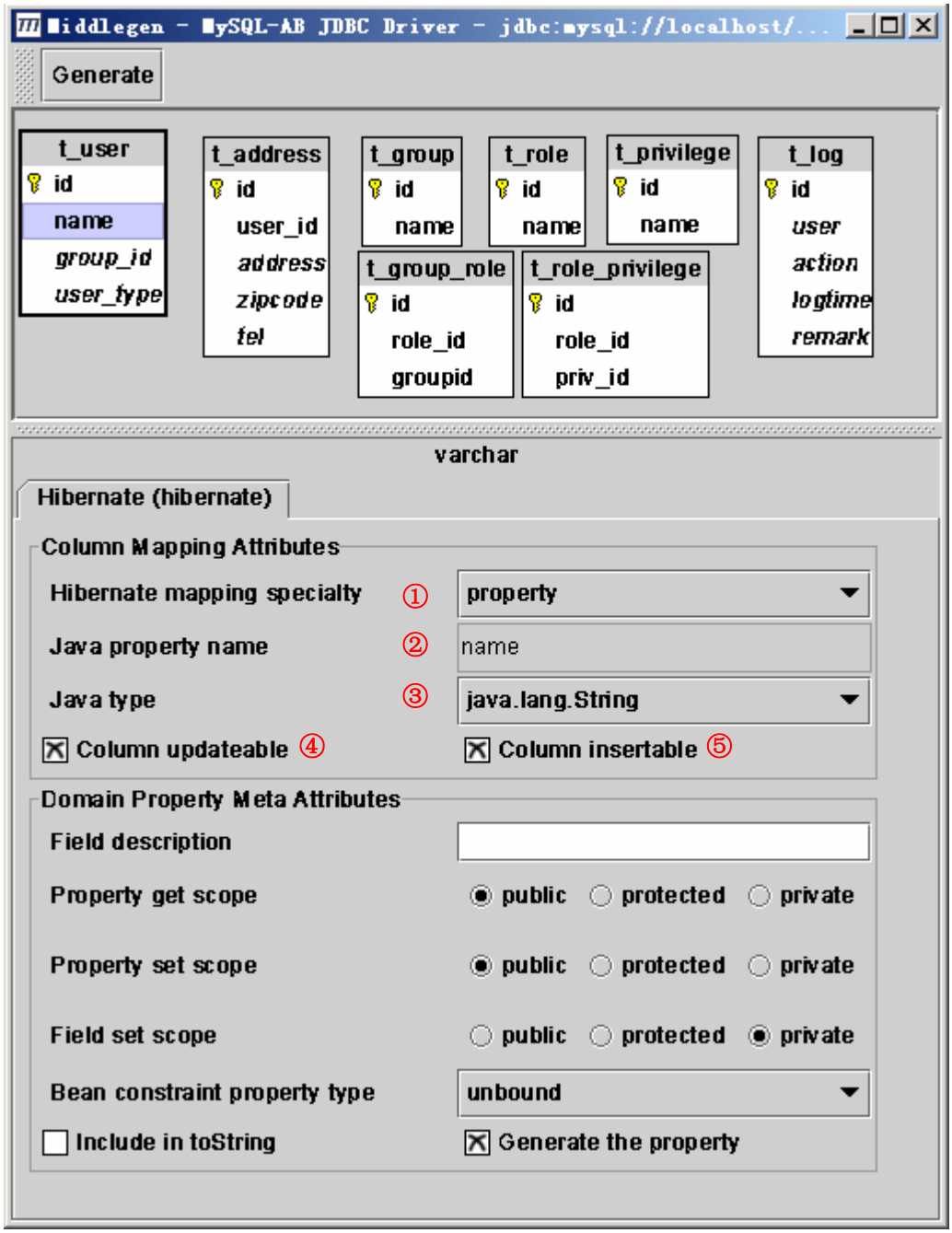
通过实现 Lifecyle 接口，我们可以在数据库操作中加入回调（Call Back）机制，如在数据库操作之前，之后触发指定操作。

10 Implement the Validatable interface

是否实现 Validatable 接口。通过实现 Validatable 接口，我们可以在数据被固化到数据库表之前对其合法性进行验证。

值得注意的是，通过实现 Lifecyle 接口，我们同样可以在数据操作之前验证数据合法性，不同的是，Validatable 接口中定义的 validate 方法可能会被调用多次，因此设计中应避免在 Validatable 接口的 validate 方法实现中加入业务逻辑的验证。

以上是针对 Class 的设置，同样，在 MiddleGen 中，我们也可以设定字段属性。在 MiddleGen 中选定某个字段，界面下方即出现字段设置栏：



在这里我们可以设置字段的属性，其中：

1 Hibernate mapping specialty

映射类型：

Key ：主键

Property ：属性

Version ：用于实现 optimistic locking，参见“高级特性”章节中关于 optimistic locking 的描述

2 Java property name

字段对应的 Java 属性名

3 Java Type
字段对应的 Java 数据类型

4 Column updateable
生成 Update SQL 时是否包含本字段。

5 Column insertable
生成 Insert SQL 时是否包含本字段。

单击窗口顶部的 Generate 按钮，MiddleGen 即为我们生成这些数据库表所对应的 Hibernate 映射文件。在 MiddleGen 根目录下的\build\gen-src\net\hibernate\sample 目录中，我们可以看到对应的以.hbm.xml 作为后缀的多个映射文件，每个映射文件都对应了数据库的一个表。

仅有映射文件还不够，我们还需要根据这些文件生成对应的 POJO。

POJO 的生成工作可以通过 Hibernate Extension 来完成，Hibernate Extension 的 tools\bin 目录下包含三个工具：

1. hbm2java.bat
根据映射文件生成对应的 POJO。通过 MiddleGen 我们已经得到了映射文件，下一步就是通过 hbm2java.bat 工具生成对应的 POJO。

2. class2hbm.bat
根据 POJO class 生成映射文件，这个工具很少用到，这里也就不再详细介绍。

3. ddl2hbm.bat
由数据库导出库表结构，并生成映射文件以及 POJO。这个功能与 MiddleGen 的功能重叠，但由于目前还不够成熟（实际上已经被废弃，不再维护），提供的功能也有限，所以我们还是采用 MiddleGen 生成映射文件，之后由 hbm2java 根据映射文件生成 POJO 的方式。

为了使用以上工具，首先我们需要配置一些参数，打开 tools\bin\setenv.bat 文件，修改其中的 JDBC_DRIVER 和 HIBERNATE_HOME 环境变量，使其指向我们的实际 JDBC Driver 文件和 Hibernate 所在目录，如

```
set JDBC_DRIVER=c:\mysql\mysql.jar
set HIBERNATE_HOME=c:\hibernate
```

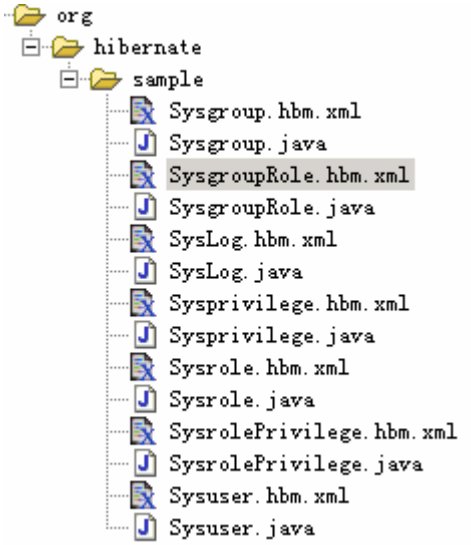
同时检查一下环境变量 CP 中的各个项目中是否实际存在，特别是%CORELIB%下的 jar 文件，某些版本的发行包中，默认配置中的文件名与实际的文件名有所出入（如 %CORELIB%\commons-logging.jar, 在 Hibernate 发行包中，可能实际的文件名是

commons-logging-1.0.3.jar，诸如此类）。

使用 hbm2java，根据 MiddleGen 生成的映射文件生成 Java 代码：
打开 Command Window，在 tools\bin 目录下执行：

```
hbm2java c:\sample\org\hibernate\sample\*.xml --output=c:\sample\
```

即可生成对应的 POJO。生成的 POJO 保存在我们指定的输出目录下（c:\sample）。



目前为止，我们已经完成了通过 MiddleGen 产生 Hibernate 基础代码的工作。配置 MiddleGen 也许并不是一件轻松的事情，对于 Eclipse 的用户而言，目前已经出现了好几个 Hibernate 的 Plugin，通过这些 Plugin 我们可以更加轻松的完成上述工作，具体的使用方式请参见附录。

Hibernate 配置

前面已经得到了映射文件和 POJO，为了使 Hibernate 能真正运作起来，我们还需要一个配置文件。

Hibernate 同时支持 xml 格式的配置文件，以及传统的 properties 文件配置方式，不过这里建议采用 xml 型配置文件。xml 配置文件提供了更易读的结构和更强的配置能力，可以直接对映射文件加以配置，而在 properties 文件中则无法配置，必须通过代码中的 Hard Coding 加载相应的映射文件。下面如果不作特别说明，都指的是基于 xml 格式文件的配置方式。

配置文件名默认为“hibernate.cfg.xml”（或者 hibernate.properties），Hibernate 初始化期间会自动在 CLASSPATH 中寻找这个文件，并读取其中的配置信息，为后期数据库操作做好准备。

配置文件应部署在 CLASSPATH 中，对于 Web 应用而言，配置文件应放置在在\WEB-INF\classes 目录下。

一个典型的 hibernate.cfg.xml 配置文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.
dtd">
<hibernate-configuration>
 <!-- SessionFactory 配置 -->
    <session-factory>
     <!-- 数据库URL -->
     <property name="hibernate.connection.url">
            jdbc:mysql://localhost/sample
     </property>

     <!-- 数据库JDBC驱动 -->
     <property name="hibernate.connection.driver_class">
            org.gjt.mm.mysql.Driver
     </property>

     <!-- 数据库用户名 -->
     <property name="hibernate.connection.username">
            User
     </property>

     <!-- 数据库用户密码 -->
     <property name="hibernate.connection.password">
```

```
            Mypass
    </property>

    <!--dialect ，每个数据库都有其对应的Dialet以匹配其平台特性 -->
       <property name="dialect">
            net.sf.hibernate.dialect.MySQLDialect
    </property>

    <!—- 是否将运行期生成的SQL输出到日志以供调试 -->
       <property name="hibernate.show_sql">
            True
    </property>

       <!—- 是否使用数据库外连接 -->
    <property name="hibernate.use_outer_join">
            True
    </property>

    <!—- 事务管理类型，这里我们使用JDBC Transaction -->
       <property name="hibernate.transaction.factory_class">
            net.sf.hibernate.transaction.JDBCTransactionFactory
    </property>

    <!—映射文件配置，注意配置文件名必须包含其相对于根的全路径 -->
       <mapping resource="net/xiaxin/xdoclet/TUser.hbm.xml"/>
       <mapping resource="net/xiaxin/xdoclet/TGroup.hbm.xml"/>

   </session-factory>
</hibernate-configuration>
```

一个典型的 hibernate.properties 配置文件如下：

```
    hibernate.dialect net.sf.hibernate.dialect.MySQLDialect
    hibernate.connection.driver_class org.gjt.mm.mysql.Driver
    hibernate.connection.driver_class com.mysql.jdbc.Driver
    hibernate.connection.url jdbc:mysql:///sample
    hibernate.connection.username user
    hibernate.connection.password mypass
```

第一段代码

上面我们已经完成了 Hiberante 的基础代码，现在先从一段最简单的代码入手，感受一下 Hibernate 所提供的强大功能。

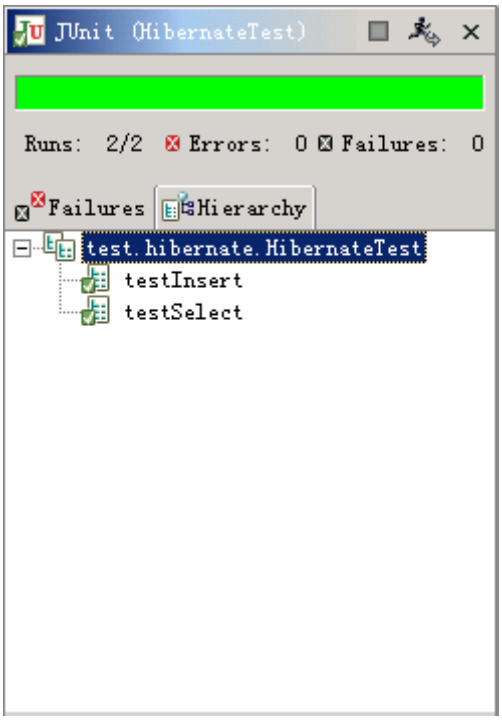
下面这段代码是一个 JUnit TestCase，演示了 TUser 对象的保存和读取。考虑到读者可能没有 JUnit 的使用经验，代码中加入了一些 JUnit 相关注释。

```
public class HibernateTest extends TestCase {

    Session session = null;

    /**
     * JUnit中setUp方法在TestCase初始化的时候会自动调用
     *  一般用于初始化公用资源
     * 此例中，用于初始化Hibernate Session
     */
    protected void setUp(){
        try {

            /**
             * 采用hibernate.properties配置文件的初始化代码：
             * Configuration config = new Configuration();
             * config.addClass(TUser.class);
             */

            //采用hibernate.cfg.xml配置文件
            //请注意初始化Configuration时的差异：
            // 1.Configuration的初始化方式
            // 2.xml文件中已经定义了Mapping文件，因此无需再Hard Coding导入
            //   POJO文件的定义
            Configuration config = new Configuration().configure();

            SessionFactory sessionFactory =
                      config.buildSessionFactory();
            session = sessionFactory.openSession();

        } catch (HibernateException e) {
            e.printStackTrace();
        }
    }


    /**
```

```
     * 与setUp方法相对应，JUnit TestCase执行完毕时，会自动调用tearDown方法
     * 一般用于资源释放
     * 此例中，用于关闭在setUp方法中打开的Hibernate Session
     */
    protected void tearDown(){
        try {
            session.close();
        } catch (HibernateException e) {
            e.printStackTrace();
        }
    }

    /**
     * 对象持久化（Insert）测试方法
     *
     * JUnit中，以"test"作为前缀的方法为测试方法，将被JUnit自动添加
     * 到测试计划中运行
     */
    public void testInsert(){

        try {
            TUser user = new TUser();
            user.setName("Emma");

            session.save(user);
            session.flush();
        } catch (HibernateException e) {
            e.printStackTrace();
            Assert.fail(e.getMessage());
        }
    }

    /**
     *  对象读取（Select）测试
     *  请保证运行之前数据库中已经存在name='Erica'的记录
     */
    public void testSelect(){
        String hql=
            " from TUser where name='Erica'";

        try {
                List userList = session.find(hql);
                TUser user =(TUser)userList.get(0);
                Assert.assertEquals(user.getName(),"Erica");
```

```
        } catch (HibernateException e) {
            e.printStackTrace();
            Assert.fail(e.getMessage());
        }
    }
}
```

主流 IDE，如 Eclipse、Intellij IDEA 和 JBuilder 中都内置了 JUnit 支持。下面是 Eclipse 中运行该代码的结果（在 Run 菜单中选择 Run as -> JUnit Test 即可）：



现在我们已经成功实现了一个简单的 TUser 实例的保存和读取。可以看到，程序中通过少量代码实现了 Java 对象和数据库数据的同步，同时借助 Hibernate 的有力支持，轻松实现了对象到关系型数据库的映射。

相对传统的 JDBC 数据访问模式，这样的实现无疑更符合面向对象的思想，同时也大大提高了开发效率。

上面的代码中引入了几个 Hibernate 基础语义：

1. Configuration
2. SessionFactory
3. Session

下面我们就这几个关键概念进行探讨。

Hibernate 基础语义

Configuration

正如其名，Configuration 类负责管理 Hibernate 的配置信息。Hibernate 运行时需要获取一些底层实现的基本信息，其中几个关键属性包括：

1. 数据库 URL
2. 数据库用户
3. 数据库用户密码
4. 数据库 JDBC 驱动类
5. 数据库 dialect，用于对特定数据库提供支持，其中包含了针对特定数据库特性的实现，如 Hibernate 数据类型到特定数据库数据类型的映射等。

使用 Hibernate 必须首先提供这些基础信息以完成初始化工作，为后继操作做好准备。这些属性在 hibernate 配置文件（hibernate.cfg.xml 或 hibernate.properties）中加以设定（参见前面“Hibernate 配置”中的示例配置文件内容）。

当我们调用：

```
Configuration config = new Configuration().configure();
```

时，Hibernate 会自动在当前的 CLASSPATH 中搜寻 hibernate.cfg.xml 文件并将其读取到内存中作为后继操作的基础配置。Configuration 类一般只有在获取 SessionFactory 时需要涉及，当获取 SessionFactory 之后，由于配置信息已经由 Hibernate 维护并绑定在返回的 SessionFactory 之上，因此一般情况下无需再对其进行操作。

我们也可以指定配置文件名，如果不希望使用默认的 hibernate.cfg.xml 文件作为配置文件的话：

```
File file = new File("c:\\sample\\myhibernate.xml");
Configuration config = new Configuration().configure(file);
```

SessionFactory

SessionFactory 负责创建 Session 实例。我们可以通过 Configuation 实例构建 SessionFactory：

```
Configuration config = new Configuration().configure();
SessionFactory sessionFactory = config.buildSessionFactory();
```

Configuration 实例 config 会根据当前的配置信息，构造 SessionFactory 实例并返回。SessionFactory 一旦构造完毕，即被赋予特定的配置信息。也就是说，之后 config 的任何变更将不会影响到已经创建的 SessionFactory 实例（`sessionFactory`）。如果需要使用基于改动后的 config 实例的 SessionFactory，需要从 config 重新构建一个 SessionFactory 实例。

Session

Session 是持久层操作的基础，相当于 JDBC 中的 Connection。

Session 实例通过 SessionFactory 实例构建：

```
Configuration config = new Configuration().configure();
SessionFactory sessionFactory = config.buildSessionFactory();
Session session = sessionFactory.openSession();
```

之后我们就可以调用 Session 所提供的 save、find、flush 等方法完成持久层操作：

Find:

```
String hql= " from TUser where name='Erica'";
List userList = session.find(hql);
```

Save:

```
TUser user = new TUser();
user.setName("Emma");
session.save(user);
session.flush();
```

最后调用Session.flush方法强制数据库同步，这里即强制Hibernate将user实例立即同步到数据库中。如果在事务中则不需要flush方法，在事务提交的时候，hibernate自动会执行flush方法，另外当Session关闭时，也会自动执行flush方法。

Hibernate 高级特性

XDoclet 与 Hibernate 映射

在 POJO 中融合 XDoclet 的映射文件自动生成机制，提供了除手动编码和由数据库导出基础代码的第三种选择。

本章将结合 XDoclet 对 Hibernate 中的数据映射进行介绍。

实际开发中，往往首先使用 MiddleGen 和 hbm2java 工具生成带有 XDoclet tag 的 POJO（MiddleGen build.xml 中的 genXDocletTags 选项决定了是否在映射文件中生成 XDoclet Tag，详见 Hibernate Quick Start 章节中关于 MiddleGen 的说明）。之后通过修改 POJO 中的 XDoclet tag 进行映射关系调整。

XDoclet 已经广泛运用在 EJB 开发中，在其最新版本里，包含了一个为 Hibernate 提供支持的子类库 Hibernate Doclet，其中包含了生成 Hibernate 映射文件所需的 ant 构建支持以及 java doc tag 支持。

XDoclet 实现基本原理是，通过在 Java 代码加入特定的 JavaDoc tag，从而为其添加特定的附加语义，之后通过 XDoclet 工具对代码中 JavaDoc Tag 进行分析，自动生成与代码对应的配置文件，XDoclet。

在 Hibernate-Doclet 中，通过引入 Hibernate 相关的 JavaDoc tag，我们就可以由代码生成对应的 Hibernate 映射文件。

下面是一个代码片断，演示了 Hibernate-Doclet 的使用方式：

```
/**
 * @hibernate.class
 *      table="TUser"
 */
public class TUser implements Serializable {
……
    /**
     * @hibernate.property
     *     column="name"
     *     length="50"
     *     not-null="true"
     *
     * @return String
     */
    public String getName() {
        return this.name;
```

```
    }

    ……
}
```

以上是使用 Hibernate-Doclet 描述 POJO（TUser）及其对应表（TUser）之间映射关系的一个例子。

其中用到了两个 hibernate doclet tag，@hibernate.class 和@hibernate.property。这两个 tag 分别描述了 POJO 所对应的数据库表信息，以及其字段对应的库表字段信息。之后 Hibernate Doclet 就会根据这些信息生成映射文件：

```
<hibernate-mapping>
    <class
        name="net.xiaxin.xdoclet.TUser"
        table="TUser"
>
    <property
        name="name"
        type="java.lang.String"
        column="name"
        not-null="true"
        length="50"
    >
</class>
</hibernate-mapping>
```

这样我们只需要维护 Java 代码，而无需再手动编写具体的映射文件即可完成 Hibernate 基础代码。

熟记 Hibernate-Doclet 众多的 Tag，显然不是件轻松的事情，好在目前的主流 IDE 都提供了 Live Template 支持。我们只需进行一些配置工作，就可以实现 Hibernate-Doclet Tag 的自动补全功能，从而避免了手工编写过程中可能出现的问题。

附录中提供了主流 IDE，包括 JBuilder，Intellij IDEA，Eclipse 的 Hibernate-Doclet 集成指南。

下面我们就 Hibernate Doclet 中常用的 Tag 进行探讨，关于 Tag 的详细参考，请参见 XDoclet 的官方指南（http://xdoclet.sourceforge.net/xdoclet/tags/hibernate-tags.html）以及 Hibernate Reference（http://www.hibernate.org）。

常用 Hibernate-Doclet Tag 介绍：

1. Class 层面：
   1) @hibernate.class

描述 POJO 与数据库表之间的映射关系，并指定相关的运行参数。

参数	描述	类型	必须
table	类对应的表名 默认值：当前类名	**Text**	**N**
dynamic-update	生成 **Update SQL** 时，仅包含发生变动的字段 默认值**: false**	**Bool**	**N**
dynamic-insert	生成 **Insert SQL** 时，仅包含非空**(null)** 字段 默认值：**false**	**Bool**	**N**
Proxy	代理类 默认值：空	**Text**	**N**
discriminator-value	子类辨别标识，用于多态支持。	**Text**	**N**
where	数据甄选条件，如果只需要处理库表中某些特定数据的时候，可通过此选项设定结果集限定条件。 如用户表中保存了全国所有用户的数据，而我们的系统只是面向上海用户，则可指定 **where="location='Shanghai'"**	**Text**	**N**

典型场景：

```
/**
 * @hibernate.class
 *      table="TUser"                                   (1)
 *      dynamic-update="true"                           (2)
 *      dynamic-insert="true"                           (3)
 *      proxy=""                                        (4)
 *      discriminator-value="1"                         (5)
 */
public class TUser implements Serializable {
   ……
}
```

本例中：

1 table 参数指定了当前类（TUser）对应数据库表“TUser”。

2 dynamic-update 参数设定为生成 Update SQL 时候，只包括当前发生变化的字段（提高 DB Update 性能）。

3 Dynamic-insert 参数设定为生成 Insert SQL 时候，只包括当前非空字段。（提高 DB Insert 性能）

4 Proxy 参数为空，表明当前类不使用代理（Proxy）。代理类的作用是为 Lazy Loading 提供支持，请参见下面关于 Lazy Loading 的有关内容。

5 discriminator-value 参数设为"1"。
discriminator-value 参数的目的是对多态提供支持。请参见下面关于 @hibernate.discriminator 的说明。

2) @hibernate.discriminator
@hibernate.discriminator（识别器） 用于提供多态支持。

参数	描述	类型	必须
column	用于区分各子类的字段名称。 默认值：当前类名	**text**	**Y**
type	对应的 **Hibernate** 类型	**Bool**	**N**
length	字段长度	**Bool**	**N**

如：

TUser类对应数据库表TUser，并且User类有两个派生类SysAdmin、SysOperator。

在TUser表中， 根据user_type字段区分用户类型。

为了让Hibernate根据user_type能自动识别对应的Class类型（如 user_type==1 则自动映射到SysAdmin类，user_type==2 则自动映射到SysOperator类），我们需要在映射文件中进行配置，而在Hibernate-Doclet中，对应的就是 @hibernate.discriminator 标识和 @hibernate.class 以及 @hibernate.subclass 的 **discriminator-value**属性。

典型场景：

```
/**
 *
 * @hibernate.class
 *      table="TUser"
 *      dynamic-update="true"
 *      dynamic-insert="true"
 *
 * @hibernate.discriminator column="user_type" type="integer"
 */
public class TUser implements Serializable {
    ……
}
```

根类 TUser 中，通过@hibernate.discriminator 指定了以"user_type"字段作为识别字段。

```
/**
 * @hibernate.subclass
 *         discriminator-value="1"
 */
public class SysAdmin extends TUser {
    ……
}
```

```
/**
 * @hibernate.subclass
 *         discriminator-value="2"
 */
public class SysOperator extends TUser {
    ……
 }
```

SysAdmin 和 SysOperator 均继承自 TUser，其 discriminator-value 分别设置为"1"和"2"，运行期 Hibernate 在读取 t_user 表数据时，会根据其 user_type 字段进行判断，如果是 1 的话则映射到 SysAdmin 类，如果是 2 映射到 SysOperator 类。

上例中，描述 SysAdmin 和 SysOperator 时，我们引入了一个 Tag：

@hibernate.subclass，顾名思义，@hibernate.subclass 与@hibernate.class 不同之处就在于，@hibernate.subclass 描述的是一个子类，实际上，这两个 Tag 除去名称不同外，并没有什么区别。

2. Method 层面：

1) @hibernate.id

描述 POJO 中关键字段与数据库表主键之间的映射关系。

参数	描述	类型	必须
column	主键字段名 默认值：当前类名	**Text**	**N**
type	字段类型。 **Hibernate** 总是使用对象型数据类型作为字段类型，如 **int** 对应 **Integer**，因此这里将 **id** 设为基本类型**[**如 **int]**以避免对象创建的开销的思路是没有实际意义的，即使这里设置为基本类型，**Hibernate** 内部还是会使用对象型数据对其进行处理，只是返回数据的时候再转换为基本类型而已。	**Text**	**N**
length	字段长度	**Text**	**N**
unsaved-value	用于对象是否已经保存的判定值。 详见“数据访问”章节的相关讨论。	**Text**	**N**
generator-class	主键产生方式（详见 **Hibernate Quick Start** 中关于 **MiddleGen** 的相关说明） 取值可为下列值中的任意一个： **assigned** **hilo** **seqhilo** **increment** **identity** **sequence** **native** **uuid.hex** **uuid.string** **foreign**	**Text**	**Y**

2) @hibernate.property

描述 POJO 中属性与数据库表字段之间的映射关系。

参数	描述	类型	必须
column	数据库表字段名 默认值：当前类名	**Text**	**N**
type	字段类型	**Text**	**N**
length	字段长度	**Text**	**N**
not-null	字段是否允许为空	**Bool**	**N**
unique	字段是否唯一（是否允许重复值）	**Bool**	**N**
insert	**Insert** 操作时是否包含本字段数据 默认：**true**	**Bool**	**N**
update	**Update** 操作时是否包含本字段数据 默认：**true**	**Bool**	**N**

典型场景：

```
/**
*  @hibernate.property
*        column="name"
*        length="50"
*        not-null="true"
*
* @return String
*/
public String getName() {
    return this.name;
}
```

注意：在编写代码的时候请，对将POJO的getter/setter方法设定为public，如果设定为private，Hibernate将无法对属性的存取进行优化，只能转而采用传统的反射机制进行操作，这将导致大量的性能开销（特别是在1.4之前的Sun JDK版本以及IBM JDK中，反射所带来的系统开销相当可观）。

包含XDoclet Tag的代码必须由xdoclet程序进行处理以生成对应的映射文件，xdoclet的处理模块可通过ant进行加载，下面是一个简单的hibernate xdoclet的ant构建脚本（注意实际使用时需要根据实际情况对路径和CLASSPATH设定进行调整）：

```
<?xml version="1.0"?>

<project name="Hibernate" default="hibernate" basedir=".">

    <property name="xdoclet.lib.home"
```

```
                value="C:\xdoclet-1.2.1\lib"/>

    <target name="hibernate" depends=""
      description="Generates Hibernate class descriptor files.">

        <taskdef name="hibernatedoclet"
         classname="xdoclet.modules.hibernate.HibernateDocletTask">

           <classpath>
             <fileset dir="${xdoclet.lib.home}">
                <include name="*.jar"/>
             </fileset>
           </classpath>

        </taskdef>

        <hibernatedoclet
             destdir="./src/"
             excludedtags="@version,@author,@todo"
             force="true"
             verbose="true"
             mergedir=".">

              <fileset dir="./src/">
                <include name="**/hibernate/sample/*.java"/>
              </fileset>

              <hibernate version="2.0"/>

        </hibernatedoclet>
    </target>

</project>
```

除了上面我们介绍的Hibernate Doclet Tag，其他还有：

Class层面；

@hibernate.cache

@hibernate.jcs-cache

@hibernate.joined-subclass

@hibernate.joined-subclass-key

@hibernate.query

Method层面
    @hibernate.array
    @hibernate.bag
    @hibernate.collection-cache
    @hibernate.collection-composite-element
    @hibernate.collection-element
    @hibernate.collection-index
    @hibernate.collection-jcs-cache
    @hibernate.collection-key
    @hibernate.collection-key-column
    @hibernate.collection-many-to-many
    @hibernate.collection-one-to-many
    @hibernate.column
    @hibernate.component
    @hibernate.generator-param
    @hibernate.index-many-to-many
@hibernate.list
@hibernate.many-to-one
@hibernate.map
@hibernate.one-to-one
@hibernate.primitive-array
@hibernate.set
@hibernate.timestamp
@hibernate.version

具体的Tag描述请参见XDoclet官方网站提供的Tag说明[1]。下面的Hibernate高级特性介绍中，我们也将涉及到这些Tag的实际使用。

[1] http://xdoclet.sourceforge.net/xdoclet/tags/hibernate-tags.html

数据检索

数据查询与检索是 Hibernate 中的一个亮点。相对其他 ORM 实现而言，Hibernate 提供了灵活多样的查询机制。其中包括：

1. Criteria Query
2. **H**ibernate **Q**uery **L**anguage (HQL)
3. SQL

Criteria Query

Criteria Query 通过面向对象化的设计，将数据查询条件封装为一个对象。简单来讲，Criteria Query 可以看作是传统 SQL 的对象化表示，如：

```
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("name","Erica"));
criteria.add(Expression.eq("sex",new Integer(1)));
```

这里的 criteria 实例实际上是 SQL "Select * from t_user where name='Erica' and sex=1"的封装（我们可以打开 Hibernate 的 show_sql 选项，以观察 Hibernate 在运行期生成的 SQL 语句）。

Hibernate 在运行期会根据 Criteria 中指定的查询条件（也就是上面代码中通过 criteria.add 方法添加的查询表达式）生成相应的 SQL 语句。

这种方式的特点是比较符合 Java 程序员的编码习惯，并且具备清晰的可读性。正因为此，不少 ORM 实现中都提供了类似的实现机制（如 Apache OJB）。

对于 Hibernate 的初学者，特别是对 SQL 了解有限的程序员而言，Criteria Query 无疑是上手的极佳途径，相对 HQL，Criteria Query 提供了更易于理解的查询手段，借助 IDE 的 Coding Assist 机制，Criteria 的使用几乎不用太多的学习。

Criteria 查询表达式

Criteria 本身只是一个查询容器，具体的查询条件需要通过 Criteria.add 方法添加到 Criteria 实例中。

如前例所示，Expression 对象具体描述了查询条件。针对 SQL 语法，Expression 提供了对应的查询限定机制，包括：

方法	描述
Expression.eq	对应 **SQL "field = value"** 表达式。 如 **Expression.eq("name","Erica")**
Expression.allEq	参数为一个 **Map** 对象，其中包含了多个属性－值对应关系。相当于多个 Expression.eq 关系的叠加。
Expression.gt	对应 **SQL** 中的 **"field > value "** 表达式

Expression.ge	对应 **SQL** 中的 **"field >= value"** 表达式
Expression.lt	对应 **SQL** 中的 **"field < value"** 表达式
Expression.le	对应 **SQL** 中的 **"field <= value"** 表达式
Expression.between	对应 **SQL** 中的 **"between"** 表达式 如下面的表达式表示年龄（**age**）位于 **13** 到 **50** 区间内。 `Expression.between("age",new Integer(13),new Integer(50));`
Expression.like	对应 **SQL** 中的 **"field like value"** 表达式
Expression.in	对应 **SQL** 中的 **"field in ..."** 表达式

Expression.eqProperty	用于比较两个属性之间的值，对应 **SQL** 中的 **"field = field"**。
	如：
	`Expression.eqProperty(`
	`"TUser.groupID",`
	`"TGroup.id"`
	`);`

Expression.gtProperty	用于比较两个属性之间的值，对应 **SQL** 中的 **"field > field"**。
Expression.geProperty	用于比较两个属性之间的值，对应 **SQL** 中的 **"field >= field"**。
Expression.ltProperty	用于比较两个属性之间的值，对应 **SQL** 中的 **"field < field"**。
Expression.leProperty	用于比较两个属性之间的值，对应 **SQL** 中的 **"field <= field"**。

Expression.and	**and** 关系组合。
	如：
	`Expression.and(`
	`Expression.eq("name","Erica"),`
	`Expression.eq(`
	`"sex",`
	`new Integer(1)`
	`)`
	`);`

Expression.or	**or** 关系组合。 如：

	Expression.or(Expression.eq("name","Erica"), Expression.eq("name","Emma"));
Expression.sql	作为补充，本方法提供了原生 **SQL** 语法的支持。我们可以通过这个方法直接通过 **SQL** 语句限定查询条件。 下面的代码返回所有名称以“**Erica**”起始的记录： Expression.sql(“lower({alias}.name) like lower(?)”, "Erica%", Hibernate.STRING); 其中的“{alias}”将由 Hibernate 在运行期使用当前关联的 POJO 别名替换。

注意 **Expression** 各方法中的属性名参数（如 **Express.eq** 中的第一个参数），这里所谓属性名是 **POJO** 中对应实际库表字段的属性名（大小写敏感），而非库表中的实际字段名称。

Criteria 高级特性

限定返回的记录范围

通过 criteria. setFirstResult/setMaxResults 方法可以限制一次查询返回的记录范围：

```
Criteria criteria = session.createCriteria(TUser.class);
//限定查询返回检索结果中，从第一百条结果开始的20条记录
criteria.setFirstResult(100);
criteria.setMaxResults(20);
```

对查询结果进行排序

```
//查询所有groupId=2的记录
//并分别按照姓名(顺序)和groupId（逆序）排序

Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("groupId",new Integer(2)));

criteria.addOrder(Order.asc("name"));
```

```
criteria.addOrder(Order.desc("groupId"));
```

Criteria 作为一种对象化的查询封装模式，不过由于 Hibernate 在实现过程中将精力更加集中在 HQL 查询语言上，因此 Criteria 的功能实现还没做到尽善尽美（这点上，OJB 的 Criteria 实现倒是值得借鉴），因此，在实际开发中，建议还是采用 Hibernate 官方推荐的查询封装模式：HQL。

Hibernate Query Language (HQL)

Criteria 提供了更加符合面向对象编程模式的查询封装模式。不过，HQL（**H**ibernate **Q**uery **L**anguage）提供了更加强大的功能，在官方开发手册中，也将 HQL 作为推荐的查询模式。

相对 Criteria，HQL 提供了更接近传统 SQL 语句的查询语法，也提供了更全面的特性。

最简单的一个例子：

```
String hql = "from org.hibernate.sample.TUser";
Query query = session.createQuery(hql);
List userList = query.list();
```

上面的代码将取出 TUser 的所有对应记录。

如果我们需要取出名为“Erica”的用户的记录，类似 SQL，我们可以通过 SQL 语句加以限定：

```
String hql =
  "from org.hibernate.sample.TUser as user where user.name='Erica'";
Query query = session.createQuery(hql);

List userList = query.list();
```

其中我们新引入了两个子句“as”和“where”，as 子句为类名创建了一个别名，而 where 子句指定了限定条件。

HQL 子句本身大小写无关，但是其中出现的类名和属性名必须注意大小写区分。

关于 HQL，Hibernate 官方开发手册中已经提供了极其详尽的说明和示例，详见 Hibernate 官方开发手册（Chapter 11）。

数据关联

一对一关联

配置：

Hibernate 中的一对一关联由“one-to-one”节点定义。

在我们的权限管理系统示例中，每个用户都从属于一个用户组。如用户“Erica”从属于“System Admin”组，从用户的角度出发，这就是一个典型的（单向）一对一关系。

每个用户对应一个组，这在我们的系统中反映为 TUser 到 TGroup 的 one-to-one 关系。其中 TUser 是主控方，TGroup 是被动方。

one-to-one 关系定义比较简单，只需在主控方加以定义。这里，我们的目标是由 TUser 对象获取其对应的 TGroup 对象。因此 TUser 对象是主控方，为了实现一对一关系，我们在 TUser 对象的映射文件 TUser.hbm.xml 中加入 one-to-one 节点，对 TGroup 对象进行一对一关联：

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TUser"
        table="t_user"
        dynamic-update="true"
        dynamic-insert="true"
    >
    ……
        <one-to-one
            name="group"
            class="org.hibernate.sample.TGroup"
            cascade="none"
            outer-join="auto"
            constrained="false"
        />
    ……
    </class>

</hibernate-mapping>
```

如果采用 XDoclet，则对应的 Tag 如下：

```
/**
 *  @hibernate.class
 *         table="t_user"
 *         dynamic-update="true"
```

```
 *        dynamic-insert="true"
 *
*/
public class TUser implements Serializable {
    ……
    private TGroup group;
    /**
     * @hibernate.one-to-one
     *              name="group"
     *              cascade="none"
     *              class="org.hibernate.sample.TGroup"
     *              outer-join="auto"
     * @return
     */
    public TGroup getGroup() {
        return group;
    }
    ……

}
```

one-to-one 节点有以下属性：

属性	描述	类型	必须
name	映射属性	**Text**	**N**
class	目标映射类。 注意要设为包含 **Package name** 的全路径名称。	**Text**	**N**
cascade	操作级联（**cascade**）关系。 可选值： **all：** 所有情况下均进行级联操作。 **none**：所有情况下均不进行级联操作。 **save-update:**在执行 **save-update** 时进行级联操作。 **delete**：在执行 **delete** 时进行级联操作。 级联（**cascade**）在 **Hibernate** 映射关系中是个非常重要的概念。它指的是当主控方执行操作时，关联对象（被动方）是否同步执行同一操作。如对主控对象调用 **save-update** 或 **delete** 方法时，是否同时对关联对象（被动方）进行	**Text**	**N**

	save-update 或 **delete**。 这里，当用户（**TUser**）被更新或者删除时，其所关联的组（**TGroup**）不应被修改或者删除，因此，这里的级联关系设置为 **none**。		
constrained	约束 表明主控表的主键上是否存在一个外键（**foreign key**）对其进行约束。这个选项关系到 **save**、**delete** 等方法的级联操作顺序。	**Bool**	**N**
outer-join	是否使用外联接。 **true**：总是使用 **outer-join** **false**：不使用 **outer-join** **auto(**默认**)** ：如果关联对象没有采用 **Proxy** 机制，则使用 **outer-join.**	**Text**	**N**
property-ref	关联类中用于与主控类相关联的属性名称。 默认为关联类的主键属性名。 这里我们通过主键达成一对一的关联，所以采用默认值即可。如果一对一的关联并非建立在主键之间，则可通过此参数指定关联属性。	**Text**	**N**
access	属性值的读取方式。 可选项： **field** **property**（默认） **ClassName**	**Text**	**N**

一对多关联

一对多关系在系统实现中也很常见。典型的例子就是父亲与孩子的关系。 而在我们现在的这个示例中，每个用户（TUser）都关联到多个地址(TAddress)，如一个用户可能拥有办公室地址、家庭地址等多个地址属性。这样，在系统中，就反应为一个“一对多”关联。

一对多关系分为单向一对多关系和双向一对多关系。

单向一对多关系只需在“一”方进行配置，双向一对多关系需要在关联双方均加以配置。

- **单向一对多关系**

配置：

对于主控方（TUser）：
TUser.hbm.xml:

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TUser"
        table="t_user"
        dynamic-update="true"
        dynamic-insert="true"
    >
    ……
        <set
            name="addresses"
            table="t_address"
            lazy="false"
            inverse="false"
            cascade="all"
            sort="unsorted"
             order-by="zipcode asc"
        >
              <key
                  column="user_id"
              >
              </key>

              <one-to-many
                  class="org.hibernate.sample.TAddress"
              />
        </set>
    ……
    </class>

</hibernate-mapping>
```

对应的 XDoclet Tag 如下：

```
    /**
     * @hibernate.collection-one-to-many
     *            class="org.hibernate.sample.TAddress"
     *
     * @hibernate.collection-key column="user_id"
     *
     * @hibernate.set
```

```
     *                  name="addresses"
     *                  table="t_address"
     *                  inverse="false"
     *                  cascade="all"
     *                  lazy="false"
     *                  sort="unsorted"
     *                  order-by="zipcode asc"
     *
     */
    public Set getAddresses() {
        return addresses;
    }
```

被动方（Taddress）的记录由 Hibernate 负责读取，之后存放在主控方指定的 Collection 类型属性中。

对于 one-to-many 关联关系，我们可以采用 java.util.Set（或者 net.sf.hibernate.collection.Bag）类型的 Collection，表现在 XML 映射文件中也就是<set>…</set>（或<bag>…</bag>）节点。关于 Hibernate 的 Collection 实现，请参见 Hibernate Reference.

one-to-many 节点有以下属性：

属性	描述	类型	必须
name	映射属性	**Text**	**Y**
table	目标关联数据库表。	**Text**	**Y**
lazy	是否采用延迟加载。 关于延迟加载，请参见后面相关章节。	**Text**	**N**
inverse	用于标识双向关联中的被动方一端。 **inverse=false 的一方（主控方）负责维护关联关系。** 默认值： **false**	**Bool**	**N**
cascade	操作级联（**cascade**）关系。 可选值： **all：** 所有情况下均进行级联操作。 **none**：所有情况下均不进行级联操作。 **save-update:**在执行 **save-update** 时进行级联操作。 **delete**：在执行 **delete** 时进行级联操作。	**Text**	**N**
sort	排序类型。	**Text**	**N**

	可选值： **unsorted** ： 不排序（默认） **natural** ： 自然顺序（避免与 **order-by** 搭配使用） **comparatorClass** ： 指以某个实现了 **java.util.Comparator** 接口的类作为排序算法。		
order-by	指定排序字段及其排序方式。 （**JDK1.4** 以上版本有效）。 对应 **SQL** 中的 **order by** 子句。 避免与 **sort** 的 “**natural**”模式同时使用。	**Text**	**N**
where	数据甄选条件，如果只需要处理库表中某些特定数据的时候，可通过此选项设定结果集限定条件。	**Text**	**N**
outer-join	是否使用外联接。 **true**：总是使用 **outer-join** **false**：不使用 **outer-join** **auto(**默认**)** ： 如果关联对象没有采用 **Proxy** 机制，则使用 **outer-join.**	**Text**	**N**
batch-size	采用延迟加载特性时（**Lazy Loading**）一次读入的数据数量。 此处未采用延迟加载机制，因此此属性忽略。	**Int**	**N**
access	属性值的读取方式。 可选项： **field** **property**（默认） **ClassName**	**Text**	**N**

通过单向一对多关系进行关联相对简单，但是存在一个问题。由于是单向关联，为了保持关联关系，我们只能通过主控方对被动方进行级联更新。且如果被关联方的关联字段为“NOT NULL”，当Hibernate创建或者更新关联关系时，还可能出现约束违例。

例如我们想为一个已有的用户“Erica”添加一个地址对象：

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

TAddress addr = new TAddress();
```

```
addr.setTel("1123");
addr.setZipcode("233123");
addr.setAddress("Hongkong");

user.getAddresses().add(addr);

session.save(user);//通过主控对象级联更新

tx.commit();
```

为了完成这个操作，Hibernate会分两步（两条SQL）来完成新增t_address记录的操作：

1. save(user)时：
   ```
   insert into t_address (user_id, address, zipcode, tel)
   values (null, "Hongkong", "233123", "1123")
   ```
2. tx.commit()时
   ```
   update t_address set user_id=”1”, address="Hongkong",
   zipcode="233123", tel="1123" where id=2
   ```

第一条SQL用于插入新的地址记录。

第二条SQL用于更新t_address，将user_id设置为其关联的user对象的id值。

问题就出在这里，数据库中，我们的t_address.user_id字段为“NOT NULL”型，当Hibernate执行第一条语句创建t_address记录时，试图将user_id字段的值设为null，于是引发了一个约束违例异常：

```
net.sf.hibernate.PropertyValueException: not-null property
references a null or transient value:
org.hibernate.sample.TAddress.userId
```

因为关联方向是单向，关联关系由TUser对象维持，而被关联的addr对象本身并不知道自己与哪个TUser对象相关联，也就是说，addr对象本身并不知道user_id应该设为什么数值。

因此，在保存addr时，只能先在关联字段插入一个空值。之后，再由TUser对象将自身的id值赋予关联字段addr.user_id，这个赋值操作导致addr对象属性发生变动，在事务提交时，hibernate会发现这一改变，并通过update sql将变动后的数据保存到数据库。

第一个步骤中，企图向数据库的非空字段插入空值，因此导致了约束违例。

既然TUser对象是主控方，为什么就不能自动先设置好下面的TAddress对象的关俩字段值再一次做Insert操作呢？莫名其妙？Ha，don’t ask me ,go to ask Hibernate Team**J**。

我们可以在设计的时候通过一些手段进行调整，以避免这样的约束违例，如将关联字段设为允许NULL值、直接采用数值型字段作为关联（有的时候这样的调整并不可行，很多情况下我们必须针对现有数据库结构进行开发），或者手动为关联字段属性赋一个任意非空值（即使在这里通过手工设置了正确的user_id也没有意义，hibernate还是会自动再调用一条Update语句进行更新）。

甚至我们可以将被动方的关联字段从其映射文件中剔除（如将user_id字段的映射从TAddress.hbm.xml中剔除）。这样Hibernate在生成第一条insert语句的时候就不会包含这个字段（数据库会使用字段默认值填充），如：之后update语句会根据主控方的one-to-many映射配置中的关联字段去更新被动方关联字段的内容。在我们这里的例子中，如果将user_id字段从TAddress.hbm.xml文件中剔除，Hibernate在保存数据时会生成下面几条SQL:

```
1.  insert into t_address (address, zipcode, tel) values
    ('Hongkong', '233123', '1123')
2.  update t_address set user_id=1 where id=7
```

生成第一条insert语句时，没有包含user_id字段，数据库会使用该字段的默认值（如果有的话）进行填充。因此不会引发约束违例。之后，根据第一条语句返回的记录id，再通过update语句对user_id字段进行更新。

但是，纵使采用这些权益之计，由于Hibernate实现机制中，采用了两条SQL进行一次数据插入操作，相对单条insert，几乎是两倍的性能开销，效率较低，因此，对于性能敏感的系统而言，这样的解决方案所带来的开销可能难以承受。

针对上面的情况，我们想到，如果addr对象知道如何获取user_id字段的内容，那么执行insert语句的时候直接将数据植入即可。这样不但绕开了约束违例的可能，而且还节省了一条Update语句的开销，大幅度提高了性能。

双向一对多关系的出现则解决了这个问题。它除了避免约束违例和提高性能的好处之外，还带来另外一个优点，由于建立了双向关联，我们可以在关联双方中任意一方，访问关联的另一方（如可以通过TAddress对象直接访问其关联的TUser对象），这提供了更丰富灵活的控制手段。

Ø 双向一对多关系

双向一对多关系，实际上是“单向一对多关系”与“多对一关系”的组合。也就是说我们必须在主控方配置单向一对多关系的基础上，在被控方配置多对一关系与其对应。

配置：

上面我们已经大致完成了单向方一对多关系的配置，我们只需在此基础上稍做修改，并对（t_address）的相关属性进行配置即可：

TUser.hbm.xml:

```
<hibernate-mapping>
```

```
    <class
        name="org.hibernate.sample.TUser"
        table="t_user"
        dynamic-update="true"
        dynamic-insert="true"
    >
    ……
        <set
            name="addresses"
            table="t_address"
            lazy="false"
            inverse="true"                                  ①
            cascade="all"
            sort="unsorted"
            order-by="zipcode asc"
        >

              <key
                  column="user_id"
              >
              </key>

              <one-to-many
                  class="org.hibernate.sample.TAddress"
              />
        </set>
    </class>

</hibernate-mapping>
```

① 这里与前面不同，**inverse** 被设为“**true**”，这意味着 **TUser** 不再作为主控方，而是将关联关系的维护工作交给关联对象 org.hibernate.sample.TAddress 来完成。这样 **TAddress** 对象在持久化过程中，就可以主动获取其关联的 **TUser** 对象的 **id**，并将其作为自己的 **user_id**，之后执行一次 **insert** 操作即可完成全部工作。

在 **one-to-many** 关系中，将 **many** 一方设为主动方（**inverse=false**）将有助性能的改善。（现实中也一样，如果要让胡锦涛记住全国人民的名字，估计花个几十年也不可能，但要让全国人民知道胡锦涛，可就不需要那么多时间了。**J**）

对应的 xdoclet tag 如下：

```
public class TUser implements Serializable {
    ……
    private Set addresses = new HashSet();
    ……
  /**
```

```
    * @hibernate.collection-one-to-many
    *       class="org.hibernate.sample.TAddress"
    *
    * @hibernate.collection-key column="user_id"
    *
    * @hibernate.set
    *                name="addresses"
    *                table="t_address"
    *                inverse="true"
    *                lazy="false"
    *                cascade="all"
    *                sort="unsorted"
    *                order-by="zipcode asc"
    */
   public Set getAddresses() {
       return addresses;
   }
   ……
}
```

TAddress.hbm.xml:

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TAddress"
        table="t_address"
        dynamic-update="false"
        dynamic-insert="false"
    >
    ……
        <many-to-one
            name="user"                                    ①
            class="org.hibernate.sample.TUser"
            cascade="none"
            outer-join="auto"
            update="true"
            insert="true"
            access="property"
            column="user_id"
            not-null="true"
        />
    </class>

</hibernate-mapping>
```

① 在 **TAddress** 对象中新增一个 **TUser field** “**user**”，并为其添加对应的

getter/setter 方法。同时删除原有的 user_id 属性及其映射配置，否则运行期会报字段重复映射错误：“Repeated column in mapping”。

对应 Xdoclet tag:

```
public class TAddress implements Serializable {
  ……
    private TUser user;
  ……
    /**
   * @hibernate.many-to-one
   *          name="user"
   *          column="user_id"
   *          not-null="true"
     *
     */
    public TUser getUser() {
        return this.user;
    }
  ……
}
```

再看上面那段代码片断：

```
  Criteria criteria = session.createCriteria(TUser.class);
  criteria.add(Expression.eq("name","Erica"));

  List userList = criteria.list();

  TUser user =(TUser)userList.get(0);

  Transaction tx = session.beginTransaction();

  TAddress addr = new TAddress();
  addr.setTel("1123");
  addr.setZipcode("233123");
  addr.setAddress("Hongkong");

  user.getAddresses().add(addr);

  session.save(user);//通过主控对象级联更新

  tx.commit();
```

尝试运行这段代码，结果凄凉的很，还是约束违例。

为什么会这样，我们已经配置了 TAddress 的 many-to-one 关系，这么看来似乎没什么效果……

不过，别忘了上面提到的 inverse 属性，这里我们把 TUser 的 inverse 设为“true”，即指定由对方维护关联关系，在这里也就是由 TAddress 维护关联关系。TUser 既然不再维护关联关系，那么 TAddress 的 user_id 属性它也自然不会关心，必须由 TAddress 自己去维护 user_id：

```
……
TAddress addr = new TAddress();
addr.setTel("1123");
addr.setZipcode("233123");
addr.setAddress("Hongkong");

addr.setUser(user);//设置关联的TUser对象
user.getAddresses().add(addr);

session.save(user);//级联更新
……
```

观察 Hibernate 执行过程中调用的 SQL 语句：

```
insert into t_address (user_id, address, zipcode, tel) values
(1, 'Hongkong', '233123', '1123')
```

正如我们所期望的，保存工作通过单条 Insert 语句的执行来完成。

many-to-one 节点有以下属性：

属性	描述	类型	必须
name	映射属性	**Text**	**Y**
column	关联字段。	**Text**	**N**
class	类名 默认为映射属性所属类型	**Text**	**N**
cascade	操作级联（**cascade**）关系。 可选值： **all**：所有情况下均进行级联操作。 **none**：所有情况下均不进行级联操作。 **save-update**:在执行 **save-update** 时进行级联操作。 **delete**：在执行 **delete** 时进行级联操作。	**Text**	**N**
update	是否对关联字段进行 **Update** 操作	**Bool**	**N**

	默认：**true**		
insert	是否对关联字段进行 **Insert** 操作 默认：**true**	**Bool**	**N**
outer-join	是否使用外联接。 **true**：总是使用 **outer-join** **false**：不使用 **outer-join** **auto(**默认**)** ：如果关联对象没有采用 **Proxy** 机制，则使用 **outer-join.**	**Text**	**N**
property-ref	用于与主控类相关联的属性的名称。 默认为关联类的主键属性名。 这里我们通过主键进行关联，所以采用默认值即可。如果关联并非建立在主键之间，则可通过此参数指定关联属性。	**Text**	**N**
access	属性值的读取方式。 可选项： **field** **property**（默认） **ClassName**	**Text**	**N**

级联与关联关系的差别?

多对多关联

Hibernate 关联关系中相对比较特殊的就是多对多关联，多对多关联与一对一关联和一对多关联不同，多对多关联需要另外一张映射表用于保存多对多映射信息。

由于多对多关联的性能不佳（由于引入了中间表，一次读取操作需要反复数次查询），因此在设计中应该避免大量使用。同时，在对多对关系中，应根据情况，采取延迟加载（Lazy Loading 参见后续章节）机制来避免无谓的性能开销。

在一个权限管理系统中，一个常见的多对多的映射关系就是 Group 与 Role，以及 Role 与 Privilege 之间的映射。

- Group 代表“组”（如“业务主管”）；
- Role 代表“角色”（如“出纳”、“财务”）；
- Privilege 代表某个特定资源的访问权限（如“修改财务报表”，“查询财务报表”）。

这里我们以 Group 和 Role 之间的映射为例：

- 一个 Group 中包含了多个 Role，如某个“业务主管”拥有“出纳”和“财

务”的双重角色。

Ø 而一个 Role 也可以属于不同的 Group。

配置：

在我们的实例中，TRole 和 TPrivilege 对应数据库中的 t_role、t_privilege 表。

TGroup.hbm.xml 中关于多对多关联的配置片断：

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TGroup"
        table="t_group"
        dynamic-update="false"
        dynamic-insert="false"
    >
    ……
        <set
            name="roles"
            table="t_group_role"                    ①
            lazy="false"
            inverse="false"
            cascade="save-update"                   ②
        >

              <key
                  column="group_id"                 ③
              >
              </key>

              <many-to-many
                  class="org.hibernate.sample.TRole"
                  column="role_id"                  ④
               />
        </set>

    </class>
</hibernate-mapping>
```

① 这里为 t_group 和 t_role 之间的映射表。

② 一般情况下，cascade 应该设置为“save-update”，对于多对多逻辑而言，很少出现删除一方需要级联删除所有关联数据的情况，如删除一个 Group，一般不会删除其中包含的 Role（这些 Role 可能还被其他的 Group 所引用）。反之删除 Role 一般也不会删除其所关联的所有 Group。

③ 映射表中对于 **t_group** 表记录的标识字段。

④ 映射表中对于 **t_role** 表记录的标识字段。

对应的 xdoclet tag 如下：

```
public class TGroup implements Serializable {
  ……
  private Set roles = new HashSet();
  /**
   * @hibernate.set
   *         name="roles"
   *         table="t_group_role"
   *         lazy="false"
   *         inverse="false"
   *         cascade="save-update"
   *         sort="unsorted"
   *
   * @hibernate.collection-key
   *         column="group_id"
   *
   * @hibernate.collection-many-to-many
   *         class="org.hibernate.sample.TRole"
   *         column="role_id"
   *
   */
  public Set getRoles() {
      return roles;
  }
  ……
}
```

TRole.hbm.xml 中关于多对多关联的配置片断：

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TRole"
        table="t_role"
        dynamic-update="false"
        dynamic-insert="false"
    >
    ……
        <set
```

```
            name="groups"
            table="t_group_role"
            lazy="false"
            inverse="true"
            cascade="save-update"
            sort="unsorted"
        >

            <key
                column="role_id"
            >
            </key>

            <many-to-many
                class="org.hibernate.sample.TGroup"
                column="group_id"
                outer-join="auto"
             />
        </set>
    </class>

</hibernate-mapping>
```

对应的 xdoclet 如下：

```
public class TRole implements Serializable {
    private Set groups = new HashSet();
  ……
  /**
   *
   * @hibernate.set
   *         name="groups"
   *         table="t_group_role"
   *         cascade="save-update"
   *         inverse="true"
   *         lazy="false"
   *
   * @hibernate.collection-key
   *         column="role_id"
   *
   * @hibernate.collection-many-to-many
   *         class="org.hibernate.sample.TGroup"
   *         column="group_id"
   *
   *
```

```
    */

    public Set getGroups() {
        return groups;
    }
}
```

many-to-many 节点中各个属性描述：

属性	描述	类型	必须
column	中间映射表中，关联目标表的关联字段。	**Text**	**Y**
class	类名 关联目标类。	**Text**	**Y**
outer-join	是否使用外联接。 **true**：总是使用 **outer-join** **false**：不使用 **outer-join** **auto(**默认**)** ：如果关联对象没有采用 **Proxy** 机制，则使用 **outer-join.**	**Text**	**N**

使用：

多对多关系中，由于关联关系是两张表相互引用，因此在保存关联状态时必须对双方同时保存。

```
public void testPersist(){

    TRole role1 = new TRole();
    role1.setName("Role1");

    TRole role2 = new TRole();
    role2.setName("Role2");

    TRole role3 = new TRole();
    role3.setName("Role3");

    TGroup group1 = new TGroup();
    group1.setName("group1");

    TGroup group2 = new TGroup();
    group2.setName("group2");
```

```
    TGroup group3 = new TGroup();
    group3.setName("group3");

    group1.getRoles().add(role1);
    group1.getRoles().add(role2);

    group2.getRoles().add(role2);
    group2.getRoles().add(role3);

    group3.getRoles().add(role1);
    group3.getRoles().add(role3);

    role1.getGroups().add(group1);
    role1.getGroups().add(group3);

    role2.getGroups().add(group1);
    role2.getGroups().add(group2);

    role3.getGroups().add(group2);
    role3.getGroups().add(group3);


    try {

        Transaction tx = session.beginTransaction();

        //多对多关系必须同时对关联双方进行保存
        session.save(role1);
        session.save(role2);
        session.save(role3);

        session.save(group1);
        session.save(group2);
        session.save(group3);

        tx.commit();

    } catch (Exception e) {
        e.printStackTrace();
        Assert.fail(e.getMessage());
    }
}
```

上面的代码创建 3 个 TGroup 对象和 3 个 TRole 对象，并形成了多对多关系。

数据访问

PO 和 VO

PO 即 **Persistence Object**
VO 即 **Value Object**

PO 和 **VO** 是 **Hibernate** 中两个比较关键的概念。
首先，何谓 **VO**，很简单，**VO** 就是一个简单的值对象。
如：

```
TUser user = new TUser();
user.setName("Emma");
```

这里的 **user** 就是一个 **VO**。**VO** 只是简单携带了对象的一些属性信息。

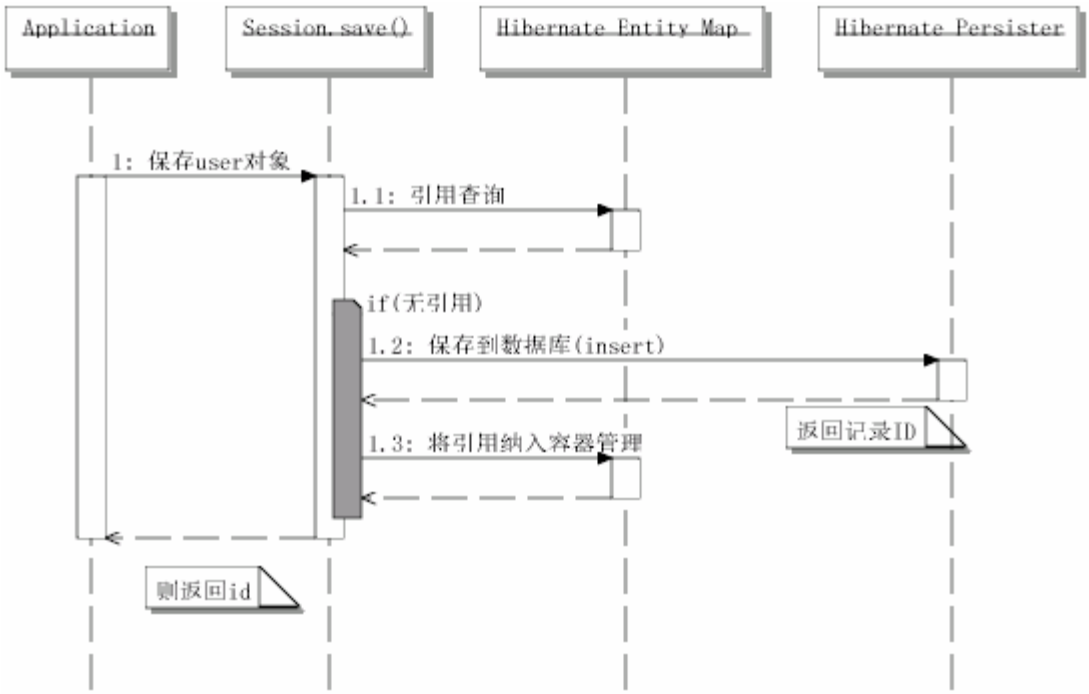
何谓 **PO?** 即纳入 **Hibernate** 管理框架中的 **VO**。看下面两个例子：

```
TUser user = new TUser();
TUser anotherUser = new TUser();

user.setName("Emma");
anotherUser.setName("Kevin");
//此时user和anotherUser都是VO

Transaction tx = session.beginTransaction();
session.save(user);
//此时的user已经经过Hibernate的处理，成为一个PO
//而anotherUser仍然是个VO
tx.commit();
//事务提交之后，库表中已经插入一条用户"Emma"的记录
//对于anotherUser则无任何操作

Transaction tx = session.beginTransaction();
user.setName("Emma_1"); //PO
anotherUser.setName("Kevin_1");//VO

tx.commit();
//事务提交之后，PO的状态被固化到数据库中
//也就是说数据库中“Emma”的用户记录已经被更新为“Emma_1”

//此时anotherUser仍然是个普通Java对象，它的属性更改不会
//对数据库产生任何影响
```

另外，通过Hibernate返回的对象也是PO:

```
//由Hibernate返回的PO
TUser user = (TUser)session.load(TUser.class,new Integer(1));
```

VO经过Hibernate进行处理，就变成了PO。
上面的示例代码session.save(user)中，我们把一个VO “user”传递给Hibernate的Session.save方法进行保存。在save方法中，Hibernate对其进行如下处理：



1. 在当前session所对应的实体容器（Entity Map）中查询是否存在user对象的引用。

2. 如果引用存在，则直接返回user对象id，save过程结束.
   Hibernate中，针对每个Session有一个实体容器（实际上是一个Map对象），如果此容器中已经保存了目标对象的引用，那么hibernate会认为此对象已经与Session相关联。
   对于save操作而言，如果对象已经与Session相关联（即已经被加入Session的实体容器中），则无需进行具体的操作。因为之后的Session.flush过程中，Hibernate会对此实体容器中的对象进行遍历，查找出发生变化的实体，生成并执行相应的update语句。

3. 如果引用不存在，则根据映射关系，执行insert操作。
   a) 在我们这里的示例中，采用了native的id生成机制，因此hibernate会从数据库取得insert操作生成的id并赋予user对象的id属性。
   b) 将user对象的引用纳入Hibernate的实体容器。
   c) save过程结束，返回对象id.

而Session.load方法中，再返回对象之前，Hibernate就已经将此对象纳入其实体容器中。

VO和PO的主要区别在于：

- VO是独立的Java Object。
- PO是由Hibernate纳入其实体容器（Entity Map）的对象，它代表了与数据库中某条记录对应的Hibernate实体，PO的变化在事务提交时将反应到实际数据库中。

如果一个PO与Session对应的实体容器中分离（如Session关闭后的PO），那么此时，它又会变成一个VO。

由PO、VO的概念，又引申出一些系统层次设计方面的问题。如在传统的MVC架构中，位于Model层的PO，是否允许被传递到其他层面。由于PO的更新最终将被映射到实际数据库中，如果PO在其他层面（如View层）发生了变动，那么可能会对Model层造成意想不到的破坏。

因此，一般而言，应该避免直接PO传递到系统中的其他层面，一种解决办法是，通过一个VO，通过属性复制使其具备与PO相同属性值，并以其为传输媒质（实际上，这个VO被用作Data Transfer Object，即所谓的DTO），将此VO传递给其他层面以实现必须的数据传送。

属性复制可以通过Apache Jakarta Commons Beanutils（http://jakarta.apache.org/commons/beanutils/）组件提供的属性批量复制功能，避免繁复的get/set操作。
下面的例子中，我们把user对象的所有属性复制到anotherUser对象中：

```
TUser user = new TUser();
TUser anotherUser = new TUser();

user.setName("Emma");
user.setUserType(1);

try {

    BeanUtils.copyProperties(anotherUser,user);

    System.out.println("UserName => "
           +anotherUser.getName()
           );
    System.out.println("UserType => "
           + anotherUser.getUserType()
           );
```

```
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
```

关于 unsaved-value

在非显示数据保存时，**Hibernate** 将根据这个值来判断对象是否需要保存。

所谓显式保存，是指代码中明确调用 **session** 的 **save**、**update**、**saveOrupdate** 方法对对象进行持久化。如：

session.save(user);

而在某些情况下，如映射关系中，**Hibernate** 根据级联（**Cascade**）关系对联接类进行保存。此时代码中没有针对级联对象的显示保存语句，需要 **Hibernate** 根据对象当前状态判断是否需要保存到数据库。此时，**Hibernate** 即将根据 **unsaved-value** 进行判定。

首先 **Hibernate** 会取出目标对象的 **id**。

之后，将此值与 **unsaved-value** 进行比对，如果相等，则认为目标对象尚未保存，否则，认为对象已经保存，无需再进行保存操作。

如：**user** 对象是之前由 **hibernate** 从数据库中获取，同时，此 **user** 对象的若干个关联对象 **address** 也被加载，此时我们向 **user** 对象新增一个 **address** 对象，此时调用 **session.save(user)**，**hibernate** 会根据 **unsaved-value** 判断 **user** 对象的数个 **address** 关联对象中，哪些需要执行 **save** 操作，而哪些不需要。

对于我们新加入的 **address** 对象而言，由于其 **id**（**Integer** 型）尚未赋值，因此为 **null**，与我们设定的 **unsaved-value**（**null**）相同，因此 **hibernate** 将其视为一个未保存对象，将为其生成 **insert** 语句并执行。

这里可能会产生一个疑问，如果“原有”关联对象发生变动（如 **user** 的某个“原有”的 **address** 对象的属性发生了变化，所谓“原有”即此 **address** 对象已经与 **user** 相关联，而不是我们在此过程中为之新增的），此时 **id** 值是从数据库中读出，并没有发生改变，自然与 **unsaved-value**（**null**）也不一样，那么 **Hibernate** 是不是就不保存了？

上面关于 **PO**、**VO** 的讨论中曾经涉及到数据保存的问题，实际上，这里的“保存”，实际上是“**insert**”的概念，只是针对新关联对象的加入，而非数据库中原有关联对象的“**update**”。所谓新关联对象，一般情况下可以理解为未与 **Session** 发生关联的 **VO**。而“原有”关联对象，则是 **PO**。如上面关于 **PO**、**VO** 的讨论中所述：

> 对于save操作而言，如果对象已经与Session相关联（即已经被加入Session的实体

容器中），则无需进行具体的操作。因为之后的Session.flush过程中，Hibernate会对此实体容器中的对象进行遍历，查找出发生变化的实体，生成并执行相应的update语句。

Inverse 和 Cascade

Inverse，直译为“反转”。在 Hibernate 语义中，Inverse 指定了关联关系中的方向。

关联关系中，inverse=”false”的为主动方，由主动方负责维护关联关系。具体可参见一对多关系中的描述。

而 Cascade，译为“级联”，表明对象的级联关系，如 TUser 的 Cascade 设为 all，就表明如果发生对 user 对象的操作，需要对 user 所关联的对象也进行同样的操作。如对 user 对象执行 save 操作，则必须对 user 对象相关联的 address 也执行 save 操作。

初学者常常混淆 inverse 和 cascade，实际上，这是两个互不相关的概念。Inverse 指的是关联关系的控制方向，而 cascade 指的是层级之间的连锁操作。

延迟加载（Lazy Loading）

为了避免一些情况下，关联关系所带来的无谓的性能开销。Hibernate引入了延迟加载的概念。

如，示例中user对象在加载的时候，会同时读取其所关联的多个地址（address）对象，对于需要对address进行操作的应用逻辑而言，关联数据的自动加载机制的确非常有效。

但是，如果我们只是想要获得user的性别（sex）属性，而不关心user的地址（address）信息，那么自动加载address的特性就显得多余，并且造成了极大的性能浪费。为了获得user的性别属性，我们可能还要同时从数据库中读取数条无用的地址数据，这导致了大量无谓的系统开销。

延迟加载特性的出现，正是为了解决这个问题。

所谓延迟加载，就是在需要数据的时候，才真正执行数据加载操作。

对于我们这里的user对象的加载过程，也就意味着，加载user对象时只针对其本身的属性，而当我们需要获取user对象所关联的address信息时（如执行user.getAddresses时），才真正从数据库中加载address数据并返回。

我们将前面一对多关系中的lazy属性修改为true，即指定了关联对象采用延迟加载：

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TUser"
        table="t_user"
        dynamic-update="true"
        dynamic-insert="true"
    >
    ……
```

```
        <set
            name="addresses"
            table="t_address"
            lazy="true"                                    ★
            inverse="false"
            cascade="all"
            sort="unsorted"
             order-by="zipcode asc"
        >
              <key
                  column="user_id"
              >
              </key>

              <one-to-many
                  class="org.hibernate.sample.TAddress"
              />
        </set>
    ……
    </class>

</hibernate-mapping>
```

尝试执行以下代码：

```
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("name","Erica"));

List userList = criteria.list();
TUser user =(TUser)userList.get(0);

System.out.println("User name => "+user.getName());

Set hset = user.getAddresses();

session.close();//关闭Session


TAddress addr = (TAddress)hset.toArray()[0];

System.out.println(addr.getAddress());
```

运行时抛出异常：

```
LazyInitializationException - Failed to lazily initialize a
```

```
collection - no session or session was closed
```

如果我们稍做调整，将session.close放在代码末尾，则不会发生这样的问题。

这意味着，只有我们实际加载user关联的address时，Hibernate才试图通过session从数据库中加载实际的数据集，而由于我们读取address之前已经关闭了session，所以报出session已关闭的错误。

这里有个问题，如果我们采用了延迟加载机制，但希望在一些情况下，实现非延迟加载时的功能，也就是说，我们希望在Session关闭后，依然允许操作user的addresses属性。如，为了向View层提供数据，我们必须提供一个完整的User对象，包含其所关联的address信息，而这个User对象必须在Session关闭之后仍然可以使用。

Hibernate.initialize方法可以通过强制加载关联对象实现这一功能：

```
Hibernate.initialize(user.getAddresses());

session.close();

//通过Hibernate.initialize方法强制读取数据
//addresses对象即可脱离session进行操作
Set  hset= user.getAddresses();
TAddress addr = (TAddress)hset.toArray()[0];
System.out.println(addr.getAddress());
```

为了实现透明化的延迟加载机制，hibernate进行了大量努力。其中包括JDK Collection接口的独立实现。

如果我们尝试用HashSet强行转化Hibernate返回的Set型对象：

```
Set hset = (HashSet)user.getAddresses();
```

就会在运行期得到一个java.lang.ClassCastException,实际上，此时返回的是一个Hibernate的特定Set实现“net.sf.hibernate.collection.Set”对象，而非传统意义上的JDK Set实现。

这也正是我们为什么在编写POJO时，必须用JDK Collection接口（如Set,Map），而非特定的JDK Collection实现类（如HashSet、HashMap）申明Collection属性的原因。

回到前面TUser类的定义：

```
public class TUser implements Serializable {
    ……
    private Set addresses = new HashSet();
    ……
```

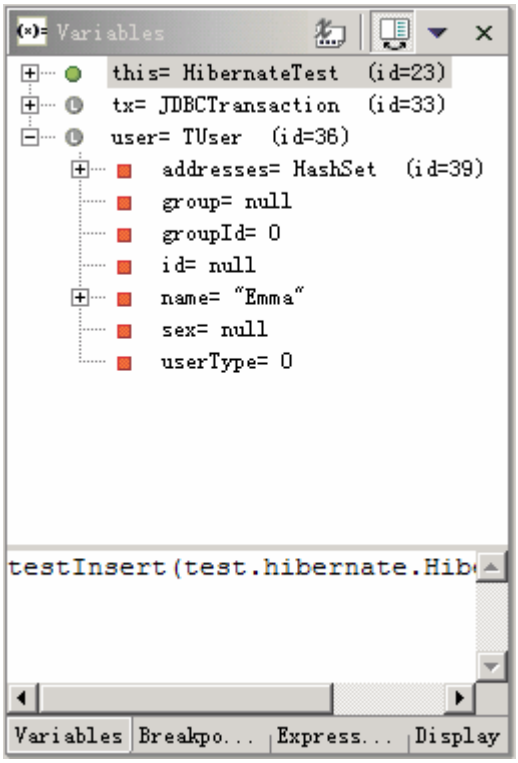
```
}
```

我们通过Set接口，申明了一个addresses属性，并创建了一个HashSet作为addresses的初始实例，以便我们创建TUser实例后，就可以为其添加关联的address对象：

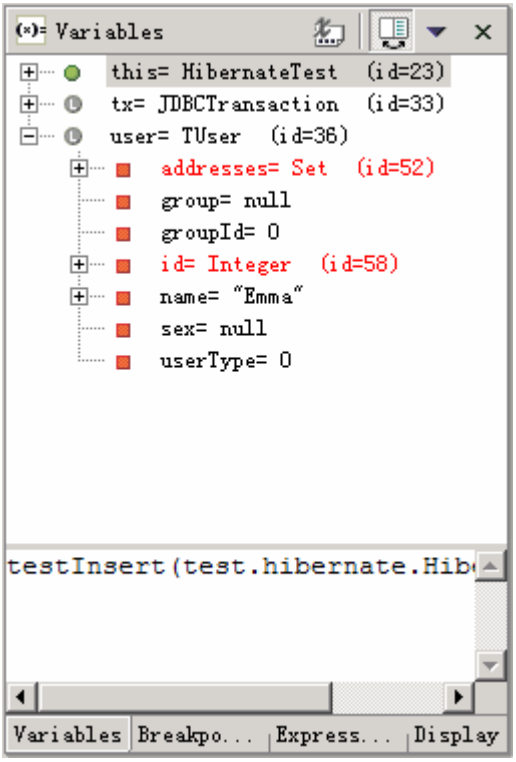
```
    TUser user = new TUser();

    TAddress addr = new TAddress();
    addr.setAddress("Hongkong");
    user.getAddresses().add(addr);

    session.save(user);
```

此时，这里的addresses属性还是一个HashSet对象，其中包含了一个address对象的引用。那么，当调用session.save(user)时，Hibernate是如何处理这个HashSet型属性的呢？

通过Eclipse的Debug窗口，我们可以看到session.save方法执行前后user对象发生的变化：



图一 session.save方法之前的user对象



图二 session.save方法之后的user对象

可以看到，user对象在通过Hibernate处理之后已经发生了变化。

首先，由于insert操作，Hibernate获得数据库产生的id值（在我们的例子中，采用native方式的主键生成机制），并填充到user对象的id属性。这个变化比较容易理解。

另一方面，Hibernate使用了自己的Collection实现"net.sf.hibernate.collection.Set"对user中的HashSet型addresses属性进行了替换，并用数据对其进行填充，保证新的addresses与原有的addresses包含同样的实体元素。

由于拥有自身的Collection实现，Hibernate就可以在Collection层从容的实现延迟加载特性。只有程序真正读取这个Collection时，才激发底层实际的数据库操作。

事务管理

Hibernate 是 JDBC 的轻量级封装，本身并不具备事务管理能力。在事务管理层，Hibernate 将其委托给底层的 JDBC 或者 JTA，以实现事务管理和调度功能。

Hibernate 的默认事务处理机制基于 JDBC Transaction。我们也可以通过配置文件设定采用 JTA 作为事务管理实现：

```
<hibernate-configuration>
   <session-factory>
     ……
      <property name="hibernate.transaction.factory_class">
         net.sf.hibernate.transaction.JTATransactionFactory
     <!--net.sf.hibernate.transaction.JDBCTransactionFactory-->
      </property>
```

```
        ……
    </session-factory>
</hibernate-configuration>
```

基于 JDBC 的事务管理

将事务管理委托给 JDBC 进行处理无疑是最简单的实现方式，Hibernate 对于 JDBC 事务的封装也极为简单。

我们来看下面这段代码：

```
session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
……
tx.commit();
```

从 JDBC 层面而言，上面的代码实际上对应着：

```
Connection dbconn = getConnection();
dbconn.setAutoCommit(false);
……
dbconn.commit();
```

就是这么简单，Hibernate 并没有做更多的事情（实际上也没法做更多的事情），只是将这样的 JDBC 代码进行了封装而已。

这里要注意的是，在 sessionFactory.openSession()中，hibernate 会初始化数据库连接，与此同时，将其 AutoCommit 设为关闭状态（false）。而其后，在 Session.beginTransaction 方法中，Hibernate 会再次确认 Connection 的 AutoCommit 属性被设为关闭状态（为了防止用户代码对 session 的 Connection.AutoCommit 属性进行修改）。

这也就是说，我们一开始从 SessionFactory 获得的 session，其自动提交属性就已经被关闭（AutoCommit=false），下面的代码将不会对数据库产生任何效果：

```
session = sessionFactory.openSession();
session.save(user);
session.close();
```

这实际上相当于 JDBC Connection 的 AutoCommit 属性被设为 false，执行了若干 JDBC 操作之后，没有调用 commit 操作即将 Connection 关闭。

如果要使代码真正作用到数据库，我们必须显式的调用 Transaction 指令：

```
session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();
session.save(user);
tx.commit();

session.close();
```

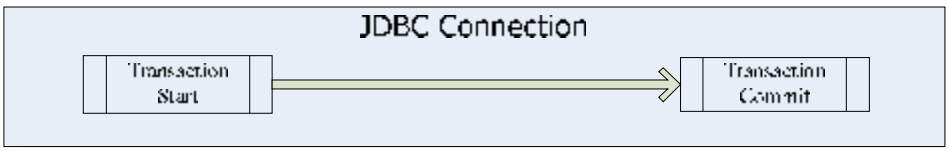
基于 JTA 的事务管理

JTA 提供了跨 Session 的事务管理能力。这一点是与 JDBC Transaction 最大的差异。

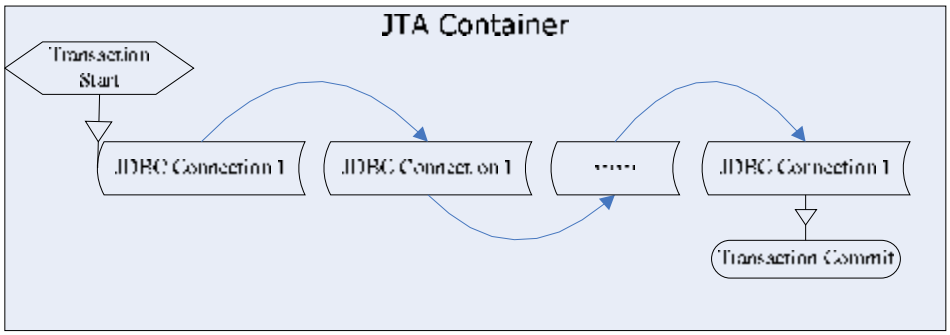
JDBC 事务由 Connnection 管理，也就是说，事务管理实际上是在 JDBC Connection 中实现。事务周期限于 Connection 的生命周期之类。同样，对于基于 JDBC Transaction 的 Hibernate 事务管理机制而言，事务管理在 Session 所依托的 JDBC Connection 中实现，事务周期限于 Session 的生命周期。

JTA 事务管理则由 JTA 容器实现，JTA 容器对当前加入事务的众多 Connection 进行调度，实现其事务性要求。JTA 的事务周期可横跨多个 JDBC Connection 生命周期。同样对于基于 JTA 事务的 Hibernate 而言，JTA 事务横跨可横跨多个 Session。

下面这幅图形象的说明了这个问题：



JDBC Transaction



JTA Transaction

图中描述的是 JDBC Connection 与事务之间的关系，而 Hibernate Session 在这里与 JDBC Connection 具备同等的逻辑含义。

从上图中我们可以看出，JTA 事务是由 JTA Container 维护，而参与事务的 Connection 无需对事务管理进行干涉。这也就是说，如果采用 JTA Transaction，我们不应该再调用 Hibernate 的 Transaction 功能。

上面基于 JDBC Transaction 的正确代码，这里就会产生问题：

```
public class ClassA{
 public void saveUser(User user){
        session = sessionFactory.openSession();

        Transaction tx = session.beginTransaction();
        session.save(user);
        tx.commit();
```

```
        session.close();
 }
}

public class ClassB{
 public void saveOrder(Order order){
        session = sessionFactory.openSession();

        Transaction tx = session.beginTransaction();
        session.save(order);
        tx.commit();

        session.close();
 }
}

public class ClassC{
 public void save(){
     ……
        UserTransaction tx = new InitialContext().lookup("……");
        ClassA.save(user);
        ClassB.save(order);
        tx.commit();
     ……
 }
}
```

这里有两个类 ClassA 和 ClassB，分别提供了两个方法：saveUser 和 saveOrder，用于保存用户信息和订单信息。在 ClassC 中，我们接连调用了 ClassA.saveUser 方法和 ClassB.saveOrder 方法，同时引入了 JTA 中的 UserTransaction 以实现 ClassC.save 方法中的事务性。

问题出现了，ClassA 和 ClassB 中分别都调用了 Hibernate 的 Transaction 功能。在 Hibernate 的 JTA 封装中，Session.beginTransaction 同样也执行了 InitialContext.lookup 方法获取 UserTransaction 实例，Transaction.commit 方法同样也调用了 UserTransaction.commit 方法。实际上，这就形成了两个嵌套式的 JTA Transaction：ClassC 申明了一个事务，而在 ClassC 事务周期内，ClassA 和 ClassB 也企图申明自己的事务，这将导致运行期错误。

因此，如果决定采用 JTA Transaction，应避免再重复调用 Hibernate 的 Transaction 功能，上面的代码修改如下：

```
public class ClassA{
```

```
 public void save(TUser user){
        session = sessionFactory.openSession();
        session.save(user);
        session.close();
 }
 ……
}

public class ClassB{
 public void save (Order order){
        session = sessionFactory.openSession();
        session.save(order);
        session.close();
 }
 ……
}

public class ClassC{
 public void save(){
     ……
        UserTransaction tx = new InitialContext().lookup("……");
        classA.save(user);
        classB.save(order);
        tx.commit();
     ……
 }
}
```

上面代码中的 ClassC.save 方法，也可以改成这样：

```
public class ClassC{
 public void save(){
     ……
        session = sessionFactory.openSession();

        Transaction tx = session.beginTransaction();

        classA.save(user);
        classB.save(order);

        tx.commit();
     ……
 }
}
```

实际上，这是利用 Hibernate 来完成启动和提交 UserTransaction 的功能，但这样的做法比原本直接通过 InitialContext 获取 UserTransaction 的做法消耗了更多的资源，得不偿失。

在 EJB 中使用 JTA Transaction 无疑最为简便，我们只需要将 save 方法配置为 JTA 事务支持即可，无需显式申明任何事务，下面是一个 Session Bean 的 save 方法，它的事务属性被申明为“Required”，EJB 容器将自动维护此方法执行过程中的事务：

```
/**
 * @ejb.interface-method
 *    view-type="remote"
 *
 * @ejb.transaction type = "Required"
**/
public void save(){
  //EJB环境中，通过部署配置即可实现事务申明，而无需显式调用事务
  classA.save(user);
  classB.save(log);

}//方法结束时，如果没有异常发生，则事务由EJB容器自动提交。
```

锁（locking）

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，我们希望针对某个 cut-off 时间点的数据进行处理，而不希望在结算进行过程中（可能是几秒种，也可能是几个小时），数据再发生变化。此时，我们就需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，在这里，也就是所谓的“锁”，即给我们选定的目标数据上锁，使其无法被其他程序修改。

Hibernate 支持两种锁机制：即通常所说的“悲观锁（Pessimistic Locking）”和“乐观锁（Optimistic Locking）”。

悲观锁（Pessimistic Locking）

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

一个典型的倚赖数据库的悲观锁调用：

```
select * from account where name=”Erica” for update
```

这条 sql 语句锁定了 account 表中所有符合检索条件（name=”Erica”）的记录。本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。

Hibernate 的悲观锁，也是基于数据库的锁机制实现。

下面的代码实现了对查询记录的加锁：

```
String hqlStr =
    "from TUser as user where user.name='Erica'";
Query query = session.createQuery(hqlStr);
query.setLockMode("user",LockMode.UPGRADE); //加锁

List userList = query.list();//执行查询，获取数据
```

query.setLockMode 对查询语句中，特定别名所对应的记录进行加锁（我们为 TUser 类指定了一个别名“user”），这里也就是对返回的所有 user 记录进行加锁。

观察运行期 Hibernate 生成的 SQL 语句：

```
select tuser0_.id as id, tuser0_.name as name, tuser0_.group_id
as group_id, tuser0_.user_type as user_type, tuser0_.sex as sex
from t_user tuser0_ where (tuser0_.name='Erica' ) for update
```

这里 Hibernate 通过使用数据库的 for update 子句实现了悲观锁机制。

Hibernate 的加锁模式有：

- LockMode.NONE ： 无锁机制。
- LockMode.WRITE ： Hibernate 在 Insert 和 Update 记录的时候会自动获取。
- LockMode.READ ： Hibernate 在读取记录的时候会自动获取。

以上这三种锁机制一般由 Hibernate 内部使用，如 Hibernate 为了保证 Update 过程中对象不会被外界修改，会在 save 方法实现中自动为目标对象加上 WRITE 锁。

- LockMode.UPGRADE ： 利用数据库的 for update 子句加锁。
- LockMode. UPGRADE_NOWAIT ：Oracle 的特定实现，利用 Oracle 的 for update nowait 子句实现加锁。

上面这两种锁机制是我们在应用层较为常用的，加锁一般通过以下方法实现：

Criteria.setLockMode
Query.setLockMode
Session.lock

注意，只有在查询开始之前（也就是 Hiberate 生成 SQL 之前）设定加锁，才会真正通过数据库的锁机制进行加锁处理，否则，数据已经通过不包含 for update 子句的 Select SQL 加载进来，所谓数据库加锁也就无从谈起。

乐观锁（Optimistic Locking）

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

如一个金融系统，当某个操作员读取用户的数据，并在读出的用户数据的基础上进行修改时（如更改用户帐户余额），如果采用悲观锁机制，也就意味着整个操作过程中（从操作员读出数据、开始修改直至提交修改结果的全过程，甚至还包括操作员中途去煮咖啡的时间），数据库记录始终处于加锁状态，可以想见，如果面对几

百上千个并发，这样的情况将导致怎样的后果。

乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（Version）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。
读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

对于上面修改用户帐户信息的例子而言，假设数据库中帐户信息表中有一个version字段，当前值为1；而当前帐户余额字段（balance）为$100。

1. 操作员 A 此时将其读出（version=1），并从其帐户余额中扣除$50（$100-$50）。
2. 在操作员A操作的过程中，操作员B也读入此用户信息（version=1），并从其帐户余额中扣除$20（$100-$20）。
3. 操作员A完成了修改工作，将数据版本号加一（version=2），连同帐户扣除后余额（balance=$50），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录version更新为2。
4. 操作员B完成了操作，也将版本号加一（version=2）试图向数据库提交数据（balance=$80），但此时比对数据库记录版本时发现，操作员B提交的数据版本号为2，数据库记录当前版本也为2，不满足“提交版本必须大于记录当前版本才能执行更新“的乐观锁策略，因此，操作员 B 的提交被驳回。这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A的操作结果的可能。

从上面的例子可以看出，乐观锁机制避免了长事务中的数据库加锁开销（操作员A和操作员B操作过程中，都没有对数据库数据加锁），大大提升了大并发量下的系统整体性能表现。
需要注意的是，乐观锁机制往往基于系统中的数据存储逻辑，因此也具备一定的局限性，如在上例中，由于乐观锁机制是在我们的系统中实现，来自外部系统的用户余额更新操作不受我们系统的控制，因此可能会造成脏数据被更新到数据库中。在系统设计阶段，我们应该充分考虑到这些情况出现的可能性，并进行相应调整（如将乐观锁策略在数据库存储过程中实现，对外只开放基于此存储过程的数据更新途径，而不是将数据库表直接对外公开）。

Hibernate 在其数据访问引擎中内置了乐观锁实现。如果不用考虑外部系统对数据库的更新操作，利用Hibernate提供的透明化乐观锁实现，将大大提升我们的生产力。

Hibernate中可以通过class描述符的optimistic-lock属性结合version描述符指定。

现在，我们为之前示例中的TUser加上乐观锁机制。

1. 首先为 TUser 的 class 描述符添加 optimistic-lock 属性：

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TUser"
        table="t_user"
        dynamic-update="true"
        dynamic-insert="true"
        optimistic-lock="version"
    >
      ……
    </class>
</hibernate-mapping>
```

optimistic-lock 属性有如下可选取值：

- Ø none
  无乐观锁
- Ø version
  通过版本机制实现乐观锁
- Ø dirty
  通过检查发生变动过的属性实现乐观锁
- Ø all
  通过检查所有属性实现乐观锁

其中通过 version 实现的乐观锁机制是 Hibernate 官方推荐的乐观锁实现，同时也是 Hibernate 中，目前唯一在数据对象脱离 Session 发生修改的情况下依然有效的锁机制。因此，一般情况下，我们都选择 version 方式作为 Hibernate 乐观锁实现机制。

2. 添加一个 Version 属性描述符

```
<hibernate-mapping>
    <class
        name="org.hibernate.sample.TUser"
        table="t_user"
        dynamic-update="true"
        dynamic-insert="true"
        optimistic-lock="version"
    >

        <id
            name="id"
            column="id"
            type="java.lang.Integer"
        >
            <generator class="native">
```

```
            </generator>
        </id>

        <version
            column="version"
            name="version"
            type="java.lang.Integer"
        />

        ……
    </class>
</hibernate-mapping>
```

注意 version 节点必须出现在 ID 节点之后。

这里我们声明了一个 version 属性，用于存放用户的版本信息，保存在 TUser 表的 version 字段中。

此时如果我们尝试编写一段代码，更新 TUser 表中记录数据，如：

```
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("name","Erica"));

List userList = criteria.list();
TUser user =(TUser)userList.get(0);

Transaction tx = session.beginTransaction();

user.setUserType(1); //更新UserType字段

tx.commit();
```

每次对 TUser 进行更新的时候，我们可以发现，数据库中的 version 都在递增。

而如果我们尝试在 tx.commit 之前，启动另外一个 Session，对名为 Erica 的用户进行操作，以模拟并发更新时的情形：

```
Session session= getSession();
Criteria criteria = session.createCriteria(TUser.class);
criteria.add(Expression.eq("name","Erica"));

Session session2 = getSession();
Criteria criteria2 = session2.createCriteria(TUser.class);
criteria2.add(Expression.eq("name","Erica"));

List userList = criteria.list();
List userList2 = criteria2.list();
```

```
    TUser user =(TUser)userList.get(0);
    TUser user2 =(TUser)userList2.get(0);

    Transaction tx = session.beginTransaction();

    Transaction tx2 = session2.beginTransaction();

    user2.setUserType(99);

    tx2.commit();

    user.setUserType(1);

    tx.commit();
```

执行以上代码，代码将在 tx.commit()处抛出 StaleObjectStateException 异常，并指出版本检查失败，当前事务正在试图提交一个过期数据。通过捕捉这个异常，我们就可以在乐观锁校验失败时进行相应处理。

Hibernate 分页

数据分页显示，在系统实现中往往带来了较大的工作量，对于基于 JDBC 的程序而言，不同数据库提供的分页（部分读取）模式往往各不相同，也带来了数据库间可移植性上的问题。

Hibernate 中，通过对不同数据库的统一接口设计，实现了透明化、通用化的分页实现机制。

我们可以通过 Criteria.setFirstResult 和 Criteria.setFetchSize 方法设定分页范围，如：

```
    Criteria criteria = session.createCriteria(TUser.class);
    criteria.add(Expression.eq("age","20"));
    //从检索结果中获取第100条记录开始的20条记录
    criteria.setFirstResult(100);
    criteria.setFetchSize(20);
```

同样，Query 接口也提供了与其一致的方法。

Hibernate 中，抽象类 net.sf.hibernate.dialect 指定了所有底层数据库的对外统一接口。通过针对不同数据库提供相应的 dialect 实现，数据库之间的差异性得以消除，从而为上层机制提供了透明的、数据库无关的存储层基础。

对于分页机制而言， dialect 中定义了一个方法如下：

```
Public String getLimitString(
                String querySelect,
```

```
                boolean hasOffset
                )
```

此方法用于在现有 Select 语句基础上，根据各数据库自身特性，构造对应的记录返回限定子句。如 MySQL 中对应的记录限定子句为 Limit，而 Oracle 中，可通过 rownum 子句实现。

我们来看 MySQLDialect 中的 getLimitString 实现：

```
public String getLimitString(String sql, boolean hasOffset) {
       return new StringBuffer( sql.length()+20 )
           .append(sql)
           .append( hasOffset ? " limit ?, ?" : " limit ?")
           .toString();
}
```

从上面可以看到，MySQLDialect.getLimitString 方法的实现实际上是在给定的 Select 语句后追加 MySQL 所提供的专有 SQL 子句 limit 来实现。

下面是 Oracle9Dialect 中的 getLimitString 实现，其中通过 Oracle 特有的 rownum 子句实现了数据的部分读取。

```
public String getLimitString(String sql, boolean hasOffset)
{
    StringBuffer pagingSelect =
         new StringBuffer( sql.length()+100 );
    if (hasOffset) {
        pagingSelect.append(
          "select * from ( select row_.*, rownum rownum_ from ( "
           );
    }else {
           pagingSelect.append("select * from ( ");
    }
    pagingSelect.append(sql);
    if (hasOffset) {
        pagingSelect.append(
              " ) row_ where rownum <= ?) where rownum_ > ?"
           );
    }else {
        pagingSelect.append(" ) where rownum <= ?");
    }
    return pagingSelect.toString();
}
```

大多数主流数据库都提供了数据部分读取机制，而对于某些没有提供相应机制的数据库而言，Hibernate 也通过其他途径实现了分页，如通过 Scrollable ResultSet，如果 JDBC 不支持 Scrollable ResultSet，Hibernate 也会自动通过 ResultSet 的 next 方法进行记录定位。这样，Hibernate 通过底层对分页机制的良好封装，使得开发人员无需关心数据分页的细节实现，将数据逻辑和存储逻辑分离开来，在提高生产效率的

同时，也大大加强了系统在不同数据库平台之间的可移植性。

Cache 管理

Cache 往往是提高系统性能的最重要的手段。在笔者记忆中，DOS 时代 SmartDrv[2]所带来的磁盘读写性能提升还历历在目（记得 95 年时安装 Windowns 3.0，在没有 SmartDrv 常驻内存的情况下，大概需要 15 分钟左右，而加载了 SmartDrv，只需要 2 分钟即可完成整个安装过程）。

Cache 对于大量倚赖数据读取操作的系统而言（典型的，如 114 查号系统）尤为重要，在大并发量的情况下，如果每次程序都需要向数据库直接做查询操作，所带来的性能开销显而易见，频繁的网络传输、数据库磁盘的读写操作（大多数数据库本身也有 Cache，但即使如此，访问数据库本身的开销也极为可观），这些都大大降低了系统的整体性能。

此时，如果能把数据在本地内存中保留一个镜像，下次访问时只需从内存中直接获取，那么显然可以带来显著的性能提升（可能是几倍，甚至几十倍的整体读取性能提升）.

引入 Cache 机制的难点是如何保证内存中数据的有效性，否则脏数据的出现将给系统带来难以预知的严重后果。

Hibernate 中实现了良好的 Cache 机制，我们可以借助 Hibernate 内部的 Cache 迅速提高系统数据读取性能。

需要注意的是：Hibernate 做为一个应用级的数据访问层封装，只能在其作用范围内保持 Cache 中数据的的有效性，也就是说，在我们的系统与第三方系统共享数据库的情况下，Hibernate 的 Cache 机制可能失效。

Hibernate 在本地 JVM 中维护了一个缓冲池，并将从数据库获得的数据保存到池中以供下次重复使用（如果在 Hibernate 中数据发生了变动，Hibernate 同样也会更新池中的数据版本）。

此时，如果有第三方系统对数据库进行了更改，那么，Hibernate 并不知道数据库中的数据已经发生了变化，也就是说，池中的数据还是修改之前的版本，下次读取时，Hibernate 会将此数据返回给上层代码，从而导致潜在的问题。

外部系统的定义，并非限于本系统之外的第三方系统，即使在本系统中，如果出现了绕过 Hibernate 数据存储机制的其他数据存取手段，那么 Cache 的有效性也必须细加考量。如，在同一套系统中，基于 Hibernate 和基于 JDBC 的两种数据访问方式并存，那么通过 JDBC 更新数据库的时候，Hibernate 同样无法获知数据更新的情况，从而导致脏数据的出现。

基于 Java 的 Cache 实现，最简单的莫过于 HashTable，hibernate 提供了基于 Hashtable 的 Cache 实现机制，不过，由于其性能和功能上的局限，仅供开发调试中使用。同时，Hibernate 还提供了面向第三方 Cache 实现的接口，如 JCS、EHCache、OSCache、JBoss Cache、SwarmCache 等。

Hibernate 中的 Cache 大致分为两层，第一层 Cache 在 Session 实现，属于事务级数据缓冲，一旦事务结束，这个 Cache 也就失效。此层 Cache 为内置实现，无需我们

[2] DOS 下的磁盘读写缓冲程序

进行干涉。

第二层 Cache，是 Hibernate 中对其实例范围内的数据进行缓存的管理容器。也是这里我们讨论的主题。

Hibernate 早期版本中采用了 JCS（Java Caching System —Apache Turbine 项目中的一个子项目）作为默认的第二层 Cache 实现。由于 JCS 的发展停顿，以及其内在的一些问题（在某些情况下，可能导致内存泄漏以及死锁），新版本的 Hibernate 已经将 JCS 去除，并用 EHCache 作为其默认的第二级 Cache 实现。

相对 JCS，EHCache 更加稳定，并具备更好的缓存调度性能，缺陷是目前还无法做到分布式缓存，如果我们的系统需要在多台设备上部署，并共享同一个数据库，必须使用支持分布式缓存的 Cache 实现（如 JCS、JBossCache）以避免出现不同系统实例之间缓存不一致而导致脏数据的情况。

Hibernate 对 Cache 进行了良好封装，透明化的 Cache 机制使得我们在上层结构的实现中无需面对繁琐的 Cache 维护细节。

目前 Hibernate 支持的 Cache 实现有：

名称	类	集群支持	查询缓冲
HashTable	**net.sf.hibernate.cache.HashtableCacheProvider**	**N**	**Y**
EHCache	**net.sf.ehcache.hibernate.Provider**	**N**	**Y**
OSCache	**net.sf.hibernate.cache.OSCacheProvider**	**N**	**Y**
SwarmCache	**net.sf.hibernate.cache.SwarmCacheProvider**	**Y**	
JBossCache	**net.sf.hibernate.cache.TreeCacheProvider**	**Y**	

其中 SwarmCache 和 JBossCache 均提供了分布式缓存实现（Cache 集群）。

(注：最新版本的 OSCache 也提供了分布式实现。)

其中 SwarmCache 提供的是 invalidation 方式的分布式缓存，即当集群中的某个节点更新了缓存中的数据，即通知集群中的其他节点将此数据废除，之后各个节点需要用到这个数据的时候，会重新从数据库中读入并填充到缓存中。

而 JBossCache 提供的是 Reapplication 式的缓冲，即如果集群中某个节点的数据发生改变，此节点会将发生改变的数据的最新版本复制到集群中的每个节点中以保持所有节点状态一致。

使用第二层 Cache，需要在 hibernate.cfg.xml 配置以下参数（以 EHCache 为例）：

```
<hibernate-configuration>
    <session-factory>
        ……
       <property name="hibernate.cache.provider_class">
          net.sf.ehcache.hibernate.Provider
        </property>
```

```
        ……
    </session-factory>

</hibernate-configuration>
```

另外还需要针对 Cache 实现本身进行配置，如 EHCache 的配置文件：

```
<ehcache>
    <diskStore path="java.io.tmpdir"/>
    <defaultCache
        maxElementsInMemory="10000"   //Cache中最大允许保存的数据数量
        eternal="false"               //Cache中数据是否为常量
        timeToIdleSeconds="120"       //缓存数据钝化时间
        timeToLiveSeconds="120"       //缓存数据的生存时间
        overflowToDisk="true"         //内存不足时，是否启用磁盘缓存
        />
</ehcache>
```

其中“//”开始的注释是笔者追加，实际配置文件中不应出现。

之后，需要在我们的映射文件中指定各个映射实体的 Cache 策略：

```
<class name=" org.hibernate.sample.TUser" ....  >
    <cache usage="read-write"/>
    ....
    <set name="addresses" .... >
        <cache usage="read-only"/>
        ....
    </set>
</class>
```

缓冲描述符 cache 可用于描述映射类和集合属性。

上例中，Class 节点下的 cache 描述符指定了针对类 TUser 的缓存策略为“read-write”，即缓冲中的 TUser 实例为可读可写，而集合属性 addresses 的缓存策略为只读。

cache usage 可选值有以下几种：

1. read-only
   只读。
2. read-write
   可读可写。
3. nonstrict-read-write
   如果程序对并发数据修改要求不是非常严格，只是偶尔需要更新数据，可以采用本选项，以减少无谓的检查，获得较好的性能。
4. transactional
   事务性 cache。在事务性 Cache 中，Cache 的相关操作也被添加到事务之中，如果由于某种原因导致事务失败，我们可以连同缓冲池中的数据一同回滚到事务开始之前的状态。目前 Hibernate 内置的 Cache 中，只有 JBossCache 支持

事务性的 Cache 实现。

不同的 Cache 实现，支持的 usage 也各不相同：

名称	read-only	read-write	nonstrict-read-write	transactional
HashTable	**Y**	**Y**	**Y**	
EHCache	**Y**	**Y**	**Y**	
OSCache	**Y**	**Y**	**Y**	
SwarmCache	**Y**	**Y**		
JBossCache	**Y**			**Y**

配置好 Cache 之后，Hibernate 在运行期会自动应用 Cache 机制，也就是说，我们对 PO 的更新，会自动同步到 Cache 中去，而数据的读取，也会自动化的优先从 Cache 中获取，对于上层逻辑代码而言，有没有应用 Cache 机制，并没有什么影响。

需要注意的是 Hibernate 的数据库查询机制。我们从查询结果中取出数据的时候，用的最多的是两个方法：

```
Query.list();
Query.iterate();
```

对于 list 方法而言，实际上 Hibernate 是通过一条 Select SQL 获取所有的记录。并将其读出，填入到 POJO 中返回。

而 iterate 方法，则是首先通过一条 Select SQL 获取所有符合查询条件的记录的 id，再对这个 id 集合进行循环操作，通过单独的 Select SQL 取出每个 id 所对应的记录，之后填入 POJO 中返回。

也就是说，对于 list 操作，需要一条 SQL 完成。而对于 iterate 操作，需要 n+1 条 SQL。

看上去 iterate 方法似乎有些多余，但在不同的情况下确依然有其独特的功效，如对海量数据的查询，如果用 list 方法将结果集一次取出，内存的开销可能无法承受。

另一方面，对于我们现在的 Cache 机制而言，list 方法将不会从 Cache 中读取数据，它总是一次性从数据库中直接读出所有符合条件的记录。而 iterate 方法因为每次根据 id 获取数据，这样的实现机制也就为从 Cache 读取数据提供了可能，hibernate 首先会根据这个 id 在本地 Cache 内寻找对应的数据，如果没找到，再去数据库中检索。如果系统设计中对 Cache 比较倚重，则请注意编码中这两种不同方法的应用组合，有针对性的改善代码，最大程度提升系统的整体性能表现。

通观以上内容，Hibernate 通过对 Cache 的封装，对上层逻辑层而言，实现了 Cache

的透明化实现，程序员编码时无需关心数据在 Cache 中的状态和调度，从而最大化协调了性能和开发效率之间的平衡。

Session 管理

无疑，Session 是 Hibernate 运作的灵魂，作为贯穿 Hibernate 应用的关键，Session 中包含了数据库操作相关的状态信息。如对 JDBC Connection 的维护，数据实体的状态维持等。

对 Session 进行有效管理的意义，类似 JDBC 程序设计中对于 JDBC Connection 的调度管理。有效的 Session 管理机制，是 Hibernate 应用设计的关键。

大多数情况下，Session 管理的目标聚焦于通过合理的设计，避免 Session 的频繁创建和销毁，从而避免大量的内存开销和频繁的 JVM 垃圾回收，保证系统高效平滑运行。

在各种 Session 管理方案中， ThreadLocal 模式得到了大量使用。ThreadLocal 是 Java 中一种较为特殊的线程绑定机制。通过 ThreadLocal 存取的数据，总是与当前线程相关，也就是说，JVM 为每个运行的线程，绑定了私有的本地实例存取空间，从而为多线程环境常出现的并发访问问题提供了一种隔离机制。

首先，我们需要知道，SessionFactory 负责创建 Session，SessionFactory 是线程安全的，多个并发线程可以同时访问一个 SessionFactory 并从中获取 Session 实例。而 Session 并非线程安全，也就是说，如果多个线程同时使用一个 Session 实例进行数据存取，则将会导致 Session 数据存取逻辑混乱。下面是一个典型的 Servlet，我们试图通过一个类变量 session 实现 Session 的重用，以避免每次操作都要重新创建：

```
public class TestServlet extends HttpServlet {

    private Session session;

    public void doGet( HttpServletRequest request,
                       HttpServletResponse response)
            throws ServletException, IOException {

      session = getSession();
      doSomething();
      session.flush();
    }

    public void doSomething(){
      ......//基于session的存取操作
    }

}
```

代码看上去正确无误，甚至在我们单机测试的时候可能也不会发生什么问题，但这样的代

码一旦编译部署到实际运行环境中，接踵而来的莫名其妙的错误很可能会使得我们摸不找头脑。问题出在哪里？

首先，Servlet 运行是多线程的，而应用服务器并不会为每个线程都创建一个 Servlet 实例，也就是说，TestServlet 在应用服务器中只有一个实例（在 Tomcat 中是这样，其他的应用服务器可能有不同的实现），而这个实例会被许多个线程并发调用，doGet 方法也将被不同的线程反复调用，可想而知，每次调用 doGet 方法，这个唯一的 TestServlet 实例的 session 变量都会被重置，线程 A 的运行过程中，其他的线程如果也被执行，那么 session 的引用将发生改变，之后线程 A 再调用 session，可能此时的 session 与其之前所用的 session 就不再一致，显然，错误也就不期而至。

ThreadLocal 的出现，使得这个问题迎刃而解。

我们对上面的例子进行一些小小的修改：

```
public class TestServlet extends HttpServlet {

    private ThreadLocal  localSession = new ThreadLocal();

    public void doGet( HttpServletRequest request,
                    HttpServletResponse response)
            throws ServletException, IOException {

        localSession.set(getSession());
        doSomething();
        session.flush();
    }

    public void doSomething(){
        Session session = (Session)localSession.get();
        ......//基于session的存取操作
    }

    }
```

可以看到，localSession 是一个 ThreadLocal 类型的对象，在 doGet 方法中，我们通过其 set 方法将获取的 session 实例保存，而在 doSomething 方法中，通过 get 方法取出 session 实例。

这也就是 ThreadLocal 的独特之处，它会为每个线程维护一个私有的变量空间。实际上，其实现原理是在 JVM 中维护一个 Map，这个 Map 的 key 就是当前的线程对象，而 value 则是线程通过 ThreadLocal.set 方法保存的对象实例。当线程调用 ThreadLocal.get 方法时，ThreadLocal 会根据当前线程对象的引用，取出 Map 中对应的对象返回。

这样，ThreadLocal 通过以各个线程对象的引用作为区分，从而将不同线程的变量隔离开来。

回到上面的例子，通过应用 ThreadLocal 机制，线程 A 的 session 实例只能为线程 A 所用，同样，其他线程的 session 实例也各自从属于自己的线程。这样，我们就实现了线程安

全的 Session 共享机制。

Hibernate 官方开发手册的示例中，提供了一个通过 ThreadLocal 维护 Session 的好榜样：

```
public class HibernateUtil {

    private static SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new
           Configuration().configure().buildSessionFactory();
        } catch (HibernateException ex) {
            throw new RuntimeException(
              "Configuration problem: " + ex.getMessage(),
              ex
              );
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException
    {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```

在代码中，只要借助上面这个工具类获取 Session 实例，我们就可以实现线程范围内的 Session 共享，从而避免了在线程中频繁的创建和销毁 Session 实例。不过注意在线程结束时关闭 Session。

同时值得一提的是，新版本的 Hibernate 在处理 Session 的时候已经内置了延迟加载机制，只有在真正发生数据库操作的时候，才会从数据库连接池获取数据库连接，我们不必过于担心 Session 的共享会导致整个线程生命周期内数据库连接被持续占用。

上面的 HibernateUtil 类可以应用在任何类型的 Java 程序中。特别的，对于 Web 程序而言，我们可以借助 Servlet2.3 规范中新引入的 Filter 机制，轻松实现线程生命周期内的 Session 管理（关于 Filter 的具体描述，请参考 Servlet2.3 规范）。

Filter 的生命周期贯穿了其所覆盖的 Servlet（JSP 也可以看作是一种特殊的 Servlet）及其底层对象。Filter 在 Servlet 被调用之前执行，在 Servlet 调用结束之后结束。因此，在 Filter 中管理 Session 对于 Web 程序而言就显得水到渠成。下面是一个通过 Filter 进行 Session 管理的典型案例：

```
public class PersistenceFilter implements Filter
{
    protected static ThreadLocal hibernateHolder = new ThreadLocal();

    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
                throws IOException, ServletException
    {
        hibernateHolder.set(getSession());
        try
        {
            ……
            chain.doFilter(request, response);
            ……
        }
        finally
        {
            Session sess = (Session)hibernateHolder.get();
            if (sess != null)
            {
                hibernateHolder.set(null);

                try
                {
                    sess.close();
                }
                catch (HibernateException ex) {
                    throw new ServletException(ex);
                 }
            }
        }
    }
    ……
```

```
}
```

通过在 doFilter 中获取和关闭 Session，并在周期内运行的所有对象（Filter 链中其余的 Filter，及其覆盖的 Servlet 和其他对象）对此 Session 实例进行重用，保证了一个 Http Request 处理过程中只占用一个 Session，提高了整体性能表现。

在实际设计中，Session 的重用做到线程级别一般已经足够，企图通过 HttpSession 实现用户级的 Session 重用反而可能导致其他的问题。凡事不能过火，Session 重用也一样。**J**

Hibernate in Spring

下面主要就**Hibernate**在**Spring**中的应用加以介绍，关于**Spring Framework**请参见笔者另外一篇文献：

《**Spring**开发指南》**http://www.xiaxin.net/Spring_Dev_Guide.rar**

Spring的参数化事务管理功能相当强大，笔者建议在基于**Spring Framework**的应用开发中，尽量使用容器管理事务，以获得数据逻辑代码的最佳可读性。下面的介绍中，将略过代码控制的事务管理部分，而将重点放在参数化的容器事务管理应用。代码级事务管理实现原理请参见《**Spring**开发指南》中的相关内容。

首先，针对**Hibernate**，我们需要进行如下配置：

Hibernate-Context.xml:

```
<beans>
  <bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
      <property name="driverClassName">
          <value>net.sourceforge.jtds.jdbc.Driver</value>
      </property>
      <property name="url">
          <value>jdbc:jtds:sqlserver://127.0.0.1:1433/Sample</value>
      </property>
      <property name="username">
          <value>test</value>
      </property>
      <property name="password">
          <value>changeit</value>
      </property>
  </bean>

  <bean id="sessionFactory"
  class="org.springframework.orm.hibernate.LocalSessionFactoryBean"
>
      <property name="dataSource">
          <ref local="dataSource" />
      </property>
      <property name="mappingResources">
          <list>
              <value>net/xiaxin/dao/entity/User.hbm.xml</value>
          </list>
      </property>
      <property name="hibernateProperties">
```

```
        <props>
            <prop key="hibernate.dialect">
                net.sf.hibernate.dialect.SQLServerDialect
            </prop>
            <prop key="hibernate.show_sql">
                true
            </prop>
        </props>
    </property>
  </bean>

  <bean id="transactionManager"
  class="org.springframework.orm.hibernate.HibernateTransactionMana
ger">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
  </bean>

  <bean id="userDAO" class="net.xiaxin.dao.UserDAO">
    <property name="sessionFactory">
        <ref local="sessionFactory" />
    </property>
  </bean>

  <bean id="userDAOProxy"
  class="org.springframework.transaction.interceptor.TransactionPro
xyFactoryBean">

    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>

    <property name="target">
        <ref local="userDAO" />
    </property>

    <property name="transactionAttributes">
        <props>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
  </bean>
```

```
</beans>
```

其中：

1. **SessionFactory**的配置

   Hibernate中通过**SessionFactory**创建和维护**Session**。**Spring**对**SessionFactory**的配置也进行了整合，无需再通过**Hibernate.cfg.xml**对**SessionFactory**进行设定。

   SessionFactory节点的**mappingResources**属性包含了映射文件的路径，**list**节点下可配置多个映射文件。

   hibernateProperties节点则容纳了所有的属性配置。

   可以对应传统的**Hibernate.cfg.xml**文件结构对这里的**SessionFactory**配置进行解读。

2. 采用面向**Hibernate**的**TransactionManager**实现：
   `org.springframework.orm.hibernate.HibernateTransactionManager`

这里引入了一个非常简单的库表：**Users**，建立如下映射类：

User.java:

```
/**
 * @hibernate.class table="users"
 */
public class User {

    public Integer id;

    public String username;

    public String password;

    /**
     * @hibernate.id
     *      column="id"
     *      type="java.lang.Integer"
     *      generator-class="native"
     */
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
```

```
    }

    /**
     * @hibernate.property column="password" length="50"
     */
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    /**
     * @hibernate.property column="username" length="50"
     */
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

上面的代码中，通过 **xdoclet** 指定了类**/**表；属性**/**字段的映射关系，通过 **xdoclet ant task** 我们可以根据代码生成对应的 **user.hbm.xml** 文件。

下面是生成的 **user.hbm.xml**：

```
<hibernate-mapping>
    <class
        name="net.xiaxin.dao.entity.User"
        table="users"
        dynamic-update="false"
        dynamic-insert="false"
    >
        <id
            name="id"
            column="id"
            type="java.lang.Integer"
        >
            <generator class="native">
            </generator>
        </id>
```

```
        <property
            name="password"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="password"
            length="50"
        />

        <property
            name="username"
            type="java.lang.String"
            update="true"
            insert="true"
            access="property"
            column="username"
            length="50"
        />
    </class>
</hibernate-mapping>
```

UserDAO.java:

```
public class UserDAO extends HibernateDaoSupport implements IUserDAO
{
    public void insertUser(User user) {
        getHibernateTemplate().saveOrUpdate(user);
    }

}
```

看到这段代码想必会有点诧异，似乎太简单了一点……，不过这已经足够。短短一行代码我们已经实现了与上一章中示例相同的功能，这也正体现了**Spring+Hibernate**的威力所在。

上面的**UserDAO**实现了自定义的**IUserDAO**接口，并扩展了抽象类：

HibernateDaoSupport

HibernateSupport实现了**HibernateTemplate**和**SessionFactory**实例的关联。

HibernateTemplate对**Hibernate Session**操作进行了封装，而**HibernateTemplate.execute**方法则是一封装机制的核心，感兴趣的读者可以研究一下其实现机制。

借助**HibernateTemplate**我们可以脱离每次数据操作必须首先获得**Session**实例、启动事务、提交**/**回滚事务以及烦杂的**try/catch/finally**等繁琐操作。从而获得以上代码中精干集中的逻辑呈现效果。

对比下面这段实现了同样功能的**Hibernate**原生代码，想必更有体会：

```
Session session
try {
  Configuration config = new Configuration().configure();

  SessionFactory sessionFactory =
      config.buildSessionFactory();
  session = sessionFactory.openSession();

  Transaction tx = session.beginTransaction();

  User user = new User();
  user.setName("erica");
    user.setPassword("mypass");
  session.save(user);

  tx.commit();


} catch (HibernateException e) {
      e.printStackTrace();
      tx.rollback();
}finally{
      session.close();
}
```

附上例的测试代码：

```
InputStream is = new FileInputStream("Hibernate-Context.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
IUserDAO userDAO = (IUserDAO)factory.getBean("userDAOProxy");

User user = new User();
user.setUsername("erica");
user.setPassword("mypass");

userDAO.insertUser(user);
```

编后赘言

Hibernate 是一个优秀的 ORM 实现，不过请注意，它只是一个 ORM 实现而已，也不能保证是最优秀的。

笔者使用过的 ORM 实现中，Apache OJB、Oracle TopLink、IBatis 和 Jaxor 都给笔者留下了深刻映像，是否选择 Hibernate 作为持久层实现，需要结合实际情况考虑（在很多情况下，比如对遗留系统的改造项目中、ibatis 可能更加合适）。

合理的设计，冷静的取舍是考量一个系统架构师功底的最实际的标准。常在网上看到对 Hibernate 以及其他一些项目或者框架标准的狂热鼓吹，不禁想起自己三年前逢项目必定为 EJB 摇旗呐喊的样子。

设计需要用时间来沉淀，建筑、服装都是如此，软件产品也一样……