HIBERNATE - Relational Persistence for Idiomatic Java

1

Hibernate Reference Documentation

3.6.0.Beta2

由 Gavin King、Christian Bauer、Max Rydahl Andersen、Emmanuel Bernard、Steve Ebersole和Hardy Ferentschik

and thanks to James Cobb (Graphic Design)、Cheyenne Weaver (Graphic Design)和Cao RedSaga Xiaogang

前言

Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming. Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases. Hibernate is an Object/Relational Mapping solution for Java environments. The term Object/Relational Mapping refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema; see http://en.wikipedia.org/wiki/Object-relational_mapping for a discussion.

注意

While having a strong background in SQL is not required to use Hibernate, having a basic understanding of the concepts can greatly help you understand Hibernate more fully and quickly. Probably the single best background is an understanding of data modeling principles. You might want to consider these resources as a good starting point:

- http://www.agiledata.org/essays/dataModeling101.html
- http://en.wikipedia.org/wiki/Data_modeling

Hibernate 不仅管理 Java 类到数据库表的映射（包括 Java 数据类型到 SQL 数据类型的映射），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对人工使用 SQL 和 JDBC 处理数据的时间。

Hibernate's design goal is to relieve the developer from 95% of common data persistence-related programming tasks by eliminating the need for manual, hand-crafted data processing using SQL and JDBC. However, unlike many other persistence solutions, Hibernate does not hide the power of SQL from you and guarantees that your investment in relational technology and knowledge is as valid as always.

Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

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

1. Read 第 1 章 教程 for a tutorial with step-by-step instructions. The source code for the tutorial is included in the distribution in the doc/reference/tutorial/ directory.

2. Read 第 2 章 体系结构（Architecture） to understand the environments where Hibernate can be used.

3. 查看 Hibernate 发行包中的 eg/ 目录，里面有个一简单的独立运行的程序。把你的 JDBC 驱动复制到 lib/ 目录并修改一下 etc/hibernate.properties，指定数据库的信息。然后进入命令行，切换到发行包的目录，输入 ant eg（使用 Ant），或者在 Windows 系统下使用 build eg。

4. Use this reference documentation as your primary source of information. Consider reading [JPwH] if you need more help with application design, or if you prefer a step-by-step tutorial. Also visit http://caveatemptor.hibernate.org and download the example application from [JPwH].

5. 在 Hibernate 网站上可以找到问题和解答（FAQ）。

6. 在 Hibernate 网站上还有第三方的演示、示例和教程的链接。

7. Hibernate 网站的社区是讨论关于设计模式以及很多整合方案（Tomcat、JBoss AS、Struts、EJB 等）的好地方。

There are a number of ways to become involved in the Hibernate community, including

- Trying stuff out and reporting bugs. See http://hibernate.org/issuetracker.html details.
- Trying your hand at fixing some bugs or implementing enhancements. Again, see http://hibernate.org/issuetracker.html details.
- http://hibernate.org/community.html list a few ways to engage in the community.
  - There are forums for users to ask questions and receive help from the community.
  - There are also IRC [http://en.wikipedia.org/wiki/Internet_Relay_Chat] channels for both user and developer discussions.
- Helping improve or translate this documentation. Contact us on the developer mailing list if you have interest.
- Evangelizing Hibernate within your organization.

教程

面向新用户，从一个简单的使用内存数据库的例子开始，本章提供对 Hibernate 的逐步介绍。本教程基于 Michael Gloegl 早期编写的手册。所有代码都包含在 tutorials/web 目录下。

重要

本教程期望用户具备 Java 和 SQL 知识。如果你这方面的知识有限，我们建议你在学习 Hibernate 之前先好好了解这些技术。

注意

本版本在源代码目录 tutorial/eg 下还包含另外一个例程。

1.1. 第一部分 – 第一个 Hibernate 应用程序

在这个例子里，我们将设立一个小应用程序可以保存我们希望参加的活动（events）和这些活动主办方的相关信息。（译者注：在本教程的后面部分，我们将直接使用 event 而不是它的中文翻译“活动”，以免混淆。）

注意

虽然你可以使用任何数据库，我们还是用 HSQLDB [http://hsqldb.org/]（一个用 Java 编写的内存数据库）来避免花费篇章对数据库服务器的安装/配置进行解释。

1.1.1. 设置

我们需要做的第一件事情是设置开发环境。我们将使用许多构建工具如 Maven [http://maven.org] 所鼓吹的“标准格式”。特别是 Maven，它的资源对这个格式（layout） [http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html]有着很好的描述。因为本教程使用的是 web 应用程序，我么将创建和使用 src/main/java、src/main/resources 和 src/main/webapp 目录。

在本教程里我们将使用 Maven，利用其 transitive dependency 管理以及根据 Maven 描述符用 IDE 自动设置项目的能力。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">

    <modelVersion
```

```
>4.0.0</modelVersion>

    <groupId
>org.hibernate.tutorials</groupId>
    <artifactId
>hibernate-tutorial</artifactId>
    <version
>1.0.0-SNAPSHOT</version>
    <name
>First Hibernate Tutorial</name>

    <build>
         <!-- we dont want the version to be part of the generated war file name -->
         <finalName
>${artifactId}</finalName>
    </build>

    <dependencies>
        <dependency>
            <groupId
>org.hibernate</groupId>
            <artifactId
>hibernate-core</artifactId>
        </dependency>

        <!-- Because this is a web app, we also have a dependency on the servlet api. -->
        <dependency>
            <groupId
>javax.servlet</groupId>
            <artifactId
>servlet-api</artifactId>
        </dependency>

        <!-- Hibernate uses slf4j for logging, for our purposes here use the simple backend -->
        <dependency>
            <groupId
>org.slf4j</groupId>
            <artifactId
>slf4j-simple</artifactId>
        </dependency>

        <!-- Hibernate gives you a choice of bytecode providers between cglib and javassist -->
        <dependency>
            <groupId
>javassist</groupId>
            <artifactId
>javassist</artifactId>
        </dependency>
    </dependencies>

</project
>
```

提示

It is not a requirement to use Maven. If you wish to use something else to build this tutorial (such as Ant), the layout will remain the same. The only change is that you will need to manually account for all the needed dependencies. If you use something like Ivy [http://ant.apache.org/ivy/] providing transitive dependency management you would still use the dependencies mentioned below. Otherwise, you'd need to grab all dependencies, both explicit and transitive, and add them to the project's classpath. If working from the Hibernate distribution bundle, this would mean hibernate3.jar, all artifacts in the lib/required directory and all files from either the lib/bytecode/cglib or lib/bytecode/javassist directory; additionally you will need both the servlet-api jar and one of the slf4j logging backends.

把这个文件保存为项目根目录下的 pom.xml。

1.1.2. 第一个 class

接下来我们创建一个类，用来代表那些我们希望储存在数据库里的 event，这是一个具有一些属性的简单 JavaBean 类：

```
package org.hibernate.tutorial.domain;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

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

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }
```

```
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

你可以看到这个类对属性的存取方法（getter and setter method）使用了标准 JavaBean 命名约定，同时把类属性（field）的访问级别设成私有的（private）。这是推荐的设计，但并不是必须的。Hibernate 也可以直接访问这些 field，而使用访问方法（accessor method）的好处是提供了重构时的健壮性（robustness）。

对一特定的 event，id 属性持有唯一的标识符（identifier）的值。如果我们希望使用 Hibernate 提供的所有特性，那么所有的持久化实体（persistent entity）类（这里也包括一些次要依赖类）都需要一个这样的标识符属性。而事实上，大多数应用程序（特别是 web 应用程序）都需要通过标识符来区别对象，所以你应该考虑使用标识符属性而不是把它当作一种限制。然而，我们通常不会操作对象的标识（identity），因此它的 setter 方法的访问级别应该声明 private。这样当对象被保存的时候，只有 Hibernate 可以为它分配标识符值。你可看到Hibernate可以直接访问 public，private 和 protected 的访问方法和 field。所以选择哪种方式完全取决于你，你可以使你的选择与你的应用程序设计相吻合。

所有的持久化类（persistent classes）都要求有无参的构造器，因为 Hibernate 必须使用 Java 反射机制来为你创建对象。构造器（constructor）的访问级别可以是 private，然而当生成运行时代理（runtime proxy）的时候则要求使用至少是 package 级别的访问控制，这样在没有字节码指令（bytecode instrumentation）的情况下，从持久化类里获取数据会更有效率。

把这个文件保存到 src/main/java/org/hibernate/tutorial/domain 目录下。

1.1.3. 映射文件

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

一个映射文件的基本结构看起来像这样:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.hibernate.tutorial.domain">
[...]
</hibernate-mapping
>
```

注意 Hibernate 的 DTD 是非常复杂的。你的编辑器或者 IDE 里使用它来自动完成那些用来映射的 XML 元素（element）和属性（attribute）。你也可以在文本编辑器里打开 DTD — 这是最简单的方式来概览所有的元素和 attribute，并查看它们的缺省值以及注释。注意 Hibernate 不会从 web 加载 DTD 文件，但它会首先在应用程序的 classpath 中查找。DTD 文件已包括在 hibernate3.jar 里，同时也在 Hibernate 发布包的 src/ 目录下。

重要

为缩短代码长度，在以后的例子里我们会省略 DTD 的声明。当然，在实际的应用程序中，DTD 声明是必需的。

在 hibernate-mapping 标签（tag）之间，含有一个 class 元素。所有的持久化实体类（再次声明，或许接下来会有依赖类，就是那些次要的实体）都需要一个这样的映射，来把类对象映射到 SQL 数据库里的表：

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">

    </class>

</hibernate-mapping
>
```

到目前为止，我们告诉了 Hibernate 怎样把 Events 类的对象持久化到数据库的 EVENTS 表里，以及怎样从 EVENTS 表加载到 Events 类的对象。每个实例对应着数据库表中的一行。现在我们将继续讨论有关唯一标识符属性到数据库表的映射。另外，由于我们不关心怎样处理这个标识符，我们就配置由 Hibernate 的标识符生成策略来产生代理主键字段：

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping
>
```

id 元素是对 identifier 属性的声明。name="id" 映射属性声明了 JavaBean 属性的名称并告诉 Hibernate 使用 getId() 和 setId() 方法来访问这个属性。column 属性告诉 Hibernate EVENTS 表的哪个字段持有主键值。

嵌套的 generator 元素指定标识符的生成策略（也就是标识符值是怎么产生的）。在这个例子里，我们选择 native，它提供了取决于数据库方言的可移植性。Hibernate 数据库生成的、全局性唯一的以及应用程序分配的标识符。标识符值的生成也是 Hibernate 的扩展功能之一，你可以插入自己的策略。

提示

native is no longer consider the best strategy in terms of portability. for further discussion, see 第 28.4 节 “标识符的生成”

最后我们在映射文件里面包含需要持久化属性的声明。默认情况下，类里面的属性都被视为非持久化的：

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping
>
```

和 id 元素一样，property 元素的 name 属性告诉 Hibernate 使用哪个 getter 和 setter 方法。在此例中，Hibernate 会寻找 getDate()、setDate()、getTitle() 和 setTitle() 方法。

注意

为什么 date 属性的映射含有 column attribute，而 title 却没有？当没有设定 column attribute 的时候，Hibernate 缺省地使用 JavaBean 的属性名作为字段名。对于 title，这样工作得很好。然而，date 在多数的数据库里，是一个保留关键字，所以我们最好把它映射成一个不同的名字。

另一有趣的事情是 title 属性缺少一个 type attribute。我们在映射文件里声明并使用的类型，却不是我们期望的那样，是 Java 数据类型，同时也不是 SQL 数据库的数据类型。这些类型就是所谓的 Hibernate 映射类型（mapping types），它们能把 Java 数据类型转换到 SQL 数据类型，反之亦然。再次重申，如果在映射文件中没有设置 type 属性的话，Hibernate 会自己试着去确定正确的转换类型和它的映射类型。在某些情况下这个自动检测机制（在 Java 类上使用反射机制）不会产生你所期待或需要的缺省值。date 属性就是个很好的例子，Hibernate 无法知道这个属性（java.util.Date 类型的）应该被映射成：SQL date，或 timestamp，还是 time 字段。在此例中，把这个属性映射成 timestamp 转换器，这样我们预留了日期和时间的全部信息。

提示

当处理映射文件时，Hibernate 用反射（reflection）来决定这个映射类型。这需要时间和资源，所以如果你注重启动性能，你应该考虑显性地定义所用的类型。

把这个映射文件保存为 src/main/resources/org/hibernate/tutorial/domain/Event.hbm.xml。

1.1.4. Hibernate 配置

此时，你应该有了持久化类和它的映射文件。现在是配置 Hibernate 的时候了。首先让我们设立 HSQLDB 使其运行在“服务器模式”。

注意

数据在程序运行期间需要保持有效。

在开发的根目录下创建一个 data 目录 — 这是 HSQL DB 存储数据文件的地方。此时在 data 目录中运行 java -classpath ../lib/hsqldb.jar org.hsqldb.Server 就可启动数据库。你可以在 log 中看到它的启动，及绑定到 TCP/IP 套接字，这正是我们的应用程序稍后会连接的地方。如果你希望在本例中运行一个全新的数据库，就在窗口中按下 CTRL + C 来关闭 HSQL 数据库，并删除 data/ 目录下的所有文件，再重新启动 HSQL 数据库。

Hibernate 将为你的应用程序连接到数据库，所以它需要知道如何获取连接。在这个教程里，我们使用一个独立连接池（和 javax.sql.DataSource 相反）。Hibernate 支持两个第三方的开源 JDBC 连接池：c3p0 [https://sourceforge.net/projects/c3p0] 和 proxool [http://proxool.sourceforge.net/]。然而，在本教程里我们将使用 Hibernate 内置的连接池。

小心

嵌入的 Hibernate 连接池不用于产品环境。它缺乏连接池里的几个功能。

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

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
```

```
        <!-- Database connection settings -->
        <property name="connection.driver_class"
>org.hsqldb.jdbcDriver</property>
        <property name="connection.url"
>jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username"
>sa</property>
        <property name="connection.password"
></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size"
>1</property>

        <!-- SQL dialect -->
        <property name="dialect"
>org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class"
>thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class"
>org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql"
>true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto"
>update</property>

        <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration
>
```

注意

请注意，这个配置文件指定了一个不同的 DTD。

注意这个 XML 配置使用了一个不同的 DTD。在这里，我们配置了 Hibernate 的SessionFactory — 一个关联于特定数据库全局的工厂（factory）。如果你要使用多个数据库，就要用多个的 <session-factory>，通常把它们放在多个配置文件中（为了更容易启动）。

签名 4 个 property 元素包含了 JDBC 连接所必需的配置。方言 property 元素指定了 Hibernate 生成的特定 SQL 语句。

提示

In most cases, Hibernate is able to properly determine which dialect to use. See 第 28.3 节 “方言的使用” for more information.

最开始的 4 个 property 元素包含必要的 JDBC 连接信息。方言 (dialect) 的 property 元素指明 Hibernate 生成的特定 SQL 变量。你很快会看到，Hibernate 对持久化上下文的自动 session 管理就会派上用场。 打开 hbm2ddl.auto 选项将自动生成数据库模式 (schema) — 直接加入数据库中。当然这个选项也可以被关闭 (通过去除这个配置选项) 或者通过 Ant 任务 SchemaExport 的帮助来把数据库 schema 重定向到文件中。最后，在配置中为持久化类加入映射文件。

把这个文件保存为 src/main/resources 目录下的 hibernate.cfg.xml。

1.1.5. 用 Maven 构建

我们将用 Maven 构建这个教程。你将需要安装 Maven；你可以从 Maven 下载页面 [http://maven.apache.org/download.html]获得 Maven。Maen 将读取我们先前创建的 /pom.xml 并知道执行基本的项目任务。首先，让我们运行 compile 目标来确保我们可以编译到目前为止的所有程序：

```
[hibernateTutorial]$ mvn compile
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building First Hibernate Tutorial
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /home/steve/projects/sandbox/hibernateTutorial/target/classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jun 09 12:25:25 CDT 2009
[INFO] Final Memory: 5M/547M
[INFO] ------------------------------------------------------------------------
```

1.1.6. 启动和辅助类

是时候来加载和储存一些 Event 对象了，但首先我们得编写一些基础的代码以完成设置。我们必须启动 Hibernate，此过程包括创建一个全局的 SessoinFactory，并把它储存在应用程序代码容易访问的地方。SessionFactory 可以创建并打开新的 Session。一个 Session 代表一个单线程的单元操作，org.hibernate.SessionFactory 则是个线程安全的全局对象，只需要被实例化一次。

我们将创建一个 HibernateUtil 辅助类 (helper class) 来负责启动 Hibernate 和更方便地操作 org.hibernate.SessionFactory。让我们来看一下它的实现：

```
package org.hibernate.tutorial.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory = buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

把这段代码保存为 src/main/java/org/hibernate/tutorial/util/HibernateUtil.java。

这个类不但在它的静态初始化过程（仅当加载这个类的时候被 JVM 执行一次）中产生全局的 org.hibernate.SessionFactory，而且隐藏了它使用了静态 singleton 的事实。它也可能在应用程序服务器中的 JNDI 查找 org.hibernate.SessionFactory。

如果你在配置文件中给 org.hibernate.SessionFactory 一个名字，在 它创建后，Hibernate 会试着把它绑定到 JNDI。要完全避免这样的代码，你也可以使用 JMX 部署，让具有 JMX 能力的容器来实例化 HibernateService 并把它绑定到 JNDI。这些高级可选项在后面的章节中会讨论到。

再次编译这个应用程序应该不会有问题。最后我们需要配置一个日志（logging）系统 — Hibernate 使用通用日志接口，允许你在 Log4j 和 JDK 1.4 日志之间进行选择。多数开发者更喜欢 Log4j：从 Hibernate 的发布包中（它在 etc/ 目录下）拷贝 log4j.properties 到你的 src 目录，与 hibernate.cfg.xml 放在一起。看一下配置示例，如果你希望看到更加详细的输出信息，你可以修改配置。默认情况下，只有 Hibernate 的启动信息才会显示在标准输出上。

示例的基本框架完成了 — 现在我们可以用 Hibernate 来做些真正的工作。

1.1.7. 加载并存储对象

We are now ready to start doing some real work with Hibernate. Let's start by writing an EventManager class with a main() method:

```
package org.hibernate.tutorial;
```

```
import org.hibernate.Session;

import java.util.*;

import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);
        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

在 createAndStoreEvent() 来里我们创建了一个新的 Event 对象并把它传递给 Hibernate。现在 Hibernate 负责与 SQL 打交道，并把 INSERT 命令传给数据库。

A org.hibernate.Session is designed to represent a single unit of work (a single atomic piece of work to be performed). For now we will keep things simple and assume a one-to-one granularity between a Hibernate org.hibernate.Session and a database transaction. To shield our code from the actual underlying transaction system we use the Hibernate org.hibernate.Transaction API. In this particular case we are using JDBC-based transactional semantics, but it could also run with JTA.

sessionFactory.getCurrentSession() 是干什么的呢？首先，只要你持有 org.hibernate.SessionFactory，大可在任何时候、任何地点调用这个方法。getCurrentSession() 方法总会返回“当前的”工作单元。记得我们在 src/main/resources/hibernate.cfg.xml 中把这一配置选项调整为 "thread" 了吗？因此，因此，当前工作单元被绑定到当前执行我们应用程序的 Java 线程。但是，这并非是完全准确的，你还得考虑工作单元的生命周期范围（scope），它何时开始，又何时结束。

重要

Hibernate 提供三种跟踪当前会话的方法。基于“线程”的方法不适合于产品环境，它仅用于 prototyping 和教学用途。后面将更详细地讨论会话跟踪。

org.hibernate.Session 在第一次被使用的时候，即第一次调用 getCurrentSession() 的时候,其生命周期就开始。然后它被 Hibernate 绑定到当前线程。当事务结束的时候，不管是提交还是回滚，Hibernate 会自动把 org.hibernate.Session 从当前线程剥离，并且关闭它。假若你再次调用 getCurrentSession()，你会得到一个新的 org.hibernate.Session，并且开始一个新的工作单元。

和工作单元的生命周期这个话题相关，Hibernate org.hibernate.Session 是否被应该用来执行多次数据库操作? 上面的例子对每一次操作使用了一个 org.hibernate.Session，这完全是巧合，这个例子不是很复杂，无法展示其他方式。Hibernate org.hibernate.Session 的生命周期可以很灵活，但是你绝不要把你的应用程序设计成为每一次数据库操作都用一个新的 Hibernate org.hibernate.Session。因此就算下面的例子（它们都很简单）中你可以看到这种用法，记住每次操作一个 session 是一个反模式。在本教程的后面会展示一个真正的（web）程序。

See 第 13 章 事务和并发 for more information about transaction handling and demarcation. The previous example also skipped any error handling and rollback.

要运行它，我们将使用 Maven exec 插件以及必要的 classpath 设置来进行调用：mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="store"。

注意

你可能需要先执行 mvn compile。

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

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

执行 HQL INSERT 语句的例子如下：

我们想要列出所有已经被存储的 events，就要增加一个条件分支选项到 main 方法中：

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

```
                System.out.println(
                        "Event: " + theEvent.getTitle() + " Time: " + theEvent.getDate()
                );
            }
        }
```

我们也增加一个新的 listEvents() 方法:

```
    private List listEvents() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        List result = session.createQuery("from Event").list();
        session.getTransaction().commit();
        return result;
    }
```

Here, we are using a Hibernate Query Language (HQL) query to load all existing Event objects from the database. Hibernate will generate the appropriate SQL, send it to the database and populate Event objects with the data. You can create more complex queries with HQL. See 第 16 章 HQL: Hibernate 查询语言 for more information.

现在我们可以再次用 Maven exec plugin - mvn exec:java -Dexec.mainClass="org.hibernate.tutorial.EventManager" -Dexec.args="list" 调用新的功能了。

1.2. 第二部分 — 关联映射

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

1.2.1. 映射 Person 类

最初简单的 Person 类:

```
package org.hibernate.tutorial.domain;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

把它保存为文件 src/main/java/org/hibernate/tutorial/domain/Person.java。

然后，创建新的映射文件 src/main/resources/org/hibernate/tutorial/domain/Person.hbm.xml。

```
<hibernate-mapping package="org.hibernate.tutorial.domain">

    <class name="Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

最后，把新的映射加入到 Hibernate 的配置中：

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

现在我们在这两个实体之间创建一个关联。显然，persons 可以参与一系列 events，而 events 也有不同的参加者（persons）。我们需要处理的设计问题是关联方向（directionality），阶数（multiplicity）和集合（collection）的行为。

1.2.2. 单向 Set-based 的关联

我们将向 Person 类增加一连串的 events。那样，通过调用 aPerson.getEvents()，就可以轻松地导航到特定 person 所参与的 events，而不用去执行一个显式的查询。我们使用 Java 的集合类（collection）：Set，因为 set 不包含重复的元素及与我们无关的排序。

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

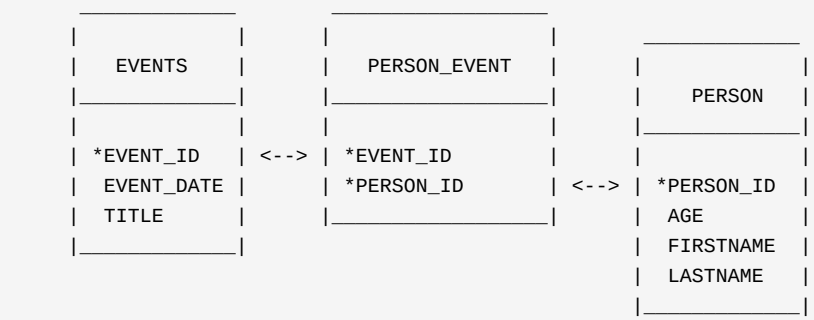
在映射这个关联之前，先考虑一下此关联的另外一端。很显然，我们可以保持这个关联是单向的。或者，我们可以在 Event 里创建另外一个集合，如果希望能够双向地导航，如：anEvent.getParticipants()。从功能的角度来说，这并不是必须的。因为你总可以显式地执行

一个查询，以获得某个特定 event 的所有参与者。这是个在设计时需要做出的选择，完全由你来决定，但此讨论中关于关联的阶数是清楚的：即两端都是“多”值的，我们把它叫做多对多（many-to-many）关联。因而，我们使用 Hibernate 的多对多映射：

```
<class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="Event"/>
    </set>

</class>
```

Hibernate 支持各种各样的集合映射，<set> 使用的最为普遍。对于多对多关联（或叫 n:m 实体关系），需要一个关联表（association table）。表里面的每一行代表从 person 到 event 的一个关联。表名是由 set 元素的 table 属性配置的。关联里面的标识符字段名，对于 person 的一端，是由 <key> 元素定义，而 event 一端的字段名是由 <many-to-many> 元素的 column 属性定义。你也必须告诉 Hibernate 集合中对象的类（也就是位于这个集合所代表的关联另外一端的类）。

因而这个映射的数据库 schema 是：



1.2.3. 使关联工作

我们把一些 people 和 events 一起放到 EventManager 的新方法中：

```
    private void addPersonToEvent(Long personId, Long eventId) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Person aPerson = (Person) session.load(Person.class, personId);
```

```
        Event anEvent = (Event) session.load(Event.class, eventId);
        aPerson.getEvents().add(anEvent);

        session.getTransaction().commit();
    }
```

在加载一 Person 和 Event 后，使用普通的集合方法就可容易地修改我们定义的集合。如你所见，没有显式的 update() 或 save()，Hibernate 会自动检测到集合已经被修改并需要更新回数据库。这叫做自动脏检查（automatic dirty checking），你也可以尝试修改任何对象的 name 或者 date 属性，只要他们处于持久化状态，也就是被绑定到某个 Hibernate 的 Session 上（如：他们刚刚在一个单元操作被加载或者保存），Hibernate 监视任何改变并在后台隐式写的方式执行 SQL。同步内存状态和数据库的过程，通常只在单元操作结束的时候发生，称此过程为清理缓存（flushing）。在我们的代码中，工作单元由数据库事务的提交（或者回滚）来结束——这是由 CurrentSessionContext 类的 thread 配置选项定义的。

当然，你也可以在不同的单元操作里面加载 person 和 event。或在 Session 以外修改不是处在持久化（persistent）状态下的对象（如果该对象以前曾经被持久化，那么我们称这个状态为脱管（detached））。你甚至可以在一个集合被脱管时修改它：

```
    private void addPersonToEvent(Long personId, Long eventId) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        Person aPerson = (Person) session
               .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
                .setParameter("pid", personId)
                .uniqueResult(); // Eager fetch the collection so we can use it detached
        Event anEvent = (Event) session.load(Event.class, eventId);

        session.getTransaction().commit();

        // End of first unit of work

        aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

        // Begin second unit of work

        Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
        session2.beginTransaction();
        session2.update(aPerson); // Reattachment of aPerson

        session2.getTransaction().commit();
    }
```

对 update 的调用使一个脱管对象重新持久化，你可以说它被绑定到一个新的单元操作上，所以在脱管状态下对它所做的任何修改都会被保存到数据库里。这也包括你对这个实体对象的集合所作的任何改动（增加/删除）。

这对我们当前的情形不是很有用，但它是非常重要的概念，你可以把它融入到你自己的应用程序设计中。在EventManager的 main 方法中添加一个新的动作，并从命令行运行它来完成我们所做的

练习。如果你需要 person 及 event 的标识符 — 那就用 save() 方法返回它（你可能需要修改前面的一些方法来返回那个标识符）：

```
        else if (args[0].equals("addpersontoevent")) {
            Long eventId = mgr.createAndStoreEvent("My Event", new Date());
            Long personId = mgr.createAndStorePerson("Foo", "Bar");
            mgr.addPersonToEvent(personId, eventId);
            System.out.println("Added person " + personId + " to event " + eventId);
        }
```

上面是个关于两个同等重要的实体类间关联的例子。像前面所提到的那样，在特定的模型中也存在其它的类和类型，这些类和类型通常是“次要的”。你已看到过其中的一些，像 int 或 String。我们称这些类为值类型（value type），它们的实例依赖（depend）在某个特定的实体上。这些类型的实例没有它们自己的标识（identity），也不能在实体间被共享（比如，两个 person 不能引用同一个 firstname 对象，即使他们有相同的 first name）。当然，值类型并不仅仅在 JDK 中存在（事实上，在一个 Hibernate 应用程序中，所有的 JDK 类都被视为值类型），而且你也可以编写你自己的依赖类，例如 Address，MonetaryAmount。

你也可以设计一个值类型的集合，这在概念上与引用其它实体的集合有很大的不同，但是在 Java 里面看起来几乎是一样的。

1.2.4. 值类型的集合

让我们在 Person 实体里添加一个电子邮件的集合。这将以 java.lang.String 实例的 java.util.Set 出现：

```
    private Set emailAddresses = new HashSet();

    public Set getEmailAddresses() {
        return emailAddresses;
    }

    public void setEmailAddresses(Set emailAddresses) {
        this.emailAddresses = emailAddresses;
    }
```

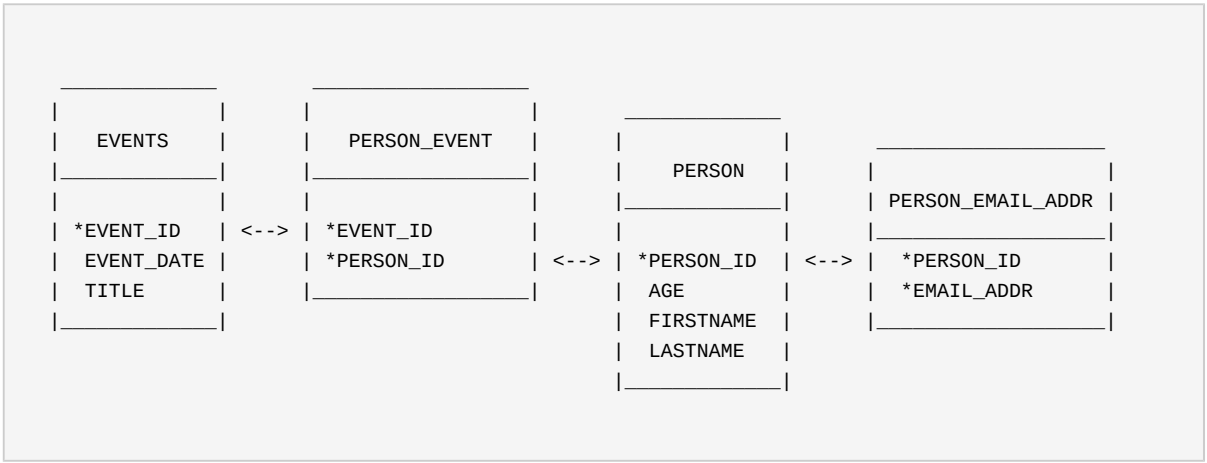
这个 Set 的映射如下：

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

比较这次和此前映射的差别，主要在于 element 部分，这次并没有包含对其它实体引用的集合，而是元素类型为 String 的集合（在映射中使用小写的名字”string“是向你表明它是一个 Hibernate 的映射类型或者类型转换器）。和之前一样，set 元素的 table 属性决定了用于集合

的表名。key 元素定义了在集合表中外键的字段名。element 元素的 column 属性定义用于实际保存 String 值的字段名。

看一下修改后的数据库 schema。



你可以看到集合表的主键实际上是个复合主键，同时使用了两个字段。这也暗示了对于同一个 person 不能有重复的 email 地址，这正是 Java 里面使用 Set 时候所需要的语义（Set 里元素不能重复）。

你现在可以试着把元素加入到这个集合，就像我们在之前关联 person 和 event 的那样。其实现的 Java 代码是相同的：

```
private void addEmailToPerson(Long personId, String emailAddress) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    // adding to the emailAddress collection might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

这次我们没有使用 fetch 查询来初始化集合。因此，调用其 getter 方法会触发另一附加的 select 来初始化集合，这样我们才能把元素添加进去。检查 SQL log，试着通过预先抓取来优化它。

1.2.5. 双向关联

接下来我们将映射双向关联（bi-directional association）— 在 Java 里让 person 和 event 可以从关联的任何一端访问另一端。当然，数据库 schema 没有改变，我们仍然需要多对多的阶数。一个关系型数据库要比网络编程语言更加灵活，所以它并不需要任何像导航方向（navigation direction）的东西 — 数据可以用任何可能的方式进行查看和获取。

注意

关系型数据库比网络编程语言更为灵活，因为它不需要方向导航，其数据可以用任何可能的方式进行查看和提取。

首先，把一个参与者（person）的集合加入 Event 类中：

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

在 Event.hbm.xml 里面也映射这个关联。

```
    <set name="participants" table="PERSON_EVENT" inverse="true">
        <key column="EVENT_ID"/>
        <many-to-many column="PERSON_ID" class="events.Person"/>
    </set>
```

如你所见，两个映射文件里都有普通的 set 映射。注意在两个映射文件中，互换了 key 和 many-to-many 的字段名。这里最重要的是 Event 映射文件里增加了 set 元素的 inverse="true" 属性。

这意味着在需要的时候，Hibernate 能在关联的另一端 — Person 类得到两个实体间关联的信息。这将会极大地帮助你理解双向关联是如何在两个实体间被创建的。

1.2.6. 使双向连起来

首先请记住，Hibernate 并不影响通常的 Java 语义。 在单向关联的例子中，我们是怎样在 Person 和 Event 之间创建联系的？我们把 Event 实例添加到 Person 实例内的 event 引用集合里。因此很显然，如果我们要让这个关联可以双向地工作，我们需要在另外一端做同样的事情 — 把 Person 实例加入 Event 类内的 Person 引用集合。这“在关联的两端设置联系”是完全必要的而且你都得这么做。

许多开发人员防御式地编程，创建管理关联的方法来保证正确的设置了关联的两端，比如在 Person 里：

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
```

```
        this.events = events;
    }

    public void addToEvent(Event event) {
        this.getEvents().add(event);
        event.getParticipants().add(this);
    }

    public void removeFromEvent(Event event) {
        this.getEvents().remove(event);
        event.getParticipants().remove(this);
    }
```

注意现在对于集合的 get 和 set 方法的访问级别是 protected 而不是 private — 这允许在位于同一个包（package）中的类以及继承自这个类的子类可以访问这些方法，但禁止其他任何人的直接访问，避免了集合内容的混乱。你应尽可能地在另一端也把集合的访问级别设成 protected。

inverse 映射属性究竟表示什么呢？对于你和 Java 来说，一个双向关联仅仅是在两端简单地正确设置引用。然而，Hibernate 并没有足够的信息去正确地执行 INSERT 和 UPDATE 语句（以避免违反数据库约束），所以它需要一些帮助来正确的处理双向关联。把关联的一端设置为 inverse 将告诉 Hibernate 忽略关联的这一端，把这端看成是另外一端的一个镜象（mirror）。这就是所需的全部信息，Hibernate 利用这些信息来处理把一个有向导航模型转移到数据库 schema 时的所有问题。你只需要记住这个直观的规则：所有的双向关联需要有一端被设置为 inverse。在一对多关联中它必须是代表多（many）的那端。而在多对多（many-to-many）关联中，你可以任意选取一端，因为两端之间并没有差别。

1.3. 第三部分 - EventManager web 应用程序

Hibernate web 应用程序使用 Session 和 Transaction 的方式几乎和独立应用程序是一样的。但是，有一些常见的模式（pattern）非常有用。现在我们编写一个 EventManagerServlet。这个 servlet 可以列出数据库中保存的所有的 events，还提供一个 HTML 表单来增加新的 events。

1.3.1. 编写基本的 servlet

这个 servlet 只处理 HTTP GET 请求，因此，我们要实现的是 doGet() 方法：

```
package org.hibernate.tutorial.web;

// Imports

public class EventManagerServlet extends HttpServlet {

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

        SimpleDateFormat dateFormatter = new SimpleDateFormat( "dd.MM.yyyy" );

        try {
            // Begin unit of work
```

```
            HibernateUtil.getSessionFactory().getCurrentSession().beginTransaction();

            // Process request and render page...

            // End unit of work
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().commit();
        }
        catch (Exception ex) {
            HibernateUtil.getSessionFactory().getCurrentSession().getTransaction().rollback();
            if ( ServletException.class.isInstance( ex ) ) {
                throw ( ServletException ) ex;
            }
            else {
                throw new ServletException( ex );
            }
        }
    }

}
```

把这个 servlet 保存为 src/main/java/org/hibernate/tutorial/web/EventManagerServlet.java。

我们称这里应用的模式为每次请求一个 session(session-per-request)。当有请求到达这个 servlet 的时候，通过对 SessionFactory 的第一次调用，打开一个新的 Hibernate Session。然后启动一个数据库事务 — 所有的数据访问都是在事务中进行，不管是读还是写（我们在应用程序中不使用 auto-commit 模式）。

不要为每次数据库操作都使用一个新的 Hibernate Session。将 Hibernate Session 的范围设置为整个请求。要用 getCurrentSession()，这样它自动会绑定到当前 Java 线程。

下一步，对请求的可能动作进行处理，渲染出反馈的 HTML。我们很快就会涉及到那部分。

最后，当处理与渲染都结束的时候，这个工作单元就结束了。假若在处理或渲染的时候有任何错误发生，会抛出一个异常，回滚数据库事务。这样，session-per-request 模式就完成了。为了避免在每个 servlet 中都编写事务边界界定的代码，可以考虑写一个 servlet 过滤器（filter）来更好地解决。关于这一模式的更多信息，请参阅 Hibernate 网站和 Wiki，这一模式叫做 Open Session in View — 只要你考虑用JSP来渲染你的视图（view），而不是在servlet中，你就会很快用到它。

1.3.2. 处理与渲染

我们来实现处理请求以及渲染页面的工作。

```
        // Write HTML header
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Event Manager</title></head><body>");

        // Handle actions
        if ( "store".equals(request.getParameter("action")) ) {

            String eventTitle = request.getParameter("eventTitle");
            String eventDate = request.getParameter("eventDate");
```

```
            if ( "".equals(eventTitle) || "".equals(eventDate) ) {
                out.println("<b><i>Please enter event title and date.</i></b>");
            }
            else {
                createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
                out.println("<b><i>Added event.</i></b>");
            }
        }

        // Print page
        printEventForm(out);
        listEvents(out, dateFormatter);

        // Write HTML footer
        out.println("</body></html>");
        out.flush();
        out.close();
```

必须承认，这种编码风格把 Java 和 HTML 混在一起，在更复杂的应用程序里不应该大量使用 — 记住，在本章里我们仅仅是展示了 Hibernate 的基本概念。这段代码打印出了 HTML 页眉和页脚，在这个页面里，还打印了一个输入 events 条目的表单单并列出了数据库里的有的 events。第一个方法微不足道，仅仅是输出 HTML:

```
    private void printEventForm(PrintWriter out) {
        out.println("<h2>Add new event:</h2>");
        out.println("<form>");
        out.println("Title: <input name='eventTitle' length='50'/><br/>");
        out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
        out.println("<input type='submit' name='action' value='store'/>");
        out.println("</form>");
    }
```

listEvents() 方法使用绑定到当前线程的 Hibernate Session 来执行查询:

```
    private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

        List result = HibernateUtil.getSessionFactory()
                .getCurrentSession().createCriteria(Event.class).list();
        if (result.size() > 0) {
            out.println("<h2>Events in database:</h2>");
            out.println("<table border='1'>");
            out.println("<tr>");
            out.println("<th>Event title</th>");
            out.println("<th>Event date</th>");
            out.println("</tr>");
            Iterator it = result.iterator();
            while (it.hasNext()) {
                Event event = (Event) it.next();
                out.println("<tr>");
                out.println("<td>" + event.getTitle() + "</td>");
                out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
                out.println("</tr>");
            }
```

```
            out.println("</table>");
        }
    }
```

最后，store 动作会被导向到 createAndStoreEvent() 方法，它也使用当前线程的 Session：

```
    protected void createAndStoreEvent(String title, Date theDate) {
        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        HibernateUtil.getSessionFactory()
                .getCurrentSession().save(theEvent);
    }
```

大功告成，这个 servlet 写完了。Hibernate 会在单一的 Session 和 Transaction 中处理到达的 servlet 请求。如同在前面的独立应用程序中那样，Hibernate 可以自动的把这些对象绑定到当前运行的线程中。这给了你用任何你喜欢的方式来对代码分层及访问 SessionFactory 的自由。通常，你会用更加完备的设计，把数据访问代码转移到数据访问对象中（DAO 模式）。请参见 Hibernate Wiki，那里有更多的例子。

1.3.3. 部署与测试

要部署这个应用程序以进行测试，我们必须出具一个 Web ARchive（WAR）。首先我们必须定义 WAR 描述符为 src/main/webapp/WEB-INF/web.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>org.hibernate.tutorial.web.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

在你的开发目录中，调用 ant war 来构建、打包，然后把 hibernate-tutorial.war 文件拷贝到你的 tomcat 的 webapps 目录下。假若你还没安装 Tomcat，就去下载一个，按照指南来安装。对此应用的发布，你不需要修改任何 Tomcat 的配置。

注意

If you do not have Tomcat installed, download it from http://tomcat.apache.org/ and follow the installation instructions. Our application requires no changes to the standard Tomcat configuration.

在部署完，启动 Tomcat 之后，通过 http://localhost:8080/hibernate-tutorial/eventmanager 进行访问你的应用，在第一次 servlet 请求发生时，请在 Tomcat log 中确认你看到 Hibernate 被初始化了（HibernateUtil 的静态初始化器被调用），假若有任何异常抛出，也可以看到详细的输出。

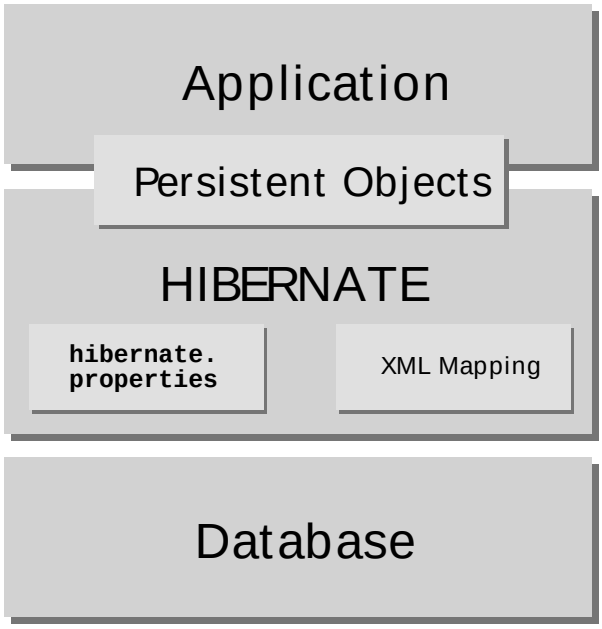
1.4. 总结

本章覆盖了如何编写一个简单独立的 Hibernate 命令行应用程序及小型的 Hibernate web 应用程序的基本要素。更多的教程可以在 website [http://hibernate.org] 上找到。

体系结构 (Architecture)

2.1. 概况 (Overview)

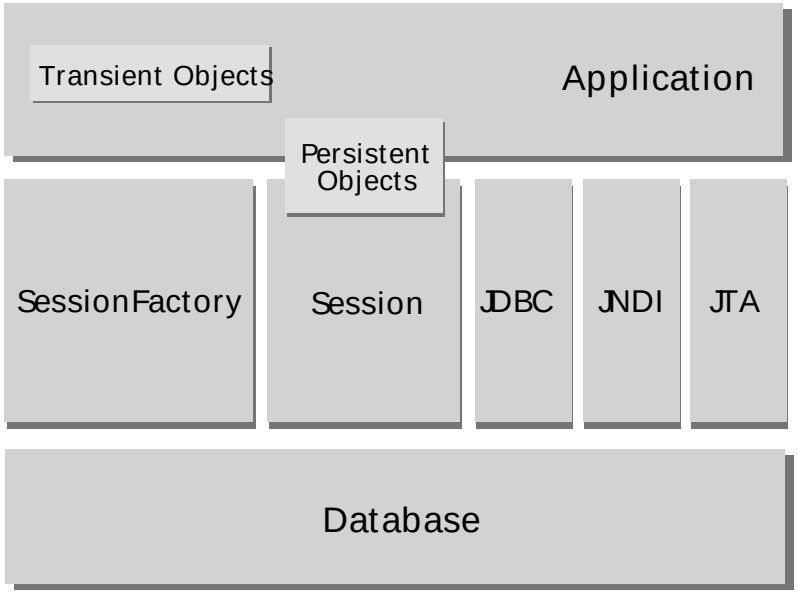
下面的图表提供了 Hibernate 体系结构的高层视图:



Unfortunately we cannot provide a detailed view of all possible runtime architectures. Hibernate is sufficiently flexible to be used in a number of ways in many, many architectures. We will, however, illustrate 2 specifically since they are extremes.

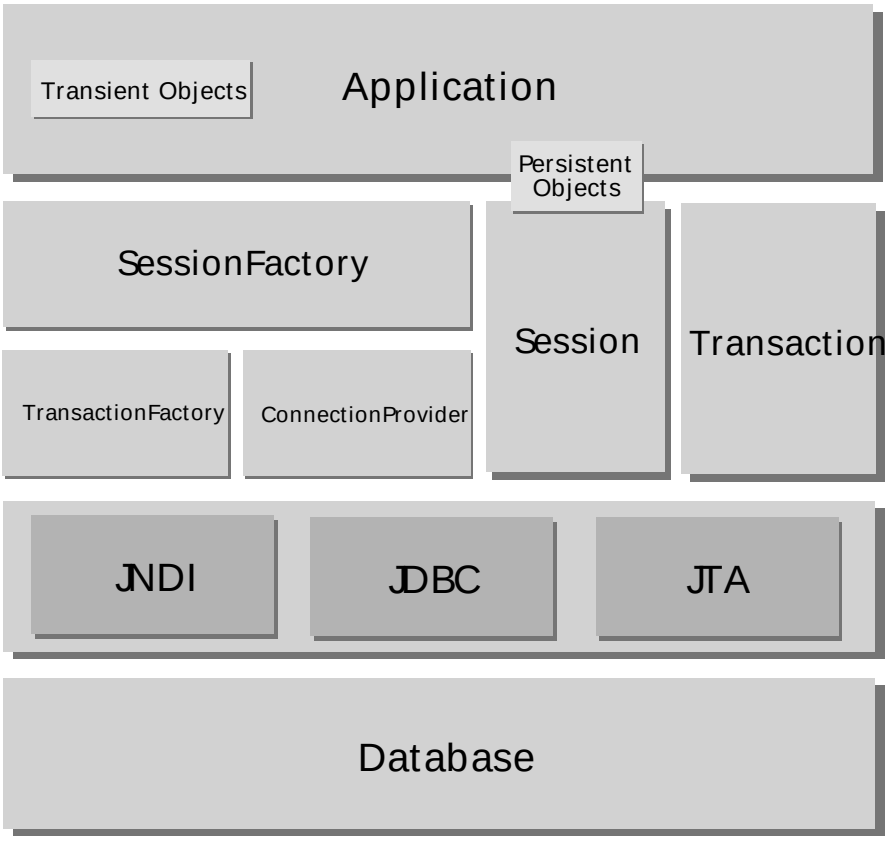
2.1.1. Minimal architecture

The "minimal" architecture has the application manage its own JDBC connections and provide those connections to Hibernate; additionally the application manages transactions for itself. This approach uses a minimal subset of Hibernate APIs.



2.1.2. Comprehensive architecture

“全面解决”的体系结构方案，将应用层从底层的 JDBC/JTA API 中抽象出来，而让 Hibernate 来处理这些细节。



2.1.3. Basic APIs

Here are quick discussions about some of the API objects depicted in the preceding diagrams (you will see them again in more detail in later chapters).

SessionFactory (org.hibernate.SessionFactory)
: A thread-safe, immutable cache of compiled mappings for a single database. A factory for org.hibernate.Session instances. A client of org.hibernate.connection.ConnectionProvider. Optionally maintains a second level cache of data that is reusable between transactions at a process or cluster level.

Session (org.hibernate.Session)
: A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC java.sql.Connection. Factory for org.hibernate.Transaction. Maintains a first level cache of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

持久的对象及其集合
: Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one org.hibernate.Session. Once the org.hibernate.Session is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation). 第 11 章 与对象共事 discusses transient, persistent and detached object states.

瞬态 (transient) 和脱管 (detached) 的对象及其集合
: Instances of persistent classes that are not currently associated with a org.hibernate.Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed org.hibernate.Session. 第 11 章 与对象共事 discusses transient, persistent and detached object states.

Transaction (org.hibernate.Transaction)
: (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A org.hibernate.Session might span several org.hibernate.Transactions in some cases. However, transaction demarcation, either using the underlying API or org.hibernate.Transaction, is never optional.

ConnectionProvider (org.hibernate.connection.ConnectionProvider)
: (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying javax.sql.DataSource or java.sql.DriverManager. It is not exposed to application, but it can be extended and/or implemented by the developer.

TransactionFactory (org.hibernate.TransactionFactory)
: (Optional) A factory for org.hibernate.Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

Extension Interfaces

Hibernate 提供了很多可选的扩展接口，你可以通过实现它们来定制你的持久层的行为。具体请参考 API 文档。

2.2. JMX 整合

JMX 是管理 Java 组件的 J2EE 标准。Hibernate 可以通过一个 JMX 标准服务来管理。在这个发行版本中，我们提供了一个 MBean 接口的实现,即 org.hibernate.jmx.HibernateService。

Another feature available as a JMX service is runtime Hibernate statistics. See 第 3.4.6 节 “Hibernate 的统计（statistics）机制” for more information.

2.3. 上下文相关的会话（Contextual Session）

使用 Hibernate 的大多数应用程序需要某种形式的“上下文相关的”会话，特定的会话在整个特定的上下文范围内始终有效。然而，对不同类型的应用程序而言，要为什么是组成这种“上下文”下一个定义通常是困难的；不同的上下文对“当前”这个概念定义了不同的范围。在 3.0 版本之前，使用 Hibernate 的程序要么采用自行编写的基于 ThreadLocal 的上下文会话，要么采用 HibernateUtil 这样的辅助类，要么采用第三方框架（比如 Spring 或 Pico），它们提供了基于代理（proxy）或者基于拦截器（interception）的上下文相关的会话。

从 3.0.1 版本开始，Hibernate 增加了 SessionFactory.getCurrentSession() 方法。一开始，它假定了采用 JTA 事务，JTA 事务定义了当前 session 的范围和上下文（scope 和 context）。因为有好几个独立的 JTA TransactionManager 实现稳定可用，不论是否被部署到一个 J2EE 容器中，大多数（假若不是所有的）应用程序都应该采用 JTA 事务管理。基于这一点，采用 JTA 的上下文相关的会话可以满足你一切需要。

更好的是，从 3.1 开始，SessionFactory.getCurrentSession() 的后台实现是可拔插的。因此，我们引入了新的扩展接口（org.hibernate.context.CurrentSessionContext）和新的配置参数（hibernate.current_session_context_class），以便对什么是当前会话的范围（scope）和上下文（context）的定义进行拔插。

请参阅 org.hibernate.context.CurrentSessionContext 接口的 Javadoc，那里有关于它的契约的详细讨论。它定义了单一的方法，currentSession()，特定的实现用它来负责跟踪当前的上下文相关的会话。Hibernate 内置了此接口的三种实现：

- org.hibernate.context.JTASessionContext：当前会话根据 JTA 来跟踪和界定。这和以前的仅支持 JTA 的方法是完全一样的。详情请参阅 Javadoc。
- org.hibernate.context.ThreadLocalSessionContext：当前会话通过当前执行的线程来跟踪和界定。详情也请参阅 Javadoc。
- org.hibernate.context.ManagedSessionContext：当前会话通过当前执行的线程来跟踪和界定。但是，你需要负责使用这个类的静态方法将 Session 实例绑定、或者取消绑定，它并不会打开（open）、flush 或者关闭（close）任何 Session。

The first two implementations provide a "one session - one database transaction" programming model. This is also known and used as session-per-request. The beginning

and end of a Hibernate session is defined by the duration of a database transaction. If you use programmatic transaction demarcation in plain JSE without JTA, you are advised to use the Hibernate Transaction API to hide the underlying transaction system from your code. If you use JTA, you can utilize the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you do not need any transaction or session demarcation operations in your code. Refer to 第 13 章 事务和并发 for more information and code examples.

hibernate.current_session_context_class 配置参数定义了应该采用哪个 org.hibernate.context.CurrentSessionContext 实现。注意，为了向下兼容，如果未配置此参数，但是存在 org.hibernate.transaction.TransactionManagerLookup 的配置，Hibernate 会采用org.hibernate.context.JTASessionContext。一般而言，此参数的值指明了要使用的实现类的全名，但那三种内置的实现可以使用简写，即 "jta"、"thread" 和 "managed"。

配置

由于 Hibernate 是为了能在各种不同环境下工作而设计的，因此存在着大量的配置参数。幸运的是多数配置参数都有比较直观的默认值，并有随 Hibernate 一同分发的配置样例 hibernate.properties（位于 etc/）来展示各种配置选项。所需做的仅仅是将这个样例文件复制到类路径（classpath）下并进行定制。

3.1. 可编程的配置方式

org.hibernate.cfg.Configuration 实例代表了一个应用程序中 Java 类型到SQL数据库映射的完整集合。org.hibernate.cfg.Configuration 被用来构建一个（不可变的（immutable））org.hibernate.SessionFactory。映射定义则由不同的 XML 映射定义文件编译而来。

你可以直接实例化 org.hibernate.cfg.Configuration 来获取一个实例，并为它指定 XML 映射定义文件。如果映射定义文件在类路径（classpath）中，请使用 addResource ()。例如：

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

一个替代方法（有时是更好的选择）是，指定被映射的类，让 Hibernate 帮你寻找映射定义文件：

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate 将会在类路径（classpath）中寻找名字为 /org/hibernate/auction/Item.hbm.xml 和 /org/hibernate/auction/Bid.hbm.xml 映射定义文件。这种方式消除了任何对文件名的硬编码（hardcoded）。

org.hibernate.cfg.Configuration > 也允许你指定配置属性。例如：

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

当然这不是唯一的传递 Hibernate 配置属性的方式，其他可选方式还包括：

1. 传一个 java.util.Properties 实例给 Configuration.setProperties()。

2. 将 hibernate.properties 放置在类路径（classpath）的根目录下（root directory）。
3. 通过 java -Dproperty=value 来设置系统（System）属性。
4. 在 hibernate.cfg.xml 中加入元素 <property>（稍后讨论）。

如果你想快速上路，hibernate.properties 就是最容易的途径。

org.hibernate.cfg.Configuration 实例被设计成启动期间（startup-time）对象，一旦 SessionFactory 创建完成它就被丢弃了。

3.2. 获得 SessionFactory

当所有映射定义被 org.hibernate.cfg.Configuration 解析后，应用程序必须获得一个用于构造 org.hibernate.Session 实例的工厂。这个工厂将被应用程序的所有线程共享：

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

Hibernate 允许你的应用程序创建多个 org.hibernate.SessionFactory 实例。这对 使用多个数据库的应用来说很有用。

3.3. JDBC 连接

通常你希望 org.hibernate.SessionFactory 来为你创建和缓存（pool）JDBC 连接。如果你采用这种方式，只需要如下例所示那样，打开一个 org.hibernate.Session：

```
Session session = sessions.openSession(); // open a new Session
```

一旦你需要进行数据访问时，就会从连接池（connection pool）获得一个 JDBC 连接。

为了使这种方式工作起来，我们需要向 Hibernate 传递一些 JDBC 连接的属性。所有 Hibernate 属性的名字和语义都在 org.hibernate.cfg.Environment 中定义。我们现在将描述 JDBC 连接配置中最重要的设置。

如果你设置如下属性，Hibernate 将使用 java.sql.DriverManager 来获得（和缓存）JDBC 连接：

表 3.1. Hibernate JDBC 属性

属性名	用途
hibernate.connection.driver_class	JDBC driver class
hibernate.connection.url	JDBC URL
hibernate.connection.username	database user
hibernate.connection.password	数据库用户密码
hibernate.connection.pool_size	maximum number of pooled connections

但 Hibernate 自带的连接池算法相当不成熟。它只是为了让你快些上手，并不适合用于产品系统或性能测试中。 出于最佳性能和稳定性考虑你应该使用第三方的连接池。只需要用特定连接池的设置替换 hibernate.connection.pool_size 即可。这将关闭 Hibernate 自带的连接池。例如，你可能会想用 C3P0。

C3P0 是一个随 Hibernate 一同分发的开源的 JDBC 连接池，它位于 lib目录下。 如果你设置了 hibernate.c3p0.* 相关的属性，Hibernate将使用 C3P0ConnectionProvider 来缓存 JDBC 连接。如果你更原意使用 Proxool，请参考发行包中的 hibernate.properties 并到 Hibernate 网站获取更多的信息。

这是一个使用 C3P0 的 hibernate.properties 样例文件：

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

为了能在应用程序服务器（application server）中使用 Hibernate，应当总是将 Hibernate 配置成从注册在 JNDI 中的 Datasource 处获得连接，你至少需要设置下列属性中的一个：

表 3.2. Hibernate 数据源属性

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

这是一个使用应用程序服务器提供的 JNDI 数据源的 hibernate.properties 样例文件：

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

从 JNDI 数据源获得的 JDBC 连接将自动参与到应用程序服务器中容器管理的事务（container-managed transactions）中去。

任何连接（connection）属性的属性名都要以 "hibernate.connnection" 开头。例如，你可能会使用 hibernate.connection.charSet 来指定 charSet 连接属性。

通过实现 org.hibernate.connection.ConnectionProvider 接口，你可以定义属于你自己的获得JDBC连接的插件策略。通过设置hibernate.connection.provider_class，你可以选择一个自定义的实现。

3.4. 可选的配置属性

有大量属性能用来控制 Hibernate 在运行期的行为。它们都是可选的，并拥有适当的默认值。

警告

其中一些属性是"系统级（system-level）的"。系统级属性只能通过java -Dproperty=value 或 hibernate.properties 来设置，而不能用上面描述的其他方法来设置。

表 3.3. Hibernate 配置属性

属性名	用途
hibernate.dialect	允许 Hibernate 针对特定的关系数据库生成优化的 SQL 的 org.hibernate.dialect.Dialect 的类名。 例如：full.classname.of.Dialect 在大多数情况下，Hibernate 可以根据 JDBC 驱动返回的 JDBC metadata 选择正确的 org.hibernate.dialect.Dialect 实现。
hibernate.show_sql	输出所有 SQL 语句到控制台。有一个另外的选择是把 org.hibernate.SQL 这个 log category 设为 debug。 例如：true | false
hibernate.format_sql	在 log 和 console 中打印出更漂亮的 SQL。 例如：true | false
hibernate.default_schema	在生成的 SQL 中，将给定的 schema/tablespace 附加于非全限定名的表名上。 例如：SCHEMA_NAME
hibernate.default_catalog	在生成的 SQL 中，将给定的 catalog 附加于非全限定名的表名上。 例如：CATALOG_NAME

属性名	用途
hibernate.session_factory_name	org.hibernate.SessionFactory 创建后，将自动使用这个名字绑定到 JNDI 中。 例如：jndi/composite/name
hibernate.max_fetch_depth	为单向关联（一对一，多对一）的外连接抓取（outer join fetch）树设置最大深度。值为 0 意味着将关闭默认的外连接抓取。 例如： 建议在 0 到 3 之间取值
hibernate.default_batch_fetch_size	为 Hibernate 关联的批量抓取设置默认数量。 例如：建议的取值为 4，8，和 16
hibernate.default_entity_mode	为由这个 SessionFactory 打开的所有 Session 指定默认的实体表现模式。 取值dynamic-map，dom4j，pojo
hibernate.order_updates	强制 Hibernate 按照被更新数据的主键，为 SQL 更新排序。这么做将减少在高并发系统中事务的死锁。 例如：true | false
hibernate.generate_statistics	如果开启，Hibernate 将收集有助于性能调节的统计数据。 例如：true | false
hibernate.use_identifier_rollback	如果开启，在对象被删除时生成的标识属性将被重设为默认值。 例如：true | false
hibernate.use_sql_comments	如果开启，Hibernate 将在 SQL 中生成有助于调试的注释信息，默认值为 false。 例如：true | false
hibernate.id.new_generator_mappings	Setting is relevant when using @GeneratedValue. It indicates whether or not the new IdentifierGenerator implementations are used for javax.persistence.GenerationType.AUTO, javax.persistence.GenerationType.TABLE and javax.persistence.GenerationType.SEQUENCE. Default to false to keep backward compatibility. 例如：true | false

注意

We recommend all new projects which make use of to use @GeneratedValue to also set hibernate.id.new_generator_mappings=true as the new generators are more efficient and closer to the JPA 2 specification semantic. However they are not backward compatible with existing databases (if a sequence or a table is used for id generation).

表 3.4. Hibernate JDBC 和连接（connection）属性

属性名	用途
hibernate.jdbc.fetch_size	非零值，指定 JDBC 抓取数量的大小（调用 Statement.setFetchSize()）。
hibernate.jdbc.batch_size	非零值，允许 Hibernate 使用 JDBC2 的批量更新。 例如：建议取 5 到 30 之间的值
hibernate.jdbc.batch_versioned_data	Set this property to true if your JDBC driver returns correct row counts from executeBatch(). It is usually safe to turn this option on. Hibernate will then use batched DML for automatically versioned data. Defaults to false. 例如：true | false
hibernate.jdbc.factory_class	选择一个自定义的 Batcher。多数应用程序不需要这个配置属性。 例如：classname.of.Batcher
hibernate.jdbc.use_scrollable_resultset	允许 Hibernate 使用 JDBC2 的可滚动结果集。只有在使用用户提供的 JDBC 连接时，这个选项才是必要的，否则 Hibernate 会使用连接的元数据。 例如：true | false
hibernate.jdbc.use_streams_for_binary	在 JDBC 读写 binary 或 serializable 的类型时使用流（stream）（系统级属性）。 例如：true | false
hibernate.jdbc.use_get_generated_keys	在数据插入数据库之后，允许使用 JDBC3 PreparedStatement.getGeneratedKeys() 来获取数据库生成的 key（键）。需要 JDBC3+ 驱动和 JRE1.4+，如果你的数据库驱动在使用

属性名	用途
	Hibernate 的标识生成器时遇到问题，请将此值设为 false。默认情况下将使用连接的元数据来判定驱动的能力。 例如：true | false
hibernate.connection.provider_class	自定义 ConnectionProvider 的类名，此类用来向 Hibernate 提供 JDBC 连接。 例如：classname.of.ConnectionProvider
hibernate.connection.isolation	设置 JDBC 事务隔离级别。查看 java.sql.Connection 来了解各个值的具体意义，但请注意多数数据库都不支持所有的隔离级别。 例如：1，2，4，8
hibernate.connection.autocommit	允许被缓存的 JDBC 连接开启自动提交（autocommit）（不推荐）。 例如：true | false
hibernate.connection.release_mode	指定 Hibernate 在何时释放 JDBC 连接。默认情况下，直到 Session 被显式关闭或被断开连接时，才会释放 JDBC 连接。对于应用程序服务器的 JTA 数据源，你应当使用 after_statement，这样在每次 JDBC 调用后，都会主动的释放连接。对于非 JTA 的连接，使用 after_transaction 在每个事务结束时释放连接是合理的。auto 将为 JTA 和 CMT 事务策略选择 after_statement，为JDBC事务策略选择 after_transaction。 例如：auto（默认） | on_close | after_transaction | after_statement This setting only affects Sessions returned from SessionFactory.openSession. For Sessions obtained through SessionFactory.getCurrentSession, the CurrentSessionContext implementation configured for use controls the connection release mode for those Sessions. See 第 2.3 节 “上下文相关的会话（Contextual Session）”

属性名	用途
hibernate.connection.<propertyName>	把 JDBC 属性 <propertyName> 传递给 DriverManager.getConnection()。
hibernate.jndi.<propertyName>	把 <propertyName> 属性传递给 JNDI InitialContextFactory。

表 3.5. Hibernate 缓存属性

属性名	用途
hibernate.cache.provider_class	自定义的 CacheProvider 的类名。 例如：classname.of.CacheProvider
hibernate.cache.use_minimal_puts	以频繁的读操作为代价，优化二级缓存来最小化写操作。在 Hibernate3 中，这个设置对的集群缓存非常有用，对集群缓存的实现而言，默认是开启的。 例如：true | false
hibernate.cache.use_query_cache	允许查询缓存，个别查询仍然需要被设置为可缓存的。 例如：true | false
hibernate.cache.use_second_level_cache	能用来完全禁止使用二级缓存。对那些在类的映射定义中指定 <cache> 的类，会默认开启二级缓存。 例如：true | false
hibernate.cache.query_cache_factory	自定义实现 QueryCache 接口的类名，默认为内建的 StandardQueryCache。 例如：classname.of.QueryCache
hibernate.cache.region_prefix	二级缓存区域名的前缀。 例如：prefix
hibernate.cache.use_structured_entries	强制 Hibernate 以更人性化的格式将数据存入二级缓存。 例如：true | false
hibernate.cache.default_cache_concurrency_strategy	Setting used to give the name of the default org.hibernate.annotations.CacheConcurrencyStrategy to use when either @Cacheable or @Cache is used. @Cache(strategy="..") is used to override this default.

表 3.6. Hibernate 事务属性

属性名	用途
hibernate.transaction.factory_class	一个 TransactionFactory 的类名，用于 Hibernate Transaction API（默认为 JDBCTransactionFactory）。 例如：classname.of.TransactionFactory
jta.UserTransaction	一个 JNDI 名字，被 JTATransactionFactory 用来从应用服务器获取 JTA UserTransaction。 例如：jndi/composite/name
hibernate.transaction.manager_lookup_class	一个 TransactionManagerLookup 的类名 — 当使用 JVM 级缓存，或在 JTA 环境中使用 hilo 生成器的时候需要该类。 例如：classname.of.TransactionManagerLookup
hibernate.transaction.flush_before_completion	If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred, see 第 2.3 节 “上下文相关的会话（Contextual Session）”. 例如：true | false
hibernate.transaction.auto_close_session	If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred, see 第 2.3 节 “上下文相关的会话（Contextual Session）”. 例如：true | false

表 3.7. 其他属性

属性名	用途
hibernate.current_session_context_class	Supply a custom strategy for the scoping of the "current" Session. See 第 2.3 节 “上下文相关的会话（Contextual Session）” for more information about the built-in strategies. 例如：jta | thread | managed | custom.Class
hibernate.query.factory_class	选择 HQL 解析器的实现。

属性名	用途
	例 如：org.hibernate.hql.ast.ASTQueryTranslatorFactory 或 org.hibernate.hql.classic.ClassicQueryTranslatorFactory
hibernate.query.substitutions	将 Hibernate 查询中的符号映射到 SQL 查询中的符号（符号可能是函数名或常量名字）。 例如：hql1Literal=SQL_LITERAL, hql1Function=SQLFUNC
hibernate.hbm2ddl.auto	在 SessionFactory 创建时，自动检查数据库结构，或者将数据库 schema 的 DDL 导出到数据库。使用 create-drop 时，在显式关闭 SessionFactory 时，将删除掉数据库 schema。 例如：validate | update | create | create-drop
hibernate.hbm2ddl.import_file	Comma-separated names of the optional files containing SQL DML statements executed during the SessionFactory creation. This is useful for testing or demoing: by adding INSERT statements for example you can populate your database with a minimal set of data when it is deployed. File order matters, the statements of a give file are executed before the statements of the following files. These statements are only executed if the schema is created ie if hibernate.hbm2ddl.auto is set to create or create-drop. e.g. /humans.sql,/dogs.sql
hibernate.bytecode.use_reflection_optimizer	Enables the use of bytecode manipulation instead of runtime reflection. This is a System-level property and cannot be set in hibernate.cfg.xml. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either CGLIB or javassist even if you turn off the optimizer. 例如：true | false

属性名	用途
hibernate.bytecode.provider	Both javassist or cglib can be used as byte manipulation engines; the default is javassist. e.g. javassist | cglib

3.4.1. SQL 方言

你应当总是为你的数据库将 hibernate.dialect 属性设置成正确的 org.hibernate.dialect.Dialect 子类。如果你指定一种方言，Hibernate 将为上面列出的一些属性使用合理的默认值，这样你就不用手工指定它们。

表 3.8. Hibernate SQL 方言 (hibernate.dialect)

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

3.4.2. 外连接抓取（Outer Join Fetching）

如果你的数据库支持 ANSI、Oracle 或 Sybase 风格的外连接，外连接抓取通常能通过限制往返数据库次数（更多的工作交由数据库自己来完成）来提高效率。外连接抓取允许在单个 SELECT SQL 语句中， 通过 many-to-one、one-to-many、many-to-many 和 one-to-one 关联获取连接对象的整个对象图。

将 hibernate.max_fetch_depth 设为 0 能在全局 范围内禁止外连接抓取。设为 1 或更高值能启用 one-to-one 和 many-to-oneouter 关联的外连接抓取，它们通过 fetch="join" 来映射。

See 第 21.1 节 “抓取策略（Fetching strategies）” for more information.

3.4.3. 二进制流（Binary Streams）

Oracle 限制那些通过 JDBC 驱动传输的字节数组的数目。如果你希望使用二进值（binary）或 可序列化的（serializable）类型的大对象，你应该开启 hibernate.jdbc.use_streams_for_binary 属性。这是系统级属性。

3.4.4. 二级缓存与查询缓存

The properties prefixed by hibernate.cache allow you to use a process or cluster scoped second-level cache system with Hibernate. See the 第 21.2 节 “二级缓存（The Second Level Cache）” for more information.

3.4.5. 查询语言中的替换

你可以使用 hibernate.query.substitutions 在 Hibernate 中定义新的查询符号。例如:

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

将导致符号 true 和 false 在生成的 SQL 中被翻译成整数常量。

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

将允许你重命名 SQL 中的 LOWER 函数。

3.4.6. Hibernate 的统计（statistics）机制

如果你开启 hibernate.generate_statistics，那么当你通过 SessionFactory.getStatistics() 调整正在运行的系统时，Hibernate 将导出大量有用的数据。Hibernate 甚至能被配置成通过 JMX 导出这些统计信息。参考 org.hibernate.stats 中接口的 Javadoc，以获得更多信息。

3.5. 日志

Hibernate 利用 Simple Logging Facade for Java [http://www.slf4j.org/] (SLF4J) 来记录不同系统事件的日志。SLF4J 可以根据你选择的绑定把日志输出到几个日志框架

（NOP、Simple、log4j version 1.2、JDK 1.4 logging、JCL 或 logback）上。为了设置日志，你需要在 classpath 里加入 slf4j-api.jar 和你选择的绑定的 JAR 文件（使用 Log4J 时加入 slf4j-log4j12.jar）。更多的细节请参考 SLF4J 文档 [http://www.slf4j.org/manual.html]。要使用 Log4j，你也需要在 classpath 里加入 log4j.properties 文件。Hibernate 里的 src/ 目录里带有一个属性文件的例子。

我们强烈建议你熟悉一下 Hibernate 的日志消息。在不失可读性的前提下，我们做了很多工作，使 Hibernate 的日志可能地详细。这是必要的查错利器。最令人感兴趣的日志分类有如下这些：

表 3.9. Hibernate 日志类别

类别	功能
org.hibernate.SQL	在所有 SQL DML 语句被执行时为它们记录日志
org.hibernate.type	为所有 JDBC 参数记录日志
org.hibernate.tool.hbm2ddl	在所有 SQL DDL 语句执行时为它们记录日志
org.hibernate.pretty	在 session 清洗（flush）时，为所有与其关联的实体（最多 20 个）的状态记录日志
org.hibernate.cache	为所有二级缓存的活动记录日志
org.hibernate.transaction	为事务相关的活动记录日志
org.hibernate.jdbc	为所有 JDBC 资源的获取记录日志
org.hibernate.hql.ast.AST	在解析查询的时候，记录 HQL 和 SQL 的 AST 分析日志
org.hibernate.secure	为 JAAS 认证请求做日志
org.hibernate	为任何 Hibernate 相关信息记录日志（信息量较大，但对查错非常有帮助）

在使用 Hibernate 开发应用程序时，你应当总是为 org.hibernate.SQL 开启 debug 级别的日志记录，或者开启 hibernate.show_sql 属性。

3.6. 实现 NamingStrategy

org.hibernate.cfg.NamingStrategy 接口允许你为数据库中的对象和 schema 元素指定一个“命名标准”。

你可能会提供一些通过 Java 标识生成数据库标识或将映射定义文件中"逻辑"表/列名处理成"物理"表/列名的规则。这个特性有助于减少冗长的映射定义文件，消除重复内容（如 TBL_ 前缀）。Hibernate 使用的缺省策略是相当精简的。

在加入映射定义前，你可以调用 Configuration.setNamingStrategy() 指定一个不同的命名策略：

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

org.hibernate.cfg.ImprovedNamingStrategy 是一个内建的命名策略，对一些应用程序而言，可能是非常有用的起点。

3.7. XML 配置文件

另一个配置方法是在 hibernate.cfg.xml 文件中指定一套完整的配置。这个文件可以当成 hibernate.properties 的替代。若两个文件同时存在，它将覆盖前者的属性。

XML 配置文件被默认是放在 CLASSPATH 的根目录下。下面是一个例子：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

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

        <!-- properties -->
        <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

如你所见，这个方法优势在于，在配置文件中指出了映射定义文件的名字。一旦你需要调整 Hibernate 的缓存，hibernate.cfg.xml 也是更方便。注意，使用 hibernate.properties 还是 hibernate.cfg.xml 完全是由你来决定，除了上面提到的 XML 语法的优势之外，两者是等价的。

使用 XML 配置，使得启动 Hibernate 变的异常简单：

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

你可以使用如下代码来添加一个不同的 XML 配置文件：

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

3.8. J2EE 应用程序服务器的集成

针对 J2EE 体系，Hibernate 有如下几个集成的方面：

- 容器管理的数据源（Container-managed datasources）： Hibernate 能使用通过容器管理，并由 JNDI 提供的 JDBC 连接。通常，特别是当处理多个数据源的分布式事务的时候，由一个 JTA 兼容的 TransactionManager 和一个 ResourceManager 来处理事务管理（CMT，容器管理的事务）。当然你可以通过 编程方式来划分事务边界（BMT，Bean 管理的事务）。或者为了代码的可移植性，你也也许会想使用可选的 Hibernate Transaction API。

- 自动 JNDI 绑定：Hibernate 可以在启动后将 SessionFactory 绑定到 JNDI。

- JTA Session 绑定： Hibernate Session 可以自动绑定到 JTA 事务作用的范围。只需简单地从 JNDI 查找 SessionFactory 并获得当前的 Session。当 JTA 事务完成时，让 Hibernate来处理 Session 的清洗（flush）与关闭。事务的划分可以是声明式的（CMT），也可以是编程式的（BMT/UserTransaction）。

- JMX 部署： 如果你使用支持 JMX 应用程序服务器（如，JBoss AS），那么你可以选择将 Hibernate 部署成托管 MBean。这将为你省去一行从Configuration 构建 SessionFactory 的启动代码。容器将启动你的 HibernateService，并完美地处理好服务间的依赖关系（在 Hibernate 启动前，数据源必须是可用的，等等）。

如果应用程序服务器抛出 "connection containment" 异常，根据你的环境，也许该将配置属性 hibernate.connection.release_mode 设为 after_statement。

3.8.1. 事务策略配置

在你的架构中，Hibernate 的 Session API 是独立于任何事务分界系统的。如果你让 Hibernate 通过连接池直接使用 JDBC，你需要调用 JDBC API 来打开和关闭你的事务。如果你运行在 J2EE 应用程序服务器中，你也许想用 Bean 管理的事务并在需要的时候调用 JTA API 和 UserTransaction。

为了让你的代码在两种（或其他）环境中可以移植，我们建议使用可选的 Hibernate Transaction API，它包装并隐藏了底层系统。你必须通过设置 Hibernate 配置属性 hibernate.transaction.factory_class 来指定一个 Transaction 实例的工厂类。

有三个标准（内建）的选择：

org.hibernate.transaction.JDBCTransactionFactory
: 委托给数据库（JDBC）事务（默认）

org.hibernate.transaction.JTATransactionFactory
: 如果在上下文环境中存在运行着的事务（如，EJB 会话 Bean 的方法），则委托给容器管理的事务。否则，将启动一个新的事务，并使用 Bean 管理的事务。

org.hibernate.transaction.CMTTransactionFactory
: 委托给容器管理的 JTA 事务

你也可以定义属于你自己的事务策略（如，针对 CORBA 的事务服务）。

Hibernate 的一些特性（比如二级缓存，Contextual Sessions with JTA 等等）需要访问在托管环境中的 JTA TransactionManager。由于 J2EE 没有标准化一个单一的机制，Hibernate 在应用程序服务器中，你必须指定 Hibernate 如何获得 TransactionManager 的引用：

表 3.10. JTA TransactionManagers

Transaction 工厂类	应用程序服务器
org.hibernate.transaction.JBossTransactionManagerLookup	JBoss AS
org.hibernate.transaction.WeblogicTransactionManagerLookup	Weblogic
org.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
org.hibernate.transaction.WebSphereExtendedJTATransactionLookup	WebSphere 6
org.hibernate.transaction.OrionTransactionManagerLookup	Orion
org.hibernate.transaction.ResinTransactionManagerLookup	Resin
org.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
org.hibernate.transaction.JOnASTransactionManagerLookup	JOnAS
org.hibernate.transaction.JRun4TransactionManagerLookup	JRun4
org.hibernate.transaction.BESTransactionManagerLookup	Borland ES
org.hibernate.transaction.JBossTSStandaloneTransactionManagerLookup	JBoss TS used standalone (ie. outside JBoss AS and a JNDI environment generally). Known to work for org.jboss.jbossts:jbossjta:4.11.0.Final

3.8.2. JNDI 绑定的 SessionFactory

与 JNDI 绑定的 Hibernate 的 SessionFactory 能简化工厂的查询，简化创建新的 Session。需要注意的是这与 JNDI 绑定 Datasource 没有关系，它们只是恰巧用了相同的注册表。

如果你希望将 SessionFactory 绑定到一个 JNDI 的名字空间，用属性 hibernate.session_factory_name 指定一个名字（如，java: hibernate/SessionFactory）。如果不设置这个属性，SessionFactory 将不会被绑定到 JNDI 中（在以只读 JNDI 为默认实现的环境中，这个设置尤其有用，如 Tomcat）。

在将 SessionFactory 绑定至 JNDI 时，Hibernate 将使用 hibernate.jndi.url，和 hibernate.jndi.class 的值来实例化初始环境（initial context）。如果它们没有被指定，将使用默认的 InitialContext。

在你调用 cfg.buildSessionFactory()后，Hibernate 会自动将 SessionFactory 注册到 JNDI。这意味这你至少需要在你应用程序的启动代码（或工具类）中完成这个调用，除非你使用 HibernateService 来做 JMX 部署（见后面讨论）。

假若你使用 JNDI SessionFactory，EJB 或者任何其它类都可以从 JNDI 中找到此 SessionFactory。

It is recommended that you bind the SessionFactory to JNDI in a managed environment and use a static singleton otherwise. To shield your application code from these details, we also recommend to hide the actual lookup code for a SessionFactory in a helper class, such as HibernateUtil.getSessionFactory(). Note that such a class is also a convenient way to startup Hibernate—see chapter 1.

3.8.3. 在 JTA 环境下使用 Current Session context（当前 session 上下文）管理

The easiest way to handle Sessions and transactions is Hibernate's automatic "current" Session management. For a discussion of contextual sessions see 第 2.3 节 “上下文相关的会话（Contextual Session）”. Using the "jta" session context, if there is no Hibernate Session associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call sessionFactory.getCurrentSession(). The Sessions retrieved via getCurrentSession() in the "jta" context are set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the Sessions to be managed by the life cycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through UserTransaction, or (recommended for portable code) use the Hibernate Transaction API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

3.8.4. JMX 部署

为了将 SessionFactory 注册到 JNDI 中，cfg.buildSessionFactory() 这行代码仍需在某处被执行。你可在一个 static 初始化块（像 HibernateUtil 中的那样）中执行它或将 Hibernate 部署为一个托管的服务。

为了部署在一个支持 JMX 的应用程序服务器上，Hibernate 和 org.hibernate.jmx.HibernateService 一同分发，如 Jboss AS。 实际的部署和配置是由应用程序服务器提供者指定的。这里是 JBoss 4.0.x 的 jboss-service.xml 样例：

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

```
<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">

    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
```

这个文件是部署在 META-INF 目录下的，并会被打包到以 .sar（service archive）为扩展名的 JAR 文件中。同时，你需要将 Hibernate、它所需要的第三方库、你编译好的持久化类以及你的映射定义文件打包进同一个文档。你的企业 Bean（一般为会话 Bean）可能会被打包成它们自己的 JAR 文件，但你也许会将 EJB JAR 文件一同包含进能独立（热）部署的主服务文档。参考 JBoss AS 文档以了解更多的 JMX服务与 EJB 部署的信息。

持久化类 (Persistent Classes)

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). The term "persistent" here means that the classes are able to be persisted, not that they are in the persistent state (see 第 11.1 节 “Hibernate 对象状态 (object states)” for discussion).

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of java.util.Map instances, for example).

4.1. 一个简单的 POJO 例子

例 4.1. Simple POJO representing a cat

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

public class Cat {
private Long id; // identifier

private Date birthdate;
private Color color;
private char sex;
private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

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

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

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

在后续的章节里我们将介绍持久性类的 4 个主要规则的更多细节。

4.1.1. 实现一个默认的（即无参数的）构造方法（constructor）

Cat has a no-argument constructor. All persistent classes must have a default constructor (which can be non-public) so that Hibernate can instantiate them using java.lang.reflect.Constructor.newInstance(). It is recommended that this constructor be defined with at least package visibility in order for runtime proxy generation to work properly.

4.1.2. Provide an identifier property

注意

Historically this was considered option. While still not (yet) enforced, this should be considered a deprecated feature as it will be completely required to provide a identifier property in an upcoming release.

Cat has a property named id. This property maps to the primary key column(s) of the underlying database table. The type of the identifier property can be any "basic" type (see ???). See 第 9.4 节 “组件作为联合标识符（Components as composite identifiers）” for information on mapping composite (multi-column) identifiers.

注意

Identifiers do not necessarily need to identify column(s) in the database physically defined as a primary key. They should just identify columns that can be used to uniquely identify rows in the underlying table.

我们建议你对持久化类声明命名一致的标识属性。我们还建议你使用一个可以为空（也就是说，不是原始类型）的类型。

4.1.3. Prefer non-final classes (semi-optional)

A central feature of Hibernate, proxies (lazy loading), depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods. You can persist final classes that do not implement an interface with Hibernate; you will not, however, be able to use proxies for lazy association fetching which will ultimately limit your options for performance tuning. To persist a final class which does not implement a "full" interface you must disable proxy generation. See 例 4.2 “Disabling proxies in hbm.xml” and 例 4.3 “Disabling proxies in annotations”.

例 4.2. Disabling proxies in hbm.xml

```
<class name="Cat" lazy="false"...>...</class>
```

例 4.3. Disabling proxies in annotations

```
@Entity @Proxy(lazy=false) public class Cat { ... }
```

If the final class does implement a proper interface, you could alternatively tell Hibernate to use the interface instead when generating the proxies. See 例 4.4 “Proxying an interface in hbm.xml” and 例 4.5 “Proxying an interface in annotations”.

例 4.4. Proxying an interface in hbm.xml

```
<class name="Cat" proxy="ICat"...>...</class>
```

例 4.5. Proxying an interface in annotations

```
@Entity @Proxy(proxyClass=ICat.class) public class Cat implements ICat { ... }
```

You should also avoid declaring public final methods as this will again limit the ability to generate proxies from this class. If you want to use a class with public final methods, you must explicitly disable proxying. Again, see 例 4.2 “Disabling proxies in hbm.xml” and 例 4.3 “Disabling proxies in annotations”.

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

Cat declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. It is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties and recognizes method names of the form getFoo, isFoo and setFoo. If required, you can switch to direct field access for particular properties.

Properties need not be declared public. Hibernate can persist a property declared with package, protected or private visibility as well.

4.2. 实现继承 (Inheritance)

子类也必须遵守第一条和第二条规则。它从超类 Cat 继承了标识属性。例如:

```
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. 实现 equals() 和 hashCode() 方法:

如果你有如下需求，你必须重载 equals() 和 hashCode() 方法:

- 想把持久类的实例放入 Set 中（当表示多值关联时，推荐这么做），而且
- 想重用脱管实例

Hibernate 保证，仅在特定会话范围内，持久化标识（数据库的行）和 Java 标识是等价的。因此，一旦我们混合了从不同会话中获取的实例，如果希望 Set 有明确的语义，就必须实现 equals() 和 hashCode()。

实现 equals()/hashCode() 最显而易见的方法是比较两个对象 标识符的值。如果值相同，则两个对象对应于数据库的同一行，因此它们是相等的（如果都被添加到 Set，则在 Set 中只有一个元素）。不幸的是，对生成的标识不能 使用这种方法。Hibernate 仅对那些持久化对象赋标识值，一个新创建的实例将不会有任何标识值。此外， 如果一个实例没有被保存（unsaved），并且它当前正在一个 Set 中，保存它将会给这个对象赋一个标识值。如果 equals() 和 hashCode() 是基于标识值 实现的，则其哈希码将会改变，这违反了 Set 的契约。建议去 Hibernate 的站点阅读关于这个问题的全部讨论。注意，这不是 Hibernate 的问题，而是一般的 Java 对象标识和 Java 对象等价的语义问题。

我们建议使用业务键值相等（Business key equality）来实现 equals() 和 hashCode()。业务键值相等的意思是，equals() 方法仅仅比较形成业务键的属性，它能在现实世界里标识我们的实例（是一个自然的候选码）。

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

A business key does not have to be as solid as a database primary key candidate (see 第 13.1.3 节 “关注对象标识 (Considering object identity)”). Immutable or unique properties are usually good candidates for a business key.

4.4. 动态模型 (Dynamic models)

注意

The following features are currently considered experimental and may change in the near future.

运行期的持久化实体没有必要一定表示为像 POJO 类或 JavaBean 对象那样的形式。Hibernate 也支持动态模型 (在运行期使用 Map 的 Map) 和象 DOM4J 的树模型那样的实体表示。使用这种方法，你不用写持久化类，只写映射文件就行了。

By default, Hibernate works in normal POJO mode. You can set a default entity representation mode for a particular SessionFactory using the default_entity_mode configuration option (see 表 3.3 “Hibernate 配置属性”).

下面是用 Map 来表示的例子。首先，在映射文件中，要声明 entity-name 来代替一个类名 (或作为一种附属)。

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>
```

```
    </class>

</hibernate-mapping>
```

注意，虽然是用目标类名来声明关联的，但是关联的目标类型除了是 POJO 之外，也可以是一个动态的实体。

在使用 dynamic-map 为 SessionFactory 设置了默认的实体模式之后，可以在运行期使用 Map 的 Map:

```
Session s = openSession();
Transaction tx = s.beginTransaction();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

动态映射的好处是，变化所需要的时间少了，因为原型不需要实现实体类。然而，你无法进行编译期的类型检查，并可能由此会处理很多的运行期异常。幸亏有了 Hibernate 映射，它使得数据库的 schema 能容易的规格化和合理化，并允许稍后在此之上添加合适的领域模型实现。

实体表示模式也能在每个 Session 的基础上设置:

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

请注意，用 EntityMode 调用 getSession() 是在 Session 的 API 中，而不是 SessionFactory。 这样，新的 Session 共享底层的 JDBC 连接，事务，和其他的上下文信息。这意味着，你不需要在第二个 Session 中调用 flush() 和 close()，同样的，把事务和连接的处理交给原来的工作单元。

More information about the XML representation capabilities can be found in 第 20 章 XML 映射.

4.5. 元组片断映射 (Tuplizers)

org.hibernate.tuple.Tuplizer and its sub-interfaces are responsible for managing a particular representation of a piece of data given that representation's org.hibernate.EntityMode. If a given piece of data is thought of as a data structure, then a tuplizer is the thing that knows how to create such a data structure, how to extract values from such a data structure and how to inject values into such a data structure. For example, for the POJO entity mode, the corresponding tuplizer knows how create the POJO through its constructor. It also knows how to access the POJO properties using the defined property accessors.

There are two (high-level) types of Tuplizers:

- org.hibernate.tuple.entity.EntityTuplizer which is responsible for managing the above mentioned contracts in regards to entities
- org.hibernate.tuple.component.ComponentTuplizer which does the same for components

Users can also plug in their own tuplizers. Perhaps you require that java.util.Map implementation other than java.util.HashMap be used while in the dynamic-map entity-mode. Or perhaps you need to define a different proxy generation strategy than the one used by default. Both would be achieved by defining a custom tuplizer implementation. Tuplizer definitions are attached to the entity or component mapping they are meant to manage. Going back to the example of our Customer entity, 例 4.6 “Specify custom tuplizers in annotations” shows how to specify a custom org.hibernate.tuple.entity.EntityTuplizer using annotations while 例 4.7 “Specify custom tuplizers in hbm.xml” shows how to do the same in hbm.xml

例 4.6. Specify custom tuplizers in annotations

```
@Entity
@Tuplizer(impl = DynamicEntityTuplizer.class)
public interface Cuisine {
    @Id
    @GeneratedValue
    public Long getId();
    public void setId(Long id);

    public String getName();
    public void setName(String name);
```

```
    @Tuplizer(impl = DynamicComponentTuplizer.class)
    public Country getCountry();
    public void setCountry(Country country);
}
```

例 4.7. Specify custom tuplizers in `hbm.xml`

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>
```

4.6. EntityNameResolvers

`org.hibernate.EntityNameResolver` is a contract for resolving the entity name of a given entity instance. The interface defines a single method `resolveEntityName` which is passed the entity instance and is expected to return the appropriate entity name (null is allowed and would indicate that the resolver does not know how to resolve the entity name of the given entity instance). Generally speaking, an `org.hibernate.EntityNameResolver` is going to be most useful in the case of dynamic models. One example might be using proxied interfaces as your domain model. The hibernate test suite has an example of this exact style of usage under the `org.hibernate.test.dynamicentity.tuplizer2`. Here is some of the code from that package for illustration.

```
/**
 * A very trivial JDK Proxy InvocationHandler implementation where we proxy an
 * interface as the domain model and simply store persistent state in an internal
 * Map.  This is an extremely trivial example meant only for illustration.
 */
public final class DataProxyHandler implements InvocationHandler {
        private String entityName;
        private HashMap data = new HashMap();

        public DataProxyHandler(String entityName, Serializable id) {
                this.entityName = entityName;
                data.put( "Id", id );
        }
```

```
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                String methodName = method.getName();
                if ( methodName.startsWith( "set" ) ) {
                        String propertyName = methodName.substring( 3 );
                        data.put( propertyName, args[0] );
                }
                else if ( methodName.startsWith( "get" ) ) {
                        String propertyName = methodName.substring( 3 );
                        return data.get( propertyName );
                }
                else if ( "toString".equals( methodName ) ) {
                        return entityName + "#" + data.get( "Id" );
                }
                else if ( "hashCode".equals( methodName ) ) {
                        return new Integer( this.hashCode() );
                }
                return null;
        }

        public String getEntityName() {
                return entityName;
        }

        public HashMap getData() {
                return data;
        }
}

public class ProxyHelper {
    public static String extractEntityName(Object object) {
        // Our custom java.lang.reflect.Proxy instances actually bundle
        // their appropriate entity name, so we simply extract it from there
        // if this represents one of our proxies; otherwise, we return null
        if ( Proxy.isProxyClass( object.getClass() ) ) {
            InvocationHandler handler = Proxy.getInvocationHandler( object );
            if ( DataProxyHandler.class.isAssignableFrom( handler.getClass() ) ) {
                DataProxyHandler myHandler = ( DataProxyHandler ) handler;
                return myHandler.getEntityName();
            }
        }
        return null;
    }

    // various other utility methods ....

}

/**
 * The EntityNameResolver implementation.
 *
 * IMPL NOTE : An EntityNameResolver really defines a strategy for how entity names
 * should be resolved.  Since this particular impl can handle resolution for all of our
 * entities we want to take advantage of the fact that SessionFactoryImpl keeps these
 * in a Set so that we only ever have one instance registered.  Why?  Well, when it
 * comes time to resolve an entity name, Hibernate must iterate over all the registered
 * resolvers.  So keeping that number down helps that process be as speedy as possible.
 * Hence the equals and hashCode implementations as is
 */
```

```
public class MyEntityNameResolver implements EntityNameResolver {
    public static final MyEntityNameResolver INSTANCE = new MyEntityNameResolver();

    public String resolveEntityName(Object entity) {
        return ProxyHelper.extractEntityName( entity );
    }

    public boolean equals(Object obj) {
        return getClass().equals( obj.getClass() );
    }

    public int hashCode() {
        return getClass().hashCode();
    }
}

public class MyEntityTuplizer extends PojoEntityTuplizer {
       public MyEntityTuplizer(EntityMetamodel entityMetamodel, PersistentClass mappedEntity) {
               super( entityMetamodel, mappedEntity );
        }

        public EntityNameResolver[] getEntityNameResolvers() {
               return new EntityNameResolver[] { MyEntityNameResolver.INSTANCE };
        }

    public String determineConcreteSubclassEntityName(Object entityInstance, SessionFactoryImplementor factory) {
        String entityName = ProxyHelper.extractEntityName( entityInstance );
        if ( entityName == null ) {
            entityName = super.determineConcreteSubclassEntityName( entityInstance, factory );
        }
        return entityName;
    }

    ...
```

为了注册 org.hibernate.EntityNameResolver，用户必须：

1. Implement a custom tuplizer (see 第 4.5 节 “元组片断映射（Tuplizers）”), implementing the getEntityNameResolvers method
2. 用 registerEntityNameResolver 方法注册到 org.hibernate.impl.SessionFactoryImpl（它是 org.hibernate.SessionFactory 的实现类）。

对象/关系数据库映射基础 (Basic O/R Mapping)

5.1. 映射定义 (Mapping declaration)

Object/relational mappings can be defined in three approaches:

- using Java 5 annotations (via the Java Persistence 2 annotations)
- using JPA 2 XML deployment descriptors (described in chapter XXX)
- using the Hibernate legacy XML files approach known as hbm.xml

Annotations are split in two categories, the logical mapping annotations (describing the object model, the association between two entities etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc). We will mix annotations from both categories in the following code examples.

JPA annotations are in the `javax.persistence.*` package. Hibernate specific extensions are in `org.hibernate.annotations.*`. You favorite IDE can auto-complete annotations and their attributes for you (even without a specific "JPA" plugin, since JPA annotations are plain Java 5 annotations).

Here is an example of mapping

```
package eg;

@Entity
@Table(name="cats") @Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorValue("C") @DiscriminatorColumn(name="subclass", discriminatorType=CHAR)
public class Cat {

   @Id @GeneratedValue
   public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public BigDecimal getWeight() { return weight; }
   public void setWeight(BigDecimal weight) { this.weight = weight; }
   private BigDecimal weight;

   @Temporal(DATE) @NotNull @Column(updatable=false)
   public Date getBirthdate() { return birthdate; }
   public void setBirthdate(Date birthdate) { this.birthdate = birthdate; }
   private Date birthdate;

   @org.hibernate.annotations.Type(type="eg.types.ColorUserType")
   @NotNull @Column(updatable=false)
   public ColorType getColor() { return color; }
```

```
   public void setColor(ColorType color) { this.color = color; }
   private ColorType color;

   @NotNull @Column(updatable=false)
   public String getSex() { return sex; }
   public void setSex(String sex) { this.sex = sex; }
   private String sex;

   @NotNull @Column(updatable=false)
   public Integer getLitterId() { return litterId; }
   public void setLitterId(Integer litterId) { this.litterId = litterId; }
   private Integer litterId;

   @ManyToOne @JoinColumn(name="mother_id", updatable=false)
   public Cat getMother() { return mother; }
   public void setMother(Cat mother) { this.mother = mother; }
   private Cat mother;

   @OneToMany(mappedBy="mother") @OrderBy("litterId")
   public Set<Cat> getKittens() { return kittens; }
   public void setKittens(Set<Cat> kittens) { this.kittens = kittens; }
   private Set<Cat> kittens = new HashSet<Cat>();
}

@Entity @DiscriminatorValue("D")
public class DomesticCat extends Cat {

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

@Entity
public class Dog { ... }
```

The legacy hbm.xml approach uses an XML schema designed to be readable and hand-editable. The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations and not table declarations.

请注意，虽然很多 Hibernate 用户选择手写 XML 映射文档，但也有一些工具可以用来生成映射文档，包括 XDoclet、Middlegen 和 AndroMDA。

下面是一个映射的例子：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">
```

```
                <id name="id">
                        <generator class="native"/>
                </id>

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <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>
```

We will now discuss the concepts of the mapping documents (both annotations and XML). We will only describe, however, the document elements and attributes that are used by Hibernate at runtime. The mapping document also contains some extra optional attributes

and elements that affect the database schemas exported by the schema export tool (for example, the `not-null` attribute).

5.1.1. Entity

An entity is a regular Java object (aka POJO) which will be persisted by Hibernate.

To mark an object as an entity in annotations, use the `@Entity` annotation.

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() { return id; }

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

That's pretty much it, the rest is optional. There are however any options to tweak your entity mapping, let's explore them.

`@Table` lets you define the table the entity will be persisted into. If undefined, the table name is the unqualified class name of the entity. You can also optionally define the catalog, the schema as well as unique constraints on the table.

```
@Entity
@Table(name="TBL_FLIGHT",
       schema="AIR_COMMAND",
       uniqueConstraints=
           @UniqueConstraint(
               name="flight_number",
               columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {
    @Column(name="comp_prefix")
    public String getCompagnyPrefix() { return companyPrefix; }

    @Column(name="flight_number")
    public String getNumber() { return number; }
}
```

The constraint name is optional (generated if left undefined). The column names composing the constraint correspond to the column names as defined before the Hibernate `NamingStrategy` is applied.

`@Entity.name` lets you define the shortcut name of the entity you can used in JP-QL and HQL queries. It defaults to the unqualified class name of the class.

Hibernate goes beyond the JPA specification and provide additional configurations. Some of them are hosted on `@org.hibernate.annotations.Entity`:

- dynamicInsert / dynamicUpdate (defaults to false): specifies that INSERT / UPDATE SQL should be generated at runtime and contain only the columns whose values are not null. The dynamic-update and dynamic-insert settings are not inherited by subclasses. Although these settings can increase performance in some cases, they can actually decrease performance in others.

- selectBeforeUpdate (defaults to false): specifies that Hibernate should never perform an SQL UPDATE unless it is certain that an object is actually modified. Only when a transient object has been associated with a new session using update(), will Hibernate perform an extra SQL SELECT to determine if an UPDATE is actually required. Use of select-before-update will usually decrease performance. It is useful to prevent a database update trigger being called unnecessarily if you reattach a graph of detached instances to a Session.

- polymorphisms (defaults to IMPLICIT): determines whether implicit or explicit query polymorphisms is used. Implicit polymorphisms means that instances of the class will be returned by a query that names any superclass or implemented interface or class, and that instances of any subclass of the class will be returned by a query that names the class itself. Explicit polymorphisms means that class instances will be returned only by queries that explicitly name that class. Queries that name the class will return only instances of subclasses mapped. For most purposes, the default polymorphisms=IMPLICIT is appropriate. Explicit polymorphisms is useful when two different classes are mapped to the same table This allows a "lightweight" class that contains a subset of the table columns.

- persister: specifies a custom ClassPersister. The persister attribute lets you customize the persistence strategy used for the class. You can, for example, specify your own subclass of org.hibernate.persister.EntityPersister, or you can even provide a completely new implementation of the interface org.hibernate.persister.ClassPersister that implements, for example, persistence via stored procedure calls, serialization to flat files or LDAP. See org.hibernate.test.CustomPersister for a simple example of "persistence" to a Hashtable.

- optimisticLock (defaults to VERSION): determines the optimistic locking strategy. If you enable dynamicUpdate, you will have a choice of optimistic locking strategies:

  - 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，允许某些并行更新

  - none（不检查）：不使用乐观锁定

  我们强烈建议你在 Hibernate 中使用 version/timestamp 字段来进行乐观锁定。这个选择可以优化性能，且能够处理对脱管实例的修改（例如：在使用 Session.merge() 的时候）。

提示

Be sure to import @javax.persistence.Entity to mark a class as an entity. It's a common mistake to import @org.hibernate.annotations.Entity by accident.

Some entities are not mutable. They cannot be updated or deleted by the application. This allows Hibernate to make some minor performance optimizations.. Use the @Immutable annotation.

You can also alter how Hibernate deals with lazy initialization for this class. On @Proxy, use lazy=false to disable lazy fetching (not recommended). You can also specify an interface to use for lazy initializing proxies (defaults to the class itself): use proxyClass on @Proxy. Hibernate will initially return proxies (Javassist or CGLIB) that implement the named interface. The persistent object will load when a method of the proxy is invoked. See "Initializing collections and proxies" below.

@BatchSize specifies a "batch size" for fetching instances of this class by identifier. Not yet loaded instances are loaded batch-size at a time (default 1).

You can specific an arbitrary SQL WHERE condition to be used when retrieving objects of this class. Use @Where for that.

In the same vein, @Check lets you define an SQL expression used to generate a multi-row check constraint for automatic schema generation.

There is no difference between a view and a base table for a Hibernate mapping. This is transparent at the database level, although some DBMS do not support views properly, especially with updates. Sometimes you want to use a view, but you cannot create one in the database (i.e. with a legacy schema). In this case, you can map an immutable and read-only entity to a given SQL subselect expression using @org.hibernate.annotations.Subselect:

```
@Entity
@Subselect("select item.name, max(bid.amount), count(*) "
        + "from item "
        + "join bid on bid.item_id = item.id "
        + "group by item.name")
@Synchronize( {"item", "bid"} ) //tables impacted
public class Summary {
    @Id
    public String getId() { return id; }
    ...
}
```

定义这个实体用到的表为同步（synchronize），确保自动刷新（auto-flush）正确执行，并且依赖原实体的查询不会返回过期数据。在属性元素和嵌套映射元素中都可使用 <subselect>。

We will now explore the same options using the hbm.xml structure. You can declare a persistent class using the class element. For example:

```
<class
        name="ClassName"                                                    ❶
        table="tableName"                                                   ❷
        discriminator-value="discriminator_value"                           ❸
        mutable="true|false"                                                ❹
        schema="owner"                                                      ❺
        catalog="catalog"                                                   ❻
        proxy="ProxyInterface"                                              ❼
        dynamic-update="true|false"                                         ❽
        dynamic-insert="true|false"                                         ❾
        select-before-update="true|false"                                   ❿
        polymorphism="implicit|explicit"                                    ⓫
        where="arbitrary sql where condition"                               ⓬
        persister="PersisterClass"                                          ⓭
        batch-size="N"                                                      ⓮
        optimistic-lock="none|version|dirty|all"                            ⓯
        lazy="true|false"                                                   (16)
        entity-name="EntityName"                                            (17)
        check="arbitrary sql check condition"                               (18)
        rowid="rowid"                                                       (19)
        subselect="SQL expression"                                          (20)
        abstract="true|false"                                               (21)
        node="element-name"
/>
```

❶ name（可选）：持久化类（或者接口）的 Java 全限定名。 如果这个属性不存在，Hibernate 将假定这是一个非 POJO 的实体映射。

❷ table（可选 — 默认是类的非全限定名）：对应的数据库表名。

❸ discriminator-value（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。它可以接受的值包括 null 和 not null。

❹ mutable（可选，默认值为 true）：表明该类的实例是可变的或者不可变的。

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

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

❼ 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。

❽ dynamic-update（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。

❾ dynamic-insert（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。

10. select-before-update（可选，默认为 false）：指定 Hibernate 除非确定对象真正被修改了（如果该值为 true — 译注），否则不会执行 SQL UPDATE 操作。在特定场合（实际上，它只在一个瞬时对象（transient object）关联到一个新的 session 中时执行的 update() 中生效），这说明 Hibernate 会在 UPDATE 之前执行一次额外的 SQL SELECT 操作来决定是否确实需要执行 UPDATE。
11. polymorphisms (optional - defaults to implicit): determines whether implicit or explicit query polymorphisms is used.
12. where（可选）指定一个附加的 SQL WHERE 条件，在抓取这个类的对象时会一直增加这个条件。
13. persister（可选）：指定一个定制的 ClassPersister。
14. batch-size（可选,默认是 1）指定一个用于 根据标识符（identifier）抓取实例时使用的 "batch size"（批次抓取数量）。
15. optimistic-lock（乐观锁定）（可选，默认是 version）：决定乐观锁定的策略。
16. lazy（可选）：通过设置 lazy="false"，所有的延迟加载（Lazy fetching）功能将被全部禁用（disabled）。
17. entity-name (optional - defaults to the class name): Hibernate3 allows a class to be mapped multiple times, potentially to different tables. It also allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity. See 第 4.4 节 “动态模型（Dynamic models）” and 第 20 章 XML 映射 for more information.
18. check（可选）：这是一个 SQL 表达式， 用于为自动生成的 schema 添加多行（multi-row）约束检查。
19. rowid（可选）：Hibernate 可以使用数据库支持的所谓的 ROWIDs，例如：Oracle 数据库，如果你设置这个可选的 rowid，Hibernate 可以使用额外的字段 rowid 实现快速更新。ROWID 是这个功能实现的重点，它代表了一个存储元组（tuple）的物理位置。
20. subselect（可选）：它将一个不可变（immutable）并且只读的实体映射到一个数据库的子查询中。当你想用视图代替一张基本表的时候，这是有用的，但最好不要这样做。更多的介绍请看下面内容。
21. abstract（可选）：用于在 <union-subclass> 的层次结构（hierarchies）中标识抽象超类。

若指明的持久化类实际上是一个接口，这也是完全可以接受的。之后你可以用元素 <subclass> 来指定该接口的实际实现类。你可以持久化任何 static（静态的）内部类。你应该使用标准的类名格式来指定类名，比如：FooSBar。

Here is how to do a virtual view (subselect) in XML:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
```

```
</class>
```

The <subselect> is available both as an attribute and a nested mapping element.

5.1.2. Identifiers

Mapped classes must declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance.

Mark the identifier property with @Id.

```
@Entity
public class Person {
   @Id Integer getId() { ... }
   ...
}
```

In hbm.xml, use the <id> element which defines the mapping from that property to the primary key column.

```
<id
        name="propertyName"                                          (1)
        type="typename"                                              (2)
        column="column_name"                                         (3)
        unsaved-value="null|any|none|undefined|id_value"             (4)
        access="field|property|ClassName">                           (5)
        node="element-name|@attribute-name|element/@attribute|."

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

(1) name（可选）：标识属性的名字。
(2) type（可选）：一个 Hibernate 类型的名字。
(3) column（可选 — 默认为属性名）：主键字段的名字。
(4) unsaved-value（可选 — 默认为一个切合实际（sensible）的值）：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。
(5) 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。

The unsaved-value attribute is almost never needed in Hibernate3 and indeed has no corresponding element in annotations.

You can also declare the identifier as a composite identifier. This allows access to legacy data with composite keys. Its use is strongly discouraged for anything else.

5.1.2.1. Composite identifier

You can define a composite primary key through several syntaxes:

- use a component type to represent the identifier and map it as a property in the entity: you then annotated the property as `@EmbeddedId`. The component type has to be `Serializable`.
- map multiple properties as `@Id` properties: the identifier type is then the entity class itself and needs to be `Serializable`. This approach is unfortunately not standard and only supported by Hibernate.
- map multiple properties as `@Id` properties and declare an external class to be the identifier type. This class, which needs to be `Serializable`, is declared on the entity via the `@IdClass` annotation. The identifier type must contain the same properties as the identifier properties of the entity: each property name must be the same, its type must be the same as well if the entity property is of a basic type, its type must be the type of the primary key of the associated entity if the entity property is an association (either a `@OneToOne` or a `@ManyToOne`).

As you can see the last case is far from obvious. It has been inherited from the dark ages of EJB 2 for backward compatibilities and we recommend you not to use it (for simplicity sake).

Let's explore all three cases using examples.

5.1.2.1.1. id as a property using a component type

Here is a simple example of `@EmbeddedId`.

```
@Entity
class User {
   @EmbeddedId
   @AttributeOverride(name="firstName", column=@Column(name="fld_firstname")
   UserId id;

   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;
}
```

You can notice that the `UserId` class is serializable. To override the column mapping, use `@AttributeOverride`.

An embedded id can itself contains the primary key of an associated entity.

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;

   @MapsId("userId")
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   @OneToOne User user;
}

@Embeddable
class CustomerId implements Serializable {
   UserId userId;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In the embedded id object, the association is represented as the identifier of the associated entity. But you can link its value to a regular association in the entity via the `@MapsId` annotation. The `@MapsId` value correspond to the property name of the embedded id object containing the associated entity's identifier. In the database, it means that the `Customer.user` and the `CustomerId.userId` properties share the same underlying column (`user_fk` in this case).

提示

The component type used as identifier must implement `equals()` and `hashCode()`.

In practice, your code only sets the `Customer.user` property and the user id value is copied by Hibernate into the `CustomerId.userId` property.

警告

The id value can be copied as late as flush time, don't rely on it until after flush time.

While not supported in JPA, Hibernate lets you place your association directly in the embedded id component (instead of having to use the @MapsId annotation).

```
@Entity
class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;
}

@Embeddable
class CustomerId implements Serializable {
   @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;


   //implements equals and hashCode
}
```

Let's now rewrite these examples using the hbm.xml syntax.

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName"
        node="element-name|.">
```

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

First a simple example:

```
<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName" column="fld_firstname"/>
      <key-property name="lastName"/>
   </composite-id>
</class>
```

Then an example showing how an association can be mapped.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-property name="firstName" column="userfirstname_fk"/>
      <key-property name="lastName" column="userfirstname_fk"/>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>

   <many-to-one name="user">
      <column name="userfirstname_fk" updatable="false" insertable="false"/>
      <column name="userlastname_fk" updatable="false" insertable="false"/>
   </many-to-one>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

Notice a few things in the previous example:

- the order of the properties (and column) matters. It must be the same between the association and the primary key of the associated entity
- the many to one uses the same columns as the primary key and thus must be marked as read only (`insertable` and `updatable` to false).
- unlike with `@MapsId`, the id value of the associated entity is not transparently copied, check the `foreign` id generator for more information.

The last example shows how to map association directly in the embedded id component.

```
<class name="Customer">
   <composite-id name="id" class="CustomerId">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

This is the recommended approach to map composite identifier. The following options should not be considered unless some constraint are present.

5.1.2.1.2. Multiple id properties without identifier type

Another, arguably more natural, approach is to place `@Id` on multiple properties of your entity. This approach is only supported by Hibernate (not JPA compliant) but does not require an extra embeddable component.

```
@Entity
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;
}
```

```
@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

In this case Customer is its own identifier representation: it must implement Serializable and must implement equals() and hashCode().

In hbm.xml, the same mapping is:

```
<class name="Customer">
   <composite-id>
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

5.1.2.1.3. Multiple id properties with with a dedicated identifier type

@IdClass on an entity points to the class (component) representing the identifier of the class. The properties marked @Id on the entity must have their corresponding property on the @IdClass. The return type of search twin property must be either identical for basic properties or must correspond to the identifier class of the associated entity for an association.

警告

This approach is inherited from the EJB 2 days and we recommend against its use. But, after all it's your application and Hibernate supports it.

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;

   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   UserId user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;

   //implements equals and hashCode
}
```

Customer and CustomerId do have the same properties customerNumber as well as user. CustomerId must be Serializable and implement equals() and hashCode().

While not JPA standard, Hibernate let's you declare the vanilla associated property in the @IdClass.

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;

   @Id String customerNumber;
```

```
   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   @OneToOne User user;
   String customerNumber;

   //implements equals and hashCode
}

@Entity
class User {
   @EmbeddedId UserId id;
   Integer age;

   //implements equals and hashCode
}

@Embeddable
class UserId implements Serializable {
  String firstName;
  String lastName;
}
```

This feature is of limited interest though as you are likely to have chosen the `@IdClass` approach to stay JPA compliant or you have a quite twisted mind.

Here are the equivalent on hbm.xml files:

```
<class name="Customer">
   <composite-id class="CustomerId" mapped="true">
      <key-many-to-one name="user">
         <column name="userfirstname_fk"/>
         <column name="userlastname_fk"/>
      </key-many-to-one>
      <key-property name="customerNumber"/>
   </composite-id>

   <property name="preferredCustomer"/>
</class>

<class name="User">
   <composite-id name="id" class="UserId">
      <key-property name="firstName"/>
      <key-property name="lastName"/>
   </composite-id>

   <property name="age"/>
</class>
```

5.1.2.2. Identifier generator

Hibernate can generate and populate identifier values for you automatically. This is the recommended approach over "business" or "natural" id (especially composite ids).

Hibernate offers various generation strategies, let's explore the most common ones first that happens to be standardized by JPA:

- IDENTITY: supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type `long`, `short` or `int`.
- SEQUENCE (called `seqhilo` in Hibernate): uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a named database sequence.
- TABLE (called `MultipleHiLoPerTableGenerator` in Hibernate) : uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a table and column as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
- AUTO: selects `IDENTITY`, `SEQUENCE` or `TABLE` depending upon the capabilities of the underlying database.

重要

We recommend all new projects to use the new enhanced identifier generators. They are deactivated by default for entities using annotations but can be activated using `hibernate.id.new_generator_mappings=true`. These new generators are more efficient and closer to the JPA 2 specification semantic.

However they are not backward compatible with existing Hibernate based application (if a sequence or a table is used for id generation). See XXXXXXX ??? for more information on how to activate them.

To mark an id property as generated, use the `@GeneratedValue` annotation. You can specify the strategy used (default to `AUTO`) by setting `strategy`.

```
@Entity
public class Customer {
   @Id @GeneratedValue
   Integer getId() { ... };
}

@Entity
public class Invoice {
   @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
   Integer getId() { ... };
}
```

SEQUENCE and TABLE require additional configurations that you can set using @SequenceGenerator and @TableGenerator:

- name: name of the generator
- table / sequenceName: name of the table or the sequence (defaulting respectively to hibernate_sequences and hibernate_sequence)
- catalog / schema:
- initialValue: the value from which the id is to start generating
- allocationSize: the amount to increment by when allocating id numbers from the generator

In addition, the TABLE strategy also let you customize:

- pkColumnName: the column name containing the entity identifier
- valueColumnName: the column name containing the identifier value
- pkColumnValue: the entity identifier
- uniqueConstraints: any potential column constraint on the table containing the ids

To link a table or sequence generator definition with an actual generated property, use the same name in both the definition name and the generator value generator as shown below.

```
@Id
@GeneratedValue(
    strategy=GenerationType.SEQUENCE,
    generator="SEQ_GEN")
@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
public Integer getId() { ... }
```

The scope of a generator definition can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined in JPA's XML deployment descriptors (see XXXXXX ???):

```
<table-generator name="EMP_GEN"
            table="GENERATOR_TABLE"
            pk-column-name="key"
            value-column-name="hi"
            pk-column-value="EMP"
            allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi"
    pkColumnValue="EMP",
    allocationSize=20
)

<sequence-generator name="SEQ_GEN"
    sequence-name="my_sequence"
    allocation-size="20"/>

//and the annotation equivalent

@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence",
    allocationSize=20
)
```

If a JPA XML descriptor (like META-INF/orm.xml) is used to define the generators, EMP_GEN and SEQ_GEN are application level generators.

注意

Package level definition is not supported by the JPA specification. However, you can use the @GenericGenerator at the package level (see ???).

These are the four standard JPA generators. Hibernate goes beyond that and provide additional generators or additional options as we will see below. You can also write your own custom identifier generator by implementing org.hibernate.id.IdentifierGenerator.

To define a custom generator, use the @GenericGenerator annotation (and its plural counter part @GenericGenerators) that describes the class of the identifier generator or its short cut name (as described below) and a list of key/value parameters. When using @GenericGenerator and assigning it via @GeneratedValue.generator, the @GeneratedValue.strategy is ignored: leave it blank.

```
@Id @GeneratedValue(generator="system-uuid")
```

```
@GenericGenerator(name="system-uuid", strategy = "uuid")
public String getId() {

@Id @GeneratedValue(generator="trigger-generated")
@GenericGenerator(
    name="trigger-generated",
    strategy = "select",
    parameters = @Parameter(name="key", value = "socialSecurityNumber")
)
public String getId() {
```

The hbm.xml approach uses the optional <generator> child element inside <id>. If any parameters are required to configure or initialize the generator instance, they are passed using the <param> element.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

5.1.2.2.1. Various additional generators

所有的生成器都实现 org.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
: 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
: 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
: 使用一个高/低位算法高效的生成 long，short 或者 int 类型的标识符。给定一个表和字段（默认分别是 hibernate_unique_key 和 next_hi）作为高位值的来源。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。

seqhilo
: 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

Generates a 128-bit UUID based on a custom algorithm. The value generated is represented as a string of 32 hexidecimal digits. Users can also configure it to use a separator (config parameter "separator") which separates the hexidecimal digits into 8{sep}8{sep}4{sep}8{sep}4. Note specifically that this is different than the IETF RFC 4122 representation of 8-4-4-4-12. If you need RFC 4122 compliant UUIDs, consider using "uuid2" generator discussed below.

uuid2

Generates a IETF RFC 4122 compliant (variant 2) 128-bit UUID. The exact "version" (the RFC term) generated depends on the pluggable "generation strategy" used (see below). Capable of generating values as java.util.UUID, java.lang.String or as a byte array of length 16 (byte[16]). The "generation strategy" is defined by the interface org.hibernate.id.UUIDGenerationStrategy. The generator defines 2 configuration parameters for defining which generation strategy to use:

uuid_gen_strategy_class

Names the UUIDGenerationStrategy class to use

uuid_gen_strategy

Names the UUIDGenerationStrategy instance to use

Out of the box, comes with the following strategies:

- org.hibernate.id.uuid.StandardRandomStrategy (the default) - generates "version 3" (aka, "random") UUID values via the randomUUID method of java.util.UUID
- org.hibernate.id.uuid.CustomVersionOneStrategy - generates "version 1" UUID values, using IP address since mac address not available. If you need mac address to be used, consider leveraging one of the existing third party UUID generators which sniff out mac address and integrating it via the org.hibernate.id.UUIDGenerationStrategy contract. Two such libraries known at time of this writing include http://johannburkard.de/software/uuid/ and http://commons.apache.org/sandbox/id/uuid.html

guid

在 MS SQL Server 和 MySQL 中使用数据库生成的 GUID 字符串。

native

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

assigned

让应用程序在调用 save() 之前为对象分配一个标识符。这是 <generator> 元素没有指定时的默认生成策略。

select

通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。

foreign
: 使用另外一个相关联的对象的标识符。它通常和 <one-to-one> 联合起来使用。

sequence-identity
: 一种特别的序列生成策略，它使用数据库序列来生成实际值，但将它和 JDBC3 的 getGeneratedKeys 结合在一起，使得在插入语句执行的时候就返回生成的值。目前为止只有面向 JDK 1.4 的 Oracle 10g 驱动支持这一策略。由于 Oracle 驱动程序的一个 bug，这些插入语句的注释被关闭了。

5.1.2.2.2. 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>
```

可惜的是，你在为 Hibernate 自行提供 Connection 时无法使用 hilo。当 Hibernate 使用 JTA 获取应用服务器的数据源连接时，你必须正确地配置 hibernate.transaction.manager_lookup_class。

5.1.2.2.3. 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。

5.1.2.2.4. 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 和 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 和 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。例如：

```
<id name="id" type="long" column="person_id">
```

```
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" 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，选择哪一个，这取决于底层数据库的支持能力。

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

If you want the application to assign identifiers, as opposed to having Hibernate generate them, you can use the assigned generator. This special generator uses the identifier value already assigned to the object's identifier property. The generator is used when the primary key is a natural key instead of a surrogate key. This is the default behavior if you do not specify @GeneratedValue nor <generator> elements.

当选择 assigned 生成器时，除非有一个 version 或 timestamp 属性，或者你定义了 Interceptor.isUnsaved()，否则需要让 Hiberante 使用 unsaved-value="undefined"，强制 Hibernatet 查询数据库来确定一个实例是瞬时的（transient）还是脱管的（detached）。

5.1.2.2.6. 触发器实现的主键生成器（Primary keys assigned by triggers）

仅仅用于遗留的 schema 中（Hibernate 不能用触发器生成 DDL）。

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

在上面的例子中，类定义了一个命名为 socialSecurityNumber 的具有唯一值的属性，它是一个自然键（natural key），命名为 person_id 的代理键（surrogate key）的值由触发器生成。

5.1.2.2.7. Identity copy (foreign generator)

Finally, you can ask Hibernate to copy the identifier from another associated entity. In the Hibernate jargon, it is known as a foreign generator but the JPA mapping reads better and is encouraged.

```
@Entity
class MedicalHistory implements Serializable {
  @Id @OneToOne
  @JoinColumn(name = "person_id")
```

```
  Person patient;
}

@Entity
public class Person implements Serializable {
  @Id @GeneratedValue Integer id;
}
```

Or alternatively

```
@Entity
class MedicalHistory implements Serializable {
  @Id Integer id;

  @MapsId @OneToOne
  @JoinColumn(name = "patient_id")
  Person patient;
}

@Entity
class Person {
  @Id @GeneratedValue Integer id;
}
```

In hbm.xml use the following approach:

```
<class name="MedicalHistory">
   <id name="id">
      <generator class="foreign">
         <param name="property">patient</param>
      </generator>
   </id>
   <one-to-one name="patient" class="Person" constrained="true"/>
</class>
```

5.1.2.3. 增强的标识符生成器

从 3.2.3 版本开始，有两个代表不同标识符生成概念的新的生成器。第一个概念是数据库移植性；第二个是优化。优化表示你不需对每个新标识符的请求都查询数据库。从 3.3.x 开始，这两个新的生成器都是用来取代上面所述的生成器的。然而，它们也包括在当前版本里且可以由 FQN 进行引用。

这些生成器的第一个是 org.hibernate.id.enhanced.SequenceStyleGenerator，首先，它是作为 sequence 生成器的替代物，其次，它是比 native 具有更好移植性的生成器。这是因为 native 通常在 identity 和 sequence 之间选择，它有差别很大的 semantic，在移植时会导致潜在的问题。然而，org.hibernate.id.enhanced.SequenceStyleGenerator 以不同的方式实现移植性。它根据所使用的方言的能力，在数据库表或序列之间选择以存储其增量。这和 native 的区别是基于表或序列的存储具有恰好相同的 semantic。实际上，序列就是 Hibernate 试图用基于表的生成器来模拟的。这个生成器有如下的配置参数：

- sequence_name（可选 — 默认为 hibernate_sequence）：序列或表的名字
- initial_value（可选，默认为 1）：从序列/表里获取的初始值。按照序列创建的术语，这等同于子句 "STARTS WITH"。
- increment_size（可选 - 缺省为 1）：对序列/表的调用应该区分的值。按照序列创建的术语，这等同于子句 "INCREMENT BY"。
- force_table_use（可选 - 缺省为 false）：即使方言可能支持序列，是否也应该强制把表用作后台结构。
- value_column（可选 - 缺省为 next_val）：只和表结构相关，它是用于保存值的字段的名称。
- optimizer (optional - defaults to none): See 第 5.1.2.3.1 节 “标识符生成器的优化”

新生成器的第二个是 org.hibernate.id.enhanced.TableGenerator，它的目的首先是替代 table 生成器，即使它实际上比 org.hibernate.id.MultipleHiLoPerTableGenerator 功能要强得多；其次，作为利用可插拔 optimizer 的 org.hibernate.id.MultipleHiLoPerTableGenerator 的替代品。基本上这个生成器定义了一个可以利用多个不同的键值记录存储大量不同增量值的表。这个生成器有如下的配置参数：

- table_name（可选 — 默认是 hibernate_sequences）：所用的表的名称。
- value_column_name（可选 — 默认为 next_val）：用于存储这些值的表的字段的名字。
- segment_column_name（可选，默认为 sequence_name）：用于保存 "segment key" 的字段的名称。这是标识使用哪个增量值的值。
- segment_value（可选，默认为 default）：我们为这个生成器获取增量值的 segment 的 "segment key"。
- segment_value_length（可选 — 默认为 255）：用于 schema 生成；创建 Segment Key 字段的字段大小。
- initial_value（可选 — 默认是 1）：从表里获取的初始值。
- increment_size（可选 — 默认是 1）：对表随后的调用应该区分的值。
- optimizer (optional - defaults to ??): See 第 5.1.2.3.1 节 “标识符生成器的优化”.

5.1.2.3.1. 标识符生成器的优化

For identifier generators that store values in the database, it is inefficient for them to hit the database on each and every call to generate a new identifier value. Instead, you can group a bunch of them in memory and only hit the database when you have exhausted your in-memory value group. This is the role of the pluggable optimizers. Currently only the two enhanced generators (第 5.1.2.3 节 “增强的标识符生成器” support this operation.

- none（如果没有指定 optimizer，通常这是缺省配置）：这不会执行任何优化，在每次请求时都访问数据库。
- hilo：对从数据库获取的值应用 hi/lo 算法。用于这个 optimizer 的从数据库获取的值应该是有序的。它们表明“组编号”。increment_size 将乘以内存里的值来定义组的“hi 值”。
- pooled：和 hilo 一样，这个 optimizer 试图最小化对数据库的访问。然而，我们只是简单地把“下一组”的起始值而不是把序列值和分组算法的组合存入到数据库结构里。在这里，increment_size 表示数据库里的值。

5.1.2.4. Partial identifier generation

Hibernate supports the automatic generation of some of the identifier properties. Simply use the `@GeneratedValue` annotation on one or several id properties.

警告

The Hibernate team has always felt such a construct as fundamentally wrong. Try hard to fix your data model before using this feature.

```
@Entity
public class CustomerInventory implements Serializable {
  @Id
  @TableGenerator(name = "inventory",
    table = "U_SEQUENCES",
    pkColumnName = "S_ID",
    valueColumnName = "S_NEXTNUM",
    pkColumnValue = "inventory",
    allocationSize = 1000)
  @GeneratedValue(strategy = GenerationType.TABLE, generator = "inventory")
  Integer id;


  @Id @ManyToOne(cascade = CascadeType.MERGE)
  Customer customer;
}

@Entity
public class Customer implements Serializable {
   @Id
   private int id;
}
```

You can also generate properties inside an `@EmbeddedId` class.

5.1.3. Optimistic locking properties (optional)

When using long transactions or conversations that span several database transactions, it is useful to store versioning data to ensure that if the same entity is updated by two conversations, the last to commit changes will be informed and not override the other conversation's work. It guarantees some isolation while still allowing for good scalability and works particularly well in read-often write-sometimes situations.

You can use two approaches: a dedicated version number or a timestamp.

一个脱管（detached）实例的 version 或 timestamp 属性不能为空（null），因为 Hibernate 不管 unsaved-value 被指定为何种策略，它将任何属性为空的 version 或 timestamp 实例看作为瞬时（transient）实例。 避免 Hibernate 中的传递重附（transitive reattachment）问题的

一个简单方法是 定义一个不能为空的 version 或 timestamp 属性，特别是在人们使用程序分配的标识符（assigned identifiers） 或复合主键时非常有用。

5.1.3.1. Version number

You can add optimistic locking capability to an entity using the @Version annotation:

```
@Entity
public class Flight implements Serializable {
...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}
```

The version property will be mapped to the OPTLOCK column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

The version column may be a numeric. Hibernate supports any kind of type provided that you define and implement the appropriate UserVersionType.

The application must not alter the version number set up by Hibernate in any way. To artificially increase the version number, check in Hibernate Entity Manager's reference documentation LockModeType.OPTIMISTIC_FORCE_INCREMENT or LockModeType.PESSIMISTIC_FORCE_INCREMENT.

If the version number is generated by the database (via a trigger for example), make sure to use @org.hibernate.annotations.Generated(GenerationTime.ALWAYS).

To declare a version property in hbm.xml, use:

```
<version
        column="version_column"                                  (1)
        name="propertyName"                                      (2)
        type="typename"                                          (3)
        access="field|property|ClassName"                        (4)
        unsaved-value="null|negative|undefined"                  (5)
        generated="never|always"                                 (6)
        insert="true|false"                                      (7)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) column（可选 — 默认为属性名）：指定持有版本号的字段名。
(2) name：持久化类的属性名。
(3) type（可选 — 默认是 integer）：版本号的类型。

4. access（可选 — 默认为 property）：Hibernate 用来访问属性值的策略。
5. 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 中保存或装载的脱管（detached）实例区分开来。（undefined 指明应被使用的标识属性值。）
6. generated（可选 — 默认是 never）：表明此版本属性值是否实际上是由数据库生成的。请参阅 generated properties 部分的讨论。
7. insert（可选 — 默认是 true）：表明此版本列应该包含在 SQL 插入语句中。只有当数据库字段有默认值 0 的时候，才可以设置为 false。

5.1.3.2. Timestamp

Alternatively, you can use a timestamp. Timestamps are a less safe implementation of optimistic locking. However, sometimes the application might use the timestamps in other ways.

Simply mark a property of type Date or Calendar as @Version.

```
@Entity
public class Flight implements Serializable {
...
    @Version
    public Date getLastUpdate() { ... }
}
```

Like version numbers, the timestamp can be generated by the database instead of Hibernate. To do that, use @org.hibernate.annotations.Generated(GenerationTime.ALWAYS).

In hbm.xml, use the <timestamp> element:

```
<timestamp
        column="timestamp_column"                                    1
        name="propertyName"                                          2
        access="field|property|ClassName"                            3
        unsaved-value="null|undefined"                               4
        source="vm|db"                                               5
        generated="never|always"                                     6
        node="element-name|@attribute-name|element/@attribute|."
/>
```

1. column（可选 — 默认为属性名）：存有时间戳的字段名。
2. 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。
3. access（可选 — 默认为 property）：Hibernate 用来访问属性值的策略。
4. 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 中保存或装载的脱管（detached）实例区分开来。（undefined 指明使用标识属性值进行这种判断。）

5. source（可选 — 默认是 vm）：Hibernate 如何才能获取到时间戳的值呢？从数据库，还是当前 JVM？从数据库获取会带来一些负担，因为 Hibernate 必须访问数据库来获得“下一个值”，但是在集群环境中会更安全些。还要注意，并不是所有的 Dialect（方言）都支持获得数据库的当前时间戳的，而支持的数据库中又有一部分因为精度不足，用于锁定是不安全的（例如 Oracle 8）。
6. generated（可选 - 默认是 never）：指出时间戳值是否实际上是由数据库生成的。请参阅 generated properties 的讨论。

注意

注意，<timestamp> 和 <version type="timestamp"> 是等价的。并且 <timestamp source="db"> 和 <version type="dbtimestamp"> 是等价的。

5.1.4. Property

You need to decide which property needs to be made persistent in a given entity. This differs slightly between the annotation driven metadata and the hbm.xml files.

5.1.4.1. Property mapping with annotations

In the annotations world, every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as @Transient. Not having an annotation for your property is equivalent to the appropriate @Basic annotation.

The @Basic annotation allows you to declare the fetching strategy for a property. If set to LAZY, specifies that this property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation, if your classes are not instrumented, property level lazy loading is silently ignored. The default is EAGER. You can also mark a property as not optional thanks to the @Basic.optional attribute. This will ensure that the underlying column are not nullable (if possible). Note that a better approach is to use the @NotNull annotation of the Bean Validation specification.

Let's look at a few examples:

```
public transient int counter; //transient property

private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property

String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property
```

```
@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

counter, a transient field, and lengthInMeter, a method annotated as @Transient, and will be ignored by the Hibernate. name, length, and firstname properties are mapped persistent and eagerly fetched (the default for simple properties). The detailedComment property value will be lazily fetched from the database once a lazy property of the entity is accessed for the first time. Usually you don't need to lazy simple properties (not to be confused with lazy association fetching). The recommended alternative is to use the projection capability of JP-QL (Java Persistence Query Language) or Criteria queries.

JPA support property mapping of all basic types supported by Hibernate (all basic Java types , their respective wrappers and serializable classes). Hibernate Annotations supports out of the box enum type mapping either into a ordinal column (saving the enum ordinal) or a string based column (saving the enum string representation): the persistence representation, defaulted to ordinal, can be overridden through the @Enumerated annotation as shown in the note property example.

In plain Java APIs, the temporal precision of time is not defined. When dealing with temporal data you might want to describe the expected precision in database. Temporal data can have DATE, TIME, or TIMESTAMP precision (ie the actual date, only the time, or both). Use the @Temporal annotation to fine tune that.

@Lob indicates that the property should be persisted in a Blob or a Clob depending on the property type: java.sql.Clob, Character[], char[] and java.lang.String will be persisted in a Clob. java.sql.Blob, Byte[], byte[] and Serializable type will be persisted in a Blob.

```
@Lob
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
}
```

If the property type implements java.io.Serializable and is not a basic type, and if the property is not annotated with @Lob, then the Hibernate serializable type is used.

5.1.4.1.1. Type

You can also manually specify a type using the @org.hibernate.annotations.Type and some parameters if needed. @Type.type could be:

1. Hibernate 基本类型名（比如：integer, string, character,date, timestamp, float, binary, serializable, object, blob）。
2. 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）。
3. 一个可以序列化的 Java 类的名字。
4. 一个自定义类型的类的名字。（比如：com.i11flow.type.MyCustomType）。

If you do not specify a type, Hibernate will use reflection upon the named property and guess the correct Hibernate type. Hibernate will attempt to interpret the name of the return class of the property getter using, in order, rules 2, 3, and 4.

@org.hibernate.annotations.TypeDef and @org.hibernate.annotations.TypeDefs allows you to declare type definitions. These annotations can be placed at the class or package level. Note that these definitions are global for the session factory (even when defined at the class level). If the type is used on a single entity, you can place the definition on the entity itself. Otherwise, it is recommended to place the definition at the package level. In the example below, when Hibernate encounters a property of class PhoneNumer, it delegates the persistence strategy to the custom mapping type PhoneNumberType. However, properties belonging to other classes, too, can delegate their persistence strategy to PhoneNumberType, by explicitly using the @Type annotation.

注意

Package level annotations are placed in a file named package-info.java in the appropriate package. Place your annotations before the package declaration.

```
@TypeDef(
   name = "phoneNumber",
   defaultForType = PhoneNumber.class,
   typeClass = PhoneNumberType.class
)

@Entity
public class ContactDetails {
   [...]
   private PhoneNumber localPhoneNumber;
   @Type(type="phoneNumber")
   private OverseasPhoneNumber overseasPhoneNumber;
   [...]
}
```

The following example shows the usage of the parameters attribute to customize the TypeDef.

```
//in org/hibernate/test/annotations/entity/package-info.java
@TypeDefs(
    {
    @TypeDef(
        name="caster",
        typeClass = CasterStringType.class,
        parameters = {
            @Parameter(name="cast", value="lower")
        }
    )
    }
)
package org.hibernate.test.annotations.entity;

//in org/hibernate/test/annotations/entity/Forest.java
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

When using composite user type, you will have to express column definitions. The @Columns has been introduced for that purpose.

```
@Type(type="org.hibernate.test.annotations.entity.MonetaryAmountUserType")
@Columns(columns = {
    @Column(name="r_amount"),
    @Column(name="r_currency")
})
public MonetaryAmount getAmount() {
    return amount;
}


public class MonetaryAmount implements Serializable {
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

5.1.4.1.2. Access type

By default the access type of a class hierarchy is defined by the position of the @Id or @EmbeddedId annotations. If these annotations are on a field, then only fields are considered for persistence and the state is accessed via the field. If there annotations are on a getter, then only the getters are considered for persistence and the state is accessed via the getter/setter. That works well in practice and is the recommended approach.

注意

The placement of annotations within a class hierarchy has to be consistent (either field or on property) to be able to determine the default access type. It is recommended to stick to one single annotation placement strategy throughout your whole application.

However in some situations, you need to:

- force the access type of the entity hierarchy
- override the access type of a specific entity in the class hierarchy
- override the access type of an embeddable type

The best use case is an embeddable class used by several entities that might not use the same access type. In this case it is better to force the access type at the embeddable class level.

To force the access type on a given class, use the @Access annotation as showed below:

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   @Embedded private Address address;
   public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Entity
public class User {
   private Long id;
   @Id public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }

   private Address address;
   @Embedded public Address getAddress() { return address; }
   public void setAddress() { this.address = address; }
}

@Embeddable
@Access(AcessType.PROPERTY)
public class Address {
   private String street1;
   public String getStreet1() { return street1; }
   public void setStreet1() { this.street1 = street1; }

   private hashCode; //not persistent
}
```

You can also override the access type of a single property while keeping the other properties standard.

```
@Entity
public class Order {
   @Id private Long id;
   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   @Transient private String userId;
   @Transient private String orderId;

   @Access(AccessType.PROPERTY)
   public String getOrderNumber() { return userId + ":" + orderId; }
   public void setOrderNumber() { this.userId = ...; this.orderId = ...; }
}
```

In this example, the default access type is FIELD except for the orderNumber property. Note that the corresponding field, if any must be marked as @Transient or transient.

@org.hibernate.annotations.AccessType

The annotation @org.hibernate.annotations.AccessType should be considered deprecated for FIELD and PROPERTY access. It is still useful however if you need to use a custom access type.

5.1.4.1.3. Optimistic lock

It is sometimes useful to avoid increasing the version number even if a given property is dirty (particularly collections). You can do that by annotating the property (or collection) with @OptimisticLock(excluded=true).

More formally, specifies that updates to this property do not require acquisition of the optimistic lock.

5.1.4.1.4. Declaring column attributes

The column(s) used for a property mapping can be defined using the @Column annotation. Use it to override default values (see the JPA specification for more information on the defaults). You can use this annotation at the property level for properties that are:

- not annotated at all
- annotated with @Basic
- annotated with @Version
- annotated with @Lob
- annotated with @Temporal

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

The name property is mapped to the flight_name column, which is not nullable, has a length of 50 and is not updatable (making the property immutable).

This annotation can be applied to regular properties as well as @Id or @Version properties.

```
@Column(
    name="columnName";                                          (1)
    boolean unique() default false;                             (2)
    boolean nullable() default true;                            (3)
    boolean insertable() default true;                          (4)
    boolean updatable() default true;                           (5)
    String columnDefinition() default "";                       (6)
    String table() default "";                                  (7)
    int length() default 255;                                   (8)
    int precision() default 0; // decimal precision             (9)
    int scale() default 0; // decimal scale
```

1. name (optional): the column name (default to the property name)
2. unique (optional): set a unique constraint on this column or not (default false)
3. nullable (optional): set the column as nullable (default true).
4. insertable (optional): whether or not the column will be part of the insert statement (default true)
5. updatable (optional): whether or not the column will be part of the update statement (default true)
6. columnDefinition (optional): override the sql DDL fragment for this particular column (non portable)
7. table (optional): define the targeted table (default primary table)
8. length (optional): column length (default 255)
8. precision (optional): column decimal precision (default 0)
10. scale (optional): column decimal scale if useful (default 0)

5.1.4.1.5. Formula

Sometimes, you want the Database to do some computation for you rather than in the JVM, you might also create some kind of virtual column. You can use a SQL fragment (aka

formula) instead of mapping a property into a column. This kind of property is read only (its value is calculated by your formula fragment).

```
@Formula("obj_length * obj_height * obj_width")
public long getObjectVolume()
```

The SQL fragment can be as complex as you want and even include subselects.

5.1.4.1.6. Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as @Basic
- Otherwise, if the type of the property is annotated as @Embeddable, it is mapped as @Embedded
- Otherwise, if the type of the property is Serializable, it is mapped as @Basic in a column holding the object in its serialized version
- Otherwise, if the type of the property is java.sql.Clob or java.sql.Blob, it is mapped as @Lob with the appropriate LobType

5.1.4.2. Property mapping with hbm.xml

<property> 元素为类定义了一个持久化的、JavaBean 风格的属性。

```
<property
        name="propertyName"                                          1
        column="column_name"                                         2
        type="typename"                                              3
        update="true|false"                                          4
        insert="true|false"                                          4
        formula="arbitrary SQL expression"                           5
        access="field|property|ClassName"                            6
        lazy="true|false"                                            7
        unique="true|false"                                          8
        not-null="true|false"                                        9
        optimistic-lock="true|false"                                 10
        generated="never|insert|always"                              11
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
```

```
/>
```

1. name：属性的名字，以小写字母开头。
2. column（可选 — 默认为属性名字）：对应的数据库字段名。 也可以通过嵌套的 <column> 元素指定。
3. type（可选）：一个 Hibernate 类型的名字。
4. update，insert（可选 — 默认为 true）： 表明用于 UPDATE 和/或 INSERT 的 SQL 语句中是否包含这个被映射了的字段。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）或其他程序生成。
5. 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。
6. access（可选 — 默认为 property）：Hibernate 用来访问属性值的策略。
7. lazy（可选 — 默认为 false）：指定 指定实例变量第一次被访问时，这个属性是否延迟抓取（fetched lazily） （ 需要运行时字节码增强）。
8. unique（可选）：使用 DDL 为该字段添加唯一的约束。同样，允许它作为 property-ref 引用的目标。
9. not-null（可选）：使用 DDL 为该字段添加可否为空（nullability）的约束。
10. optimistic-lock（可选 — 默认为 true）：指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。
11. generated（可选 — 默认为 never）：表明此属性值是否实际上是由数据库生成的。请参阅 generated properties 的讨论。

typename 可以是如下几种：

1. Hibernate 基本类型名（比如：integer，string，character,date，timestamp，float，binary，serializable，object，blob）。
2. 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）。
3. 一个可以序列化的 Java 类的名字。
4. 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 会按照规则 2，3，4 的顺序对属性读取器（getter方法）的返回类进行解释。然而，这还不够。 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（pair）。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 org.hibernate.property.PropertyAccessor 接口，再在 access 中设置你自定义策略类的名字。

衍生属性（derive propertie）是一个特别强大的特征。这些属性应该定义为只读，属性值在装载时计算生成。 你用一个 SQL 表达式生成计算的结果，它会在这个实例转载时翻译成一个 SQL 查询的 SELECT 子查询语句。

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

注意，你可以使用实体自己的表，而不用为这个特别的列定义别名（上面例子中的 customerId）。同时注意，如果你不喜欢使用属性， 你可以使用嵌套的 <formula> 映射元素。

5.1.5. Embedded objects (aka components)

Embeddable objects (or components) are objects whose properties are mapped to the same table as the owning entity's table. Components can, in turn, declare their own properties, components or collections

It is possible to declare an embedded component inside an entity and even override its column mapping. Component classes have to be annotated at the class level with the @Embeddable annotation. It is possible to override the column mapping of an embedded object for a particular entity using the @Embedded and @AttributeOverride annotation in the associated property:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
            @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    } )
    Country bornIn;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
    String city;
    Country nationality; //no overriding here
}
```

```
@Embeddable
public class Country implements Serializable {
    private String iso2;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
```

```
    public void setIso2(String iso2) { this.iso2 = iso2; }


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

An embeddable object inherits the access type of its owning entity (note that you can override that using the @Access annotation).

The Person entity has two component properties, homeAddress and bornIn. homeAddress property has not been annotated, but Hibernate will guess that it is a persistent component by looking for the @Embeddable annotation in the Address class. We also override the mapping of a column name (to bornCountryName) with the @Embedded and @AttributeOverride annotations for each mapped attribute of Country. As you can see, Country is also a nested component of Address, again using auto-detection by Hibernate and JPA defaults. Overriding columns of embedded objects of embedded objects is through dotted expressions.

```
@Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="city", column = @Column(name="fld_city") ),
            @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
         @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
            //nationality columns in homeAddress are overridden
    } )
    Address homeAddress;
```

Hibernate Annotations supports something that is not explicitly supported by the JPA specification. You can annotate a embedded object with the @MappedSuperclass annotation to make the superclass properties persistent (see @MappedSuperclass for more informations).

You can also use association annotations in an embeddable object (ie @OneToOne, @ManyToOne, @OneToMany or @ManyToMany). To override the association columns you can use @AssociationOverride.

If you want to have the same embeddable object type twice in the same entity, the column name defaulting will not work as several embedded objects would share the same set of columns. In plain JPA, you need to override at least one set of columns. Hibernate, however, allows you to enhance the default naming mechanism through the NamingStrategy interface. You can write a strategy that prevent name clashing in such a situation. DefaultComponentSafeNamingStrategy is an example of this.

If a property of the embedded object points back to the owning entity, annotate it with the @Parent annotation. Hibernate will make sure this property is properly loaded with the entity reference.

In XML, use the <component> element.

```
<component
        name="propertyName"                                    ❶
        class="className"                                      ❷
        insert="true|false"                                    ❸
        update="true|false"                                    ❹
        access="field|property|ClassName"                      ❺
        lazy="true|false"                                      ❻
        optimistic-lock="true|false"                           ❼
        unique="true|false"                                    ❽
        node="element-name|."
>

        <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 — 默认为 property）：Hibernate 用来访问属性值的策略。
❻ lazy（可选 — 默认是 false）：表明此组件应在实例变量第一次被访问的时候延迟加载（需要编译时字节码装置器）。
❼ optimistic-lock（可选 — 默认是 true）：表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号（Version）。
❽ unique（可选 — 默认是 false）：表明组件映射的所有字段上都有唯一性约束。

其 <property> 子标签为子类的一些属性与表字段之间建立映射。

<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。

The <dynamic-component> element allows a Map to be mapped as a component, where the property names refer to keys of the map. See 第 9.5 节 “动态组件（Dynamic components）” for more information. This feature is not supported in annotations.

5.1.6. Inheritance strategy

Java is a language supporting polymorphism: a class can inherit from another. Several strategies are possible to persist a class hierarchy:

- Single table per class hierarchy strategy: a single table hosts all the instances of a class hierarchy

- Joined subclass strategy: one table per class and subclass is present and each table persist the properties specific to a given subclass. The state of the entity is then stored in its corresponding class table and all its superclasses
- Table per class strategy: one table per concrete class and subclass is present and each table persist the properties of the class and its superclasses. The state of the entity is then stored entirely in the dedicated table for its class.

5.1.6.1. Single table per class hierarchy strategy

With this approach the properties of all the subclasses in a given mapped class hierarchy are stored in a single table.

Each subclass declares its own persistent properties and subclasses. Version and id properties are assumed to be inherited from the root class. Each subclass in a hierarchy must define a unique discriminator value. If this is not specified, the fully qualified Java class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, for the table-per-class-hierarchy mapping strategy, the <subclass> declaration is used. For example:

```
<subclass
        name="ClassName"                                    1
        discriminator-value="discriminator_value"           2
        proxy="ProxyInterface"                              3
        lazy="true|false"                                   4
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

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

1. name：子类的全限定名。
2. 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。
3. proxy（可选）：指定一个类或者接口，在延迟装载时作为代理使用。
4. lazy（可选，默认是 true）：设置为 lazy="false" 禁止使用延迟装载。

For information about inheritance mappings see 第 10 章 继承映射（Inheritance Mapping）.

5.1.6.1.1. 鉴别器（discriminator）

Discriminators are required for polymorphic persistence using the table-per-class-hierarchy mapping strategy. It declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. A restricted set of types can be used: string, character, integer, byte, short, boolean, yes_no, true_false.

Use the @DiscriminatorColumn to define the discriminator column as well as the discriminator type. Alternatively, you can also use @DiscriminatorFormula to express in SQL what would be in a virtual discriminator column. This is particularly handy when the discriminator value can be extracted from one or more columns of the table. Both @DiscriminatorColumn and @DiscriminatorFormula are to be set on the root entity (once per persisted hierarchy).

Finally, use @DiscriminatorValue on each class of the hierarchy to specify the value stored in the discriminator column for a given entity. If you do not set @DiscriminatorValue on a class, the fully qualified class name is used.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

In hbm.xml, the <discriminator> element is used to define the discriminator column or formula:

```
<discriminator
        column="discriminator_column"                    (1)
        type="discriminator_type"                        (2)
```

```
        force="true|false"                                    ❸
        insert="true|false"                                   ❹
        formula="arbitrary sql expression"                    ❺
/>
```

❶ column（可选 — 默认为 class）discriminator 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 指定允许的鉴别器值,即使当取得的所有实例都是根类的。

❹ insert（可选 - 默认为true）如果你的鉴别器字段也是映射为复合标识（composite identifier）的一部分，则需将这个值设为 false。（告诉 Hibernate 在做 SQL INSERT 时不包含该列）

❺ formula（可选）一个 SQL 表达式，在类型判断（判断是父类还是具体子类 — 译注）时执行。可用于基于内容的鉴别器。

鉴别器字段的实际值是根据 <class> 和 <subclass> 元素中的 discriminator-value 属性得来的。

force 属性仅仅在这种情况下有用的：表中包含没有被映射到持久化类的附加辨别器值。这种情况不会经常遇到。

使用 formula 属性你可以定义一个 SQL 表达式，用来判断一行数据的类型。

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

5.1.6.2. Joined subclass strategy

Each subclass can also be mapped to its own table. This is called the table-per-subclass mapping strategy. An inherited state is retrieved by joining with the table of the superclass. A discriminator column is not required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier. The primary key of this table is also a foreign key to the superclass table and described by the @PrimaryKeyJoinColumns or the <key> element.

```
@Entity @Table(name="CATS")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cat implements Serializable {
    @Id @GeneratedValue(generator="cat-uuid")
    @GenericGenerator(name="cat-uuid", strategy="uuid")
    String getId() { return id; }

    ...
}

@Entity @Table(name="DOMESTIC_CATS")
@PrimaryKeyJoinColumn(name="CAT")
```

```
public class DomesticCat extends Cat {
    public String getName() { return name; }
}
```

注意

The table name still defaults to the non qualified class name. Also if @PrimaryKeyJoinColumn is not set, the primary key / foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass.

In hbm.xml, use the <joined-subclass> element. For example:

```
<joined-subclass
        name="ClassName"                                    (1)
        table="tablename"                                   (2)
        proxy="ProxyInterface"                              (3)
        lazy="true|false"                                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

1. name：子类的全限定名。
2. table：子类的表名。
3. proxy（可选）：指定一个类或者接口，在延迟装载时作为代理使用。
4. lazy（可选，默认是 true）：设置为 lazy="false" 禁止使用延迟装载。

Use the <key> element to declare the primary key / foreign key column. The mapping at the start of the chapter would then be re-written as:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.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>
```

For information about inheritance mappings see 第 10 章 继承映射 (Inheritance Mapping) .

5.1.6.3. Table per class strategy

A third option is to map only the concrete classes of an inheritance hierarchy to tables. This is called the table-per-concrete-class strategy. Each table defines all persistent states of the class, including the inherited state. In Hibernate, it is not necessary to explicitly map such inheritance hierarchies. You can map each class as a separate entity root. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable { ... }
```

Or in hbm.xml:

```
<union-subclass
        name="ClassName"                                    ❶
        table="tablename"                                   ❷
        proxy="ProxyInterface"                              ❸
```

```
        lazy="true|false"                                                    ❹
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

❶ name：子类的全限定名。
❷ table：子类的表名。
❸ proxy（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（可选，默认是 true）：设置为 lazy="false" 禁止使用延迟装载。

这种映射策略不需要指定辨别标志（discriminator）字段。

For information about inheritance mappings see 第 10 章 继承映射（Inheritance Mapping）.

5.1.6.4. Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity). For that purpose you can map them as @MappedSuperclass.

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}

@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

In database, this hierarchy will be represented as an Order table having the id, lastUpdate and lastUpdater columns. The embedded superclass property mappings are copied into their entity subclasses. Remember that the embeddable superclass is not the root of the hierarchy though.

注意

Properties from superclasses not mapped as @MappedSuperclass are ignored.

注意

The default access type (field or methods) is used, unless you use the @Access annotation.

注意

The same notion can be applied to @Embeddable objects to persist properties from their superclasses. You also need to use @MappedSuperclass to do that (this should not be considered as a standard EJB3 feature though)

注意

It is allowed to mark a class as @MappedSuperclass in the middle of the mapped inheritance hierarchy.

注意

Any class in the hierarchy non annotated with @MappedSuperclass nor @Entity will be ignored.

You can override columns defined in entity superclasses at the root entity level using the @AttributeOverride annotation.

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

    @Transient
    public int getMetricAltitude() {
        return metricAltitude;
    }

    @ManyToOne
    public PropulsionType getPropulsion() {
```

```
        return metricAltitude;
    }
    ...
}

@Entity
@AttributeOverride( name="altitude", column = @Column(name="fld_altitude") )
@AssociationOverride(
   name="propulsion",
   joinColumns = @JoinColumn(name="fld_propulsion_fk")
)
public class Plane extends FlyingObject {
    ...
}
```

The altitude property will be persisted in an fld_altitude column of table Plane and the propulsion association will be materialized in a fld_propulsion_fk foreign key column.

You can define @AttributeOverride(s) and @AssociationOverride(s) on @Entity classes, @MappedSuperclass classes and properties pointing to an @Embeddable object.

In hbm.xml, simply map the properties of the superclass in the <class> element of the entity that needs to inherit them.

5.1.6.5. Mapping one entity to several tables

While not recommended for a fresh schema, some legacy databases force your to map a single entity on several tables.

Using the @SecondaryTable or @SecondaryTables class level annotations. To express that a column is in a particular table, use the table parameter of @Column or @JoinColumn.

```
@Entity
@Table(name="MainCat")
@SecondaryTables({
    @SecondaryTable(name="Cat1", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")
    ),
    @SecondaryTable(name="Cat2", uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})
})
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
```

```
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In this example, name will be in MainCat. storyPart1 will be in Cat1 and storyPart2 will be in Cat2. Cat1 will be joined to MainCat using the cat_id as a foreign key, and Cat2 using id (ie the same column name, the MainCat id column has). Plus a unique constraint on storyPart2 has been set.

There is also additional tuning accessible via the @org.hibernate.annotations.Table annotation:

- fetch: If set to JOIN, the default, Hibernate will use an inner join to retrieve a secondary table defined by a class or its superclasses and an outer join for a secondary table defined by a subclass. If set to SELECT then Hibernate will use a sequential select for a secondary table defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a secondary defined by the class and its superclasses.
- inverse: If true, Hibernate will not try to insert or update the properties defined by this join. Default to false.
- optional: If enabled (the default), Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.
- foreignKey: defines the Foreign Key name of a secondary table pointing back to the primary table.

Make sure to use the secondary table name in the appliesto property

```
@Entity
@Table(name="MainCat")
@SecondaryTable(name="Cat1")
@org.hibernate.annotations.Table(
   appliesTo="Cat1",
   fetch=FetchMode.SELECT,
   optional=true)
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
```

```
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
}
```

In hbm.xml, use the <join> element.

```
<join
        table="tablename"                    (1)
        schema="owner"                       (2)
        catalog="catalog"                    (3)
        fetch="join|select"                  (4)
        inverse="true|false"                 (5)
        optional="true|false">               (6)

        <key ... />

        <property ... />
        ...
</join>
```

1. table：被连接表的名称。
2. schema（可选）：覆盖在根 <hibernate-mapping> 元素中指定的 schema 名字。
3. catalog（可选）：覆盖在根 <hibernate-mapping> 元素中指定的 catalog 名字。
4. fetch（可选 — 默认是 join）：如果设置为默认值 join，Hibernate 将使用一个内连接来得到这个类或其超类定义的 <join>，而使用一个外连接来得到其子类定义的 <join>。如果设置为 select，则 Hibernate 将为子类定义的 <join> 使用顺序选择。这仅在一行数据表示一个子类的对象的时候才会发生。对这个类和其超类定义的 <join>，依然会使用内连接得到。
5. inverse（可选 — 默认是 false）：如果打开，Hibernate 不会插入或者更新此连接定义的属性。
6. optional（可选 — 默认是 false）：如果打开，Hibernate 只会在此连接定义的属性非空时插入一行数据，并且总是使用一个外连接来得到这些属性。

例如，一个人（person）的地址（address）信息可以被映射到单独的表中（并保留所有属性的值类型语义）：

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

此特性常常对遗留数据模型有用，我们推荐表个数比类个数少，以及细粒度的领域模型。然而，在单独的继承树上切换继承映射策略是有用的，后面会解释这点。

5.1.7. Mapping one to one and one to many associations

To link one entity to an other, you need to map the association property as a to one association. In the relational model, you can either use a foreign key or an association table, or (a bit less common) share the same primary key value between the two entities.

To mark an association, use either @ManyToOne or @OnetoOne.

@ManyToOne and @OneToOne have a parameter named targetEntity which describes the target entity name. You usually don't need this parameter since the default value (the type of the property that stores the association) is good in almost all cases. However this is useful when you want to use interfaces as the return type instead of the regular entity.

Setting a value of the cascade attribute to any meaningful value other than nothing will propagate certain operations to the associated object. The meaningful values are divided into three categories.

1. basic operations, which include: persist, merge, delete, save-update, evict, replicate, lock and refresh;
2. special values: delete-orphan or all ;
3. comma-separated combinations of operation names: cascade="persist,merge,evict" or cascade="all,delete-orphan". See 第 11.11 节 “传播性持久化（transitive persistence）” for a full explanation. Note that single valued many-to-one associations do not support orphan delete.

By default, single point associations are eagerly fetched in JPA 2. You can mark it as lazily fetched by using @ManyToOne(fetch=FetchType.LAZY) in which case Hibernate will proxy the association and load it when the state of the associated entity is reached.

You can force Hibernate not to use a proxy by using @LazyToOne(NO_PROXY). In this case, the property is fetched lazily when the instance variable is first accessed. This requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched.

With the default JPA options, single-ended associations are loaded with a subsequent select if set to LAZY, or a SQL JOIN is used for EAGER associations. You can however adjust the fetching strategy, ie how data is fetched by using @Fetch. FetchMode can be SELECT (a select is triggered when the association needs to be loaded) or JOIN (use a SQL JOIN to load the association while loading the owner entity). JOIN overrides any lazy attribute (an association loaded through a JOIN strategy cannot be lazy).

5.1.7.1. Using a foreign key or an association table

An ordinary association to another persistent class is declared using a

- @ManyToOne if several entities can point to the the target entity
- @OneToOne if only a single entity can point to the the target entity

and a foreign key in one table is referencing the primary key column(s) of the target table.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

The @JoinColumn attribute is optional, the default value(s) is the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column in the owned side. In this example company_id because the property name is company and the column id of Company is id.

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}

public interface Company {
```

```
    ...
}
```

You can also map a to one association through an association table. This association table described by the @JoinTable annotation will contains a foreign key referencing back the entity table (through @JoinTable.joinColumns) and a a foreign key referencing the target entity table (through @JoinTable.inverseJoinColumns).

```
@Entity
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinTable(name="Flight_Company",
        joinColumns = @JoinColumn(name="FLIGHT_ID"),
        inverseJoinColumns = @JoinColumn(name="COMP_ID")
    )
    public Company getCompany() {
        return company;
    }
    ...
}
```

注意

You can use a SQL fragment to simulate a physical join column using the @JoinColumnOrFormula / @JoinColumnOrformulas annotations (just like you can use a SQL fragment to simulate a property column via the @Formula annotation).

```
@Entity
public class Ticket implements Serializable {
    @ManyToOne
    @JoinColumnOrFormula(formula="(firstname + ' ' + lastname)")
    public Person getOwner() {
        return person;
    }
    ...
}
```

You can mark an association as mandatory by using the optional=false attribute. We recommend to use Bean Validation's @NotNull annotation as a better alternative however. As a consequence, the foreign key column(s) will be marked as not nullable (if possible).

When Hibernate cannot resolve the association because the expected associated element is not in database (wrong id on the association column), an exception is raised. This might be inconvenient for legacy and badly maintained schemas. You can ask Hibernate to ignore such elements instead of raising an exception using the @NotFound annotation.

例 5.1. @NotFound annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    public Parent getParent() { ... }
    ...
}
```

Sometimes you want to delegate to your database the deletion of cascade when a given entity is deleted. In this case Hibernate generates a cascade delete constraint at the database level.

例 5.2. @OnDelete annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @OnDelete(action=OnDeleteAction.CASCADE)
    public Parent getParent() { ... }
    ...
}
```

Foreign key constraints, while generated by Hibernate, have a fairly unreadable name. You can override the constraint name using @ForeignKey.

例 5.3. @ForeignKey annotation

```
@Entity
public class Child {
    ...
    @ManyToOne
    @ForeignKey(name="FK_PARENT")
    public Parent getParent() { ... }
    ...
}

alter table Child add constraint FK_PARENT foreign key (parent_id) references Parent
```

Sometimes, you want to link one entity to an other not by the target entity primary key but by a different unique key. You can achieve that by referencing the unique key column(s) in @JoinColumn.referenceColumnName.

```
@Entity
```

```
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
      @JoinColumn(name="first_name", referencedColumnName="firstName"),
      @JoinColumn(name="init", referencedColumnName="I"),
      @JoinColumn(name="last_name", referencedColumnName="lastName"),
   })
   Person owner
}
```

This is not encouraged however and should be reserved to legacy mappings.

In hbm.xml, mapping an association is similar. The main difference is that a `@OneToOne` is mapped as `<many-to-one unique="true"/>`, let's dive into the subject.

```
<many-to-one
        name="propertyName"                                          1
        column="column_name"                                         2
        class="ClassName"                                            3
        cascade="cascade_style"                                      4
        fetch="join|select"                                          5
        update="true|false"                                          6
        insert="true|false"                                          6
        property-ref="propertyNameFromAssociatedClass"               7
        access="field|property|ClassName"                            8
        unique="true|false"                                          9
        not-null="true|false"                                        10
        optimistic-lock="true|false"                                 11
        lazy="proxy|no-proxy|false"                                  12
        not-found="ignore|exception"                                 13
        entity-name="EntityName"                                     14
        formula="arbitrary SQL expression"                           15
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

1. name：属性名。
2. column（可选）：外键字段的名称。也可以通过嵌套的 <column> 指定。
3. 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。
4. cascade（级联）（可选）表明操作是否从父对象级联到被关联的对象。
5. fetch（可选 — 默认为 select）：在外连接抓取（outer-join fetching）和序列选择抓取（sequential select fetching）两者中选择其一。
6. update, insert（可选 — 默认为 true）指定对应的字段是否包含在用于 UPDATE 和/或 INSERT 的 SQL 语句中。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 或者通过 trigger（触发器）、或其他程序生成。
7. property-ref：（可选）被关联到此外键的类中的对应属性的名字。如果没有指定，被关联类的主键将被使用。
8. access（可选 — 默认为 property）：Hibernate 用来访问属性值的策略。
9. unique（可选）：使用 DDL 为外键字段生成一个唯一约束。此外， 这也可以用作 property-ref 的目标属性。这使关联同时具有一对一的效果。
10. not-null（可选）：使用 DDL 为外键字段生成一个非空约束。
11. optimistic-lock（可选 — 默认为 true）：指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。
12. lazy（可选 — 默认为 proxy）：默认情况下，单点关联是经过代理的。lazy="no-proxy" 指定此属性应该在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节码的增强）。lazy="false" 指定此关联总是被预先抓取。
13. not-found（可选 - 默认为exception）：指定如何处理引用缺失行的外键：ignore 会把缺失的行作为一个空关联处理。
14. entity-name（可选）：被关联的类的实体名。
15. formula（可选）：SQL 表达式，用于定义 computed（计算出的）外键值。

Setting a value of the cascade attribute to any meaningful value other than none will propagate certain operations to the associated object. The meaningful values are divided into three categories. First, basic operations, which include: persist, merge, delete, save-update, evict, replicate, lock and refresh; second, special values: delete-orphan; and third,all comma-separated combinations of operation names: cascade="persist,merge,evict" or cascade="all,delete-orphan". See 第 11.11 节 “传播性持久化（transitive persistence）” for a full explanation. Note that single valued, many-to-one and one-to-one, associations do not support orphan delete.

一个典型的简单 many-to-one 定义例子：

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

property-ref 属性只应该用来对付遗留下来的数据库系统，可能有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）的情况下。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 工具进行的 DDL 生成。）

```
<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。

如果被引用的唯一主键由关联实体的多个属性组成，你应该在名称为 <properties> 的元素 里面映射所有关联的属性。

假若被引用的唯一主键是组件的属性，你可以指定属性路径：

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

5.1.7.2. Sharing the primary key with the associated entity

The second approach is to ensure an entity and its associated entity share the same primary key. In this case the primary key column is also a foreign key and there is no extra column. These associations are always one to one.

例 5.4. One to One association

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @MapsId
    public Heart getHeart() {
        return heart;
    }
    ...
}

@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

注意

Many people got confused by these primary key based one to one associations. They can only be lazily loaded if Hibernate knows that the other side of the association is always present. To indicate to Hibernate that it is the case, use @OneToOne(optional=false).

In hbm.xml, use the following mapping.

```
<one-to-one
        name="propertyName"                                          (1)
        class="ClassName"                                            (2)
        cascade="cascade_style"                                      (3)
        constrained="true|false"                                     (4)
        fetch="join|select"                                          (5)
        property-ref="propertyNameFromAssociatedClass"               (6)
        access="field|property|ClassName"                            (7)
        formula="any SQL expression"                                 (8)
        lazy="proxy|no-proxy|false"                                  (9)
        entity-name="EntityName"                                     (10)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

1. name：属性名。
2. 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。
3. cascade（级联）（可选）表明操作是否从父对象级联到被关联的对象。
4. 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以及决定该关联能否被委托（也在 schema export tool 中被使用）。
5. fetch（可选 — 默认为 select）：在外连接抓取（outer-join fetching）和序列选择抓取（sequential select fetching）两者中选择其一。
6. property-ref：（可选）指定关联类的属性名，这个属性将会和本类的主键相对应。如果没有指定，会使用对方关联类的主键。
7. access（可选 — 默认为 property）：Hibernate 用来访问属性值的策略。
8. formula （可选）：绝大多数一对一的关联都指向其实体的主键。在一些少见的情况中，你可能会指向其他的一个或多个字段，或者是一个表达式，这些情况下，你可以用一个 SQL 公式来表示。（可以在 org.hibernate.test.onetooneformula 找到例子）
9. lazy（可选 — 默认为 proxy）：默认情况下，单点关联是经过代理的。lazy="no-proxy"指定此属性应该在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节

码的增强）。 lazy="false"指定此关联总是被预先抓取。注意，如果constrained="false"，不可能使用代理，Hibernate会采取预先抓取。

10. entity-name（可选）：被关联的类的实体名。

主键关联不需要额外的表字段；如果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"/>
```

现在我们必须确保 PERSON 和 EMPLOYEE 中相关的字段是相等的。我们使用一个被成为 foreign 的特殊的 hibernate 标识符生成策略：

```
<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。

5.1.8. 自然 ID (natural-id)

Although we recommend the use of surrogate keys as primary keys, you should try to identify natural keys for all entities. A natural key is a property or combination of properties that is unique and non-null. It is also immutable. Map the properties of the natural key as @NaturalId or map them inside the <natural-id> element. Hibernate will generate the necessary unique key and nullability constraints and, as a result, your mapping will be more self-documenting.

```
@Entity
public class Citizen {
    @Id
    @GeneratedValue
```

```
    private Integer id;
    private String firstname;
    private String lastname;

    @NaturalId
    @ManyToOne
    private State state;

    @NaturalId
    private String ssn;
    ...
}



//and later on query
List results = s.createCriteria( Citizen.class )
                .add( Restrictions.naturalId().set( "ssn", "1234" ).set( "state", ste ) )
                .list();
```

Or in XML,

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

我们强烈建议你实现 equals() 和 hashCode() 方法,来比较实体的自然键属性。

这一映射不是为了把自然键作为主键而准备的。

- mutable（可选，默认为 false）：默认情况下，自然标识属性被假定为不可变的（常量）。

5.1.9. Any

There is one more type of property mapping. The @Any mapping defines a polymorphic association to classes from multiple tables. This type of mapping requires more than one column. The first column contains the type of the associated entity. The remaining columns contain the identifier. It is impossible to specify a foreign key constraint for this kind of association. This is not the usual way of mapping polymorphic associations and you should use this only in special cases. For example, for audit logs, user session data, etc.

The @Any annotation describes the column holding the metadata information. To link the value of the metadata information and an actual entity type, The @AnyDef and @AnyDefs annotations are used. The metaType attribute allows the application to specify a custom type that maps database column values to persistent classes that have identifier properties of the type specified by idType. You must specify the mapping from values of the metaType to class names.

```
@Any( metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
@AnyMetaDef(
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
@JoinColumn( name = "property_id" )
public Property getMainProperty() {
    return mainProperty;
}
```

Note that @AnyDef can be mutualized and reused. It is recommended to place it as a package metadata in this case.

```
//on a package
@AnyMetaDef( name="property"
    idType = "integer",
    metaType = "string",
    metaValues = {
        @MetaValue( value = "S", targetEntity = StringProperty.class ),
        @MetaValue( value = "I", targetEntity = IntegerProperty.class )
    } )
package org.hibernate.test.annotations.any;


//in a class
    @Any( metaDef="property", metaColumn = @Column( name = "property_type" ), fetch=FetchType.EAGER )
    @JoinColumn( name = "property_id" )
    public Property getMainProperty() {
        return mainProperty;
    }
```

The hbm.xml equivalent is:

```
<any name="being" 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>
```

注意

You cannot mutualize the metadata in hbm.xml as you can in annotations.

```
<any
        name="propertyName"                    (1)
        id-type="idtypename"                   (2)
        meta-type="metatypename"               (3)
        cascade="cascade_style"                (4)
        access="field|property|ClassName"      (5)
        optimistic-lock="true|false"           (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) name：属性名
(2) id-type：标识符类型
(3) meta-type（可选 -默认是 string）：允许辨别标志(discriminator)映射的任何类型。
(4) cascade（可选 — 默认是none）：级联的类型。
(5) access（可选 — 默认为 property）：Hibernate 用来访问属性值的策略。
(6) optimistic-lock（可选 — 默认是 true）：表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号（Version）。

5.1.10. 属性（Properties）

<properties> 元素允许定义一个命名的逻辑分组（grouping）包含一个类中的多个属性。这个元素最重要的用处是允许多个属性的组合作为 property-ref 的目标（target）。这也是定义多字段唯一约束的一种方便途径。例如：

```
<properties
        name="logicalName"                     (1)
        insert="true|false"                    (2)
        update="true|false"                    (3)
        optimistic-lock="true|false"           (4)
        unique="true|false"                    (5)
>

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

(1) name：分组的逻辑名称 — 不是 实际属性的名称。

2. insert：被映射的字段是否出现在 SQL 的 INSERT 语句中？
3. update：被映射的字段是否出现在 SQL 的 UPDATE 语句中？
4. optimistic-lock（可选 — 默认是 true）：表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号（Version）。
5. unique（可选 — 默认是 false）：表明组件映射的所有字段上都有唯一性约束。

例如，如果我们有如下的 <properties> 映射：

```
<class name="Person">
    <id name="personNumber"/>

    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

然后，我们可能有一些遗留的数据关联，引用 Person 表的这个唯一键，而不是主键：

```
<many-to-one name="owner"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

注意

When using annotations as a mapping strategy, such construct is not necessary as the binding between a column and its related column on the associated table is done directly

```
@Entity
class Person {
   @Id Integer personNumber;
   String firstName;
   @Column(name="I")
   String initial;
   String lastName;
}

@Entity
class Home {
   @ManyToOne
   @JoinColumns({
```

```
        @JoinColumn(name="first_name", referencedColumnName="firstName"),
        @JoinColumn(name="init", referencedColumnName="I"),
        @JoinColumn(name="last_name", referencedColumnName="lastName"),
    })
    Person owner
}
```

我们并不推荐这样使用，除非在映射遗留数据的情况下。

5.1.11. Some hbm.xml specificities

The hbm.xml structure has some specificities naturally not present when using annotations, let's describe them briefly.

5.1.11.1. 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，也可以从 hibernate-x.x.x/src/org/hibernate 目录中、或 hibernate.jar 文件中找到。Hibernate 总是会首先在它的 classptah 中搜索 DTD 文件。如果你发现它是通过连接 Internet 查找 DTD 文件，就对照你的 classpath 目录检查 XML 文件里的 DTD 声明。

5.1.11.1.1. EntityResolver

Hibernate 首先试图在其 classpath 中解析 DTD。这是依靠在系统中注册的 org.xml.sax.EntityResolver 的一个具体实现，SAXReader 依靠它来读取 xml 文件。这个自定义的 EntityResolver 能辨认两种不同的 systenId 命名空间：

- 若 resolver 遇到了一个以 http://hibernate.sourceforge.net/ 为开头的 systemId，它会辨认出是 hibernate namespace，resolver 就试图通过加载 Hibernate 类的 classloader 来查找这些实体。
- 若 resolver 遇到了一个使用 classpath:// URL 协议的 systemId，它会辨认出这是 user namespace，resolver 试图通过（1）当前线程上下文的 classloader 和（2）加载 Hibernate class 的 classloader 来查找这些实体。

下面是一个使用用户命名空间（user namespace）的例子：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC '-//Hibernate/Hibernate Mapping DTD 3.0//EN' 'http://
hibernate.sourceforge.net/hibernate-mapping-3.0.dtd' [
<!ENTITY version "3.6.0.Beta2">
<!ENTITY today "August 4, 2010">

    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">


]>
```

```
<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

这里的 types.xml 是 your.domain 包中的一个资源，它包含了一个自定义的 typedef。

5.1.11.2. Hibernate-mapping

这个元素包括一些可选的属性。schema 和 catalog 属性， 指明了这个映射所连接（refer）的表所在的 schema 和/或 catalog 名称。假若指定了这个属性，表名会加上所指定的 schema 和 catalog 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 Hibernate 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"                                  (1)
        catalog="catalogName"                                (2)
        default-cascade="cascade_style"                      (3)
        default-access="field|property|ClassName"            (4)
        default-lazy="true|false"                            (5)
        auto-import="true|false"                             (6)
        package="package.name"                               (7)
/>
```

1. schema（可选）：数据库 schema 的名称。
2. catalog（可选）：数据库 catalog 的名称。
3. default-cascade（可选 — 默认为 none）：默认的级联风格。
4. default-access（可选 — 默认为 property）：Hibernate 用来访问所有属性的策略。可以通过实现 PropertyAccessor 接口自定义。
5. default-lazy（可选 — 默认为 true）：指定了未明确注明 lazy 属性的 Java 属性和集合类，Hibernate 会采取什么样的默认加载风格。
6. auto-import（可选 — 默认为 true）：指定我们是否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）。
7. 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的类名，就使用这个作为包名。

假若你有两个持久化类，它们的非全限定名是一样的（就是两个类的名字一样，所在的包不一样 — 译者注），你应该设置 auto-import="false"。如果你把一个“导入过”的名字同时对应两个类，Hibernate 会抛出一个异常。

注意 hibernate-mapping 元素允许你嵌套多个如上所示的 <class> 映射。但是最好的做法（也许一些工具需要的）是一个持久化类（或一个类的继承层次）对应一个映射文件，并以持久化的超类名称命名，例如：Cat.hbm.xml、Dog.hbm.xml，或者如果使用继承，Animal.hbm.xml。

5.1.11.3. Key

The <key> element is featured a few times within this guide. It appears anywhere the parent mapping element defines a join to a new table that references the primary key of the original table. It also defines the foreign key in the joined table:

```
<key
        column="columnname"                    1
        on-delete="noaction|cascade"           2
        property-ref="propertyName"            3
        not-null="true|false"                  4
        update="true|false"                    5
        unique="true|false"                    6
/>
```

1. column（可选）：外键字段的名称。也可以通过嵌套的 <column> 指定。
2. on-delete（可选，默认是 noaction）：表明外键关联是否打开数据库级别的级联删除。
3. property-ref（可选）：表明外键引用的字段不是原表的主键（提供给遗留数据）。
4. not-null（可选）：表明外键的字段不可为空（这意味着无论何时外键都是主键的一部分）。
5. update（可选）：表明外键决不应该被更新（这意味着无论何时外键都是主键的一部分）。
6. unique（可选）：表明外键应有唯一性约束（这意味着无论何时外键都是主键的一部分）。

对那些看重删除性能的系统，我们推荐所有的键都应该定义为 on-delete="cascade"，这样 Hibernate 将使用数据库级的 ON CASCADE DELETE 约束，而不是多个 DELETE 语句。注意，这个特性会绕过 Hibernate 通常对版本数据（versioned data）采用的乐观锁策略。

not-null 和 update 属性在映射单向一对多关联的时候有用。如果你映射一个单向一对多关联到非空的（non-nullable）外键，你必须 用 <key not-null="true"> 定义此键字段。

5.1.11.4. 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被明确映射的类和接口。

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

```
<import
```

```
        class="ClassName"                                    (1)
        rename="ShortName"                                   (2)
/>
```

(1) class：任何 Java 类的全限定名。
(2) 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。

注意

This feature is unique to hbm.xml and is not supported in annotations.

5.1.11.5. 字段和规则元素（column and formula elements）

任何接受 column 属性的映射元素都可以选择接受 <column> 子元素。同样的，formula 子元素也可以替换 <formula> 属性。

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"
        read="SQL expression"
        write="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

column 上的大多数属性都提供了在自动模式生成过程中对 DDL 进行裁剪的方法。read 和 write 属性允许你指定 Hibernate 用于访问字段值的自定义的 SQL。关于更多的内容，请参考 column read and write expressions。

column 和 formula 属性甚至可以在同一个属性或关联映射中被合并来表达，例如，一些奇异的连接条件。

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
```

```
</many-to-one>
```

5.2. Hibernate 的类型

5.2.1. 实体（Entities）和值（values）

和持久化服务相比，Java 级别的对象分为两个组别：

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。

一个实体的持久状态包含指向其他实体和值类型实例的引用。值可以是原始类型，集合（不是集合中的对象），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）是随着包含他们的实体而被持久化和删除的，他们不能被独立的加上版本信息。值没有独立的标识，所以他们不能被两个实体或者集合共享。

直到现在，我们都一直使用术语“持久类”（persistent class）来代表实体。我们仍然会这么做。然而严格说来，不是所有的用户自定义的，带有持久化状态的类都是实体。组件就是用户自定义类，却是值语义的。java.lang.String 类型的 java 属性也是值语义的。给了这个定义以后，我们可以说所有 JDK 提供的类型（类）都是值类型的语义，而用于自定义类型可能被映射为实体类型或值类型语义。采用哪种类型的语义取决于开发人员。在领域模型中，寻找实体类的一个好线索是共享引用指向这个类的单一实例，而组合或聚合通常被转化为值类型。

我们会在本文档中重复碰到这两个概念。

挑战在于将 java 类型系统(和开发者定义的实体和值类型)映射到 SQL/数据库类型系统。Hibernate 提供了连接两个系统之间的桥梁：对于实体类型，我们使用 <class>，<subclass> 等等。对于值类型，我们使用 <property>，<component> 及其他，通常跟随着 type 属性。这个属性的值是Hibernate 的映射类型的名字。Hibernate 提供了许多现成的映射（标准的 JDK 值类型）。你也可以编写自己的映射类型并实现自定义的变换策略，随后我们会看到这点。

所有的 Hibernate 内建类型，除了 collections 以外，都支持空（null）语义。

5.2.2. 基本值类型

内置的 basic mapping types 可以大致地分类为：

integer, long, short, float, double, character, byte, boolean, yes_no, true_false
: 这些类型都对应 Java 的原始类型或者其封装类，来符合（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, big_integer
从 java.math.BigDecimal 和 java.math.BigInteger 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。）

imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary
一般来说，映射类型被假定为是可变的 Java 类型，只有对不可变 Java 类型，Hibernate 会采取特定的优化措施，应用程序会把这些对象作为不可变对象处理。比如，你不应该对作为 imm_timestamp 映射的 Date 执行 Date.setTime()。要改变属性的值，并且保存这一改变，应用程序必须对这一属性重新设置一个新的（不一样的）对象。

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

在 org.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. 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 org.hibernate.UserType 或 org.hibernate.CompositeUserType 中的任一个，并且使用类型的 Java 全限定类名来定义属性。请查看 org.hibernate.test.DoubleStringType 这个例子，看看它是怎么做的。

```
<property name="twoStrings" type="org.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。

CompositeUserType，EnhancedUserType，UserCollectionType 和 UserVersionType 接口为更特殊的使用方式提供支持。

你甚至可以在一个映射文件中提供参数给一个 UserType。 为了这样做，你的 UserType 必须实现 org.hibernate.usertype.ParameterizedType 接口。为了给自定义类型提供参数，你可以在映射文件中使用 <type> 元素。

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

现在，UserType 可以从传入的 Properties 对象中得到 default 参数的值。

如果你非常频繁地使用某一 UserType，可以为他定义一个简称。这可以通过使用 <typedef> 元素来实现。Typedefs 为一自定义类型赋予一个名称，并且如果此类型是参数化的，还可以包含一系列默认的参数值。

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

也可以根据具体案例通过属性映射中的类型参数覆盖在 typedef 中提供的参数。

尽管 Hibernate 内建的丰富的类型和对组件的支持意味着你可能很少 需要使用自定义类型。不过，为那些在你的应用中经常出现的(非实体)类使用自定义类型也是一个好方法。例如，一个 MonetaryAmount 类使用 CompositeUserType 来映射是不错的选择，虽然他可以很容易地被映射成组件。这样做的动机之一是抽象。使用自定义类型，以后假若你改变表示金额的方法时，它可以保证映射文件不需要修改。

5.3. 多次映射同一个类

对特定的持久化类，映射多次是允许的。这种情形下，你必须指定 entity name 来区别不同映射实体的对象实例。（默认情况下，实体名字和类名是相同的。） Hibernate 在操作持久化对象、编写查询条件，或者把关联映射到指定实体时，允许你指定这个 entity name（实体名字）。

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

注意这里关联是如何用 entity-name 来代替 class 的。

注意

This feature is not supported in Annotations

5.4. SQL 中引号包围的标识符

你可通过在映射文档中使用反向引号（`）把表名或者字段名包围起来，以强制 Hibernate 在生成的 SQL 中把标识符用引号包围起来。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)。

```
@Entity @Table(name="`Line Item`")
class LineItem {
   @id @Column(name="`Item Id`") Integer id;
```

```
   @Column(name="`Item #`") int itemNumber
}

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

5.5. 数据库生成属性（Generated Properties）

Generated properties 指的是其值由数据库生成的属性。一般来说，如果对象有任何属性由数据库生成值，Hibernate 应用程序需要进行刷新（refresh）。但如果把属性标明为 generated，就可以转由 Hibernate 来负责这个动作。实际上。对定义了 generated properties 的实体,每当 Hibernate 执行一条 SQL INSERT 或者 UPDATE 语句，会立刻执行一条 select 来获得生成的值。

被标明为 generated 的属性还必须是 non-insertable 和 non-updateable 的。只有 versions、timestamps 和 simple properties 可以被标明为 generated。

never（默认）标明此属性值不是从数据库中生成。

insert — 标明此属性值在 insert 的时候生成，但是不会在随后的 update 时重新生成。比如说创建日期就归属于这类。注意虽然 version 和 timestamp 属性可以被标注为 generated，但是不适用这个选项。

always — 标明此属性值在 insert 和 update 时都会被生成。

To mark a property as generated, use @Generated.

5.6. 字段的读写表达式

Hibernate allows you to customize the SQL it uses to read and write the values of columns mapped to simple properties. For example, if your database provides a set of data encryption functions, you can invoke them for individual columns like this:

```
<property name="creditCardNumber">
        <column
          name="credit_card_num"
          read="decrypt(credit_card_num)"
          write="encrypt(?)"/>
</property>
```

每当属性在查询里被引用时，Hibernate 都自动应用自定义的表达式。这种功能和 derived-property formula 相似，但有两个不同的地方：

- 属性由一个或多个属性组成，它作为自动模式生成的一部分导出。

・属性是可读写的，非只读的。

如果指定了 write 表达式，它必须只包含一个“?”占位符。

注意

This feature is not supported in Annotations

5.7. 辅助数据库对象（Auxiliary Database Objects）

允许 CREATE 和 DROP 任意数据库对象，与 Hibernate 的 schema 交互工具组合起来，可以提供在 Hibernate 映射文件中完全定义用户 schema 的能力。虽然这是为创建和销毁 trigger（触发器）或stored procedure（存储过程）等特别设计的，实际上任何可以在 Java.sql.Statement.execute() 方法中执行的 SQL 命令都可以在此使用（比如ALTER，INSERT，等等）。本质上有两种模式来定义辅助数据库对象...

第一种模式是在映射文件中显式声明 CREATE 和 DROP 命令：

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

第二种模式是提供一个类，这个类知道如何组织 CREATE 和 DROP 命令。这个特别类必须实现 org.hibernate.mapping.AuxiliaryDatabaseObject 接口。

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

还有，这些数据库对象可以特别指定为仅在特定的方言中才使用。

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9iDialect"/>
        <dialect-scope name="org.hibernate.dialect.Oracle10gDialect"/>
    </database-object>
</hibernate-mapping>
```

注意

This feature is not supported in Annotations

Types

As an Object/Relational Mapping solution, Hibernate deals with both the Java and JDBC representations of application data. An online catalog application, for example, most likely has Product object with a number of attributes such as a sku, name, etc. For these individual attributes, Hibernate must be able to read the values out of the database and write them back. This 'marshalling' is the function of a Hibernate type, which is an implementation of the org.hibernate.type.Type interface. In addition, a Hibernate type describes various aspects of behavior of the Java type such as "how is equality checked?" or "how are values cloned?".

重要

A Hibernate type is neither a Java type nor a SQL datatype; it provides a information about both.

When you encounter the term type in regards to Hibernate be aware that usage might refer to the Java type, the SQL/JDBC type or the Hibernate type.

Hibernate categorizes types into two high-level groups: value types (see 第 6.1 节 “Value types”) and entity types (see 第 6.2 节 “Entity types”).

6.1. Value types

The main distinguishing characteristic of a value type is the fact that they do not define their own lifecycle. We say that they are "owned" by something else (specifically an entity, as we will see later) which defines their lifecycle. Value types are further classified into 3 sub-categories: basic types (see 第 6.1.1 节 “Basic value types”), composite types (see 第 6.1.2 节 “Composite types”) amd collection types (see 第 6.1.3 节 “Collection types”).

6.1.1. Basic value types

The norm for basic value types is that they map a single database value (column) to a single, non-aggregated Java type. Hibernate provides a number of built-in basic types, which we will present in the following sections by the Java type. Mainly these follow the natural mappings recommended in the JDBC specification. We will later cover how to override these mapping and how to provide and use alternative type mappings.

6.1.1.1. java.lang.String

org.hibernate.type.StringType
: Maps a string to the JDBC VARCHAR type. This is the standard mapping for a string if no Hibernate type is specified.

  Registered under string and java.lang.String in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.MaterializedClob
: Maps a string to a JDBC CLOB type

  Registered under materialized_clob in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.TextType
: Maps a string to a JDBC LONGVARCHAR type

  Registered under text in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.2. java.lang.Character (or char primitive)

org.hibernate.type.CharacterType
: Maps a char or java.lang.Character to a JDBC CHAR

  Registered under char and java.lang.Character in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.3. java.lang.Boolean (or boolean primitive)

org.hibernate.type.BooleanType
: Maps a boolean to a JDBC BIT type

  Registered under boolean and java.lang.Boolean in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.NumericBooleanType
: Maps a boolean to a JDBC INTEGER type as 0 = false, 1 = true

  Registered under numeric_boolean in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.YesNoType
: Maps a boolean to a JDBC CHAR type as ('N' | 'n') = false, ('Y' | 'y') = true

Registered under yes_no in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.TrueFalseType

Maps a boolean to a JDBC CHAR type as ('F' | 'f') = false, ('T' | 't') = true

Registered under true_false in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.4. java.lang.Byte (or byte primitive)

org.hibernate.type.ByteType

Maps a byte or java.lang.Byte to a JDBC TINYINT

Registered under byte and java.lang.Byte in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.5. java.lang.Short (or short primitive)

org.hibernate.type.ShortType

Maps a short or java.lang.Short to a JDBC SMALLINT

Registered under short and java.lang.Short in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.6. java.lang.Integer (or int primitive)

org.hibernate.type.IntegerTypes

Maps an int or java.lang.Integer to a JDBC INTEGER

Registered under int and java.lang.Integerin the type registry (see 第 6.5 节 “Type registry”).

6.1.1.7. java.lang.Long (or long primitive)

org.hibernate.type.LongType

Maps a long or java.lang.Long to a JDBC BIGINT

Registered under long and java.lang.Long in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.8. java.lang.Float (or float primitive)

org.hibernate.type.FloatType

Maps a float or java.lang.Float to a JDBC FLOAT

Registered under float and java.lang.Float in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.9. java.lang.Double (or double primitive)

org.hibernate.type.DoubleType

Maps a double or java.lang.Double to a JDBC DOUBLE

Registered under double and java.lang.Double in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.10. java.math.BigInteger

org.hibernate.type.BigIntegerType

Maps a java.math.BigInteger to a JDBC NUMERIC

Registered under big_integer and java.math.BigInteger in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.11. java.math.BigDecimal

org.hibernate.type.BigDecimalType

Maps a java.math.BigDecimal to a JDBC NUMERIC

Registered under big_decimal and java.math.BigDecimal in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.12. java.util.Date or java.sql.Timestamp

org.hibernate.type.TimestampType

Maps a java.sql.Timestamp to a JDBC TIMESTAMP

Registered under timestamp, java.sql.Timestamp and java.util.Date in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.13. java.sql.Time

org.hibernate.type.TimeType

Maps a java.sql.Time to a JDBC TIME

Registered under time and java.sql.Time in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.14. java.sql.Date

org.hibernate.type.DateType
: Maps a java.sql.Date to a JDBC DATE

  Registered under date and java.sql.Date in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.15. java.util.Calendar

org.hibernate.type.CalendarType
: Maps a java.util.Calendar to a JDBC TIMESTAMP

  Registered under calendar, java.util.Calendar and java.util.GregorianCalendar in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.CalendarDateType
: Maps a java.util.Calendar to a JDBC DATE

  Registered under calendar_date in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.16. java.util.Currency

org.hibernate.type.CurrencyType
: Maps a java.util.Currency to a JDBC VARCHAR (using the Currency code)

  Registered under currency and java.util.Currency in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.17. java.util.Locale

org.hibernate.type.LocaleType
: Maps a java.util.Locale to a JDBC VARCHAR (using the Locale code)

  Registered under locale and java.util.Locale in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.18. java.util.TimeZone

org.hibernate.type.TimeZoneType
: Maps a java.util.TimeZone to a JDBC VARCHAR (using the TimeZone ID)

  Registered under timezone and java.util.TimeZone in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.19. java.net.URL

org.hibernate.type.UrlType
: Maps a java.net.URL to a JDBC VARCHAR (using the external form)

  Registered under url and java.net.URL in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.20. java.lang.Class

org.hibernate.type.ClassType
: Maps a java.lang.Class to a JDBC VARCHAR (using the Class name)

  Registered under class and java.lang.Class in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.21. java.sql.Blob

org.hibernate.type.BlobType
: Maps a java.sql.Blob to a JDBC BLOB

  Registered under blob and java.sql.Blob in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.22. java.sql.Clob

org.hibernate.type.ClobType
: Maps a java.sql.Clob to a JDBC CLOB

  Registered under clob and java.sql.Clob in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.23. byte[]

org.hibernate.type.BinaryType
: Maps a primitive byte[] to a JDBC VARBINARY

  Registered under binary and byte[] in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.MaterializedBlobType
: Maps a primitive byte[] to a JDBC BLOB

Registered under materialized_blob in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.24. byte[]

org.hibernate.type.BinaryType
Maps a java.lang.Byte[] to a JDBC VARBINARY

Registered under wrapper-binary, Byte[] and java.lang.Byte[] in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.25. char[]

org.hibernate.type.CharArrayType
Maps a char[] to a JDBC VARCHAR

Registered under characters and char[] in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.26. char[]

org.hibernate.type.CharacterArrayType
Maps a java.lang.Character[] to a JDBC VARCHAR

Registered under wrapper-characters, Character[] and java.lang.Character[] in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.27. java.util.UUID

org.hibernate.type.UUIDBinaryType
Maps a java.util.UUID to a JDBC BINARY

Registered under uuid-binary and java.util.UUID in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.UUIDCharType
Maps a java.util.UUID to a JDBC CHAR (though VARCHAR is fine too for existing schemas)

Registered under uuid-char in the type registry (see 第 6.5 节 “Type registry”).

org.hibernate.type.PostgresUUIDType
Maps a java.util.UUID to the PostgreSQL UUID data type (through Types#OTHER which is how the PostgreSQL JDBC driver defines it).

Registered under pg-uuid in the type registry (see 第 6.5 节 “Type registry”).

6.1.1.28. java.io.Serializable

org.hibernate.type.SerializableType

Maps implementors of java.lang.Serializable to a JDBC VARBINARY

Unlike the other value types, there are multiple instances of this type. It gets registered once under java.io.Serializable. Additionally it gets registered under the specific java.io.Serializable implementation class names.

6.1.2. Composite types

注意

The Java Persistence API calls these embedded types, while Hibernate traditionally called them components. Just be aware that both terms are used and mean the same thing in the scope of discussing Hibernate.

Components represent aggregations of values into a single Java type. For example, you might have an Address class that aggregates street, city, state, etc information or a Name class that aggregates the parts of a person's Name. In many ways a component looks exactly like an entity. They are both (generally speaking) classes written specifically for the application. They both might have references to other application-specific classes, as well as to collections and simple JDK types. As discussed before, the only distinguishing factory is the fact that a component does not own its own lifecycle nor does it define an identifier.

6.1.3. Collection types

重要

It is critical understand that we mean the collection itself, not its contents. The contents of the collection can in turn be basic, component or entity types (though not collections), but the collection itself is owned.

Collections are covered in 第 7 章 集合映射 (Collection mappings) .

6.2. Entity types

The definition of entities is covered in detail in 第 4 章 持久化类 (Persistent Classes) . For the purpose of this discussion, it is enough to say that entities are (generally application-specific) classes which correlate to rows in a table. Specifically they correlate to the row by means of a unique identifier. Because of this

unique identifier, entities exist independently and define their own lifecycle. As an example, when we delete a Membership, both the User and Group entities remain.

注意

This notion of entity independence can be modified by the application developer using the concept of cascades. Cascades allow certain operations to continue (or "cascade") across an association from one entity to another. Cascades are covered in detail in 第 8 章 关联关系映射.

6.3. Significance of type categories

Why do we spend so much time categorizing the various types of types? What is the significance of the distinction?

The main categorization was between entity types and value types. To review we said that entities, by nature of their unique identifier, exist independently of other objects whereas values do not. An application cannot "delete" a Product sku; instead, the sku is removed when the Product itself is deleted (obviously you can update the sku of that Product to null to make it "go away", but even there the access is done through the Product).

Nor can you define an association to that Product sku. You can define an association to Product based on its sku, assuming sku is unique, but that is totally different.

TBC...

6.4. Custom types

Hibernate makes it relatively easy for developers to create their own value types. For example, you might want to persist properties of type java.lang.BigInteger to VARCHAR columns. Custom types are not limited to mapping values to a single table column. So, for example, you might want to concatenate together FIRST_NAME, INITIAL and SURNAME columns into a java.lang.String.

There are 3 approaches to developing a custom Hibernate type. As a means of illustrating the different approaches, lets consider a use case where we need to compose a java.math.BigDecimal and java.util.Currency together into a custom Money class.

6.4.1. Custom types using org.hibernate.type.Type

The first approach is to directly implement the org.hibernate.type.Type interface (or one of its derivatives). Probably, you will be more interested in the more specific org.hibernate.type.BasicType contract which would allow registration of the type (see 第 6.5 节 “Type registry”). The benefit of this registration is that whenever the metadata for a particular property does not specify the Hibernate type to use, Hibernate

will consult the registry for the exposed property type. In our example, the property type would be Money, which is the key we would use to register our type in the registry:

例 6.1. Defining and registering the custom Type

```
public class MoneyType implements BasicType {
    public String[] getRegistrationKeys() {
        return new String[] { Money.class.getName() };
    }

        public int[] sqlTypes(Mapping mapping) {
             // We will simply use delegation to the standard basic types for BigDecimal and
 Currency for many of the
            // Type methods...
            return new int[] {
                    BigDecimalType.INSTANCE.sqlType(),
                    CurrencyType.INSTANCE.sqlType(),
            };
            // we could also have honored any registry overrides via...
            //return new int[] {
                                                    //
 mappings.getTypeResolver().basic( BigDecimal.class.getName() ).sqlTypes( mappings )[0],
                                                    //
 mappings.getTypeResolver().basic( Currency.class.getName() ).sqlTypes( mappings )[0]
            //};
        }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLE
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, boolean[] settable, SessionImplementor
            throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}

Configuration cfg = new Configuration();
```

```
cfg.registerTypeOverride( new MoneyType() );
cfg...;
```

重要

It is important that we registered the type before adding mappings.

6.4.2. Custom types using org.hibernate.usertype.UserType

注意

Both org.hibernate.usertype.UserType and org.hibernate.usertype.CompositeUserType were originally added to isolate user code from internal changes to the org.hibernate.type.Type interfaces.

The second approach is the use the org.hibernate.usertype.UserType interface, which presents a somewhat simplified view of the org.hibernate.type.Type interface. Using a org.hibernate.usertype.UserType, our Money custom type would look as follows:

例 6.2. Defining the custom UserType

```
public class MoneyType implements UserType {
    public int[] sqlTypes() {
        return new int[] {
                BigDecimalType.INSTANCE.sqlType(),
                CurrencyType.INSTANCE.sqlType(),
        };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(ResultSet rs, String[] names, Object owner) throws SQLException {
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

   public void nullSafeSet(PreparedStatement st, Object value, int index) throws SQLException {
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
```

```
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

There is not much difference between the org.hibernate.type.Type example and the org.hibernate.usertype.UserType example, but that is only because of the snippets shown. If you choose the org.hibernate.type.Type approach there are quite a few more methods you would need to implement as compared to the org.hibernate.usertype.UserType.

6.4.3. Custom types using org.hibernate.usertype.CompositeUserType

The third and final approach is the use the org.hibernate.usertype.CompositeUserType interface, which differs from org.hibernate.usertype.UserType in that it gives us the ability to provide Hibernate the information to handle the composition within the Money class (specifically the 2 attributes). This would give us the capability, for example, to reference the amount attribute in an HQL query. Using a org.hibernate.usertype.CompositeUserType, our Money custom type would look as follows:

例 6.3. Defining the custom CompositeUserType

```
public class MoneyType implements CompositeUserType {
    public String[] getPropertyNames() {
          // ORDER IS IMPORTANT!  it must match the order the columns are defined in the
 property mapping
        return new String[] { "amount", "currency" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { BigDecimalType.INSTANCE, CurrencyType.INSTANCE };
    }

    public Class getReturnedClass() {
        return Money.class;
    }

    public Object getPropertyValue(Object component, int propertyIndex) {
        if ( component == null ) {
            return null;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                return money.getAmount();
            }
            case 1: {
                return money.getCurrency();
            }
            default: {
```

```
                throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
    }

        public void setPropertyValue(Object component, int propertyIndex, Object value) throws HibernateException
        if ( component == null ) {
            return;
        }

        final Money money = (Money) component;
        switch ( propertyIndex ) {
            case 0: {
                money.setAmount( (BigDecimal) value );
                break;
            }
            case 1: {
                money.setCurrency( (Currency) value );
                break;
            }
            default: {
               throw new HibernateException( "Invalid property index [" + propertyIndex + "]" );
            }
        }
        }

    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws SQLEx
        assert names.length == 2;
       BigDecimal amount = BigDecimalType.INSTANCE.get( names[0] ); // already handles null check
        Currency currency = CurrencyType.INSTANCE.get( names[1] ); // already handles null check
        return amount == null && currency == null
                ? null
                : new Money( amount, currency );
    }

    public void nullSafeSet(PreparedStatement st, Object value, int index, SessionImplementor session) throws SQLE
        if ( value == null ) {
            BigDecimalType.INSTANCE.set( st, null, index );
            CurrencyType.INSTANCE.set( st, null, index+1 );
        }
        else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set( st, money.getAmount(), index );
            CurrencyType.INSTANCE.set( st, money.getCurrency(), index+1 );
        }
    }

    ...
}
```

6.5. Type registry

Internally Hibernate uses a registry of basic types (see 第 6.1.1 节 “Basic value types”) when it needs to resolve the specific org.hibernate.type.Type to use in certain situations. It also provides a way for applications to add extra basic type registrations as well as override the standard basic type registrations.

To register a new type or to override an existing type registration, applications would make use of the registerTypeOverride method of the org.hibernate.cfg.Configuration class when bootstrapping Hibernate. For example, lets say you want Hibernate to use your custom SuperDuperStringType; during bootstrap you would call:

例 6.4. Overriding the standard StringType

```
Configuration cfg = ...;
cfg.registerTypeOverride( new SuperDuperStringType() );
```

The argument to registerTypeOverride is a org.hibernate.type.BasicType which is a specialization of the org.hibernate.type.Type we saw before. It adds a single method:

例 6.5. Snippet from BasicType.java

```
/**
         * Get the names under which this type should be registered in the type registry.
         *
         * @return The keys under which to register this type.
         */
        public String[] getRegistrationKeys();
```

One approach is to use inheritance (SuperDuperStringType extends org.hibernate.type.StringType); another is to use delegation.

集合映射（Collection mappings）

7.1. 持久化集合类（Persistent collections）

Naturally Hibernate also allows to persist collections. These persistent collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities. The distinction between value and reference semantics is in this context very important. An object in a collection might be handled with "value" semantics (its life cycle fully depends on the collection owner), or it might be a reference to another entity with its own life cycle. In the latter case, only the "link" between the two objects is considered to be a state held by the collection.

As a requirement persistent collection-valued fields must be declared as an interface type (see 例 7.2 “Collection mapping using @OneToMany and @JoinColumn”). The actual interface might be java.util.Set, java.util.Collection, java.util.List, java.util.Map, java.util.SortedSet, java.util.SortedMap or anything you like ("anything you like" means you will have to write an implementation of org.hibernate.usertype.UserCollectionType).

Notice how in 例 7.2 “Collection mapping using @OneToMany and @JoinColumn” the instance variable parts was initialized with an instance of HashSet. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent, by calling persist(), Hibernate will actually replace the HashSet with an instance of Hibernate's own implementation of Set. Be aware of the following error:

例 7.1. Hibernate uses its own collection implementations

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);

kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

根据不同的接口类型，被 Hibernate 注射的持久化集合类的表现类似 HashMap、HashSet、TreeMap、TreeSet 或 ArrayList。

集合类实例具有值类型的通常行为。当被持久化对象引用后，他们会自动被持久化，当不再被引用后，自动被删除。假若实例被从一个持久化对象传递到另一个，它的元素可能从一个表转移到另一个表。两个实体不能共享同一个集合类实例的引用。因为底层关系数据库模型的原因，集合值属性无法支持空值语义；Hibernate 对空的集合引用和空集合不加区别。

注意

Use persistent collections the same way you use ordinary Java collections. However, ensure you understand the semantics of bidirectional associations (see 第 7.3.2 节 “双向关联 (Bidirectional associations)”).

7.2. How to map collections

Using annotations you can map Collections, Lists, Maps and Sets of associated entities using @OneToMany and @ManyToMany. For collections of a basic or embeddable type use @ElementCollection. In the simplest case a collection mapping looks like this:

例 7.2. Collection mapping using @OneToMany and @JoinColumn

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();

    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinColumn(name="PART_ID")
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


@Entity
public class Part {
   ...
}
```

Product describes a unidirectional relationship with Part using the join column PART_ID. In this unidirectional one to many scenario you can also use a join table as seen in 例 7.3 “Collection mapping using @OneToMany and @JoinTable”.

例 7.3. Collection mapping using @OneToMany and @JoinTable

```
@Entity
public class Product {

    private String serialNumber;
    private Set<Part> parts = new HashSet<Part>();
```

```
    @Id
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }

    @OneToMany
    @JoinTable(
            name="PRODUCT_PARTS",
            joinColumns = @JoinColumn( name="PRODUCT_ID"),
            inverseJoinColumns = @JoinColumn( name="PART_ID")
    )
    public Set<Part> getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
}


@Entity
public class Part {
   ...
}
```

Without describing any physical mapping (no @JoinColumn or @JoinTable), a unidirectional one to many with join table is used. The table name is the concatenation of the owner table name, _, and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table, _, and the owner primary key column(s) name. The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s) name. A unique constraint is added to the foreign key referencing the other side table to reflect the one to many.

Lets have a look now how collections are mapped using Hibernate mapping files. In this case the first step is to chose the right mapping element. It depends on the type of interface. For example, a <set> element is used for mapping properties of type Set.

例 7.4. Mapping a Set using <set>

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

In 例 7.4 “Mapping a Set using <set>” a one-to-many association links the Product and Part entities. This association requires the existence of a foreign key column and possibly an index column to the Part table. This mapping loses certain semantics of normal Java collections:

・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。

Looking closer at the used <one-to-many> tag we see that it has the following options.

例 7.5. options of <one-to-many> element

```
<one-to-many
        class="ClassName"                                    (1)
        not-found="ignore|exception"                         (2)
        entity-name="EntityName"                             (3)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) class（必需）：被关联类的名称。
(2) not-found（可选 - 默认为exception）：指明若缓存的标示值关联的行缺失，该如何处理：ignore 会把缺失的行作为一个空关联处理。
(3) entity-name（可选）：被关联的类的实体名，作为 class 的替代。

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

警告

If the foreign key column of a <one-to-many> association is declared NOT NULL, you must declare the <key> mapping not-null="true" or use a bidirectional association with the collection mapping marked inverse="true". See 第 7.3.2 节 “双向关联（Bidirectional associations）”.

Apart from the <set> tag as shown in 例 7.4 “Mapping a Set using <set>”, there is also <list>, <map>, <bag>, <array> and <primitive-array> mapping elements. The <map> element is representative:

例 7.6. Elements of the <map> mapping

```
<map
    name="propertyName"                                      (1)
    table="table_name"                                       (2)
    schema="schema_name"                                     (3)
    lazy="true|extra|false"                                  (4)
    inverse="true|false"                                     (5)
    cascade="all|none|save-update|delete|all-delete-orphan|delete-orphan"  (6)
    sort="unsorted|natural|comparatorClass"                  (7)
    order-by="column_name asc|desc"                          (8)
    where="arbitrary sql where condition"                    (9)
```

```
    fetch="join|select|subselect"                                    ⑩
    batch-size="N"                                                   ⑪
    access="field|property|ClassName"                                ⑫
    optimistic-lock="true|false"                                     ⑬
    mutable="true|false"                                             ⑭
    node="element-name|."
    embed-xml="true|false"
>

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

1. name：集合属性的名称
2. 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）。
3. schema（可选）：表的 schema 的名称，他将覆盖在根元素中定义的 schema
4. lazy（可选--默认为 true）可以用来关闭延迟加载（false）：指定一直使用预先抓取，或者打开 "extra-lazy" 抓取，此时大多数操作不会初始化集合类（适用于非常大的集合）。
5. 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。
6. cascade（可选 — 默认为 none）让操作级联到子实体。
7. 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。
8. order-by (optional): specifies a table column or columns that define the iteration order of the Map, Set or bag, together with an optional asc or desc.
9. 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）。
10. fetch（可选，默认为 select）：用于在外连接抓取、通过后续 select 抓取和通过后续 subselect 抓取之间选择。
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 取得集合属性值时使用的策略。
13. 乐观锁（可选 - 默认为 true）：对集合的状态的改变会是否导致其所属的实体的版本增长（对一对多关联来说，关闭这个属性常常是有理的）。
14. mutable（可变）（可选 — 默认为 true）：若值为 false，表明集合中的元素不会改变（在某些情况下可以进行一些小的性能优化）。

After exploring the basic mapping of collections in the preceding paragraphs we will now focus details like physical mapping considerations, indexed collections and collections of value types.

7.2.1. 集合外键（Collection foreign keys）

On the database level collection instances are distinguished by the foreign key of the entity that owns the collection. This foreign key is referred to as the collection key

column, or columns, of the collection table. The collection key column is mapped by the @JoinColumn annotation respectively the <key> XML element.

There can be a nullability constraint on the foreign key column. For most collections, this is implied. For unidirectional one-to-many associations, the foreign key column is nullable by default, so you may need to specify

```
@JoinColumn(nullable=false)
```

or

```
<key column="productSerialNumber" not-null="true"/>
```

The foreign key constraint can use ON DELETE CASCADE. In XML this can be expressed via:

```
<key column="productSerialNumber" on-delete="cascade"/>
```

In annotations the Hibernate specific annotation @OnDelete has to be used.

```
@OnDelete(action=OnDeleteAction.CASCADE)
```

See 第 5.1.11.3 节 “Key” for more information about the <key> element.

7.2.2. 索引集合类 (Indexed collections)

In the following paragraphs we have a closer at the indexed collections List and Map how the their index can be mapped in Hibernate.

7.2.2.1. Lists

Lists can be mapped in two different ways:

- as ordered lists, where the order is not materialized in the database
- as indexed lists, where the order is materialized in the database

To order lists in memory, add @javax.persistence.OrderBy to your property. This annotation takes as parameter a list of comma separated properties (of the target entity) and orders the collection accordingly (eg firstname asc, age desc), if the string is empty, the collection will be ordered by the primary key of the target entity.

例 7.7. Ordered lists using `@OrderBy`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderBy("number")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

To store the index value in a dedicated column, use the `@javax.persistence.OrderColumn` annotation on your property. This annotations describes the column name and attributes of the column keeping the index value. This column is hosted on the table containing the association foreign key. If the column name is not specified, the default is the name of the referencing property, followed by underscore, followed by `ORDER` (in the following example, it would be `orders_ORDER`).

例 7.8. Explicit index column using `@OrderColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
```

```
   private Integer id;

   @OneToMany(mappedBy="customer")
   @OrderColumn(name="orders_index")
   public List<Order> getOrders() { return orders; }
   public void setOrders(List<Order> orders) { this.orders = orders; }
   private List<Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|--------------| |----------|
| Order        | | Customer |
|--------------| |----------|
| id           | | id       |
| number       | |----------|
| customer_id  |
| orders_order |
|--------------|
```

注意

We recommend you to convert the legacy @org.hibernate.annotations.IndexColumn usages to @OrderColumn unless you are making use of the base property. The base property lets you define the index value of the first element (aka as base index). The usual value is 0 or 1. The default is 0 like in Java.

Looking again at the Hibernate mapping file equivalent, the index of an array or list is always of type integer and is mapped using the <list-index> element. The mapped column contains sequential integers that are numbered from zero by default.

例 7.9. index-list element for indexed collections in xml mapping

```
<list-index
        column="column_name"
```

```
        base="0|1|..."/>
```

1. column_name（必需）：持有集合索引值的字段的名称。
1. base（可选 — 默认为 0）对应列表或队列的第一个元素的索引字段的值。

假若你的表没有一个索引字段，当你仍然希望使用 List 作为属性类型，你应该把此属性映射为 Hibernate <bag>。从数据库中获取的时候，bag 不维护其顺序，但也可选择性的进行排序。

7.2.2.2. Maps

The question with Maps is where the key value is stored. There are everal options. Maps can borrow their keys from one of the associated entity properties or have dedicated columns to store an explicit key.

To use one of the target entity property as a key of the map, use @MapKey(name="myProperty"), where myProperty is a property name in the target entity. When using @MapKey without the name attribuate, the target entity primary key is used. The map key uses the same column as the property pointed out. There is no additional column defined to hold the map key, because the map key represent a target property. Be aware that once loaded, the key is no longer kept in sync with the property. In other words, if you change the property value, the key will not change automatically in your Java model.

例 7.10. Use of target entity property as map key via @MapKey

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany(mappedBy="customer")
   @MapKey(name="number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> order) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}
```

```
-- Table schema
|-------------| |----------|
| Order       | | Customer |
|-------------| |----------|
| id          | | id       |
| number      | |----------|
| customer_id |
|-------------|
```

Alternatively the map key is mapped to a dedicated column or columns. In order to customize the mapping use one of the following annotations:

- `@MapKeyColumn` if the map key is a basic type. If you don't specify the column name, the name of the property followed by underscore followed by `KEY` is used (for example `orders_KEY`).
- `@MapKeyEnumerated` / `@MapKeyTemporal` if the map key type is respectively an enum or a `Date`.
- `@MapKeyJoinColumn`/`@MapKeyJoinColumns` if the map key type is another entity.
- `@AttributeOverride`/`@AttributeOverrides` when the map key is a embeddable object. Use `key.` as a prefix for your embeddable object property names.

You can also use `@MapKeyClass` to define the type of the key if you don't use generics.

例 7.11. Map key as basic type using `@MapKeyColumn`

```
@Entity
public class Customer {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   @OneToMany @JoinTable(name="Cust_Order")
   @MapKeyColumn(name="orders_number")
   public Map<String,Order> getOrders() { return orders; }
   public void setOrders(Map<String,Order> orders) { this.orders = orders; }
   private Map<String,Order> orders;
}

@Entity
public class Order {
   @Id @GeneratedValue public Integer getId() { return id; }
   public void setId(Integer id) { this.id = id; }
   private Integer id;

   public String getNumber() { return number; }
   public void setNumber(String number) { this.number = number; }
   private String number;

   @ManyToOne
   public Customer getCustomer() { return customer; }
```

```
   public void setCustomer(Customer customer) { this.customer = customer; }
   private Customer number;
}

-- Table schema
|-------------| |----------| |---------------|
| Order       | | Customer | | Cust_Order    |
|-------------| |----------| |---------------|
| id          | | id       | | customer_id   |
| number      | |----------| | order_id      |
| customer_id |              | orders_number |
|-------------|              |---------------|
```

注意

We recommend you to migrate from @org.hibernate.annotations.MapKey / @org.hibernate.annotation.MapKeyManyToMany to the new standard approach described above

Using Hibernate mapping files there exists equivalent concepts to the descibed annotations. You have to use <map-key>, <map-key-many-to-many> and <composite-map-key>. <map-key> is used for any basic type, <map-key-many-to-many> for an entity reference and <composite-map-key> for a composite type.

例 7.12. map-key xml mapping element

```
<map-key
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="type_name"                        (3)
        node="@attribute-name"
        length="N"/>
```

(1) column（可选）：持有集合索引值的字段的名称。
(2) formula（可选）：用于对表键求值的 SQL 公式。
(3) type（必需）：映射键的类型。

例 7.13. map-key-many-to-many

```
<map-key-many-to-many
        column="column_name"                    (1)
        formula="any SQL expression"            (2)(3)
        class="ClassName"
/>
```

1. column（可选）：用于集合索引值的外键字段的名称。
2. formula（可选）：用于对映射键的外键求值的 SQL 公式。
3. class（必需）：用作映射键的实体类的名称。

7.2.3. Collections of basic types and embeddable objects

In some situations you don't need to associate two entities but simply create a collection of basic types or embeddable objects. Use the @ElementCollection for this case.

例 7.14. Collection of basic types mapped via @ElementCollection

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Nicknames", joinColumns=@JoinColumn(name="user_id"))
   @Column(name="nickname")
   public Set<String> getNicknames() { ... }
}
```

The collection table holding the collection data is set using the @CollectionTable annotation. If omitted the collection table name defaults to the concatenation of the name of the containing entity and the name of the collection attribute, separated by an underscore. In our example, it would be User_nicknames.

The column holding the basic type is set using the @Column annotation. If omitted, the column name defaults to the property name: in our example, it would be nicknames.

But you are not limited to basic types, the collection type can be any embeddable object. To override the columns of the embeddable object in the collection table, use the @AttributeOverride annotation.

例 7.15. @ElementCollection for embeddable objects

```
@Entity
public class User {
   [...]
   public String getLastname() { ...}

   @ElementCollection
   @CollectionTable(name="Addresses", joinColumns=@JoinColumn(name="user_id"))
   @AttributeOverrides({
      @AttributeOverride(name="street1", column=@Column(name="fld_street"))
   })
   public Set<Address> getAddresses() { ... }
}

@Embeddable
```

```
public class Address {
   public String getStreet1() {...}
   [...]
}
```

Such an embeddable object cannot contains a collection itself.

注意

in @AttributeOverride, you must use the value. prefix to override properties of the embeddable object used in the map value and the key. prefix to override properties of the embeddable object used in the map key.

```
@Entity
public class User {
   @ElementCollection
   @AttributeOverrides({
      @AttributeOverride(name="key.street1", column=@Column(name="fld_street")),
      @AttributeOverride(name="value.stars", column=@Column(name="fld_note"))
   })
   public Map<Address,Rating> getFavHomes() { ... }
```

注意

We recommend you to migrate from @org.hibernate.annotations.CollectionOfElements to the new @ElementCollection annotation.

Using the mapping file approach a collection of values is mapped using the <element> tag. For example:

例 7.16. <element> tag for collection values using mapping files

```
<element
        column="column_name"                                    1
        formula="any SQL expression"                            2
        type="typename"                                         3
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

1. column（可选）：持有集合元素值的字段的名称。
2. formula（可选）：用于对元素求值的 SQL 公式。
3. type（必需）：集合元素的类型。

7.3. 高级集合映射（Advanced collection mappings）

7.3.1. 有序集合（Sorted collections）

Hibernate supports collections implementing java.util.SortedMap and java.util.SortedSet. With annotations you declare a sort comparator using @Sort. You chose between the comparator types unsorted, natural or custom. If you want to use your own comparator implementation, you'll also have to specify the implementation class using the comparator attribute. Note that you need to use either a SortedSet or a SortedMap interface.

例 7.17. Sorted collection with @Sort

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

Using Hibernate mapping files you specify a comparator in the mapping file with <sort>:

例 7.18. Sorted collection using xml mapping

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

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key 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。

提示

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

If you want the database itself to order the collection elements, use the order-by attribute of set, bag or map mappings. This solution is implemented using LinkedHashSet or LinkedHashMap and performs the ordering in the SQL query and not in the memory.

例 7.19. Sorting in database using order-by

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

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

注意

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集合 filter() 根据任意的条件来排序：

例 7.20. Sorting via a query filter

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

7.3.2. 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

Often there exists a many to one association which is the owner side of a bidirectional relationship. The corresponding one to many association is in this case annotated by @OneToMany(mappedBy=...)

例 7.21. Bidirectional one to many with many to one side as association owner

```
@Entity
```

```
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

Troop has a bidirectional one to many relationship with Soldier through the troop property. You don't have to (must not) define any physical mapping in the mappedBy side.

To map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the mappedBy element and set the many to one @JoinColumn as insertable and updatable to false. This solution is not optimized and will produce additional UPDATE statements.

例 7.22. Bidirectional associtaion with one to many side as owner

```
@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

How does the mappping of a bidirectional mapping look like in Hibernate mapping xml? There you define 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".

例 7.23. Bidirectional one to many via Hibernate mapping files

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
```

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

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

在“一”这一端定义 inverse="true" 不会影响级联操作，二者是正交的概念。

A many-to-many association is defined logically using the @ManyToMany annotation. You also have to describe the association table and the join conditions using the @JoinTable annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

例 7.24. Many to many association via @ManyToMany

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
```

```
}
```

In this example @JoinTable defines a name, an array of join columns, and an array of inverse join columns. The latter ones are the columns of the association table which refer to the Employee primary key (the "other side"). As seen previously, the other side don't have to (must not) describe the physical mapping: a simple mappedBy argument containing the owner side property name bind the two.

As any other annotations, most values are guessed in a many to many relationship. Without describing any physical mapping in a unidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table name, _ and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

例 7.25. Default values for @ManyToMany (uni-directional)

```
@Entity
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}
```

A Store_City is used as the join table. The Store_id column is a foreign key to the Store table. The implantedIn_id column is a foreign key to the City table.

Without describing any physical mapping in a bidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, _, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

例 7.26. Default values for @ManyToMany (bi-directional)

```
@Entity
public class Store {
```

```
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}
```

A Store_Customer is used as the join table. The stores_id column is a foreign key to the Store table. The customers_id column is a foreign key to the Customer table.

Using Hibernate mapping files you can map a bidirectional many-to-many association by mapping two many-to-many associations to the same database table and declaring one end as inverse.

注意

You cannot select an indexed collection.

例 7.27 “Many to many association using Hibernate mapping files” shows a bidirectional many-to-many association that illustrates how each category can have many items and each item can be in many categories:

例 7.27. Many to many association using Hibernate mapping files

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="ITEM_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_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。 这表示 Hibernate 为每个双向关联在内存中存在两次表现，一个从 A 连接到 B，另一个从 B 连接到 A。如果你回想一下 Java 对象模型，我们是如何在 Java 中创建多对多关系的，这可以让你更容易理解：

例 7.28. Effect of inverse vs. non-inverse side of many to many associations

```
category.getItems().add(item);          // The category now "knows" about the relationship
item.getCategories().add(category);     // The item now "knows" about the relationship

session.persist(item);                  // The relationship won't be saved!
session.persist(category);              // The relationship will be saved
```

非反向端用于把内存中的表示保存到数据库中。

7.3.3. 双向关联，涉及有序集合类

There are some additional considerations for bidirectional mappings with indexed collections (where one end is represented as a <list> or <map>) when using Hibernate mapping files. If there is a property of the child class that maps to the index column you can use inverse="true" on the collection mapping:

例 7.29. Bidirectional association with indexed collection

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

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

但是，假若子类中没有这样的属性存在，我们不能认为这个关联是真正的双向关联（信息不对称，在关联的一端有一些另外一端没有的信息）。在这种情况下，我们不能使用 inverse="true"。我们需要这样用：

例 7.30. Bidirectional association with indexed collection, but no index column

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

注意在这个映射中，关联中集合类"值"一端负责来更新外键。

7.3.4. 三重关联（Ternary associations）

有三种可能的途径来映射一个三重关联。第一种是使用一个 Map，把一个关联作为其索引：

例 7.31. Ternary association mapping

```
@Entity
public class Company {
   @Id
   int id;
   ...
   @OneToMany // unidirectional
   @MapKeyJoinColumn(name="employee_id")
   Map<Employee, Contract> contracts;
}

// or

<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

A second approach is to remodel the association as an entity class. This is the most common approach. A final alternative is to use composite elements, which will be discussed later.

7.3.5. Using an <idbag>

The majority of the many-to-many associations and collections of values shown previously all map to tables with composite keys, even though it has been suggested that entities should have synthetic identifiers (surrogate keys). A pure association table does not seem to benefit much from a surrogate key, although a collection of composite values might. For this reason Hibernate provides a feature that allows you to map many-to-many associations and collections of values to a table with a surrogate key.

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

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</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 标识符生成器策略来生成 <idbag> 集合的标识符。

7.4. 集合例子 (Collection example)

集合例子 (Collection example)。

下面的代码是用来添加一个新的 Child:

例 7.32. Example classes Parent and Child

```
public class Parent {
    private long id;
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
```

```
   private long id;
   private String name


   // getter/setter
   ...
}
```

这个类有一个 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：

例 7.33. One to many unidirectional Parent-Child relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
}
```

例 7.34. One to many unidirectional Parent-Child relationship using mapping files

```
<hibernate-mapping>

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

```
    </class>

    <class name="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:

例 7.35. Table definitions for unidirectional Parent-Child relationship

```
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:

例 7.36. One to many bidirectional Parent-Child relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(mappedBy="parent")
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   @ManyToOne
   private Parent parent;


   // getter/setter
   ...
}
```

例 7.37. One to many bidirectional Parent-Child relationship using mapping files

```
<hibernate-mapping>

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

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

</hibernate-mapping>
```

请注意 NOT NULL 的约束:

例 7.38. Table definitions for bidirectional Parent-Child relationship

```
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
```

Alternatively, if this association must be unidirectional you can enforce the NOT NULL constraint.

例 7.39. Enforcing NOT NULL constraint in unidirectional relation using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @OneToMany(optional=false)
    private Set<Child> children;
```

```
    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;
   private String name;


   // getter/setter
   ...
}
```

例 7.40. Enforcing NOT NULL constraint in unidirectional relation using mapping files

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

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

</hibernate-mapping>
```

On the other hand, if a child has multiple parents, a many-to-many association is appropriate.

例 7.41. Many to many Parent-Child relationship using annotations

```
public class Parent {
    @Id
    @GeneratedValue
    private long id;

    @ManyToMany
```

```
    private Set<Child> children;

    // getter/setter
    ...
}


public class Child {
   @Id
   @GeneratedValue
   private long id;

   private String name;


   // getter/setter
   ...
}
```

例 7.42. Many to many Parent-Child relationship using mapping files

```
<hibernate-mapping>

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

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

</hibernate-mapping>
```

表定义：

例 7.43. Table definitions for many to many releationship

```
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
```

For more examples and a complete explanation of a parent/child relationship mapping, see 第 24 章 示例：父子关系 (Parent/Child) for more information. Even more complex association mappings are covered in the next chapter.

关联关系映射

8.1. 介绍

关联关系映射通常情况是最难配置正确的。在这个部分中，我们从单向关系映射开始，然后考虑双向关系映射，逐步讲解典型的案例。在所有的例子中，我们都使将用 Person 和 Address。

我们根据映射关系是否涉及连接表以及多样性（multiplicity）来划分关联类型。

在传统的数据建模中，允许为 Null 值的外键被认为是一种不好的实践，因此我们所有的例子中都使用不允许为 Null 的外键。这并不是 Hibernate的 要求，即使你删除掉不允许为 Null 的约束，Hibernate 映射一样可以工作的很好。

8.2. 单向关联（Unidirectional associations）

8.2.1. 多对一（many-to-one）

单向 many-to-one 关联是最常见的单向关联关系。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

8.2.2. 一对一（One-to-one）

基于外键关联的单向一对一关联和单向多对一关联几乎是一样的。唯一的不同就是单向一对一关联中的外键字段具有唯一性约束。

```
<class name="Person">
```

```
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

基于主键关联的单向一对一关联通常使用一个特定的 `id` 生成器，然而在这个例子中我们掉换了关联的方向：

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property"
>person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

8.2.3. 一对多（one-to-many）

基于外键关联的单向一对多关联是一种很少见的情况，我们不推荐使用它。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

我们认为对于这种关联关系最好使用连接表。

8.3. 使用连接表的单向关联（Unidirectional associations with join tables）

8.3.1. 一对多（one-to-many）

基于连接表的单向一对多关联 应该优先被采用。请注意，通过指定unique="true"，我们可以把多样性从多对多改变为一对多。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
```

```
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

8.3.2. 多对一（many-to-one）

基于连接表的单向多对一关联在关联关系可选的情况下应用也很普遍。例如：

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

8.3.3. 一对一（One-to-one）

基于连接表的单向一对一关联也是可行的，但非常少见。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
```

```
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
 unique )
create table Address ( addressId bigint not null primary key )
```

8.3.4. 多对多（many-to-many）

最后，这里是一个单向多对多关联的例子。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.4. 双向关联（Bidirectional associations）

8.4.1. 一对多（one to many）/多对一（many to one）

双向多对一关联 是最常见的关联关系。下面的例子解释了这种标准的父/子关联关系。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

如果你使用 List（或者其他有序集合类），你需要设置外键对应的 key 列为 not null。Hibernate 将从集合端管理关联，维护每个元素的索引，并通过设置 update="false" 和 insert="false" 来对另一端反向操作。

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
```

```
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class
>
```

假若集合映射的 <key> 元素对应的底层外键字段是 NOT NULL 的，那么为这一 key 元素定义 not-null="true" 是很重要的。不要仅仅为可能的嵌套 <column>元素定义 not-null="true"，<key> 元素也是需要的。

8.4.2. 一对一（One-to-one）

基于外键关联的双向一对一关联也很常见。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class
>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

基于主键关联的一对一关联需要使用特定的 id 生成器：

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>
```

```
<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property"
>person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

8.5. 使用连接表的双向关联 (Bidirectional associations with join tables)

8.5.1. 一对多 (one to many)/多对一 (many to one)

下面是一个基于连接表的双向一对多关联的例子。注意 `inverse="true"` 可以出现在关联的任意一端，即 collection 端或者 join 端。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class
```

```
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

8.5.2. 一对一（one to one）

基于连接表的双向一对一关联也是可行的，但极为罕见。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null
 unique )
create table Address ( addressId bigint not null primary key )
```

8.5.3. 多对多（many-to-many）

下面是一个双向多对多关联的例子。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class
>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key
 (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8.6. 更复杂的关联映射

更复杂的关联连接极为罕见。通过在映射文档中嵌入 SQL 片断，Hibernate 也可以处理更为复杂的情况。比如，假若包含历史帐户数据的表定义了 accountNumber、effectiveEndDate 和 effectiveStartDate 字段，按照下面映射：

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula
>case when effectiveEndDate is null then 1 else 0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

那么我们可以对目前（current）实例（其 effectiveEndDate 为 null）使用这样的关联映射：

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula
>'1'</formula>
</many-to-one
>
```

在更复杂的例子中，假想 Employee 和 Organization 之间的关联是通过一个 Employment 中间表维护的，而中间表中填充了很多历史雇员数据。那“雇员的最新雇主”这个关联（最新雇主就是具有最新的 startDate 的那个）可以这样映射：

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join
>
```

使用这一功能时可以充满创意和灵活性，但通常更加实用的是用 HQL 或条件查询来处理这些情况。

组件（Component）映射

组件（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。

9.1. 依赖对象（Dependent objects）

组件（Component）是一个被包含的对象，在持久化的过程中，它被当作值类型，而并非一个实体的引用。在这篇文档中，组件这一术语指的是面向对象的合成概念（而并不是系统构架层次上的组件的概念）。举个例子，你对人（Person）这个概念可以像下面这样来建模：

```
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;
    }
}
```

在持久化的过程中，姓名（Name）可以作为人（Person）的一个组件。需要注意的是：你应该为姓名的持久化属性定义 getter 和 setter 方法，但是你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 Hibernate 映射文件：

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </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，组件不支持共享引用。换句话说，两个人可能重名，但是两个 Person 对象应该包含两个独立的 Name 对象，只不过这两个 Name 对象具有“同样”的值。组件的值可以为空，其定义如下。 每当 Hibernate 重新加载一个包含组件的对象，如果该组件的所有字段为空，Hibernate 将假定整个组件为空。在大多数情况下，这样假定应该是没有问题的。

组件的属性可以是任意一种 Hibernate 类型（包括集合，多对多关联，以及其它组件等等）。嵌套组件不应该被当作一种特殊的应用（Nested components should not be considered an exotic usage）。Hibernate 倾向于支持细颗粒度的（fine-grained）对象模型。

<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。

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

```
    </component>
</class
>
```

9.2. 在集合中出现的依赖对象（Collections of dependent objects）

Hibernate 支持组件的集合（例如：一个元素是姓名 Name 这种类型的数组）。你可以使用 <composite-element> 标签替代 <element> 标签来定义你的组件集合。

```
<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
>
```

重要

注意，如果你定义的 Set 包含组合元素（composite-element），正确地实现 equals() 和 hashCode() 是非常重要的。

组合元素可以包含组件，但是不能包含集合。如果你的组合元素自身包含组件，你必须使用 <nested-composite-element> 标签。这是一个相当特殊的案例 — 在一个组件的集合里，那些组件本身又可以包含其他的组件。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会有细微的变化。

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值来确定一条记录（在组合元素表中，没有单独的关键字段），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用法是它可以包含一个<many-to-one>元素。类似这样的映射允许你将一个 many-to-many 关联表的额外字段映射为组合元素类。接下来的的例子是从 Order 到 Item 的一个多对多的关联关系，关联属性是 purchaseDate，price 和 quantity 。

```
<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
>
```

当然，当你定义 Item 时，你无法引用这些 purchase，因此你无法实现双向关联查询。记住组件是值类型，并且不允许共享引用。某一个特定的 Purchase 可以放在 Order 的集合中，但它不能同时被 Item 所引用。

其实组合元素的这个用法可以扩展到三重或多重关联：

```
<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，表达组合元素的语法和关联到其他实体的语法是一样的。

9.3. 组件作为 Map 的索引（Components as Map indices ）

<composite-map-key> 元素允许你映射一个组件类作为一个 Map 的 key，前提是你必须正确的在这个类中重写了 hashCode() 和 equals() 方法。

9.4. 组件作为联合标识符（Components as composite identifiers）

你可以使用一个组件作为一个实体类的标识符。你的组件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

注意

注意：在 Hibernate3 中，第二个要求并非是 Hibernate 强制必须的。但最好这样做。

你不能使用一个 IdentifierGenerator 产生组合关键字。一个应用程序必须分配它自己的标识符。

使用 <composite-id> 标签（并且内嵌 <key-property> 元素）代替通常的 <id> 标签。比如，OrderLine 类具有一个主键，这个主键依赖于 Order 的（联合）主键。

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class
>
```

现在，任何指向 OrderLine 的外键都是复合的。在你的映射文件中，必须为其他类也这样声明。例如，一个指向 OrderLine 的关联可能被这样映射：

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one
>
```

提示

注意在各个地方 column 标签都是 column 属性的替代写法。使用 column 元素只是给出一个更详细的选项，在使用 hbm2ddl 时会更有用。

指向 OrderLine 的多对多关联也使用联合外键：

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set
>
```

在 Order 中，OrderLine 的集合则是这样：

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set
>
```

与通常一样，<one-to-many> 元素不声明任何列。

假若 OrderLine 本身拥有一个集合，它也具有组合外键。

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key
>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class
>
```

9.5. 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" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</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，也可以在程序运行时操作映射文件。更好的是，你可以通过 Configuration 对象来访问（或者修改）Hibernate 的运行时元模型。

继承映射（Inheritance Mapping）

10.1. 三种策略

Hibernate 支持三种基本的继承映射策略：

- 每个类分层结构一张表（table per class hierarchy）
- table per subclass
- 每个具体类一张表（table per concrete class）

此外，Hibernate 还支持第四种稍有不同的多态映射策略：

- 隐式多态（implicit polymorphism）

对于同一个继承层次内的不同分支，可以采用不同的映射策略，然后用隐式多态来完成跨越整个层次的多态。但是在同一个 <class> 根元素下，Hibernate 不支持混合了元素 <subclass>、<joined-subclass> 和 <union-subclass> 的映射。在同一个 <class> 元素下，可以混合使用“每个类分层结构一张表”（table per hierarchy）和“每个子类一张表”（table per subclass）这两种映射策略，这是通过结合元素 <subclass> 和 <join> 来实现的（见后）。

在多个映射文件中，可以直接在 hibernate-mapping 根下定义 subclass，union-subclass 和 joined-subclass。也就是说，你可以仅加入一个新的映射文件来扩展类层次。你必须在 subclass 的映射中指明 extends 属性，给出一个之前定义的超类的名字。注意，在以前，这一功能对映射文件的顺序有严格的要求，从 Hibernate 3 开始，使用 extends 关键字的时侯，对映射文件的顺序不再有要求；但在每个映射文件里，超类必须在子类之前定义。

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

10.1.1. 每个类分层结构一张表（Table per class hierarchy）

假设我们有接口Payment和它的几个实现类：CreditCardPayment，CashPayment 和ChequePayment。则“每个类分层结构一张表”（Table per class hierarchy）的映射代码如下所示：

```
<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">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class
>
```

采用这种策略只需要一张表即可。它有一个很大的限制：要求那些由子类定义的字段，　　如CCTYPE，不能有非空（NOT NULL）约束。

10.1.2. 每个子类一张表（Table per subclass）

对于上例中的几个类而言，采用“每个子类一张表”的映射策略，代码如下所示：

```
<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"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class
>
```

需要四张表。三个子类表通过主键关联到超类表（因而关系模型实际上是一对一关联）。

10.1.3. 每个子类一张表（Table per subclass），使用辨别标志（Discriminator）

注意，对“每个子类一张表”的映射策略，Hibernate 的实现不需要辨别字段，而其他的对象／关系映射工具使用了一种不同于Hibernate的实现方法，该方法要求在超类表中有一个类型辨别字

段（type discriminator column）。Hibernate 采用的方法更难实现，但从关系（数据库）的角度来看，按理说它更正确。若你愿意使用带有辨别字段的“每个子类一张表”的策略，你可以结合使用 <subclass> 与<join>，如下所示：

```
<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">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class
>
```

可选的声明 fetch="select"，是用来告诉 Hibernate，在查询超类时，不要使用外部连接（outer join）来抓取子类 ChequePayment 的数据。

10.1.4. 混合使用“每个类分层结构一张表”和“每个子类一张表”

你甚至可以采取如下方法混和使用“每个类分层结构一张表”和“每个子类一张表”这两种策略：

```
<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">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
```

```
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class
>
```

对上述任何一种映射策略而言，指向根类 Payment 的关联是使用 <many-to-one> 进行映射的。

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

10.1.5. 每个具体类一张表（Table per concrete class）

对于“每个具体类一张表”的映射策略，可以采用两种方法。第一种方法是使用 <union-subclass>。

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class
>
```

这里涉及三张与子类相关的表。每张表为对应类的所有属性（包括从超类继承的属性）定义相应字段。

这种方式的局限在于，如果一个属性在超类中做了映射，其字段名必须与所有子类表中定义的相同。（我们可能会在 Hibernate 的后续发布版本中放宽此限制。）不允许在联合子类（union subclass）的继承层次中使用标识生成器策略（identity generator strategy），实际上，主键的种子（primary key seed）不得不为同一继承层次中的全部被联合子类所共用。

假若超类是抽象类，请使用 abstract="true"。当然，假若它不是抽象的，需要一个额外的表（上面的例子中，默认是 PAYMENT），来保存超类的实例。

10.1.6. 每个具体类一张表，使用隐式多态

另一种可供选择的方法是采用隐式多态：

```
<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
>
```

请注意，这里没有显性地提及 `Payment` 接口。`Payment` 的属性映射到每个子类。如果你想避免重复，请考虑使用 XML 实体（如：`DOCTYPE` 声明里的 `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` 和映射里的 `&allproperties;`）。

这种方法的缺陷在于，在 Hibernate 执行多态查询时（polymorphic queries）无法生成带 `UNION` 的 SQL 语句。

对于这种映射策略而言，通常用 `<any>` 来实现到 `Payment` 的多态关联映射。

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any
>
```

10.1.7. 隐式多态和其他继承映射混合使用

对这一映射还有一点需要注意。因为每个子类都在各自独立的元素 <class> 中映射（并且 Payment 只是一个接口），每个子类可以很容易的成为另一个继承体系中的一部分！（你仍然可以对接口 Payment 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
>
```

我们还是没有明确的提到 Payment。如果我们针对接口 Payment 执行查询 — 如 from Payment — Hibernate 自动返回 CreditCardPayment（和它的子类，因为 它们也实现了接口 Payment）、CashPayment 和 Chequepayment 的实例，但不返回 NonelectronicTransaction 的实例。

10.2. 限制

对“每个具体类映射一张表”（table per concrete-class）的映射策略而言，隐式多态的方式有一定的限制。而 <union-subclass> 映射的限制则没有那么严格。

下面表格中列出了在 Hibernte 中“每个具体类一张表”的策略和隐式多态的限制。

表 10.1. 继承映射特性 (Features of inheritance mappings)

继承策略 (Inherit strategy)	多态多对多	多态一对一	多态一对多	多态多对多	Polymorph load()/ get()	多态查询	多态连接 (join)	支持外连接 (Outer join) 读取。
每个类分层结构 一张表 (table per class hierarchy)	<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	supported
table per subclass	<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	supported
每个具体类 一张表 (union-subclass)	<many-to-one>	<one-to-one>	<one-to-many> (仅适用于 inverse="true")	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p	supported
每个具体类 一张表 (隐式多态)	<any>	not supported	not supported	<many-to-any>	s.createCriteria(Payment.class).add(Restrictions.idEq(id))	from Payment p	not supported	not supported

与对象共事

Hibernate 是完整的对象/关系映射解决方案，它提供了对象状态管理（state management）的功能，使开发者不再需要理会底层数据库系统的细节。也就是说，相对于常见的 JDBC/SQL 持久层方案中需要管理 SQL 语句，Hibernate 采用了更自然的面向对象的视角来持久化 Java 应用中的数据。

换句话说，使用 Hibernate 的开发者应该总是关注对象的状态（state），不必考虑 SQL 语句的执行。这部分细节已经由 Hibernate 掌管妥当，只有开发者在进行系统性能调优的时候才需要进行了解。

11.1. Hibernate 对象状态（object states）

Hibernate 定义并支持下列对象状态（state）：

- 瞬时（Transient） — 由 new 操作符创建，且尚未与Hibernate Session 关联的对象被认定为瞬时（Transient）的。瞬时（Transient）对象不会被持久化到数据库中，也不会被赋予持久化标识（identifier）。 如果瞬时（Transient）对象在程序中没有被引用，它会被垃圾回收器（garbage collector）销毁。 使用 Hibernate Session可以将其变为持久（Persistent）状态。（Hibernate会自动执行必要的SQL语句）
- 持久（Persistent） — 持久（Persistent）的实例在数据库中有对应的记录，并拥有一个持久化标识（identifier）。 持久（Persistent）的实例可能是刚被保存的，或刚被加载的，无论哪一种，按定义，它存在于相关联的Session作用范围内。 Hibernate会检测到处于持久（Persistent）状态的对象的任何改动，在当前操作单元（unit of work）执行完毕时将对象数据（state）与数据库同步（synchronize）。 开发者不需要手动执行UPDATE。将对象从持久（Persistent）状态变成瞬时（Transient）状态同样也不需要手动执行 DELETE 语句。
- 脱管（Detached） — 与持久（Persistent）对象关联的Session被关闭后，对象就变为脱管（Detached）的。对脱管（Detached）对象的引用依然有效，对象可继续被修改。脱管（Detached）对象如果重新关联到某个新的 Session 上， 会再次转变为持久（Persistent）的（在Detached其间的改动将被持久化到数据库）。 这个功能使得一种编程模型，即中间会给用户思考时间（user think-time）的长时间运行的操作单元（unit of work）的编程模型成为可能。我们称之为应用程序事务，即从用户观点看是一个操作单元（unit of work）。

接下来我们来细致地讨论下状态（states）及状态间的转换（state transitions）（以及触发状态转换的 Hibernate 方法）。

11.2. 使对象持久化

Hibernate 认为持久化类（persistent class）新实例化的对象是瞬时（Transient）的。我们可通过将瞬时（Transient）对象与 session 关联而把它变为持久的（Persistent）。

```
DomesticCat fritz = new DomesticCat();
```

```
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

如果 Cat 的持久化标识（identifier）是 generated 类型的， 那么该标识（identifier）会自动在 save() 被调用时产生并分配给 cat。如果 Cat 的持久化标识（identifier）是assigned类型的，或是一个复合主键（composite key），那么该标识（identifier）应当在调用 save() 之前手动赋予给 cat。你也可以按照 EJB3 early draft 中定义的语义，使用 persist() 替代save()。

- persist() 使一个临时实例持久化。然而，它不保证立即把标识符值分配给持久性实例，这会发生在冲刷（flush）的时候。persist() 也保证它在事务边界外调用时不会执行 INSERT 语句。这对于长期运行的带有扩展会话/持久化上下文的会话是很有用的。
- save() 保证返回一个标识符。如果需要运行 INSERT 来获取标识符（如 "identity" 而非 "sequence" 生成器），这个 INSERT 将立即执行，不管你是否在事务内部还是外部。这对于长期运行的带有扩展会话/持久化上下文的会话来说会出现问题。

此外，你可以用一个重载版本的 save() 方法。

```
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) );
```

如果你持久化的对象有关联的对象（associated objects）（例如上例中的 kittens 集合） 那么对这些对象（译注：pk 和 kittens）进行持久化的顺序是任意的（也就是说可以先对 kittens 进行持久化也可以先对 pk 进行持久化）， 除非你在外键列上有 NOT NULL 约束。 Hibernate 不会违反外键约束，但是如果你用错误的顺序持久化对象（译注：在 pk 持久化之前持久化kitten），那么可能会违反 NOT NULL 约束。

通常你不会为这些细节烦心，因为你很可能会使用 Hibernate 的传播性持久化（transitive persistence）功能自动保存相关联那些对象。这样连违反 NOT NULL 约束的情况都不会出现了 — Hibernate 会管好所有的事情。传播性持久化（transitive persistence）将在本章稍后讨论。

11.3. 装载对象

如果你知道某个实例的持久化标识（identifier），你就可以使用 Session 的 load() 方法来获取它。load() 的另一个参数是指定类的对象。本方法会创建指定类的持久化实例，并从数据库加载其数据（state）。

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

此外，你可以把数据（state）加载到指定的对象实例上（覆盖掉该实例原来的数据）。

```
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异常（unrecoverable exception）。如果类的映射使用了代理（proxy），load() 方法会返回一个未初始化的代理，直到你调用该代理的某方法时才会去访问数据库。 若你希望在某对象中创建一个指向另一个对象的关联，又不想在从数据库中装载该对象时同时装载相关联的那个对象，那么这种操作方式就用得上的了。如果为相应类映射关系设置了 batch-size，那么使用这种操作方式允许多个对象被一批装载（因为返回的是代理，无需从数据库中抓取所有对象的数据）。

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

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

你甚至可以选用某个 LockMode，用 SQL 的 SELECT ... FOR UPDATE 装载对象。 请查阅 API 文档以获取更多信息。

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

注意，任何关联的对象或者包含的集合都不会被以 FOR UPDATE 方式返回， 除非你指定了 lock 或者 all 作为关联（association）的级联风格（cascade style）。

任何时候都可以使用 refresh() 方法强迫装载对象和它的集合。如果你使用数据库触发器功能来处理对象的某些属性，这个方法就很有用了。

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

How much does Hibernate load from the database and how many SQL SELECTs will it use? This depends on the fetching strategy. This is explained in 第 21.1 节 “抓取策略（Fetching strategies）”.

11.4. 查询

如果不知道所要寻找的对象的持久化标识，那么你需要使用查询。Hibernate 支持强大且易于使用的面向对象查询语言（HQL）。如果希望通过编程的方式创建查询，Hibernate 提供了完善的按条件（Query By Criteria，QBC）以及按样例（Query By Example，QBE）进行查询的功能。你也可以用原生 SQL（native SQL）描述查询，Hibernate 额外提供了将结果集（result set）转化为对象的支持。

11.4.1. 执行查询

HQL 和原生 SQL（native SQL）查询要通过为 org.hibernate.Query 的实例来表达。这个接口提供了参数绑定、结果集处理以及运行实际查询的方法。你总是可以通过当前 Session 获取一个 Query 对象：

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

一个查询通常在调用 list() 时被执行，执行结果会完全装载进内存中的一个集合（collection）。查询返回的对象处于持久（persistent）状态。如果你知道的查询只会返回一个对象，可使用 list() 的快捷方式 uniqueResult()。注意，使用集合预先抓取的查询往往会返回多次根对象（他们的集合类都被初始化了）。你可以通过一个集合（Set）来过滤这些重复对象。

11.4.1.1. 迭代式获取结果（Iterating results）

某些情况下，你可以使用 iterate() 方法得到更好的性能。 这通常是你预期返回的结果在 session，或二级缓存（second-level cache）中已经存在时的情况。如若不然，iterate() 会比 list() 慢，而且可能简单查询也需要进行多次数据库访问：iterate() 会首先使用 1 条语句得到所有对象的持久化标识（identifiers），再根据持久化标识执行 n 条附加的 select 语句实例化实际的对象。

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
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;
    }
}
```

11.4.1.2. 返回元组（tuples）的查询

（译注：元组（tuples）指一条结果行包含多个对象） Hibernate 查询有时返回元组（tuples），每个元组（tuples）以数组的形式返回：

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = (Cat) tuple[0];
    Cat mother = (Cat) tuple[1];
    ....
}
```

11.4.1.3. 标量（Scalar）结果

查询可在 select 从句中指定类的属性，甚至可以调用 SQL 统计（aggregate）函数。属性或统计结果被认定为"标量（Scalar）"的结果（而不是持久（persistent state）的实体）。

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();
```

```
while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

11.4.1.4. 绑定参数

接口 Query 提供了对命名参数（named parameters）、JDBC 风格的问号（?）参数进行绑定的方法。不同于 JDBC，Hibernate 对参数从 0 开始计数。 命名参数（named parameters）在查询字符串中是形如 :name 的标识符。命名参数（named parameters）的优点是：

- 命名参数（named parameters）与其在查询串中出现的顺序无关
- 它们可在同一查询串中多次出现
- 它们本身是自我说明的

```
//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();
```

11.4.1.5. 分页

如果你需要指定结果集的范围（希望返回的最大行数/或开始的行数），应该使用 Query 接口提供的方法：

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

```
List cats = q.list();
```

Hibernate 知道如何将这个有限定条件的查询转换成你的数据库的原生 SQL（native SQL）。

11.4.1.6. 可滚动遍历（Scrollable iteration）

如果你的 JDBC 驱动支持可滚动的 ResultSet，Query 接口可以使用 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) );

}
cats.close()
```

请注意，使用此功能需要保持数据库连接（以及游标（cursor））处于一直打开状态。如果你需要断开连接使用分页功能，请使用 setMaxResult()/setFirstResult()。

11.4.1.7. 外置命名查询（Externalizing named queries）

Queries can also be configured as so called named queries using annotations or Hibernate mapping documents. @NamedQuery and @NamedQueries can be defined at the class level as seen in 例 11.1 “Defining a named query using @NamedQuery” . However their definitions are global to the session factory/entity manager factory scope. A named query is defined by its name and the actual query string.

例 11.1. Defining a named query using @NamedQuery

```
@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}
```

```
public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}
```

Using a mapping document can be configured using the <query> node. Remember to use a CDATA section if your query contains characters that could be interpreted as markup.

例 11.2. Defining a named query using <query>

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

Parameter binding and executing is done programatically as seen in 例 11.3 “Parameter binding of a named query” .

例 11.3. Parameter binding of a named query

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

请注意实际的程序代码与所用的查询语言无关，你也可在元数据中定义原生 SQL（native SQL）查询，或将原有的其他的查询语句放在配置文件中，这样就可以让 Hibernate 统一管理，达到迁移的目的。

也请注意在 <hibernate-mapping> 元素中声明的查询必须有一个全局唯一的名字，而在 <class> 元素中声明的查询自动具有全局名，是通过类的全名加以限定的。比如 eg.Cat.ByNameAndMaximumWeight。

11.4.2. 过滤集合

集合过滤器（filter）是一种用于一个持久化集合或者数组的特殊的查询。查询字符串中可以使用 "this" 来引用集合中的当前元素。

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
```

```
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

返回的集合可以被认为是一个包（bag，无顺序可重复的集合（collection）），它是所给集合的副本。 原来的集合不会被改动（这与“过滤器（filter）”的隐含的含义不符，不过与我们期待的行为一致）。

请注意过滤器（filter）并不需要 `from` 子句（当然需要的话它们也可以加上）。过滤器（filter）不限定于只能返回集合元素本身。

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

即使无条件的过滤器（filter）也是有意义的。例如，用于加载一个大集合的子集：

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

11.4.3. 条件查询（Criteria queries）

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 提供了直观的 Criteria 查询 API。

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Restrictions.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

The Criteria and the associated Example API are discussed in more detail in 第 17 章 条件查询（Criteria Queries）.

11.4.4. 使用原生 SQL 的查询

你可以使用 `createSQLQuery()` 方法，用 SQL 来描述查询，并由 Hibernate 将结果集转换成对象。请注意，你可以在任何时候调用 `session.connection()` 来获得并使用 JDBC Connection 对象。 如果你选择使用 Hibernate 的 API，你必须把 SQL 别名用大括号包围起来：

```
List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10")
```

```
    .addEntity("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 {cat} WHERE ROWNUM<10")
    .addEntity("cat", Cat.class)
.list()
```

SQL queries can contain named and positional parameters, just like Hibernate queries. More information about native SQL queries in Hibernate can be found in 第 18 章 Native SQL 查询.

11.5. 修改持久对象

事务中的持久实例（就是通过 session 装载、保存、创建或者查询出的对象） 被应用程序操作所造成的任何修改都会在 Session 被刷出（flushed）的时候被持久化（本章后面会详细讨论）。这里不需要调用某个特定的方法（比如 update()，设计它的目的是不同的）将你的修改持久化。所以最直接的更新一个对象的方法就是在 Session 处于打开状态时 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
```

有时这种程序模型效率低下，因为它在同一 Session 里需要一条 SQL SELECT 语句（用于加载对象） 以及一条 SQL UPDATE 语句（持久化更新的状态）。为此 Hibernate 提供了另一种途径，使用脱管（detached）实例。

11.6. 修改脱管（Detached）对象

很多程序需要在某个事务中获取对象，然后将对象发送到界面层去操作，最后在一个新的事务保存所做的修改。在高并发访问的环境中使用这种方式，通常使用附带版本信息的数据来保证这些“长“工作单元之间的隔离。

Hibernate 通过提供 Session.update() 或 Session.merge() 重新关联脱管实例的办法来支持这种模型。

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

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

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

如果具有 catId 持久化标识的 Cat 之前已经被另一Session (secondSession) 装载了， 应用程序进行重关联操作 (reattach) 的时候会抛出一个异常。

如果你确定当前 session 没有包含与之具有相同持久化标识的持久实例，使用 update()。如果想随时合并你的的改动而不考虑 session 的状态，使用 merge()。换句话说，在一个新 session 中通常第一个调用的是 update() 方法，以便保证重新关联脱管 (detached) 对象的操作首先被执行。

The application should individually update() detached instances that are reachable from the given detached instance only if it wants their state to be updated. This can be automated using transitive persistence. See 第 11.11 节 “传播性持久化 (transitive persistence)” for more information.

lock() 方法也允许程序重新关联某个对象到一个新 session 上。不过，该脱管 (detached) 的对象必须是没有修改过的。

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

请注意，lock() 可以搭配多种 LockMode，更多信息请阅读 API 文档以及关于事务处理 (transaction handling) 的章节。重新关联不是 lock() 的唯一用途。

Other models for long units of work are discussed in 第 13.3 节 “乐观并发控制 (Optimistic concurrency control)”.

11.7. 自动状态检测

Hibernate 的用户曾要求一个既可自动分配新持久化标识 (identifier) 保存瞬时 (transient) 对象，又可更新/重新关联脱管 (detached) 实例的通用方法。saveOrUpdate() 方法实现了这个功能。

```
// 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() 用途和语义可能会使新用户感到迷惑。首先，只要你没有尝试在某个 session 中使用来自另一 session 的实例，你就应该不需要使用 update()， saveOrUpdate()，或 merge()。有些程序从来不用这些方法。

通常下面的场景会使用 update() 或 saveOrUpdate()：

- 程序在第一个 session 中加载对象
- 该对象被传递到表现层
- 对象发生了一些改动
- 该对象被返回到业务逻辑层
- 程序调用第二个 session 的 update() 方法持久这些改动

saveOrUpdate() 做下面的事：

- 如果对象已经在本 session 中持久化了，不做任何事
- 如果另一个与本 session 关联的对象拥有相同的持久化标识（identifier），抛出一个异常
- 如果对象没有持久化标识（identifier）属性，对其调用 save()
- 如果对象的持久标识（identifier）表明其是一个新实例化的对象，对其调用 save()。
- 如果对象是附带版本信息的（通过 <version> 或 <timestamp>）并且版本属性的值表明其是一个新实例化的对象，save() 它。
- 否则 update() 这个对象

merge() 可非常不同：

- 如果 session 中存在相同持久化标识（identifier）的实例，用用户给出的对象的状态覆盖旧有的持久实例
- 如果 session 没有相应的持久实例，则尝试从数据库中加载，或创建新的持久化实例
- 最后返回该持久实例
- 用户给出的这个对象没有被关联到 session 上，它依旧是脱管的

11.8. 删除持久对象

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已删除对象的引用。所以，最好这样理解：delete() 的用途是把一个持久实例变成瞬时（transient）实例。

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

你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。例如你删除了父对象，但是忘记删除其子对象。

11.9. 在两个不同数据库间复制对象

偶尔会用到不重新生成持久化标识（identifier），将持久实例以及其关联的实例持久到不同的数据库中的操作。

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

ReplicationMode 决定在和数据库中已存在记录由冲突时，replicate() 如何处理。

- ReplicationMode.IGNORE：当某个现有数据库记录具有相同标识符时忽略它
- ReplicationMode.OVERWRITE：用相同的标识符覆盖现有数据库记录
- ReplicationMode.EXCEPTION：当某个现有数据库记录具有相同标识符时抛出异常
- ReplicationMode.LATEST_VERSION：如果当前的版本较新，则覆盖，否则忽略

这个功能的用途包括使录入的数据在不同数据库中一致，产品升级时升级系统配置信息，回滚 non-ACID 事务中的修改等等。（译注，non-ACID，非 ACID;ACID，Atomic，Consistent，Isolated and Durable 的缩写）

11.10. Session 刷出（flush）

每间隔一段时间，Session 会执行一些必需的 SQL 语句来把内存中的对象的状态同步到 JDBC 连接中。这个过程被称为刷出（flush），默认会在下面的时间点执行：

- 在某些查询执行之前
- 在调用 org.hibernate.Transaction.commit() 的时候
- 在调用 Session.flush() 的时候

涉及的 SQL 语句会按照下面的顺序发出执行：

1. 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
2. 所有对实体进行更新的语句
3. 所有进行集合删除的语句
4. 所有对集合元素进行删除，更新或者插入的语句
5. 所有进行集合插入的语句

6. 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

有一个例外是，如果对象使用 native 方式来生成 ID（持久化标识）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 何时会执行这些 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，Query.list(..) 绝对不会返回已经失效的数据，也不会返回错误数据。

It is possible to change the default behavior so that flush occurs less frequently. The FlushMode class defines three different modes: only flush at commit time when the Hibernate Transaction API is used, flush automatically using the explained routine, or never flush unless flush() is called explicitly. The last mode is useful for long running units of work, where a Session is kept open and disconnected for a long time (see 第 13.3.2 节 “扩展周期的 session 和自动版本化”).

```
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);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

During flush, an exception might occur (e.g. if a DML operation violates a constraint). Since handling exceptions involves some understanding of Hibernate's transactional behavior, we discuss it in 第 13 章 事务和并发 .

11.11. 传播性持久化（transitive persistence）

对每一个对象都要执行保存，删除或重关联操作让人感觉有点麻烦，尤其是在处理许多彼此关联的对象的时候。一个常见的例子是父子关系。考虑下面的例子：

如果一个父子关系中的子对象是值类型（value typed）（例如，地址或字符串的集合）的，他们的生命周期会依赖于父对象，可以享受方便的级联操作（Cascading），不需要额外的动作。父对象被保存时，这些值类型（value typed）子对象也将被保存；父对象被删除时，子对象也将被删除。这对将一个子对象从集合中移除是同样有效：Hibernate 会检测到，并且因为值类型（value typed）的对象不可能被其他对象引用，所以 Hibernate 会在数据库中删除这个子对象。

现在考虑同样的场景，不过父子对象都是实体（entities）类型，而非值类型（value typed）（例如，类别与个体，或母猫和小猫）。实体有自己的生命期，允许共享对其的引用（因此从集合中移除一个实体，不意味着它可以被删除），并且实体到其他关联实体之间默认没有级联操作

的设置。 Hibernate 默认不实现所谓的可到达即持久化（persistence by reachability）的策略。

每个 Hibernate session 的基本操作 — 包括 persist(), merge(), saveOrUpdate(), delete(), lock(), refresh(), evict(), replicate() — 都有对应的级联风格（cascade style）。这些级联风格（cascade style）风格分别命名为 create, merge, save-update, delete, lock, refresh, evict, replicate。如果你希望一个操作被顺着关联关系级联传播，你必须在映射文件中指出这一点。例如：

```
<one-to-one name="person" cascade="persist"/>
```

级联风格（cascade style）是可组合的：

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

你可以使用 cascade="all" 来指定全部操作都顺着关联关系级联（cascaded）。默认值是 cascade="none"，即任何操作都不会被级联（cascaded）。

In case you are using annotatons you probably have noticed the cascade attribute taking an array of CascadeType as a value. The cascade concept in JPA is very is similar to the transitive persistence and cascading of operations as described above, but with slightly different semantics and cascading types:

- CascadeType.PERSIST: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed
- CascadeType.MERGE: cascades the merge operation to associated entities if merge() is called or if the entity is managed
- CascadeType.REMOVE: cascades the remove operation to associated entities if delete() is called
- CascadeType.REFRESH: cascades the refresh operation to associated entities if refresh() is called
- CascadeType.DETACH: cascades the detach operation to associated entities if detach() is called
- CascadeType.ALL: all of the above

注意

CascadeType.ALL also covers Hibernate specific operations like save-update, lock etc...

A special cascade style, delete-orphan, applies only to one-to-many associations, and indicates that the delete() operation should be applied to any child object that is removed from the association. Using annotations there is no CascadeType.DELETE-ORPHAN equivalent. Instead you can use the attribute orphanRemoval as seen in 例 11.4 “@OneToMany with orphanRemoval”. If an entity is removed from a @OneToMany collection or an associated entity is dereferenced from a @OneToOne association, this associated entity can be marked for deletion if orphanRemoval is set to true.

例 11.4. @OneToMany with orphanRemoval

```
@Entity
public class Customer {
   private Set<Order> orders;

   @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)
   public Set<Order> getOrders() { return orders; }

   public void setOrders(Set<Order> orders) { this.orders = orders; }

   [...]
}

@Entity
public class Order { ... }

Customer customer = em.find(Customer.class, 1l);
Order order = em.find(Order.class, 1l);
customer.getOrders().remove(order); //order will be deleted by cascade
```

建议：

- It does not usually make sense to enable cascade on a many-to-one or many-to-many association. In fact the @ManyToOne and @ManyToMany don't even offer a orphanRemoval attribute. Cascading is often useful for one-to-one and one-to-many associations.
- If the child object's lifespan is bounded by the lifespan of the parent object, make it a life cycle object by specifying cascade="all,delete-orphan"(@OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)).
- 其他情况，你可根本不需要级联（cascade）。但是如果你认为你会经常在某个事务中同时用到父对象与子对象，并且你希望少打点儿字，可以考虑使用 cascade="persist,merge,save-update"。

可以使用 cascade="all" 将一个关联关系（无论是对值对象的关联，或者对一个集合的关联）标记为父/子关系的关联。 这样对父对象进行 save/update/delete 操作就会导致子对象也进行 save/update/delete 操作。

Furthermore, a mere reference to a child from a persistent parent will result in save/update of the child. This metaphor is incomplete, however. A child which becomes unreferenced by its parent is not automatically deleted, except in the case of a one-to-

many association mapped with cascade="delete-orphan". The precise semantics of cascading operations for a parent/child relationship are as follows:

- 如果父对象被 persist()，那么所有子对象也会被 persist()
- 如果父对象被 merge()，那么所有子对象也会被 merge()
- 如果父对象被 save()，update() 或 saveOrUpdate()，那么所有子对象则会被 saveOrUpdate()
- 如果某个持久的父对象引用了瞬时（transient）或者脱管（detached）的子对象，那么子对象将会被 saveOrUpdate()
- 如果父对象被删除，那么所有子对象也会被 delete()
- 除非被标记为 cascade="delete-orphan"（删除“孤儿”模式，此时不被任何一个父对象引用的子对象会被删除），否则子对象失掉父对象对其的引用时，什么事也不会发生。如果有特殊需要，应用程序可通过显式调用 delete() 删除子对象。

最后，注意操作的级联可能是在调用期（call time）或者写入期（flush time）作用到对象图上的。所有的操作，如果允许，都在操作被执行的时候级联到可触及的关联实体上。然而，save-upate 和 delete-orphan 是在Session flush 的时候才作用到所有可触及的被关联对象上的。

11.12. 使用元数据

Hibernate 中有一个非常丰富的元级别（meta-level）的模型，含有所有的实体和值类型数据的元数据。 有时这个模型对应用程序本身也会非常有用。比如说，应用程序可能在实现一种“智能”的深度拷贝算法时，通过使用 Hibernate 的元数据来了解哪些对象应该被拷贝（比如，可变的值类型数据），那些不应该（不可变的值类型数据，也许还有某些被关联的实体）。

Hibernate 提供了 ClassMetadata 接口，CollectionMetadata 接口和 Type 层次体系来访问元数据。可以通过 SessionFactory 获取元数据接口的实例。

```
Cat 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
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] );
    }
}
```

Read-only entities

重要

Hibernate's treatment of read-only entities may differ from what you may have encountered elsewhere. Incorrect usage may cause unexpected results.

When an entity is read-only:

- Hibernate does not dirty-check the entity's simple properties or single-ended associations;
- Hibernate will not update simple properties or updatable single-ended associations;
- Hibernate will not update the version of the read-only entity if only simple properties or single-ended updatable associations are changed;

In some ways, Hibernate treats read-only entities the same as entities that are not read-only:

- Hibernate cascades operations to associations as defined in the entity mapping.
- Hibernate updates the version if the entity has a collection with changes that dirties the entity;
- A read-only entity can be deleted.

Even if an entity is not read-only, its collection association can be affected if it contains a read-only entity.

For details about the affect of read-only entities on different property and association types, see 第 12.2 节 “Read-only affect on property type”.

For details about how to make entities read-only, see 第 12.1 节 “Making persistent entities read-only”

Hibernate does some optimizing for read-only entities:

- It saves execution time by not dirty-checking simple properties or single-ended associations.
- It saves memory by deleting database snapshots.

12.1. Making persistent entities read-only

Only persistent entities can be made read-only. Transient and detached entities must be put in persistent state before they can be made read-only.

Hibernate provides the following ways to make persistent entities read-only:

- you can map an entity class as immutable; when an entity of an immutable class is made persistent, Hibernate automatically makes it read-only. see 第 12.1.1 节 “Entities of immutable classes” for details
- you can change a default so that entities loaded into the session by Hibernate are automatically made read-only; see 第 12.1.2 节 “Loading persistent entities as read-only” for details
- you can make an HQL query or criteria read-only so that entities loaded when the query or criteria executes, scrolls, or iterates, are automatically made read-only; see 第 12.1.3 节 “Loading read-only entities from an HQL query/criteria” for details
- you can make a persistent entity that is already in the in the session read-only; see 第 12.1.4 节 “Making a persistent entity read-only” for details

12.1.1. Entities of immutable classes

When an entity instance of an immutable class is made persistent, Hibernate automatically makes it read-only.

An entity of an immutable class can created and deleted the same as an entity of a mutable class.

Hibernate treats a persistent entity of an immutable class the same way as a read-only persistent entity of a mutable class. The only exception is that Hibernate will not allow an entity of an immutable class to be changed so it is not read-only.

12.1.2. Loading persistent entities as read-only

注意

Entities of immutable classes are automatically loaded as read-only.

To change the default behavior so Hibernate loads entity instances of mutable classes into the session and automatically makes them read-only, call:

```
Session.setDefaultReadOnly( true );
```

To change the default back so entities loaded by Hibernate are not made read-only, call:

```
Session.setDefaultReadOnly( false );
```

You can determine the current setting by calling:

```
Session.isDefaultReadOnly();
```

If Session.isDefaultReadOnly() returns true, entities loaded by the following are automatically made read-only:

- Session.load()
- Session.get()
- Session.merge()
- executing, scrolling, or iterating HQL queries and criteria; to override this setting for a particular HQL query or criteria see 第 12.1.3 节 “Loading read-only entities from an HQL query/criteria”

Changing this default has no effect on:

- persistent entities already in the session when the default was changed
- persistent entities that are refreshed via Session.refresh(); a refreshed persistent entity will only be read-only if it was read-only before refreshing
- persistent entities added by the application via Session.persist(), Session.save(), and Session.update() Session.saveOrUpdate()

12.1.3. Loading read-only entities from an HQL query/criteria

注意

Entities of immutable classes are automatically loaded as read-only.

If Session.isDefaultReadOnly() returns false (the default) when an HQL query or criteria executes, then entities and proxies of mutable classes loaded by the query will not be read-only.

You can override this behavior so that entities and proxies loaded by an HQL query or criteria are automatically made read-only.

For an HQL query, call:

```
Query.setReadOnly( true );
```

Query.setReadOnly(true) must be called before Query.list(), Query.uniqueResult(), Query.scroll(), or Query.iterate()

For an HQL criteria, call:

```
Criteria.setReadOnly( true );
```

Criteria.setReadOnly(true) must be called before Criteria.list(), Criteria.uniqueResult(), or Criteria.scroll()

Entities and proxies that exist in the session before being returned by an HQL query or criteria are not affected.

Uninitialized persistent collections returned by the query are not affected. Later, when the collection is initialized, entities loaded into the session will be read-only if Session.isDefaultReadOnly() returns true.

Using Query.setReadOnly(true) or Criteria.setReadOnly(true) works well when a single HQL query or criteria loads all the entities and intializes all the proxies and collections that the application needs to be read-only.

When it is not possible to load and initialize all necessary entities in a single query or criteria, you can temporarily change the session default to load entities as read-only before the query is executed. Then you can explicitly initialize proxies and collections before restoring the session default.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

setDefaultReadOnly( true );
Contract contract =
   ( Contract ) session.createQuery(
           "from Contract where customerName = 'Sherman'" )
           .uniqueResult();
Hibernate.initialize( contract.getPlan() );
Hibernate.initialize( contract.getVariations() );
Hibernate.initialize( contract.getNotes() );
setDefaultReadOnly( false );
...
tx.commit();
```

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

If Session.isDefaultReadOnly() returns true, then you can use Query.setReadOnly(false) and Criteria.setReadOnly(false) to override this session setting and load entities that are not read-only.

12.1.4. Making a persistent entity read-only

注意

Persistent entities of immutable classes are automatically made read-only.

To make a persistent entity or proxy read-only, call:

```
Session.setReadOnly(entityOrProxy, true)
```

To change a read-only entity or proxy of a mutable class so it is no longer read-only, call:

```
Session.setReadOnly(entityOrProxy, false)
```

重要

When a read-only entity or proxy is changed so it is no longer read-only, Hibernate assumes that the current state of the read-only entity is consistent with its database representation. If this is not true, then any non-flushed changes made before or while the entity was read-only, will be ignored.

To throw away non-flushed changes and make the persistent entity consistent with its database representation, call:

```
session.refresh( entity );
```

To flush changes made before or while the entity was read-only and make the database representation consistent with the current state of the persistent entity:

```
// evict the read-only entity so it is detached
session.evict( entity );
```

```
// make the detached entity (with the non-flushed changes) persistent
session.update( entity );

// now entity is no longer read-only and its changes can be flushed
s.flush();
```

12.2. Read-only affect on property type

The following table summarizes how different property types are affected by making an entity read-only.

表 12.1. Affect of read-only entity on property types

Property/Association Type	Changes flushed to DB?
Simple (第 12.2.1 节 “Simple properties”)	no*
Unidirectional one-to-one Unidirectional many-to-one (第 12.2.2.1 节 “Unidirectional one-to-one and many-to-one”)	no* no*
Unidirectional one-to-many Unidirectional many-to-many (第 12.2.2.2 节 “Unidirectional one-to-many and many-to-many”)	yes yes
Bidirectional one-to-one (第 12.2.3.1 节 “Bidirectional one-to-one”)	only if the owning entity is not read-only*
Bidirectional one-to-many/many-to-one inverse collection non-inverse collection (第 12.2.3.2 节 “Bidirectional one-to-many/many-to-one”)	only added/removed entities that are not read-only* yes
Bidirectional many-to-many (第 12.2.3.3 节 “Bidirectional many-to-many”)	yes

* Behavior is different when the entity having the property/association is read-only, compared to when it is not read-only.

12.2.1. Simple properties

When a persistent object is read-only, Hibernate does not dirty-check simple properties.

Hibernate will not synchronize simple property state changes to the database. If you have automatic versioning, Hibernate will not increment the version if any simple properties change.

```
Session session = factory.openSession();
Transaction tx = session.beginTransaction();

// get a contract and make it read-only
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );

// contract.getCustomerName() is "Sherman"
contract.setCustomerName( "Yogi" );
tx.commit();

tx = session.beginTransaction();

contract = ( Contract ) session.get( Contract.class, contractId );
// contract.getCustomerName() is still "Sherman"
...
tx.commit();
session.close();
```

12.2.2. Unidirectional associations

12.2.2.1. Unidirectional one-to-one and many-to-one

Hibernate treats unidirectional one-to-one and many-to-one associations in the same way when the owning entity is read-only.

We use the term unidirectional single-ended association when referring to functionality that is common to unidirectional one-to-one and many-to-one associations.

Hibernate does not dirty-check unidirectional single-ended associations when the owning entity is read-only.

If you change a read-only entity's reference to a unidirectional single-ended association to null, or to refer to a different entity, that change will not be flushed to the database.

注意

If an entity is of an immutable class, then its references to unidirectional single-ended associations must be assigned when that entity is first created. Because the entity is automatically made read-only, these references can not be updated.

If automatic versioning is used, Hibernate will not increment the version due to local changes to unidirectional single-ended associations.

In the following examples, Contract has a unidirectional many-to-one association with Plan. Contract cascades save and update operations to the association.

The following shows that changing a read-only entity's many-to-one association reference to null has no effect on the entity's database representation.

```
// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
contract.setPlan( null );
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );

// contract.getPlan() still refers to the original plan;

tx.commit();
session.close();
```

The following shows that, even though an update to a read-only entity's many-to-one association has no affect on the entity's database representation, flush still cascades the save-update operation to the locally changed association.

```
// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.beginTransaction();
Contract contract = ( Contract ) session.get( Contract.class, contractId );
session.setReadOnly( contract, true );
Plan newPlan = new Plan( "new plan"
contract.setPlan( newPlan);
tx.commit();

// get the same contract
tx = session.beginTransaction();
contract = ( Contract ) session.get( Contract.class, contractId );
```

```
newPlan = ( Contract ) session.get( Plan.class, newPlan.getId() );

// contract.getPlan() still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

tx.commit();
session.close();
```

12.2.2.2. Unidirectional one-to-many and many-to-many

Hibernate treats unidirectional one-to-many and many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks unidirectional one-to-many and many-to-many associations;

The collection can contain entities that are read-only, as well as entities that are not read-only.

Entities can be added and removed from the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in the collection if they dirty the owning entity.

12.2.3. Bidirectional associations

12.2.3.1. Bidirectional one-to-one

If a read-only entity owns a bidirectional one-to-one association:

- Hibernate does not dirty-check the association.
- updates that change the association reference to null or to refer to a different entity will not be flushed to the database.
- If automatic versioning is used, Hibernate will not increment the version due to local changes to the association.

注意

If an entity is of an immutable class, and it owns a bidirectional one-to-one association, then its reference must be assigned when that entity is first created. Because the entity is automatically made read-only, these references cannot be updated.

When the owner is not read-only, Hibernate treats an association with a read-only entity the same as when the association is with an entity that is not read-only.

12.2.3.2. Bidirectional one-to-many/many-to-one

A read-only entity has no impact on a bidirectional one-to-many/many-to-one association if:

- the read-only entity is on the one-to-many side using an inverse collection;
- the read-only entity is on the one-to-many side using a non-inverse collection;
- the one-to-many side uses a non-inverse collection that contains the read-only entity

When the one-to-many side uses an inverse collection:

- a read-only entity can only be added to the collection when it is created;
- a read-only entity can only be removed from the collection by an orphan delete or by explicitly deleting the entity.

12.2.3.3. Bidirectional many-to-many

Hibernate treats bidirectional many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

Hibernate dirty-checks bidirectional many-to-many associations.

The collection on either side of the association can contain entities that are read-only, as well as entities that are not read-only.

Entities are added and removed from both sides of the collection; changes are flushed to the database.

If automatic versioning is used, Hibernate will update the version due to changes in both sides of the collection if they dirty the entity owning the respective collections.

事务和并发

Hibernate 的事务和并发控制很容易掌握。Hibernate 直接使用 JDBC 连接和 JTA 资源，不添加任何附加锁定行为。我们强烈推荐你花点时间了解 JDBC 编程，ANSI SQL 查询语言和你使用的数据库系统的事务隔离规范。

Hibernate 不锁定内存中的对象。你的应用程序会按照你的数据库事务的隔离级别规定的那样运作。幸亏有了 Session，使得 Hibernate 通过标识符查找，和实体查询（不是返回标量值的报表查询）提供了可重复的读取（Repeatable reads）功能，Session 同时也是事务范围内的缓存（cache）。

除了对自动乐观并发控制提供版本管理，针对行级悲观锁定，Hibernate 也提供了辅助的（较小的）API，它使用了 SELECT FOR UPDATE 的 SQL 语法。本章后面会讨论乐观并发控制和这个API。

我们从 Configuration层、SessionFactory 层，和 Session 层开始讨论 Hibernate 的并行控制、数据库事务和应用程序的长事务。

13.1. Session 和事务范围（transaction scope）

SessionFactory 对象的创建代价很昂贵，它是线程安全的对象，它为所有的应用程序线程所共享。它只创建一次，通常是在应用程序启动的时候，由一个 Configuraion 的实例来创建。

Session 对象的创建代价比较小，是非线程安全的，对于单个请求，单个会话、单个的 工作单元而言，它只被使用一次，然后就丢弃。只有在需要的时候，一个 Session 对象 才会获取一个 JDBC 的 Connection（或一个Datasource）对象，因此假若不使用的时候它不消费任何资源。

此外我们还要考虑数据库事务。数据库事务应该尽可能的短，降低数据库中的锁争用。数据库长事务会阻止你的应用程序扩展到高的并发负载。因此，假若在用户思考期间让数据库事务开着，直到整个工作单元完成才关闭这个事务，这绝不是一个好的设计。

一个操作单元（Unit of work）的范围是多大？单个的 Hibernate Session 能跨越多个数据库事务吗？还是一个 Session 的作用范围对应一个数据库事务的范围？应该何时打开 Session，何时关闭 Session，你又如何划分数据库事务的边界呢？我们将在后续章节解决这些问题。

13.1.1. 操作单元（Unit of work）

First, let's define a unit of work. A unit of work is a design pattern described by Martin Fowler as “[maintaining] a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.”[PoEAA] In other words, its a series of operations we wish to carry out against the database together. Basically, it is a transaction, though fulfilling a unit of work will often span multiple physical database transactions (see 第 13.1.2 节 “长对话”). So really we are talking about a more abstract notion of a transaction. The term "business transaction" is also sometimes used in lieu of unit of work.

首先，别用 session-per-operation 这种反模式了，也就是说，在单个线程中， 不要因为一次简单的数据库调用，就打开和关闭一次 Session！数据库事务也是如此。 应用程序中的数据库调

用是按照计划好的次序，分组为原子的操作单元。（注意，这也意味着，应用程 序中，在单个的 SQL 语句发送之后，自动事务提交（auto-commit）模式失效了。这种模式专门为SQL控制台操作设计的。 Hibernate 禁止立即自动事务提交模式，或者期望应用服务器禁止立即自动事务提交模式。）数据库事务绝不是可有可无的，任何与数据库之间的通讯都必须在某个事务中进行，不管你是在读还是在写数据。对读数据而言，应该避免 auto-commit 行为，因为很多小的事务比一个清晰定义的工作单元性能差。后者也更容易维护和扩展。

在多用户的 client/server 应用程序中，最常用的模式是 每个请求一个会话（session-per-request）。 在这种模式下，来自客户端的请求被发送到服务器端（即 Hibernate 持久化层运行的地方），一个新的 Hibernate Session 被打开，并且执行这个操作单元中所有的数据库操作。一旦操作完成（同时对客户端的响应也准备就绪），session 被同步，然后关闭。你也可以使用单 个数据库事务来处理客户端请求，在你打开 Session 之后启动事务，在你关闭 Session 之前提交事务。会话和请求之间的关系是一对一的关系，这种模式对 于大多数应用程序来说是很棒的。

实现才是真正的挑战。Hibernate 内置了对"当前 session（current session）" 的管理，用于简化此模式。你要做的一切就是在服务器端要处理请求的时候，开启事务，在响应发送给客户之前结束事务。你可以用任何方式来完成这一操作，通常的方案有 ServletFilter，在 service 方法中进行 pointcut 的 AOP 拦截器，或者 proxy/interception 容器。EJB 容器是实现横切诸如 EJB session bean 上的事务分界，用 CMT 对事务进行声明等方面的标准手段。假若你决定使用编程式的事务分界，请参考本章后面讲到的 Hibernate Transaction API，这对易用性和代码可移植性都有好处。

Your application code can access a "current session" to process the request by calling sessionFactory.getCurrentSession(). You will always get a Session scoped to the current database transaction. This has to be configured for either resource-local or JTA environments, see 第 2.3 节 “上下文相关的会话（Contextual Session）”.

有时，将 Session 和数据库事务的边界延伸到"展示层被渲染后"会带来便利。有些 servlet 应用程序在对请求进行处理后，有个单独的渲染期，这种延伸对这种程序特别有用。假若你实现你自己的拦截器，把事务边界延伸到展示层渲染结束后非常容易。然而，假若你依赖有容器管理事务的 EJB，这就不太容易了，因为事务会在 EJB 方法返回后结束，而那是在任何展示层渲染开始之前。请访问 Hibernate 网站和论坛，你可以找到 Open Session in View 这一模式的提示和示例。

13.1.2. 长对话

session-per-request 模式不仅仅是一个可以用来设计操作单元的有用概念。很多业务处理都需要一系列完整的与用户之间的交互，而这些用户是指对数据库有交叉访问的用户。在基于 web 的应用和企业应用中，跨用户交互的数据库事务是无法接受的。考虑下面的例子：

- 在界面的第一屏，打开对话框，用户所看到的数据是被一个特定的 Session 和数据 库事务载入（load）的。用户可以随意修改对话框中的数据对象。
- 5 分钟后，用户点击“保存”，期望所做出的修改被持久化；同时他也期望自己是唯一修改这个信息的人，不会出现修改冲突。

从用户的角度来看，我们把这个操作单元称为长时间运行的对话（conversation），或者应用事务（application transaction）。在你的应用程序中，可以有很多种方法来实现它。

头一个幼稚的做法是，在用户思考的过程中，保持 Session 和数据库事务是打开的，保持数据库锁定，以阻止并发修改，从而保证数据库事务隔离级别和原子操作。这种方式当然是一个反模式，因为锁争用会导致应用程序无法扩展并发用户的数目。

很明显，我们必须使用多个数据库事务来实现这个对话。在这个例子中，维护业务处理的 事务隔离变成了应用程序层的部分责任。一个对话通常跨越多个数据库事务。如果仅仅只有一个数据库事务（最后的那个事务）保存更新过的数据，而所有其他事务只是单纯的读取数据（例如在一个跨越多个请求/响应周期的向导风格的对话框中），那么应用程序事务将保证其原子性。这种方式比听起来还要容易实现，特别是当你使用了 Hibernate 的下述特性的时候：

- 自动版本化：Hibernate 能够自动进行乐观并发控制，如果在用户思考的过程中发生并发修改，Hibernate 能够自动检测到。一般我们只在对话结束时才检查。
- 脱管对象（Detached Objects）：如果你决定采用前面已经讨论过的 session-per-request 模式，所有载入的实例在用户思考的过程中都处于与 Session 脱离的状态。Hibernate 允许你把与 Session 脱离的对象重新关联到 Session 上，并且对修改进行持久化，这种模式被称为 session-per-request-with-detached-objects。自动版本化被用来隔离并发修改。
- Extended（or Long）Session：Hibernate 的 Session 可以在数据库事务提交之后和底层的 JDBC 连接断开，当一个新的客户端请求到来的时候，它又重新连接上底层的 JDBC 连接。这种模式被称之为session-per-conversation，这种情况可 能会造成不必要的 Session 和 JDBC 连接的重新关联。自动版本化被用来隔离并发修改，Session 通常不允许自动 flush，而是显性地 flush。

session-per-request-with-detached-objects 和 session-per-conversation 各有优缺点，我们在本章后面乐观并发控制那部分再进行讨论。

13.1.3. 关注对象标识（Considering object identity）

应用程序可能在两个不同的 Session 中并发访问同一持久化状态，但是，一个持久化类的实例无法在两个 Session 中共享。因此有两种不同的标识语义：

数据库标识

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

JVM 标识

```
foo==bar
```

对于那些关联到 特定 Session（也就是在单个 Session 的范围内）上的对象来说，这两种标识的语义是等价的，与数据库标识对应的 JVM 标识是由 Hibernate 来保证的。不过，当应用程序在两个不同的 session 中并发访问具有同一持久化标识的业务对象实例的时候，这个业务对象的两个实例事实上是不相同的（从 JVM 识别来看）。这种冲突可以通过在同步和提交的时候使用自动版本化和乐观锁定方法来解决。

这种方式把关于并发的头疼问题留给了 Hibernate 和数据库；由于在单个线程内，操作单元中的对象识别不 需要代价昂贵的锁定或其他意义上的同步，因此它同时可以提供最好的可伸缩性。只要在单个线程只持有一个 Session，应用程序就不需要同步任何业务对象。在 Session 的范围内，应用程序可以放心的使用 == 进行对象比较。

不过，应用程序在 Session 的外面使用 == 进行对象比较可能会 导致无法预期的结果。在一些无法预料的场合，例如，如果你把两个脱管对象实例放进同一个 Set 的时候，就可能发生。这两个对象实例可能有同一个数据库标识（也就是说， 他们代表了表的同一行数据），从 JVM 标识的定义上来说，对脱管的对象而言，Hibernate 无法保证他们 的的 JVM 标识一致。开发人员必须覆盖持久化类的 equals() 方法和 hashCode() 方法，从而实现自定义的对象相等语义。警告：不要使用数据库标识来实现对象相等，应该使用业务键值，由唯一的，通常不变的属性组成。当一个瞬时对象被持久化的时候，它的数据库标识会发生改变。如果一个瞬时对象（通常也包括脱管对象实例）被放入一个 Set，改变它的 hashcode 会导致与这个 Set 的关系中断。虽 然业务键值的属性不象数据库主键那样稳定不变，但是你只需要保证在同一个 Set 中的对象属性的稳定性就足够了。请到 Hibernate 网站去寻求这个问题更多的详细的讨论。请注意，这不是一个有关 Hibernate 的问题，而仅仅是一个关于 Java 对象标识和判等行为如何实现的问题。

13.1.4. 常见问题

决不要使用反模式 session-per-user-session 或者 session-per-application（当然，这个规定几乎没有例外）。请注意，下述一些问题可能也会出现在我们推荐的模式中，在你作出某个设计决定之前，请务必理解该模式的应用前提。

- Session 对象是非线程安全的。如果一个 Session 实例允许共享的话，那些支持并发运行的东东，例如 HTTP request，session beans 或者是 Swing workers，将会导致出现资源争用（race condition）。如果在 HttpSession 中有 Hibernate 的 Session 的话（稍后讨论），你应该考虑同步访问你的 Http session。 否则，只要用户足够快的点击浏览器的“刷新”，就会导致两个并发运行线程使用同一个 Session。

- 一个由 Hibernate 抛出的异常意味着你必须立即回滚数据库事务，并立即关闭 Session（稍后会展开讨论）。如果你的 Session 绑定到一个应用程序上，你必须停止该应用程序。回滚数据库事务并不会把你的业务对象退回到事务启动时候的状态。这意味着数据库状态和业务对象状态不同步。通常情况下，这不是什么问题，因为异常是不可恢复的，你必须在回滚之后重新开始执行。

- The Session caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an OutOfMemoryException. One solution is to call clear() and evict() to manage the Session cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in 第 15 章 批量处理（Batch processing）. Keeping a Session open for the duration of a user session also means a higher probability of stale data.

13.2. 数据库事务声明

数据库（或者系统）事务的声明总是必须的。在数据库事务之外，就无法和数据库通讯（这可能会让那些习惯于自动提交事务模式的开发人员感到迷惑）。永远使用清晰的事务声明，即使只读操作也是如此。进行 显式的事务声明并不总是需要的，这取决于你的事务隔离级别和数据库的能力，但不管怎么说，声明事务总归有益无害。当然，一个单独的数据库事务总是比很多琐碎的事务性能更好，即时对读数据而言也是一样。

一个 Hibernate 应用程序可以运行在非托管环境中（也就是独立运行的应用程序，简单 Web 应用程序，或者Swing图形桌面应用程序），也可以运行在托管的 J2EE 环境中。在一个非托管环境中，Hibernate 通常自己负责管理数据库连接池。应用程序开发人员必须手工设置事务声明，换句话说，就是手工启 动，提交，或者回滚数据库事务。一个托管的环境通常提供了容器管理事务（CMT），例如事务装配通过可声明的方式定义在 EJB session beans 的部署描述符中。可编程式事务声明不再需要，即使是 Session 的同步也可以自动完成。

让持久层具备可移植性是人们的理想,这种移植发生在非托管的本地资源环境，与依赖 JTA 但是使用 BMT 而非 CMT 的系统之间。在两种情况下你都可以使用编程式的事务管理。Hibernate 提供了一套称为 Transaction 的封装 API， 用来把你的部署环境中的本地事务管理系统转换到 Hibernate 事务上。这个 API 是可选的，但是我们强烈推荐你使用，除非你用 CMT session bean。

通常情况下，结束 Session 包含了四个不同的阶段:

- 同步 session（flush，刷出到磁盘）
- 提交事务
- 关闭 session
- 处理异常

session 的同步（flush，刷出）前面已经讨论过了，我们现在进一步考察在托管和非托管环境下的事务声明和异常处理。

13.2.1. 非托管环境

如果 Hibernat 持久层运行在一个非托管环境中，数据库连接通常由 Hibernate 的简单（即非 DataSource）连接池机制 来处理。session/transaction 处理方式如下所示:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

你不需要显式 flush() Session 一 对 commit() 的调用会自动触发 session 的同步（取决于 session 的 第 11.10 节 “Session 刷出（flush）”）。调用 close() 标志 session 的结

束。close() 方法重要的暗示是，session 释放了 JDBC 连接。这段 Java 代码在非托管环境下和 JTA 环境下都可以运行。

更加灵活的方案是 Hibernate 内置的 "current session" 上下文管理，前文已经讲过：

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

你很可能从未在一个通常的应用程序的业务代码中见过这样的代码片断：致命的（系统）异常应该总是 在应用程序“顶层”被捕获。换句话说，执行 Hibernate 调用的代码（在持久层）和处理 RuntimeException 异常的代码（通常只能清理和退出应用程序）应该在不同 的应用程序逻辑层。Hibernate 的当前上下文管理可以极大地简化这一设计，你所有的一切就是 SessionFactory。异常处理将在本章稍后进行讨论。

请注意，你应该选择 org.hibernate.transaction.JDBCTransactionFactory （这是默认选项），对第二个例子来说，hibernate.current_session_context_class应该是 "thread"。

13.2.2. 使用 JTA

如果你的持久层运行在一个应用服务器中（例如，在 EJB session beans 的后面），Hibernate 获取的每个数据源连接将自动成为全局 JTA 事务的一部分。你可以安装一个独立的 JTA 实现，使用它而不使用 EJB。Hibernate 提供了两种策略进行 JTA 集成。

如果你使用 bean 管理事务（BMT），可以通过使用 Hibernate 的 Transaction API 来告诉应用服务器启动和结束 BMT 事务。因此，事务管理代码和在非托管环境下是一样的。

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
```

```
    throw e; // or display error message
}
finally {
    sess.close();
}
```

如果你希望使用与事务绑定的 Session，也就是使用 getCurrentSession() 来简化上下文管理，你将不得不直接使用 JTA UserTransaction API。

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

在 CMT 方式下，事务声明是在 session bean 的部署描述符中，而不需要编程。因此，代码被简化为：

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

在 CMT/EJB 中甚至会自动 rollback，因为假若有未捕获的 RuntimeException 从 session bean 方法中抛出，这就会通知容器把全局事务回滚。这就意味着，在 BMT 或者 CMT 中，你根本就不需要使用 Hibernate Transaction API，你自动得到了绑定到事务的“当前” Session。

注意，当你配置 Hibernate 的 transaction factory 的时候，在直接使用 JTA 的时候（BMT），你应该选择 org.hibernate.transaction.JTATransactionFactory，在 CMT session bean 中选择 org.hibernate.transaction.CMTTransactionFactory。记得也要设置 hibernate.transaction.manager_lookup_class。还有，确认你的 hibernate.current_session_context_class 未设置（为了向下兼容），或者设置为 "jta"。

getCurrentSession()在 JTA 环境中有一个弊端。对 after_statement 连接释放方式有一个警告，这是被默认使用的。因为 JTA 规范的一个很愚蠢的限制，Hibernate 不可能自动清理任何未关闭的 ScrollableResults 或者Iterator，它们是由 scroll() 或 iterate() 产生的。你 must 通过在

finally 块中，显式调用 ScrollableResults.close() 或者 Hibernate.close(Iterator) 方法来释放底层数据库游标。（当然，大部分程序完全可以很容易的避免在 JTA 或 CMT 代码中出现 scroll() 或 iterate()。）

13.2.3. 异常处理

如果 Session 抛出异常（包括任何 SQLException），你应该立即回滚数据库事务，调用 Session.close() ，丢弃该 Session 实例。Session 的某些方法可能会导致 session 处于不一致的状态。所有由 Hibernate 抛出的异常都视为不可以恢复的。确保在 finally 代码块中调用 close() 方法，以关闭掉 Session。

HibernateException 是一个非检查期异常（这不同于 Hibernate 老的版本），它封装了 Hibernate 持久层可能出现的大多数错误。我们的观点是，不应该强迫应用程序开发人员 在底层捕获无法恢复的异常。在大多数软件系统中，非检查期异常和致命异常都是在相应方法调用 的堆栈的顶层被处理的（也就是说，在软件上面的逻辑层），并且提供一个错误信息给应用软件的用户 （或者采取其他某些相应的操作）。请注意，Hibernate 也有可能抛出其他并不属于 HibernateException 的非检查期异常。这些异常同样也是无法恢复的，应该 采取某些相应的操作去处理。

在和数据库进行交互时，Hibernate 把捕获的 SQLException 封装为 Hibernate 的 JDBCException。事实上，Hibernate 尝试把异常转换为更有实际含义的 JDBCException 异常的子类。底层的 SQLException 可以通过 JDBCException.getCause() 来得到。Hibernate 通过使用关联到 SessionFactory 上的 SQLExceptionConverter 来把 SQLException 转换为一个对应的 JDBCException 异常的子类。默认情况下，SQLExceptionConverter 可以通过配置 dialect 选项指定；此外，也可以使用用户自定义的实现类（参考 javadocs SQLExceptionConverterFactory 类来了解详情）。标准的 JDBCException 子类型是：

- JDBCConnectionException：指明底层的 JDBC 通讯出现错误。
- SQLGrammarException：指明发送的 SQL 语句的语法或者格式错误。
- ConstraintViolationException：指明某种类型的约束违例错误
- LockAcquisitionException：指明了在执行请求操作时，获取所需的锁级别时出现的错误。
- GenericJDBCException：不属于任何其他种类的原生异常。

13.2.4. 事务超时

EJB 这样的托管环境有一项极为重要的特性，而它从未在非托管环境中提供过，那就是事务超时。在出现错误的事务行为的时候，超时可以确保不会无限挂起资源、对用户没有交代。在托管（JTA）环境之外，Hibernate 无法完全提供这一功能。但是，Hiberante 至少可以控制数据访问，确保数据库级别的死锁，和返回巨大结果集的查询被限定在一个规定的时间内。在托管环境中，Hibernate 会把事务超时转交给 JTA。这一功能通过 Hibernate Transaction 对象进行抽象。

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();
```

```
    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

注意 setTimeout() 不应该在 CMT bean 中调用，此时事务超时值应该是被声明式定义的。

13.3. 乐观并发控制（Optimistic concurrency control）

唯一能够同时保持高并发和高可伸缩性的方法就是使用带版本化的乐观并发控制。版本检查使用版本号、 或者时间戳来检测更新冲突（并且防止更新丢失）。Hibernate 为使用乐观并发控制的代码提供了三种可 能的方法，应用程序在编写这些代码时，可以采用它们。我们已经在前面应用程序对话那部分展示了 乐观并发控制的应用场景，此外，在单个数据库事务范围内，版本检查也提供了防止更新丢失的好处。

13.3.1. 应用程序级别的版本检查（Application version checking）

未能充分利用 Hibernate 功能的实现代码中，每次和数据库交互都需要一个新的 Session，而且开发人员必须在显示数据之前从数据库中重新载入所有的持久化对象实例。这种方式迫使应用程序自己实现版本检查来确保对话事务的隔离，从数据访问的角度来说是最低效的。这种使用方式和 entity EJB 最相似。

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

version 属性使用 <version> 来映射，如果对象是脏数据，在同步的时候，Hibernate 会自动增加版本号。

当然，如果你的应用是在一个低数据并发环境下，并不需要版本检查的话，你照样可以使用这种方式，只不过跳过版本检查就是了。在这种情况下，最晚提交生效 （last commit wins）就是你

的长对话的默认处理策略。请记住这种策略可能会让应用软件的用户感到困惑，因为他们有可能会碰上更新丢失掉却没有出错信息，或者需要合并更改冲突的情况。

很明显，手工进行版本检查只适合于某些软件规模非常小的应用场景，对于大多数软件应用场景来说并不现实。通常情况下，不仅是单个对象实例需要进行版本检查，整个被修改过的关联对象图也都需要进行版本检查。作为标准设计范例，Hibernate 使用扩展周期的 Session 的方式，或者脱管对象实例的方式来提供自动版本检查。

13.3.2. 扩展周期的 session 和自动版本化

单个 Session 实例和它所关联的所有持久化对象实例都被用于整个对话，这被称为 session-per-conversation。Hibernate 在同步的时候进行对象实例的版本检查，如果检测到并发修改则抛出异常。由开发人员来决定是否需要捕获和处理这个异常（通常的抉择是给用户 提供一个合并更改，或者在无脏数据情况下重新进行业务对话的机会）。

在等待用户交互的时候， Session 断开底层的 JDBC 连接。这种方式以数据库访问的角度来说是最高效的方式。应用程序不需要关心版本检查或脱管对象实例的重新关联，在每个数据库事务中，应用程序也不需要载入读取对象实例。

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

foo 对象知道它是在哪个 Session 中被装入的。在一个旧 session 中开启一个新的数据库事务，会导致 session 获取一个新的连接，并恢复 session 的功能。将数据库事务提交，使得 session 从 JDBC 连接断开，并将此连接交还给连接池。在重新连接之后，要强制对你没有更新的数据进行一次版本检查，你可以对所有可能被其他事务修改过的对象，使用参数 LockMode.READ 来调用 Session.lock()。你不用 lock 任何你正在更新的数据。一般你会在扩展的 Session 上设置 FlushMode.NEVER，因此只有最后一个数据库事务循环才会真正的把整个对话中发生的修改发送到数据库。因此，只有这最后一次数据库事务才会包含 flush() 操作，然后在整个对话结束后，还要 close() 这个 session。

如果在用户思考的过程中，Session 因为太大了而不能保存，那么这种模式是有问题的。举例来说，一个 HttpSession 应该尽可能的小。由于 Session 是一级缓存，并且保持了所有被载入过的对象，因此我们只应该在那些少量的 request/response 情况下使用这种策略。你应该只把一个 Session 用于单个对话，因为它很快就会出现脏数据。

注意

注意，早期的 Hibernate 版本需要明确的对 Session 进行 disconnect 和 reconnect。这些方法现在已经过时了，打开事务和关闭事务会起到同样的效果。

此外，也请注意，你应该让与数据库连接断开的 Session 对持久层保持关闭状态。换句话说，在三层环境中，使用有状态的 EJB session bean 来持 有Session， 而不要把它传递到 web 层（甚至把它序列化到一个单独的层），保存在 HttpSession 中。

扩展 session 模式，或者被称为每次对话一个session（session-per-conversation），自动管理当前 session 上下文联用的时候会更困难。你需要提供你自己的 CurrentSessionContext 实现。请参阅 Hibernate Wiki 以获得示例。

13.3.3. 脱管对象（deatched object）和自动版本化

这种方式下，与持久化存储的每次交互都发生在一个新的 Session 中。然而，同一持久化对象实例可以在多次与数据库的交互中重用。应用程序操纵脱管对象实例 的状态，这个脱管对象实例最初是在另一个 Session 中载入的，然后调用 Session.update()，Session.saveOrUpdate()，或者 Session.merge() 来重新关联该对象实例。

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

Hibernate 会再一次在同步的时候检查对象实例的版本，如果发生更新冲突，就抛出异常。

如果你确信对象没有被修改过，你也可以调用 lock() 来设置 LockMode.READ（绕过所有的缓存，执行版本检查），从而取代 update() 操作。

13.3.4. 定制自动版本化行为

对于特定的属性和集合，通过为它们设置映射属性 optimistic-lock 的值为 false，来禁止 Hibernate 的版本自动增加。这样的话，如果该属性脏数据，Hibernate 将不再增加版本号。

遗留系统的数据库 Schema 通常是静态的，不可修改的。或者，其他应用程序也可能访问同一数据库，根本无法得知如何处理版本号，甚至时间戳。在以上的所有场景中，实现版本化不能依靠数据库表的某个特定列。在 <class> 的映射中设置 optimistic-lock="all" 可以在没有版本或者时间戳属性映射的情况下实现版本检查，此时 Hibernate 将比较一行记录的每个字段的状态。请注意，只有当 Hibernate 能够比较新旧状态的情况下，这种方式才能生效，也就是说，你必须使用单个长生命周期 Session 模式，而不能使用 session-per-request-with-detached-objects 模式。

有些情况下，只要更改不发生交错，并发修改也是允许的。当你在 <class> 的映射中设置 optimistic-lock="dirty"，Hibernate 在同步的时候将只比较有脏数据的字段。

在以上所有场景中，不管是专门设置一个版本/时间戳列，还是进行全部字段/脏数据字段比较，Hibernate 都会针对每个实体对象发送一条 UPDATE（带有相应的 WHERE 语句 ）的 SQL 语句来执行版本检查和数据更新。如果你对关联实体 设置级联关系使用传播性持久化（transitive

persistence），那么 Hibernate 可能会执行不必 要的update语句。这通常不是个问题，但是数据库里面对 on update 点火 的触发器可能在脱管对象没有任何更改的情况下被触发。因此，你可以在 <class> 的映射中，通过设置select-before-update="true" 来定制这一行为，强制 Hibernate SELECT 这个对象实例，从而保证，在更新记录之前，对象的确是被修改过。

13.4. 悲观锁定（Pessimistic Locking）

用户其实并不需要花很多精力去担心锁定策略的问题。通常情况下，只要为 JDBC 连接指定一下隔离级别，然后让数据库去搞定一切就够了。然而，高级用户有时候希望进行一个排它的悲观锁定，或者在一个新的事务启动的时候，重新进行锁定。

Hibernate 总是使用数据库的锁定机制，从不在内存中锁定对象。

类 LockMode 定义了 Hibernate 所需的不同的锁定级别。一个锁定可以通过以下的机制来设置:

- 当 Hibernate 更新或者插入一行记录的时候，锁定级别自动设置为 LockMode.WRITE。
- 当用户显式的使用数据库支持的 SQL 格式 SELECT ... FOR UPDATE 发送 SQL 的时候，锁定级别设置为 LockMode.UPGRADE。
- 当用户显式的使用 Oracle 数据库的 SQL 语句 SELECT ... FOR UPDATE NOWAIT 的时候，锁定级别设置 LockMode.UPGRADE_NOWAIT。
- 当 Hibernate 在“可重复读”或者是“序列化”数据库隔离级别下读取数据的时候，锁定模式自动设置为 LockMode.READ。这种模式也可以通过用户显式指定进行设置。
- LockMode.NONE 代表无需锁定。在 Transaction 结束时， 所有的对象都切换到该模式上来。与 session 相关联的对象通过调用 update() 或者 saveOrUpdate() 脱离该模式。

"显式的用户指定"可以通过以下几种方式之一来表示:

- 调用 Session.load() 的时候指定锁定模式（LockMode）。
- 调用 Session.lock()。
- 调用 Query.setLockMode()。

如果在 UPGRADE 或者 UPGRADE_NOWAIT 锁定模式下调用 Session.load()，并且要读取的对象尚未被 session 载入过，那么对象通过 SELECT ... FOR UPDATE 这样的 SQL 语句被载入。如果为一个对象调用 load() 方法时，该对象已经在另一个较少限制的锁定模式下被载入了，那么 Hibernate 就对该对象调用 lock() 方法。

如果指定的锁定模式是 READ，UPGRADE 或 UPGRADE_NOWAIT，那么 Session.lock() 就执行版本号检查。（在 UPGRADE 或者 UPGRADE_NOWAIT 锁定模式下，执行 SELECT ... FOR UPDATE这样的SQL语句。）

如果数据库不支持用户设置的锁定模式，Hibernate 将使用适当的替代模式（而不是扔出异常）。这一点可以确保应用程序的可移植性。

13.5. 连接释放模式（Connection Release Modes）

Hibernate 关于 JDBC 连接管理的旧（2.x）行为是，Session 在第一次需要的时候获取一个连接，在 session 关闭之前一直会持有这个连接。Hibernate 引入了连接释放的概念，来告诉 session

如何处理它的 JDBC 连接。注意，下面的讨论只适用于采用配置 ConnectionProvider 来提供连接的情况，用户自己提供的连接与这里的讨论无关。通过 org.hibernate.ConnectionReleaseMode 的不同枚举值来使用不用的释放模式：

- ON_CLOSE：基本上就是上面提到的老式行为。Hibernate session 在第一次需要进行 JDBC 操作的时候获取连接，然后持有它，直到 session 关闭。
- AFTER_TRANSACTION：在 org.hibernate.Transaction 结束后释放连接。
- AFTER_STATEMENT（也被称做积极释放）：在每一条语句被执行后就释放连接。但假若语句留下了与 session 相关的资源，那就不会被释放。目前唯一的这种情形就是使用 org.hibernate.ScrollableResults。

hibernate.connection.release_mode 配置参数用来指定使用哪一种释放模式。可能的值有：

- auto（默认）：这一选择把释放模式委派给 org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode() 方法。对 JTATransactionFactory 来说，它会返回 ConnectionReleaseMode.AFTER_STATEMENT；对 JDBCTransactionFactory 来说，则是ConnectionReleaseMode.AFTER_TRANSACTION。很少需要修改这一默认行为，因为假若设置不当，就会带来 bug，或者给用户代码带来误导。
- on_close：使用 ConnectionReleaseMode.ON_CLOSE。这种方式是为了向下兼容的,但是已经完全不被鼓励使用了。
- after_transaction：使用 ConnectionReleaseMode.AFTER_TRANSACTION。这一设置不应该在 JTA 环境下使用。也要注意，使用 ConnectionReleaseMode.AFTER_TRANSACTION 的时候，假若 session 处于 auto-commit 状态，连接会像 AFTER_STATEMENT 那样被释放。
- after_statement：使用 ConnectionReleaseMode.AFTER_STATEMENT。除此之外，会查询配置的 ConnectionProvider，是否它支持这一设置（supportsAggressiveRelease()）。假若不支持，释放模式会被设置为 ConnectionReleaseMode.AFTER_TRANSACTION。只有在你每次调用 ConnectionProvider.getConnection() 获取底层 JDBC 连接的时候，都可以确信获得同一个连接的时候，这一设置才是安全的；或者在 auto-commit 环境中，你可以不管是否每次都获得同一个连接的时候，这才是安全的。

拦截器与事件（Interceptors and events）

应用程序能够响应 Hibernate 内部产生的特定事件是非常有用的。这样就允许实现某些通用的功能以及允许对 Hibernate 功能进行扩展。

14.1. 拦截器（Interceptors）

Interceptor 接口提供了从会话（session）回调（callback）应用程序（application）的机制， 这种回调机制可以允许应用程序在持久化对象被保存、更新、删除或是加载之前，检查并（或）修改其 属性。一个可能的用途，就是用来跟踪审核（auditing）信息。例如：下面的这个拦截器，会在一个实现了 Auditable 接口的对象被创建时自动地设置 createTimestamp 属性，并在实现了 Auditable 接口的对象被更新时，同步更新 lastUpdateTimestamp 属性。

你可以直接实现 Interceptor 接口，也可以（最好）继承自 EmptyInterceptor。

```
package org.hibernate.test;

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

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    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) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        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 afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
         System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

拦截器可以有两种：Session 范围内的和 SessionFactory 范围内的。

当使用某个重载的 SessionFactory.openSession() 使用 Interceptor 作为参数调用打开一个 session 的时候，就指定了 Session 范围内的拦截器。

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

SessionFactory 范围内的拦截器要通过 Configuration 中注册，而这必须在创建 SessionFactory 之前。在这种情况下，给出的拦截器会被这个 SessionFactory 所打开的所有 session 使用了；除非 session 打开时明确指明了使用的拦截器。SessionFactory 范围内的拦截器，必须是线程安全的。确保你没有保存 session 专有的状态，因为多个 session 可能并发使用这个拦截器。

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

14.2. 事件系统（Event system）

如果需要响应持久层的某些特殊事件，你也可以使用 Hibernate3 的事件框架。该事件系统可以用来替代拦截器，也可以作为拦截器的补充来使用。

基本上，Session 接口的每个方法都有相对应的事件。比如 LoadEvent，FlushEvent，等等（查阅 XML 配置文件的 DTD，以及 org.hibernate.event 包来获得所有已定义的事件的列表）。当某个方 法被调用时，Hibernate Session会生成一个相对应的事件并激活所 有配置好的事件监听器。系统预设的监听器实现的处理过程就是被监听的方法要做的（被监听的方法所做的其实仅仅是激活监听器，“实际”的工作是由监听器完成的）。不过，你可以自由地选择实现 一个自己定制的监听器（比如，实现并注册用来处理处理 LoadEvent 的 LoadEventListener 接口），来负责处理所有的调用 Session 的 load() 方法的请求。

监听器应该被看作是单例（singleton）对象，也就是说，所有同类型的事件的处理共享同一个监听器实例，因此监听器不应该保存任何状态（也就是不应该使用成员变量）。

用户定制的监听器应该实现与所要处理的事件相对应的接口，或者从一个合适的基类继承（甚至是从 Hibernate 自带的默认事件监听器类继承，为了方便你这样做，这些类都被声明成 non-final 的了）。用户定制的监听器可以通过编程使用 Configuration 对象 来注册，也可以在 Hibernate的 XML 格式的配置文件中进行声明（不支持在 Properties 格式的配置文件声明监听器）。下面是一个用户定制的加载事件（load event）的监听器：

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

你还需要修改一处配置，来告诉 Hibernate，除了默认的监听器，还要附加选定的监听器。

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
```

```
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration
>
```

或者，你可以通过编程的方式来注册它：

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

通过在XML配置文件声明而注册的监听器不能共享实例。如果在多个 <listener/> 节点中使用 了相同的类的名字，则每一个引用都将会产生一个独立的实例。如果你需要在多个监听器类型之间共享 监听器的实例，则你必须使用编程的方式来进行注册。

为什么我们实现了特定监听器的接口，在注册的时候还要明确指出我们要注册哪个事件的监听器呢？ 这是因为一个类可能实现多个监听器的接口。在注册的时候明确指定要监听的事件，可以让启用或者禁用对某个事件的监听的配置工作简单些。

14.3. Hibernate 的声明式安全机制

通常，Hibernate 应用程序的声明式安全机制由会话外观层（session facade）所管理。现在，Hibernate3允许某些特定的行为由 JACC 进行许可管理，由 JAAS 进行授权管理。本功能是一个建立在事件框架之上的可选的功能。

首先，你必须要配置适当的事件监听器（event listener），来激活使用 JAAS 管理授权的功能。

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

注意，<listener type="..." class="..."/> 只是 <event type="..."><listener class="..."/></event> 的简写，对每一个事件类型都必须严格的有一个监听器与之对应。

接下来，仍然在 hibernate.cfg.xml 文件中，绑定角色的权限：

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

这些角色的名字就是你的 JACC provider 所定义的角色的名字。

批量处理（Batch processing）

使用 Hibernate 将 100,000 条记录插入到数据库的一个很天真的做法可能是这样的：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

这段程序大概运行到 50，000 条记录左右会失败并抛出内存溢出异常（OutOfMemoryException）。这是因为 Hibernate 把所有新插入的客户（Customer）实例在 session 级别的缓存区进行了缓存的缘故。

我们会在本章告诉你如何避免此类问题。首先，如果你要执行批量处理并且想要达到一个理想的性能，那么使用 JDBC 的批量（batching）功能是至关重要。将 JDBC 的批量抓取数量（batch size）参数设置到一个合适值（比如，10 - 50 之间）：

```
hibernate.jdbc.batch_size 20
```

注意，假若你使用了 identiy 标识符生成器，Hibernate 在 JDBC 级别透明的关闭插入语句的批量执行。

你也可能想在执行批量处理时完全关闭二级缓存：

```
hibernate.cache.use_second_level_cache false
```

但是，这不是绝对必须的，因为我们可以显式设置 CacheMode 来关闭与二级缓存的交互。

15.1. 批量插入（Batch inserts）

如果要将很多对象持久化，你必须通过经常的调用 flush() 以及稍后调用 clear() 来控制第一级缓存的大小。

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

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
```

```
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

15.2. 批量更新（Batch updates）

此方法同样适用于检索和更新数据。此外，在进行会返回很多行数据的查询时，你需要使用 scroll() 方法以便充分利用服务器端游标所带来的好处。

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

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

15.3. StatelessSession（无状态 session）接口

作为选择，Hibernate 提供了基于命令的 API，可以用 detached object 的形式把数据以流的方法加入到数据库，或从数据库输出。StatelessSession 没有持久化上下文，也不提供多少高层的生命周期语义。特别是，无状态 session 不实现第一级 cache，也不和第二级缓存，或者查询缓存交互。它不实现事务化写，也不实现脏数据检查。用 stateless session 进行的操作甚至不级联到关联实例。stateless session 忽略集合类（Collections）。通过 stateless session 进行的操作不触发 Hibernate 的事件模型和拦截器。无状态 session 对数据的混淆现象免疫，因为它没有第一级缓存。无状态 session 是低层的抽象，和低层 JDBC 相当接近。

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
```

```
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

注意在上面的例子中，查询返回的 Customer 实例立即被脱管（detach）。它们与任何持久化上下文都没有关系。

StatelessSession 接口定义的 insert()，update() 和 delete() 操作是直接的数据库行级别操作，其结果是立刻执行一条 INSERT，UPDATE 或 DELETE 语句。因此，它们的语义和 Session 接口定义的 save()，saveOrUpdate() 和delete() 操作有很大的不同。

15.4. DML（数据操作语言）风格的操作（DML-style operations）

就像已经讨论的那样，自动和透明的对象/关系映射（object/relational mapping）关注于管理对象的状态。这就意味着对象的状态存在于内存，因此直接操作（使用 SQL Data Manipulation Language（DML,数据操作语言）语句：INSERT ,UPDATE 和 DELETE）数据库中的数据将不会影响内存中的对象状态和对象数据。不过，Hibernate 提供通过 Hibernate 查询语言（HQL）来执行大批量 SQL 风格的 DML 语句的方法。

UPDATE 和 DELETE 语句的伪语法为：(UPDATE | DELETE) FROM? EntityName (WHERE where_conditions)?。

要注意的事项：

- 在 FROM 子句（from-clause）中，FROM 关键字是可选的
- 在 FROM 子句（from-clause）中只能有一个实体名，它可以是别名。如果实体名是别名，那么任何被引用的属性都必须加上此别名的前缀；如果不是别名，那么任何有前缀的属性引用都是非法的。
- 不能在大批量 HQL 语句中使用 joins 连接（显式或者隐式的都不行）。不过在 WHERE 子句中可以使用子查询。可以在 where 子句中使用子查询，子查询本身可以包含 join。
- 整个 WHERE 子句是可选的。

举个例子，使用 Query.executeUpdate() 方法执行一个 HQL UPDATE语句（方法命名是来源于 JDBC 的 PreparedStatement.executeUpdate()）：

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

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
```

```
        .executeUpdate();
tx.commit();
session.close();
```

HQL UPDATE 语句，默认不会影响更新实体的 version 或 the timestamp 属性值。这和 EJB3 规范是一致的。但是，通过使用 versioned update，你可以强制 Hibernate 正确的重置version 或者 timestamp 属性值。这通过在 UPDATE 关键字后面增加 VERSIONED 关键字来实现的。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

注意，自定义的版本类型（org.hibernate.usertype.UserVersionType）不允许和 update versioned 语句联用。

执行一个 HQL DELETE，同样使用 Query.executeUpdate() 方法：

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

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

由 Query.executeUpdate() 方法返回的整型值表明了受此操作影响的记录数量。注意这个数值可能与数据库中被（最后一条 SQL 语句）影响了的“行”数有关，也可能没有。一个大批量 HQL 操作可能导致多条实际的SQL语句被执行，举个例子，对 joined-subclass 映射方式的类进行的此类操作。这个返回值代表了实际被语句影响了的记录数量。在那个 joined-subclass 的例子中， 对一个子类的删除实际上可能不仅仅会删除子类映射到的表而且会影响“根”表，还有可能影响与之有继承关系的 joined-subclass 映射方式的子类的表。

INSERT 语句的伪码是：INSERT INTO EntityName properties_list select_statement。要注意的是：

- 只支持 INSERT INTO ... SELECT ... 形式，不支持 INSERT INTO ... VALUES ... 形式。

  properties_list 和 SQL INSERT 语句中的字段定义（column speficiation）类似。对参与继承树映射的实体而言，只有直接定义在给定的类级别的属性才能直接在 properties_list 中使用。超类的属性不被支持；子类的属性无意义。换句话说，INSERT 天生不支持多态性。

- select_statement 可以是任何合法的 HQL 选择查询，不过要保证返回类型必须和要插入的类型完全匹配。目前，这一检查是在查询编译的时候进行的，而不是把它交给数据库。注意，在 HibernateType 间如果只是等价（equivalent）而非相等（equal），会导致问题。定义为 org.hibernate.type.DateType 和 org.hibernate.type.TimestampType 的两个属性可能会产生类型不匹配错误，虽然数据库级可能不加区分或者可以处理这种转换。
- 对 id 属性来说，insert 语句给你两个选择。你可以明确地在 properties_list 表中指定 id 属性（这样它的值是从对应的 select 表达式中获得），或者在 properties_list 中省略它（此时使用生成指）。后一种选择只有当使用在数据库中生成值的 id 产生器时才能使用；如果是“内存”中计算的类型生成器，在解析时会抛出一个异常。注意，为了说明这一问题，数据库产生值的生成器是 org.hibernate.id.SequenceGenerator（和它的子类），以及任何 org.hibernate.id.PostInsertIdentifierGenerator 接口的实现。这儿最值得注意的意外是 org.hibernate.id.TableHiLoGenerator，它不能在此使用，因为它没有得到其值的途径。
- 对映射为 version 或 timestamp 的属性来说，insert 语句也给你两个选择，你可以在 properties_list 表中指定（此时其值从对应的 select 表达式中获得），或者在 properties_list 中省略它（此时，使用在 org.hibernate.type.VersionType 中定义的 seed value（种子值））。

下面是一个执行 HQL INSERT 语句的例子：

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

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer
 c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

HQL: Hibernate 查询语言

Hibernate 配备了一种非常强大的查询语言，这种语言看上去很像 SQL。但是不要被语法结构上的相似所迷惑，HQL 是非常有意识的被设计为完全面向对象的查询，它可以理解如继承、多态和关联之类的概念。

16.1. 大小写敏感性问题

除了 Java 类与属性的名称外，查询语句对大小写并不敏感。 所以 SeLeCT 与 sELEct 以及 SELECT 是相同的，但是 org.hibernate.eg.FOO 并不等价于 org.hibernate.eg.Foo 并且 foo.barSet 也不等价于 foo.BARSET。

本手册中的 HQL 关键字将使用小写字母。很多用户发现使用完全大写的关键字会使查询语句的可读性更强，但我们发现，当把查询语句嵌入到 Java 语句中的时候使用大写关键字比较难看。

16.2. from 子句

Hibernate 中最简单的查询语句的形式如下：

```
from eg.Cat
```

该子句简单的返回 eg.Cat 类的所有实例。通常我们不需要使用类的全限定名，因为 auto-import（自动引入）是缺省的情况。所以我们几乎只使用如下的简单写法：

```
from Cat
```

为了在这个查询的其他部分里引用 Cat，你将需要分配一个别名。例如：

```
from Cat as cat
```

这个语句把别名 cat 指定给类Cat 的实例，这样我们就可以在随后的查询中使用此别名了。关键字 as 是可选的，我们也可以这样写：

```
from Cat cat
```

子句中可以同时出现多个类，其查询结果是产生一个笛卡儿积或产生跨表的连接。

```
from Formula, Parameter
```

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

查询语句中别名的开头部分小写被认为是实践中的好习惯，这样做与 Java 变量的命名标准保持了一致（比如，domesticCat）。

16.3. 关联（Association）与连接（Join）

我们也可以为相关联的实体甚至是对一个集合中的全部元素指定一个别名，这时要使用关键字 join。

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

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

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

受支持的连接类型是从 ANSI SQL 中借鉴来的:

- inner join
- left outer join
- right outer join
- full join（全连接，并不常用）

语句 inner join, left outer join 以及 right outer join 可以简写。

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

通过 HQL 的 with 关键字，你可以提供额外的 join 条件。

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight
> 10.0
```

A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select. This is particularly useful in the

case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See 第 21.1 节 “抓取策略（Fetching strategies）” for more information.

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

一个 fetch 连接通常不需要被指定别名，因为相关联的对象不应当被用在 where 子句（或其它任何子句）中。同时，相关联的对象并不在查询的结果中直接返回，但可以通过他们的父对象来访问到他们。

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

假若使用 iterate() 来调用查询，请注意 fetch 构造是不能使用的（scroll() 可以使用）。fetch 也不应该与 setMaxResults() 或 setFirstResult() 共用，这是因为这些操作是基于结果集的，而在预先抓取集合类时可能包含重复的数据，也就是说无法预先知道精确的行数。fetch 还不能与独立的 with 条件一起使用。通过在一次查询中 fetch 多个集合，可以制造出笛卡尔积，因此请多加注意。对 bag 映射来说，同时 join fetch 多个集合角色可能在某些情况下给出并非预期的结果，也请小心。最后注意，使用 full join fetch 与 right join fetch 是没有意义的。

如果你使用属性级别的延迟获取（lazy fetching）（这是通过重新编写字节码实现的），可以使用 fetch all properties 来强制 Hibernate 立即取得那些原本需要延迟加载的属性（在第一个查询中）。

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

16.4. join 语法的形式

HQL 支持两种关联 join 的形式：implicit（隐式） 与 explicit（显式）。

上一节中给出的查询都是使用 explicit（显式）形式的，其中 form 子句中明确给出了 join 关键字。这是建议使用的方式。

implicit（隐式）形式不使用 join 关键字。关联使用"点号"来进行“引用”。implicit join 可以在任何 HQL 子句中出现。implicit join 在最终的 SQL 语句中以 inner join 的方式出现。

```
from Cat as cat where cat.mate.name like '%s%'
```

16.5. 引用 identifier 属性

通常有两种方法来引用实体的 identifier 属性:

- 特殊属性 (lowercase) id 可以用来引用实体的 identifier 属性 假设这个实体没有定义用 non-identifier 属性命名的 id。
- 如果这个实体定义了 identifier 属性, 你可以使用属性名。

对组合 identifier 属性的引用遵循相同的命名规则。如果实体有一个 non-identifier 属性命名的 id, 这个组合 identifier 属性只能用自己定义的名字来引用; 否则, 特殊 id 属性可以用来引用 identifier 属性。

重要

注意: 从 3.2.2 版本开始, 这已经改变了很多。在前面的版本里, 不管实际的名字, id 总是指向 identifier 属性; 而用 non-identifier 属性命名的 id 就从来不在 Hibernate 查询里引用。

16.6. select 子句

select 子句选择将哪些对象与属性返回到查询结果集中。考虑如下情况:

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

该语句将选择其它 Cat 的 mate (其他猫的配偶) 。实际上, 你可以更简洁的用以下的查询语句表达相同的含义:

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

查询语句可以返回值为任何类型的属性, 包括返回类型为某种组件 (Component) 的属性:

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

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

查询语句可以返回多个对象和（或）属性，存放在 Object[] 队列中，

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

或存放在一个 List 对象中：

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

假设类 Family 有一个合适的构造函数 - 作为实际的类型安全的 Java 对象：

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

你可以使用关键字 as 给“被选择了的表达式”指派别名：

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

这种做法在与子句 select new map 一起使用时最有用：

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

该查询返回了一个 Map 的对象，内容是别名与被选择的值组成的名-值映射。

16.7. 聚集函数

HQL 查询甚至可以返回作用于属性之上的聚集函数的计算结果：

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

受支持的聚集函数如下：

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

你可以在选择子句中使用数学操作符、连接以及经过验证的 SQL 函数:

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

关键字 distinct 与 all 也可以使用, 它们具有与 SQL 相同的语义。

```
select distinct cat.name from Cat cat

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

16.8. 多态查询

一个如下的查询语句:

```
from Cat as cat
```

不仅返回 Cat 类的实例, 也同时返回子类 DomesticCat 的实例。Hibernate 可以在 from 子句中指定任何 Java 类或接口。查询会返回继承了该类的所有持久化子类的实例或返回声明了该接口的所有持久化类的实例。下面的查询语句返回所有的被持久化的对象:

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

接口 Named 可能被各种各样的持久化类声明:

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

注意, 最后的两个查询将需要超过一个的 SQL SELECT。这表明 order by 子句没有对整个结果集进行正确的排序。(这也说明你不能对这样的查询使用 Query.scroll() 方法。)

16.9. where 子句

where 子句允许你将返回的实例列表的范围缩小。如果没有指定别名，你可以使用属性名来直接引用属性：

```
from Cat where name='Fritz'
```

如果指派了别名，需要使用完整的属性名：

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

返回名为（属性 name 等于）'Fritz' 的 Cat 类的实例。

下面的查询：

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

将返回所有满足下面条件的 Foo 类的实例： 存在如下的 bar 的一个实例，其 date 属性等于 Foo 的 startDate 属性。复合路径表达式使得 where 子句非常的强大，考虑如下情况：

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

该查询将被翻译成为一个含有表连接（内连接）的 SQL 查询。如果你打算写像这样的查询语句：

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

在 SQL 中，你为达此目的将需要进行一个四表连接的查询。

= 运算符不仅可以被用来比较属性的值，也可以用来比较实例：

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

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

The special property (lowercase) id can be used to reference the unique identifier of an object. See 第 16.5 节 “引用 identifier 属性” for more information.

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

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

第二个查询是有效的。此时不需要进行表连接。

同样也可以使用复合标识符。比如 Person 类有一个复合标识符，它由 country 属性与 medicareNumber 属性组成:

```
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
```

第二个查询也不需要进行表连接。

See 第 16.5 节 “引用 identifier 属性” for more information regarding referencing identifier properties)

同样的，特殊属性 class 在进行多态持久化的情况下被用来存取一个实例的鉴别值（discriminator value）。一个嵌入到 where 子句中的 Java 类的名字将被转换为该类的鉴别值。

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

You can also use components or composite user types, or properties of said component types. See 第 16.17 节 “组件” for more information.

一个“任意”类型有两个特殊的属性 id 和 class，来允许我们按照下面的方式表达一个连接（AuditLog.item 是一个属性，该属性被映射为 <any>）。

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

注意，在上面的查询与句中，log.item.class 和 payment.class 将涉及到完全不同的数据库中的列。

16.10. 表达式

在 where 子句中允许使用的表达式包括 大多数你可以在 SQL 使用的表达式种类:

- 数学运算符 +, -, *, /
- 二进制比较运算符 =, >=, <=, <>, !=, like
- 逻辑运算符 and, or, not
- 括号 ( ), 表示分组
- in, not in, between, is null, is not null, is empty, is not empty, member of and not member of
- "Simple" case, case ... when ... then ... else ... end, and "searched" case, case when ... then ... else ... end
- 字符串连接符 ...||... or concat(...,...)
- current_date(), current_time(), and current_timestamp()
- second(...)、minute(...)、hour(...)、day(...)、month(...) 和 year(...)
- EJB-QL 3.0 定义的任何功能或操作符: substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()
- coalesce() 和 nullif()
- str() 把数字或者时间值转换为可读的字符串
- cast(... as ...), 其第二个参数是某 Hibernate 类型的名字, 以及 extract(... from ...), 只要 ANSI cast() 和 extract() 被底层数据库支持
- HQL index() 函数, 作用于 join 的有序集合的别名。
- HQL 函数, 把集合作为参数: size(), minelement(), maxelement(), minindex(), maxindex(), 还有特别的 elements() 和 indices 函数, 可以与数量词加以限定: some, all, exists, any, in。
- 任何数据库支持的 SQL 标量函数, 比如 sign(), trunc(), rtrim(), sin()
- JDBC 风格的参数传入 ?
- 命名参数 :name, :start_date, :x1
- SQL 直接常量 'foo', 69, 6.66E+2, '1970-01-01 10:00:01.0'
- Java public static final 类型的常量 eg.Color.TABBY

关键字 in 与 between 可按如下方法使用:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

而且否定的格式也可以如下书写:

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

同样，子句 is null 与 is not null 可以被用来测试空值（null）。

在 Hibernate 配置文件中声明 HQL“查询替代（query substitutions）”之后，布尔表达式（Booleans）可以在其他表达式中轻松的使用：

```
<property name="hibernate.query.substitutions"
>true 1, false 0</property
>
```

系统将该 HQL 转换为 SQL 语句时，该设置表明将用字符 1 和 0 来取代关键字 true 和 false：

```
from Cat cat where cat.alive = true
```

你可以用特殊属性 size，或是特殊函数 size() 测试一个集合的大小。

```
from Cat cat where cat.kittens.size
> 0
```

```
from Cat cat where size(cat.kittens)
> 0
```

对于索引了（有序）的集合，你可以使用 minindex 与 maxindex 函数来引用到最小与最大的索引序数。同理，你可以使用 minelement 与 maxelement 函数来引用到一个基本数据类型的集合中最小与最大的元素。例如：

```
from Calendar cal where maxelement(cal.holidays)
> current_date
```

```
from Order order where maxindex(order.items)
> 100
```

```
from Order order where minelement(order.items)
> 10000
```

在传递一个集合的索引集或者是元素集（elements 与 indices 函数）或者传递一个子查询的结果的时候，可以使用 SQL 函数 any, some, all, exists, in：

```
select mother from Cat as mother, Cat as kit
```

```
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3
> all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

注意，在 Hibernate3 中，这些结构变量 — size, elements, indices, minindex, maxindex, minelement, maxelement — 只能在 where 子句中使用。

一个被索引过的（有序的）集合的元素（arrays, lists, maps）可以在其他索引中被引用（只能在 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
```

在 [] 中的表达式甚至可以是一个算数表达式：

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

对于一个一对多的关联（one-to-many association）或是值的集合中的元素，HQL 也提供内建的 index() 函数。

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

如果底层数据库支持标量的 SQL 函数，它们也可以被使用：

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

如果你还不能对所有的这些深信不疑，想想下面的查询。如果使用 SQL，语句长度会增长多少，可读性会下降多少：

```
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： 会像如下的语句

```
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
    )
```

16.11. order by 子句

查询返回的列表（list）可以按照一个返回的类或组件（components）中的任何属性（property）进行排序：

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

可选的 asc 或 desc 关键字指明了按照升序或降序进行排序。

16.12. group by 子句

一个返回聚集值（aggregate values）的查询可以按照一个返回的类或组件（components）中的任何属性（property）进行分组:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

having 子句在这里也允许使用。

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

如果底层的数据库支持的话（例如不能在 MySQL 中使用），SQL 的一般函数与聚集函数也可以出现在 having 与 order by 子句中。

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight)
> 100
order by count(kitten) asc, sum(kitten.weight) desc
```

注意 group by 子句与 order by 子句中都不能包含算术表达式（arithmetic expressions）。也要注意 Hibernate 目前不会扩展 group 的实体，因此你不能写 group by cat，除非 cat 的所有属性都不是聚集的（non-aggregated）。你必须明确的列出所有的非聚集属性。

16.13. 子查询

对于支持子查询的数据库，Hibernate 支持在查询中使用子查询。一个子查询必须被圆括号包围起来（经常是 SQL 聚集函数的圆括号）。甚至相互关联的子查询（引用到外部查询中的别名的子查询）也是允许的。

```
from Cat as fatcat
where fatcat.weight
> (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

注意，HQL 自查询只可以在 select 或者 where 子句中出现。

Note that subqueries can also utilize row value constructor syntax. See 第 16.18 节 “Row value 构造函数语法” for more information.

16.14. HQL 示例

Hibernate 查询可以非常的强大与复杂。实际上，Hibernate 的一个主要卖点就是查询语句的威力。这里有一些例子，它们与我在最近的一个项目中使用的查询非常相似。注意你能用到的大多数查询比这些要简单的多。

下面的查询对于某个特定的客户的所有未支付的账单，在给定给最小总价值的情况下，返回订单的 id，条目的数量和总价值，返回值按照总价值的结果进行排序。为了决定价格，查询使用了当前目录。作为转换结果的 SQL 查询，使用了ORDER，ORDER_LINE，PRODUCT，CATALOG 和 PRICE 库表。

```
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
```

这简直是一个怪物！实际上，在现实生活中，我并不热衷于子查询，所以我的查询语句看起来更像这个：

```
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
```

下面一个查询计算每一种状态下的支付的数目，除去所有处于 AWAITING_APPROVAL 状态的支付，因为在该状态下 当前的用户作出了状态的最新改变。该查询被转换成含有两个内连接以及一个相关联的子选择的 SQL 查询，该查询使用了表 PAYMENT，PAYMENT_STATUS 以及 PAYMENT_STATUS_CHANGE。

```
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
```

如果我把 statusChanges 实例集映射为一个列表（list）而不是一个集合（set），书写查询语句将更加简单。

```
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
```

下面一个查询使用了 MS SQL Server 的 isNull() 函数用以返回当前用户所属组织的组织帐号及组织未支付的账。它被转换成一个对表 ACCOUNT, PAYMENT, PAYMENT_STATUS, ACCOUNT_TYPE, ORGANIZATION 以及 ORG_USER 进行的三个内连接，一个外连接和一个子选择的 SQL 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
```

对于一些数据库，我们需要弃用（相关的）子选择。

```
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
```

16.15. 批量的 UPDATE 和 DELETE

HQL now supports update, delete and insert ... select ... statements. See 第 15.4 节 “DML（数据操作语言）风格的操作（DML-style operations）” for more information.

16.16. 小技巧 & 小窍门

你可以统计查询结果的数目而不必实际的返回他们：

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue()
```

若想根据一个集合的大小来进行排序，可以使用如下的语句：

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

如果你的数据库支持子选择，你可以在你的查询的 where 子句中为选择的大小（selection size）指定一个条件：

```
from User usr where size(usr.messages)
>= 1
```

如果你的数据库不支持子选择语句，使用下面的查询：

```
select usr.id, usr.name
from User usr
    join usr.messages msg
group by usr.id, usr.name
having count(msg)
>= 1
```

因为内连接（inner join）的原因，这个解决方案不能返回含有零个信息的 User 类的实例，所以这种情况下使用下面的格式将是有帮助的：

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

JavaBean 的属性可以被绑定到一个命名查询（named query）的参数上：

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

通过将接口 Query 与一个过滤器（filter）一起使用，集合（Collections）是可以分页的：

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

通过使用查询过滤器（query filter）可以将集合（Collection）的元素分组或排序：

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

不用通过初始化，你就可以知道一个集合（Collection）的大小：

```
( (Integer) session.createQuery("select count(*) from ....").iterate().next() ).intValue();
```

16.17. 组件

在 HQL 查询里，组件可以和简单值类型一样使用。它们可以出现在 select 子句里：

```
select p.name from Person p
```

```
select p.name.first from Person p
```

在这里，Person 的 name 属性是一个组件。组件也可以用在 where 子句里：

```
from Person p where p.name = :name
```

```
from Person p where p.name.first = :firstName
```

组件也可以用在 order by 子句里：

```
from Person p order by p.name
```

```
from Person p order by p.name.first
```

组件的另外一个常见用法是在 第 16.18 节 “Row value 构造函数语法” 行值（row value）构造函数里。

16.18. Row value 构造函数语法

HQL 支持 ANSI SQL row value constructor 语法（有时也叫作 tuple 语法），即使底层数据库可能不支持这个概念。在这里我们通常指的是多值（multi-valued）的比较，典型地是和组件相关联。来看看一个定义了 name 组件的实体 Person：

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

那是有效的语法，虽然有点冗长。我们可以使它更加简洁一点，并使用 row value constructor 语法：

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

在 select 子句里指定这个也是很有用的：

```
select p.name from Person p
```

当使用需要比较多个值的子查询时，采用 row value constructor 语法也很有用处：

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

决定是否使用这个语法的一件因素就是：这个查询将依赖于元数据里的组件子属性（sub-properties）的顺序。

条件查询（Criteria Queries）

具有一个直观的、可扩展的条件查询 API 是 Hibernate 的特色。

17.1. 创建一个 Criteria 实例

org.hibernate.Criteria接口表示特定持久类的一个查询。Session 是 Criteria 实例的工厂。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

17.2. 限制结果集内容

一个单独的查询条件是 org.hibernate.criterion.Criterion 接口的一个实例。org.hibernate.criterion.Restrictions 类定义了获得某些内置 Criterion 类型的工厂方法。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

约束可以按逻辑分组。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Hibernate 提供了相当多的内置 criterion 类型（Restrictions 子类），但是尤其有用的是可以允许你直接使用 SQL。

```
List cats = sess.createCriteria(Cat.class)
          .add(   Restrictions.sqlRestriction("lower({alias}.name)   like   lower(?)",   "Fritz
%", Hibernate.STRING) )
    .list();
```

{alias} 占位符应当被替换为被查询实体的列别名。

Property 实例是获得一个条件的另外一种途径。你可以通过调用 Property.forName() 创建一个 Property:

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

17.3. 结果集排序

你可以使用 org.hibernate.criterion.Order 来为查询结果排序。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

17.4. 关联

通过使用 createCriteria() 对关联进行导航，你可以指定相关实体的约束。

```
List cats = sess.createCriteria(Cat.class)
```

```
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

注意第二个 createCriteria() 返回一个新的 Criteria 实例，该实例引用 kittens 集合中的元素。

接下来，替换形态在某些情况下也是很有用的。

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

（createAlias() 并不创建一个新的 Criteria 实例。）

Cat 实例所保存的之前两次查询所返回的 kittens 集合是 没有被条件预过滤的。如果你希望只获得符合条件的 kittens，你必须使用 ResultTransformer。

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .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");
}
```

此外，你可以用一个 left outer join 来操纵结果集：

```
               List cats = session.createCriteria( Cat.class )
                   .createAlias("mate", "mt", Criteria.LEFT_JOIN, Restrictions.like("mt.name",
 "good%") )
                      .addOrder(Order.asc("mt.age"))
                      .list();
```

这将返回配偶的名字以 "good" 起始的所有 Cat，并根据其配偶的年龄进行排序。当需要在返回复杂/大型结果集前进行排序或限制、在多个查询必须执行且结果通过 Java 在内存里组合从而删除许多实例时，这很有用。

如果没有这个功能，那么没有配偶的猫就需要在一次查询里进行加载。

第二个查询将需要获取配偶名以 "good" 起始并按照配偶年龄排序的猫。

第三点，列表需要在内存中进行手工联合。

17.5. 动态关联抓取

你可以使用 setFetchMode() 在运行时定义动态关联抓取的语义。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

This query will fetch both mate and kittens by outer join. See 第 21.1 节 “抓取策略 (Fetching strategies)” for more information.

17.6. 查询示例

org.hibernate.criterion.Example 类允许你通过一个给定实例构建一个条件查询。

```
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、标识符和关联被忽略。默认情况下值为 null 的属性将被排除。

你可以自行调整 Example 使之更实用。

```
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();
```

你甚至可以使用 examples 在关联对象上放置条件。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
```

```
    .list();
```

17.7. 投影（Projections）、聚合（aggregation）和分组（grouping）

org.hibernate.criterion.Projections 是 Projection 的实例工厂。我们通过调用 setProjection() 应用投影到一个查询。

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

在一个条件查询中没有必要显式的使用 "group by" 。某些投影类型就是被定义为分组投影，他们也出现在 SQL 的 group by 子句中。

你可以选择把一个别名指派给一个投影，这样可以使投影值被约束或排序所引用。下面是两种不同的实现方式：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

alias() 和 as() 方法简便的将一个投影实例包装到另外一个 别名的 Projection 实例中。简而言之，当你添加一个投影到一个投影列表中时你可以为它指定一个别名：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
```

```
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

你也可以使用 `Property.forName()` 来表示投影:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

17.8. 离线 (detached) 查询和子查询

`DetachedCriteria` 类使你在一个 `session` 范围之外创建一个查询，并且可以使用任意的 `Session` 来执行它。

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
```

```
txn.commit();
session.close();
```

DetachedCriteria 也可以用以表示子查询。条件实例包含子查询可以通过 Subqueries 或者 Property 获得。

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight").gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

甚至相互关联的子查询也是有可能的：

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight").gt(avgWeightForSex) )
    .list();
```

17.9. 根据自然标识查询（Queries by natural identifier）

对大多数查询，包括条件查询而言，因为查询缓存的失效（invalidation）发生得太频繁，查询缓存不是非常高效。然而，有一种特别的查询，可以通过不变的自然键优化缓存的失效算法。在某些应用中，这种类型的查询比较常见。条件查询 API 对这种用例提供了特别规约。

首先，你应该对你的 entity 使用 <natural-id> 来映射自然键，然后打开第二级缓存。

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
```

```
</class
>
```

注意,此功能对具有mutable自然键的 entity 并不适用。

现在，我们可以用 Restrictions.naturalId() 来使用更加高效的缓存算法。

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

Native SQL 查询

你也可以使用你的数据库的 Native SQL 语言来查询数据。这对你在要使用数据库的某些特性的时候（比如说在查询提示或者 Oracle 中的 CONNECT 关键字），这是非常有用的。这就能够扫清你把原来直接使用 SQL/JDBC 的程序迁移到基于 Hibernate 应用的道路上的障碍。

Hibernate3 允许你使用手写的 sql 来完成所有的 create、update、delete 和 load 操作（包括存储过程）

18.1. 使用 SQLQuery

对原生 SQL 查询执行的控制是通过 SQLQuery 接口进行的，通过执行Session.createSQLQuery()获取这个接口。下面来描述如何使用这个 API 进行查询。

18.1.1. 标量查询（Scalar queries）

最基本的 SQL 查询就是获得一个标量（数值）的列表。

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

它们都将返回一个 Object 数组（Object[]）组成的 List，数组每个元素都是 CATS 表的一个字段值。Hibernate 会使用 ResultSetMetadata 来判定返回的标量值的实际顺序和类型。

如果要避免过多的使用 ResultSetMetadata，或者只是为了更加明确的指名返回值，可以使用 addScalar()：

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

这个查询指定：

- SQL 查询字符串
- 要返回的字段和类型

它仍然会返回 Object 数组，但是此时不再使用 ResultSetMetdata，而是明确的将 ID，NAME 和 BIRTHDATE 按照 Long，String 和 Short 类型从 resultset 中取出。同时，也指明了就算 query 是使用 * 来查询的，可能获得超过列出的这三个字段，也仅仅会返回这三个字段。

对全部或者部分的标量值不设置类型信息也是可以的。

```
sess.createSQLQuery("SELECT * FROM CATS")
```

```
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

基本上这和前面一个查询相同,只是此时使用 ResultSetMetaData 来决定 NAME 和 BIRTHDATE 的类型，而 ID 的类型是明确指出的。

关于从 ResultSetMetaData 返回的 java.sql.Types 是如何映射到 Hibernate 类型，是由方言（Dialect）控制的。假若某个指定的类型没有被映射，或者不是你所预期的类型，你可以通过 Dialet 的 registerHibernateType 调用自行定义。

18.1.2. 实体查询（Entity queries）

上面的查询都是返回标量值的，也就是从 resultset 中返回的“裸”数据。下面展示如何通过 addEntity() 让原生查询返回实体对象。

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

这个查询指定:

- SQL 查询字符串
- 要返回的实体

假设 Cat 被映射为拥有 ID，NAME 和 BIRTHDATE 三个字段的类，以上的两个查询都返回一个 List，每个元素都是一个 Cat 实体。

假若实体在映射时有一个 many-to-one 的关联指向另外一个实体，在查询时必须也返回那个实体，否则会导致发生一个 "column not found" 的数据库错误。这些附加的字段可以使用 * 标注来自动返回，但我们希望还是明确指明，看下面这个具有指向 Dog 的 many-to-one 的例子:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

这样 cat.getDog() 就能正常运作。

18.1.3. 处理关联和集合类（Handling associations and collections）

通过提前抓取将 Dog 连接获得，而避免初始化 proxy 带来的额外开销也是可能的。这是通过 addJoin() 方法进行的，这个方法可以让你将关联或集合连接进来。

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d
 WHERE c.DOG_ID = d.D_ID")
```

```
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

上面这个例子中，返回的 Cat 对象，其 dog 属性被完全初始化了，不再需要数据库的额外操作。注意，我们加了一个别名（"cat"），以便指明 join 的目标属性路径。通过同样的提前连接也可以作用于集合类，例如，假若 Cat 有一个指向 Dog 的一对多关联。

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE
 c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

到此为止，我们碰到了天花板：若不对 SQL 查询进行增强，这些已经是在 Hibernate 中使用原生 SQL 查询所能做到的最大可能了。下面的问题即将出现：返回多个同样类型的实体怎么办？或者默认的别名／字段不够又怎么办？

18.1.4. 返回多个实体（Returning multiple entities）

到目前为止,结果集字段名被假定为和映射文件中指定的的字段名是一致的。假若 SQL 查询连接了多个表，同一个字段名可能在多个表中出现多次，这就会造成问题。

下面的查询中需要使用字段别名注射（这个例子本身会失败）：

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询的本意是希望每行返回两个 Cat 实例，一个是 cat,另一个是它的妈妈。但是因为它们的字段名被映射为相同的，而且在某些数据库中，返回的字段别名是“c.ID”，"c.NAME" 这样的形式，而它们和在映射文件中的名字（"ID" 和 "NAME"）不匹配，这就会造成失败。

下面的形式可以解决字段名重复：

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询指定：

- SQL 查询语句，其中包含占位附来让 Hibernate 注射字段别名
- 查询返回的实体

上面使用的 {cat.*} 和 {mother.*} 标记是作为“所有属性”的简写形式出现的。当然你也可以明确地罗列出字段名，但在这个例子里面我们让 Hibernate 来为每个属性注射

SQL 字段别名。字段别名的占位符是属性名加上表别名的前缀。在下面的例子中，我们从另外一个表（cat_log）中通过映射元数据中的指定获取 Cat 和它的妈妈。注意，要是我们愿意，我们甚至可以在 where 子句中使用属性别名。

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

18.1.4.1. 别名和属性引用（Alias and property references）

大多数情况下，都需要上面的属性注射，但在使用更加复杂的映射，比如复合属性、通过标识符构造继承树，以及集合类等等情况下，也有一些特别的别名，来允许 Hibernate 注入合适的别名。

下表列出了使用别名注射参数的不同可能性。注意：下面结果中的别名只是示例，实用时每个别名需要唯一并且不同的名字。

表 18.1. 别名注射（alias injection names）

描述	语法	示例
简单属性	{[aliasname].[propertyname]	A_NAME as {item.name}
复合属性	{[aliasname].[componentname].[propertyname]}	CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}
实体辨别器（Discriminator of an entity）	{[aliasname].class}	DISC as {item.class}
实体的所有属性	{[aliasname].*}	{item.*}
集合键（collection key）	{[aliasname].key}	ORGID as {coll.key}
集合 id	{[aliasname].id}	EMPID as {coll.id}
集合元素	{[aliasname].element}	XID as {coll.element}
集合元素的属性	{[aliasname].element.[propertyname]}	NAME as {coll.element.name}
集合元素的所有属性	{[aliasname].element.*}	{coll.element.*}
集合的所有属性	{[aliasname].*}	{coll.*}

18.1.5. 返回非受管实体（Returning non-managed entities）

可以对原生 sql 查询使用 ResultTransformer。这会返回不受 Hibernate 管理的实体。

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

这个查询指定:

- SQL 查询字符串
- 结果转换器（result transformer）

上面的查询将会返回 CatDTO 的列表,它将被实例化并且将 NAME 和 BIRTHDAY 的值注射入对应的属性或者字段。

18.1.6. 处理继承（Handling inheritance）

原生 SQL 查询假若其查询结果实体是继承树中的一部分，它必须包含基类和所有子类的所有属性。

18.1.7. 参数（Parameters）

原生查询支持位置参数和命名参数:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

18.2. 命名 SQL 查询

Named SQL queries can also be defined in the mapping document and called in exactly the same way as a named HQL query (see 第 11.4.1.7 节 “外置命名查询（Externalizing named queries）”). In this case, you do not need to call addEntity().

例 18.1. Named sql query using the <sql-query> maping element

```
<sql-query name="persons">
    <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 :namePattern
</sql-query>
```

例 18.2. Execution of a named query

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

<return-join> 和 <load-collection> 元素是用来连接关联以及将查询定义为预先初始化各个集合的。

例 18.3. Named sql query with association

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

一个命名查询可能会返回一个标量值。你必须使用 <return-scalar> 元素来指定字段的别名和 Hibernate 类型:

例 18.4. Named query returning a scalar

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

你可以把结果集映射的信息放在外部的 <resultset> 元素中，这样就可以在多个命名查询间，或者通过 setResultSetMapping() API 来访问。

例 18.5. <resultset> mapping used to externalize mapping information

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           address.STREET AS {address.street},
           address.CITY AS {address.city},
           address.STATE AS {address.state},
           address.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS address
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

另外，你可以在 java 代码中直接使用 hbm 文件中的结果集定义信息。

例 18.6. Programmatically specifying the result mapping information

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

So far we have only looked at externalizing SQL queries using Hibernate mapping files. The same concept is also available with anntations and is called named native queries. You can use @NamedNativeQuery (@NamedNativeQueries) in conjunction with @SqlResultSetMapping (@SqlResultSetMappings). Like @NamedQuery, @NamedNativeQuery and @SqlResultSetMapping can be defined at class level, but their scope is global to the application. Lets look at a view examples.

例 18.7 “Named SQL query using @NamedNativeQuery together with @SqlResultSetMapping” shows how a resultSetMapping parameter is defined in @NamedNativeQuery. It represents the name of a defined @SqlResultSetMapping. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property. In the example 2 entities, Night and Area, are returned and each property is declared and associated to a column name, actually the column name retrieved by the query.

In 例 18.8 “Implicit result set mapping” the result set mapping is implicit. We only describe the entity class of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the model property is bound to the model_txt column.

Finally, if the association to a related entity involve a composite primary key, a @FieldResult element should be used for each foreign key column. The @FieldResult name is composed of the property name for the relationship, followed by a dot ("."), followed by the name or the field or property of the primary key. This can be seen in 例 18.9 “Using dot notation in @FieldResult for specifying associations ” .

例 18.7. Named SQL query using @NamedNativeQuery together with @SqlResultSetMapping

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
    + " night.night_date, area.id aid, night.area_id, area.name "
    + "from Night night, Area area where night.area_id = area.id",
                resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)
```

例 18.8. Implicit result set mapping

```
@Entity
@SqlResultSetMapping(name="implicit",
                    entities=@EntityResult(entityClass=SpaceShip.class))
@NamedNativeQuery(name="implicitSample",
                query="select * from SpaceShip",
                resultSetMapping="implicit")
public class SpaceShip {
    private String name;
    private String model;
    private double speed;

    @Id
    public String getName() {
        return name;
    }

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

```
    }

    @Column(name="model_txt")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }
}
```

例 18.9. Using dot notation in @FieldResult for specifying associations

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=SpaceShip.class,
            fields = {
                    @FieldResult(name="name", column = "name"),
                    @FieldResult(name="model", column = "model"),
                    @FieldResult(name="speed", column = "speed"),
                    @FieldResult(name="captain.firstname", column = "firstn"),
                    @FieldResult(name="captain.lastname", column = "lastn"),
                    @FieldResult(name="dimensions.length", column = "length"),
                    @FieldResult(name="dimensions.width", column = "width")
                    }),
        columns = { @ColumnResult(name = "surface"),
                    @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
     query="select name, model, speed, lname as lastn, fname as firstn, length, width, length
 * width as surface from SpaceShip",
    resultSetMapping="compositekey")
} )
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {
        return name;
    }

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

```
    }

    @ManyToOne(fetch= FetchType.LAZY)
    @JoinColumns( {
            @JoinColumn(name="fname", referencedColumnName = "firstname"),
            @JoinColumn(name="lname", referencedColumnName = "lastname")
            } )
    public Captain getCaptain() {
        return captain;
    }

    public void setCaptain(Captain captain) {
        this.captain = captain;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }

    public void setDimensions(Dimensions dimensions) {
        this.dimensions = dimensions;
    }
}

@Entity
@IdClass(Identity.class)
public class Captain implements Serializable {
    private String firstname;
    private String lastname;

    @Id
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    @Id
    public String getLastname() {
        return lastname;
    }
```

```
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

提示

If you retrieve a single entity using the default mapping, you can specify the resultClass attribute instead of resultSetMapping:

```
@NamedNativeQuery(name="implicitSample", query="select * from
 SpaceShip", resultClass=SpaceShip.class)
public class SpaceShip {
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the @SqlResultsetMapping through @ColumnResult. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

例 18.10. Scalar values via @ColumnResult

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar", query="select length*width as dimension from
 SpaceShip", resultSetMapping="scalar")
```

An other query hint specific to native queries has been introduced: org.hibernate.callable which can be true or false depending on whether the query is a stored procedure or not.

18.2.1. 使用 return-property 来明确地指定字段／别名

使用 <return-property> 你可以明确的告诉 Hibernate 使用哪些字段别名，这取代了使用 {}-语法 来让 Hibernate 注入它自己的别名。例如:

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
```

```
</sql-query>
```

<return-property> 也可用于多个字段，它解决了使用 {}-语法不能细粒度控制多个字段的限制。

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

注意在这个例子中，我们使用了 <return-property> 结合 {} 的注入语法。允许用户来选择如何引用字段以及属性。

如果你映射一个识别器（discriminator），你必须使用 <return-discriminator> 来指定识别器字段。

18.2.2. 使用存储过程来查询

Hibernate 3 引入了对存储过程查询（stored procedure）和函数（function）的支持。以下的说明中，这二者一般都适用。存储过程／函数必须返回一个结果集，作为 Hibernate 能够使用的第一个外部参数。下面是一个 Oracle9 和更高版本的存储过程例子。

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

在 Hibernate 里要要使用这个查询，你需要通过命名查询来映射它。

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
```

```
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

注意存储过程当前仅仅返回标量和实体现在。不支持 <return-join> 和 <load-collection>。

18.2.2.1. 使用存储过程的规则和限制

为了在 Hibernate 中使用存储过程，你必须遵循一些规则。不遵循这些规则的存储过程将不可用。如果你仍然想使用他们，你必须通过 session.connection() 来执行他们。这些规则针对于不同的数据库。因为数据库提供商有各种不同的存储过程语法和语义。

对存储过程进行的查询无法使用 setFirstResult()/setMaxResults() 进行分页。

建议采用的调用方式是标准 SQL92: { ? = call functionName(<parameters>) } 或者 { ? = call procedureName(<parameters>) }。原生调用语法不被支持。

对于 Oracle 有如下规则:

- 函数必须返回一个结果集。存储过程的第一个参数必须是 OUT，它返回一个结果集。这是通过 Oracle 9 或 10 的 SYS_REFCURSOR 类型来完成的。在 Oracle 中你需要定义一个 REF CURSOR 类型，参见 Oracle 的手册。

对于 Sybase 或者 MS SQL server 有如下规则:

- 存储过程必须返回一个结果集。注意这些 servers 可能返回多个结果集以及更新的数目。Hibernate 将取出第一条结果集作为它的返回值，其他将被丢弃。
- 如果你能够在存储过程里设定 SET NOCOUNT ON，这可能会效率更高，但这不是必需的。

18.3. 定制 SQL 用来 create, update 和 delete

Hibernate3 can use custom SQL for create, update, and delete operations. The SQL can be overridden at the statement level or inidividual column level. This section describes statement overrides. For columns, see 第 5.6 节 “字段的读写表达式”. 例 18.11 “Custom CRUD via annotations” shows how to define custom SQL operatons using annotations.

例 18.11. Custom CRUD via annotations

```
@Entity
```

```
@Table(name="CHAOS")
@SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname, id) VALUES(?,upper(?),?,?)")
@SQLUpdate( sql="UPDATE CHAOS SET size = ?, name = upper(?), nickname = ? WHERE id = ?")
@SQLDelete( sql="DELETE CHAOS WHERE id = ?")
@SQLDeleteAll( sql="DELETE CHAOS")
@Loader(namedQuery = "chaos")
@NamedNativeQuery(name="chaos", query="select id, size, name, lower( nickname ) as nickname from
 CHAOS where id= ?", resultClass = Chaos.class)
public class Chaos {
    @Id
    private Long id;
    private Long size;
    private String name;
    private String nickname;
```

@SQLInsert, @SQLUpdate, @SQLDelete, @SQLDeleteAll respectively override the INSERT, UPDATE, DELETE, and DELETE all statement. The same can be achieved using Hibernate mapping files and the <sql-insert>, <sql-update> and <sql-delete> nodes. This can be seen in 例 18.12 “Custom CRUD XML” .

例 18.12. Custom CRUD XML

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

If you expect to call a store procedure, be sure to set the callable attribute to true. In annotations as well as in xml.

To check that the execution happens correctly, Hibernate allows you to define one of those three strategies:

- none: no check is performed: the store procedure is expected to fail upon issues
- count: use of rowcount to check that the update is successful
- param: like COUNT but using an output parameter rather that the standard mechanism

To define the result check style, use the check parameter which is again available in annoations as well as in xml.

You can use the exact same set of annotations respectively xml nodes to override the collection related statements -see 例 18.13 “Overriding SQL statements for collections using annotations” .

例 18.13. Overriding SQL statements for collections using annotations

```
@OneToMany
@JoinColumn(name="chaos_fk")
@SQLInsert( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = ? where id = ?")
@SQLDelete( sql="UPDATE CASIMIR_PARTICULE SET chaos_fk = null where id = ?")
private Set<CasimirParticle> particles = new HashSet<CasimirParticle>();
```

提示

The parameter order is important and is defined by the order Hibernate handles properties. You can see the expected order by enabling debug logging for the org.hibernate.persister.entity level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL through annotations or mapping files as that will override the Hibernate generated static sql)

Overriding SQL statements for secondary tables is also possible using @org.hibernate.annotations.Table and either (or all) attributes sqlInsert, sqlUpdate, sqlDelete:

例 18.14. Overriding SQL statements for secondary tables

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "`Cat nbr1`"),
    @SecondaryTable(name = "Cat2"})
@org.hibernate.annotations.Tables( {
    @Table(appliesTo = "Cat", comment = "My cat table" ),
    @Table(appliesTo = "Cat2", foreignKey = @ForeignKey(name="FK_CAT2_CAT"), fetch = FetchMode.SELECT,
        sqlInsert=@SQLInsert(sql="insert into Cat2(storyPart2, id) values(upper(?), ?)") )
} )
public class Cat implements Serializable {
```

The previous example also shows that you can give a comment to a given table (primary or secondary): This comment will be used for DDL generation.

提示

The SQL is directly executed in your database, so you can use any dialect you like. This will, however, reduce the portability of your mapping if you use database specific SQL.

Last but not least, stored procedures are in most cases required to return the number of rows inserted, updated and deleted. Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations:

例 18.15. Stored procedures and their return value

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

18.4. 定制装载 SQL

You can also declare your own SQL (or HQL) queries for entity loading. As with inserts, updates, and deletes, this can be done at the individual column level as described in 第 5.6 节 “字段的读写表达式” or at the statement level. Here is an example of a statement level override:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

这只是一个前面讨论过的命名查询声明，你可以在类映射里引用这个命名查询。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

这也可以用于存储过程

你甚至可以定一个用于集合装载的查询:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

你甚至还可以定义一个实体装载器，它通过连接抓取装载一个集合：

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

The annotation equivalent <loader> is the @Loader annotation as seen in 例 18.11 “Custom CRUD via annotations” .

过滤数据

Hibernate3 提供了一种创新的方式来处理具有“显性（visibility）”规则的数据，那就是使用Hibernate 过滤器。Hibernate 过滤器是全局有效的、具有名字、可以带参数的过滤器，对于某个特定的 Hibernate session 您可以选择是否启用（或禁用）某个过滤器。

19.1. Hibernate 过滤器（filters）

Hibernate3 新增了对某个类或者集合使用预先定义的过滤器条件（filter criteria）的功能。过滤器条件相当于定义一个 非常类似于类和各种集合上的“where”属性的约束子句，但是过滤器条件可以带参数。 应用程序可以在运行时决定是否启用给定的过滤器，以及使用什么样的参数值。过滤器的用法很像数据库视图，只不过是在应用程序中确定使用什么样的参数的。

Using annotatons filters are defined via @org.hibernate.annotations.FilterDef or @org.hibernate.annotations.FilterDefs. A filter definition has a name() and an array of parameters(). A parameter will allow you to adjust the behavior of the filter at runtime. Each parameter is defined by a @ParamDef which has a name and a type. You can also define a defaultCondition() parameter for a given @FilterDef to set the default condition to use when none are defined in each individual @Filter. @FilterDef(s) can be defined at the class or package level.

We now need to define the SQL filter clause applied to either the entity load or the collection load. @Filter is used and placed either on the entity or the collection element. The connection between @FilterName and @Filter is a matching name.

例 19.1. @FilterDef and @Filter annotations

```
@Entity
@FilterDef(name="minLength", parameters=@ParamDef( name="minLength", type="integer" ) )
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

When the collection use an association table as a relational representation, you might want to apply the filter condition to the association table itself or to the target entity table. To apply the constraint on the target entity, use the regular @Filter annotation. However, if you want to target the association table, use the @FilterJoinTable annotation.

例 19.2. Using @FilterJoinTable for filterting on the association table

```
@OneToMany
    @JoinTable
```

```
    //filter on the target entity table
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length")
    //filter on the association table
    @FilterJoinTable(name="security", condition=":userlevel >= requredLevel")
    public Set<Forest> getForests() { ... }
```

Using Hibernate mapping files for defining filters the situtation is very similar. The filters must first be defined and then attached to the appropriate mapping elements. To define a filter, use the <filter-def/> element within a <hibernate-mapping/> element:

例 19.3. Defining a filter definition via <filter-def>

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

This filter can then be attached to a class or collection (or, to both or multiples of each at the same time):

例 19.4. Attaching a filter to a class or collection using <filter>

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>

    <set ...>
        <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
    </set>
</class>
```

Session 对象中会用到的方法有：enableFilter(String filterName)，getEnabledFilter(String filterName)，和 disableFilter(String filterName)。Session 中默认是不启用过滤器的，必须通过 Session.enabledFilter() 方法显式的启用。该方法返回被启用的 Filter 的实例。以上文定义的过滤器为例：

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

注意，org.hibernate.Filter 的方法允许链式方法调用。（类似上面例子中启用 Filter 之后设定 Filter 参数这个“方法链”） Hibernate 的其他部分也大多有这个特性。

下面是一个比较完整的例子，使用了记录生效日期模式过滤有时效的数据：

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>
```

```
<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Department"/>
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

定义好后，如果想要保证取回的都是目前处于生效期的记录，只需在获取雇员数据的操作之前先开启过滤器即可：

```
Session session = ...;
session.enableFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

在上面的 HQL 中，虽然我们仅仅显式的使用了一个薪水条件，但因为启用了过滤器，查询将仅返回那些目前雇用关系处于生效期的，并且薪水高于一百万美元的雇员的数据。

注意：如果你打算在使用外连接（或者通过 HQL 或 load fetching）的同时使用过滤器，要注意条件表达式的方向（左还是右）。最安全的方式是使用左外连接（left outer joining）。并且通常来说，先写参数，然后是操作符，最后写数据库字段名。

在 Filter 定义之后,它可能被附加到多个实体和/或集合类,每个都有自己的条件。假若这些条件都是一样的，每次都要定义就显得很繁琐。因此，<filter-def/> 被用来定义一个默认条件，它可能作为属性或者 CDATA 出现：

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

当这个 filter 被附加到任何目的地，而又没有指明条件时，这个缺省条件就会被使用。注意，换句话说，你可以通过给 filter 附加特别的条件来重载默认条件。

XML 映射

XML Mapping is an experimental feature in Hibernate 3.0 and is currently under active development.

20.1. 用 XML 数据进行工作

Hibernate 使得你可以用 XML 数据来进行工作，恰如你用持久化的 POJO 进行工作那样。解析过的 XML 树 可以被认为是代替 POJO 的另外一种在对象层面上表示关系型数据的途径。

Hibernate 支持采用 dom4j 作为操作 XML 树的 API。你可以写一些查询从数据库中检索出 dom4j 树，随后你对这颗树做的任何修改都将自动同步回数据库。你甚至可以用 dom4j 解析 一篇 XML 文档，然后使用 Hibernate 的任一基本操作将它写入数据库：persist()，saveOrUpdate()，merge()，delete()，replicate() （合并操作merge()目前还不支持）。

这一特性可以应用在很多场合，包括数据导入导出，通过 JMS 或 SOAP 具体化实体数据以及 基于 XSLT 的报表。

一个单一的映射就可以将类的属性和 XML 文档的节点同时映射到数据库。如果不需要映射类，它也可以用来只映射 XML 文档。

20.1.1. 指定同时映射 XML 和类

这是一个同时映射 POJO 和 XML 的例子：

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class
>
```

20.1.2. 只定义 XML 映射

这是一个不映射 POJO 的例子：

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class
>
```

这个映射使得你既可以把数据作为一棵 dom4j 树那样访问，又可以作为由属性键值对（java Map）组成的图那样访问。属性名字纯粹是逻辑上的结构，你可以在 HQL 查询中引用它。

20.2. XML 映射元数据

许多 Hibernate 映射元素具有 node 属性。这使你可以指定用来保存 属性或实体数据的 XML 属性或元素。node 属性必须是下列格式之一：

- "element-name"：映射为指定的 XML 元素
- "@attribute-name"：映射为指定的 XML 属性
- "."：映射为父元素
- "element-name/@attribute-name"：映射为指定元素的指定属性

对于集合和单值的关联，有一个额外的 embed-xml 属性可用。这个属性的缺省值是真（embed-xml="true"）。如果 embed-xml="true"，则对应于被关联实体或值类型的集合的XML树将直接嵌入拥有这些关联的实体的 XML 树中。否则，如果 embed-xml="false"，那么对于单值的关联，仅被引用的实体的标识符出现在 XML 树中（被引用实体本身不出现），而集合则根本不出现。

你应该小心，不要让太多关联的 embed-xml 属性为真（embed-xml="true"），因为 XML 不能很好地处理循环引用。

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class
>
```

在这个例子中，我们决定嵌入帐目号码（account id）的集合，但不嵌入实际的帐目数据。下面的 HQL 查询：

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

返回的数据集将是这样：

```
<customer id="123456789">
    <account short-desc="Savings"
>987632567</account>
    <account short-desc="Credit Card"
>985612323</account>
    <name>
        <first-name
>Gavin</first-name>
        <initial
```

```
>A</initial>
        <last-name
>King</last-name>
    </name>
    ...
</customer
>
```

如果你把一对多映射 <one-to-many> 的 embed-xml 属性置为真（embed-xml="true"），则数据看上去就像这样：

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance
>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance
>-2370.34</balance>
    </account>
    <name>
        <first-name
>Gavin</first-name>
        <initial
>A</initial>
        <last-name
>King</last-name>
    </name>
    ...
</customer
>
```

20.3. 操作 XML 数据

你也可以重新读入和更新应用程序中的 XML 文档。通过获取一个 dom4j 会话可以做到这一点：

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}
```

```
tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

将这一特色与 Hibernate 的 replicate() 操作结合起来对于实现的基于 XML 的数据导入/导出将非常有用。

提升性能

21.1. 抓取策略（Fetching strategies）

当应用程序需要在（Hibernate实体对象图的）关联关系间进行导航的时候，Hibernate 使用 抓取策略（fetching strategy） 获取关联对象。抓取策略可以在 O/R 映射的元数据中声明，也可以在特定的 HQL 或条件查询（Criteria Query）中重载声明。

Hibernate3 定义了如下几种抓取策略：

- 连接抓取（Join fetching）：Hibernate 通过在 SELECT 语句使用 OUTER JOIN（外连接）来获得对象的关联实例或者关联集合。
- 查询抓取（Select fetching）：另外发送一条 SELECT 语句抓取当前对象的关联实体或集合。除非你显式的指定 lazy="false" 禁止 延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条 select 语句。
- 子查询抓取（Subselect fetching）：另外发送一条 SELECT 语句抓取在前面查询到（或者抓取到）的所有实体对象的关联集合。除非你显式的指定 lazy="false" 禁止延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条 select 语句。
- 批量抓取（Batch fetching）：对查询抓取的优化方案，通过指定一个主键或外键列表，Hibernate 使用单条 SELECT 语句获取一批对象实例或集合。

Hibernate 会区分下列各种情况：

- Immediate fetching，立即抓取：当宿主被加载时，关联、集合或属性被立即抓取。
- Lazy collection fetching，延迟集合抓取：直到应用程序对集合进行了一次操作时，集合才被抓取（对集合而言这是默认行为）。
- "Extra-lazy" collection fetching,"Extra-lazy" 集合抓取：对集合类中的每个元素而言，都是直到需要时才去访问数据库。除非绝对必要，Hibernate 不会试图去把整个集合都抓取到内存里来（适用于非常大的集合）。
- Proxy fetching，代理抓取：对返回单值的关联而言，当其某个方法被调用，而非对其关键字进行 get 操作时才抓取。
- "No-proxy" fetching，非代理抓取：对返回单值的关联而言，当实例变量被访问的时候进行抓取。与上面的代理抓取相比，这种方法没有那么“延迟”得厉害（就算只访问标识符，也会导致关联抓取）但是更加透明，因为对应用程序来说，不再看到 proxy。这种方法需要在编译期间进行字节码增强操作，因此很少需要用到。
- Lazy attribute fetching，属性延迟加载：对属性或返回单值的关联而言，当其实例变量被访问的时候进行抓取。需要编译期字节码强化，因此这一方法很少是必要的。

这里有两个正交的概念：关联何时被抓取，以及被如何抓取（会采用什么样的 SQL 语句）。注意不要混淆它们。我们使用抓取来改善性能。我们使用延迟来定义一些契约，对某特定类的某个脱管的实例，知道有哪些数据是可以使用的。

21.1.1. 操作延迟加载的关联

默认情况下，Hibernate 3 对集合使用延迟 select 抓取，对返回单值的关联使用延迟代理抓取。对几乎是所有的应用而言，其绝大多数的关联，这种策略都是有效的。

假若你设置了 hibernate.default_batch_fetch_size，Hibernate 会对延迟加载采取批量抓取优化措施（这种优化也可能会在更细化的级别打开）。

然而，你必须了解延迟抓取带来的一个问题。在一个打开的 Hibernate session 上下文之外调用延迟集合会导致一次意外。比如：

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

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

在 Session 关闭后，permessions 集合将是未实例化的、不再可用，因此无法正常载入其状态。Hibernate 对脱管对象不支持延迟实例化。这里的修改方法是将 permissions 读取数据的代码移到事务提交之前。

除此之外，通过对关联映射指定 lazy="false"，我们也可以使用非延迟的集合或关联。但是，对绝大部分集合来说，更推荐使用延迟方式抓取数据。如果在你的对象模型中定义了太多的非延迟关联，Hibernate 最终几乎需要在每个事务中载入整个数据库到内存中。

但是，另一方面，在一些特殊的事务中，我们也经常需要使用到连接抓取（它本身上就是非延迟的），以代替查询抓取。 下面我们将会很快明白如何具体的定制 Hibernate 中的抓取策略。在 Hibernate3 中，具体选择哪种抓取策略的机制是和选择 单值关联或集合关联相一致的。

21.1.2. 调整抓取策略（Tuning fetch strategies）

查询抓取（默认的）在 N+1 查询的情况下是极其脆弱的，因此我们可能会要求在映射文档中定义使用连接抓取：

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

在映射文档中定义的抓取策略将会对以下列表条目产生影响:

- 通过 get() 或 load() 方法取得数据。
- 只有在关联之间进行导航时，才会隐式的取得数据。
- 条件查询
- 使用了 subselect 抓取的 HQL 查询

不管你使用哪种抓取策略，定义为非延迟的类图会被保证一定装载入内存。注意这可能意味着在一条 HQL 查询后紧跟着一系列的查询。

通常情况下，我们并不使用映射文档进行抓取策略的定制。更多的是，保持其默认值，然后在特定的事务中， 使用 HQL 的左连接抓取（left join fetch） 对其进行重载。这将通知 Hibernate在第一次查询中使用外部关联（outer join），直接得到其关联数据。在条件查询 API 中，应该调用 setFetchMode（FetchMode.JOIN）语句。

也许你喜欢仅仅通过条件查询，就可以改变 get() 或 load() 语句中的数据抓取策略。例如:

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

这就是其他 ORM 解决方案的“抓取计划（fetch plan）”在 Hibernate 中的等价物。

截然不同的一种避免 N+1 次查询的方法是，使用二级缓存。

21.1.3. 单端关联代理（Single-ended association proxies）

在 Hinerbate 中，对集合的延迟抓取的采用了自己的实现方法。但是，对于单端关联的延迟抓取，则需要采用 其他不同的机制。单端关联的目标实体必须使用代理，Hihernate 在运行期二进制级（通过优异的 CGLIB 库）， 为持久对象实现了延迟载入代理。

默认的，Hibernate3 将会为所有的持久对象产生代理（在启动阶段），然后使用他们实现 多对一（many-to-one）关联和一对一（one-to-one） 关联的延迟抓取。

在映射文件中，可以通过设置 proxy 属性为目标 class 声明一个接口供代理接口使用。 默认的，Hibernate 将会使用该类的一个子类。注意：被代理的类必须实现一个至少包可见的默认构造函数，我们建议所有的持久类都应拥有这样的构造函数。

在如此方式定义一个多态类的时候，有许多值得注意的常见性的问题，例如:

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
```

```
        .....
    </subclass>
</class>
```

首先，Cat 实例永远不可以被强制转换为 DomesticCat，即使它本身就是 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);  // acquire new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

虽然如此，但实际情况并没有看上去那么糟糕。虽然我们现在有两个不同的引用，分别指向这两个不同的代理对象，但实际上，其底层应该是同一个实例对象：

```
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 类或具有 final 方法的类使用 CGLIB 代理。

最后，如果你的持久化对象在实例化时需要某些资源（例如，在实例化方法、默认构造方法中），那么代理对象也同样需要使用这些资源。实际上，代理类是持久化类的子类。

这些问题都源于 Java 的单根继承模型的天生限制。如果你希望避免这些问题，那么你的每个持久化类必须实现一个接口， 在此接口中已经声明了其业务方法。然后，你需要在映射文档中再指定这些接口，如 CatImpl 实现 Cat 而 DomesticCatImpl 实现 DomesticCat 接口。例如：

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

然后，load() 和 iterate() 永远也不会返回 Cat 和 DomesticCat 实例的代理。

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
```

```
Iterator iter = session.createQuery("from CatImpl as cat where cat.name='fritz'").iterate();
Cat fritz = (Cat) iter.next();
```

注意

list() 通常不返回代理。

这里，对象之间的关系也将被延迟载入。这就意味着，你应该将属性声明为 Cat，而不是 CatImpl。

有些方法中是不需要代理初始化的：

- equals() 方法，如果持久类没有重载 equals() 方法。
- hashCode()：如果持久类没有重载 hashCode() 方法。
- 标志符的 getter 方法。

Hibernate 将会识别出那些重载了 equals()、或 hashCode() 方法的持久化类。

若选择 lazy="no-proxy" 而非默认的 lazy="proxy"，我们可以避免类型转换带来的问题。然而，这样我们就需要编译期字节码增强，并且所有的操作都会导致立刻进行代理初始化。

21.1.4. 实例化集合和代理（Initializing collections and proxies）

在 Session 范围之外访问未初始化的集合或代理，Hibernate 将会抛出 LazyInitializationException 异常。也就是说，在分离状态下，访问一个实体所拥有的集合，或者访问其指向代理的属性时，会引发此异常。

有时候我们需要保证某个代理或者集合在 Session 关闭前就已经被初始化了。当然，我们可以通过强行调用 cat.getSex() 或者 cat.getKittens().size() 之类的方法来确保这一点。 但是这样的程序会造成读者的疑惑，也不符合通常的代码规范。

静态方法 Hibernate.initialized() 为你的应用程序提供了一个便捷的途径来延迟加载集合或代理。 只要它的 Session 处于 open 状态，Hibernate.initialize(cat) 将会为 cat 强制对代理实例化。同样，Hibernate.initialize(cat.getKittens()) 对 kittens 的集合具有同样的功能。

还有另外一种选择，就是保持 Session 一直处于 open 状态，直到所有需要的集合或代理都被载入。 在某些应用架构中，特别是对于那些使用 Hibernate 进行数据访问的代码，以及那些在不同应用层和不同物理进程中使用 Hibernate 的代码。 在集合实例化时，如何保证 Session 处于 open 状态经常会是一个问题。有两种方法可以解决此问题：

- 在一个基于 Web 的应用中，可以利用 servlet 过滤器（filter），在用户请求（request）结束、页面生成 结束时关闭 Session（这里使用了在展示层保持打开 Session 模式（Open Session in View））， 当然，这将依赖于应用框架中异常需要被正确的处理。在返回界面给用

户之前，乃至在生成界面过程中发生异常的情况下，正确关闭 Session 和结束事务将是非常重要的， 请参见 Hibernate wiki 上的 "Open Session in View" 模式，你可以找到示例。

- 在一个拥有单独业务层的应用中，业务层必须在返回之前，为 web 层“准备”好其所需的数据集合。这就意味着 业务层应该载入所有表现层/web 层所需的数据，并将这些已实例化完毕的数据返回。通常，应用程序应该为 web 层所需的每个集合调用 Hibernate.initialize()（这个调用必须发生咱 session 关闭之前）；或者使用带有 FETCH 从句，或 FetchMode.JOIN 的 Hibernate 查询，事先取得所有的数据集合。如果你在应用中使用了 Command 模式，代替 Session Facade，那么这项任务将会变得简单的多。

- 你也可以通过 merge() 或 lock() 方法，在访问未实例化的集合（或代理）之前，为先前载入的对象绑定一个新的 Session。显然，Hibernate 将不会，也不应该自动完成这些任务，因为这将引入一个特殊的事务语义。

有时候，你并不需要完全实例化整个大的集合，仅需要了解它的部分信息（例如其大小）、或者集合的部分内容。

你可以使用集合过滤器得到其集合的大小，而不必实例化整个集合：

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

这里的 createFilter() 方法也可以被用来有效的抓取集合的部分内容，而无需实例化整个集合：

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

21.1.5. 使用批量抓取（Using batch fetching）

Hibernate 可以充分有效的使用批量抓取，也就是说，如果仅一个访问代理（或集合），那么 Hibernate 将不载入其他未实例化的代理。批量抓取是延迟查询抓取的优化方案，你可以在两种批量抓取方案之间进行选择：在类级别和集合级别。

类/实体级别的批量抓取很容易理解。假设你在运行时将需要面对下面的问题：你在一个 Session 中载入了 25 个 Cat 实例，每个 Cat 实例都拥有一个引用成员 owner，其指向 Person，而 Person 类是代理，同时 lazy="true"。如果你必须遍历整个 cats 集合，对每个元素调用 getOwner() 方法，Hibernate 将会默认的执行 25 次 SELECT 查询， 得到其 owner 的代理对象。这时，你可以通过在映射文件的 Person 属性，显式声明 batch-size，改变其行为：

```
<class name="Person" batch-size="10">...</class>
```

随之，Hibernate 将只需要执行三次查询，分别为 10、10、 5。

你也可以在集合级别定义批量抓取。例如，如果每个 Person 都拥有一个延迟载入的 Cats 集合，现在，Sesssion 中载入了 10 个 person 对象，遍历 person 集合将会引起 10 次 SELECT 查询，

每次查询都会调用 getCats() 方法。如果你在 Person 的映射定义部分，允许对 cats 批量抓取，那么，Hibernate 将可以预先抓取整个集合。请看例子：

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

如果整个的 batch-size 是 3，那么 Hibernate 将会分四次执行 SELECT 查询， 按照 3、3、3、1 的大小分别载入数据。这里的每次载入的数据量还具体依赖于当前 Session 中未实例化集合的个数。

如果你的模型中有嵌套的树状结构，例如典型的帐单－原料结构（bill-of-materials pattern），集合的批量抓取是非常有用的。（尽管在更多情况下对树进行读取时，嵌套集合（nested set）或原料路径（materialized path）可能是更好的解决方法。）

21.1.6. 使用子查询抓取（Using subselect fetching）

假若一个延迟集合或单值代理需要抓取，Hibernate 会使用一个 subselect 重新运行原来的查询，一次性读入所有的实例。这和批量抓取的实现方法是一样的，不会有破碎的加载。

21.1.7. Fetch profile（抓取策略）

Another way to affect the fetching strategy for loading associated objects is through something called a fetch profile, which is a named configuration associated with the org.hibernate.SessionFactory but enabled, by name, on the org.hibernate.Session. Once enabled on a org.hibernate.Session, the fetch profile will be in affect for that org.hibernate.Session until it is explicitly disabled.

So what does that mean? Well lets explain that by way of an example which show the different available approaches to configure a fetch profile:

例 21.1. Specifying a fetch profile using @FetchProfile

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {

  @FetchProfile.FetchOverride(entity = Customer.class, association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
   @Id
   @GeneratedValue
   private long id;

   private String name;

   private long customerNumber;
```

```
    @OneToMany
    private Set<Order> orders;

    // standard getter/setter
    ...
}
```

例 21.2. Specifying a fetch profile using <fetch-profile> outside <class> node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>
    <class name="Order">
        ...
    </class>
    <fetch-profile name="customer-with-orders">
        <fetch entity="Customer" association="orders" style="join"/>
    </fetch-profile>
</hibernate-mapping>
```

例 21.3. Specifying a fetch profile using <fetch-profile> inside <class> node

```
<hibernate-mapping>
    <class name="Customer">
        ...
        <set name="orders" inverse="true">
            <key column="cust_id"/>
            <one-to-many class="Order"/>
        </set>
        <fetch-profile name="customer-with-orders">
            <fetch association="orders" style="join"/>
        </fetch-profile>
    </class>
    <class name="Order">
        ...
    </class>
</hibernate-mapping>
```

Now normally when you get a reference to a particular customer, that customer's set of orders will be lazy meaning we will not yet have loaded those orders from the database. Normally this is a good thing. Now lets say that you have a certain use case where it is more efficient to load the customer and their orders together. One way certainly is to use "dynamic fetching" strategies via an HQL or criteria queries. But another option is to use a fetch profile to achieve that. The following code will load both the customer andtheir orders:

例 21.4. Activating a fetch profile for a given Session

```
Session session = ...;
session.enableFetchProfile( "customer-with-orders" );  // name matches from mapping
Customer customer = (Customer) session.get( Customer.class, customerId );
```

注意

@FetchProfile definitions are global and it does not matter on which class you place them. You can place the @FetchProfile annotation either onto a class or package (package-info.java). In order to define multiple fetch profiles for the same class or package @FetchProfiles can be used.

目前只有 join 风格的抓取策略被支持，但其他风格也将被支持。更多细节请参考 HHH-3414 [http://opensource.atlassian.com/projects/hibernate/browse/HHH-3414]。

21.1.8. 使用延迟属性抓取（Using lazy property fetching）

Hibernate3 对单独的属性支持延迟抓取，这项优化技术也被称为组抓取（fetch groups）。请注意，该技术更多的属于市场特性。在实际应用中，优化行读取比优化列读取更重要。但是，仅载入类的部分属性在某些特定情况下会有用，例如在原有表中拥有几百列数据、数据模型无法改动的情况下。

可以在映射文件中对特定的属性设置 lazy，定义该属性为延迟载入。

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

属性的延迟载入要求在其代码构建时加入二进制指示指令（bytecode instrumentation），如果你的持久类代码中未含有这些指令， Hibernate 将会忽略这些属性的延迟设置，仍然将其直接载入。

你可以在 Ant 的 Task 中，进行如下定义，对持久类代码加入“二进制指令。”

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
```

```
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

还有一种可以优化的方法，它使用 HQL 或条件查询的投影（projection）特性，可以避免读取非必要的列， 这一点至少对只读事务是非常有用的。它无需在代码构建时“二进制指令”处理，因此是一个更加值得选择的解决方法。

有时你需要在 HQL 中通过抓取所有属性，强行抓取所有内容。

21.2. 二级缓存（The Second Level Cache）

Hibernate 的 Session 在事务级别进行持久化数据的缓存操作。 当然，也有可能分别为每个类（或集合），配置集群、或 JVM 级别（SessionFactory 级别）的缓存。你甚至可以为之插入一个集群的缓存。注意，缓存永远不知道其他应用程序对持久化仓库（数据库）可能进行的修改 （即使可以将缓存数据设定为定期失效）。

You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements org.hibernate.cache.CacheProvider using the property hibernate.cache.provider_class. Hibernate is bundled with a number of built-in integrations with the open-source cache providers that are listed in 表 21.1 “缓存策略提供商（Cache Providers）”. You can also implement your own and plug it in as outlined above. Note that versions prior to Hibernate 3.2 use EhCache as the default cache provider.

表 21.1. 缓存策略提供商（Cache Providers）

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable （not intended for production use）	org.hibernate.cache.HashtableCacheProvider	memory		yes
EHCache	org.hibernate.cache.EhCacheProvider	memory, disk		yes
OSCache	org.hibernate.cache.OSCacheProvider	memory, disk		yes
SwarmCache	org.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
JBoss Cache 1.x	org.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)
JBoss Cache 2	org.hibernate.cache.jbc.JBossCacheRegionFactory	clustered (ip multicast), transactional	yes (replication or invalidation)	yes (clock sync req.)

21.2.1. 缓存映射（Cache mappings）

As we have done in previous chapters we are looking at the two different possibiltites to configure caching. First configuration via annotations and then via Hibernate mapping files.

By default, entities are not part of the second level cache and we recommend you to stick to this setting. However, you can override this by setting the shared-cache-mode element in your persistence.xml file or by using the javax.persistence.sharedCache.mode property in your configuration. The following values are possible:

- ENABLE_SELECTIVE (Default and recommended value): entities are not cached unless explicitly marked as cacheable.
- DISABLE_SELECTIVE: entities are cached unless explicitly marked as not cacheable.
- ALL: all entities are always cached even if marked as non cacheable.
- NONE: no entity are cached even if marked as cacheable. This option can make sense to disable second-level cache altogether.

The cache concurrency strategy used by default can be set globaly via the hibernate.cache.default_cache_concurrency_strategy configuration property. The values for this property are:

- read-only
- read-write
- nonstrict-read-write
- transactional

注意

It is recommended to define the cache concurrency strategy per entity rather than using a global one. Use the @org.hibernate.annotations.Cache annotation for that.

例 21.5. Definition of cache concurrency strategy via @Cache

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

Hibernate also let's you cache the content of a collection or the identifiers if the collection contains other entities. Use the @Cache annotation on the collection property.

例 21.6. Caching collections using annotations

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
    return tickets;
}
```

例 21.7 “@Cache annotation with attributes” shows the @org.hibernate.annotations.Cache annotations with its attributes. It allows you to define the caching strategy and region of a given second level cache.

例 21.7. @Cache annotation with attributes

```
@Cache(
    CacheConcurrencyStrategy usage();                    (1)
    String region() default "";                          (2)
    String include() default "all";                      (3)
)
```

(1) usage: the given cache concurrency strategy (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)

(2) region (optional): the cache region (default to the fqcn of the class or the fq role name of the collection)

❸ include (optional): all to include all properties, non-lazy to only include non lazy properties (default all).

Let's now take a look at Hibernate mapping files. There the <cache> element of a class or collection mapping is used to configure the second level cache. Looking at 例 21.8 “The Hibernate <cache> mapping element” the parallels to anotations is obvious.

例 21.8. The Hibernate <cache> mapping element

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"    ❶
    region="RegionName"                                                ❷
    include="all|non-lazy"                                             ❸
/>
```

❶ usage（必须）说明了缓存的策略：transactional、 read-write、nonstrict-read-write 或 read-only。

❷ region （可选，默认为类或者集合的名字（class or collection role name））指定第二级缓存的区域名（name of the second level cache region）

❸ include（可选,默认为 all） non-lazy 当属性级延迟抓取打开时，标记为 lazy="true" 的实体的属性可能无法被缓存

Alternatively to <cache>, you can use <class-cache> and <collection-cache> elements in hibernate.cfg.xml.

Let's now have a closer look at the different usage strategies

21.2.2. 策略：只读缓存（Strategy：read only）

如果你的应用程序只需读取一个持久化类的实例，而无需对其修改， 那么就可以对其进行只读 缓存。这是最简单，也是实用性最好的方法。甚至在集群中，它也能完美地运作。

21.2.3. 策略：读写/缓存（Strategy：read/write）

如果应用程序需要更新数据，那么使用读/写缓存 比较合适。 如果应用程序要求“序列化事务”的隔离级别（serializable transaction isolation level），那么就决不能使用这种缓存策略。 如果在 JTA 环境中使用缓存，你必须指定 hibernate.transaction.manager_lookup_class 属性的值， 通过它，Hibernate 才能知道该应用程序中 JTA 的TransactionManager的具体策略。 在其它环境中，你必须保证在 Session.close()、或 Session.disconnect() 调用前，整个事务已经结束。 如果你想在集群环境中使用此策略，你必须保证底层的缓存实现支持锁定（locking）。Hibernate 内置的缓存策略并不支持锁定功能。

21.2.4. 策略：非严格读/写缓存（Strategy：nonstrict read/write）

如果应用程序只偶尔需要更新数据（也就是说，两个事务同时更新同一记录的情况很不常见），也不需要十分严格的事务隔离，那么比较适合使用非严格读/写缓存策略。如果在 JTA 环境中使用该策略，你必须为其指定 hibernate.transaction.manager_lookup_class 属性的值，在其它环境中，你必须保证在Session.close()、或 Session.disconnect() 调用前，整个事务已经结束。

21.2.5. 策略：事务缓存（transactional）

Hibernate 的事务缓存策略提供了全事务的缓存支持，例如对 JBoss TreeCache 的支持。这样的缓存只能用于 JTA 环境中，你必须指定为其 hibernate.transaction.manager_lookup_class 属性。

21.2.6. 各种缓存提供商/缓存并发策略的兼容性

重要

没有一种缓存提供商能够支持上列的所有缓存并发策略。下表中列出了各种提供器、及其各自适用的并发策略。

没有一种缓存提供商能够支持上列的所有缓存并发策略。下表中列出了各种提供器、及其各自适用的并发策略。

表 21.2. 各种缓存提供商对缓存并发策略的支持情况（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 Cache 1.x	yes			yes
JBoss Cache 2	yes			yes

21.3. 管理缓存（Managing the caches）

无论何时，当你给 save()、update() 或 saveOrUpdate() 方法传递一个对象时，或使用 load()、get()、list()、iterate() 或 scroll() 方法获得一个对象时，该对象都将被加入到 Session 的内部缓存中。

当随后 flush() 方法被调用时，对象的状态会和数据库取得同步。如果你不希望此同步操作发生，或者你正处理大量对象、需要对有效管理内存时，你可以调用 evict() 方法，从一级缓存中去掉这些对象及其集合。

例 21.9. Explcitly evicting a cached instance from the first level cache using Session.evict()

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

Session 还提供了一个 contains() 方法，用来判断某个实例是否处于当前 session 的缓存中。

如若要把所有的对象从 session 缓存中彻底清除，则需要调用 Session.clear()。

对于二级缓存来说，在 SessionFactory 中定义了许多方法，清除缓存中实例、整个类、集合实例或者整个集合。

例 21.10. Second-level cache eviction via SessionFactoty.evict() and SessionFacyory.evictCollection()

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

CacheMode 参数用于控制具体的 Session 如何与二级缓存进行交互。

- CacheMode.NORMAL：从二级缓存中读、写数据。
- CacheMode.GET：从二级缓存中读取数据，仅在数据更新时对二级缓存写数据。
- CacheMode.PUT：仅向二级缓存写数据，但不从二级缓存中读数据。
- CacheMode.REFRESH：仅向二级缓存写数据，但不从二级缓存中读数据。通过 hibernate.cache.use_minimal_puts 的设置，强制二级缓存从数据库中读取数据，刷新缓存内容。

如若需要查看二级缓存或查询缓存区域的内容，你可以使用统计（Statistics） API。

例 21.11. Browsing the second-level cache entries via the Statistics API

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
```

```
        .getEntries();
```

此时，你必须手工打开统计选项。可选的，你可以让 Hibernate 更人工可读的方式维护缓存内容。

例 21.12. Enabling Hibernate statistics

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

21.4. 查询缓存（The Query Cache）

查询的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。

21.4.1. 启用查询缓存

按照应用程序的事务性处理过程，查询结果的缓存将产生一些负荷。例如，如果缓存针对 Person 的查询结果，在 Person 发生了修改时，Hibernate 将需要跟踪这些结果什么时候失效。因为大多数应用程序不会从缓存查询结果中受益，所以 Hibernate 在缺省情况下将禁用缓存。要使用查询缓存，你首先需要启用查询缓存：

```
hibernate.cache.use_query_cache true
```

这个设置创建了两个新的缓存 region：

- org.hibernate.cache.StandardQueryCache，保存缓存的查询结果
- org.hibernate.cache.UpdateTimestampsCache，保存对可查询表的最近更新的时间戳。它们用于检验查询结果。

重要

If you configure your underlying cache implementation to use expiry or timeouts is very important that the cache timeout of the underlying cache region for the UpdateTimestampsCache be set to a higher value than the timeouts of any of the query caches. In fact, we recommend that the the UpdateTimestampsCache region not be configured for expiry at all. Note, in particular, that an LRU cache expiry policy is never appropriate.

如上面所提及的，绝大多数的查询并不能从查询缓存中受益，所以 Hibernate 默认是不进行查询缓存的。如若需要进行缓存，请调用 org.hibernate.Query.setCacheable(true) 方法。这个调用会让查询在执行过程中时先从缓存中查找结果，并将自己的结果集放到缓存中去。

注意

查询缓存不会缓存缓存中实际实体的状态；它只缓存标识符值和值类型的结果。出于这个原因，对于那些作为查询结果缓存的一部分（和集合缓存一样）进行缓存的实体，查询缓存应该和二级缓存一起使用。

21.4.2. 查询缓存区

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()` 方法，为每个查询指定其命名的缓存区域。

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
        .setEntity("blogger", blogger)
        .setMaxResults(15)
        .setCacheable(true)
        .setCacheRegion("frontpages")
        .list();
```

如果查询需要强行刷新其查询缓存区域，那么你应该调用 `org.hibernate.Query.setCacheMode（CacheMode.REFRESH）` 方法。 这对在其他进程中修改底层数据（例如，不通过Hibernate修改数据），或对那些需要选择性更新特定查询结果集的情况特别有用。这是对 `org.hibernate.SessionFactory.evictQueries()` 的更为有效的替代方案，同样可以清除查询缓存区域。

21.5. 理解集合性能（Understanding Collection performance）

在前面的章节里我们已经讨论了集合和相关应用程序。在本节我么将探索运行时集合的更多问题。

21.5.1. 分类（Taxonomy）

Hibernate 定义了三种基本类型的集合：

- 值数据集合
- 一对多关联（One-to-many Associations）
- 多对多关联

这个分类是区分了不同的表和外键关系类型，但是它没有告诉我们关系模型的所有内容。 要完全理解他们的关系结构和性能特点，我们必须同时考虑“用于 Hibernate 更新或删除集合行数据的主键的结构”。因此得到了如下的分类：

- 有序集合类

・集合（sets）

・包（bags）

所有的有序集合类（maps，lists，arrays）都拥有一个由 <key> 和 <index> 组成的主键。这种情况下集合类的更新是非常高效的 — 主键已经被有效的索引，因此当 Hibernate 试图更新或删除一行时，可以迅速找到该行数据。

集合（sets）的主键由 <key> 和其他元素字段构成。对于有些元素类型来说，这很低效，特别是组合元素或者大文本、大二进制字段；数据库可能无法有效的对复杂的主键进行索引。另一方面，对于一对多、多对多关联，特别是合成的标识符来说，集合也可以达到同样的高效性能。（附注：如果你希望 SchemaExport 为你的 <set> 创建主键，你必须把所有的字段都声明为 not-null="true"。）

<idbag> 映射定义了代理键，因此它总是可以很高效的被更新。事实上，<idbag> 拥有着最好的性能表现。

Bag 是最差的。因为 bag 允许重复的元素值，也没有索引字段，因此不可能定义主键。Hibernate 无法判断出重复的行。当这种集合被更改时，Hibernate 将会先完整地移除（通过一个（in a single DELETE））整个集合，然后再重新创建整个集合。因此 Bag 是非常低效的。

请注意：对于一对多关联来说，“主键”很可能并不是数据库表的物理主键。但就算在此情况下，上面的分类仍然是有用的。（它仍然反映了 Hibernate 在集合的各数据行中是如何进行“定位”的。）

21.5.2. Lists, maps 和 sets 用于更新效率最高

根据我们上面的讨论，显然有序集合类型和大多数 set 都可以在增加、删除、修改元素中拥有最好的性能。

可论证的是对于多对多关联、值数据集合而言，有序集合类比集合（set）有一个好处。因为 Set 的内在结构， 如果“改变”了一个元素，Hibernate 并不会更新（UPDATE）这一行。对于 Set 来说，只有在插入（INSERT）和删除（DELETE） 操作时“改变”才有效。再次强调：这段讨论对“一对多关联”并不适用。

注意到数组无法延迟载入，我们可以得出结论，list，map 和 idbags 是最高效的（非反向）集合类型，set 则紧随其后。 在 Hibernate 中，set 应该时最通用的集合类型，这时因为“set”的语义在关系模型中是最自然的。

但是，在设计良好的 Hibernate 领域模型中，我们通常可以看到更多的集合事实上是带有 inverse="true" 的一对多的关联。对于这些关联，更新操作将会在多对一的这一端进行处理。因此对于此类情况，无需考虑其集合的更新性能。

21.5.3. Bag 和 list 是反向集合类中效率最高的

在把 bag 扔进水沟之前，你必须了解，在一种情况下，bag 的性能（包括list）要比 set 高得多：对于指明了 inverse="true" 的集合类（比如说，标准的双向的一对多关联），我们可以在未初始化（fetch）包元素的情况下直接向 bag 或 list 添加新元素！ 这是因为 Collection.add())

或者 Collection.addAll() 方法对 bag 或者 List 总是返回 true（这点与与 Set 不同）。因此对于下面的相同代码来说，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();
```

21.5.4. 一次性删除（One shot delete）

偶尔的，逐个删除集合类中的元素是相当低效的。Hibernate 并没那么笨，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，然后再删除两个。Hibernate 会安排一条 INSERT 语句和两条 DELETE 语句（除非集合类是一个 bag）。这当然是令人满意的。

但是，假设我们删除了 18 个数据，只剩下 2 个，然后新增 3 个。则有两种处理方式：

- 逐一的删除这 18 个数据，再新增三个；
- 删除整个集合类（只用一句 DELETE 语句），然后逐一添加 5 个数据。

Hibernate 还没那么聪明，知道第二种选择可能会比较快。（也许让 Hibernate 不这么聪明也是好事，否则可能会引发意外的“数据库触发器”之类的问题。）

幸运的是，你可以强制使用第二种策略。你需要取消原来的整个集合类（解除其引用），然后再返回一个新的实例化的集合类，只包含需要的元素。有些时候这是非常有用的。

显然，一次性删除并不适用于被映射为 inverse="true" 的集合。

21.6. 监测性能（Monitoring performance）

没有监测和性能参数而进行优化是毫无意义的。Hibernate 为其内部操作提供了一系列的示意图，因此可以从 每个 SessionFactory 抓取其统计数据。

21.6.1. 监测 SessionFactory

你可以有两种方式访问 SessionFactory 的数据记录，第一种就是自己直接调用 sessionFactory.getStatistics() 方法读取、显示统计数据。

此外，如果你打开 StatisticsService MBean 选项，那么 Hibernate 则可以使用 JMX 技术 发布其数据记录。你可以让应用中所有的 SessionFactory 同时共享一个 MBean，也可以每个 SessionFactory 分配一个 MBean。下面的代码即是其演示代码：

```
// MBean service registration for a specific SessionFactory
```

```
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

你可以通过以下方法打开或关闭 SessionFactory 的监测功能:

- 在配置期间，将 hibernate.generate_statistics 设置为 true 或 false;
- 在运行期间，则可以可以通过 sf.getStatistics().setStatisticsEnabled(true) 或 hibernateStatsBean.setStatisticsEnabled(true)

你也可以在程序中调用 clear() 方法重置统计数据，调用 logSummary() 在日志中记录（info 级别）其总结。

21.6.2. 数据记录（Metrics）

Hibernate 提供了一系列数据记录，其记录的内容包括从最基本的信息到与具体场景的特殊信息。所有的测量值都可以由 Statistics 接口 API 进行访问，主要分为三类:

- 使用 Session 的普通数据记录，例如打开的 Session 的个数、取得的 JDBC 的连接数等;
- 实体、集合、查询、缓存等内容的统一数据记录。
- 和具体实体、集合、查询、缓存相关的详细数据记录

例如：你可以检查缓存的命中成功次数，缓存的命中失败次数，实体、集合和查询的使用概率，查询的平均时间等。请注意 Java 中时间的近似精度是毫秒。Hibernate 的数据精度和具体的 JVM 有关，在有些平台上其精度甚至只能精确到 10 秒。

你可以直接使用 getter 方法得到全局数据记录（例如，和具体的实体、集合、缓存区无关的数据），你也可以在具体查询中通过标记实体名、 或 HQL、SQL 语句得到某实体的数据记录。请参考 Statistics、EntityStatistics、CollectionStatistics、SecondLevelCacheStatistics 和 QueryStatistics 的 API 文档以抓取更多信息。下面的代码则是个简单的例子:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

如果你想得到所有实体、集合、查询和缓存区的数据，你可以通过以下方法获得实体、集合、查询和缓存区列表：`getQueries()`、`getEntityNames()`、`getCollectionRoleNames()` 和 `getSecondLevelCacheRegionNames()`。

工具箱指南

可以通过一系列 Eclipse 插件、命令行工具和 Ant 任务来进行与 Hibernate 关联的转换。

除了 Ant 任务外，当前的 Hibernate Tools 也包含了 Eclipse IDE 的插件，用于与现存数据库的逆向工程。

- Mapping Editor: Hibernate XML 映射文件的编辑器，支持自动完成和语法高亮。它也支持对类名和属性/字段名的语义自动完成，比通常的 XML 编辑器方便得多。
- Console: Console 是 Eclipse 的一个新视图。除了对你的 console 配置的树状概览，你还可以获得对你持久化类及其关联的交互式视图。Console 允许你对数据库执行 HQL 查询，并直接在 Eclipse 中浏览结果。
- Development Wizards: 在 Hibernate Eclipse tools 中还提供了几个向导；你可以用向导快速生成 Hibernate 配置文件（cfg.xml），你甚至还可以同现存的数据库 schema 中反向工程出 POJO 源代码与 Hibernate 映射文件。反向工程支持可定制的模版。
-

要得到更多信息，请查阅 Hibernate Tools 包及其文档。

同时，Hibernate 主发行包还附带了一个集成的工具（它甚至可以在 Hibernate“内部”快速运行）SchemaExport ，也就是 hbm2ddl。

22.1. Schema 自动生成（Automatic schema generation）

可以从你的映射文件使用一个 Hibernate 工具生成 DDL。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），因为 DDL 是与供应商高度相关的。

首先，要定制你的映射文件，来改善生成的 schema。下章将涵盖 schema 定制。

22.1.1. 对 schema 定制化（Customizing the schema）

很多 Hibernate 映射元素定义了可选的 length、precision 或者 scale 属性。你可以通过这个属性设置字段的长度、精度、小数点位数。

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

有些 tag 还接受 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）。

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

unique-key 属性可以对成组的字段指定一个唯一键约束（unique key constraint）。目前，unique-key 属性指定的值在生成 DDL 时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。

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

index 属性会用对应的字段（一个或多个）生成一个 index，它指出了这个 index 的名字。如果多个字段对应的 index 名字相同，就会生成包含这些字段的 index。

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

foreign-key 属性可以用来覆盖任何生成的外键约束的名字。

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

很多映射元素还接受 <column> 子元素。这在定义跨越多字段的类型时特别有用。

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property
>
```

default 属性为字段指定一个默认值（在保存被映射的类的新实例之前，你应该将同样的值赋于对应的属性）。

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property
```

```
>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</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。

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property
>
```

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
>
```

下表总结了这些可选属性：

表 22.1. 总结

属性（Attribute）	值（Values）	解释（Interpretation）
length	数字	字段长度
precision	数字	精度（decimal precision）
scale	数字	小数点位数（decimal scale）
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
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）

属性（Attribute）	值（Values）	解释（Interpretation）
foreign-key	foreign_key_name	指明一个外键的名字，它是为关联生成的，或者是为 <one-to-one>, <many-to-one>, <key>, or <many-to-many> 映射元素。注意 inverse="true" 会被 SchemaExport 忽略。
sql-type	SQL column type	覆盖默认的字段类型（只能用于 <column> 属性）
default	SQL 表达式	为字段指定默认值
check	SQL 表达式	对字段或表加入 SQL 约束检查

<comment> 元素可以让你在生成的 schema 中加入注释。

```
<class name="Customer" table="CurCust">
    <comment
>Current customers only</comment>
    ...
</class
>
```

```
<property name="balance">
    <column name="bal">
        <comment
>Balance in USD</comment>
    </column>
</property
>
```

结果是在生成的 DDL 中包含 comment on table 或者 comment on column 语句（假若支持的话）。

22.1.2. 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。

下表显示了 SchemaExport 命令行选项

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaExport options mapping_files
```

表 22.2. SchemaExport 命令行选项

选项	描述
--quiet	不要把脚本输出到 stdout
--drop	只进行 drop tables 的步骤
--create	只创建表
--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

选项	描述
--naming=eg.MyNamingStrategy	选择 NamingStrategy
--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=;	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);
```

22.1.3. 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：

表 22.3. SchemaExport 连接属性

属性名	描述
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）

22.1.4. 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="org.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
>
```

22.1.5. 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 org.hibernate.tool.hbm2ddl.SchemaUpdate options mapping_files

表 22.4. SchemaUpdate 命令行选项

选项	描述
--quiet	不要把脚本输出到 stdout
--text	不把脚本输出到数据库
--naming=eg.MyNamingStrategy	选择 NamingStrategy
--properties=hibernate.properties	从文件读入数据库属性
--config=hibernate.cfg.xml	指定一个 .cfg.xml 文件

你可以在你的应用程序中嵌入 SchemaUpdate 工具：

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

22.1.6. 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="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
```

```
        </fileset>
    </schemaupdate>
</target
>
```

22.1.7. Schema 校验

SchemaValidator 工具会比较数据库现状是否与映射文档“匹配”。注意，SchemaValidator 严重依赖于 JDBC 的 metadata API，因此不是对所有的 JDBC 驱动都适用。这一工具在测试的时候特别有用。

java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaValidator options mapping_files

下表显示了 SchemaValidator 命令行参数：

表 22.5. SchemaValidator 命令行参数

选项	描述
--naming=eg.MyNamingStrategy	选择 NamingStrategy
--properties=hibernate.properties	从文件读入数据库属性
--config=hibernate.cfg.xml	指定一个 .cfg.xml 文件

你可以在你的应用程序中嵌入 SchemaValidator：

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

22.1.8. 使用 Ant 进行 schema 校验

你可以在 Ant 脚本中调用 SchemaValidator：

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemavalidator>
</target
>
```

Additional modules

Hibernate Core also offers integration with some external modules/projects. This includes Hibernate Validator the reference implementation of Bean Validation (JSR 303) and Hibernate Search.

23.1. Bean Validation

Bean Validation standardizes how to define and declare domain model level constraints. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. Bean Validation can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Following the DRY principle, Bean Validation and its reference implementation Hibernate Validator has been designed for that purpose.

The integration between Hibernate and Bean Validation works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts, updates and deletes done by Hibernate.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in a set of `ConstraintViolations`. Among other information, the `ConstraintViolation` contains an error description message that can embed the parameter values bundle with the annotation (eg. size limit), and message strings that may be externalized to a `ResourceBundle`.

23.1.1. Adding Bean Validation

To enable Hibernate's Bean Validation integration, simply add a Bean Validation provider (preferably Hibernate Validation 4) on your classpath.

23.1.2. Configuration

By default, no configuration is necessary.

The `Default` group is validated on entity insert and update and the database model is updated accordingly based on the `Default` group as well.

You can customize the Bean Validation integration by setting the validation mode. Use the javax.persistence.validation.mode property and set it up for example in your persistence.xml file or your hibernate.cfg.xml file. Several options are possible:

- auto (default): enable integration between Bean Validation and Hibernate (callback and ddl generation) only if Bean Validation is present in the classpath.
- none: disable all integration between Bean Validation and Hibernate
- callback: only validate entities when they are either inserted, updated or deleted. An exception is raised if no Bean Validation provider is present in the classpath.
- ddl: only apply constraints to the database schema when generated by Hibernate. An exception is raised if no Bean Validation provider is present in the classpath. This value is not defined by the Java Persistence spec and is specific to Hibernate.

注意

You can use both callback and ddl together by setting the property to callback, ddl

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.mode"
                value="callback, ddl"/>
    </properties>
  </persistence-unit>
</persistence>
```

This is equivalent to auto except that if no Bean Validation provider is present, an exception is raised.

If you want to validate different groups during insertion, update and deletion, use:

- javax.persistence.validation.group.pre-persist: groups validated when an entity is about to be persisted (default to Default)
- javax.persistence.validation.group.pre-update: groups validated when an entity is about to be updated (default to Default)
- javax.persistence.validation.group.pre-remove: groups validated when an entity is about to be deleted (default to no group)
- org.hibernate.validator.group.ddl: groups considered when applying constraints on the database schema (default to Default)

Each property accepts the fully qualified class names of the groups validated separated by a comma (,)

例 23.1. Using custom groups for validation

```
<persistence ...>
  <persistence-unit ...>
    ...
    <properties>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.group.Default, com.acme.group.Strict"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value="com.acme.group.OnDelete"/>
      <property name="org.hibernate.validator.group.ddl"
                value="com.acme.group.DDL"/>
    </properties>
  </persistence-unit>
</persistence>
```

注意

You can set these properties in hibernate.cfg.xml, hibernate.properties or programmatically.

23.1.3. Catching violations

If an entity is found to be invalid, the list of constraint violations is propagated by the ConstraintViolationException which exposes the set of ConstraintViolations.

This exception is wrapped in a RollbackException when the violation happens at commit time. Otherwise the ConstraintViolationException is returned (for example when calling flush(). Note that generally, catchable violations are validated at a higher level (for example in Seam / JSF 2 via the JSF - Bean Validation integration or in your business layer by explicitly calling Bean Validation).

An application code will rarely be looking for a ConstraintViolationException raised by Hibernate. This exception should be treated as fatal and the persistence context should be discarded (EntityManager or Session).

23.1.4. Database schema

Hibernate uses Bean Validation constraints to generate an accurate database schema:

- @NotNull leads to a not null column (unless it conflicts with components or table inheritance)
- @Size.max leads to a varchar(max) definition for Strings

- @Min, @Max lead to column checks (like value <= max)
- @Digits leads to the definition of precision and scale (ever wondered which is which? It's easy now with @Digits :))

These constraints can be declared directly on the entity properties or indirectly by using constraint composition.

For more information check the Hibernate Validator reference documentation [http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/].

23.2. Hibernate Search

23.2.1. Description

Full text search engines like Apache Lucene™ are a very powerful technology to bring free text/efficient queries to applications. If suffers several mismatches when dealing with a object domain model (keeping the index up to date, mismatch between the index structure and the domain model, querying mismatch...) Hibernate Search indexes your domain model thanks to a few annotations, takes care of the database / index synchronization and brings you back regular managed objects from free text queries. Hibernate Search is using Apache Lucene [http://lucene.apache.org] under the cover.

23.2.2. Integration with Hibernate Annotations

Hibernate Search integrates with Hibernate Core transparently provided that the Hibernate Search jar is present on the classpath. If you do not wish to automatically register Hibernate Search event listeners, you can set hibernate.search.autoregister_listeners to false. Such a need is very uncommon and not recommended.

Check the Hibernate Search reference documentation [http://docs.jboss.org/hibernate/stable/search/reference/en-US/html/] for more information.

示例：父子关系（Parent/Child）

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。由于种种原因，最方便的方法是把 Parent 和 Child 都建模成实体类，并创建一个从 Parent 指向 Child 的 <one-to-many> 关联，对新手来说尤其如此。还有一种方法，就是将 Child 声明为一个 <composite-element>（组合元素）。事实上在 Hibernate 中 one to many 关联的默认语义远没有 composite element 贴近 parent / child 关系的通常语义。下面我们会阐述如何使用带有级联的双向一对多关联（idirectional one to many association with cascades）去建立有效、优美的 parent / child 关系。

24.1. 关于 collections 需要注意的一点

Hibernate collections 被当作其所属实体而不是其包含实体的一个逻辑部分。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，collection 所属者的版本值会递增。
- 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。对于父子关系则是完全不适合的，在这种关系下，子对象的生存绑定于父对象的生存周期。

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

假设我们要实现一个简单的从 Parent 到 Child 的 <one-to-many> 关联。

```
<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 会产生两条 SQL 语句：

- 一条 INSERT 语句，为 c 创建一条记录
- 一条 UPDATE 语句，创建从 p 到 c 的连接

这样做不仅效率低，而且违反了 parent_id 列 parent_id 非空的限制。我们可以通过在集合类映射上指定 not-null="true" 来解决违反非空约束的问题：

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set
>
```

然而，这并非是推荐的解决方法。

这种现象的根本原因是从 p 到 c 的连接（外键 parent_id）没有被当作 Child 对象状态的一部分，因而没有在 INSERT 语句中被创建。因此解决的办法就是把这个连接添加到 Child 的映射中。

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

你还需要为类 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();
```

24.3. 级联生命周期（Cascading lifecycle）

需要显式调用 save() 仍然很麻烦，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
>
```

这样上面的代码可以简化为：

```
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）。你需要显式调用 delete() 来删除 Child。

```
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，如果没有父对象，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"，级联仍然是通过遍历 collection 中的元素来处理的。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。

24.4. 级联与未保存值（unsaved-value）

Suppose we loaded up a Parent in one Session, made some changes in a UI action and wanted to persist these changes in a new session by calling update(). The Parent will contain a collection of children and, since the cascading update is enabled, Hibernate needs to know which children are newly instantiated and which represent existing rows in the database. We will also assume that both Parent and Child have generated identifier properties of type Long. Hibernate will use the identifier and version/timestamp property value to determine which of the children are new. (See 第 11.7 节 “自动状态检测”.) In Hibernate3, it is no longer necessary to specify an unsaved-value explicitly.

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();
```

这对于自动生成标识的情况是非常好的，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 Hibernate 没有办法区分新实例化的对象（标识被用户指定了）和前一个 Session 装入的对象。在这种情况下，Hibernate 会使用 timestamp 或 version 属性，或者查询第二级缓存，或者最坏的情况，查询数据库，来确认是否此行存在。

24.5. 结论

这里有不少东西需要融会贯通，可能会让新手感到迷惑。但是在实践中它们都工作地非常好。大部分 Hibernate 应用程序都会经常用到父子对象模式。

在第一段中我们曾经提到另一个方案。上面的这些问题都不会出现在 <composite-element> 映射中，它准确地表达了父子关系的语义。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。

示例：Weblog 应用程序

25.1. 持久化类（Persistent Classes）

下面的持久化类表示一个 weblog 和在其中张贴的一个贴子。他们是标准的父/子关系模型，但是我们会用一个有序包（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;
    }
}
```

25.2. Hibernate 映射

下列的 XML 映射应该是很直白的。例如：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <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"
            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 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.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
>
```

25.3. Hibernate 代码

下面的类演示了我们可以使用 Hibernate 对这些类进行的一些操作：

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.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.persist(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.uniqueResult();
            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);
```

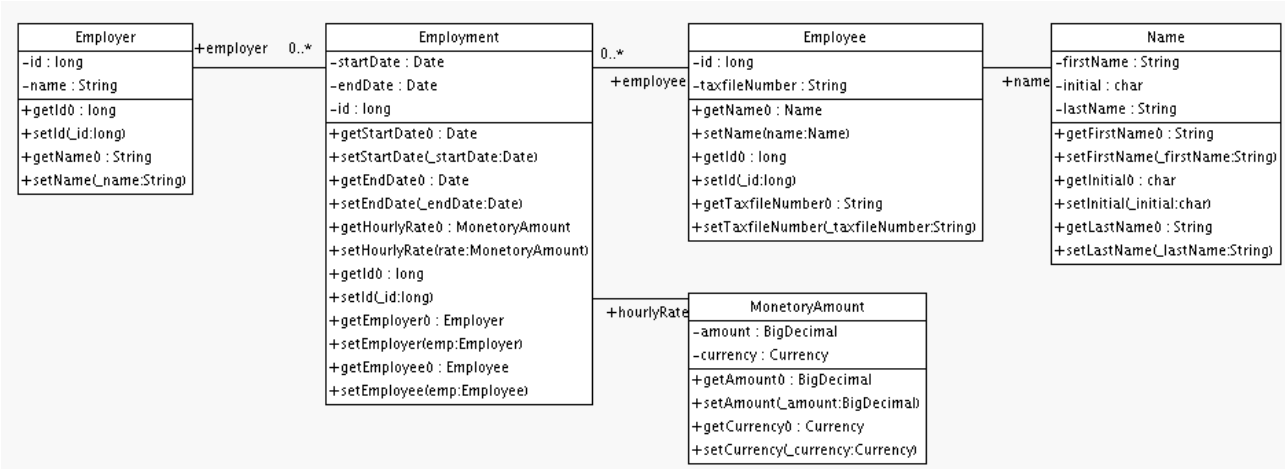
```
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

示例：复杂映射实例

本章展示了一些较为复杂的关系映射。

26.1. Employer（雇主）/Employee（雇员）

下面关于 Employer 和 Employee 的关系模型使用了一个真实的实体类（Employment）来表述，这是因为对于相同的雇员和雇主可能会有多个雇佣时间段。对于金额和雇员姓名，用 Components 建模。



映射文件可能是这样：

```
<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="MonetaryAmount">
            <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
>
```

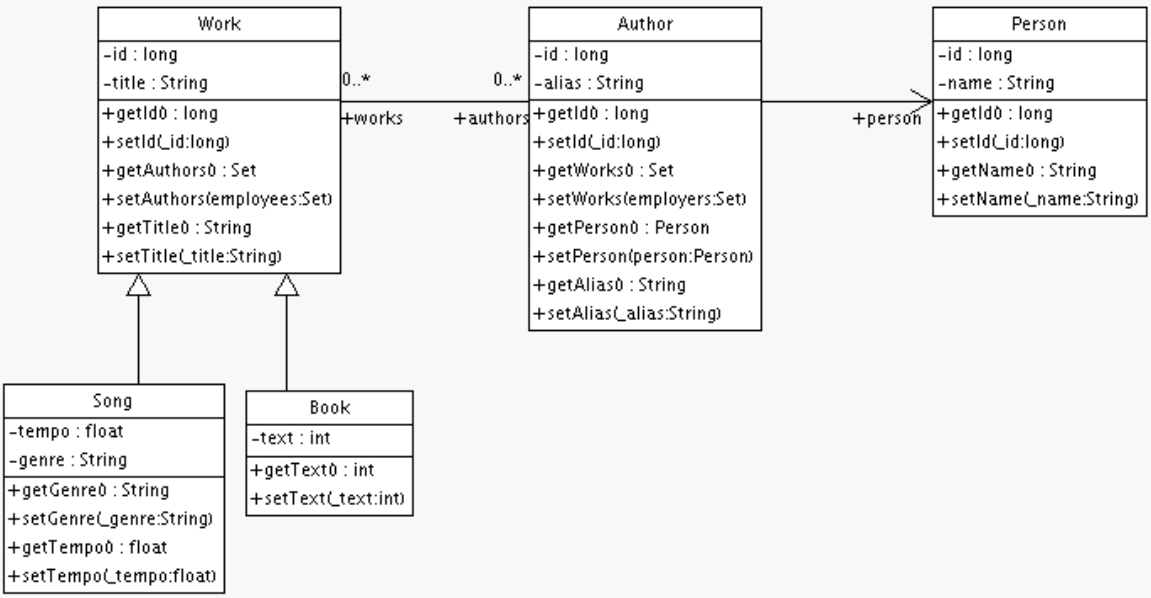
用 SchemaExport 生成表结构。

```
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
```

26.2. Author（作家）/Work（作品）

考虑下面的 Work, Author 和 Person 模型的关系。我们用多对多关系来描述 Work 和 Author，用一对一关系来描述 Author 和 Person，另一种可能性是 Author 继承 Person。



下面的映射文件正确的描述了这些关系：

```
<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">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </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">
            <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
>
```

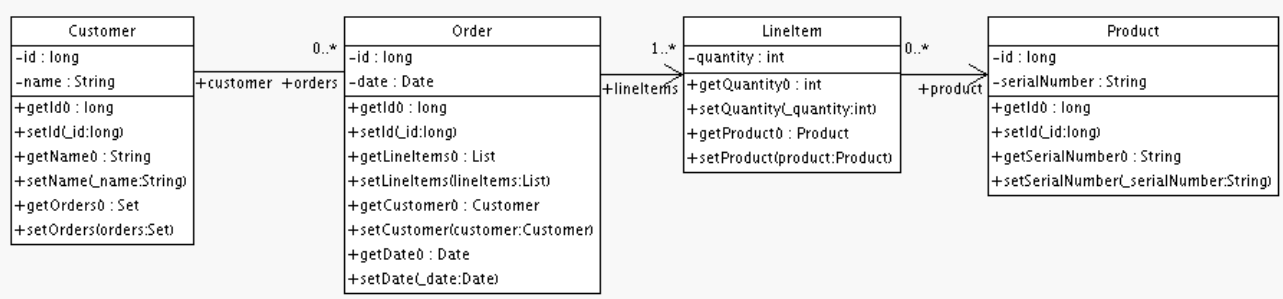
映射中有 4 个表。works，authors 和 persons 分别保存着 work，author 和 person 的数据。author_work 是 authors 和 works 的关联表。表结构是由 SchemaExport 生成的：

```
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
```

26.3. Customer（客户）/Order（订单）/Product（产品）

现在来考虑 Customer，Order， LineItem 和 Product 关系的模型。Customer 和 Order 之间 是一对多的关系，但是我们怎么来描述 Order / LineItem / Product呢？ 我可以把 LineItem 作为描述 Order 和 Product 多对多关系的关联类，在 Hibernate，这叫做组合元素。



映射文件如下：

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="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">
            <key column="order_id"/>
            <list-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 分别保存着 customer, order, order line item 和 product 的数据。 line_items 也作为连接 orders 和 products 的关联表。

```
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
```

26.4. 杂例

这些例子全部来自于 Hibernate 的 test suite，同时你也可以找到其他有用的例子。可以参考 Hibernate 的 test 目录。

26.4.1. "Typed" 一对一关联

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula
>name</formula>
        <formula
>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula
>name</formula>
        <formula
>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class
>
```

26.4.2. 组合键示例

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true" length="100"/>
```

```
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
```

```
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class
>
```

26.4.3. 共有组合键属性的多对多（Many-to-many with shared composite key attribute）

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
```

```
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula
>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula
>org</formula>
        </many-to-many>
    </set>
</class>
```

26.4.4. 基于内容的识别

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>
```

```
    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class
>
```

26.4.5. 备用键的联合

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>
```

```
<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class
>
```

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（identifier properties）：

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）。

使用自然键（natural keys）标识：

对所有的实体都标识出自然键，用 <natural-id> 进行映射。实现 equals() 和 hashCode()，在其中用组成自然键的属性进行比较。

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，这一点尤其重要。

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。把查询字符串放在映射文件中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。你也应该考虑在查询中使用命名参数。

不要自己来管理 JDBC 连接：

Hibernate 允许应用程序自己来管理 JDBC 连接，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 org.hibernate.connection.ConnectionProvider。

考虑使用用户自定义类型（custom 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 org.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：

在系统中对性能要求很严格的一些部分，某些操作也许直接使用 JDBC 会更好。但是请先确认这的确是一个瓶颈，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 然后将 JDBC 操作包裹为 org.hibernate.jdbc.Work 并使用 JDBC 连接。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 冲刷（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，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，在一个特定的 transaction 中改变查询和其它操作的顺序。

在三层结构中，考虑使用脱管对象（detached object）：

当使用一个 servlet / session bean 类型的架构的时候，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.merge() 或者 Session.saveOrUpdate() 来与数据库同步。

在两层结构中，考虑使用长持久上下文（long persistence contexts）：

为了得到最佳的可伸缩性，数据库事务（Database Transaction）应该尽可能的短。但是，程序常常需要实现长时间运行的“应用程序事务（Application Transaction）”，包含一个从用户的观点来看的原子操作。这个应用程序事务可能跨越多次从用户请求到得到反馈的循环。用脱管对象（与 session 脱离的对象）来实现应用程序事务是常见的。或者，尤其在两层结构中，把 Hibernate Session 从 JDBC 连接中脱离开，下次需要用的时候再连接上。绝不要把一个 Session 用在多个应用程序事务（Application Transaction）中，否则你的数据可能会过期失效。

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，必须要回滚 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 Session.get() 或者进行一次查询。

对于关联优先考虑 lazy fetching：

谨慎的使用主动抓取（eager fetching）。对于关联来说，若其目标是无法在第二级缓存中完全缓存所有实例的类，应该使用代理（proxies）与/或具有延迟加载属性的集合（lazy collections）。若目标是可以被缓存的，尤其是缓存的命中率非常高的情况下，应该使用 lazy="false"，明确的禁止掉 eager fetching。如果那些特殊的确实适合使用 join fetch 的场合，请在查询中使用 left join fetch。

使用 open session in view 模式，或者执行严格的装配期（assembly phase）策略来避免再次抓取数据带来的问题：

Hibernate 让开发者们摆脱了繁琐的 Data Transfer Objects（DTO）。在传统的 EJB 结构中，DTO 有双重作用：首先，他们解决了 entity bean 无法序列化的问题；其次，他们隐含地定义了一个装配期，在此期间，所有在 view 层需要用到的数据，都被抓取、集中到了 DTO 中，然后控制才被装到表示层。Hibernate 终结了第一个作用。然而，除非你做好了在整个渲染过程中都维护一个打开的持久化上下文（session）的准备，你仍然需要一个装配期（想象一下，你的业务方法与你的表示层有严格的契约，数据总是被放置到脱管对象中）。这并非是 Hibernate 的限制，这是实现安全的事务化数据访问的基本需求。

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。然而，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。

不要用怪异的连接映射：

多对多连接用得好的例子实际上相当少见。大多数时候你在“连接表”中需要保存额外的信息。这种情况下，用两个指向中介类的一对多的连接比较好。实际上，我们认为绝大多数的连接是一对多和多对一的。i因此，你应该谨慎使用其它连接风格。

偏爱双向关联：

单向关联更加难于查询。在大型应用中，几乎所有的关联必须在查询中可以双向导航。

数据库移植性考量

28.1. 移植性基础

Hibernate（实际上是整个 Object/Relational Mapping）的一个卖点是数据库的移植性。这意味着内部的 IT 用户可以改变数据库供应商，或者可部署的应用程序/框架使用 Hibernate 来同时使用多个数据库产品。不考虑具体的应用情景，这里的基本概念是 Hibernate 可帮助你运行多种数据库而无需修改你的代码，理想情况下甚至不用修改映射元数据。

28.2. Dialect

Hibernate 的移植性的首要问题是方言（dialect），也就是 org.hibernate.dialect.Dialect 合约的具体实例。方言封装了 Hibernate 和特定数据库通讯以完成某些任务如获取序列值或构建 SELECT 查询等的所有差异。Hibernate 捆绑了用于许多最常用的数据库的方言。如果你发现自己使用的数据库不在其中，编写自定义的方言也不是很困难的事情。

28.3. 方言的使用

最开始，Hibernate 总是要求用户指定所使用的方言（dialect）。在用户希望同时使用多个数据库时就会出现问题。通常这要求用户配置 Hibernate 方言或者定义自己设置这个值的方法。

从版本 3.2 开始，Hibernate 引入了方言的自动检测，它基于从该数据库的 java.sql.Connection 上获得的 java.sql.DatabaseMetaData。这是一个更好的方案，但它局限于 Hibernate 已知的数据库且无法进行配置和覆盖。

Starting with version 3.3, Hibernate has a fare more powerful way to automatically determine which dialect to should be used by relying on a series of delegates which implement the org.hibernate.dialect.resolver.DialectResolver which defines only a single method:

```
public Dialect resolveDialect(DatabaseMetaData metaData) throws JDBCConnectionException
```

The basic contract here is that if the resolver 'understands' the given database metadata then it returns the corresponding Dialect; if not it returns null and the process continues to the next resolver. The signature also identifies org.hibernate.exception.JDBCConnectionException as possibly being thrown. A JDBCConnectionException here is interpreted to imply a "non transient" (aka non-recoverable) connection problem and is used to indicate an immediate stop to resolution attempts. All other exceptions result in a warning and continuing on to the next resolver.

这些解析者最棒的功能是用户也可以注册自定义的解析者，它们将在内置的解析者之前被调用。在许多情况下这可能很有用：它可以轻易地集成内置方言之外的方言的自动检测；它让你可以使用自定义的方言等。要注册一个或多个解析者，只要用 'hibernate.dialect_resolvers' 配置设

置指定它们（由逗号、制表符或空格隔开）就可以了（请参考 org.hibernate.cfg.Environment 上的 DIALECT_RESOLVERS）。

28.4. 标识符的生成

When considering portability between databases, another important decision is selecting the identifier generation stratagy you want to use. Originally Hibernate provided the native generator for this purpose, which was intended to select between a sequence, identity, or table strategy depending on the capability of the underlying database. However, an insidious implication of this approach comes about when targtetting some databases which support identity generation and some which do not. identity generation relies on the SQL definition of an IDENTITY (or auto-increment) column to manage the identifier value; it is what is known as a post-insert generation strategy becauase the insert must actually happen before we can know the identifier value. Because Hibernate relies on this identifier value to uniquely reference entities within a persistence context it must then issue the insert immediately when the users requests the entitiy be associated with the session (like via save() e.g.) regardless of current transactional semantics.

注意

Hibernate was changed slightly once the implication of this was better understood so that the insert is delayed in cases where that is feasible.

The underlying issue is that the actual semanctics of the application itself changes in these cases.

Starting with version 3.2.3, Hibernate comes with a set of enhanced [http://in.relation.to/2082.lace] identifier generators targetting portability in a much different way.

注意

There are specifically 2 bundled enhancedgenerators:

- org.hibernate.id.enhanced.SequenceStyleGenerator
- org.hibernate.id.enhanced.TableGenerator

The idea behind these generators is to port the actual semantics of the identifer value generation to the different databases. For example, the org.hibernate.id.enhanced.SequenceStyleGenerator mimics the behavior of a sequence on databases which do not support sequences by using a table.

28.5. 数据库函数

警告

这是 Hibernate 需要提高的一个领域。从可移植性来说，这个功能可以很好地处理 HQL 的内容，但在其他方面就有所欠缺。

用户可以以多种方式引用 SQL 函数。然而，不是所有的数据库都支持相同的函数集。Hibernate 提供了一种映射逻辑函数名到代理的方法，这个代理知道如何解析特定的函数，甚至可能使用完全不同的物理函数调用。

重要

从技术上来讲，这个函数注册是通过 org.hibernate.dialect.function.SQLFunctionRegistry 类进行处理的，它的目的是允许用户提供自定义的函数定义而无需提供自定义的方言。这种特殊的行为目前还未全部开发完毕。

其中一些功能已经实现，如用户可以在程序里用 org.hibernate.cfg.Configuration 注册函数且这些函数可被 HQL 识别。

28.6. 类型映射

本节内容仍未完成...

参考资料

[PoEAA] Patterns of Enterprise Application Architecture. 0-321-12742-0. 由 Martin Fowler. 版权 © 2003 Pearson Education, Inc.. Addison-Wesley Publishing Company.

[JPwH] Java Persistence with Hibernate. Second Edition of Hibernate in Action. 1-932394-88-5. http://www.manning.com/bauer2 . 由 Christian Bauer和Gavin King. 版权 © 2007 Manning Publications Co.. Manning Publications Co..