hibernate 官方入门教程
第一部分 — 第一个 Hibernate 程序
首先我们将创建一个简单的控制台(console-based)Hibernate 程序。我们使用内置数据库(in-memory database) (HSQL DB)，所以我们不必安装任何数据库服务器。

让我们假设我们希望有一个小程序可以保存我们希望关注的事件（Event）和这些事件的信息。 （译者注：在本教程的后面部分，我们将直接使用 Event 而不是它的中文翻译“事件”，以免混淆。）

我们做的第一件事是建立我们的开发目录，并把所有需要用到的 Java 库文件放进去。 从 Hibernate 网站的下载页面下载 Hibernate 分发版本。解压缩包并把/lib 下面的所有库文件放到我们新的开发目录下面的/lib 目录下面。 看起来就像这样：

.+lib antlr.jar cglib-full.jar asm.jar asm-attrs.jars commons-collections.jar commons-logging.jar ehcache.jar hibernate3.jar jta.jar dom4j.jar log4j.jar
This is the minimum set of required libraries (note that we also copied hibernate3.jar, the main archive) for Hibernate. See the README.txt file in the lib/ directory of the Hibernate distribution for more information about required and optional third-party libraries. (Actually, Log4j is not required but preferred by many developers.) 这个是 Hibernate 运行所需要的最小库文件集合（注意我们也拷贝了 Hibernate3.jar，这个是最重要的库）。可以在 Hibernate 分发版本的 lib/目录下查看 README.txt，以获取更多关于所需和可选的第三方库文件信息 （事实上，Log4j 并不是必须的库文件但是许多开发者都喜欢用它）。

接下来我们创建一个类，用来代表那些我们希望储存在数据库里面的 event.

2.2.1. 第一个 class
我们的第一个持久化类是 一个简单的 JavaBean class，带有一些简单的属性（property）。 让我们来看一下代码：

```
import java.util.Date;
public class Event {
   private Long id;
   private String title;
   private Date date;

   Event() {}
   public Long getId() {
         return id;
   }
   private void setId(Long id) {
          this.id = id;
   }
   public Date getDate() {
          return date;
```

```
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

你可以看到这个 class 对属性（property）的存取方法（getter and setter method） 使用标准的 JavaBean 命名约定，同时把内部字段（field）隐藏起来（private visibility）。 这个是个受推荐的设计方式，但并不是必须这样做。 Hibernate 也可以直接访问这些字段（field），而使用访问方法（accessor method）的好处是提供了程序重构的时候健壮性（robustness）。

id 属性（property） 为一个 Event 实例提供标识属性（identifier property）的值－ 如果我们希望使用 Hibernate 的所有特性，那么我们所有的持久性实体类（persistent entity class）(这里也包括一些次要依赖类) 都需要一个标识属性（identifier property）。而事实上，大多数应用程序（特别是 web 应用程序）都需要识别特定的对象，所以你应该 考虑使用标识属性而不是把它当作一种限制。然而，我们通常不会直接操作一个对象的标识符（identifier）， 因此标识符的 setter 方法应该被声明为私有的（private）。这样当一个对象被保存的时候，只有 Hibernate 可以为它分配标识符。你会发现 Hibernate 可以直接访问被声明为 public，private 和 protected 等不同级别访问控制的方法（accessor method）和字段（field）。 所以选择哪种方式来访问属性是完全取决于你，你可以使你的选择与你的程序设计相吻合。

所有的持久类（persistent classes）都要求有无参的构造器（no-argument constructor）； 因为 Hibernate 必须要使用 Java 反射机制（Reflection）来实例化对象。构造器（constructor）的访问控制可以是私有的（private）， 然而当生成运行时代理（runtime proxy）的时候将要求使用至少是 package 级别的访问控制，这样在没有字节码编入（bytecode instrumentation）的情况下，从持久化类里获取数据会更有效率一些。

我们把这个 Java 源代码文件放到我们的开发目录下面一个叫做 src 的目录里。 这个目录现在应该看起来像这样：

```
.+lib   <Hibernate and third-party libraries>+src   Event.java
```

在下一步里，我们将把这个持久类（persisten class）的信息通知 Hibernate

2.2.2. 映射文件

Hibernate 需要知道怎样去加载（load）和存储（store）我们的持久化类的对象。这里正是 Hibernate 映射文件（mapping file）发挥作用的地方。 映射文件告诉 Hibernate 它应该访问数据库里面的哪个表（table）和应该使用表里面的哪些字段（column）。

一个映射文件的基本结构看起来像这样：

<?xml version="1.0"?><!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd"> <hibernate-mapping>[...]</hibernate-mapping>
注意 Hibernate 的 DTD 是非常复杂的。 你可以在你的编辑器或者 IDE 里面使用它来自动提示并完成（auto-completion）那些用来映射的 XML 元素（element）和属性（attribute）。 你也可以用你的文本编辑器打开 DTD－这是最简单的方式来浏览所有元素和参数，查看它们的缺省值以及它们的注释，以得到一个整体的概观。 同时也要注意 Hibernate 不会从 web 上面获取 DTD 文件，虽然 XML 里面的 URL 也许会建议它这样做，但是 Hibernate 会首先查看你的程序的 classpath。 DTD 文件被包括在 hibernate3.jar，同时也在 Hibernate 分发版的 src/ 路径下。

在以后的例子里面，我们将通过省略 DTD 的声明来缩短代码长度。但是显然，在实际的程序中，DTD 声明是必须的。

在两个 hibernate-mapping 标签（tag）中间, 我们包含了一个 class 元素（element)。所有的持久性实体类（persistent entity classes）(再次声明， 这里也包括那些依赖类，就是那些次要的实体)都需要一个这样的映射，来映射到我们的 SQL database。

<hibernate-mapping> <class name="Event" table="EVENTS"> </class></hibernate-mapping>

我们到现在为止做的一切是告诉 Hibernate 怎样从数据库表（table）EVENTS 里持久化和 加载 Event 类的对象，每个实例对应数据库里面的一行。现在我们将继续讨论有关唯一标识属性（unique identifier property）的映射。 另外，我们不希望去考虑怎样产生这个标识属性，我们将配置 Hibernate 的标识符生成策略（identifier generation strategy）来产生代用主键。

```
<hibernate-mapping>
   <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
          <generator class="increment"/>
        </id>
    </class>
</hibernate-mapping>
```

id 元素是标识属性（identifer property）的声明， name="id" 声明了 Java 属性（property）的名字 － Hibernate 将使用 getId()和 setId()来访问它。 字段参数（column attribute）则告诉 Hibernate 我们使用 EVENTS 表的哪个字段作为主键。 嵌套的 generator 元素指定了标识符的生成策略 － 在这里我们使用 increment，这个是非常简单的在内存中直接生成数字的方法，多数用于测试（或教程）中。 Hibernate 同时也支持使用数据库生成（database generated），全局唯一性（globally unique）和应用程序指定（application assigned） （或者你自己为任何已有策略所写的扩展） 这些方式来生成标识符。

最后我们还必须在映射文件里面包括需要持久化属性的声明。缺省的情况下，类里面的属性都被视为非持久化的：

```
<hibernate-mapping>
    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>
</hibernate-mapping>
```

和 id 元素类似，property 元素的 name 参数 告诉 Hibernate 使用哪个 getter 和 setter 方法。

为什么 date 属性的映射包括 column 参数，但是 title 却没有？ 当没有设定 column 参数的时候，Hibernate 缺省使用属性名作为字段（column）名。对于 title，这样工作得很好。 然而，date 在多数的数据库里，是一个保留关键字，所以我们最好把它映射成另外一个名字。

下一件有趣的事情是 title 属性缺少一个 type 参数。我们声明并使用在映射文件里面的 type，并不像我们假想的那样，是 Java data type， 同时也不是 SQL database type。这些类型被称作 Hibernate mapping types， 它们把数据类型从 Java 转换到 SQL data types。如果映射的参数没有设置的话，Hibernate 也将尝试去确定正确的类型转换和它的映射类型。 在某些情况下这个自动检测（在 Java class 上使用反射机制）不会产生你所期待或者 需要的缺省值。这里有个例子是关于 date 属性。Hibernate 无法知道这个属性应该被映射成下面这些类型中的哪一个： SQL date，timestamp，time。 我们通过声明属性映射 timestamp 来表示我们希望保存所有的关于日期和时间的信息。

这个映射文件（mapping file）应该被保存为 Event.hbm.xml，和我们的 EventJava 源文件放在同一个目录下。映射文件的名字可以是任意的，然而 hbm.xml 已经成为 Hibernate 开发者社区的习惯性约定。 现在目录应该看起来像这样：

```
.+lib  <Hibernate and third-party libraries>+src  Event.java  Event.hbm.xml
```

我们继续进行 Hibernate 的主要配置。

2.2.3. Hibernate 配置

我们现在已经有了一个持久化类和它的映射文件，是时候配置 Hibernate 了。在我们做这个之前，我们需要一个数据库。 HSQL DB，一个 java-based 内嵌式 SQL 数据库（in-memory SQL Database），可以从 HSQL DB 的网站上下载。实际上，你仅仅需要下载/lib/目录中的 hsqldb.jar，把这个文件放在开发文件夹的 lib/目录里面。

在开发目录下面创建一个叫做 data 的目录 — 这个是 HSQL DB 存储它的数据文件的地方。

Hibernate 是你的程序里连接数据库的那个应用层，所以它需要连接用的信息。连接（connection）是通过一个也由我们配置的 JDBC 连接池（connection pool）。 Hibernate 的分发版里面包括了一些 open source 的连接池，但是我们已经决定在这个教程里面使用内嵌式

连接池。 如果你希望使用一个产品级的第三方连接池软件，你必须拷贝所需的库文件去你的 classpath 并使用不同的连接池设置。

为了配置 Hibernate，我们可以使用一个简单的 hibernate.properties 文件， 或者一个稍微复杂的 hibernate.cfg.xml，甚至可以完全使用程序来配置 Hibernate。 多数用户喜欢使用 XML 配置文件：

```
<?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">org.hsqldb.jdbcDriver</property>
   <property name="connection.url">jdbc:hsqldb:data/tutorial</property>
   <property name="connection.username">sa</property>
   <property name="connection.password"></property>
   <!-- JDBC connection pool (use the built-in) -->
   <property name="connection.pool_size">1</property>
   <!-- SQL dialect -->
   <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
   <!-- Echo all executed SQL to stdout -->
   <property name="show_sql">true</property>
   <!-- Drop and re-create the database schema on startup -->
   <property name="hbm2ddl.auto">create</property>
   <mapping resource="Event.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

注意这个 XML 配置使用了一个不同的 DTD。我们配置 Hibernate 的 SessionFactory－ 一个关联于特定数据库全局性的工厂（factory）。如果你要使用多个数据库，通常应该在多个配置文件中使用多个<session-factory> 进行配置（在更早的启动步骤中进行）。

最开始的 4 个 property 元素包含必要的 JDBC 连接信息。dialectproperty 表明 Hibernate 应该产生针对特定数据库语法的 SQL 语句。hbm2ddl.auto 选项将自动生成数据库表定义（schema）－ 直接插入数据库中。当然这个选项也可以被关闭（通过去除这个选项）或者通过 Ant 任务 SchemaExport 来把数据库表定义导入一个文件中进行优化。 最后，为持久化类加入映射文件。

把这个文件拷贝到源代码目录下面，这样它就位于 classpath 的 root 路径上。Hibernate 在启动时会自动 在它的根目录开始寻找名为 hibernate.cfg.xml 的配置文件。

2.2.4. 用 Ant 编译

在这个教程里面，我们将用 Ant 来编译程序。你必须先安装 Ant－可以从 Ant download page 下载它。怎样安装 Ant 不是这个教程的内容，请参考 Ant manual。 当你安装完了 Ant，我们

就可以开始创建编译脚本，它的文件名是 build.xml，把它直接放在开发目录下面。

完善 Ant

注意 Ant 的分发版通常功能都是不完整的（就像 Ant FAQ 里面说得那样），所以你常常不得不需要自己动手来完善 Ant。 例如：如果你希望在你的 build 文件里面使用 JUnit 功能。为了让 JUnit 任务被激活（这个教程里面我们并不需要这个任务）， 你必须拷贝 junit.jar 到 ANT_HOME/lib 目录下或者删除 ANT_HOME/lib/ant-junit.jar 这个插件。

一个基本的 build 文件看起来像这样

```
<project name="hibernate-tutorial" default="compile">
<property name="sourcedir" value="${basedir}/src"/>
<property name="targetdir" value="${basedir}/bin"/>
<property name="librarydir" value="${basedir}/lib"/>
<path id="libraries">
<fileset dir="${librarydir}">
<include name="*.jar"/>
</fileset>
</path>
<target name="clean">
<delete dir="${targetdir}"/>
<mkdir dir="${targetdir}"/>
</target>
<target name="compile" depends="clean, copy-resources">
<javac srcdir="${sourcedir}" destdir="${targetdir}" classpathref="libraries"/>
</target>
<target name="copy-resources">
<copy todir="${targetdir}">
<fileset dir="${sourcedir}">
<exclude name="**/*.java"/>
</fileset>
</copy>
</target>
</project>
```

这个将告诉 Ant 把所有在 lib 目录下以.jar 结尾的文件加入 classpath 中用来进行编译。 它也将把所有的非 Java 源代码文件，例如配置和 Hibernate 映射文件，拷贝到目标目录下。如果你现在运行 Ant， 你将得到以下输出：

```
C:\hibernateTutorial\>antBuildfile: build.xmlcopy-resources: [copy] Copying 2 files to C:\hibernateTutorial\bincompile: [javac] Compiling 1 source file to C:\hibernateTutorial\binBUILD SUCCESSFULTotal time: 1 second
```

2.2.5. 安装和帮助

是时候来加载和储存一些 Event 对象了，但是首先我们不得不完成一些基础的代码。 **我们必须启动 Hibernate。这个启动过程包括创建一个全局性的 SessoinFactory 并把它储存在一**

个应用程序容易访问的地方。 SessionFactory 可以创建并打开新的 Session。 一个 Session 代表一个单线程的单元操作，SessionFactory 则是一个线程安全的全局对象，只需要创建一次。

我们将创建一个 HibernateUtil 帮助类（helper class）来负责启动 Hibernate 并使 操作 Session 变得容易。这个帮助类将使用被称为 ThreadLocal Session 的模式来保证当前的单元操作和当前线程相关联。让我们来看一眼它的实现：

```
import org.hibernate.*;
import org.hibernate.cfg.*;
public class HibernateUtil {
    public static final SessionFactory sessionFactory;
    static {
          try {
             // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
          } catch (Throwable ex) {
             // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
             throw new ExceptionInInitializerError(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();
             // Store it in the ThreadLocal variable
             session.set(s);
          }
         return s;
       }
        public static void closeSession() throws HibernateException {
          Session s = (Session) session.get();
          if (s != null)
             s.close();
          session.set(null);
        }
    }
```

这个类不仅仅在它的静态初始化过程（仅当加载这个类的时候被 JVM 执行一次）中产生全局 SessionFactory， 同时也有一个 ThreadLocal 变量来为当前线程保存 Session。不论你何时 调用 HibernateUtil.currentSession()，它总是返回同一个线程中的同一个 Hibernate 单元操作。而一个 HibernateUtil.closeSession()调用将终止当前线程相联系的那个单元操作。

在你使用这个帮助类之前，确定你明白 Java 关于本地线程变量（thread-local variable）的概念。一个功能更加强大的 HibernateUtil 帮助类可以在 CaveatEmptorhttp://caveatemptor.hibernate.org/找到 －它同时也出现在书：《Hibernate in Action》中。注意当你把 Hibernate 部署在一个 J2EE 应用服务器上的时候，这个类不是必须的： 一个 Session 会自动绑定到当前的 JTA 事物上，你可以通过 JNDI 来查找 SessionFactory。如果你使用 JBoss AS，Hibernate 可以被部署成一个受管理的系统服务（system service）并自动绑定 SessionFactory 到 JNDI 上。

把 HibernateUtil.java 放在开发目录的源代码路径下面，与 Event.java 放在一起：

```
.+lib  <Hibernate and third-party libraries>
   +src  Event.java  Event.hbm.xml  HibernateUtil.java  hibernate.cfg.xml+databuild.xml
```

再次编译这个程序不应该有问题。最后我们需要配置一个日志系统 － Hibernate 使用通用日志接口，这允许你在 Log4j 和 JDK 1.4 logging 之间进行选择。多数开发者喜欢 Log4j：从 Hibernate 的分发版（它在 etc/目录下）拷贝 log4j.properties 到你的 src 目录，与 hibernate.cfg.xml.放在一起。 看一眼配置示例，你可以修改配置如果你希望看到更多的输出信息。缺省情况下，只有 Hibernate 的启动信息会显示在标准输出上。

教程的基本框架完成了 － 现在我们可以用 Hibernate 来做些真正的工作。

2.2.6. 加载并存储对象

终于，我们可以使用 Hibernate 来加载和存储对象了。我们编写一个带有 main()方法 的 EventManager 类：

```
import org.hibernate.Transaction;
   import org.hibernate.Session;
   import java.util.Date;
   public class EventManager {
      public static void main(String[] args) {
         EventManager mgr = new EventManager();
        if (args[0].equals("store")) {
           mgr.createAndStoreEvent("My Event", new Date());
        }
         HibernateUtil.sessionFactory.close();
      }
   }
```

我们从命令行读入一些参数，如果第一个参数是"store"，我们创建并储存一个新的 Event：

```
private void createAndStoreEvent(String title, Date theDate)
   {
      Session session = HibernateUtil.currentSession();
      Transaction tx = session.beginTransaction();
```

```
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);
        tx.commit();
        HibernateUtil.closeSession();
    }
```

我们创建一个新的 Event 对象并把它传递给 Hibernate。Hibernate 现在负责创建 SQL 并把 INSERT 命令传给数据库。在运行它之前，让我们花一点时间在 Session 和 Transaction 的处理代码上。

每个 Session 是一个独立的单元操作。你会对我们有另外一个 API：Transaction 而感到惊奇。这暗示一个单元操作可以拥有比一个单独的数据库事务更长的生命周期 — 想像在 web 应用程序中，一个单元操作跨越多个 Http request/response 循环 （例如一个创建对话框）。根据“应用程序用户眼中的单元操作”来切割事务是 Hibernate 的基本设计思想之一。我们调用一个长生命期的单元操作 Application Transaction 时，通常包装几个更生命期较短的数据库事务。 为了简化问题，在这个教程里我们使用 Session 和 Transaction 之间是 1 对 1 关系的粒度（one-to-one granularity）。

Transaction.begin()和 commit()都做些什么？ rollback()在哪些情况下会产生错误？ Hibernate 的 Transaction API 实际上是可选的，但是我们通常会为了便利性和可移植性而使用它。 如果你宁可自己处理数据库事务（例如，调用 session.connection.commit()），通过直接和无管理的 JDBC，这样将把代码绑定到一个特定的部署环境中去。 通过在 Hibernate 配置中设置 Transaction 工厂，你可以把你的持久化层部署在任何地方。 查看第 12 章 事务和并发了解更多关于事务处理和划分的信息。在这个例子中我们也忽略任何异常处理和事务回滚。

为了第一次运行我们的应用程序，我们必须增加一个可以调用的 target 到 Ant 的 build 文件中。

```
<target name="run" depends="compile">
    <java fork="true" classname="EventManager" classpathref="libraries">
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java></target>
```

action 参数的值是在通过命令行调用这个 target 的时候设置的:

```
C:\hibernateTutorial\>ant run -Daction=store
```

你应该会看到，编译结束以后，Hibernate 根据你的配置启动，并产生一大堆的输出日志。在日志最后你会看到下面这行：

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

这是 Hibernate 执行的 INSERT 命令，问号代表 JDBC 的待绑定参数。如果想要看到绑定参数的值或者减少日志的长度， 检查你在 log4j.properties 文件里的设置。

现在我们想要列出所有已经被存储的 event，所以我们增加一个条件分支选项到 main 方法中去。

```
if (args[0].equals("store")){
        mgr.createAndStoreEvent("My Event", new Date());
    }else if (args[0].equals("list")) {
        List events = mgr.listEvents();
      for (int i = 0; i < events.size(); i++) {
            Event theEvent = (Event) events.get(i);
               System.out.println("Event: " + theEvent.getTitle() +                    " Time: " +
theEvent.getDate());
       }
    }
```

我们也增加一个新的 listEvents()方法:

```
private List listEvents() {
        Session session = HibernateUtil.currentSession();
       Transaction tx = session.beginTransaction();
        List result = session.createQuery("from Event").list();
        tx.commit();
        session.close();
        return result;
    }
```

我们在这里是用一个 HQL（Hibernate Query Language－Hibernate 查询语言）查询语句来从数据库中 加载所有存在的 Event。Hibernate 会生成正确的 SQL，发送到数据库并使用查询到的数据来生成 Event 对象。 当然你也可以使用 HQL 来创建更加复杂的查询。

如果你现在使用命令行参数-Daction=list 来运行 Ant，你会看到那些至今为止我们储存的 Event。 如果你是一直一步步的跟随这个教程进行的，你也许会吃惊这个并不能工作 － 结果永远为空。原因是 hbm2ddl.auto 打开了一个 Hibernate 的配置选项：这使得 Hibernate 会在每次运行的时候重新创建数据库。通过从配置里删除这个选项来禁止它。 运行了几次 store 之后，再运行 list，你会看到结果出现在列表里。 另外，自动生成数据库表并导出在单元测试中是非常有用的。

2.3. 第二部分 － 关联映射

我们已经映射了一个持久化实体类到一个表上。让我们在这个基础上增加一些类之间的关联性。 首先我们往我们程序里面增加人（people）的概念，并存储他们所参与的一个 Event 列表。 （译者注：与 Event 一样，我们在后面的教程中将直接使用 person 来表示“人”而不是它的中文翻译）

2.3.1. 映射 Person 类

最初的 Person 类是简单的：

```
public class Person {
        private Long id;
        private int age;
```

```
    private String firstname;
    private String lastname;
    Person() {}
   // Accessor methods for all properties, private setter for 'id'
}
```

Create a new mapping file called Person.hbm.xml:

```
<hibernate-mapping>
    <class name="Person" table="PERSON">
      <id name="id" column="PERSON_ID">
        <generator class="increment"/>
      </id>
      <property name="age"/>
      <property name="firstname"/>
      <property name="lastname"/>
    </class>
</hibernate-mapping>
```

Finally, add the new mapping to Hibernate's configuration:

```
    <mapping resource="Event.hbm.xml"/>
    <mapping resource="Person.hbm.xml"/>
```

我们现在将在这两个实体类之间创建一个关联。显然，person 可以参与一系列 Event，而 Event 也有不同的参加者（person）。 设计上面我们需要考虑的问题是关联的方向（directionality），阶数（multiplicity）和集合（collection）的行为。

2.3.2. 一个单向的 Set-based 关联

我们将向 Person 类增加一组 Event。这样我们可以轻松的通过调用 aPerson.getEvents() 得到一个 Person 所参与的 Event 列表，而不必执行一个显式的查询。我们使用一个 Java 的集合类：一个 Set，因为 Set 不允许包括重复的元素而且排序和我们无关。

目前为止我们设计了一个单向的，在一端有许多值与之对应的关联，通过 Set 来实现。 让我们为这个在 Java 类里编码并映射这个关联：

```
public class Person {
    private Set events = new HashSet();
    public Set getEvents() {
      return events;
    }
    public void setEvents(Set events) {
      this.events = events;
    }
}
```

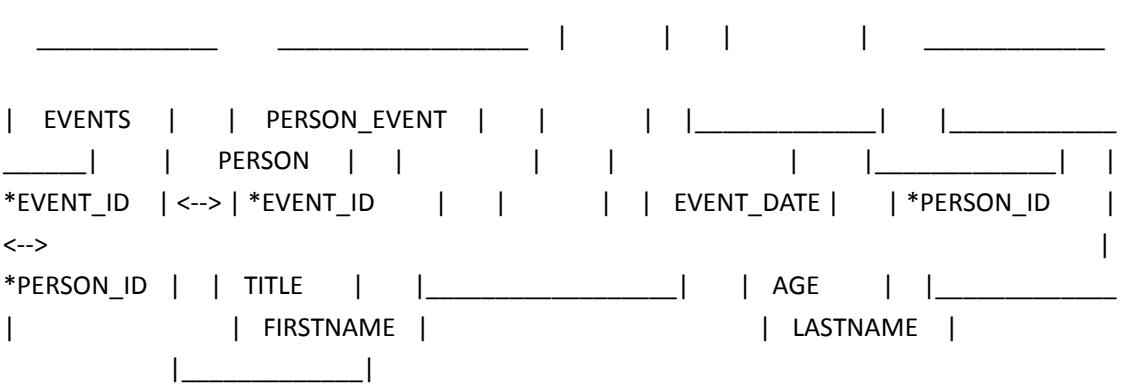
在我们映射这个关联之前，先考虑这个关联另外一端。很显然的，我们可以保持这个关联是

单向的。如果我们希望这个关联是双向的， 我们可以在 Event 里创建另外一个集合，例如: anEvent.getParticipants()。 这是留给你的一个设计选项，但是从这个讨论中我们可以很清楚的了解什么是关联的阶数（multiplicity）：在这个关联的两端都是“多”。 我们叫这个为：多对多（many-to-many）关联。因此，我们使用 Hibernate 的 many-to-many 映射：

```
<class name="Person" table="PERSON">
        <id name="id" column="PERSON_ID">
          <generator class="increment"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
        <set name="events" table="PERSON_EVENT">
           <key column="PERSON_ID"/>
           <many-to-many column="EVENT_ID" class="Event"/>
        </set>
    </class>
```

Hibernate 支持所有种类的集合映射，<set>是最普遍被使用的。对于多对多（many-to-many）关联(或者叫 n:m 实体关系), 需要一个用来储存关联的表（association table）。表里面的每一行代表从一个 person 到一个 event 的一个关联。 表名是由 set 元素的 table 属性值配置的。关联里面的标识字段名，person 的一端，是 由<key>元素定义，event 一端的字段名是由<many-to-many>元素的 column 属性定义的。你也必须告诉 Hibernate 集合中对象的类（也就是位于这个集合所代表的关联另外一端的类）。

这个映射的数据库表定义如下：



2.3.3. 使关联工作

让我们把一些 people 和 event 放到 EventManager 的一个新方法中：

```
private void addPersonToEvent(Long personId, Long eventId) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        Person aPerson = (Person) session.load(Person.class, personId);
        Event anEvent = (Event) session.load(Event.class, eventId);
        aPerson.getEvents().add(anEvent);
```

```
        tx.commit();
        HibernateUtil.closeSession();
    }
```

在加载一个 Person 和一个 Event 之后，简单的使用普通的方法修改集合。 如你所见，没有显式的 update()或者 save(), Hibernate 自动检测到集合已经被修改 并需要保存。这个叫做 automatic dirty checking，你也可以尝试修改任何对象的 name 或者 date 的参数。 只要他们处于 persistent 状态，也就是被绑定在某个 Hibernate Session 上（例如：他们 刚刚在一个单元操作从被加载或者保存），Hibernate 监视任何改变并在后台隐式执行 SQL。同步内存状态和数据库的过程，通常只在 一个单元操作结束的时候发生，这个过程被叫做 flushing。

你当然也可以在不同的单元操作里面加载 person 和 event。或者在一个 Session 以外修改一个 不是处在持久化（persistent）状态下的对象（如果该对象以前曾经被持久化，我们称这个状态为脱管（detached））。 在程序里，看起来像下面这样：

```
    private void addPersonToEvent(Long personId, Long eventId) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        Person aPerson = (Person) session.load(Person.class, personId);
        Event anEvent = (Event) session.load(Event.class, eventId);
         tx.commit();
        HibernateUtil.closeSession();
        aPerson.getEvents().add(anEvent); // aPerson is detached
        Session session2 = HibernateUtil.currentSession();
        Transaction tx2 = session.beginTransaction();
        session2.update(aPerson); // Reattachment of aPerson
        tx2.commit();
        HibernateUtil.closeSession();
    }
```

对 update 的调用使一个脱管对象（detached object）重新持久化，你可以说它被绑定到 一个新的单元操作上，所以任何你对它在脱管（detached）状态下所做的修改都会被保存到数据库里。

这个对我们当前的情形不是很有用，但是它是非常重要的概念，你可以把它设计进你自己的程序中。现在，加进一个新的 选项到 EventManager 的 main 方法中，并从命令行运行它来完成这个练习。如果你需要一个 person 和 一个 event 的标识符 — save()返回它。******* 这最后一句看不明白

上面是一个关于两个同等地位的类间关联的例子，这是在两个实体之间。像前面所提到的那样，也存在其它的特别的类和类型，这些类和类型通常是“次要的”。 其中一些你已经看到过，好像 int 或者 String。我们称呼这些类为值类型（value type），它们的实例依赖（depend）在某个特定的实体上。这些类型的实例没有自己的身份（identity），也不能在实体间共享 （比如两个 person 不能引用同一个 firstname 对象，即使他们有相同的名字）。当然，value types 并不仅仅在 JDK 中存在（事实上，在一个 Hibernate 程序中，所有的 JDK 类都被视为值类型），你也可以写你自己的依赖类，例如 Address， MonetaryAmount。

你也可以设计一个值类型的集合（collection of value types），这个在概念上与实体的集合有很大的不同，但是在 Java 里面看起来几乎是一样的。

2.3.4. 值类型的集合

我们把一个值类型对象的集合加入 Person。我们希望保存 email 地址，所以我们使用 String，而这次的集合类型又是 Set：

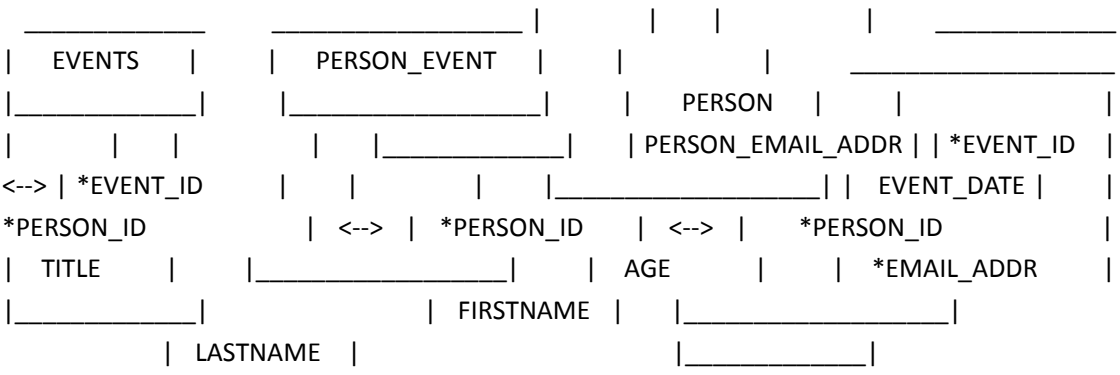
```
private Set emailAddresses = new HashSet();public Set getEmailAddresses() {  return emailAddresses;}public void setEmailAddresses(Set emailAddresses) {  this.emailAddresses = emailAddresses;}
```

Set 的映射

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR"> <key column="PERSON_ID"/> <element type="string" column="EMAIL_ADDR"/></set>
```

比较这次和较早先的映射，差别主要在 element 部分这次并没有包括对其它实体类型的引用，而是使用一个元素类型是 String 的集合（这里使用小写的名字是向你表明它是一个 Hibernate 的映射类型或者类型转换器）。 和以前一样，set 的 table 参数决定用于集合的数据库表名。key 元素 定义了在集合表中使用的外键。element 元素的 column 参数定义实际保存 String 值 的字段名。

看一下修改后的数据库表定义。



你可以看到集合表（collection table）的主键实际上是个复合主键，同时使用了 2 个字段。这也暗示了对于同一个 person 不能有重复的 email 地址，这正是 Java 里面使用 Set 时候所需要的语义（Set 里元素不能重复）。

你现在可以试着把元素加入这个集合，就像我们在之前关联 person 和 event 的那样。Java 里面的代码是相同的。

2.3.5. 双向关联

下面我们将映射一个双向关联（bi-directional association）— 在 Java 里面让 person 和 event 可以从关联的 任何一端访问另一端。当然，数据库表定义没有改变，我们仍然需要多对多（many-to-many）的阶数（multiplicity）。一个关系型数据库要比网络编程语言 更加灵活，所以它并不需要任何像导航方向（navigation direction）的东西 — 数据可以用任何可能的

方式进行查看和获取。

首先，把一个参与者（person）的集合加入 Event 类中：

private Set participants = new HashSet();public Set getParticipants() { return participants;}public void setParticipants(Set participants) { this.participants = participants;}
在 Event.hbm.xml 里面也映射这个关联。

<set name="participants" table="PERSON_EVENT" inverse="true"> <key column="EVENT_ID"/> <many-to-many column="PERSON_ID" class="Person"/></set>
如你所见，2 个映射文件里都有通常的 set 映射。注意 key 和 many-to-many 里面的字段名在两个映射文件中是交换的。这里最重要的不同是 Event 映射文件里 set 元素的 inverse="true" 参数。

这个表示 Hibernate 需要在两个实体间查找关联信息的时候，应该使用关联的另外一端 — Person 类。 这将会极大的帮助你理解双向关联是如何在我们的两个实体间创建的。

2.3.6. 使双向关联工作

首先，请牢记在心，Hibernate 并不影响通常的 Java 语义。 在单向关联中，我们是怎样在一个 Person 和一个 Event 之间创建联系的？ 我们把一个 Event 的实例加到一个 Person 类内的 Event 集合里。所以，显然如果我们要让这个关联可以双向工作， 我们需要在另外一端做同样的事情 — 把 Person 加到一个 Event 类内的 Person 集合中。 这“在关联的两端设置联系”是绝对必要的而且你永远不应该忘记做它。

许多开发者通过创建管理关联的方法来保证正确的设置了关联的两端，比如在 Person 里：

protected Set getEvents() { return events;}protected void setEvents(Set events) { this.events = events;}public void addToEvent(Event event) { this.getEvents().add(event); event.getParticipants().add(this);}public void removeFromEvent(Event event) { this.getEvents().remove(event); event.getParticipants().remove(this);}
注意现在对于集合的 get 和 set 方法的访问控制级别是 protected - 这允许在位于同一个包（package）中的类以及继承自这个类的子类 可以访问这些方法，但是禁止其它的直接外部访问，避免了集合的内容出现混乱。你应该尽可能的在集合所对应的另外一端也这样做。

inverse 映射参数究竟表示什么呢？对于你和对于 Java 来说，一个双向关联仅仅是在两端简单的设置引用。然而仅仅这样 Hibernate 并没有足够的信息去正确的产生 INSERT 和 UPDATE 语句（以避免违反数据库约束）， 所以 Hibernate 需要一些帮助来正确的处理双向关联。把关联的一端设置为 inverse 将告诉 Hibernate 忽略关联的 这一端，把这端看成是另外一端的一个镜子（mirror）。这就是 Hibernate 所需的信息，Hibernate 用它来处理如何把把 一个数据导航模型映射到关系数据库表定义。 你仅仅需要记住下面这个直观的规则：所有的双向关联需要有一端被设置为 inverse。在一个一对多（one-to-many）关联中 它必须是代表多（many）的那端。而在多对多（many-to-many）关联中，你可以任意选取一端，两端之间

并没有差别。

2.4. 总结

这个教程覆盖了关于开发一个简单的 Hibernate 应用程序的几个基础方面。

如果你已经对 Hibernate 感到自信，继续浏览开发指南里你感兴趣的内容－那些会被问到的问题大多是事务处理 (第 12 章 事务和并发)，抓取（fetch）的效率 (第 20 章 提升性能)，或者 API 的使用 (第 11 章 与对象共事)和查询的特性(第 11.4 节 “查询”)。