HIBERNATE - 符合 Java 习惯的关系数据库持久化

Hibernate2 参考文档

2.1.2

Table of Contents

前言
1. 在 Tomcat 中快速上手
   1.1. 开始 Hibernate 之旅
   1.2. 第一个可持久化类
   1.3. 映射 cat
   1.4. 与猫同乐
   1.5. 结语
2. 体系结构
   2.1. 总览
   2.2. 持久化对象标识（Persistent Object Identity ）
   2.3. JMX 集成
   2.4. JCA 支持
3. SessionFactory 配置
   3.1. 可编程配置方式
   3.2. 获取 SessionFactory
   3.3. 用户自行提供 JDBC 连接
   3.4. Hibernate 提供的 JDBC 连接
   3.5. 其它配置属性
   3.5.1. SQL Dialects SQL 方言
   3.5.2. 外连接抓取（Outer Join Fetching ）
   3.5.3. 二进制流
   3.5.4. 在控制台记录 SQL
   3.5.5. 自定义 ConnectionProvider
   3.5.6. 常用数据库属性
   3.5.7. 自定义 CacheProvider
   3.5.8. 事务策略
   3.5.9. 绑定 SessionFactory 到 JNDI
   3.5.10. 查询语言替换
   3.6. XML 配置文件
   3.7. Logging
   3.8. NamingStrategy（命名策略）
4. 持久化类(Persistent Classes)
   4.1. 简单示例
   4.1.1. 为持久化字段声明访问器(accessors)和是否可变的标志(mutators)
   4.1.2. 实现一个默认的构造方法（constructor）
   4.1.3. 提供一个标识属性（identifier property）（可选）
   4.1.4. 建议使用不是 final 的类 (可选)
   4.2. 继承（Inheritance ）
   4.3. 持久化生命周期（Lifecycle）中的回调（Callbacks）
   4.4. 合法性检查（Validatable）
   4.5. XDoclet 示例

5. O/R Mapping 基础
    5.1. 映射声明(Mapping declaration)
    5.1.1. Doctype
    5.1.2. hibernate-mapping
    5.1.3. class
    5.1.4. id
    5.1.4.1. generator
    5.1.4.2. 高/低位算法（Hi/Lo Algorithm）
    5.1.4.3. UUID 算法（UUID Algorithm ）
    5.1.4.4. 标识字段和序列（Identity Columns and Sequences）
    5.1.4.5. 程序分配的标识符（Assigned Identifiers）
    5.1.5. composite-id 联合 ID
    5.1.6. 识别器（discriminator）
    5.1.7. 版本（version）(可选)
    5.1.8. 时间戳（timestamp ）(可选)
    5.1.9. property
    5.1.10. 多对一（many-to-one）
    5.1.11. 一对一
    5.1.12. 组件（component），动态组件（dynamic-component）
    5.1.13. 子类(subclass)
    5.1.14. 连接的子类（joined-subclass）
    5.1.15. map, set, list, bag
    5.1.16. 引用（import）
    5.2. Hibernate 的类型
    5.2.1. 实体（Entities）和值（values）
    5.2.2. 基本值类型
    5.2.3. 持久化枚举（Persistent enum）类型
    5.2.4. 自定义值类型
    5.2.5. 映射到"任意"(any)类型
    5.3. SQL 中引号包围的标识符
    5.4. 自定义 DDL
    5.5. 映射文件的模块化（Modular mapping files）
6. 集合类(Collections)
    6.1. 持久化集合类(Persistent Collections)
    6.2. 映射集合（Mapping a Collection）
    6.3. 值集合和多对多关联(Collections of Values and Many To Many Associations)
    6.4. 一对多关联（One To Many Associations）
    6.5. 延迟初始化(延迟加载)（Lazy Initialization）
    6.6. 集合排序（Sorted Collections）
    6.7. 对 collection 排序的其他方法（Other Ways To Sort a Collection）
    6.8. 垃圾收集（Garbage Collection）
    6.9. 双向关联（Bidirectional Associations）
    6.10. 三重关联（Ternary Associations）
    6.11. 异类关联(Heterogeneous Associations)
    6.12. 集合例子（Collection Example）
    6.13. <idbag>
7. 组件（Components）
    7.1. 作为依赖的对象（As Dependent Objects）
    7.2. In Collections
    7.3. 作为一个 Map 的索引（As a Map Index ）
    7.4. 作为联合标识符(As Composite Identifiers)
    7.5. 动态组件 （Dynamic components）
8. 操作持久化数据(Manipulating Persistent Data)

8.1. 创建一个持久化对象
8.2. 装载对象
8.3. Querying
8.3.1. 标量查询（Scalar query）
8.3.2. 查询接口（Query interface）
8.3.3. 可滚动迭代(Scrollable iteration)
8.3.4. 过滤集合类(Filtering collections)
8.3.5. 条件查询
8.3.6. 使用本地 SQL 的查询
8.4. 更改在当前 session 中保存或者装载的对象
8.5. 更改在以前 session 中保存或者装载的对象
8.6. 把在先前的 session 中保存或装载的对象重新与新 session 建立关联(reassociate)
8.7. 删除持久化对象
8.8. 对象图（Graphs of objects）
8.8.1. 自动管理生命周期的对象（lifecycle object）
8.8.2. 通过可触及性决定持久化（Persistence by Reachability）
8.9. 清洗(Flushing) -- 这个词很难翻译，不能使用“刷新”，因为刷新一词已经被"refresh"使用了。有什么好的建议？
8.10. 结束一个 Session
8.10.1. 清洗(Flush)session
8.10.2. 提交事务
8.10.3. 关闭 session
8.10.4. 处理异常
8.11. 拦截器(Interceptors)
8.12. 元数据(Metadata) API
9. 父子关系(Parent Child Relationships)
9.1. 关于 collections
9.2. 双向的一对多关系(Bidirectional one to many)
9.3. 级联（Cascades）
9.4. 级联更新（Using cascading update()）
9.5. 结论
10. Hibernate 查询语言(Query Language)， 即 HQL
10.1. 大小写敏感性(Case Sensitivity)
10.2. from 子句
10.3. 联合（Associations）和连接（joins）
10.4. select 子句
10.5. 统计函数(Aggregate functions)
10.6. 多态(polymorphism)
10.7. where 子句
10.8. 表达式(Expressions)
10.9. order by 子句
10.10. group by 子句
10.11. 子查询
10.12. 示例
10.13. 提示和技巧（Tips & Tricks）
11. 实例(A Worked Example)
11.1. 持久化类
11.2. Hibernate 映射
11.3. Hibernate 代码
12. 性能提升（Improving Performance）
12.1. 用于延迟装载的代理
12.2. 第二层缓存(The Second Level Cache)s

12.2.1. 映射(Mapping)
12.2.2. 只读缓存
12.2.3. 读/写缓存
12.2.4. 不严格的读/写缓存
12.2.5. 事务缓存（transactional）
12.3. 管理 Session 缓存
12.4. 查询缓存(Query Cache)
13. 理解集合类的性能（Understanding Collection Performance）
13.1. 分类（Taxonomy）
13.2. Lists, maps 和 sets 用于更新效率最高
13.3. Bag 和 list 是反向集合类中效率最高的
13.4. 一次性删除(One shot delete)
14. 条件查询(Criteria Query)
14.1. 创建一个 Criteria 实例
14.2. 缩小结果集范围
14.3. 对结果排序
14.4. 关联（Associations）
14.5. 动态关联对象获取（Dynamic association fetching）
14.6. 根据示例查询（Example queries）
15. SQL 查询
15.1. 创建一个基于 SQL 的 Query
15.2. 别名和属性引用
15.3. 为 SQL 查询命名
16. 继承映射(Inheritance Mappings)
16.1. 三种策略
16.2. 限制
17. 事务和并行（Transactions And Concurrency）
17.1. 配置，会话和工厂（Configurations, Sessions and Factories）
17.2. 线程和连接（Threads and connections）
17.3. 乐观锁定 / 版本化（Optimistic Locking / Versioning）
17.3.1. 使用长生命周期带有自动版本化的会话
17.3.2. 使用带有自动版本化的多个会话
17.3.3. 应用程序自己进行版本检查
17.4. 会话断开连接（Session disconnection）
17.5. 悲观锁定（Pessimistic Locking）
18. 映射实例(Mapping Examples)
18.1. 雇员 / 雇主（Employer/Employee）
18.2. 作者 / 著作(Author/Work)
18.3. 客户 / 订单 / 产品(Customer/Order/Product)
19. 工具箱指南
19.1. Schema 生成器（Schema Generation）
19.1.1. 对 schema 定制化(Customizing the schema)
19.1.2. 运行该工具
19.1.3. 属性(Properties)
19.1.4. 使用 Ant(Using Ant)
19.1.5. 对 schema 的增量更新(Incremental schema updates)
19.1.6. 用 Ant 来增量更新 schema(Using Ant for incremental schema updates)
19.2. 代码生成（Code Generation）
19.2.1. 配置文件(可选)
19.2.2. meta 属性
19.2.3. 基本的 finder 生成器（Basic finder generator）
19.2.4. 基于 Velocity 的渲染器/生成器(Velocity based renderer/generator)
19.3. 映射文件生成器（Mapping File Generation）

19.3.1. 运行此工具
20. 最佳实践(Best Practices)

前言

在今日的企业环境中，把面向对象的软件和关系数据库一起使用可能是相当麻烦、浪费时间的。Hibernate 是一个面向 Java 环境的对象/关系数据库映射工具。对象/关系数据库映射(object/relational mapping (ORM))这个术语表示一种技术，用来把对象模型表示的对象映射到基于 SQL 的关系模型结构中去。

Hibernate 不仅仅管理 Java 类到数据库表的映射，还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用 SQL 和 JDBC 处理数据的时间。Hibernate 的目标是对于开发者通常的数据持久化相关的编程任务，解放其中的 95%。

如果你对 Hibernate 和对象/关系数据库映射还是个新手，或者甚至对 Java 也不熟悉，请按照下面的步骤来学习。

- 阅读这个 30 分钟就可以结束的 Chapter 1, *在 Tomcat 中快速上手*，它使用 Tomcat。
- 阅读 Chapter 2, *体系结构*来理解 Hibernate 可以使用的环境。
- 查看 Hibernate 发行包中的 eg/目录，里面有一个简单的独立运行的程序。把你的 JDBC 驱动拷贝到 lib/目录下，修改一下 src/hibernate.properties,指定其中你的数据库的信息。进入命令行，切换到你的发行包的目录，输入 ant eg(使用了 Ant），或者在 Windows 操作系统中使用 build eg。
- 把这份参考文档作为你学习的主要信息来源。
- 在 Hibernate 的网站上可以找到经常提问的问题与解答(FAQ)。
- 在 Hibernate 网站上还有第三方的演示、示例和教程的链接。
- Hibernate 网站的“社区(Community Area)”是讨论关于设计模式以及很多整合方案(Tomcat, JBoss, Spring,Struts, EJB,等等)的好地方。
- 离线版本的 Hibernate 网站随着 Hibernate 发行包一起发布，位于 doc/目录下。

如果你有问题，请使用 Hibernate 网站上链接的用户论坛。我们也提供一个 JIRA 问题追踪系统，来搜集 bug 报告和新功能请求。如果你对开发 Hibernate 有兴趣，请加入开发者的邮件列表。 （译者注:目前 Hibernate 已经有一个中文的用户论坛，URL 是 http://forum.hibernate.org.cn 我们随时欢迎您的访问。）

翻译说明

===

本文档的翻译是在网络上协作进行的，也会不断根据 Hibernate 的升级进行更新。提供此文档的目的是为了减缓学习 Hibernate 的坡度，而非代替原文档。我们建议所有有能力的读者都直接阅读英文原文。

若您对翻译有异议，或发现翻译错误，敬请不吝赐教，请到 Hibernate 中文论坛(http://forum.hibernate.org.cn)提出,或报告到如下 email 地址：caoxg at redsaga.com

第 6 章(集合类)、第 7 章(组件)是由 jlinux 翻译，第 10 章（父子关系）是由 muziq 翻译，第 16 章(事务和并行)、第 17 章（映射实例）是由 liangchen 翻译，其他各章节是由曹晓钢翻译的，第 18、19、20 章，bruce、robbin 也有贡献。曹晓钢也进行了全书从 2.0.4 更新到 2.1.1 版本、2.1.2 版本的工作。

更详细的翻译者与翻译更新情况，请查阅 CVS 目录下的 TRANSLATE-LOG.TXT 文件。

版权声明

===

Hibernate 英文文档属于 Hibernate 发行包的一部分，遵循 LGPL 协议。本翻译版本同样遵循 LGPL 协议。参与翻译的译者一致同意放弃除署名权外对本翻译版本的其它权利要求。

您可以自由链接、下载、传播此文档，或者放置在您的网站上，甚至作为产品的一部分发行。但前提是必须保证全文完整转载，包括完整的版权信息和作译者声明。这里“完整”的含义是，不能进行任何删除/增添/注解。若有删除/增添/注解，必须明确声明那些部分并非本文档的一部分。

Chapter 1. 在 Tomcat 中快速上手

1.1. 开始 Hibernate 之旅

这份教程讨论如何在 Apache Tomcat servlet 容器中为 web 程序安装 Hibernate 2.1。Hibernate 在大多数主流 J2EE 应用服务器 的受管理环境中都可以良好运作，也可以作为独立应用程序运行。在本例中的示例数据库系统是 PostgreSQL 7.3，当然也可以 很容易的换成 Hibernate 支持的其它 16 种数据库之一。

第一步是拷贝所有需要的运行库到 Tomcat 去。在这篇教程中，我们使用一个单独的 web 程序（webapps/quickstart）。我们要考虑全局库文件搜索路径（TOMCAT/common/lib）和本 web 应用程序上下文的类装载器搜索路径（对于 jar 来说是 webapps/quickstart/WEB-INF/lib，对于 class 文件来说是 webapps/quickstart/WEB-INF/classes）。我们把这两个类装载器级别分别称为全局类路径(global classpath)和上下文类路径(context classpath)。

- 首先，把数据库需要的 JDBC 驱动拷贝到全局类路径。这是 tomcat 附带的 DBCP 连接池软件所要求的。对于本教程来说，把 pg73jdbc3.jar 库文件（对应 PostgreSQL 7.3 和 JDK 1.4)到全局类装载器路径去。如果你使用一个不同的数据库，拷贝相应的 JDBC 驱动）。
- 不要拷贝任何其他东西到全局类装载器去。否则你可能在一些工具上遇到麻烦，比如 log4j, commons-logging 等。 记得要使用每个 web 应用程序自己的上下文类路径，就是说把你自己的类库拷贝到 WEB-INF/lib 下去，把配置文件 configuration/property 拷贝到 WEB-INF/classes 下面去。这两个目录默认都是上下文类路径级别的。
- Hibernate 本身打包成一个 JAR 库。hibernate2.jar 文件要和你应用程序的其他库文件一起放在上下文类路径中。在运行时，Hibernate 还需要一些第三方库，它们在 Hibernate 发行包的 lib/目录下。参见 Table 1.1。把你需要的第三方库文件也拷贝到上下文类路径去。
- 要为 Tomcat 和 Hibernate 都配置数据库连接。也就是说 Tomcat 要负责提供 JDBC 连接池，Hibernate 通过 JNDI 来请求这些连接。Tomcat 把连接池绑定到 JNDI。

Table 1.1. Hibernate 第三方库

库	描述
dom4j（必需）	Hibernate 在解析 XML 配置和 XML 映射元文件时需要使用 dom4j。
CGLIB（必需）	Hibernate 在运行时使用这个代码生成库强化类（与 Java 反射机制联合使用）。
Commons	Hibernat 使用 Apache Jakarta Commons 项目提供的多个工具类库。

库	描述
Collections, Commons Logging（必需）	
ODMG4（必需）	Hibernate 提供了一个可选的 ODMG 兼容持久化管理界面。如果你需要映射集合，你就需要这个类库，就算你不是为了使用 ODMG API。我们在这个教程中没有使用集合映射，但不管怎样把这个 JAR 拷贝过去总是不错的。
Log4j（可选）	Hibernate 使用 Commons Logging API,后者可以使用 Log4j 作为实施 log 的机制。如果把 Log4j 库放到上下文类目录中，Commons Logging 就会使用 Log4j 和它在上下文类路径中找到的 log4j.properties 文件。在 Hibernate 发行包中包含有一个示例的 properties 文件。所以，也把 log4j.jar 拷贝到你的上下文类路径去吧。
其他文件是不是必需的？	请察看 Hibernate 发行包中的/lib/README.txt 文件。这是一个 Hibernate 发行包中附带的第三方类库的列表，总是保持更新。你可以在那里找到所有必需或者可选的类库的列表。

好了，现在所有的类库已经被拷贝过去了，让我们在 Tomcat 的主配置文件,TOMCAT/conf/server.xml 中增加一个数据库 JDBC 连接池的资源声明，

```
<Context path="/quickstart" docBase="quickstart">
    <Resource name="jdbc/quickstart" scope="Shareable"
type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/quickstart">
        <parameter>
            <name>factory</name>

<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <!-- DBCP database connection settings -->
        <parameter>
            <name>url</name>
            <value>jdbc:postgresql://localhost/quickstart</value>
        </parameter>
        <parameter>

<name>driverClassName</name><value>org.postgresql.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>quickstart</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value>secret</value>
        </parameter>

        <!-- DBCP connection pooling options -->
        <parameter>
            <name>maxWait</name>
            <value>3000</value>
        </parameter>
        <parameter>
```

```
            <name>maxIdle</name>
            <value>100</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>10</value>
        </parameter>
    </ResourceParams>
</Context>
```

这个例子中我们要配置的上下文叫做 quickstart，它位于 TOMCAT/webapp/quickstart 目录。要访问任何 Servlet,在你的浏览器中访问 http://localhost:8080/quickstart 就可以了。

Tomcat 在这个配置下，使用 DBCP 连接池，通过 JNDI 位置：java:comp/env/jdbc/quickstart 提供带有缓冲池的 JDBCConnections。如果你在让连接池工作的时候遇到困难，请查阅 Tomcat 文档。如果你得到了 JDBC 驱动的 exception 信息，请先不要用 Hibernate,测试 JDBC 连接池本身是否正确。Tomcat 和 JDBC 的教程可以在 Web 上查到。

下一步是配置 hibernate，来使用绑定到 JNDI 的连接池中提供的连接。我们使用 XML 格式的 Hibernate 配置。当然，使用 properties 文件的方式在功能上也是一样的，也不提供什么特别好处。我们用 XML 配置的原因，是因为一般会更方便。XML 配置文件放在上下文类路径(WEB-INF/classes)下面，称为 hibernate.cfg.xml:

```
<?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>

    <session-factory>

        <property
name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property
name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Mapping files -->
        <mapping resource="Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

我们关闭了 SQL 命令的 log,告诉 Hibernate 使用哪种 SQL 数据库方言(dialet)，还有如何得到 JDBC 连接（通过声明数据源池绑定的 JNDI 地址）。方言是必需的，因为不同的数据库都和 SQL "标准"有一些出入。Hibernate 会替你照管这些差异之处，发行包包含了所有主流的商业和开放源代码数据库的方言。

SessionFactory 是 Hibernate 的概念，对应一个数据存储源，如果有多个数据库，可以创建多个 XML 配置文件，也在你的程序中创建多个 Configuration 和 SessionFactory 对象。

在 hibernate.cfg.xml 中的最后一个元素声明了 Cat.hbm.xml 是一个 Hibernate XML 映射文件，对应持久化类 Cat。这个文件包含了把 POJO 类映射到数据库表（或多个数据库表）的元数据。我们稍后就回来看这个文件。让我们先编写这个 POJO 类，再在声明它的映射元数据。

1.2. 第一个可持久化类

s Hibernate 让普通的 Java 对象(Plain Old Java Objects ,就是 POJOs,有时候也称作 Plain Ordinary Java Objects）变成持久化类。一个 POJO 很像 JavaBean,属性通过 getter 和 setter 方法访问，对外隐藏了内部实现的细节。

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Hibernate 对属性使用的类型不加限制。所有的 Java JDK 类型和原始类型（比如 String,char 和 float)都可以被映射，也包括 Java 集合框架（Java collections framework)中的类。你

可以把它们映射成为值，值集合，或者与其他实体相关联。id 是一个特殊的属性，代表了这个类的数据库标识符(主键)，它对于类似于 Cat 这样的实体是必需的。

持久化类不需要实现什么特别的接口，也不需要从一个特别的持久化根类继承下来。Hibernate 也不需要使用任何编译期处理，比如字节码增强操作，它独立的使用 Java 反射机制和运行时类增强（通过 CGLIB)。所以，在 Hibernate 中，POJO 的类不需要任何前提条件，我们就可以把它映射成为数据库表。

1.3. 映射 cat

Cat.hbm.xml 映射文件包含了对象/关系映射所需的元数据。

元数据包含了持久化类的声明和把它与其属性映射到数据库表的信息（属性作为值或者是指向其他实体的关联）。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat"
table="CAT">

        <!-- A 32 hex character is our surrogate key. It's
automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-
null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -
->
        <property name="name">
            <column name="NAME" sql-type="varchar(16)" not-
null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

每个持久化类都需要一个标识属性（实际上，只是哪些代表一手对象的类，而不是代表值对象的类，后者会被映射称为一手对象中的一个组件）。这个属性用来区分持久化对象：如果 catA.getId().equals(catB.getId())结果是 true 的话，两只猫就是相同的。这个概念称为*数据库标识*。Hiernate 附带了几种不同的标识符生成器，用于不同的场合（包括数据库本地的顺序(sequence)生成器和 hi/lo 高低位标识模式）。我们在这里使用 UUID 生成器，并指定 CAT 表的 CAT_ID 字段（作为表的主键）存放生成的标识值。

Cat 的其他属性都映射到同一个表。对 name 属性来说，我们把它显式地声明映射到一个数据库字段。如果数据库 schema 是由映射声明使用 Hibernate 的 *SchemaExport* 工具自动生成的（作为 SQL DDL 指令），这特别有用。所有其它的属性都用 Hibernate 的默认值映射，大多数情况你都会这样做。数据库中的 CAT 表看起来是这样的：

```
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 cat_id | character(32)         | not null
 name   | character varying(16) | not null
 sex    | character(1)          |
 weight | real                  |
Indexes: cat_pkey primary key btree (cat_id)
```

你现在可以在你的数据库中首先创建这个表了，如果你需要使用 SchemaExport 工具把这个步骤自动化，请参阅 Chapter 19, *工具箱指南*。这个工具能够创建完整的 SQL DDL，包括表定义，自定义的字段类型约束，惟一约束和索引。

1.4. 与猫同乐

我们现在可以开始 Hibernate 的 Session 了。我们用它来从数据库中存取 Cat。首先，我们要从 SessionFactory 中获取一个 Session(Hibernate 的工作单元）。

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

SessionFactory 负责一个数据库，也只对应一个 XML 配置文件（hibernate.cfg.xml）。

这篇教程的关注点在于配置 Tomcat 的 JDBC 连接，绑定到 JNDI 上，以及 Hibernate 的基础配置。你可以用喜欢的任何方式编写一个 Servlet,包含下面的代码，只要确保 SessionFactory 只创建一次。也就是说你不能把它作为你的 Serlvet 的实例变量。一个好办法是用在辅助类中用一个静态的 SessionFactory，例如这样：

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (HibernateException ex) {
            throw new RuntimeException("Exception building
SessionFactory: " + 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();
    }
}
```

这个类不但在它的静态属性中使用了 SessionFactory，还使用了 ThreadLocal 来为当前工作线程保存 Session。

Session 不是线程安全的，代表与数据库之间的一次操作。Session 通过 SessionFactory 打开，在所有的工作完成后，需要关闭：

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

HibernateUtil.closeSession();
```

在 Session 中，每个数据库操作都是在一个事务(transaction)中进行的，这样就可以隔离开不同的操作（甚至包括只读操作）。我们使用 Hibernate 的 Transaction API 来从底层的事务策略中（本例中是 JDBC 事务）脱身。这样，如果需要把我们的程序部署到一个由容器管理事务的环境中去（使用 JTA)，我们就不需要更改源代码。请注意，我们上面的例子没有处理任何异常。

也请注意，你可以随心所欲的多次调用 HibernateUtil.currentSession();，你每次都会得到同一个当前线程的 Session。你必须确保 Session 在你的数据库事务完成后关闭，不管是在你的 Servlet 代码中，或者在 ServletFilter 中，HTTP 结果返回之前。

Hibernate 有不同的方法来从数据库中取回对象。最灵活的方式是使用 Hibernate 查询语言(HQL),这是一种容易学习的语言，是对 SQL 的面向对象的强大扩展。

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

Query query = session.createQuery("select cat from Cat as cat where
cat.sex = :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
```

```
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Hibernate 也提供一种面向对象的*按条件查询* API，可以执行公式化的类型安全的查询。当然，Hibernate 在所有与数据库的交互中都使用 `PrepatedStatement` 和参数绑定。

1.5. 结语

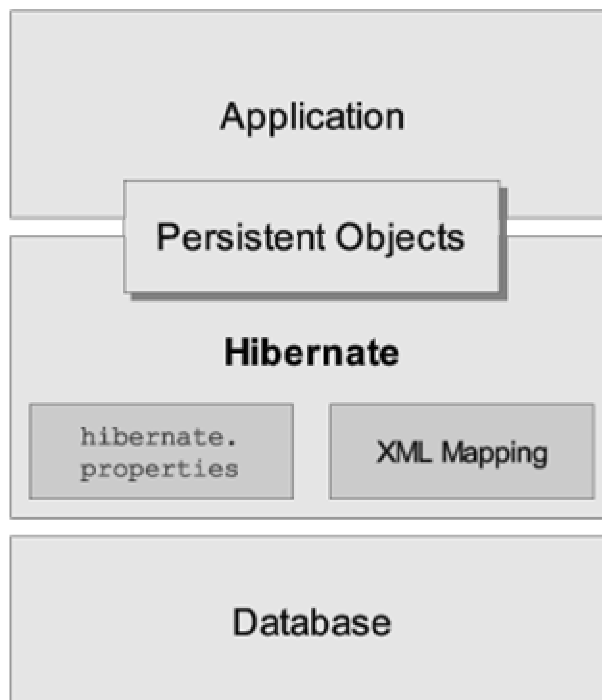
在这个短小的教程中，我们只描绘了 Hibernate 的基本面目。请注意我们没有在例子中包含 Servlet 相关代码。你必须自行编写 Servlet，然后把你认为合适的 Hibernate 代码插入。

请记住 Hibernate 作为数据库访问层，是与你的程序紧密相关的。一般，所有其他层次都依赖持久机制。请确信你理解了这种设计的含义。

Chapter 2. 体系结构

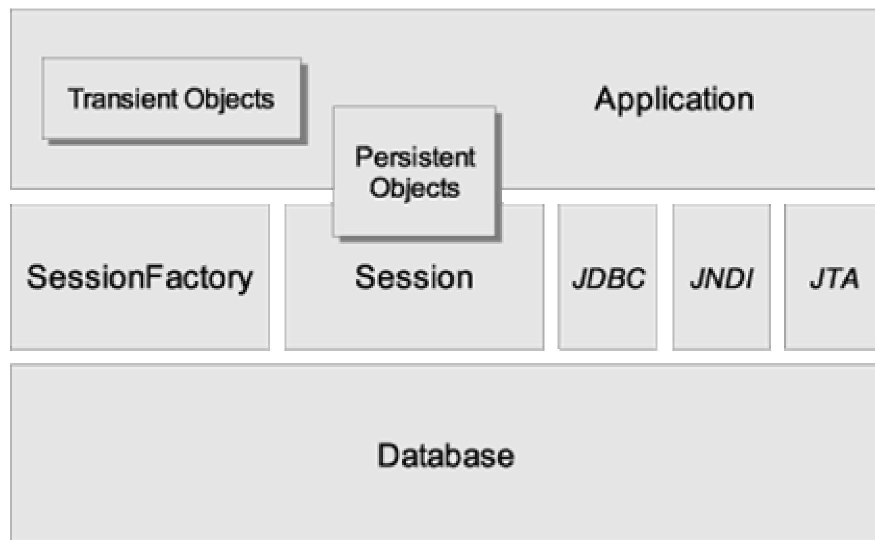
2.1. 总览

对 Hibernate 非常高层的概览：

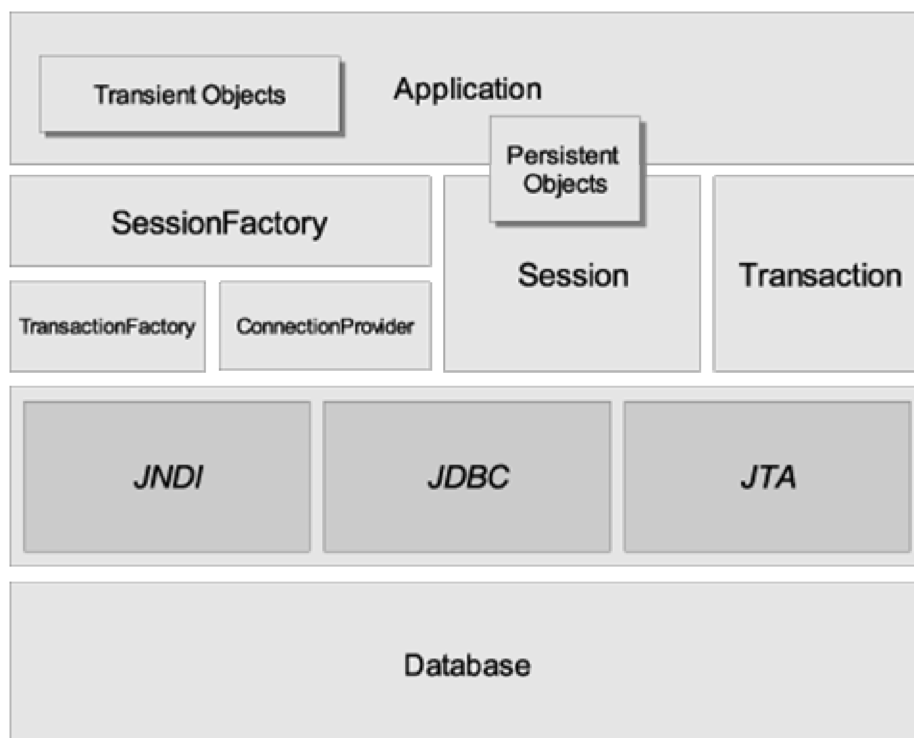


这幅图展示了 Hibernate 使用数据库和配置文件数据来为应用程序提供持久化服务（和持久化的对象）。

让我们更细致地观察一下运行时的体系结构。 挺不幸的，Hibernate 是比较复杂的，提供了好几种不同的运行方式。我们展示一下两种极端情况。轻型体系中，应用程序自己提供 JDBC 连接，并且自行管理事务。这种方式使用了 Hibernate API 的一个最小子集。



全面解决体系中，对于应用程序来说，所有的底层 JDBC/JTA API 都被抽象了，Hibernate 会替你照管所有的细节。



下面是图中一些对象的定义：

SessionFactory (**net.sf.hibernate.SessionFactory**)

对编译过的映射文件的一个线程安全的，不可变的缓存快照。它是 Session 的工厂。是 ConnectionProvider 的客户。

可能持有事务之间重用的数据的缓存。

会话，Session (**net.sf.hibernate.Session**)

单线程，生命期短促的对象，代表应用程序和持久化层之间的一次对话。封装了一个 JDBC 连接。也是 Transaction 的工厂。

持有持久化对象的缓存。

持久化对象（Persistent Object)及其集合(Collection)

生命期短促的单线程的对象，包含了持久化状态和商业功能。它们可能是普通的 JavaBeans，唯一特别的是他们现在从属于且仅从属于一个 Session。

临时对象(Transient Object)及其集合（Collection）

目前没有从属于一个 Session 的持久化类的实例。他们可能是刚刚被程序实例化，还没有来得及被持久化，或者是被一个已经关闭的 Session 所实例化的。

事务，Transaction (**net.sf.hibernate.Transaction**)

（可选） 单线程，生命期短促的对象，应用程序用它来表示一批工作的原子操作。是底层的 JDBC,JTA 或者 CORBA 事务的抽象。一个 Session 可能跨越多个 Transaction 事务。

ConnectionProvider (**net.sf.hibernate.connection.ConnectionProvider**)

（可选）JDBC 连接的工厂和池。从底层的 Datasource 或者 DriverManager 抽象而来。对应用程序不可见。

TransactionFactory (**net.sf.hibernate.TransactionFactory**)

（可选）事务实例的工厂。对应用程序不可见。

在上面的轻型结构中，程序没有使用 Transaction / TransactionFactory 或者 ConnectionProvider API,直接和 JTA/JDBC 对话了。

2.2. 持久化对象标识（Persistent Object Identity ）

应用程序可能同时在两个不同的 session 中存取同一个持久化对象。然而，两个 Session 实例是不可能共享一个持久化类的实例的。有两种不同的用来辨别对象是否相同的方法。

Persistent Identity，持久化辨别

```
foo.getId().equals( bar.getId() )
```

JVM Identity， JVM 辨别

```
foo==bar
```

对于同一个*特定的*Session 返回的对象来说，这二者是等价的。然而，当程序并行在两个不同的 session 中访问含义上“相同”（持久化辨别）的商业对象时，两个对象实例从 JVM 的角度上来看却是“不同”的（JVM 辨别）

这种方式把并行访问（应用程序不需要对任何商业对象进行同步，只要求遵循每个 Session 一个线程的原则)和对象辨别(在应用程序的一个 session 之中，可以安全的用==来比较对象)的难题留给了 Hibernate 和数据库。

2.3. JMX 集成

JMX 是用来管理 Java 组件的 J2EE 标准。Hibernate 可以被标准的 JMX Mbean 管理，但是因为大多数程序还没有支持 JMX,Hibernate 也支持一些非标准的配置方式。

请查阅 Hibernate 网站，可以得到关于如何在 JBOSS 中把 Hibernate 配置成为一个 JMX 组件的更多信息。

2.4. JCA 支持

Hibernate 也可以被配置成为一个 JCA 连接器。更多细节，请参阅网站。

Chapter 3. SessionFactory 配置

因为 Hibernate 被设计为可以在许多不同环境下工作，所以它有很多配置参数。幸运的是，大部分都已经有默认值了，Hibernate 发行包中还附带有示例的 hibernate.properties 文件，它演示了一些可变的参数。

3.1. 可编程配置方式

net.sf.hibernate.cfg.Configuration 的一个实例代表了应用程序中所有的 Java 类到关系数据库的映射的集合。这些映射是从一些 XML 映射文件中编译得来的。你可以得到一个 Configuration 的实例，直接实例化它即可。下面有一个例子，用来从两个 XML 配置文件中的映射中初始化：

```
Configuration cfg = new Configuration()
    .addFile("Vertex.hbm.xml")
    .addFile("Edge.hbm.xml");
```

另外一个（或许是更好的）方法是让 Hibernate 自行用 getResourceAsStream()来装载映射文件。

```
Configuration cfg = new Configuration()
    .addClass(eg.Vertex.class)
    .addClass(eg.Edge.class);
```

Hibernate 就会在 classpath 中寻找叫做/eg/Vertex.hbm.xml、/eg/Edge.hbm.xml 的映射文件。这种方法取消了所有对文件名的硬编码。

Configuration 也可以指定一些可选的配置项。

```
Properties props = new Properties();
...
Configuration cfg = new Configuration()
    .addClass(eg.Vertex.class)
    .addClass(eg.Edge.class)
    .setProperties(props);
```

Configuration 是仅在配置期使用的对象，从第一个 SessionFactory 开始建立的时候，它就失效了。

3.2. 获取 SessionFactory

当所有的映射都被 Configuration 解析之后，应用程序为了得到 Session 实例，必须先得到它的工厂。这个工厂应该是被应用程序的所有线程共享的。当然，Hibernate 并不禁止你的程序实例化多个 SessionFactory。在你使用不止一个数据库的时候，这就有用了。

```
SessionFactory sessions = cfg.buildSessionFactory();
```

3.3. 用户自行提供 JDBC 连接

SessionFactory 可以使用一个用户自行提供的 JDBC 连接来打开一个 Session。这种设计可以让应用程序来自己管理 JDBC 连接。应用程序必须小心，不能在同一个连接上打开多个并行的 session。

```
java.sql.Connection conn = datasource.getConnection();
Session sess = sessions.openSession(conn);

// start a new transaction (optional)
Transaction tx = sess.beginTransaction();
```

上面的最后一行是可选的——应用程序也可能选择自行管理 JTA 或者 JDBC 事务。当然，假若你使用 Hibernate Transaction，你的客户代码就可以从底层的实现中抽象出来了。（比如说，你可以将来在需要的时候切换到 CORBA 连接，而不需要更改程序代码。）

3.4. Hibernate 提供的 JDBC 连接

另一种方法就是，你可以让 SessionFactory 替你打开连接。SessionFactory 必须事先知道连接的参数，有几种不同的方法设置参数：

- 传递一个 java.util.Properties 到 Configuration.setProperties()方法。
- 在 classpath 的根目录中提供 hibernate.properties 文件。
- 通过 java -Dproperty=value 指定使用系统属性。
- 在 hibernate.cfg.xml 文件中包含<property>元素。详情见后。

如果你使用这种方法，打开一个 Session 是非常简单的：

```
Session sess = sessions.openSession(); // obtain a JDBC connection
and
                                       // instantiate a new Session
// start a new transaction (optional)
Transaction tx = sess.beginTransaction();
```

所有的 Hibernate 属性名和约束都在 net.sf.hibernate.cfg.Environment 类中定义。我们讨论一下最重要的几项设置：

假若你设置了如下的属性，Hibernate 会使用 java.sql.DriverManager 来得到连接，并建立连接池：

Table 3.1. Hibernate JDBC 属性

属性名	用途
hibernate.connection.driver_class	*jdbc 驱动类*
hibernate.connection.url	*jdbc URL*
hibernate.connection.username	*数据库用户名*
hibernate.connection.password	*数据库用户密码*
hibernate.connection.pool_size	*连接池容量最大数*

Hibernate 的连接池算法是非常可配置的。它的用途是让你上手，但是*并非让你在生产系统中使用的*，甚至不是用来做性能测试的。

C3P0 是随 Hibernate 发行包一起发布的一个开放源代码 JDBC 连接池，你可以在 lib 目录中找到。假若你设置了 hibernate.c3p0.* 属性，Hibernate 会使用内置的 C3P0ConnectionProvider 作为连接池。 对 Apache DBCP 和 Proxool 的支持也是内置的。你必须设置 hibernate.dbcp.*属性 （DBCP 连接池属性）和 hibernate.dbcp.ps.* (DBCP 语句缓存属性)才能使用 DBCPConnectionProvider。要知道它们的含义，请查阅 Apache commons-pool 的文档。如果你想要用 Proxool,你需要设置 hibernate.proxool.*系列属性。

在 Application Server 内使用时，Hibernate 可以从 JNDI 中注册的 javax.sql.Datasource 取得连接。需要设置如下属性：

Table 3.2. Hibernate 数据源（Datasource）属性

属性名	用途
hibernate.connection.datasource	*datasource JNDI 名字*
hibernate.jndi.url	*JNDI 提供者的 URL*（可选）
hibernate.jndi.class	*JNDI InitialContextFactory 的类名*（可选）
hibernate.connection.username	*数据库用户名*（可选）
hibernate.connection.password	*数据库密码*（可选）

3.5. 其它配置属性

下面是一些在运行时可以改变 Hibernate 行为的其他配置。所有这些都是可选的，也有合理的默认值。

系统级别的配置只能通过 java -Dproperty=value 或者在 hibernate.properties 文件中配置，而不能通过传递给 Configuration 的 Properties 实例来配置。

Table 3.3. Hibernate 配置属性

属性名	用途
hibernate.dialect	*Hibernate 方言（Dialect）的类名 - 可以让 Hibernate 使用某些特定的数据库平台的特性* 取值. full.classname.of.Dialect
hibernate.default_schema	*在生成的 SQL 中，scheml/tablespace 的全限定名* 取值. SCHEMA_NAME
hibernate.session_factory_name	*把 SessionFactory 绑定到 JNDI 中去.* 取值. jndi/composite/name
hibernate.use_outer_join	*允许使用外连接抓取.* 取值. true | false
hibernate.max_fetch_depth	*设置外连接抓取树的最大深度* 取值. 建议设置为 0 到 3 之间
hibernate.jdbc.fetch_size	*一个非零值，用来决定 JDBC 的获取量大小。(会调用 calls Statement.setFetchSize()).*
hibernate.jdbc.batch_size	*一个非零值，会开启 Hibernate 使用 JDBC2 的批量更新功能* 取值. 建议值在 5 和 30 之间。
hibernate.jdbc.use_scrollable_resultset	*允许 Hibernate 使用 JDBC2 提供的可滚动结果集。只有在使用用户自行提供的连接时，这个参数才是必需的。否则 Hibernate 会使用连接的元数据(metadata)。* 取值. true | false
hibernate.jdbc.use_streams_for_binary	*在从 JDBC 读写 binary（二进制）或者 serializable（可序列化）类型时,是否使用 stream(流）.* 这是一个系统级别的属性。 取值. true | false
hibernate.cglib.use_reflection_optimizer	*是否使用 CGLIB 来代替运行时反射操作。（系统级别属性，默认为在可能时都使用 CGLIB）.在调试的时候有时候使用反射会有用。* 取值. true | false
hibernate.jndi.*<propertyName>*	*把 propertyName 这个属性传递到 JNDI*

属性名	用途
	InitialContextFactory 去（可选）
hibernate.connection.isolation	*事务隔离级别*（可选） 取值. 1，2，4，8
hibernate.connection.<*propertyName*>	*把 propertyName 这个 JDBC 属性传递到 DriverManager.getConnection()去.*
hibernate.connection.provider_class	*指定一个自定义的 ConnectionProvider 类名* 取值. classname.of.ConnectionProvider
hibernate.cache.provider_class	*指定一个自定义的 CacheProvider 缓存提供者的类名* 取值. classname.of.CacheProvider
hibernate.cache.use_minimal_puts	*优化第二层缓存操作，减少写操作，代价是读操作更频繁（对于集群缓存很有用）* 取值. true|false
hibernate.cache.use_query_cache	*打开查询缓存* 取值. true|false
hibernate.cache.region_prefix	*用于第二层缓存区域名字的前缀* 取值. prefix
hibernate.transaction.factory_class	*指定一个自定义的 TransactionFactory 类名，Hibernate Transaction API 将会使用* 取值. classname.of.TransactionFactory
jta.UserTransaction	*JTATransactionFactory 用来获取 JTA UserTransaction 的 JNDI 名* 取值. jndi/composite/name
hibernate.transaction.manager_lookup_class	*TransactionManagerLookup 的类名 - 当在 JTA 环境中，JVM 级别的缓存被打开的时候使用* 取值. classname.of.TransactionManagerLookup
hibernate.query.substitutions	*把 Hibernate 查询中的一些短语替换为 SQL 短语（比如说短语可能是函数或者字符）.* 取值. hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC

属性名	用途
hibernate.show_sql	*把所有的 SQL 语句都输出到控制台(可以作为 log 功能的一个替代)* 取值. true | false
hibernate.hbm2ddl.auto	*自动输出 schema 创建 DDL 语句.* 取值. update | create | create-drop

3.5.1. SQL Dialects SQL 方言

你总是可以为你的数据库设置一个 hibernate.dialect 方言，它是 net.sf.hibernate.dialect.Dialect 的一个子类。如果你不需要使用基于 native 或者 sequence 的主键自动生成算法，或者悲观锁定（使用 Session.lock() 或 Query.setLockMode())的话，方言就可以不必指定。然而，假若你指定了一个方言，Hibernate 会为上面列出的一些属性使用特殊默认值，省得你手工指定它们。

Table 3.4. Hibernate SQL 方言 (**hibernate.dialect**)

RDBMS	方言
DB2	net.sf.hibernate.dialect.DB2Dialect
MySQL	net.sf.hibernate.dialect.MySQLDialect
SAP DB	net.sf.hibernate.dialect.SAPDBDialect
Oracle (所有版本)	net.sf.hibernate.dialect.OracleDialect
Oracle 9	net.sf.hibernate.dialect.Oracle9Dialect
Sybase	net.sf.hibernate.dialect.SybaseDialect
Sybase Anywhere	net.sf.hibernate.dialect.SybaseAnywhereDialect
Progress	net.sf.hibernate.dialect.ProgressDialect
Mckoi SQL	net.sf.hibernate.dialect.MckoiDialect
Interbase	net.sf.hibernate.dialect.InterbaseDialect
Pointbase	net.sf.hibernate.dialect.PointbaseDialect
PostgreSQL	net.sf.hibernate.dialect.PostgreSQLDialect
HypersonicSQL	net.sf.hibernate.dialect.HSQLDialect
Microsoft SQL Server	net.sf.hibernate.dialect.SybaseDialect
Ingres	net.sf.hibernate.dialect.IngresDialect
Informix	net.sf.hibernate.dialect.InformixDialect
FrontBase	net.sf.hibernate.dialect.FrontbaseDialect

3.5.2. 外连接抓取（Outer Join Fetching ）

如果你的数据库支持 ANSI 或者 Oracle 风格的外连接，*外连接抓取*可能提高性能，因为可以限制和数据库交互的数量（代价是数据库自身进行了更多的工作）。外连接抓取允许你在一个 select 语句中就可以得到一个由多对一或者一对一连接构成的对象图。

默认情况下，抓取在叶对象，拥有代理的对象或者产生对自身的引用时终止。对一个*特定关联*来说，通过在 XML 映射文件中设置 outer-join 属性可以控制是否开启抓取功能。也可以设置 hibernate.use_outer_join 为 false 来全局关闭此功能。 你也可以通过 hibernate.max_fetch_depth 来设置抓取得对象图的最大深度。

3.5.3. 二进制流

Oracle 限制通过它的 JDBC 驱动传递的 byte 数组的大小。如果你希望使用很大数量的 binary 或者 serializable 类型的话，你需要打开 hibernate.jdbc.use_streams_for_binary。*这只能通过 JVM 级别设定*

3.5.4. 在控制台记录 SQL

hibernate.show_sql 强制 Hibernate 把每一句 SQL 语句都写到控制台。这是作为打开 log 的一个简易替代。

3.5.5. 自定义 **ConnectionProvider**

你可以自定义你的获取 JDBC 连接的策略，只需要实现 net.sf.hibernate.connection.ConnectionProvider 接口。在 hibernate.connection.provider_class 设置你自己的实现的类名。

3.5.6. 常用数据库属性

几个配置属性影响除了 DatasourceConnectionProvider 之外的所有内置连接提供者.它们是： hibernate.connection.driver_class，hibernate.connection.url，hibernate.connection.username and hibernate.connection.password.

hibernate.connection.isolation 应该指定为一个整数值。(查阅 java.sql.Connection 可以得到值的含义，但注意大多数数据库不会支持所有的隔离级别。)

专用的连接属性可以通过在"hibernate.connnection"后面加上属性名来指定。比如，你可以通过 hibernate.connnection.charSet 指定一个 charSet。

3.5.7. 自定义 **CacheProvider**

通过实现 net.sf.hibernate.cache.CacheProvider 接口，你可以整合一个 JVM 级别（或者集群的）缓存进来。你可以通过 hibernate.cache.provider_class 选择某个子定义的实现。

3.5.8. 事务策略

如果你希望使用 Hibernate 的 Transaction API,你必须通过 hibernate.transaction.factory_class 属性指定一个 Transaction 实例的工厂类。 内置的两个标准选择是：

net.sf.hibernate.transaction.JDBCTransactionFactory

使用数据库(JDBC)事务

net.sf.hibernate.transaction.JTATransactionFactory

使用 JTA(假若已经存在一个事务，Session 会在这个上下文中工作，否则会启动一个新的事务。)

你也可以自行定义你的事务策略（比如说，一个 CORBA 事务服务）。

如果你希望在 JTA 环境中为可变数据使用 JVM 级别的缓存，你必须指定一个获取 JTA TransactionManager 的策略。

Table 3.5. JTA TransactionManagers

事务工厂类	Application Server
net.sf.hibernate.transaction.JBossTransactionManagerLookup	JBoss
net.sf.hibernate.transaction.WeblogicTransactionManagerLookup	Weblogic
net.sf.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
net.sf.hibernate.transaction.OrionTransactionManagerLookup	Orion
net.sf.hibernate.transaction.ResinTransactionManagerLookup	Resin
net.sf.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
net.sf.hibernate.transaction.JOnASTransactionManagerLookup	JOnAS
net.sf.hibernate.transaction.JRun4TransactionManagerLookup	JRun4

3.5.9. 绑定 SessionFactory 到 JNDI

假若你希望把 SessionFactory 绑定到一个 JNDI 命名空间，用 hibernate.session_factory_name 这个属性指定一个名字（比如，java:comp/env/hibernate/SessionFactory）。如果这个属性省略了，SessionFactory 不会被绑定到 JNDI。（在一个只读的 JNDI 默认值实现的环境中，这特别有用。比如，Tomcat。）

当把 SessionFactory 绑定到 JNDI,Hibernate 会使用 hibernate.jndi.url,hibernate.jndi.class 的值来获得一个初始化上下文的实例。如果他们没有指定，就会使用默认的 InitialContext。

如果你选择使用 JNDI,EJB 或者其他工具类就可以通过 JNDI 查询得到 SessionFactory。

3.5.10. 查询语言替换

你可以使用 hibernate.query.substitutions 定义新的 Hibernate 查询短语。比如说：

```
hibernate.query.substitutions true=1, false=0
```

会在生成的 SQL 中把短语 true 和 false 替换成整数值。

```
hibernate.query.substitutions toLowercase=LOWER
```

这可以让你重新命名 SQL 的 LOWER 函数。

3.6. XML 配置文件

另一种配置属性的方法是把所有的配置都放在一个名为 hibernate.cfg.xml 的文件中。这个文件应该放在你的 CLASSPATH 的根目录中。

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 2.0//EN"

 "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:comp/env/hibernate/SessionFactory">

        <!-- properties -->
        <property
name="connection.datasource">my/first/datasource</property>
        <property
name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="use_outer_join">true</property>
        <property
name="transaction.factory_class">net.sf.hibernate.transaction.JTATran
sactionFactory</property>
        <property
name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="eg/Edge.hbm.xml"/>
        <mapping resource="eg/Vertex.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

配置 Hibernate 只需如此简单：

```
SessionFactory sf = new
Configuration().configure().buildSessionFactory();
```

你可以使用另外一个名字的配置文件：

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

3.7. Logging

通过 Apache commons-logging，Hibernate 记录很多事件。commons-logging 服务会直接输出到 Apache log4j(如果你把 log4j.jar 放在你的 classpath 里),或者 JDK1.4 logging（如果你运行 JDK 1.4 或以上版本）。你可以从 http://jakarta.apache.org 下载 log4j。要使用 log4j,你需要在你的 classpath 中放置一个 log4j.properties 文件。Hibernate 发行包中包含一个示例的 properties 配置文件。

我们强烈建议你熟悉 Hibernate 的 log 信息。Hibernate 的很多工作都会尽量详细的留下 log,也没有让它变的难以阅读。这是用来解决问题的最基本的设施。

3.8. NamingStrategy（命名策略）

net.sf.hibernate.cfg.NamingStrategy 接口允许你对数据库对象指定“命名标准”。你可以定义从 Java 标识符自动生成数据库标识符的规则，或者是映射文件中给出的“逻辑”字段名和表名处理为“物理”表名和字段名的规则。这个功能可以让映射文件变得简洁，消除无用的噪音（比如 TBL_前缀等）。Hibernate 使用的默认策略是几乎什么都不错。你可以在增加映射(add mappings)之前调用 Configuration.setNamingStrategy()来指定不同的策略。

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Vertex.hbm.xml")
    .addFile("Edge.hbm.xml")
    .buildSessionFactory();
```

net.sf.hibernate.cfg.ImprovedNamingStrategy 是一个内置的策略，对某些程序，你可以把它作为改造的起点。

Chapter 4. 持久化类(Persistent Classes)

4.1. 简单示例

大多数 java 程序需要一个持久化类的表示方法。

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setMate(Cat mate) {
        this.mate = mate;
```

```
    }
    public Cat getMate() {
        return mate;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }
    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }
}
```

有三条主要的规则：

4.1.1. 为持久化字段声明访问器(accessors)和是否可变的标志(mutators)

`Cat` 为它的所有可持久化字段声明了访问方法。很多其他 ORM 工具直接对实例变量进行持久化。我们相信在持久化机制中不限定这种实现细节，感觉要好得多。Hibernate 对 JavaBeans 风格的属性实行持久化，采用如下格式来辨认方法：`getFoo`，`isFoo` 和 `setFoo`。

属性*不一定*需要声明为 public 的。Hibernate 可以对 default,`protected` 或者 `private` 的 get/set 方法对的属性一视同仁地执行持久化。

4.1.2. 实现一个默认的构造方法（constructor）

Cat 有一个显式的无参数默认构造方法。所有的持久化类都必须具有一个默认的构造方法（可以不是 public 的），这样的话 Hibernate 就可以使用 Constructor.newInstance()来实例化它们。

4.1.3. 提供一个标识属性（identifier property）（可选）

Cat 有一个属性叫做 id。这个属性包含了数据库表中的主关键字字段。这个属性可以叫任何名字，其类型可以是任何的原始类型、原始类型的包装类型、java.lang.String 或者是 java.util.Date。（如果你的老式数据库表有联合主键，你甚至可以用一个用户自定义的类，其中每个属性都是这些类型之一。参见后面的关于联合标识符的章节。）

用于标识的属性是可选的。你可以不管它，让 Hibernate 内部来追踪对象的识别。当然，对于大多数应用程序来说，这是一个好的（也是很流行的）设计决定。

更进一步，一些功能只能对声明了标识属性的类起作用：

- 级联更新（Cascaded updates）（参阅“自管理生命周期的对象（Lifecycle Objects）”）
- Session.saveOrUpdate()

我们建议你对所有的持久化类采取同样的名字作为标识属性。更进一步，我们建议你使用一个可以为空（也就是说，不是原始类型）的类型。

4.1.4. 建议使用不是 final 的类（可选）

Hibernate 的关键功能之一，*代理（proxies）*，要求持久化类不是 final 的，或者是一个全部方法都是 public 的接口的具体实现。

你可以对一个 final 的，也没有实现接口的类执行持久化，但是不能对它们使用代理——多多少少会影响你进行性能优化的选择。

4.2. 继承（Inheritance ）

子类也必须遵守第一条和第二条规则。它从 Cat 继承了标识属性。

```
package eg;

public class DomesticCat extends Cat {
        private String name;

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

4.3. 持久化生命周期（Lifecycle）中的回调（Callbacks）

作为一个可选的步骤，可持久化类可以实现 Lifecycle 接口，它可以提供一些用于回调的方法，可以让持久化对象在 save 或 load 之后，或者在 delete 或 update 之前进行必要的初始化与清除步骤。

```
public interface Lifecycle {
        public boolean onSave(Session s) throws CallbackException;
❶
        public boolean onUpdate(Session s) throws CallbackException;
❷
        public boolean onDelete(Session s) throws CallbackException;
❸
        public void onLoad(Session s, Serializable id);
❹
}
```

❶ onSave – 在对象即将被 save 或者 insert 的时候回调

❷ onUpdate – 在对象即将被 update 的时候回调（也就是对象被传递给 Session.update()的时候）

❸ onDelete – 在对象即将被 delete(删除)的时候回调

❹ onLoad – 在对象刚刚被 load(装载)后的时候回调

onSave(), onDelete() 和 onUpdate() 可以被用来级联保存或者删除依赖的对象。这种做法是在映射文件中声明级联操作外的另外一种选择。onLoad()可以用来让对象从其持久化（当前）状态中初始化某些暂时的属性。不能用这种方式来装载依赖的对象，因为可能无法在此方法内部调用 Session 接口。 onLoad(), onSave()和 onUpdate()另一种用法是用来在当前 Session 中保存一个引用，已备后用。

请注意 onUpdate()并不是在每次对象的持久化状态被更新的时候就被调用的。它只在处于尚未被持久化的对象被传递给 Session.update()的时候才会被调用。

如果 onSave(), onUpdate() 或者 onDelete()返回 true，那么操作就被悄悄地取消了。如果其中抛出了 CallbackException 异常，操作被取消，这个异常会被继续传递给应用程序。

请注意 onSave()是在标识符已经被赋予对象后调用的，除非是使用本地(native)方式生成关键字的。

4.4. 合法性检查（Validatable）

如果持久化类需要在保存其持久化状态前进行合法性检查，它可以实现下面的接口：

```
public interface Validatable {
        public void validate() throws ValidationFailure;
}
```

如果发现对象违反了某条规则，应该抛出一个 ValidationFailure 异常。在 Validatable 实例的 validate()方法内部不应该改变它的状态。

和 Lifecycle 接口的回调方法不同，validate()可能在任何时间被调用。应用程序不应该把 validate()调用和商业功能联系起来。

4.5. XDoclet 示例

下一节中我们将会展示 Hibernate 映射是如何用简单的，可阅读的 XML 格式表达的。很多 Hibernate 用户喜欢使用 XDoclet 的@hibernate.tags 标签直接在源代码中嵌入映射信息。我

们不会在这份文档中讨论这个话题，因为严格的来说这属于 XDoclet 的一部分。但我们仍然在这里给出一份带有 XDoclet 映射的 Cat 类的示例。

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /**
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="MATE_ID"
     */
    public Cat getMate() {
        return mate;
    }
    void setMate(Cat mate) {
        this.mate = mate;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
```

```
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  lazy="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

Chapter 5. O/R Mapping 基础

5.1. 映射声明(Mapping declaration)

对象和关系数据库之间的映射是用一个 XML 文档(XML document)来定义的。这个映射文档被设计为易读的，并且可以手工修改。映射语言是以 Java 为中心的，意味着映射是按照持久化类的定义来创建的，而非表的定义。

请注意，虽然很多 Hibernate 用户选择手工定义 XML 映射文档，也有一些工具来生成映射文档，包括 XDoclet,Middlegen 和 AndroMDA.

让我们从一个映射的例子开始：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat" discriminator-value="D">
                        <property name="name" type="string"/>
                </subclass>
        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

我们现在开始讨论映射文档的内容。我们只描述 Hibernate 在运行时用到的文档元素和属性。映射文档还包括一些额外的可选属性和元素，它们在使用 schema 导出工具的时候会影响导出的数据库 schema 结果。（比如， `not-null` 属性。）

5.1.1. Doctype

所有的 XML 映射都需要定义如上所示的 doctype。DTD 可以从上述 URL 中获取，或者在 `hibernate-x.x.x/src/net/sf/hibernate` 目录中，或 `hibernate.jar` 文件中找到。Hibernate 总是会在它的 classptah 中首先搜索 DTD 文件。

5.1.2. hibernate-mapping

这个元素包括三个可选的属性。`schema` 属性，指明了这个映射所引用的表所在的 schema 名称。假若指定了这个属性，表名会加上所指定的 schema 的名字扩展为全限定名。假若没有指定，表名就不会使用全限定名。`default-cascade` 指定了未明确注明 `cascade` 属性的 Java 属性和集合类 Java 会采取什么样的默认级联风格。`auto-import` 属性默认让我们在查询语言中可以使用非全限定名的类名。

```
<hibernate-mapping
         schema="schemaName"                          ❶
         default-cascade="none|save-update"           ❷
         auto-import="true|false"                     ❸
```

```
        package="package.name"                                ❹
/>
```

❶ schema (可选): 数据库 schema 名称。

❷ default-cascade (可选 - 默认为 none): 默认的级联风格。

❸ auto-import (可选 - 默认为 true): 指定是否我们可以在查询语言中使用非全限定的类名（仅限于本映射文件中的类）。

❹ package (可选): 指定一个包前缀，如果在映射文档中没有指定全限定名，就使用这个包名。

假若你有两个持久化类，它们的非全限定名是一样的（就是在不同的包里面--译者注），你应该设置 auto-import="false"。假若说你把一个“import 过”的名字同时对应两个类，Hibernate 会抛出一个异常。

5.1.3. class

你可以使用 class 元素来定义一个持久化类：

```
<class
        name="ClassName"                                      ❶
        table="tableName"                                     ❷
        discriminator-value="discriminator_value"             ❸
        mutable="true|false"                                  ❹
        schema="owner"                                        ❺
        proxy="ProxyInterface"                                ❻
        dynamic-update="true|false"                           ❼
        dynamic-insert="true|false"                           ❽
        select-before-update="true|false"                     ❾
        polymorphism="implicit|explicit"                      ❿
        where="arbitrary sql where condition"                 (11)
        persister="PersisterClass"                            (12)
        batch-size="N"                                        (13)
        optimistic-lock="none|version|dirty|all"              (14)
        lazy="true|false"                                     (15)
/>
```

❶ name: 持久化类（或者接口）的 Java 全限定名。

❷ table: 对应的数据库表名。

❸ discriminator-value（辨别值） (可选 - 默认和类名一样)：一个用于区分不同的子类的值，在多态行为时使用。

❹ mutable（可变） (可选, 默认值为 true): 表明该类的实例可变（不可变）。

❺ schema (可选): 覆盖在根<hibernate-mapping>元素中指定的 schema 名字。

❻ proxy (可选): 指定一个接口，在延迟装载时作为代理使用。你可以在这里使用该类自己的名字。

❼ dynamic-update（动态更新） (可选，默认为 false): 指定用于 UPDATE 的 SQL 将会在运行时动态生成，并且只更新那些改变过的字段。

❽ dynamic-insert（动态插入） (可选, 默认为 false): 指定用于 INSERT 的 SQL 将会在运行时动态生成，并且只包含那些非空值字段。

❾ select-before-update (可选，默认值为 false): 指定 Hibernate 除非确定对象的确被修改了，*不会*执行 SQL UPDATE 操作。在特定场合（实际上，只会发生在一个临时对象关联到一个新的 session 中去，执行 update()的时候），这说明 Hibernate 会在 UPDATE 之前执行一次额外的 SQL SELECT 操作，来决定是否应该进行 UPDATE。

❿ polymorphism（多形，多态）(可选, 默认值为 implicit (隐式)): 界定是隐式还是显式的使用查询多态。

(11) where (可选) 指定一个附加的 SQLWHERE 条件，在抓取这个类的对象时会一直增加这个条件。

(12) persister (可选): 指定一个定制的 ClassPersister。

(13) batch-size (可选,默认是 1) 指定一个用于根据标识符抓取实例时使用的"batch size"（批次抓取数量）。

(14) optimistic-lock（乐观锁定）(可选，默认是 version): 决定乐观锁定的策略。

(15) lazy（延迟）(可选): 假若设置 lazy="true"，就是设置这个类自己的名字作为 proxy 接口的一种等价快捷形式。

若指明的持久化类实际上是一个接口，也可以被完美地接受。其后你可以用<subclass>来指定该接口的实际实现类名。你可以持久化任何 *static*（静态的）内部类。记得应该使用标准的类名格式，就是说比如：Foo$Bar。

不可变类，mutable="false"不可以被应用程序更新或者删除。这可以让 Hibernate 做一些小小的性能优化。

可选的 proxy 属性可以允许延迟加载类的持久化实例。Hibernate 开始会返回实现了这个命名接口的 CGLIB 代理。当代理的某个方法被实际调用的时候，真实的持久化对象才会被装载。参见下面的“用于延迟装载的代理”。

Implicit (隐式)的多态是指，如果查询中给出的是任何超类、该类实现的接口或者该类的名字，都会返回这个类的实例；如果查询中给出的是子类的名字，则会返回子类的实例。*Explicit* （显式）的多态是指，只有在查询中给出的明确是该类的名字时才会返回这个类的实例；同时只有当在这个<class>的定义中作为<subclass>或者<joined-subclass>出现的子类，才会可能返回。 大多数情况下，默认的 polymorphism="implicit"都是合适的。 显式的多态在有两个不同的类映射到同一个表的时候很有用。（允许一个“轻型”的类，只包含部分表字段）。

persister 属性可以让你定制这个类使用的持久化策略。你可以指定你自己实现的 net.sf.hibernate.persister.EntityPersister 的子类，你甚至可以完全从头开始编写一个 net.sf.hibernate.persister.ClassPersister 接口的实现，可能是用储存过程调用、序列化到文件或者 LDAP 数据库来实现的。参阅 net.sf.hibernate.test.CustomPersister，这是一个简单的例子（“持久化”到一个 Hashtable）。

请注意 dynamic-update 和 dynamic-insert 的设置并不会继承到子类，所以在<subclass>或者<joined-subclass>元素中可能需要再次设置。这些设置是否能够提高效率要视情形而定。请用你的智慧决定是否使用。

使用 select-before-update 通常会降低性能. 当是在防止数据库不必要的触发 update 触发器，这就很有用了。

如果你打开了 dynamic-update，你可以选择几种乐观锁定的策略：

- version（版本检查） 检查 version/timestamp 字段
- all（全部） 检查全部字段
- dirty（脏检查）只检察修改过的字段
- none（不检查）不使用乐观锁定

我们*非常*强烈建议你在 Hibernate 中使用 version/timestamp 字段来进行乐观锁定。对性能来说，这是最好的选择，并且这也是唯一能够处理在 session 外进行操作的策略（就是说，当使用 Session.update()的时候）。

5.1.4. id

被映射的类*必须*声明对应数据库表主键字段。大多数类有一个 JavaBeans 风格的属性，为每一个实例包含唯一的标识。<id> 元素定义了该属性到数据库表主键字段的映射。

```
<id
        name="propertyName"                         ❶
        type="typename"                             ❷
        column="column_name"                        ❸
        unsaved-value="any|none|null|id_value"      ❹
        access="field|property|ClassName">          ❺

        <generator class="generatorClass"/>
</id>
```

❶ name（可选）：标识属性的名字。

❷ type（可选）：标识 Hibernate 类型的名字。

❸ column（可选 - 默认为属性名）：主键字段的名字。

❹ unsaved-value（可选 - 默认为 null）：一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。这可以把这种实例和从以前的 session 中装载过（可能又做过修改--译者注）但未再次持久化的实例区分开来。

❺ access（可选 - 默认为 property）：Hibernate 用来访问属性值的策略。

如果 name 属性不存在，会认为这个类没有标识属性。

unsaved-value 属性很重要！如果你的类的标识属性不是默认为 null 的，你应该指定正确的默认值。

还有一个另外的<composite-id>声明可以访问旧式的多主键数据。我们强烈不鼓励使用这种方式。

5.1.4.1. generator

必须声明的<generator>子元素是一个 Java 类的名字，用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数，用<param>元素来传递。

```
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="net.sf.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
```

```
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

所有的生成器都实现 net.sf.hibernate.id.IdentifierGenerator 接口。这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然，Hibernate 提供了很多内置的实现。下面是一些内置生成器的快捷名字：

increment（递增）

用于为 long, short 或者 int 类型生成唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。 *在集群下不要使用。*

identity

对 DB2,MySQL, MS SQL Server, Sybase 和 HypersonicSQL 的内置标识字段提供支持。返回的标识符是 long, short 或者 int 类型的。

sequence （序列）

在 DB2,PostgreSQL, Oracle, SAP DB, McKoi 中使用序列（sequence)，而在 Interbase 中使用生成器(generator)。返回的标识符是 long, short 或者 int 类型的。

hilo （高低位）

使用一个高/低位算法来高效的生成 long, short 或者 int 类型的标识符。给定一个表和字段（默认分别是是 hibernate_unique_key 和 next）作为高位值得来源。高/低位算法生成的标识符只在一个特定的数据库中是唯一的。*在使用 JTA 获得的连接或者用户自行提供的连接中，不要使用这种生成器。*

seqhilo（使用序列的高低位）

使用一个高/低位算法来高效的生成 long, short 或者 int 类型的标识符，给定一个数据库序列（sequence)的名字。

uuid.hex

用一个 128-bit 的 UUID 算法生成字符串类型的标识符。在一个网络中唯一（使用了 IP 地址）。UUID 被编码为一个 32 位 16 进制数字的字符串。

uuid.string

使用同样的 UUID 算法。UUID 被编码为一个 16 个字符长的任意 ASCII 字符组成的字符串。*不能使用在 PostgreSQL 数据库中*

native（本地）

根据底层数据库的能力选择 identity, sequence 或者 hilo 中的一个。

assigned（程序设置）

让应用程序在 save()之前为对象分配一个标示符。

foreign（外部引用）

使用另外一个相关联的对象的标识符。和<one-to-one>联合一起使用。

5.1.4.2. 高/低位算法（Hi/Lo Algorithm）

hilo 和 seqhilo 生成器给出了两种 hi/lo 算法的实现，这是一种很令人满意的标识符生成算法。第一种实现需要一个“特殊”的数据库表来保存下一个可用的“hi”值。第二种实现使用一个 Oracle 风格的序列（在被支持的情况下）。

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

很不幸，你在为 Hibernate 自行提供 Connection，或者 Hibernate 使用 JTA 获取应用服务器的数据源连接的时候无法使用 hilo 。Hibernate 必须能够在一个新的事务中得到一个"hi"值。在 EJB 环境中实现 hi/lo 算法的标准方法是使用一个无状态的 session bean。

5.1.4.3. UUID 算法（UUID Algorithm ）

UUID 包含：IP 地址，JVM 的启动时间（精确到 1/4 秒），系统时间和一个计数器值（在 JVM 中唯一）。在 Java 代码中不可能获得 MAC 地址或者内存地址，所以这已经是我们在不使用 JNI 的前提下的能做的最好实现了。

不要试图在 PostgreSQL 中使用 uuid.string。

5.1.4.4. 标识字段和序列（Identity Columns and Sequences）

对于内部支持标识字段的数据库(DB2,MySQL,Sybase,MS SQL)，你可以使用 identity 关键字生成。对于内部支持序列的数据库（DB2,Oracle, PostgreSQL, Interbase, McKoi,SAP DB),你可以使用 sequence 风格的关键字生成。这两种方式对于插入一个新的对象都需要两次 SQL 查询。

```
<id name="id" type="long" column="uid">
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
</id>
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="identity"/>
</id>
```

对于跨平台开发，native 策略会从 identity，sequence 和 hilo 中进行选择，取决于底层数据库的支持能力。

5.1.4.5. 程序分配的标识符（Assigned Identifiers）

如果你需要应用程序分配一个标示符（而非 Hibernate 来生成它们），你可以使用 assigned 生成器。这种特殊的生成器会使用已经分配给对象的标识符属性的标识符值。用这种特性来分配商业行为的关键字要特别小心（基本上总是一种可怕的设计决定）。

5.1.5. composite-id 联合 ID

```
<composite-id
        name="propertyName"
        class="ClassName"
        unsaved-value="any|none">

        <key-property name="propertyName" type="typename"
column="column_name"/>
        <key-many-to-one name="propertyName" class="ClassName"
column="column_name"/>
        ......
</composite-id>
```

如果表使用联合主键，你可以把类的多个属性组合成为标识符属性。<composite-id>元素接受<key-property>属性映射和<key-many-to-one>属性映射作为子元素。

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

你的持久化类*必须*重载 equals()和 hashCode()方法，来实现组合的标识符判断等价.也必须实现 Serializable 接口。

不幸的是，这种组合关键字的方法意味着一个持久化类是它自己的标识。除了对象自己之外，没有什么方便的“把手”可用。你必须自己初始化持久化类的实例，在使用组合关键字 load()持久化状态之前，必须填充他的联合属性。我们会在 Section 7.4, “作为联合标识符(As Composite Identifiers)”章中说明一种更加方便的方法，把联合标识实现为一个独立的类，下面描述的属性只对这种备用方法有效:

- name （可选）：一个组件类型，持有联合标识（参见下一节）。
- class （可选 - 默认为通过反射(reflection)得到的属性类型）：作为联合标识的组件类名(参见下一节)。
- unsaved-value （可选 - 默认为 none)：假如被设置为非 none 的值，就表示新创建，尚未被持久化的实例将持有的值。

5.1.6. 识别器（discriminator）

在"一棵对象继承树对应一个表"的策略中,<discriminator>元素是必需的,它声明了表的识别器字段。识别器字段包含标志值，用于告知持久化层应该为某个特定的行创建哪一个子类的实例。只能使用如下受到限制的一些类型： string, character, integer, byte, short, boolean, yes_no, true_false.

```
<discriminator
        column="discriminator_column"  ❶
        type="discriminator_type"      ❷
        force="true|false"             ❸
/>
```

❶ column (可选 - 默认为 class) 识别器字段的名字

❷ type (可选 - 默认为 string) 一个 Hibernate 字段类型的名字

❸ force(强制) (可选 - 默认为 false) "强制"Hibernate 指定允许的识别器值,就算取得的所有实例都是根类的。

标识器字段的实际值是根据<class> 和<subclass>元素的 discriminator-value 得来的.

force 属性仅仅是在表包含一些未指定应该映射到哪个持久化类的时候才是有用的。这种情况不是经常会遇到。

5.1.7. 版本（version）(可选)

<version>元素是可选的，表明表中包含附带版本信息的数据。这在你准备使用 *长事务（long transactions）*的时候特别有用。（见后）

```
<version
        column="version_column"  ❶
        name="propertyName"      ❷
        type="typename"          ❸
        access="field|property|ClassName"            ❹
        unsaved-value="null|negative|undefined"      ❺
/>
```

❶ column (可选 - 默认为属性名): 指定持有版本号的字段名。

❷ name: 持久化类的属性名。

❸ type (可选 - 默认是 integer): 版本号的类型。

❹ access (可选 - 默认是 property): Hibernate 用于访问属性值的策略。

❺ unsaved-value (可选 - 默认是 undefined): 用于标明某个实例时刚刚被实例化的（尚未保存）版本属性值，依靠这个值就可以把这种情况和已经在先前的 session 中保存或装载的实例区分开来。（undefined 指明使用标识属性值进行这种判断。）

版本号必须是以下类型：long, integer, short, timestamp 或者 calendar。

5.1.8. 时间戳（timestamp ）(可选)

可选的<timestamp>元素指明了表中包含时间戳数据。这用来作为版本的替代。时间戳本质上是一种对乐观锁定的一种不是特别安全的实现。当然，有时候应用程序可能在其他方面使用时间戳。

```
<timestamp
        column="timestamp_column"          ❶
        name="propertyName"                ❷
        access="field|property|ClassName"  ❸
```

```
        unsaved-value="null|undefined"        ④
/>
```

❶ column（可选 - 默认为属性名）: 持有时间戳的字段名。

❷ name: 在持久化类中的 JavaBeans 风格的属性名，其 Java 类型是 Date 或者 Timestamp 的。

❸ access（可选 - 默认是 property）: Hibernate 用于访问属性值的策略。

❹ unsaved-value（可选 - 默认是 null）: 用于标明某个实例时刚刚被实例化的（尚未保存）版本属性值，依靠这个值就可以把这种情况和已经在先前的 session 中保存或装载的实例区分开来。（undefined 指明使用标识属性值进行这种判断。）

注意，<timestamp> 和<version type="timestamp">是等价的。

5.1.9. property

<property>元素为类声明了一个持久化的,JavaBean 风格的属性。

```
<property
        name="propertyName"                  ❶
        column="column_name"                 ❷
        type="typename"                      ❸
        update="true|false"                  ❹
        insert="true|false"                  ❹
        formula="arbitrary SQL expression"   ❺
        access="field|property|ClassName"    ❻
/>
```

❶ name: 属性的名字,以小写字母开头。

❷ column（可选 - 默认为属性名字）: 对应的数据库字段名。

❸ type（可选）: 一个 Hibernate 类型的名字。

❹ update, insert（可选 - 默认为 true）:表明在用于 UPDATE 和/或 INSERT 的 SQL 语句中是否包含这个字段。这二者如果都设置为 false 则表明这是一个“外源性（derived）”的属性，它的值来源于映射到同一个（或多个）字段的某些其他属性，或者通过一个 trigger(触发器），或者其他程序。

❺ formula（可选）: 一个 SQL 表达式，定义了这个*计算（computed）* 属性的值。计算属性没有和它对应的数据库字段。

❻ access（可选 - 默认值为 property）: Hibernate 用来访问属性值的策略。

typename 可以是如下几种：

- Hibernate 基础类型之一（比如：integer, string, character,date, timestamp, float, binary, serializable, object, blob）。
- 一个 Java 类的名字，这个类属于一种默认基础类型（比如： int, float,char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob)。
- 一个 PersistentEnum 的子类的名字。(比如：. eg.Color)。
- 一个可以序列化的 Java 类的名字。
- 一个自定义类型的类的名字。（比如： com.illflow.type.MyCustomType)。

如果你没有指定类型，Hibernarte 会使用反射来得到这个名字的属性，以此来猜测正确的 Hibernate 类型。Hibernate 会对属性读取器(getter 方法）的返回类进行解释，按照规则 2,3,4 的顺序。然而，这并不足够。 在某些情况下你仍然需要 type 属性。（比如，为了区别 Hibernate.DATE 和 Hibernate.TIMESTAMP,或者为了指定一个自定义类型。）

access 属性用来让你控制 Hibernate 如何在运行时访问属性。在默认情况下，Hibernate 会使用属性的 get/set 方法对。如果你指明 access="field",Hibernate 会忽略 get/set 方法对，直接使用反射来访问成员变量。你也可以指定你自己的策略，这就需要你自己实现 net.sf.hibernate.property.PropertyAccessor 接口，再在 access 中设置你自定义策略类的名字。

5.1.10. 多对一（many-to-one）

通过 many-to-one 元素,可以定义一种常见的与另一个持久化类的关联。这种关系模型是多对一关联。（实际上是一个对象引用。）

```
<many-to-one
        name="propertyName"                                 ❶
        column="column_name"                                ❷
        class="ClassName"                                   ❸
        cascade="all|none|save-update|delete"               ❹
        outer-join="true|false|auto"                        ❺
        update="true|false"                                 ❻
        insert="true|false"                                 ❻
        property-ref="propertyNameFromAssociatedClass"          ❼
        access="field|property|ClassName"                       ❽
/>
```

❶ name: 属性名。

❷ column (可选): 字段名。

❸ class (可选 - 默认是通过反射得到属性类型): 关联的类的名字。

❹ cascade（级联） (可选): 指明哪些操作会从父对象级联到关联的对象。

❺ outer-join（外连接） (可选 - 默认为 自动): 当设置 hibernate.use_outer_join 的时候，对这个关联允许外连接抓取。

❻ update, insert (可选 - defaults to true) 指定对应的字段是否在用于 UPDATE 和/或 INSERT 的 SQL 语句中包含。如果二者都是 false,则这是一个纯粹的“外源性（derived）”关联，它的值是通过映射到同一个（或多个）字段的某些其他属性得到的，或者通过 trigger(除法器），或者是其他程序。

❼ property-ref: (可选) 指定关联类的一个属性，这个属性将会和本外键相对应。如果没有指定，会使用对方关联类的主键。

❽ access (可选 - 默认是 property): Hibernate 用来访问属性的策略。

cascade 属性允许下列值： all, save-update, delete, none。设置除了 none 以外的其它值会传播特定的操作到关联的（子）对象中。参见后面的“Lifecycle Objects(自动管理生命周期的对象）”。

outer-join 参数允许下列三个不同值：

- auto（默认）使用外连接抓取关联（对象），如果被关联的对象没有代理(proxy)

- true 一直使用外连接来抓取关联
- false 永远不使用外连接来抓取关联

一个典型的简单 many-to-one 声明例子：

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

property-ref 属性只应该用来对付老旧的数据库系统，可能出现外键指向对方关联表的是个非主键字段（但是应该是一个惟一关键字）的情况。这是一种十分丑陋的关系模型。比如说，假设 Product 类有一个惟一的序列号，它并不是主键。（unique 属性控制 Hibernate 通过 SchemaExport 工具生成 DDL 的过程。）

```
<property name="serialNumber" unique="true" type="string"
column="SERIAL_NUMBER"/>
```

那么关于 OrderItem 的映射可能是：

```
<many-to-one name="product" property-ref="serialNumber"
column="PRODUCT_SERIAL_NUMBER"/>
```

当然，我们决不鼓励这种用法。

5.1.11. 一对一

持久化对象之间一对一的关联关系是通过 one-to-one 元素定义的。

```
<one-to-one
        name="propertyName"                          ❶
        class="ClassName"                            ❷
        cascade="all|none|save-update|delete"        ❸
        constrained="true|false"                     ❹
        outer-join="true|false|auto"                 ❺
        property-ref="propertyNameFromAssociatedClass"     ❻
        access="field|property|ClassName"                  ❼

/>
```

❶ name: 属性的名字。

❷ class (可选 - 默认是通过反射得到的属性类型)：被关联的类的名字。

❸ cascade(级联) (可选) 表明操作是否从父对象级联到被关联的对象。

❹ constrained（约束） (可选) 表明该类对应的表对应的数据库表，和被关联的对象所对应的数据库表之间，通过一个外键引用对主键进行约束。这个选项影响 save()和 delete()在级联执行时的先后顺序（也在 schema export tool 中被使用）。

❺ outer-join（外连接） (可选 - 默认为 自动)：当设置 hibernate.use_outer_join 的时候，对这个关联允许外连接抓取。

❻ property-ref: (可选) 指定关联类的一个属性，这个属性将会和本外键相对应。如果没有指定，会使用对方关联类的主键。

❻ access (可选 - 默认是 property): Hibernate 用来访问属性的策略。

有两种不同的一对一关联：

- 主键关联
- 惟一外键关联

主键关联不需要额外的表字段;两行是通过这种一对一关系相关联的，那么这两行就共享同样的主关键字值。所以如果你希望两个对象通过主键一对一关联，你必须确认它们被赋予同样的标识值!

比如说，对下面的 Employee 和 Person 进行主键一对一关联:

```
<one-to-one name="person" class="Person"/>
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Now we must ensure that the primary keys of related rows in the PERSON and EMPLOYEE tables are equal. We use a special Hibernate identifier generation strategy called foreign: 现在我们必须确保 PERSON 和 EMPLOYEE 中相关的字段是相等的。我们使用一个特别的称为 foreign 的 Hibernate 标识符生成器策略:

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

一个刚刚保存的 Person 实例被赋予和该 Person 的 employee 属性所指向的 Employee 实例同样的关键字值。

另一种方式是一个外键和一个惟一关键字对应，上面的 Employee 和 Person 的例子，如果使这种关联方式，应该表达成:

```
<many-to-one name="person" class="Person" column="PERSON_ID"
unique="true"/>
```

如果在 Person 的映射加入下面几句，这种关联就是双向的:

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. 组件（component），动态组件（dynamic-component）

<component>元素把子对象的一些元素与父类对应的表的一些字段映射起来。 然后组件可以声明它们自己的属性、组件或者集合。参见后面的“Components”一章。

```
<component
        name="propertyName"                  ❶
        class="className"                    ❷
```

```
        insert="true|false"                   ❸
        upate="true|false"                    ❹
        access="field|property|ClassName">    ❺

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

❶ name: 属性名

❷ class (可选 - 默认为通过反射得到的属性类型):组件（子）类的名字。

❸ insert: 被映射的字段是否出现在 SQL 的 INSERT 语句中?

❹ update: 被映射的字段是否出现在 SQL 的 UPDATE 语句中?

❺ access (可选 - 默认是 property): Hibernate 用来访问属性的策略。

其<property>子标签为子类的一些属性和表字段建立映射。

<component>元素允许加入一个<parent>子元素，在组件类内部就可以有一个指向其容器的实体的反向引用。

<dynamic-component>元素允许把一个 Map 映射为组件，其属性名对应 map 的键值。

5.1.13. 子类(subclass)

最后，多态持久化需要为父类的每个子类都进行声明。对于我们建议的“每一棵类继承树对应一个表”的策略来说，就需要使用<subclass>声明。

```
<subclass
        name="ClassName"                                  ❶
        discriminator-value="discriminator_value"         ❷
        proxy="ProxyInterface"                            ❸
        lazy="true|false"                                 ❹
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <property .... />
        .....
</subclass>
```

❶ name: 子类的全限定名。

❷ discriminator-value(辨别标志) (可选 - 默认为类名):一个用于区分每个独立的子类的值。

❸ proxy（代理） (可选): 指定一个类或者接口，在延迟装载时作为代理使用。

❹ lazy（延迟装载） (可选): 设置 lazy="true"是把自己的名字作为 proxy 接口的一种等价快捷方式。

每个子类都应该声明它自己的持久化属性和子类。 <version> 和<id> 属性可以从根父类继承下来。在一棵继承树上的每个子类都必须声明一个唯一的 discriminator-value。如果没有指定，就会使用 Java 类的全限定名。

5.1.14. 连接的子类（joined-subclass）

另外一种情况，如果子类是持久化到一个属于它自己的表（每一个子类对应一个表的映射策略），那么就需要使用<joined-subclass>元素。

```
<joined-subclass
        name="ClassName"            ❶
        proxy="ProxyInterface"      ❷
        lazy="true|false"                     ❸
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

        <property .... />
        .....
</subclass>
```

❶ name：子类的全限定名。

❷ proxy（可选）：指定一个类或者接口，在延迟装载时作为代理使用。

❸ lazy（延迟装载）（可选）：设置 lazy="true"是把自己的名字作为 proxy 接口的一种等价快捷方式。

这种映射策略不需要指定辨别标志（discriminator）字段。但是，每一个都必须使用<key>元素指定一个表字段包含对象的标识符。本章开始的映射可以被用如下方式重写：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping  package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat"
table="DOMESTIC_CATS">
                       <key column="CAT"/>
                        <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>
```

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

5.1.15. map, set, list, bag

集合类在后面讨论。

5.1.16. 引用（import）

假设你的应用程序有两个同样名字的持久化类，但是你不想在Hibernate查询中使用他们的全限定名。除了依赖auto-import="true"以外，类也可以被显式地“import(引用）”。你甚至可以引用没有明确被映射的类和接口。

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"                                    ❶
        rename="ShortName"                                   ❷
/>
```

❶ class: 任何Java类的全限定名。

❷ rename (可选 - 默认为类的全限定名): 在查询语句中可以使用的名字。

5.2. Hibernate 的类型

5.2.1. 实体（Entities）和值（values）

为了理解很多与持久化服务相关的Java语言级对象的行为，我们需要把它们分为两类：

实体entity 独立于任何持有实体引用的对象。与通常的Java模型相比，不再被引用的对象会被当作垃圾收集掉。实体必须被显式的保存和删除（除非保存和删除是从父实体向子实体引发的*级联*）。这和ODMG模型中关于对象通过可触及保持持久性有一些不同——比较起来更加接近应用程序对象通常在一个大系统中的使用方法。实体支持循环引用和交叉引用，它们也可以加上版本信息。

实体的持久化状态包含有指向其他实体的连接和一些*值* 类型的实例。值是原始类型、集合、组件或者特定的不可变对象。与实体不同，值（特别是集合和组件）*是*通过可触及性来进行持久化和删除的。因为值对象（和原始类型数据）是随着包含它们的实体而被持久化和删除的，它们不能够被独立的加上版本信息。值没有独立的标识，所以它们不能被两个实体或者集合共享。

所有的Hibernate对象，除了集合，都支持null语义。

直到现在，我们都一直使用"持久化对象"来代表实体。我们仍然会这么做。然而严格的来说，并不是所有用户定义的，带有持久化状态的类都是实体。*组件(component)*就是一个用户定义的类，仅仅由值语义构成。

5.2.2. 基本值类型

*基本类型*可以大致的分为：

integer, long, short, float, double, character, byte, boolean, yes_no, true_false

这些类型都对应 Java 的原始类型或者其包装类，来适合（特定厂商的）SQL 字段类型。boolean, yes_no 和 true_false 都是 Java 中 boolean 或者 java.lang.Boolean 的另外说法。

string

从 java.lang.String 到 VARCHAR (或者 Oracle 的 VARCHAR2)的映射。

date, time, timestamp

从 java.util.Date 和其子类到 SQL 类型 DATE, TIME 和 TIMESTAMP (或等价类型)的映射。

calendar, calendar_date

从 java.util.Calendar 到 SQL 类型 TIMESTAMP 和 DATE（或等价类型）的映射。

big_decimal

从 java.math.BigDecimal 到 NUMERIC (或者 Oracle 的 NUMBER 类型)的映射。

locale, timezone, currency

从 java.util.Locale, java.util.TimeZone 和 java.util.Currency 到 VARCHAR (或者 Oracle 的 VARCHAR2 类型)的映射. Locale 和 Currency 的实例被映射为它们的 ISO 代码。TimeZone 的实例被影射为它的 ID。

class

从 java.lang.Class 到 VARCHAR (或者 Oracle 的 VARCHAR2 类型)的映射。Class 被映射为它的全限定名。

binary

把字节数组（byte arrays)映射为对应的 SQL 二进制类型。

text

把长 Java 字符串映射为 SQL 的 CLOB 或者 TEXT 类型。

serializable

把可序列化的 Java 类型映射到对应的 SQL 二进制类型。你也可以为一个并非默认为基本类型或者实现 PersistentEnum 接口的可序列化 Java 类或者接口指定 Hibernate 类型 serializable。

clob, blob

JDBC 类 java.sql.Clob 和 java.sql.Blob 的映射。某些程序可能不适合使用这个类型，因为 blob 和 clob 对象可能在一个事务之外是无法重用的。(而且, 驱动程序对这种类型的支持充满着补丁和前后矛盾。)

实体及其集合的唯一标识可以是任何基础类型，除了 binary、 blob 和 clob 之外。（联合标识也是允许的，后面会说到。）

在 net.sf.hibernate.Hibernate 中，定义了基础类型对应的 Type 常量。比如，Hibernate.STRING 代表 string 类型。

5.2.3. 持久化枚举（Persistent enum）类型

*枚举（enumerated）*类型是一种常见的 Java 习惯用语，它是一个类，拥有一些（不多）的不可变实例。你可以为枚举类型实现 net.sf.hibernate.PersistentEnum 接口，定义 toInt() 和 fromInt()方法：

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;
            case 1: return GINGER;
            case 2: return BLACK;
            default: throw new RuntimeException("Unknown color
code");
        }
    }
}
```

Hibernate 可以使用枚举类的名字作为类型名，这个例子中就是 eg.Color。

5.2.4. 自定义值类型

开发者创建属于他们自己的值类型也是很容易的。比如说，你可能希望持久化 java.lang.BigInteger 类型的属性，持久化成为 VARCHAR 字段。Hibernate 没有内置这样一种类型。自定义类型能够映射一个属性（或集合元素）到不止一个数据库表字段。比如说，你可能有这样的 Java 属性：getName()/setName()，这是 java.lang.String 类型的，对应的持久化到三个字段：FIRST_NAME， INITIAL， SURNAME。

要实现一个自定义类型，可以实现 net.sf.hibernate.UserType 或 net.sf.hibernate.CompositeUserType 中的任一个，并且使用类型的 Java 全限定类名来声明属性。请查看 net.sf.hibernate.test.DoubleStringType 这个例子，看看它是怎么做的。

```
<property name="twoStrings"
type="net.sf.hibernate.test.DoubleStringType">
    <column name="first_string"/>
```

```
    <column name="second_string"/>
</property>
```

注意使用<column>标签来把一个属性映射到多个字段的做法。

虽然 Hibernate 内置的丰富类型和对 component 的支持意味着你可能很少*需要*使用自定义类型，至少对于你程序中经常出现的自定义类（并非实体）来说，这是一种好方法。比如说，MonetoryAmount(价格总额)对比使用 CompositeUserType 来说更好,虽然它可以很容易的使用一个 component 实现。这样做的动机之一是抽象。通过自定义类型，以后假若你改变表示金额值的方法时，你的映射文件不需要更改，这就得到了保护。

5.2.5. 映射到"任意"(any)类型

这是属性映射的又一种类型。<any>映射元素定义了一种从多个表到类的多形联合。这种类型的映射总是需要多于一个字段。第一个字段持有被从属的实体的类型。其他的字段持有标识符。对于这种类型的联合来说，不可能指定一个外键约束，所以当然这不是（多形）联合映射的通常方式。你只应该在非常特殊的情况下使用它（比如，审计 log,用户会话数据等等）。

```
<any name="anyEntity" id-type="long" meta-
type="eg.custom.Class2TablenameType">
    <column name="table_name"/>
    <column name="id"/>
</any>
```

meta-type 属性让应用程序指定一个自定义类型，把数据库字段值映射到一个持久化类，该类的标识属性是用 id-type 定义的。如果 meta-type 返回 java.lang.Class 的实例，不需要其他处理。另一方面，如果是类似 string 或者 character 这样的基本类型，你必须指定从值到类的映射。

```
<any name="anyEntity" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"                   ❶
        id-type="idtypename"                  ❷
        meta-type="metatypename"              ❸
        cascade="none|all|save-update"        ❹
        access="field|property|ClassName"          ❺
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

❶ name: 属性名。

❷ id-type: 标识符类型。

❸ meta-type (可选 - 默认为 class): 一个用于把 java.lang.Class 映射到一个数据库字段的类或者允许分辨映射的类型。

❹ cascade(级联) (可选- 默认为 none): 级联风格。

❺ access (可选 - 默认是 property): Hibernate 用来访问属性的策略。

老式的 object 类型是用来在 Hibernate 1.2 中起到类似作用的，他仍然被支持，但是已经基本废弃了。

5.3. SQL 中引号包围的标识符

你可强制 Hibernate 在生成的 SQL 中把标识符用引号前后包围起来，这需要在映射文档中使用反向引号(`)把表名或者字段名包围（可能比较拗口，请看下面的例子）。Hibernate 会使用相应的 SQLDialect（方言）来使用正确的引号风格（通常是双引号，但是在 SQL Server 中是括号，MySQL 中是反向引号）。

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator
class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.4. 自定义 DDL

Hibernate 映射文档也包含一些只为了 SchemaExport 命令行工具生成 DDL 使用的信息。比如，你可以使用<column>元素的 sql-type 属性覆盖字段类型。

```
<property
    name="amount"
    type="big_decimal">
    <column
        name="AMOUNT"
        sql-type="NUMERIC(11, 2)"/>
</property>
```

或者,你可以指定字段长度和约束。下面是等价的：

```
<property
    name="socialSecurityNumber"
    type="string"
    length="9"
    column="SSN"
    not-null="true"
    unique="true"/>

<property
    name="socialSecurityNumber"
    type="string">
    <column
        name="SSN"
        length="9"
        not-null="true"
```

```
        unique="true"/>
</property>
```

最后,也可以通过 SchemaExport 命令行工具自动生成所需的索引。如果要这么做，请在 <column>元素上使用 index 属性。

```
<property
    name="lastname">
    <column
        name="LASTNAME"
        index="by_last_first"/>
</property>

<property
    name="firstname">
     <column
         name="FIRSTNAME"
         index="by_last_first"/>
</property>
```

关于 SchemaExport 的更多内容，请参见本文档的“工具箱”那一章。

5.5. 映射文件的模块化（Modular mapping files）

允许在独立的映射文档中定义 subclass 和 joined-subclass，直接位于 hibernate-mapping 下。这就可以让你每次扩展你的类层次的时候，加入新的映射文件就行了。在子类的映射中你必须指定一个 extents 属性，指明先前已经映射过的超类。使用这个功能的时候，一定要注意映射文件的排序是非常重要的！

```

<hibernate-mapping>
        <subclass name="eg.subclass.DomesticCat" extends="eg.Cat"
discriminator-value="D">
             <property name="name" type="string"/>
        </subclass>
</hibernate-mapping>
```

Chapter 6. 集合类(Collections)

6.1. 持久化集合类(Persistent Collections)

(译者注：在阅读本章的时候，以后整个手册的阅读过程中，我们都会面临一个名词方面的问题，那就是“集合”。"Collections"和"Set"在中文里对应都被翻译为“集合”，但是他们的含义很不一样。Collections 是一个超集，Set 是其中的一种。大部分情况下，本译稿中泛指的未加英文注明的“集合”，都应当理解为“Collections”。在有些二者同时出现，可能造成混淆的地方，我们用“集合类”来特指“Collecions”，“集合(Set)”来指"Set"，一般都会在后面的括号中给出英文。希望大家在阅读时联系上下文理解，不要造成误解。 与此同时，“元素”一词对应的英文“element”，也有两个不同的含义。其一为集合的元素，是内存中的一个变量；另一含义则是 XML 文档中的一个标签所代表的元素。也请注意区别。 本章中,特别是后半部分是需要反复阅读才能理解清楚的。如果遇到任何疑问,请记住,英文版本的 reference 是惟一标准的参考资料。)

这部分不包含大量的 Java 代码例子。我们假定你已经了解如何使用 Java 自身的集合类框架(Java's collections framework)。 其实如果是这样, 这里就真的没有什么东西需要学习了... 用一句话来做个总结,你就用你已经掌握的知识来使用它们吧，不用为了适应 Hibernate 而作出改变.

Hibernate 可以持久化以下 java 集合的实例, 包括 java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List, 和任何持久实体或值的数组。类型为 java.util.Collection 或者 java.util.List 的属性还可以使用"bag"语义来持久。

警告：用于持久化的集合，除了集合接口外，不能保留任何实现这些接口的类所附加的语义(例如:LinkedHashSet 带来的迭代顺序)。所有的持久化集合，实际上都各自按照 HashMap, HashSet, TreeMap, TreeSet 和 ArrayList 的语义直接工作。更深入地说，对于一个包含集合的属性来说，必须把 Java 类型定义为接口（也就是 Map, Set 或者 List 等），而绝不能是 HashMap, TreeSet 或者 ArrayList。存在这个限制的原因是，在你不知道的时候，Hibernate 暗中把你的 Map, Set 和 List 的实例替换成了它自己的关于 Map, Set 或者 List 的实现。（所以在你的程序中,谨慎使用==操作符。）(译者说明: 为了提高性能等方面的原因,在 Hibernate 中实现了几乎所有的 Java 集合的接口 。)

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.save(cat);
kittens = cat.getKittens(); //Okay, kittens collection is a Set
(HashSet) cat.getKittens(); //Error!
```

集合遵从对值类型的通常规则：不能共享引用, 与其包含的实体共存亡。由于存在底层的关联模型，集合不支持空值语义；并且 hibernate 不会区分一个 null 的集合引用和一个不存在元素的空集合。

集合实例在数据库中根据指向对应实体的外键而得到区别。这个外键被称为*集合的关键字*。在 Hibernate 配置文件中使用<key> 元素来映射这个集合的关键字。

集合可以包含几乎所有的 Hibernate 类型, 包括所有的基本类型, 自定义类型,实体类型和组件。集合不能包含其他集合。这些被包含的元素的类型被称为*集合元素类型*。集合的元素在 Hibernate 中被映射为<element>, <composite-element>, <one-to-many>, <many-to-many> 或者 <many-to-any>。

除了 Set 和 Bag 之外的所有集合类型都有一个*索引(index)*字段,这个字段映射到一个数组或者 List 的索引或者 Map 的 key。Map 的索引的类型可以是任何基本类型, 实体类型或者甚至是一个组合类型(但不能是一个集合类型)。数组和 list 的索引肯定是整型,integer。在 Hibernate 配置文件中使用 <index>, <index-many-to-many>, <composite-index> 或者 <index-many-to-any>等元素来映射索引。

集合类可以产生相当多种类的映射，涵盖了很多通常的关系模型。我们建议你练习使用 schema 生成工具, 以便对如何把不同的映射定义转换为数据库表有一个感性认识。

6.2. 映射集合（Mapping a Collection）

在 Hibernate 配置文件中使用<set>，<list>，<map>，<bag>，<array> 和 <primitive-array>等元素来定义集合,而<map>是最典型的一个。

```
<map
    name="propertyName"                                              ❶
    table="table_name"                                               ❷
    schema="schema_name"                                             ❸
    lazy="true|false"                                                ❹
    inverse="true|false"                                             ❺
    cascade="all|none|save-update|delete|all-delete-orphan"          ❻
    sort="unsorted|natural|comparatorClass"                          ❼
    order-by="column_name asc|desc"                                  ❽
    where="arbitrary sql where condition"                            ❾
    outer-join="true|false|auto"                                     ❿
    batch-size="N"                                                   (11)
    access="field|property|ClassName"                                (12)
>

    <key .... />
    <index .... />
    <element .... />
</map>
```

❶ name 集合属性的名称

❷ table （可选——默认为属性的名称）这个集合表的名称(不能在一对多的关联关系中使用)

❸ schema（可选）表的 schema 的名称，他将覆盖在根元素中定义的 schema

❹ lazy（可选——默认为 false）lazy(可选--默认为 false）允许延迟加载（lazy initialization ）(不能在数组中使用)

❺ inverse（可选——默认为 false）标记这个集合作为双向关联关系中的方向一端。

❻ cascade（可选——默认为 none）让操作级联到子实体

❼ sort(可选)指定集合的排序顺序，其可以为自然的(natural)或者给定一个用来比较的类。

❽ order-by（可选，仅用于 jdk1.4）指定表的字段(一个或几个)再加上 asc 或者 desc(可选)，定义 Map,Set 和 Bag 的迭代顺序

❾ where（可选）指定任意的 SQL where 条件，该条件将在重新载入或者删除这个集合时使用(当集合中的数据仅仅是所有可用数据的一个子集时这个条件非常有用)

❿ outer-join(可选)指定这个集合,只要可能,应该通过外连接(outer join)取得。在每一个 SQL 语句中，只能有一个集合可以被通过外连接抓取(译者注：这里提到的 SQL 语句是取得集合所属类的数据的 Select 语句)

(11) batch-size（可选，默认为 1）指定通过延迟加载取得集合实例的批处理块大小（"batch size"）。

(12) access(可选-默认为属性 property):Hibernate 取得属性值时使用的策略

建立列表(List)和数组(Array)需要一个单独表字段用来保存列表(List)或数组(Array)的索引(foo[i]中的 i)。如果你的关系模型中没有一个索引字段，例如:如果你处理的是老式的遗留数据，你可以用无序的 Set 来替代。这会让那些以为 List 应该是访问无序集合的比较方便的

方法的人感到气馁。Hibernate 集合严格遵守 Set,List 和 Map 接口中包涵的自然语义。 List 元素不能正确的自发对他们自己进行排序!

在另一方面, 那些准备使用 List 来模拟 *bag* 的语义的人有一个合法的委屈(a legitimate grievance)。bag 是一个无序,没有索引的集合并且可能包含多个相同的元素。在 Java 集合框架中没有 Bag 接口(虽然你可以用 List 模拟它)。Hibernate 允许你映射类型为 List 或者 Collection 的属性到<bag>元素。注意: Bag 语义事实上并不是 Collection 规范(contract)的一部分并且事实上它和 List 规范中的语义是相矛盾的。

具有 inverse="false"标记的大型 Hibernate bag 效率是相当低的，应该尽量避免。Hibernate 无法创建,删除和更新它的单个记录, 因为他们没有关键字来识别单个记录。

6.3. 值集合和多对多关联(Collections of Values and Many To Many Associations)

任何值集合和实体集合如果被映射为多对多关联(Java 集合中的语义)就需要一个集合表。这个表中包含外键字段,元素字段还可能有索引字段。

使用<key>元素来申明从集合表到其拥有者类的表(from the collection table to the table of the owning class)的外键关键字。

```
<key column="column_name"/>
```

❶ column(必需):外键字段的名称

对于类似与 map 和 list 的带索引的集合, 我们需要一个<index>元素。对于 list 来说, 这个字段包含从零开始的连续整数。对于 map 来说,这个字段可以包含任意 Hibernate 类型的值。

```
<index
        column="column_name"                    ❶
        type="typename"                         ❷
/>
```

❶ column(必需):保存集合索引值的字段名。

❷ type (可选,默认为整型 integer):集合索引的类型。

还有另外一个选择,map 可以是实体类型的对象。在这里我们使用<index-many-to-many>元素。

```
<index-many-to-many
        column="column_name"                    ❶
        class="ClassName"                       ❷
/>
```

❶ column(必需):集合索引值中外键字段的名称

❷ class (required):(必需):集合的索引使用的实体类。

对于一个值集合, 我们使用<element>标签。

```
<element
```

```
        column="column_name"                    ❶
        type="typename"                         ❷
/>
```

❶ column(必需):保存集合元素值的字段名。

❷ type (必需):集合元素的类型

一个拥有自己表的实体集合对应于*多对多(many-to-many)关联*关系概念。多对多关联是针对 Java 集合的最自然映射关联关系，但通常并不是最好的关系模型。

```
<many-to-many
        column="column_name"                              ❶
        class="ClassName"                                 ❷
        outer-join="true|false|auto"                      ❸
/>
```

❶ column(必需): 这个元素的外键关键字段名

❷ class (必需): 关联类的名称

❸ outer-join (可选 - 默认为 auto): 在 Hibernate 系统参数中 hibernate.use_outer_join 被打开的情况下,该参数用来允许使用 outer join 来载入此集合的数据。

例子:

首先, 一组字符串:

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

包含一组整数的 bag(还设置了 order-by 参数指定了迭代的顺序):

```
<bag name="sizes" table="SIZES" order-by="SIZE ASC">
    <key column="OWNER"/>
    <element column="SIZE" type="integer"/>
</bag>
```

一个实体数组,在这个案例中是一个多对多的关联(注意这里的实体是自动管理生命周期的对象(lifecycle objects),cascade="all"):

```
<array name="foos" table="BAR_FOOS" cascade="all">
    <key column="BAR_ID"/>
    <index column="I"/>
    <many-to-many column="FOO_ID" class="com.illflow.Foo"/>
</array>
```

一个 map,通过字符串的索引来指明日期:

```
<map name="holidays" table="holidays" schema="dbo" order-by="hol_name
asc">
    <key column="id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

一个组件的列表：

```
<list name="carComponents" table="car_components">
    <key column="car_id"/>
    <index column="posn"/>
    <composite-element class="com.illflow.CarComponent">
            <property name="price" type="float"/>
            <property name="type" type="com.illflow.ComponentType"/>
            <property name="serialNumber" column="serial_no"
type="string"/>
    </composite-element>
</list>
```

6.4. 一对多关联（One To Many Associations）

*一对多关联直接*连接两个类对应的表,而没有中间集合表。(这实现了一个 *一对多*的关系模型)(译者注:这有别与多对多的关联需要一张中间表)。 这个关系模型失去了一些 Java 集合的语义:

- map,set 或 list 中不能包含 null 值
- 一个被包含的实体的实例只能被包含在一个集合的实例中
- 一个被包含的实体的实例只能对应于集合索引的一个值中

一个从 Foo 到 Bar 的关联需要额外的关键字字段,可能还有一个索引字段指向这个被包含的实体类,Bar 所对应的表。这些字段在映射时使用前面提到的<key>和<index>元素。

<one-to-many>标记指明了一个一对多的关联。

```
<one-to-many class="ClassName"/>
```

❶ class(必须):被关联类的名称。

例子

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="com.illflow.Bar"/>
</set>
```

注意:<one-to-many>元素不需要定义任何字段。 也不需要指定表名。

重要提示:如果一对多关联中的<key>字段定义成 NOT NULL,那么当创建和更新关联关系时 Hibernate 可能引起约束违例。为了预防这个问题,*你必须使用双向关联*, 并且在“多”这一端（Set 或者是 bag)指明 inverse="true"。

6.5. 延迟初始化(延迟加载)（Lazy Initialization）

(译者注：本翻译稿中，对 Lazy Initiazation 和 Eager fetch 中的 lazy,eager 采取意译的方式，分别翻译为延迟初始化和预先抓取。lazt initiazation 就是指直到第一次调用时才加载。)

集合(不包括数组)是可以延迟初始化的,意思是仅仅当应用程序需要访问时，才载入他们的值。对于使用者来说，初始化是透明的，因此应用程序通常不需要关心这个(事实上,透明的延迟加载也就是为什么 Hibernate 需要自己的集合实现的主要原因)。但是，如何应用程序试图执行以下程序：

```
s = sessions.openSession();
User u = (User) s.find("from User u where u.name=?", userName,
Hibernate.STRING).get(0);
Map permissions = u.getPermissions();
s.connection().commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  //
Error!
```

这个错误可能令你感到意外。因为在这个 Session 被提交(commit)之前，permissions 没有被初始化,那么这个集合将永远不能载入他的数据了。 解决方法是把读取集合数据的语句提到 Session 被提交之前。

另外一种选择是不使用延迟初始化集合。既然延迟初始化可能引起上面这样错误,默认是不使用延迟初始化的。但是，为了效率的原因，我们希望对绝大多数集合(特别是实体集合)使用延迟初始化。

延迟初始化集合时发生的例外被封装在 LazyInitializationException 中。

使用可选的 lazy 属性来定义延迟初始化集合：

```
<set name="names" table="NAMES" lazy="true">
    <key column="group_id"/>
    <element column="NAME" type="string"/>
</set>
```

在一些应用程序的体系结构中,特别是使用 hibernate 访问数据的结构，代码可能会用在不用的应用层中，可能没有办法保证当一个集合在初始化的时候，session 仍然打开着。 这里有两个基本方法来解决这个问题：

- 在基于 Web 的应用程序中，一个 servlet 过滤器可以用来在用户请求的完成之前来关闭 Session。当然,这个地方(关闭 session)严重依赖于你的应用程序结构中例外处理的正确性。在请求返回给用户之前关闭 Session 和结束事务是非常重要的,即使是在构建视图(译者注：返回给用户的 HTML 页面)的时候发生了例外,也必须确保这一点。考虑到这一点，servlet 过滤器可以保证能够操作这个 Session。我们推荐使用一个 ThreadLocal 变量来保存当前的 Session。
- 在一个有单独的商业层的应用程序中，商业逻辑必须在返回之前“准备好”Web 层所需要的所有集合。通常，应用程序为每个 Web 层需要的集合调用 Hibernate.initialize()(必须在 Session 被关闭之前调用)或者通过使用 FETCH 子句来明确获取到整个集合。

你可以使用 Hibernate Session API 中的 `filter()` 方法来在初始化之前得到集合的大小：

```
( (Integer) s.filter( collection, "select
count(*)" ).get(0) ).intValue()
```

`filter()` 或者 `createFilter()`同样被用于有效的重新载入一个集合的子集而不需要载入整个集合。

6.6. 集合排序（Sorted Collections）

Hibernate 支持实现 `java.util.SortedMap` 和 `java.util.SortedSet` 的集合。 你必须在映射文件中指定一个比较器：

```
<set name="aliases" table="person_aliases" sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

`sort` 属性中允许的值包括 `unsorted`,`natural` 和某个实现了 `java.util.Comparator` 的类的名称。

分类集合的行为事实上象 `java.util.TreeSet` 或者 `java.util.TreeMap`。

6.7. 对 collection 排序的其他方法（Other Ways To Sort a Collection）

如果你希望数据库自己对集合元素排序，可以利用 `set`,`bag` 或者 `map` 映射中的 `order-by` 属性。这个解决方案只能在 jdk1.4 或者更高的 jdk 版本中才可以实现(通过 LinkedHashSet 或者 LinkedHashMap 实现)。 它是在 SQL 查询中完成排序，而不是在内存中。

```
<set name="aliases" table="person_aliases" order-by="name asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

注意: 这个 `order-by` 属性的值是一个 SQL 排序子句而不是 HQL 的!

关联还可以在运行时使用 `filter()`根据任意的条件来排序。

```
sortedUsers = s.filter( group.getUsers(), "order by this.name" );
```

6.8. 垃圾收集（Garbage Collection）

集合是在被持久对象引用时自动持久化，并且不再不引用时自动删除的。 如果集合被从一个持久化对象转移到另外一个，他的数据可能会被从一个表移到另外一个。 你不需要担心这些。就跟你通常使用 java 集合一样使用 Hibernate 集合即可。

*注意:*上面的论断在 inverse="true"的情况下*不*适用。 我们将在接下来的章节中解释这一点。

6.9. 双向关联（Bidirectional Associations）

*双向关联*允许通过关联的任一端访问另外一端。在 Hibernate 中，支持两种类型的双向关联：

一对多（one-to-many）

Set 或者 bag 值在一端，单独值(非集合)在另外一端

多对多（many-to-many）

两端都是 set 或 bag 值

请注意 Hibernate 不支持带有索引的集合(list,map 或者 array)作为"多"的那一端的双向 one-to-many 关联。

要建立一个双向的多对多关联，只需要映射两个 many-to-many 关联到同一个数据库表中，并再定义其中的一端为 *inverse*。这里有一个从一个类关联到*他自身*的 many-to-many 的双向关联的例子：（原文：You may specify a bidirectional many-to-many association simply by mapping two many-to-many associations to the same database table and declaring one end as *inverse*. Heres an example of a bidirectional many-to-many association from a class back to *itself*:）

```
<class name="eg.Node">
    <id name="id" column="id"/>
    ....
    <bag name="accessibleTo" table="node_access" lazy="true">
        <key column="to_node_id"/>
        <many-to-many class="eg.Node" column="from_node_id"/>
    </bag>
     <!-- inverse end -->
    <bag name="accessibleFrom" table="node_access" inverse="true"
lazy="true">
        <key column="from_node_id"/>
        <many-to-many class="eg.Node" column="to_node_id"/>
    </bag>
</class>
```

如果只对关联的反向端进行了改变，这个改变*不会*被持久化。 （原文：Changes made only to the inverse end of the association are *not* persisted.）

要建立一个一对多的双向关联，你可以通过把一个一对多关联，作为一个多对一关联映射到到同一张表的字段上，并且在"多"的那一端定义 inverse="true"。（原文： You may map a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end inverse="true".）

```
<class name="eg.Parent">
    <id name="id" column="id"/>
    ....
    <set name="children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="eg.Child"/>
    </set>
</class>

<class name="eg.Child">
    <id name="id" column="id"/>
    ....
    <many-to-one name="parent" class="eg.Parent" column="parent_id"/>
</class>
```

在"一"这一端定义 inverse="true"不会影响级联操作。（原文：Mapping one end of an association with inverse="true" doesn't affect the operation of cascades.）

6.10. 三重关联（Ternary Associations）

这里有两种可能的途径来映射一个三重关联。其中一个是使用组合元素(下面将讨论).另外一个是使用一个 map，并且带有关联作为其索引。

```
<map name="contracts" lazy="true">
    <key column="employer_id"/>
    <index-many-to-many column="employee_id" class="Employee"/>
    <one-to-many column="contract_id" class="Contract"/>
</map>
<map name="connections" lazy="true">
    <key column="node1_id"/>
    <index-many-to-many column="node2_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

6.11. 异类关联(Heterogeneous Associations)

<many-to-any>和<index-many-to-any>元素提供真正的异类关联。这些元素和<any>元素工作方式是同样的,他们都应该很少用到。

6.12. 集合例子（Collection Example）

在前面的几个章节的确非常令人迷惑。 因此让我们来看一个例子。这个类：

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
```

```
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children)
{ this.children=children; }

    ....
    ....
}
```

这个类有一个 `eg.Child` 的实例集合。如果每一个子实例至多有一个父实例，那么最自然的映射是一个 one-to-many 的关联关系：

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

在以下的表定义中反应了这个映射关系：

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name
varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references
parent
```

如果父亲是*必须*的，那么就可以使用双向 one-to-many 的关联了(请看后面父/子关系的章节)。

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
```

```
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="eg.Parent"
column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

请注意 NOT NULL 的约束:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references
parent
```

另外一方面,如果一个子实例可能有多个父实例, 那么就应该使用 many-to-many 关联:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="eg.Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

表定义:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name
varchar(255) )
create table childset ( parent_id bigint not null, child_id bigint
not null, primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id)
references parent
```

```
alter table childset add constraint childsetfk1 (child_id) references
child
```

6.13. <idbag>

如果你完全信奉我们对于“联合主键（composite keys）是个坏东西”，和“实体应该使用（无机的）自己生成的代用标识符（surrogate keys）”的观点，也许你会感到有一些奇怪，我们目前为止展示的多对多关联和值集合都是映射成为带有联合主键的表的！现在，这一点非常值得争辩；看上去一个单纯的关联表并不能从代用标识符中获得什么好处（虽然使用组合值的集合*可能*会获得一点好处）。不过，Hibernate 提供了一个（一点点试验性质的）功能，让你把多对多关联和值集合应得到一个使用代用标识符的表去。

`<idbag>` 属性让你使用 bag 语义来映射一个 `List` (或 `Collection`)。

```
<idbag name="lovers" table="LOVERS" lazy="true">
    <collection-id column="ID" type="long">
        <generator class="hilo"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="eg.Person" outer-
join="true"/>
</idbag>
```

你可以理解，`<idbag>`人工的 id 生成器，就好像是实体类一样！集合的每一行都有一个不同的人造关键字。但是，Hibernate 没有提供任何机制来让你取得某个特定行的人造关键字。

注意`<idbag>`的更新性能要比普通的`<bag>`高得多！Hibernate 可以有效的定位到不同的行，分别进行更新或删除工作，就如同处理一个 list，map 或者 set 一样。

在目前的实现中，还不支持使用 `identity` 标识符生成器策略。(In the current implementation, the `identity` identifier generation strategy is not supported.)

Chapter 7. 组件（Components）

Component 这个概念在 Hibernate 中几处不同的地方为了不同的目的被重复使用.

7.1. 作为依赖的对象（As Dependent Objects）

Component 是一个被包含的对象，它和它的所有者存储在同一张表中。也就是，它是一个值类型，而不是一个实体。 Component 术语和组成的面向对象概念相关(而并不是系统构架层次上的组件的概念)。 举个例子，你可以对一个人 (Person)象以下这样来建模：

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
```

```
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

现在,姓名(Name)是作为人(Person)的一个组成部分。需要注意的是:需要对姓名 的持久化属性定义 getter 和 setter 方法,但是不需要实现任何的接口或申明标识符字段。

以下是这个例子的 XML 映射文件:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute
optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

人员(Person)表中将包括 pid, birthday, initial, first 和 last 等字段。

就像所有的值类型一样，Component 不支持共享引用。Component 的值为空从语义学上来讲是*专有的*。 每当 重新加载一个包含组件的对象,如果 component 的所有字段为空，那么将 Hibernate 将假定整个 component 为 空。对于绝大多数目的,这样假定是没有问题的。

Component 的属性可以是 Hibernate 类型（包括 Collections, many-to-one 关联， 以及其它 Component 等等）。嵌套 Component 不应该作为特殊的应用被考虑(Nested components should not be considered an exotic usage)。 Hibernate 趋向于支持设计细致(fine-grained)的对象模型。

<component> 元素还允许有 <parent>子元素 ，用来表明 component 类中的一个属性返回包含它的实体的引用。

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name">
        <parent name="namedPerson"/> <!-- reference back to the
Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

7.2. In Collections

Hibernate 支持 component 的集合(例如: 一个元素是“姓名”这种类型的数组)。 你可以使用<composite-element>标签替代<element>标签来定义你的 component 集合。

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- class attribute required
-->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

注意，如果你决定定义一个元素是联合元素的 Set，正确地实现 equals()和 hashCode()是非常重要的。

组合元素可以包含 component 但是不能包含集合。如果你的组合元素自身包含 component, 必须使用<nested-composite-element>标签。这是一个相当特殊的案例 - 组合元素的集合自身可以包含 component。 这个时候你就应该考虑一下使用 one-to-many 关联是否会更恰当。 尝试对这个组合元素重新建模为一个实体－但是需要注意的是，虽然 Java 模型和重新建模前是一样的，关系模型和持久性语义上仍然存在轻微的区别。

请注意如果你使用<set>标签,一个组合元素的映射不支持可能为空的属性. 当删除对象时, Hibernate 必须使用每一个字段的来确定一条记录(在组合元素表中，没有单个的关键字段), 如果有为 null 的字段，这样做就不可能了。你必须作出一个选择，要么在组合元素中使用不能为空的属性， 要么选择使用<list>, <map>,<bag> 或者 <idbag>而不是 <set>。

组合元素有个特别的案例，是组合元素可以包含一个<many-to-one> 元素。类似这样的映射允许你映射一个 many-to-mang 关联表作为组合元素额外的字段。(A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class.) 接下来的的例子是从 Order 到 Item 的一个多对多的关联关系,而 purchaseDate, price 和 quantity 是 Item 的关联属性。

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class
attribute is optional -->
        </composite-element>
    </set>
</class>
```

即使三重或多重管理都是可能的:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails" class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

在查询中，组合元素使用的语法是和关联到其他实体的语法一样的。

7.3. 作为一个 Map 的索引（As a Map Index ）

<composite-index>元素允许你映射一个 Component 类作为 Map 的 key， 但是你必须确定你正确的在这个类中重写了 hashCode() 和 equals()方法。

7.4. 作为联合标识符(As Composite Identifiers)

你可以使用一个 component 作为一个实体类的标识符。 你的 component 类必须满足以下要求:

- 它必须实现 java.io.Serializable 接口
- 它必须重新实现 equals()和 hashCode()方法, 始终和组合关键字在数据库中的概念保持一致

你不能使用一个 IdentifierGenerator 产生组合关键字。作为替代应用程序必 须分配它自己的标识符.

既然联合标识符必须在对象存储之前被分配，我们就不能使用 unsaved-value 来把刚刚新建的实例和在先前的 session 保存的实例区分开来。

如果你希望使用 saveOrUpdate()或者级联保存/更新(cascading save / update)，你应该实现 Interceptor.isUnsaved()。

使用<composite-id> 标签(它和<component> 标签有同样的属性和元素)代替<id>标签。 下面有个联合标识符类的定义:

```
<class name="eg.Foo" table"FOOS">
    <composite-id name="compId" class="eg.FooCompositeID">
        <key-property name="string"/>
        <key-property name="short"/>
        <key-property name="date" column="date_" type="date"/>
    </composite-id>
    <property name="name"/>
    ....
</class>
```

这时候,任何到 FOOS 的外键也同样是联合的， 在你其他类的映射文件中也必须同样定义。 一个到 Foo 的定义应该像以下这样:

```
<many-to-one name="foo" class="eg.Foo">
<!-- the "class" attribute is optional, as usual -->
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-one>
```

新的 <column> 标签同样被用于包含多个字段的自定义类型（This new columntag is also used by multi-column custom types）。 事实上在各个地方它都是一个可选的字段属性。要定义一个元素是 Foo 的集合类，要这样写:

```
<set name="foos">
    <key column="owner_id"/>
    <many-to-many class="eg.Foo">
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </many-to-many>
</set>
```

另一方面，<one-to-many>元素通常不定义字段.

如果 Foo 自己包含集合， 那么他们也需要使用联合外键。

```
<class name="eg.Foo">
    ....
    ....
    <set name="dates" lazy="true">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="foo_string"/>
            <column name="foo_short"/>
            <column name="foo_date"/>
        </key>
        <element column="foo_date" type="date"/>
    </set>
```

```
</class>
```

7.5. 动态组件 （Dynamic components）

你甚至可以映射 Map 类型的属性：

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO"/>
    <property name="bar" column="BAR"/>
    <many-to-one name="baz" class="eg.Baz" column="BAZ"/>
</dynamic-component>
```

从<dynamic-component>映射的语义上来讲，它和<component>是相同的。 这种映射类型的优点在于通过修改映射文件，就可以具有在部署时检测真实属性的能力.(利用一个 DOM 解析器， 是有可能在运行时刻操作映射文件的。)

Chapter 8. 操作持久化数据(Manipulating Persistent Data)

8.1. 创建一个持久化对象

对象（实体的实例）对一个特定的 Session 来说，要么是一个*瞬时（transient）*对象，要么是持久化（persistent）对象。刚刚创建的对象当然是瞬时的(注：后文中 transient object 也称为临时对象)。session 则提供了把瞬时实例保存（持久化）的服务:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

单参数的 save()方法为 fritz 生成了一个惟一标识符，并赋给这个对象。双参数的形式则使用给定的标识符保存 pk。我们一般不鼓励使用双参数的形式，因为这可能会（隐含）使主键赋予业务含义。它有用的时候是在一些特殊场合下，比如使用 Hibernate 来持久化一个 BMP 实体 bean.

关联的对象可以用你喜欢的任何顺序持久化，除非有外键字段具有 NOT NULL 的约束。决不会有外键约束冲突的危险。然而，如果在 save()对象的时候用错了顺序，会触犯 NOT NULL 约束。

8.2. 装载对象

如果你已知某个持久化实例的标识符，Session 的 load()方法让你取出它。第一种形式使用一个类对象作为参数，会把状态装载到另一个新创建的对象中去。第二个版本允许你给出一个实例，会在其中装载状态。把实例作为参数的形式在你准备把 Hibernate 和 BMP 实体 bean 一起

使用的时候特别有用，它就是为此设计的。你也可以发现其他的用途（比如自己实现实例池等等）。

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
// you need to wrap primitive identifiers
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new
Long(pkId) );
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

请注意如果没有匹配的数据库记录，`load()`方法可能抛出无法恢复的 exception。如果类是通过代理映射的，`load()`方法返回一个对象，这是一个未初始化的代理，并且直到你调用该对象的某方法时才会去访问数据库。这种行为方式在你喜欢创建一个指向某对象的关联，又不想真的从数据库中装载它的时候特别有用。

如果你不确定是否有匹配的行存在，你应该使用 `get()`方法，它会立刻访问数据库，如果没有对应的行，返回 null。

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

你可以用 SQL`SELECT ... FOR UPDATE` 装载对象。下一节有关于 Hibernate `LockMode` 的讨论。

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

注意，任何关联的实例或者包含的集合都*不会*被做为 `FOR UPDATE` 返回。

任何时候都可以使用 `refresh()`方法重新装载对象和它的集合。如果你使用数据库触发器更改了对象的某些属性，这就很有用。

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

8.3. Querying

如果你不能确定你要寻找的对象的标示符，请使用 `Session` 的 `find()`方法。Hibernate 使用一种简单而强大的面向对象查询语言。

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);
```

```
List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate " +
    "where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate.bithdate is
null" );

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
    izi,
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birthday is null"
);
```

find()的第二个参数接受一个对象或者对象数组。第三个参数接受一个 Hibernate 类型或者类型的数组。这些指定的类型用来把给定的对象绑定到查询中的?占位符（实际上对应的是 JDBC PreparedStatement 的传入参数）。就像在 JDBC 中一眼，你应该优先使用这种参数绑定的方式，而非组装字符串。

Hibernate 类定义了一些静态方法和常量，提供了访问大部分内置类型的手段。这些内置类型是 net.sf.hibernate.type.Type 的实例。

如果你知道你的查询会返回非常大量的对象，但是你不希望全部使用它们，你可以用 iterate()方法获得更好的性能，它会返回一个 java.util.Iterator。这个迭代器会在需要的时候装载对象，所使用的标识符来自一个前导的 SQL 查询。(一共是 N+1 次查询)

```
// fetch ids
Iterator iter = sess.iterate("from eg.Qux q order by q.likeliness");
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

很不幸，java.util.Iterator 没有声明任何 exception。所以，发生的任何 SQL 或者 Hibernate 的 exception 都会被包装在一个 LazyInitializationException 中（它是 RuntimeException 的子类）。

如果你预期大部分的对象已经装载过，存在于 session 的缓存中了，或者查询结果包含同样的对象很多次，那么 iterator()方法也会获得更好的性能。（如果没有任何数据被缓存或者重复出现，则 find()总是会更快。）下面是一个应该使用 iterator()调用的查询例子：

```
Iterator iter = sess.iterate(
    "select customer, product " +
    "from Customer customer, " +
    "Product product " +
    "join customer.purchases purchase " +
    "where product = purchase.product"
);
```

如果对上面的查询使用 find()，会返回一个非常大的 JDBCResultSet,包含很多重复的相同数据。

有时候 Hibernate 查询会每行返回多种对象，这种情况下，每行会返回一个数组，包含多个对象元素：

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = tuple[0]; Bar bar = tuple[1];
    ....
}
```

8.3.1. 标量查询（Scalar query）

查询可以在 select 子句中指定类的属性。甚至可以调用 SQL 的统计函数。属性或者统计值被称为“标量(scalar)”结果。

```
Iterator results = sess.iterate(
        "select cat.color, min(cat.birthdate), count(cat) from Cat
cat " +
        "group by cat.color"
);
while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
Iterator iter = sess.iterate(
    "select cat.type, cat.birthdate, cat.name from DomesticCat cat"
);
List list = sess.find(
    "select cat, cat.mate.name from DomesticCat cat"
);
```

8.3.2. 查询接口（Query interface）

如果你需要为你的结果集设置边界（你需要获取的最大行数与/或你希望获取的第一行），你应该得到一个 `net.sf.hibernate.Query` 的实例：

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

你甚至可以在映射文档中定义命名查询。（记得用一个 `CDATA` 块把你的查询包含起来，否则在分析的时候可能引起误解。）

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
Query q =
sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

查询界面支持使用命名参数。命名参数用`:name` 的形式在查询字符串中表示。在 `Query` 中有方法把实际参数绑定到命名参数或者 JDBC 风格的`?`参数。 *和 JDBC 不同，Hibernate 的参数从 0 开始计数。* 使用命名参数有一些好处：

- 命名参数不依赖于它们在查询字符串中出现的顺序
- 在同一个查询中可以使用多次
- 他们可读性好

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name
= :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name
= ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in
(:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

8.3.3. 可滚动迭代(Scrollable iteration)

如果你的 JDBC 驱动支持可滚动的 `ResuleSet`,`Query` 接口可以获取一个 `ScrollableResults`，允许你在查询结果中灵活游走。

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat
" +
                            "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of
cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() )
pageOfCats.add( cats.get(1) );

}
```

scroll()的行为方式与iterate()很类似，除了对象可以有选择的用get(int)初始化，而非整个行都一次性被初始化。

8.3.4. 过滤集合类(Filtering collections)

集合 *filter* 是一种特殊的查询，用于一个持久化集合或者数组。查询字符串可以引用 this,意为当前的数组元素。

```
Collection blackKittens = session.filter(
    pk.getKittens(), "where this.color = ?", Color.BLACK,
Hibernate.enum(Color.class)
);
```

返回的集合被认为是一个包(bag)。

请注意 filter 并不需要 from 子句（当然需要的话它们也可以加上）。Filter 不限定返回它们自己的集合元素。

```
Collection blackKittenMates = session.filter(
    pk.getKittens(), "select this.mate where this.color =
eg.Color.BLACK"
);
```

8.3.5. 条件查询

HQL 极为强大，但是有些人希望能够动态的使用一种面向对象 API 创建查询，而非在他们的 Java 代码中嵌入字符串。对于那部分人来说，Hibernate 提供了一种直观的 Criteria 查询 API。

```
Criteria crit = session.createCriteria(Cat.class);
```

```
crit.add( Expression.eq("color", eg.Color.BLACK) );
crit.setMaxResults(10);
List cats = crit.list();
```

如果你对类似于 SQL 的语法不是感觉很舒服的话，用这种方法开始使用 Hibernate 可能更容易。这种 API 也比 HQL 更可扩展。程序可以提供它们自己的 Criterion 接口的实现。

8.3.6. 使用本地 SQL 的查询

你可以使用 createSQLQuery()方法，用 SQL 来表达查询。你必须把 SQL 别名用大括号包围起来。

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT AS {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, {cat}.MATE
AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT AS {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

和 Hibernate 查询一样，SQL 查询也可以包含命名参数或者顺序参数。

8.4. 更改在当前 session 中保存或者装载的对象

持久化实例（就是通过 session 装载、保存、创建或者查询出的对象）可以被程序操作，所做的任何修改都会在 Session *清洗（flushed）*的时候被持久化（参见后面的“flushing”部分）。所以最直接的更改一个对象的方法就是 load()它，然后直接修改即可。

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and
persisted
```

有些时候这种编程模式显得效率不高，因为它需要在同一个 session 中先使用 SQL SELECT(来装载对象),又有一个 SQL UPDATE（来把修改的状态写回）。因此，Hibernate 提供了另一种方式。

8.5. 更改在以前 session 中保存或者装载的对象

很多程序需要在一个事务中获取对象，然后发送到界面层去操作，用一个新的事务来保存修改。（在高同步访问的环境中使用这种方式，经常使用附带版本的数据来保证事务独立性。）这种方法需要和上一节所描述的略微不同的编程模型。Hibernate 支持这种模型，因为它提供了 Session.update()方法。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
```

```
firstSession.save(potentialMate);

// in a higher tier of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

如果拥有 catId 标识符的 Cat 在试图 update 它之前已经被 secondSession 装载了，会抛出一个异常。

对于给定的临时实例，*当且仅当* 它们触及的其他临时实例需要保存的时候， 应用程序应该对它们分别各自使用 update()。（自动管理生命周期的对象(lifecycle object)除外。）

Hibernate 用户曾经要求有一个通用的方法，可以为新建的临时实例生成标识符并保存，或者保存已经存在标识符的临时实例的改动。saveOrUpdate()方法就是用来提供这个功能的。

Hibernate 通过对象的标识符的值（或 version,或 timestamp 时间戳)来分辨这是一个“新”（未保存过的）实例，还是一个“已存在”（已经保存或者从先前的 session 中装载的）的实例。id 映射中的 unsaved-value(或<version>,或 <timestamp>)用来指定哪个值被用于表示“新”实例。

```
<id name="id" type="long" column="uid" unsaved-value="null">
    <generator class="hilo"/>
</id>
```

unsaved-value 允许的取值包括：

- any - always save 永远保存
- none - always update 永远更新
- null - 当标识符是空的时候保存（默认情况）
- valid identifier value (合法的标识符值)- 当标识符是 null 或者这个给定的值时保存
- undefined - 对于 version 或 timestamp 来说的默认值。此时使用标识符检查。(原文: the default for version or timestamp, then identifier check is used.参见下文有进一步描述.)

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat has
a non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mate has
a null id)
```

saveOrUpdate()的用法和语义看来对初学者来说容易造成困惑。首先，如果你还没有试图在另一个新 session 中使用来自原 session 的实例，你根本就不需要使用 update()或者 saveOrUpdate()方法。有一些程序完全不需要使用这些方法。

通常，update()或 saveorUpdate()方法在下列情形下使用：

- 程序在前面的 session 中装载了对象
- 对象被传递到 UI（界面）层
- 对该对象进行了一些修改
- 对象被传递回业务层
- 应用程序在第二个 session 中调用 update()保存修改

saveOrUpdate()完成了如下工作：

- 如果对象已经在这个 session 中持久化过了，什么都不用做
- 如果对象没有标识值，调用 save()来保存它
- 如果对象的标识值与 unsaved-value 中的条件匹配，调用 save()来保存它
- 如果对象使用了版本(version 或 timestamp),那么除非设置 unsaved-value="undefined",版本检查会发生在标识符检查之前.
- 如果这个 session 中有另外一个对象具有同样的标识符，抛出一个异常

8.6. 把在先前的 session 中保存或装载的对象重新与新 session 建立关联(reassociate)

lock()方法是用来让应用程序把一个未修改的对象重新关联到新 session 的方法。

```
//just reassociate: 直接重新关联
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:  进行版本检查后关联
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
使用 SELECT ... FOR UPDATE 进行版本检查后关联
sess.lock(pk, LockMode.UPGRADE);
```

8.7. 删除持久化对象

使用 Session.delete()会把对象的状态从数据库中移除。当然，你的应用程序可能仍然持有一个指向它的引用。所以，最好这样理解：delete()的用途是把一个持久化实例变成临时实例。

```
sess.delete(cat);
```

你可以通过传递给 delete()一个 Hibernate 查询字符串来一次性删除很多对象。

你现在可以用你喜欢的任何顺序删除对象，不用担心外键约束冲突。当然，如果你搞错了顺序，还是有可能引发在外键字段定义的 NOT NULL 约束冲突。

8.8. 对象图（Graphs of objects）

要保存或者更新一个对象关联图中所有的所有对象，你必须做到：

- 保证每一个对象都执行 save()，saveOrUpdate() 或 update()方法，或者，
- 在定义关联对象的映射时，使用 cascade="all"或 cascade="save-update"。

类似的，要删除一个关系图中的所有对象，必须：

- 对每一个对象都执行 delete()，或者
- 在定义关联对象的映射时，使用 cascade="all",cascade="all-delete-orphan"或 cascade="delete"。

建议：

- 如果子对象的生命期是绑定到父对象的生命期的，通过指定 cascade="all"可以把它变成一个*自动管理生命周期的对象（lifecycle object）*。
- 否则，必须在应用程序代码中明确地执行 save()和 delete()。如果你想少敲一些代码，可以使用 cascade="sve-update"，然后只需明确地 delete()。

8.8.1. 自动管理生命周期的对象（lifecycle object）

对一种关联（多对一，或者集合）使用 cascade="all"映射，就把这种关联标记为一种父/子（parent/child）风格的关系，对父对象进行保存/更新/删除会导致对（所有）子对象的保存/更新/删除。但是这个比喻并不是特别确切。如果父对象解除了对某个子对象的关联，那这个子对象就*不*会被自动删除了。除非这是一个一对多的关联，并且标明了 cascade="all-delete-orphan"（所有-删除-孤儿）。级联操作的精确语义在下面列出：

- 如果父对象被保存，所有的子对象会被传递到 saveOrUpdate()方法去执行
- 如果父对象被传递到 update()或者 saveOrUpdate()，所有的子对象会被传递到 saveOrUpdate()方法去执行
- 如果一个临时的子对象被一个持久化的父对象引用了，它会被传递到 saveOrUpdate()去执行
- 如果父对象被删除了，所有的子对象对被传递到 delete()方法执行
- 如果临时的子对象不再被持久化的父对象引用，什么都不会发生（必要时，程序应该明确的删除这个子对象），除非声明了 cascade="all-delete-orphan"，在这种情况下，成为"孤儿"的子对象会被删除。

8.8.2. 通过可触及性决定持久化（Persistence by Reachability）

Hibernate 还没有完全实现"通过可触及性决定持久化"，后者暗示会对垃圾收集进行（效率不高的）持久化。但是，因为很广泛的呼声，Hibernate 实现了一种意见，如果一个实体被一个持久化的对象引用，它也会被持久化。注明了 cascade="save-update"的关联就是按照这种思路运作的。如果你希望在你的整个程序中都贯彻这个方法，你可以在<hibernate-mapping>元素的 default-cascade 属性中指定这种级联方式。

8.9. 清洗(Flushing) -- 这个词很难翻译，不能使用"刷新"，因为刷新一词已经被"refresh"使用了。有什么好的建议？

每件隔一段时间，Session 会执行一些必需的 SQL 语句来把内存中的对象和 JDBC 连接中的状态进行同步。这个过程被称为*清洗(flush)*，默认会在下面的时间点执行：

- 在某些 find()或者 iterate()调用的时候
- 在 net.sf.hibernate.Transaction.commit()的时候
- 在 Session.flush()的时候

涉及的 SQL 语句会按照下面的顺序安排：

- 所有对实体进行插入的语句，其顺序按照对象执行 `Session.save()`的时间顺序
- 所有对实体进行更新的语句
- 所有进行集合删除的语句
- 所有对集合元素进行删除，更新或者插入的语句
- 所有进行集合插入的语句
- 所有对实体进行删除的语句，其顺序按照对象执行 `Session.delete()`的时间顺序

（有一个例外时，如果对象使用 `native` 方式进行 ID 生成的话，它们一执行 save 就会被插入。）

除非你明确地发出了 `flush()`指令，关于 Session *合时*会执行这些 JDBC 调用是完全无法保证的，只能保证它们执行的前后顺序。当然，Hibernate 保证，`Session.find(..)`绝对不会返回已经失效的数据，也不会返回错误数据。

也可以改变默认的设置，来让清洗发生的不那么频繁。`FlushMode` 类定义了三种不同的方式。大部分情况下，它们只由当你在处理“只读”的事务时才会使用，可能会得到一些（不是那么明显的）性能提高。

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); //allow queries to return stale
state
Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);
// execute some queries....
sess.find("from Cat as cat left outer join cat.kittens kitten");
//change to izi is not flushed!!
....
tx.commit(); //flush occurs
```

8.10. 结束一个 Session

结束一个 session 包括四个独立的步骤：

- 清洗 session
- 提交事务
- 关闭 session
- 处理异常

8.10.1. 清洗(Flush)session

如果你正在使用 `TransactionAPI`，你就不用担心这个步骤。在事务提交的时候，隐含就会包括这一步。否则，你应该调用 `Session.flush()`来确保你所有的修改都与数据库同步。

8.10.2. 提交事务

如果你正在使用 Hibernate 的 `Transaction` API,代码类似这样：

```
tx.commit(); // flush the Session and commit the transaction
```

如果你自行管理 JDBC 事务，你应该手工对 JDBC 连接执行 commit()。

```
sess.flush();
sess.connection().commit();  // not necessary for JTA datasource
```

如果你决定*不*提交你的更改：

```
tx.rollback();  // rollback the transaction
```

或者:

```
// not necessary for JTA datasource, important otherwise
sess.connection().rollback();
```

如果你回滚了事务,你应该立即关闭和取消当前 session,确保 Hibernate 内部状态的完整性。

8.10.3. 关闭 session

调用 Session.close()就标志这个 session 进入了尾声。close()主要的含义就是与这个 session 相关的 JDBC 连接会被放弃。

```
tx.commit();
sess.close();
sess.flush();
sess.connection().commit();  // not necessary for JTA datasource
sess.close();
```

如果你自行管理连接，close()会返回连接的一个引用，你就可以手工把它关闭，或者返回它到连接池去。其他情况下，close()会把它返回到连接池去。

8.10.4. 处理异常

如果 Session 抛出了一个 exception(包括任何 SQLException),你应该立刻回滚这个事务，调用 Session.close)()来取消这个 Session 实例。Session 中的一些特定方式会确保 session *不*会处于一个不稳定不完整的状态。

建议采用下面的异常处理片断：

```
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    sess.close();
```

```
}
```

如果你是手工管理 JDBC 事务的，用下面这段:

```
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
    sess.connection().commit();
}
catch (Exception e) {
    sess.connection().rollback();
    throw e;
}
finally {
    sess.close();
}
```

如果你是从 JTA 中获得数据源的:

```
UserTransaction ut = .... ;
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
}
catch (Exception e) {
    ut.setRollbackOnly();
    throw e;
}
finally {
    sess.close();
}
```

8.11. 拦截器(Interceptors)

Interceptor 接口提供从 session 到你的应用程序的回调方法，让你的程序可以观察和在持久化对象保存/更改/删除或者装载的时候操作它的属性。一种可能的用途是用来监视统计信息。比如，下面的 Interceptor 会自动在一个 Auditable 创建的时候设置其 createTimestamp，并且当它被更改的时候，设置其 lastUpdateTimestamp 属性。

```
package net.sf.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import net.sf.hibernate.Interceptor;
import net.sf.hibernate.type.Type;

public class AuditInterceptor implements Interceptor, Serializable {

    private int updates;
    private int creates;
```

```
    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if
( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void postFlush(Iterator entities) {
        System.out.println("Creations: " + creates + ", Updates: " +
updates);
    }

    public void preFlush(Iterator entities) {
```

```
        updates=0;
        creates=0;
    }

    ......
    ......

}
```

当 session 被创建的时候，就应该指定拦截器。

```
Session session = sf.openSession( new AuditInterceptor() );
```

8.12. 元数据(Metadata) API

Hibernate 对所有的实体和值类型都需要一个非常丰富的元级别(meta-level)模型。有时候，这个模型对应用程序本身也会非常有用。比如说，应用程序可能使用 Hibernate 的元数据来实现一种“智能”的深度拷贝算法，来理解哪些对象应该被拷贝（比如，可变的值类型），那些不应该（不可变的值类型和可能的被关联的实体）。

Hibernate 通过 `ClassMetadata` 接口，`CollectionMetadata` 接口和 `Type` 对象树，暴露出元数据。可以通过 `SessionFactory` 获取 metadata 接口的实例。

```
Cat fritz = ......;
Long id = (Long) catMeta.getIdentifier(fritz);
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);
Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();
// get a Map of all properties which are not collections or
associations
// TODO: what about components?
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType()
&& !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Chapter 9. 父子关系(Parent Child Relationships)

刚刚接触 Hibernate 的人大多是从父子关系（parent / child type relationship）的建模入手的。父子关系的建模有两种方法。比较简便、直观的方法就是在实体类 `Parent` 和 `Child` 之间建立`<one-to-many>`的关联关系，从 `Parent` 指向 `Child`，对新手来说尤其如此。但还有另一种方法，就是将 `Child` 声明为一个`<composite-element>`（组合元素）。 可以看出在 Hibernate 中使用一对多关联比 composite element 更接近于通常 parent / child 关系的语义。下面我们会阐述如何使用*双向可级联的一对多关联(bidirectional one to many association with cascades)*去建立有效、优美的 parent / child 关系。这一点也不难！

9.1. 关于 collections

在 Hibernate 下，实体类将 collection 作为自己的一个逻辑单元，而不是被容纳的多个实体。这非常重要！它主要体现为以下几点：

- 当删除或增加 collection 中对象的时候，拥有这个 collection 的实体对象的版本值会递增。
- 如果一个从 collection 中移除的对象是一个值类型(value type)的实例，比如 composite element，那么这个对象的持久化状态将会终止，其在数据库中对应的记录会被删除。同样的，向 collection 增加一个 value type 的实例将会使之立即被持久化。
- 另一方面，如果从一对多或多对多关联的 collection 中移除一个实体，在缺省情况下这个对象并不会被删除。这个行为是完全合乎逻辑的－－改变一个实体的内部状态不应该使与它关联的实体消失掉！同样的，向 collection 增加一个实体不会使之被持久化。

实际上，向 Collection 增加一个实体的缺省动作只是在两个实体之间创建一个连接而已，同样移除的时候也只是删除连接。这种处理对于所有的情况都是合适的。不适合所有情况的其实是父子关系本身，因为子对象是否存在依赖于父对象的生存周期。

9.2. 双向的一对多关系(Bidirectional one to many)

让我们从一个简单的例子开始，假设要实现一个从类 Parent 到类 Child 的一对多关系。

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

如果我们运行下面的代码

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate 就会产生下面的两条 SQL 语句:

- 一条 INSERT 语句，用于创建对象 c 对应的数据库记录
- 一条 UPDATE 语句，用于创建从对象 p 到对象 c 的连接

这样做不仅效率低，而且违反了列 parent_id 非空的限制。

底层的原因是，对象 p 到对象 c 的连接（外键 parent_id）没有被当作是 Child 对象状态的一部分，也没有在 INSERT 的时候被创建。解决的办法是，在 Child 一端设置映射。

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

（我们还需要为类 Child 添加 parent 属性）

现在实体 Child 在管理连接的状态，为了使 collection 不更新连接，我们使用 inverse 属性。

```
<set name="children" inverse="true">
```

```
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

下面的代码是用来添加一个新的 Child

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

现在，只会有一条 INSERT 语句被执行！

为了让事情变得井井有条，可以为 Parent 加一个 addChild()方法。

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

现在，添加 Child 的代码就是这样

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

9.3. 级联（Cascades）

对每个对象调用 save()方法很麻烦，我们可以用级联来解决这个问题。

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

配置级联以后，代码就可以这样写：

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

同样的，保存或删除 Parent 对象的时候并不需要遍历其子对象。 下面的代码会删除对象 p 及其所有子对象对应的数据库记录。

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
```

```
session.flush();
```

然而，这段代码

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

不会从数据库删除 c；它只会删除与 p 之间的连接（并且会导致违反 NOT NULL 约束，在这个例子中）。你需要明确调用 Child 的 delete()方法。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

在我们的例子中，如果我们规定没有父对象的话，子对象就不应该存在，如果将子对象从 collection 中移除，实际上我们是想删除它。要实现这种要求，就必须使用 cascade="all-delete-orphan"。

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

注意：即使在 collection 一方的映射中指定 inverse="true"，在遍历 collection 的时候级联操作仍然会执行。如果你想要通过级联进行子对象的插入、删除、更新操作，就必须把它加到 collection 中，只调用 setParent()是不够的。

9.4. 级联更新（Using cascading update()）

假设我们从 Session 中装入了一个 Parent 对象，用户界面对其进行了修改，然后我们希望在一个新的 Session 里面调用 update()来更新它。对象 Parent 包含了子对象的集合，由于打开了级联更新，Hibernate 需要知道哪些子对象是新的，哪些是数据库中已经存在的。我们假设 Parent 和 Child 对象的标识属性的类型为 java.lang.Long。Hibernate 会使用标识属性的值来判断哪些子对象是新的。(你也可以使用 version 或 timestamp 属性，参见 Section 8.5, “更改在以前 session 中保存或者装载的对象”.)

unsaved-value 属性是用来表示新实例的标识属性值的，缺省为"null"，用在 Long 类型的标识类型再好不过了。如果我们使用原始类型作为标识类型的话，我们在配置 Child 类映射的时候就必须写：

```
<id name="id" type="long" unsaved-value="0">
```

(为版本和时间戳属性进行映射，也会有另一个叫做 unsaved-value 的属性。)

下面的代码会更新 parent 和 child 对象，并且插入 newChild 对象。

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

好的，对于自动生成标识的情况这样做很方便，但是自分配的标识和复合标识怎么办呢？这是有点麻烦，因为 unsaved-value 无法区分新对象（标识是用户指定的）和前一个 Session 装入的对象。在这种情况下，你可能需要给 Hibernate 一些提示，在调用 update(parent)之前：

- 在这个类的<version> or <timestamp>属性映射上定义 unsaved-value="null"或者 unsaved-value="negative"。
- 在对父对象执行 update(parent)之前，设定 unsaved-value="none"并且显式的调用 save()在数据库创建新子对象
- 在对父对象执行 update(parent)之前，设定 unsaved-value="any"并且显式的调用 update()更新已经装入的子对象

none 是自分配标识和复合标识的 unsaved-value 的缺省值。

There is one further possibility. There is a new Interceptor method named isUnsaved() which lets the application implement its own strategy for distinguishing newly instantiated objects. For example, you could define a base class for your persistent classes.

还有一种可能情况，有一个名为 isUnsaved()的拦截器（Interceptor）方法，它允许应用程序自己实现新实例的判断。比如，你可以自己定义一个持久类的祖先类：

```
public class Persistent implements Lifecycle {
    private boolean _saved = false;
    public boolean onSave(Session s) {
        _saved=true;
        return NO_VETO;
    }
    public void onLoad(Session s, Serializable id) {
        _saved=true;
    }
    ......
    public boolean isSaved() {
        return _saved;
    }
}
```

And implement isUnsaved()

（saved 属性是不会被持久化的。）现在在 onLoad()和 onSave()外，还要实现 isUnsaved()。

```
public Boolean isUnsaved(Object entity) {
    if (entity instanceof Persistent) {
        return new Boolean( !( (Persistent) entity ).isSaved() );
    }
    else {
        return null;
```

```
    }
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent)
entity ).onLoad();
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent)
entity ).onSave();
    return false;
}
```

9.5. 结论

这个问题往往让新手感到迷惑，它确实不太容易消化。不过，经过一些实践以后，你会感觉越来越顺手。父子对象模式已经被广泛的应用在 Hibernate 应用程序中。

在第一段中我们曾经提到另一个方案。复合元素的语义与父子关系是等同的，但是我们并没有详细讨论。很不幸复合元素还有两个重大限制:复合元素不能拥有 collections，并且，除了用于惟一的父对象外，它们不能再作为其它任何实体的子对象。（但是，通过使用`<idbag>`映射，它们*可能*拥有代理主键。）

Chapter 10. Hibernate 查询语言(Query Language)，即 HQL

Hibernate 装备了一种极为有力的查询语言，（有意地）看上去很像 SQL。但是别被语法蒙蔽，HQL 是完全面向对象的，具备继承、多态和关联等特性。

10.1. 大小写敏感性(Case Sensitivity)

除了 Java 类和属性名称外，查询都是大小写不敏感的。 所以， `SeLeCT` 和 `sELEct` 以及 `SELECT` 相同的，但是 `net.sf.hibernate.eg.FOO` 和 `net.sf.hibernate.eg.Foo` 是不同的， `foo.barSet` 和 `foo.BARSET` 也是不同的。

本手册使用小写的 HQL 关键词。有些用户认为在查询中使用大写的关键字更加易读，但是我们认为嵌入在 Java 代码中这样很难看。

10.2. from 子句

可能最简单的 Hibernate 查询是这样的形式：

```
from eg.Cat
```

它简单的返回所有 eg.Cat 类的实例。

大部分情况下，你需要赋予它一个*别名（alias）*，因为你在查询的其他地方也会引用这个 Cat。

```
from eg.Cat as cat
```

上面的语句为 Cat 赋予了一个别名 cat 。所以后面的查询可以用这个简单的别名了。as 关键字是可以省略的，我们也可以写成这样：

```
from eg.Cat cat
```

可以出现多个类，结果是它们的笛卡尔积，或者称为“交叉”连接。

```
from Formula, Parameter
from Formula as form, Parameter as param
```

让查询中的别名服从首字母小写的规则，我们认为这是一个好习惯。这和 Java 对局部变量的命名规范是一致的。(比如，domesticCat).

10.3. 联合（Associations）和连接（joins）

你可以使用 join 定义两个实体的连接，同时指明别名。

```
from eg.Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

from eg.Cat as cat left join cat.mate.kittens as kittens

from Formula form full join form.parameter param
```

支持的连接类型是从 ANSI SQL 借用的：

- 内连接，inner join
- 左外连接，left outer join
- 右外连接，right outer join
- 全连接，full join (不常使用)

inner join, left outer join 和 right outer join 都可以简写。

```
from eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

并且，加上 "fetch"后缀的抓取连接可以让联合的对象随着它们的父对象的初始化而初始化，只需要一个 select 语句。这在初始化一个集合的时候特别有用。

```
from eg.Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

抓取连接一般不需要赋予别名，因为被联合的对象应该不会在 where 子句（或者任何其它子句）中出现。并且，被联合的对象也不会在查询结果中直接出现。它们是通过父对象进行访问的。

请注意，目前的实现中，在一次查询中只会抓取一个集合（？原文为：only one collection role may be fetched in a query）。也请注意，在使用 scroll()或者 iterate()方式调用的查询中，是禁止使用 fetch 构造的。最后，请注意 full join fetch 和 right join fetch 是没有意义的。

10.4. select 子句

select 子句选择在结果集中返回哪些对象和属性。思考一下下面的例子：

```
select mate
from eg.Cat as cat
    inner join cat.mate as mate
```

这个查询会选择出作为其它猫（Cat）朋友（mate）的那些猫。当然，你可以更加直接的写成下面的形式：

```
select cat.mate from eg.Cat cat
```

你甚至可以选择集合元素，使用特殊的 elements 功能。下面的查询返回所有猫的小猫。

```
select elements(cat.kittens) from eg.Cat cat
```

查询可以返回任何值类型的属性，包括组件类型的属性：

```
select cat.name from eg.DomesticCat cat
where cat.name like 'fri%'

select cust.name.firstName from Customer as cust
```

查询可以用元素类型是 Object[]的一个数组返回多个对象和/或多个属性。

```
select mother, offspr, mate.name
from eg.DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

或者实际上是类型安全的 Java 对象

```
select new Family(mother, mate, offspr)
from eg.DomesticCat as mother
    join mother.mate as mate
```

```
    left join mother.kittens as offspr
```

上面的代码假定 `Family` 有一个合适的构造函数。

10.5. 统计函数(Aggregate functions)

查询可以返回属性的统计函数。

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from eg.Cat cat
```

在 `select` 子句中，统计函数的变量也可以是集合。

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

下面是支持的统计函数列表：

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

`distinct` 和 `all` 关键字的用法和语义与 SQL 相同。

```
select distinct cat.name from eg.Cat cat

select count(distinct cat.name), count(cat) from eg.Cat cat
```

10.6. 多态(polymorphism)

类似下面的查询：

```
from eg.Cat as cat
```

返回的实例不仅仅是 `Cat`，也有可能是子类的实例，比如 `DomesticCat`。Hibernate 查询可以在 `from` 子句中使用*任何* Java 类或者接口的名字。查询可能返回所有继承自这个类或者实现这个接口的持久化类的实例。下列查询会返回所有的持久化对象：

```
from java.lang.Object o
```

可能有多个持久化类都实现了 `Named` 接口：

```
from eg.Named n, eg.Named m where n.name = m.name
```

请注意，上面两个查询都使用了超过一个 SQL 的 `SELECT`。这意味着 `order by` 子句将不会正确排序。（这也意味着你不能对这些查询使用 `Query.scroll()`。）

10.7. where 子句

where 子句让你缩小你要返回的实例的列表范围。

```
from eg.Cat as cat where cat.name='Fritz'
```

返回所有名字为'Fritz'的 Cat 的实例。

```
select foo
from eg.Foo foo, eg.Bar bar
where foo.startDate = bar.date
```

会返回所有的满足下列条件的 Foo 实例，它们存在一个对应的 bar 实例，其 date 属性与 Foo 的 startDate 属性相等。复合路径表达式令 where 子句变得极为有力。思考下面的例子：

```
from eg.Cat cat where cat.mate.name is not null
```

这个查询会被翻译为带有一个表间（inner)join 的 SQL 查询。如果你写下类似这样的语句：

```
from eg.Foo foo
where foo.bar.baz.customer.address.city is not null
```

你最终会得到的查询，其对应的 SQL 需要 4 个表间连接。

=操作符不仅仅用于判断属性是否相等，也可以用于实例：

```
from eg.Cat cat, eg.Cat rival where cat.mate = rival.mate

select cat, mate
from eg.Cat cat, eg.Cat mate
where cat.mate = mate
```

特别的，小写的 id 可以用来表示一个对象的惟一标识。（你可以使用它的属性名。）

```
from eg.Cat as cat where cat.id = 123

from eg.Cat as cat where cat.mate.id = 69
```

第二个查询是很高效的。不需要进行表间连接！

组合的标示符也可以使用。假设 Person 有一个组合标示符，是由 country 和 medicareNumber 组合而成的。

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456

from bank.Account account
where account.owner.id.country = 'AU'
```

```
    and account.owner.id.medicareNumber = 123456
```

又一次，第二个查询不需要表间连接。

类似的，在存在多态持久化的情况下，特殊属性 class 用于获取某个实例的辨识值。在 where 子句中嵌入的 Java 类名将会转换为它的辨识值。

```
from eg.Cat cat where cat.class = eg.DomesticCat
```

你也可以指定组件（或者是组件的组件，依次类推）或者组合类型中的属性。但是在一个存在路径的表达式中，最后不能以一个组件类型的属性结尾。（这里不是指组件的属性）。比如，假若 store.owner 这个实体的的 address 是一个组件

```
store.owner.address.city     //okay
store.owner.address          //error!
```

“任意(any)”类型也有特殊的 id 属性和 class 属性，这可以让我们用下面的形式来表达连接（这里 AuditLog.item 是一个对应到<ant>的属性）。

```
from eg.AuditLog log, eg.Payment payment
where log.item.class = 'eg.Payment' and log.item.id = payment.id
```

注意上面查询中，log.item.class 和 payment.class 会指向两个值，代表完全不同的数据库字段。

10.8. 表达式(Expressions)

where 子句允许出现的表达式包括了你在 SQL 中可以使用的大多数情况：

- 数学操作+, -, *, /
- 真假比较操作 =, >=, <=, <>, !=, like
- 逻辑操作 and, or, not
- 字符串连接 ||
- SQL 标量（ scalar）函数，例如 upper() 和 lower()
- 没有前缀的 ( )表示分组
- in, between, is null
- JDBC 传入参数?
- 命名参数 :name, :start_date, :x1
- SQL 文字 'foo', 69, '1970-01-01 10:00:01.0'
- Java 的 public static final 常量 比如 Color.TABBY

in 和 between 可以如下例一样使用:

```
from eg.DomesticCat cat where cat.name between 'A' and 'B'

from eg.DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

其否定形式为

```
from eg.DomesticCat cat where cat.name not between 'A' and 'B'

from eg.DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

类似的，`is null` 和 `is not null` 可以用来测试 null 值。

通过在 Hibernate 配置中声明 HQL 查询的替换方式，Boolean 也是很容易在表达式中使用的：

```
<property name="hibernate.query.substitutions">true 1, false
0</property>
```

在从 HQL 翻译成 SQL 的时候,关键字 `true` 和 `false` 就会被替换成 `1` 和 `0`。

```
from eg.Cat cat where cat.alive = true
```

你可以用特殊属性 `size` 来测试一个集合的长度，或者用特殊的 `size()`函数也可以。

```
from eg.Cat cat where cat.kittens.size > 0

from eg.Cat cat where size(cat.kittens) > 0
```

对于排序集合，你可以用 `minIndex` 和 `maxIndex` 来获取其最大索引值和最小索引值。类似的，`minElement` 和 `maxElement` 可以用来获取集合中最小和最大的元素，前提是必须是基本类型的集合。

```
from Calendar cal where cal.holidays.maxElement > current date
```

也有函数的形式（和上面的形式不同，函数形式是大小写不敏感的）：

```
from Order order where maxindex(order.items) > 100

from Order order where minelement(order.items) > 10000
```

SQL 中的 `any`，`some`，`all`，`exists`，`in` 功能也是支持的，前提是必须把集合的元素或者索引集作为它们的参数（使用 `element` 和 `indices` 函数），或者使用子查询的结果作为参数。

```
select mother from eg.Cat as mother, eg.Cat as kit
where kit in elements(foo.kittens)

select p from eg.NameList list, eg.Person p
where p.name = some elements(list.names)

from eg.Cat cat where exists elements(cat.kittens)

from eg.Player p where 3 > all elements(p.scores)

from eg.Show show where 'fizard' in indices(show.acts)
```

请注意这些设施：`size`, `elements`, `indices`, `minIndex`, `maxIndex`, `minElement`, `maxElement` 都有一些使用限制：

- 在 where 子句中：只对支持子查询的数据库有效
- 在 select 子句中：只有 elements 和 indices 有效

有序的集合(数组、list、map)的元素可以用索引来进行引用（只限于在 where 子句中）

```
from Order order where order.items[0].id = 1234

select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar

select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and
order.id = 11

select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

[]中的表达式允许是另一个数学表达式。

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL 也对一对多关联或者值集合提供内置的 index()函数。

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

底层数据库支持的标量 SQL 函数也可以使用

```
from eg.DomesticCat cat where upper(cat.name) like 'FRI%'
```

假如以上的这些还没有让你信服的话，请想象一下下面的查询假若用 SQL 来写，会变得多么长，多么不可读：

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

*提示：*对应的 SQL 语句可能是这样的

```
SELECT cust.name, cust.address, cust.phone, cust.id,
cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
```

```
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

10.9. order by 子句

查询返回的列表可以按照任何返回的类或者组件的属性排序：

```
from eg.DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

asc 和 desc 是可选的，分别代表升序或者降序。

10.10. group by 子句

返回统计值的查询可以按照返回的类或者组件的任何属性排序：

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color

select foo.id, avg( elements(foo.names) ), max( indices(foo.names) )
from eg.Foo foo
group by foo.id
```

请注意：你可以在 select 子句中使用 elements 和 indices 指令，即使你的数据库不支持子查询也可以。

having 子句也是允许的。

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

在 having 子句中允许出现 SQL 函数和统计函数，当然这需要底层数据库支持才行。（比如说,MySQL 就不支持）

```
select cat
from eg.Cat cat
    join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
```

```
order by count(kitten) asc, sum(kitten.weight) desc
```

注意，group by 子句和 order by 子句都不支持数学表达式。

10.11. 子查询

对于支持子查询的数据库来说，Hibernate 支持在查询中嵌套子查询。子查询必须由圆括号包围（常常是在一个 SQL 统计函数中）。也允许关联子查询（在外部查询中作为一个别名出现的子查询）。

```
from eg.Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from eg.DomesticCat cat
)

from eg.DomesticCat as cat
where cat.name = some (
    select name.nickName from eg.Name as name
)

from eg.Cat as cat
where not exists (
    from eg.Cat as mate where mate.mate = cat
)

from eg.DomesticCat as cat
where cat.name not in (
    select name.nickName from eg.Name as name
)
```

10.12. 示例

Hibernate 查询可以非常强大复杂。实际上，强有力的查询语言是 Hibernate 的主要卖点之一。下面给出的示例与我在近期实际项目中使用的一些查询很类似。请注意你编写的查询大部分等都不会这么复杂！

下面的查询对特定的客户，根据给定的最小总计值（minAmount），查询出所有未付订单，返回其订单号、货品总数、订单总金额，结果按照总金额排序。在决定价格的时候，参考当前目录。产生的 SQL 查询，在 ORDER, ORDER_LINE, PRODUCT, CATALOG 和 PRICE 表之间有四个内部连接和一个没有产生关联的字查询。

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
```

```
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

好家伙，真长！实际上，在现实生活中我并不是非常热衷于子查询，所以我的查询往往是这样的：

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

下面的查询统计付款记录处于每种状态中的数量，要排除所有处于 `AWAITING_APPROVAL` 状态的，或者最近一次状态更改是由当前用户做出的。它翻译成 SQL 查询后，在 `PAYMENT`, `PAYMENT_STATUS` 和 `PAYMENT_STATUS_CHANGE` 表之间包含两个内部连接和一个用于关联的子查询。

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

假若我已经把 `statusChange` 集合映射为一个列表而不是一个集合的话，查询写起来会简单很多。

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user
<> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

下面的查询使用了 MS SQL Server 的 isNull()函数，返回当前用户所属的组织所有账户和未付支出。翻译为 SQL 查询后，在 ACCOUNT， PAYMENT， PAYMENT_STATUS，ACCOUNT_TYPE，ORGANIZATION 和 ORG_USER 表之间有三个内部连接，一个外部连接和一个子查询。

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name,
PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber,
payment.dueDate
```

对某些数据库而言，我们可能不能依赖（关联的）子查询。

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name,
PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber,
payment.dueDate
```

10.13. 提示和技巧（Tips & Tricks）

你不返回结果集也可以查询结果集的大小：

```
( (Integer) session.iterate("select count(*)
from ....").next() ).intValue()
```

要依据一个集合的大小对结果集排序，可以用下面的查询来对付一对多或多对多的关联：

```
select usr
from User as usr
    left join usr.messages as msg
group by usr
order by count(msg)
```

如果你的数据库支持子查询，你可以在查询的 where 子句中对选择的大小进行条件限制：

```
from User usr where size(usr.messages) >= 1
```

如果你的数据库不支持子查询，可以使用下列查询：

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

因为使用了 inner join,这个解决方法不能返回没有 message 的 User.下面的方式就可以：

```
select usr
from User as usr
    left join usr.messages as msg
group by usr
having count(msg) = 0
```

JavaBean 的属性可以直接作为命名的查询参数：

```
Query q = s.createQuery("from foo in class Foo where foo.name=:name
and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

在 Query 接口中使用过滤器(filter),可以对集合分页：

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

集合元素可以使用查询过滤器(query filter)进行排序或者分组：

```
List orderedCollection = s.filter( collection, "order by
this.amount" );
List counts = s.filter( collection, "select this.type, count(this)
group by this.type" );
```

不用初始化集合就可以得到其大小：

```
( (Integer) session.iterate("select count(*)
from ....").next() ).intValue();
```

Chapter 11. 实例(A Worked Example)

我们用实例来演示上两章所讲述的概念。

11.1. 持久化类

下面的两个持久化类表示一个 weblog,和在其中张贴的一个贴子。他们是标准的父/子关系模型，但是我们会用一个排序包（ordered bag)而非集合(set)。

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;
```

```
    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
```

```
    }
}
```

11.2. Hibernate 映射

下列的 XML 映射应该是很直白的。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">
    <class
        name="Blog"
        table="BLOGS"
        lazy="true">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            lazy="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">
```

```
        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

11.3. Hibernate 代码

下面的类演示了我们可以使用 Hibernate 对这些类所进行的操作。

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Query;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
```

```
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.save(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String
text) throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }
```

```
    public BlogItem createBlogItem(Long blogid, String title, String
text) throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text) throws
HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text) throws
HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class,
itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
```

```
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max) throws
HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid) throws
HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.list().get(0);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
```

```
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

Chapter 12. 性能提升（Improving Performance）

我们已经为您展示了如何在对集合持久化时使用延迟装载（lazy initialization）。对于通常的对象引用，使用 CGLIB 代理可以达到类似的效果。我们也提到过 Hibernate 在 Session 级别缓存持久化对象。还有更多先进的缓存策略，你可以为每一个类单独配置。

这一章里，我们来教你如何使用这些特性，在必要的时候得到高得多的性能。

12.1. 用于延迟装载的代理

Hibernate 使用动态字节码增强技术来实现持久化对象的延迟装载代理（使用优秀的 CGLIB 库）。

映射文件为每一个类声明一个类或者接口作为代理接口。建议使用这个类自身：

```
<class name="eg.Order" proxy="eg.Order">
```

运行时的代理应该是 Order 的子类。注意被代理的类必须实现一个默认的构造器，并且至少在包内可见。

在扩展这种方法来对应多形的类时，要注意一些细节，比如：

```
<class name="eg.Cat" proxy="eg.Cat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.DomesticCat">
        .....
    </subclass>
</class>
```

首先，Cat 永远不能被强制转换为 DomesticCat,即使实际上该实例就是一个 DomesticCat 实例。

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy
(does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to
initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

其次，代理的==可能不再成立。

```

Cat cat = (Cat) session.load(Cat.class, id);            //
instantiate a Cat proxy
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id);  // required
new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

虽然如此，这种情况并不像看上去得那么糟。虽然我们现在有两个不同的引用来指向不同的代理对象，实际上底层的实例应该是同一个对象：

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

第三，你不能对 final 的类或者具有 final 方法的类使用 CGLIB 代理。

最后,假如你的持久化对象在实例化的时候需要某些资源（比如，在实例化方法或者默认构造方法中），这些资源也会被代理需要。代理类实际上是持久化类的子类。

这些问题都来源于 Java 的单根继承模型的天生限制。如果你希望避免这些问题，你的每个持久化类必须抽象出一个接口，声明商业逻辑方法。你应该在映射文件中指定这些接口，比如：

```
<class name="eg.Cat" proxy="eg.ICat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.IDomesticCat">
        .....
    </subclass>
</class>
```

这里 Cat 实现 ICat 接口，并且 DomesticCat 实现 IDomesticCat 接口。于是 load()或者 iterate()就会返回 Cat 和 DomesticCat 的实例的代理。(注意 find()不会返回代理。)

```
ICat cat = (ICat) session.load(Cat.class, catid);
Iterator iter = session.iterate("from cat in class eg.Cat where
cat.name='fritz'");
ICat fritz = (ICat) iter.next();
```

关系也是延迟装载的。这意味着你必须把任何属性声明为 ICat 类型，而非 Cat。

某些特定操作*不*需要初始化代理

- equals()，假如持久化类没有重载 equals()
- hashCode()，假如持久化类没有重载 hashCode()
- 标识符的 get 方法

Hibernate 会识别出重载了 equals() 或者 hashCode()方法的持久化类。

在初始化代理的时候发生的异常会被包装成 LazyInitializationException。

有时候我们需要保证在 Session 关闭前某个代理或者集合已经被初始化了。当然，我们总是可以通过调用 cat.getSex()或者 cat.getKittens().size()之类的方法来确保这一点。但是这样程序可读性不佳，也不符合通常的代码规范。静态方法 Hibernate.initialize()和 Hibernate.isInitialized()给你的应用程序一个正常的途径来加载集合或代理。Hibernate.initialize(cat) 会强制初始化一个代理,cat,只要它的 Session 仍然打开。Hibernate.initialize(cat.getKittens())对 kittens 的集合具有同样的功能。

12.2. 第二层缓存(The Second Level Cache)s

HibernateSession 是事务级别的持久化数据缓存。再为每个类或者每个集合配置一个集群或者 JVM 级别(SessionFactory 级别)的缓存也是有可能的。你甚至可以插入一个集群的缓存。要小心，缓存永远不会知道其他进程可能对持久化仓库（数据库）进行的修改（即使他们可能设定为经常对缓存的数据进行失效）。

默认情况下，Hibernate 使用 EHCache 进行 JVM 级别的缓存。但是，对 JCS 的支持现在已经被废弃了，未来版本的 Hibernate 将会去掉它。通过 hibernate.cache.provider_class 属性，你也可以指定其他缓存，只要其实现了 net.sf.hibernate.cache.CacheProvider 接口。

Table 12.1. Cache Providers

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	net.sf.hibernate.cache.HashtableCacheProvider	memory		yes
EHCache	net.sf.ehcache.hibernate.Provider	memory, disk		yes
OSCache	net.sf.hibernate.	memory,		yes

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
	cache.OSCacheProvider	disk		
SwarmCache	net.sf.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	net.sf.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactional	yes (replication)	

12.2.1. 映射(Mapping)

类或者集合映射的<cache>元素可能有下列形式：

```
<cache                                                    ❶
                usage="transactional|read-write|nonstrict-read-write|read-only" />
```

❶ usage 指定了缓存策略： transactional, read-write, nonstrict-read-write 或者 read-only

另外 (推荐首选?), 你可以在 hibernate.cfg.xml 中指定<class-cache> 和 <collection-cache> 元素。

usage 属性指明了*缓存并发策略（cache concurrency strategy）*。

12.2.2. 只读缓存

如果你的应用程序需要读取一个持久化类的实例，但是并不打算修改它们，可以使用 read-only 缓存。这是最简单，也是实用性最好的策略。甚至在集群中，它也能完美地运作。

```
<class name="eg.Immutable" mutable="false">
    ....
    <jcs-cache usage="read-only"/>
</class>
```

12.2.3. 读/写缓存

如果应用程序需要更新数据，可能 read-write 缓存比较合适。如果需要可序列化事务隔离级别（serializable transaction isolation level），这种缓存决不能使用。如果在 JTA 环境中使用这种缓存，你必须指定 hibernate.transaction.manager_lookup_class 属性的值，给出得到 JTA TransactionManager 的策略。在其它环境中，你必须确保在 Session.close() 或者 Session.disconnect()调用前，事务已经结束了。 如果你要在集群环境下使用这一策略，你必须确保底层的缓存实现支持锁定(locking)。内置的缓存提供器*并不*支持。

```
<class name="eg.Cat" .... >
    <jcs-cache usage="read-write"/>
    ....
    <set name="kittens" ... >
```

```
        <jcs-cache usage="read-write"/>
        ....
    </set>
</class>
```

12.2.4. 不严格的读/写缓存

如果程序偶尔需要更新数据（也就是说，出现两个事务同时更新同一个条目的现象很不常见），也不需要十分严格的事务隔离，可能适用 `nonstrict-read-write` 缓存。如果在 JTA 环境中使用这种缓存，你必须指定 `hibernate.transaction.manager_lookup_class` 属性的值，给出得到 JTA `TransactionManager` 的策略。在其它环境中，你必须确保在 `Session.close()`或者 `Session.disconnect()`调用前，事务已经结束了。

12.2.5. 事务缓存（transactional）

`transactional` 缓存策略提供了对全事务缓存提供,比如 JBoss TreeCache 的支持。这样的缓存只能用于 JTA 环境，你必须指定 `hibernate.transaction.manager_lookup_class`。

没有一种缓存提供器能够支持所有的缓存并发策略。下面的表列出每种提供器与各种并发策略的兼容性。

Table 12.2. 缓存并发策略支持(Cache Concurrency Strategy Support)

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

12.3. 管理 Session 缓存

不管何时你传递一个对象给 `save()`, `update()`或者 `saveOrUpdate()` ,或者不管何时你使用 `load()`, `find()`, `iterate()`或者 `filter()`取得一个对象的时候,该对象被加入到 `Session` 的内部缓存中。当后继的 `flush()`被调用时，对象的状态会和数据库进行同步。如果你在处理大量对象并且需要有效的管理内存的时候，你可能不希望发生这种同步，`evict()`方法可以从缓存中去掉对象和它的集合。

```
Iterator cats = sess.iterate("from eg.Cat as cat"); //a huge result
set
while ( cats.hasNext() ) {
    Cat cat = (Cat) iter.next();
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

`Session` 也提供了一个 `contains()`方法来判断是否一个实例处于这个 session 的缓存中。

要把所有的对象从 session 缓存中完全清除，请调用 `Session.clear()`。

For the `JVM-level` JCS cache, there are methods defined on `SessionFactory` for evicting the cached state of an instance, entire class, collection instance or entire collection role.

对于第二层缓存来说，在 `SessionFactory` 中定义了一些方法来从缓存中清除一个实例、整个类、集合实例或者整个集合。

12.4. 查询缓存(Query Cache)

查询结果集也可以被缓存。只有当经常使用同样的参数进行查询时，这才会有些用处。要使用查询缓存，首先你要打开它，设置 `hibernate.cache.use_query_cache=true` 这个属性。这样会创建两个缓存区域——一个保存查询结果集(`net.sf.hibernate.cache.QueryCache`),另一个保存最近查询的表的时间戳(`net.sf.hibernate.cache.UpdateTimestampsCache`)。请注意查询缓存并不缓存结果集中包含实体的状态；它只缓存标识符属性的值和值类型的结果。所以查询缓存通常会和第二层缓存一起使用。

大多数查询并不会从缓存中获得什么好处，所以默认查询是不进行缓存的。要进行缓存，调用 `Query.setCacheable(true)`。这个调用会让查询在执行时去从缓存中查找结果，或者把结果集放到缓存去。

如果你要对查询缓存的失效政策进行精确的控制，你必须调用 `Query.setCacheRegion()`来为每个查询指定一个命名的缓存区域。

```
List blogs = sess.createQuery("from Blog blog where blog.blogger
= :blogger order by blog.datetime desc")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

Chapter 13. 理解集合类的性能（Understanding Collection Performance）

我们已经在集合类(collections)上面花了很多口舌了。这一章，我们要着重关注集合类在运行时的一些问题。

13.1. 分类（Taxonomy）

Hibernate 定义了三种不同的基本集合类

- 值集合
- 一对多关联
- 多对多关联

这个分类方法是根据不同的表和外键关系来区分的，但是没有确切的告诉我们关系模型。要完整的了解关系结构和性能的区别，我们必须考虑 Hibernate 再更新或者删除集合类记录时的主键结构。这样的话，我们会得到如下的分类：

- 有序集合类
- 集合（sets）
- 包（bags）

所有的有序集合类（map,list,array)都有一个由<key>和<index>组合的主键字段。这种集合类的更新非常高效——主键有效地排了序，Hibernate 要更新或者删除一个元素的时候可以很高效的找到它。

集合(Sets)有一个由<key>和某个元素字段组成的主键。对于某些元素类型，特别是组合元素以及大本文、二进制字段，效率会来的比较低；数据库无法对复杂的主键有效索引。另一方面，对一对多关系或者多对多关系来说，特别是使用“人造”的标识符的时候，它的性能同样出色。（注：如果你希望 SchemaExport 为你创建一个<set>的主键，你必须把所有的字段都声明成 non-null="true"。）

包(Bags)是最差劲的。因为包允许元素值重复，也没有索引字段，所以无法定义主键。Hibernate 没有办法来区分重复的行。Hibernate 的处理方法是，每当更改的时候，完全删除（用一个 DELETE),再重新创建这个集合类。这可能是效率极低的。

请注意，对于一对多关联，"主键"可能不是数据库表的物理主键——当时就算考虑这种情况，上面分类描述仍然是正确的。(反映了 Hibernate 是如何在不同的集合类中“定位”某条记录的。)

13.2. Lists, maps 和 sets 用于更新效率最高

根据我们上面的讨论，显然有序类型和大多数 set 可以在增加/删除/修改元素的时候得到最好的性能。

但是，在多对多关联，或者对值元素而言，有序集合类比集合(set)有一个好处。因为 Set 的结构，如果“改变”了一个元素,Hibernate 并不会 UPDATE 这一行。对 Set 来说，只有 INSERT 和 DELETE 才有效。注意这一段描述对一对多关联并不适用。

注意到数组无法延迟转载，我们可以得出结论，list, map 和 set 是最高效的集合类型。（当然，我们警告过了，由于集合中的值的关系，set 可能性能下降。）

Set 可以被看作是 Hibernate 程序中最普遍的集合类型。

这个版本的 Hibernate 有一个没有写在文档中的功能。<idbag>可以对值集合和多对多关联实现 bag 语义，并且性能比上面任何类型都高！

13.3. Bag 和 list 是反向集合类中效率最高的

好了，在你把 bag 扔到水沟里面再踩上一只脚之前，有一种情况下 bag(包括 list)要比 set 性能高得多。对于指明了 inverse="true"的集合类（比如说，标准的双向一对多关联），我们可以在不初始化(fetch)包元素的情况下就增加新元素！这是因为 Collection.add()或者 Collection.addAll()对 bag 或者 List 总是返回 true 的（与 Set 不同）。对于下面的代码来说，速度会快得多。

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);  //no need to fetch the collection!
```

```
sess.flush();
```

13.4. 一次性删除(One shot delete)

有时候，一个一个的删除集合类中的元素是极度低效的。Hibernate 没那么笨，如果你想要把整个集合都删除（比如说调用 list.clear()），Hibernate 只需要一个 DELETE 就搞定了。

假设我们在一个长度为 20 的集合类中新增加了一个元素，然后删除了两个。Hibernate 会安排一个 INSERT 语句和两条 DELETE 语句（除非集合类是一个 bag)。这当然是可以想见的。

但是，如果假设我们删除了 18 个元素，只剩下 2 个，然后新增 3 个。有两种处理方式：

- 把这 18 个元素一个一个的干掉，再新增三个
- 把整个集合类都咔嚓掉（只用一句 DELETE 语句），然后增加 5 个元素。

Hibernate 还没那么聪明，知道第二种选择可能会比较快。（也许让 Hibernate 不要这么聪明也是好事，否则可能会引发意外的数据库触发器什么的。）

幸运的是，你可以强制使用第二种策略。你需要把原来的整个集合类都取消（取消其引用），然后返回一个新实例化的集合类，只包含需要的元素。有些时候这是非常有用的。

Chapter 14. 条件查询(Criteria Query)

现在 Hibernate 也支持一种直观的、可扩展的条件查询 API。目前为止，这个 API 还没有更成熟的 HQL 查询那么强大，也没有那么多查询能力。特别要指出，条件查询也不支持投影（projection）或统计函数（aggregation）。

14.1. 创建一个 Criteria 实例

net.sf.hibernate.Criteria 这个接口代表对一个特定的持久化类的查询。Session 是用来制造 Criteria 实例的工厂。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

14.2. 缩小结果集范围

一个查询条件(Criterion)是 net.sf.hibernate.expression.Criterion 接口的一个实例。类 net.sf.hibernate.expression.Expression 定义了获得一些内置的 Criterion 类型。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.between("weight", minWeight, maxWeight) )
    .list();
```

表达式（Expressions）可以按照逻辑分组.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
```

```
    .add( Expression.or(
        Expression.eq( "age", new Integer(0) ),
        Expression.isNull("age")
    ) )
    .list();
List cats = sess.createCriteria(Cat.class)
    .add( Expression.in( "name", new String[] { "Fritz", "Izi",
"Pk" } ) )
    .add( Expression.disjunction()
        .add( Expression.isNull("age") )
        .add( Expression.eq("age", new Integer(0) ) )
        .add( Expression.eq("age", new Integer(1) ) )
        .add( Expression.eq("age", new Integer(2) ) )
    ) )
    .list();
```

有很多预制的条件类型（Expression 的子类）。有一个特别有用，可以让你直接嵌入 SQL。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.sql("lower($alias.name) like lower(?)",
"Fritz%", Hibernate.STRING) )
    .list();
```

其中的{alias}是一个占位符，它将会被所查询实体的行别名所替代。(原文:The {alias} placeholder with be replaced by the row alias of the queried entity.)

14.3. 对结果排序

可以使用 net.sf.hibernate.expression.Order 对结果集排序.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

14.4. 关联（Associations）

你可以在关联之间使用 createCriteria()，很容易地在存在关系的实体之间指定约束。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .createCriteria("kittens")
        .add( Expression.like("name", "F%")
    .list();
```

注意，第二个 createCriteria()返回一个 Criteria 的新实例，指向 kittens 集合类的元素。

下面的替代形式在特定情况下有用。

```
List cats = sess.createCriteria(Cat.class)
```

```
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Expression.eqProperty("kt.name", "mt.name") )
    .list();
```

（createAlias()）并不会创建一个 Criteria 的新实例。）

请注意，前面两个查询中 Cat 实例所持有的 kittens 集合类*并没有*通过 criteria 预先过滤！如果你希望只返回满足条件的 kittens,你必须使用 returnMaps()。

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Expression.eq("name", "F%") )
    .returnMaps()
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

14.5. 动态关联对象获取（Dynamic association fetching）

可以在运行时通过 setFetchMode()来改变关联对象自动获取的策略。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .list();
```

这个查询会通过外连接(outer join)同时获得 mate 和 kittens。

14.6. 根据示例查询（Example queries）

net.sf.hibernate.expression.Example 类允许你从指定的实例创造查询条件。

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

版本属性，表示符属性和关联都会被忽略。默认情况下，null 值的属性也被排除在外。

You can adjust how the Example is applied. 你可以调整示例(Example)如何应用。

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
```

```
    .ignoreCase()              //perform case insensitive string
comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

你甚至可以用示例对关联对象建立 criteria。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

Chapter 15. SQL 查询

你也可以直接使用你的数据库方言表达查询。在你想使用数据库的某些特性的时候，这是非常有用的， 比如 Oracle 中的 CONNECT 关键字。这也会扫清你把原来直接使用 SQL/JDBC 的程序移植到 Hibernate 道路上的障碍。

15.1. 创建一个基于 SQL 的 Query

和普通的 HQL 查询一样，SQL 查询同样是从 Query 接口开始的。惟一的区别是使用 Session.createSQLQuery()方法。

```
Query sqlQuery = sess.createSQLQuery("select {cat.*} from cats
{cat}", "cat", Cat.class);
sqlQuery.setMaxResults(50);
List cats = sqlQuery.list();
```

传递给 createSQLQuer()的三个参数是:

- SQL 查询语句
- 表的别名
- 查询返回的持久化类

别名是为了在 SQL 语句中引用对应的类（本例中是 Cat)的属性的。你也可以传递一个别名的 String 数组和一个对应的 Class 的数组进去，每行就可以得到多个对象。

15.2. 别名和属性引用

上面使用的{cat.*}标记是“所有属性的”的简写。你可以显式的列出需要的属性，但是你必须让 Hibernate 为每个 属性提供 SQL 列别名。这些列的的占位表示符是以表别名为前导，再加上属性名。下面的例子中，我们从一个其它的表(cat_log) 中获取 Cat 对象，而非 Cat 对象原本在映射元数据中声明的表。注意你在 where 子句中也可以使用 属性别名。

```
String sql = "select cat.originalId as {cat.id}, cat.mateid as
{cat.mate}, cat.sex as {cat.sex}, cat.weight*10 as {cat.weight},
cat.name as {cat.name}"
    + " from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql, "cat", Cat.class)
```

```
        .setLong("catId", catId)
        .list();
```

注意： 如果你明确的列出了每个属性，你必须包含这个类*和它的子类*的属性！//??

15.3. 为 SQL 查询命名

可以在映射文档中定义 SQL 查询的名字，然后就可以像调用一个命名 HQL 查询一样直接调用命名 SQL 查询。

```
List people = sess.getNamedQuery("mySqlQuery")
    .setMaxResults(50)
    .list();
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person"/>
    SELECT {person}.NAME AS {person.name}, {person}.AGE AS
{person.age}, {person}.SEX AS {person.sex}
    FROM PERSON {person} WHERE {person}.NAME LIKE 'Hiber%'
</sql-query>
```

Chapter 16. 继承映射(Inheritance Mappings)

16.1. 三种策略

Hibernate 支持三种不同的基本继承映射策略。

- 每棵类继承树使用一个表(table per class hierarchy)
- 每个子类一个表(table per subclass)
- 每个具体类一个表(table per concrete class)（有一些限制）

甚至在一棵继承关系书中对不同的分支使用不同的映射策略也是可能的。但是和“每个具体类一个表”的映射有一样的限制。Hibernate 不支持把`<subclass>`映射与`<joined-subclass>`在同一个`<class>` 元素中混合使用。

假设我们有一个 `Payment` 接口，有不同的实现：`CreditCardPayment`，`CashPayment`，`ChequePayment`。“继承数共享一个表”的映射是这样的：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
```

```
</class>
```

只需要一个表。这种映射策略由一个大限制：子类定义的字段不能有 NOT NULL 限制。

“每个子类一个表”的映射是这样的：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </subclass>
</class>
```

需要四个表。三个子类表通过主键和超类表关联（所以实际上关系模型是一对一关联）。

注意 Hibernate 的“每子类一表”的实现并不需要一个特别的辨认字段。其他的对象/关系数据库映射工具使用另一种“每子类一表”实现，需要在超类表中有一个类型辨认字段。Hibernate 的这种实现更加困难，但是从关系(数据库)的角度来看，这样做更加正确。

对这两种映射策略来说，指向 Payment 的关联是使用<many-to-one>进行映射的。

```
<many-to-one name="payment"
    column="PAYMENT"
    class="Payment"/>
```

“每个具体类一个表”的策略非常不同

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>
```

```
<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

需要三个表。注意我们没有明确的定义 Payment 接口。我们用 Hibernate 的*隐含多态（implicit polymorphism）*机制代替。也要注意 Payment 的属性在三个字类中都进行了映射。

这种情形下，与 Payment 关联的多态关联被映射为<any>。

```
<any name="payment"
        meta-type="class"
        id-type="long">
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

如果我们定义 UserType 和 meta-type 来根据不同的标识字符串映射到 Payment，事情会更好一些。

```
<any name="payment"
        meta-type="PaymentMetaType"
        id-type="long">
    <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH or CHEQUE -->
    <column name="PAYMENT_ID"/>
</any>
```

对这个映射还有一点需要注意。因为每个子类都在各自独立的<class>元素中映射（并且 Payment 只是个接口），每个子类都可以和容易的成为另一个"每个类一个表"或者"每个子类一个表"的继承树！（并且你仍然可以对 Payment 接口使用多态查询。）

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
```

```
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

我们再一次没有明确的提到 Payment。如果我们针对 Payment 接口执行查询 ——比如，from Payment—— Hibernate 自动返回 CreditCardPayment 实例（以及它的子类，因为它们也继承了 Payment），CashPayment 和 Chequepayment，但是不会是 NonelectronicTransaction 的实例。

16.2. 限制

Hibernate 假设关联严格的和一个外键字段相映射。如果一个外键具有多个关联，也是可以容忍的（你可能需要指定 inverse="true"或者 insert="false" update="false"），但是你不能为多重外键指定任何映射的关联。这意味着：

- 当更改一个关联的时候，永远是更新的同一个外键
- 当一个关联是延迟抓取(fetched lazily)的时候，只需要用一次数据库查询
- 当一个关联是提前抓取(fetched eagerly)的时候，使用一次 outer join 即可

特别要指出的是，使用“每个具体类一个表”的策略来实行多态的一对多关联是*不支持*的。（抓取这样的关联需要多次查询或者多次 join。）

下面的表格列出了 Hibernte 中，“每个具体类一个表”策略与隐含多态机制的限制。

Table 16.1. 继承映射时的隐含多态（Implicit polymorphism in inhertiance mappings）

继承策略（Inheritance strategy）	多态多对一	多态一对一	多态一对多	多态多对多	多态 **load()/get()**	多态查询	多态连接(join)	Outer join 抓取
每继承树一表	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p	*支持*
每子类一表	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p	*支持*
每类一表(隐含多态)	<any>	*不支持*	*不支持*	<many-to-any>	*use a query*	from Payment p	*不支持*	*不支持*

Chapter 17. 事务和并行（Transactions And Concurrency）

Hibernate 本身并不是数据库，它只是一个轻量级的对象－关系数据库映射（object-relational）工具。它的事务交由底层的数据库连接管理，如果数据库连接有 JTA 的支持，那

么在 Session 中进行的操作将是整个原子性 JTA 事务的一部分。Hibernate 可以看作是添加了面向对象语义的 JDBC 瘦适配器（thin adapter）。

17.1. 配置，会话和工厂（Configurations, Sessions and Factories）

SessionFactory 的创建需要耗费大量资源，它是线程安全（threadsafe）的对象，在应用中它被所有线程共享。而 Session 的创建耗费资源很少，它不是线程安全的对象，对于一个简单商业过程（business process），它应该只被使用一次，然后被丢弃。举例来说，当 Hibernate 在基于 servlet 的应用中，servlet 能够以下面的方式得到 SessionFactory。

```
SessionFactory sf =
(SessionFactory)getServletContext().getAttribute("my.session.factory"
);
```

每次调用 SessionFactory 的 service 方法能够生成一个新的 Session 对象，然后调用 Session 的 flush()，调用 commit()提交它的连接，调用 close()关闭它，最终丢弃它。

在无状态的 session bean 中，可以同样使用类似的方法。bean 在 setSessionContext()中得到 SessionFactory 的实例，每个商业方法会生成一个 Session 对象，调用它的 flush()和 close()，当然，应用不应该 commit()connection.（把它留给 JTA.）

这里需要理解 flush()的含义。 flush()将持久化存储与内存中的变化进行同步，但*不是*将内存的变化与持久化存储进行同步。所以在调用 flush()并接着调用 commit()关闭连接时，会话将仍然含有过时的数据，在这种情况下，继续使用会话的*唯一*的方法是将会话中的数据进行版本化。

接下来的几小节将讨论利用版本化的方法来确保事务原子性，这些“高级”方法需要小心使用。

17.2. 线程和连接（Threads and connections）

You should observe the following practices when creating Hibernate Sessions:

在创建 Hibernate 会话（Session）时，你应该留意以下的实践（practices）：

- 对于一个数据库连接，不要创建一个以上的 Session 或 Transaction
- 在对于一个数据库连接、一个事务使用多个 Session 时，你尤其需要格外地小心。Session 对象会记录下调入数据更新的情况，所以另一个 Session 对象可能会遇到过时的数据。
- Session *不是*线程安全的。如果确实需要在两个同时运行的线程中共享会话，那么你应该确保线程在访问会话时，线程对 Session 具有同步锁。

17.3. 乐观锁定／版本化（Optimistic Locking / Versioning）

许多商业过程需要一系列与用户进行交互的过程，数据库访问穿插在这些过程中。对于 web 和企业应用来说，跨一个用户交互过程的数据事务是不可接受的，因而维护各商业事务间的隔离（isolocation）就成为应用层的部分责任。唯一满足高并发性以及高可扩展性的方法是使用带有版本化的乐观锁定。Hibernate 为使用乐观锁定的代码提供了三种可能的方法。

17.3.1. 使用长生命周期带有自动版本化的会话

在整个商业过程中使用一个单独的 Session 实例以及它的持久化实例，这个 Session 使用带有版本化的乐观锁定机制，来确保多个数据库事务对于应用来说只是一个逻辑上的事务。在等待用户交互时，Session 断开与数据库的连接。这个方法从数据库访问方面来看是最有效的，应用不需要关心对自己的版本检查或是重新与不需要序列化（transient）的实例进行关联。

```
// foo is an instance loaded earlier by the Session
session.reconnect();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

17.3.2. 使用带有自动版本化的多个会话

每个与持久化存储的交互出现在一个新的 Session 中，在每次与数据库的交互中，使用相同的持久化实例。应用操作那些从其它 Session 调入的不需要持久化实例的状态，通过使用 Session.update()或者 Session.saveOrUpdate()来重新建立与它们的关联。

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
session.flush();
session.connection().commit();
session.close();
```

17.3.3. 应用程序自己进行版本检查

每当一个新的 Session 中与持久化存储层出现交互的时候，这个 session 会在操作持久化实例前重新把它们从数据存储中装载进来。我们现在所说的方式就是你的应用程序自己使用版本检查来确保商业过程的隔绝性。（当然，Hibernate 仍会为你*更新*版本号）。从数据库访问方面来看，这种方法是最没有效率的，与 entity EJB 方式类似。

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new
StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

当然，如果在低数据并行（low-data-concurrency）的环境中，并不需要版本检查，你仍可以使用这个方法，只需要忽略版本检查。

17.4. 会话断开连接（Session disconnection）

The first approach described above is to maintain a single Session for a whole business process thats spans user think time. (For example, a servlet might keep a Session in the user's HttpSession.) For performance reasons you should

上面提到的第一种方法是对于对一个用户的一次登录产生的整个商业过程维护一个 Session。（举例来说，servlet 有可能会在用户的 HttpSession 中保留一个 Session）。为性能考虑，你必须

- 提交 Transaction（或者 JDBC 连接），然后
- (在等待用户操作前，)断开 Session 与 JDBC 连接。

Session.disconnect()方法会断开会话与 JDBC 的连接，把连接返还给连接池（除非是你自己提供这个连接的）。

Session.reconnect()方法会得到一个新的连接（你也可以自己提供一个），重新开始会话。在重新连接后，你可以通过对任何可能被其它事务更新的对象调用 Session.lock()方法，来强迫对你没有更新的数据进行版本检查。你不需要对*正在*更新的数据调用 lock()。

这是一个例子：

```
SessionFactory sessions;
List fooList;
Bar bar;
....
Session s = sessions.openSession();

Transaction tx = null;
try {
    tx = s.beginTransaction();

    fooList = s.find(
        "select foo from eg.Foo foo where foo.Date = current date"
        // uses db2 date function
    );
    bar = (Bar) s.create(Bar.class);

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    s.close();
    throw e;
}
s.disconnect();
```

接下来：

```
s.reconnect();

try {
    tx = sessions.beginTransaction();

    bar.setFooTable( new HashMap() );
    Iterator iter = fooList.iterator();
    while ( iter.hasNext() ) {
        Foo foo = (Foo) iter.next();
```

```
        s.lock(foo, LockMode.READ);    //check that foo isn't stale
        bar.getFooTable().put( foo.getName(), foo );
    }

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    s.close();
}
```

从上面的例子可以看到 Transaction 和 Session 之间是多对一的关系。一个 Session 表示了应用程序与持久存储之间的一个对话，Transaction 把这个对话分隔成一个个具有原子性的单元。

17.5. 悲观锁定（Pessimistic Locking）

用户不需要在锁定策略上花费过多时间，通常我们可以选定一种隔离级别（isolationn level），然后让数据库完成所有的工作。高级用户可能希望得到悲观锁定或者在新的事务开始时重新得到锁。

LockMode 类定义了 Hibernate 需要的不同的锁级别。锁由以下的机制得到：

- LockMode.WRITE 在 Hibernate 更新或插入一行数据时自动得到。
- LockMode.UPGRADE 在用户通过 SELECT ... FOR UPDATE 这样的特定请求得到，需要数据库支持这种语法。
- LockMode.UPGRADE_NOWAIT 在用户通过 SELECT ... FOR UPDATE NOWAIT 这样的特定请求在 Oracle 数据库环境下得到。
- LockMode.READ 在 Hibernate 在不断读（Repeatable Read）和序列化（Serializable）的隔离级别下读取数据时得到。也可以通过用户的明确请求重新获得。
- LockMode.NONE 表示没有锁。所有对象在 Transaction 结束时会切换到这种锁模式，通过调用 update()或者 saveOrUpdate()与会话进行关联的对象,开始时也会在这种锁模式。

“明确的用户请求”会以下的几种方式出现：

- 调用 Session.load()，指定一种 LockMode。
- 调用 Session.lock()。
- 调用 Query.setLockMode()。

如果在调用 Session.load()时指定了 UPGRADE 或者 UPGRADE_NOWAIT，并且请求的对象还没有被会话调入，那么这个对象会以 SELECT ... FOR UPDATE 的方式调入。如果调用 load()在一个已经调入的对象，并且这个对象调入时的锁级别没有请求时来得严格，Hibernate 会对这个对象调用 lock()。

Session.lock()会执行版本号检查的特定的锁模式是：READ，UPGRADE 或者 UPGRADE_NOWAIT。（在 UPGRADE 或者 UPGRADE_NOWAIT，SELECT ... FOR UPGRADE 使用的情况下。）

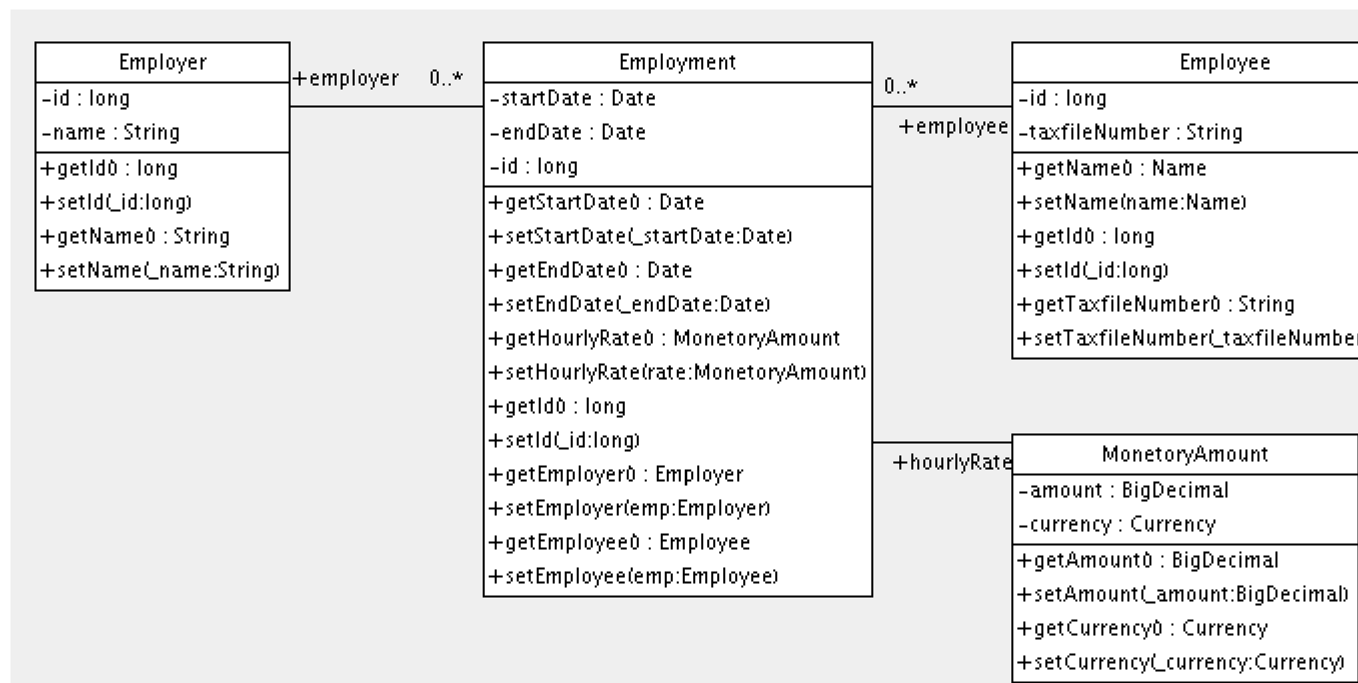
如果数据库不支持所请求的锁模式，Hibernate 将会选择一种合适的受支持的锁模式替换（而不是抛出一个异常）。这确保了应用具有可移植性。

Chapter 18. 映射实例(Mapping Examples)

本章将为你展示几个比较复杂的关联映射。

18.1. 雇员 / 雇主（Employer/Employee）

接下来关于 Employer 和 Employee 关系的模型使用了一个实体（entity）类（Employment）来表示这个关联，因为对于相同的雇主和雇员可能会有多个雇用时间段。对于雇用金额和雇员姓名，我们使用组件（component）来进行建模。



这是一个可行的映射文档：

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetoryAmount">
```

```
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12,
2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-
null="true"/>
        <many-to-one name="employee" column="employee_id" not-
null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

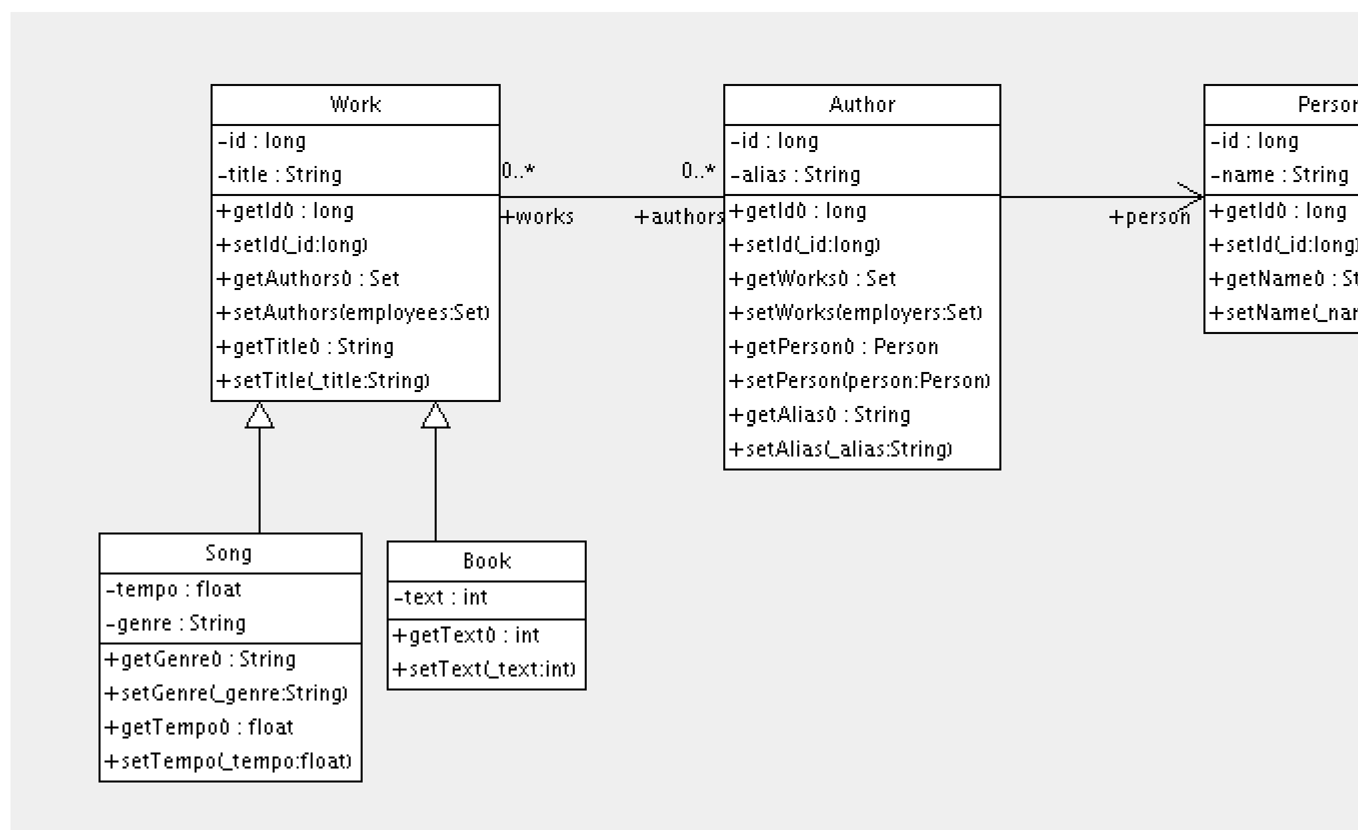
这里是由 SchemaExport 生成的表结构。

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
```

```
    add constraint employment_periodsFK0 foreign key (employer_id)
references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id)
references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

18.2. 作者 / 著作(Author/Work)

下面的例子是关于 Work、Author 和 Person。我们用多对多关系来表示 Work 和 Author 之间的关联，用一对一的关系来表示 Author 和 Person 之间的关联。另外一种可行的方式是对 Author 扩展 Person。



接下来的映射文档正确地表示这些关系：

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work" lazy="true">
            <key>
                <column name="work_id" not-null="true"/>
```

```
            </key>
            <many-to-many class="Author">
                <column name="author_id" not-null="true"/>
            </many-to-many>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the
Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true"
lazy="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

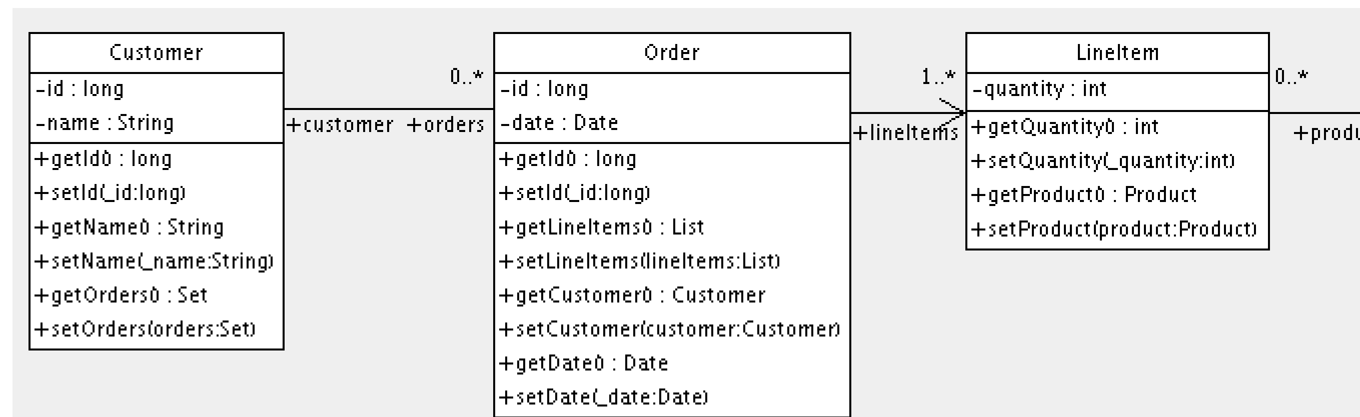
在这个映射中有四个表。works，authors 和 persons 分别存放著作、作者以及人的数据。author_work 是关联表，把作者与著作关联起来。以下是由 SchemaExport 生成的表结构。

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
```

```
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references
authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references
works
```

18.3. 客户／订单／产品(Customer/Order/Product)

接下来的例子是关于 Customer、Order、LineItem 和 Product。Customer 和 Order 之间是一对多的关联。那么 Order / LineItem / Product 之间的关联怎么表示呢？我们可以把 LineItem 作为关联表来表示 Order 和 Product 之间多对多关联，在 Hibernate 里，它被称为组合元素（composite element）。



映射文档：

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true" lazy="true">
```

```
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items" lazy="true">
            <key column="order_id"/>
            <index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

customers、orders、line_items 和 products 分别存放客户、订单、订单项以及产品的数据。line_items 作为关联表，把订单和产品关联起来。

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)
```

```
alter table orders
    add constraint ordersFK0 foreign key (customer_id) references
customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references
products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references
orders
```

Chapter 19. 工具箱指南

通过 Hibernate 项目中提供的几个命令行工具(他们也被当作项目的一部分不断得到维护),还有 XDoclet,Middlegen 和 AndroMDA 内置的对 Hibernate 的支持,可以在几个不同的环境(SQL,java 代码,xml 映射文件)中进行相互转换(roundtrip)。

Hibernate 的主发行包中附带了最重要的工具（甚至在 Hibernate 内部也可以快速调用这个工具）：

- 从映射文件到 DDL schema 的生成器（也就是 SchemaExport 和 hbm2ddl）

Hibernate 项目直接提供的其他工具在一个单独的发行包中发布，*Hibernate Extensions*。这个发行包包含了下列任务的工具：

- 从映射文件到 Java 源代码的生成器(也就是 CodeGenerator，hbm2java）
- 从已编译的 Java 类或者带有 XDoclet 标记的 Java 源代码生成映射文件（它们是 MapGenerator,class2hbm）

实际上 Hibernate Extensions 里面还有一个工具：ddl2hbm。但是它已经被废弃了，已经不再被维护了。Middlegen 完成了同样的任务，并且更加出色。

对 Hibernate 提供支持的第三方工具有：

- Middlegen（从现有的数据库 schema 中生成映射文件）
- AndroMDA（ 使用 MDA 思想(Model-Driven Architecture ，模型驱动体系)的代码生成器，它从 UML 图和其 XML/XMI 等价形式中生成持久化类的代码）

这些第三方工具没有在这篇指南中说明。请查阅 Hibernate 网站得到关于它们目前的情况。（Hibernate 主发行包中有关于整个网站的快照）

19.1. Schema 生成器（Schema Generation）

可以从你的映射文件使用一个命令行工具生成 DDL。在 Hibernate 主发行包的 hibernate-x.x.x/bin 目录下有一个批处理文件。

生成的 schema 包含有对实体和集合类表的完整性引用约束（主键和外键）。涉及到的标示符生成器所需的表和 sequence 也会同时生成。

在使用这个工具的时候，你*必须* 通过 hibernate.dialet 属性指定一个 SQL 方言(Dialet)。

19.1.1. 对 schema 定制化(Customizing the schema)

很多 Hibernate 映射元素定义了一个可选的 length 属性。你可以通过这个属性设置字段的长度。

有些 tag 接受 not-null 属性（用来在表字段上生成 NOT NULL 约束）和 unique 属性（用来在表字段上生成 UNIQUE 约束）。

有些 tag 接受 index 属性，用来指定字段的 index 名字。unique-key 属性可以对成组的字段指定一个组合键约束(unit key constraint)。目前，unique-key 属性指定的值*并不会*被当作这个约束的名字，它们只是在用来在映射文件内部用作区分的。

示例：

```
<property name="foo" type="string" length="64" not-null="true"/>

<many-to-one name="bar" foreign-key="fk_foo_bar" not-null="true"/>

<element column="serial_number" type="long" not-null="true"
unique="true"/>
```

另外，这些元素还接受<column>子元素。在定义跨越多字段的类型时特别有用。

```
<property name="foo" type="string">
    <column name="foo" length="64" not-null="true" sql-type="text"/>
</property>

<property name="bar" type="my.customtypes.MultiColumnType"/>
    <column name="fee" not-null="true" index="bar_idx"/>
    <column name="fi" not-null="true" index="bar_idx"/>
    <column name="fo" not-null="true" index="bar_idx"/>
</property>
```

sql-type 属性允许用户覆盖默认的 Hibernate 类型到 SQL 数据类型的映射。

check 属性允许用户指定一个约束检查。

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>

<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

Table 19.1. Summary

属性(Attribute)	值（Values）	解释（Interpretation）
length	true|false	字段长度
not-null	true|false	指明字段是否应该是非空的
unique	true|false	指明是否该字段具有惟一约束

属性(Attribute)	值（Values）	解释（Interpretation）
index	index_name	指明一个（多字段）的索引(index)的名字
unique-key	unique_key_name	指明多字段惟一约束的名字（参见上面的说明）
foreign-key	foreign_key_name	指明一个外键的名字，它是为关联生成的。
sql-type	column_type	覆盖默认的字段类型(只能用于<column>属性）
check	SQL 表达式	对字段或表加入 SQL 约束检查

19.1.2. 运行该工具

SchemaExport 工具把 DDL 脚本写到标准输出，同时/或者执行 DDL 语句。

java -cp *hibernate_classpaths* net.sf.hibernate.tool.hbm2ddl.SchemaExport *options mapping_files*

Table 19.2. **SchemaExport** 命令行选项

选项	说明
--quiet	不要把脚本输出到 stdout
--drop	只进行 drop tables 的步骤
--text	不执行在数据库中运行的步骤
--output=my_schema.ddl	把输出的 ddl 脚本输出到一个文件
--config=hibernate.cfg.xml	从 XML 文件读入 Hibernate 配置
--properties=hibernate.properties	从文件读入数据库属性
--format	把脚本中的 SQL 语句对齐和美化
--delimiter=x	为脚本设置行结束符

你甚至可以在你的应用程序中嵌入 SchemaExport 工具:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

19.1.3. 属性(Properties)

可以通过如下方式指定数据库属性:

- 通过-D*<property>*系统参数
- 在 hibernate.properties 文件中
- 位于一个其它名字的 properties 文件中,然后用 --properties 参数指定

所需的参数包括:

Table 19.3. SchemaExport 连接属性

属性名	说明
hibernate.connection.driver_class	jdbc driver class
hibernate.connection.url	jdbc url
hibernate.connection.username	database user
hibernate.connection.password	user password
hibernate.dialect	方言(dialect)

19.1.4. 使用 Ant(Using Ant)

你可以在你的 Ant build 脚本中调用 SchemaExport:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

19.1.5. 对 schema 的增量更新(Incremental schema updates)

SchemaUpdate 工具对已存在的 schema 采用"增量"方式进行更新。注意 SchemaUpdate 严重依赖于 JDBC metadata API,所以它并非对所有 JDBC 驱动都有效。

java -cp *hibernate_classpaths* net.sf.hibernate.tool.hbm2ddl.SchemaUpdate *options mapping_files*

Table 19.4. **SchemaUpdate** 命令行选项

选项	说明
--quiet	不要把脚本输出到 stdout
--properties=hibernate.properties	从指定文件读入数据库属性

你可以在你的应用程序中嵌入 SchemaUpdate 工具:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

19.1.6. 用 Ant 来增量更新 schema(Using Ant for incremental schema updates)

你可以在 Ant 脚本中调用 SchemaUpdate:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

19.2. 代码生成（Code Generation）

Hibernate 代码生成器可以用来为 Hibernate 映射文件生成 Java 实现类的骨架。这个工具在 Hibernate Extensions 发行包中提供（需要单独下载）。

hbm2java 解析映射文件，生成可工作的 Java 源代码文件。使用 hbm2java，你可以“只”提供.hbm 文件，不用担心要去手工编写 Java 文件。

java -cp *hibernate_classpaths* net.sf.hibernate.tool.hbm2java.CodeGenerator *options mapping_files*

Table 19.5. 代码生成器命令行选项

选项	说明
--output=*output_dir*	生成代码输出的根目录
--config=*config_file*	可选的 hvm2java 配置文件

19.2.1. 配置文件(可选)

配置文件提供了配置生成源代码的多个"渲染器(renders)"的途径,也可以声明在全局范围生效的<meta>属性。详情请参见<meta>属性的部分。

```
<codegen>
    <meta attribute="implements">codegen.test.IAuditable</meta>
    <generate
renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate
        package="autofinders.only"
        suffix="Finder"
        renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

这个配置文件声明了一个全局的 meta(元）属性“implements”，指定了两个渲染器，默认渲染器(BadicRender)和生成 Finder（参见下面的“基本 Finder 生成器”）的渲染器。

定义第二个渲染器需要一个包名和后缀属性。

包名属性指定生成后的源代码应该保存的位置，覆盖在.hbm 文件中指定的包范围。

后缀属性指定生成的文件的后缀。比如说，如果有一个 Foo.java 文件，应该变成 FooFinder.java。

19.2.2. meta 属性

<meta>标签时对 hbm.xml 文件进行的简单注解，工具可以用这个位置来保存/阅读和 Hibernate 内核不是直接相关的一些信息。

你可以用<meta>标签来告诉 hbm2java 只生成"protectd"

下面的例子：

```
<class name="Person">
    <meta attribute="class-description">
        Javadoc for the Person class
        @author Frodo
    </meta>
    <meta attribute="implements">IAuditable</meta>
    <id name="id" type="long">
        <meta attribute="scope-set">protected</meta>
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <meta attribute="field-description">The name of the
person</meta>
    </property>
</class>
```

会生成类似下面的输出（为了有助于理解，节选部分代码）。注意 Javadoc 注释和声明成 protected 的 set 方法：

```
// default package

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/**
 *         Javadoc for the Person class
 *         @author Frodo
 *
 */
public class Person implements Serializable, IAuditable {

    /** identifier field */
    public Long id;

    /** nullable persistent field */
    public String name;

    /** full constructor */
    public Person(java.lang.String name) {
        this.name = name;
```

```
    }

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

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
        this.id = id;
    }

    /**
     * The name of the person
     */
    public java.lang.String getName() {
        return this.name;
    }

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

}
```

Table 19.6. 支持的 meta 标签

属性	说明
class-description	插入到类的 javadoc 说明去
field-description	插入到 field/property 的 javadoc 说明去
interface	如果是 true，生成 interface 而非 class
implements	类要实现的接口
extends	类要继承的超类(若是 subclass，则忽略该属性)
generated-class	重新指定要生成的类名
scope-class	class 的 scope
scope-set	set 方法的 scope
scope-get	get 方法的 scope
scope-field	实际属性字段(field)的 scope
use-in-tostring	在 toString()中包含此属性
bound	为属性增加 propertyChangeListener 支持
constrained	为属性增加 vetoChangeListener 支持
gen-property	如果是 false，不会生成属性(谨慎使用)

属性	说明
property-type	覆盖属性的默认值.如果值是标签,则指定一个具体的类型而非 Object(Use this with any tag's to specify the concrete type instead of just Object.)
finder-method	参见下面的"Basic finder generator"
session-method	参见下面的"Basic finder generator"

通过<meta>标签定义的属性在一个 hbm.xml 文件中是默认"继承"的。

这究竟是什么意思？如果你希望你所有的类都实现 IAuditable 接口，那么你只需要加一个 <meta attribute="implements">IAuditable</meta> 在你 hml.xml 文件的开头，就在 <hibernate-mapping>后面。现在所有在 hbm.xml 文件中定义的类都会实现 IAuditable 了！（除了那些也特别指定了"implements"元属性的类，因为本地指定的元标签总是会覆盖任何继承的元标签）。

注意，这条规则对*所有* 的<meta>标签都有效。也就是说它可以用来指定所有的字段都被声明成 protected 的，而非默认的 private。这可以通过在<class>后面<meta attribute="scope-field">protected</meta>指定，那么这个类所有的 field 都会变成 protected。

如果你不想让<meta>标签继承，你可以简单的在标签属性上指明 inherit="false"，比如 <meta attribute="scope-class" inherit="false">public abstract</meta>，这样"class-scope"就只会对当前类起作用，不会对其子类生效。

19.2.3. 基本的 finder 生成器（Basic finder generator）

目前可以让 hbm2java 为 Hibernate 属性生成基本的 finder。这需要在 hbm.xml 文件中做两件事情。

首先是要标记出你希望生成 finder 的字段。你可以通过在 property 标签中的 meta 块来定义：

```
<property name="name" column="name" type="string">
     <meta attribute="finder-method">findByName</meta>
</property>
```

find 方法的名字就是 meta 标签中间的文字。

第二件事是为 hbm2java 建立下面格式的配置文件：

```
<codegen>
    <generate
renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate suffix="Finder"
renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

然后用参数去调用：hbm2java --config=xxx.xml，xxx.xml 就是你刚才创建的配置文件的名字。

有个可选的参数,作为一个在 class 级别的 meta 标签，格式如下：

```
<meta attribute="session-method">
    com.whatever.SessionTable.getSessionTable().getSession();
</meta>
```

他是用来管理你如何使用 *Thread Local Session* 模式(在 Hibernate 网站的 Design Patterns 部分有文档）得到 session 的。

19.2.4. 基于 Velocity 的渲染器/生成器(Velocity based renderer/generator)

目前可以使用 velocity 作为渲染机制的一个替代方案。下面的 config.xml 文件显示了如果配置 hbm2java 来使用 velocity 渲染器。

```

    <codegen>
     <generate
renderer="net.sf.hibernate.tool.hbm2java.VelocityRenderer">
      <param name="template">pojo.vm</param>
     </generate>
    </codegen>
```

名为 template 的参数是指向你希望你使用 velocity macro 文件的资源路径。这个文件必须在 hbm2java 的 classpath 中。所以要记住把 pojo.vm 所在的路径加入到你 ant 任务或者 shell 脚本中去。（默认的位置是./tools/src/velocity）

注意，当前的 pojo.vm 只生成 java beans 最基本的部分。他还没有默认的渲染器那么完整，也没有那么多功能——特别是大部分 meta 标签还不支持。

19.3. 映射文件生成器（Mapping File Generation）

映射文件的骨架可以从编译过的持久化类中使用 MapGenerator 工具生成。这工具是 Hibernate Extensions 发行包的一部分。

Hibernate 映射生成器提供了从编译过的类中产生映射的机制。他使用 Java 反射来查找*属性（ properties)*，然后使用启发式算法来从属性类型猜测合适的映射。生成出来的映射文件之应该看作是后续工作的起点。没有办法在没有用户修正的情况下生成完整的 Hibernate 映射。但是，这个工具还是替你做了很多非常琐碎和麻烦的工作。

类一个一个地加入到映射去。如果工具认为某个类不是 *Hibernate 可持久化（ persistable）*的，就会把这些类剔除。

判断是否是 *Hibernate 可持久化（ persistable）*的原则是：

- 必定不是一个原始类型
- 必定不是一个数组
- 必定不是一个接口
- 必定不是一个内部类
- 必定有一个默认的无参数的构造方法。

注意，接口和内部类实际上是可以通过 Hibernate 持久化的，但是一般来说用户不会使用。

对已经发现的类，MapGenerator 会重复回溯到超类链条上去，以尽可能的把 Hibernate 可持久化的超类加入到对同一个数据库表的映射去。如果回溯过程中某个类出现了有个属性在下列*备选 UID 名字（candidate UID names）*名单中，回溯就会停止。

默认的备选 UID 属性名有：uid, UID, id, ID, key, KEY, pk, PK。

如果类中有两个方法，一个是 setter,一个是 getter，并且 setter 的单参数的属性和 getter 的无参数返回值得类型相同，并且 setter 返回 void,就认为发现了一个属性。并且，setter 的名字必须以 set 字符串开始，getter 的名字必须以 get 开始，或者以 is 开始并且属性类型是 boolean。在上面的情况发生时，get 和 set 之后的名字还必须匹配。这个匹配就是属性的名字，然后如果第二个字母是小写的话，会把其首字母变成小写。

用来决定每个属性的数据库类型的规则如下：

- 如果 Java 类型是 Hibernate.basic()，则属性是该类型的一个普通字段。
- 对于 hibernate.type.Type 特定类型和 PersistentEnum 来说，也会使用一个普通字段。
- 如果属性类型是一个数组，那么会使用一个 Hibernate 数组,并且 MapGenerator 试图反映数组元素的类型。（attempts to reflect on the array element type.）
- 如果属性是 java.util.List,java.util.Map 或者 java.util.Set，会使用对应的 Hibernate 类型，但是 MapGenerator 不能对这些类型进行进一步处理了。
- 如果属性的类型不是上面任何一种，MapGeneraotr 把决定数据库类型的步骤留待所有的类都被处理之后再来做。在那时候，如果类在上面描述过的超类搜索过程中被发现了，这个属性会被认为是一个 many-to-one 的关联。如果类有人和属性，它则是一个组件(component)。否则它就是可序列化的（serializable），或者不是可持久化的。

19.3.1. 运行此工具

这个工具会把 XML 映射写入到标准输出或者/并且到一个文件中去。

在调用这个工具的时候，你必须把你编译过的类放到 classpath 中去。

java -cp *hibernate_and_your_class_classpaths*
net.sf.hibernate.tool.class2hbm.MapGenerator *options and classnames*

有两种操作模式：命令行或者交互式。

交互式模式当你使用一个惟一的命令行参数--interact 的时候启动。这个模式提供一个命令控制台。你可以用 uid=XXX 命令设置每个类的 UID 属性的名字，xxx 就是 UID 属性名。其他可用的命令就是类名的全限定名，或者“done”命令用来输出 XML,并且结束。

在命令行模式下，下面的参数选项和所需处理的类的全限定名可以相互间隔使用。大多数选项会使用多次，每个只影响其后出现的类。

Table 19.7. MapGenerator 命令行选项

选项	说明
--quiet	不把 O-R 映射输出到 stdout
--setUID=uid	设置备选 UID 名单
--addUID=uid	在备选 UID 名单前面增加一个新的 uid

选项	说明
--select=*mode*	对后面的 classes 使用 select 选择的*模式(mode)*(比如, *distinct* 或者 *all*)
--depth=<small-int>	限制后面的类的组件数据递归层数
--output=my_mapping.xml	把 O-R 映射输出到一个文件
full.class.Name	把这个类加入到映射中
--abstract=*full.class.Name*	参见下面的说明

abstract 开关指定本工具忽略特定的超类,所以它的继承数上的类不会被映射到一个大表中去。比如，我们来看下面的类继承树：

Animal-->Mammal-->Human

Animal-->Mammal-->Marsupial-->Kangaroo

如果*不*使用--abstract 开关，Animal 的所有子类都会被放到一个巨大的表中去，包含所有类的所有属性，还有一个用于分辨子类的字段。如果 Mammal 被标记成 abstract，Human 和 Marsupial 会被映射到不同的<class>声明，并且会有各自单独的表。Kangaroo 仍然会被认为是 Marsupial 的子类，除非 Marsupial 也标记为 anstract 的。

Chapter 20. 最佳实践(Best Practices)

设计细颗粒度的持久类并且使用<component>来实现映射。

使用一个 Address 持久类来封装 street, suburb, state, postcode. 这将有利于代码重用和简化代码重构(refactoring)的工作。

对持久类声明标识符属性。

Hibernate 中标识符属性是可选的，不过有很多原因来说明你应该使用标识符属性。我们建议标识符应该是“人造”的(自动生成，不涉及业务含义)，并且不是基本类型。为了最大的灵活性，应该使用 java.lang.Long or java.lang.String

为每个持久类写一个映射文件

不要把所有的持久类映射都写到一个大文件中。把 com.eg.Foo 映射到 com/eg/Foo.hbm.xml 中， 在团队开发环境中，这一点显得特别有意义。

把映射文件作为资源加载

把映射文件和他们的映射类放在一起进行部署。

考虑把查询字符串放在程序外面

如果你的查询中调用了非 ANSI 标准的 SQL 函数，那么这条实践经验对你适用。把查询字符串放在程序外面可以让程序具有更好的可移植性。

使用绑定变量

就像在 JDBC 编程中一样，应该总是用占位符"?"来替换非常量值，不要在查询中用字符串值来构造非常量值！更好的办法是在查询中使用命名参数。

不要自己来管理 JDBC connections

Hibernate 允许应用程序自己来管理 JDBC connections，但是应该作为最后没有办法的办法。如果你不能使用 Hibernate 内建的 connections providers，那么考虑实现自己来实现 net.sf.hibernate.connection.ConnectionProvider

考虑使用用户自定义类型(custom type)

net.sf.hibernate.UserType. This approach frees the application code from implementing transformations to / from a Hibernate type. 假设你有一个 Java 类型，来自某些类库，需要被持久化，但是该类没有提供映射操作需要的存取方法。那么你应该考虑实现 net.sf.hibernate.UserType 接口。这种办法使程序代码写起来更加自如，不再需要考虑类与 Hibernate type 之间的相互转换。

在性能瓶颈的地方使用硬编码的 JDBC

在对性能要求很严格的一些系统中，一些操作(例如批量更新和批量删除)也许直接使用 JDBC 会更好，但是请先*搞清楚*这是否是一个瓶颈，并且不要想当然认为 JDBC 一定会更快。如果确实需要直接使用 JDBC，那么最好打开一个 Hibernate Session 然后从 Session 获得 connection，按照这种办法你仍然可以使用同样的 transaction 策略和底层的 connection provider。

理解 Session 清洗（ flushing）

Session 会不时的向数据库同步持久化状态，如果这种操作进行的过于频繁，那么性能会受到一定的影响。有时候你可以通过禁止自动 flushing 尽量最小化非必要的 flushing 操作，或者更进一步，在一个特殊 transaction 中改变查询和其它操作的顺序。

在三层架构中，考虑使用 saveOrUpdate()

当使用一个 servlet / session bean 的架构的时候，你可以把已加载的持久对象在 session bean 层和 servlet / JSP 层之间来回传递。使用新的 session 来为每个请求服务，使用 Session.update() 或者 Session.saveOrUpdate()来更新对象的持久状态。

在两层架构中，考虑使用 session disconnection.

当仅仅使用 servlet 的时候，你可以在多个客户请求中复用同一个 session，只是要记得在把控制权交还给客户端之前 disconnect 掉 session。

不要把异常看成可恢复的

这一点甚至比"最佳实践"还要重要，这是"必备常识"。当异常发生的时候，回滚 Transaction ，关闭 Session。如果你不这样做的话，Hibernate 无法保证内存状态精确的反应持久状态。尤其不要使用 Session.load()来判断一个给定标识符的对象实例在数据库中是否存在，应该使用 find()。

对于关联优先考虑 lazy fetching

谨慎的使用主动外连接抓取(eager (outer-join) fetching)。对于大多数没有 JVM 级别缓存的持久对象的关联，应该使用代理(proxies)或者具有延迟加载属性的集合(lazy collections)。对于被缓存的对象的关联，尤其是缓存的命中率非常高的情况下，应该使用 `outer-join="false"`，显式的禁止掉 eager fetching。如果那些特殊的确实适合使用 outer-join fetch 的场合，请在查询中使用 `left join`。

考虑把 Hibernate 代码从业务逻辑代码中抽象出来

把 Hibernate 的数据存取代码隐藏到接口(interface)的后面，组合使用 *DAO* 和 *Thread Local Session* 模式。通过 Hibernate 的 `UserType`，你甚至可以用硬编码的 JDBC 来持久化那些本该被 Hibernate 持久化的类。（该建议更适用于规模足够大应用软件中，对于那些只有 5 张表的应用程序并不适合。）