



Hibernate教程

极客学院出版

前言

Hibernate 是一个高性能的对象/关系型持久化存储和查询的服务，其遵循开源的 GNU Lesser General Public License (LGPL) 而且可以免费下载。Hibernate 不仅关注于从 Java 类到数据库表的映射（也有 Java 数据类型到 SQL 数据类型的映射），另外也提供了数据查询和检索服务。

这个教程将指导你如何以简单的方式使用 Hibernate 来开发基于数据库的 Web 应用程序。

适用人群

这个教程是为需要理解 Hibernate 框架和 API 的 Java 编程人员设计的。读完这份教程后你将发现自己在使用 Hibernate 上从一个中等程度迈向更高的层次。

学习前提

我们假设你已经很好的理解了 Java 编程语言。若对关系型数据库，JDBC，和 SQL 有些基本的了解会更好。

目录

前言	1
第 1 章 ORM 概览	3
第 2 章 简介	7
第 3 章 架构	10
第 4 章 环境	14
第 5 章 配置	17
第 6 章 会话	21
第 7 章 持久化类	25
第 8 章 映射文件	28
第 9 章 映射类型	32
第 10 章 例子	35
第 11 章 O/R 映射	43
第 12 章 注释	46
第 13 章 查询语言	55
第 14 章 标准查询	61
第 15 章 原生 SQL	71
第 16 章 缓存	78
第 17 章 批处理	84
第 18 章 拦截器	91



ORM 概览



什么是 JDBC?

JDBC 代表 **Java Database Connectivity**，它是提供了一组 Java API 来访问关系数据库的 Java 程序。这些 Java APIs 可以使 Java 应用程序执行 SQL 语句，能够与任何符合 SQL 规范的数据库进行交互。

JDBC 提供了一个灵活的框架来编写操作数据库的独立的应用程序，该程序能够运行在不同的平台上且不需修改，能够与不同的 DBMS 进行交互。

JDBC 的优点和缺点

JDBC 的优点	JDBC 的缺点
干净整洁的 SQL 处理	大项目中使用很复杂
大数据下有良好的性能	很大的编程成本
对于小应用非常好	没有封装
易学的简易语法	难以实现 MVC 的概念
	查询需要指定 DBMS

为什么是对象关系映射 (ORM)?

当我们工作在一个面向对象的系统中时，存在一个对象模型和关系数据库不匹配的问题。RDBMSs 用表格的形式存储数据，然而像 Java 或者 C# 这样的面向对象的语言它表示一个对象关联图。考虑下面的带有构造方法和公有方法的 Java 类：

```
public class Employee {
    private int id;
    private String first_name;
    private String last_name;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.first_name = fname;
        this.last_name = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
}
```

```

public String getFirstName() {
    return first_name;
}
public String getLastName() {
    return last_name;
}
public int getSalary() {
    return salary;
}
}

```

现考虑以上的对象需要被存储和索引进下面的 RDBMS 表格中：

```

create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name VARCHAR(20) default NULL,
    salary INT default NULL,
    PRIMARY KEY (id)
);

```

第一个问题，如果我们开发了几页代码或应用程序后，需要修改数据库的设计怎么办？第二个问题，在关系型数据库中加载和存储对象时我们要面临以下五个不匹配的问题。

不匹配	描述
粒度	有时你将会有一个对象模型，该模型类的数量比数据库中关联的表的数量更多
继承	RDBMSs 不会定义任何在面向对象编程语言中本来就有的继承
身份	RDBMS 明确定义一个 'sameness' 的概念：主键。然而，Java 同时定义了对象判等（ <code>a==b</code> ）和对象值判等（ <code>a.equals(b)</code> ）
关联	面向对象的编程语言使用对象引用来表示关联，而一个 RDBMS 使用外键来表示对象关联
导航	在 Java 中和在 RDBMS 中访问对象的方式完全不相同

Object-Relational Mapping (ORM) 是解决以上所有不匹配问题的方案。

什么是 ORM？

ORM 表示 Object-Relational Mapping (ORM)，是一个方便在关系数据库和类似于 Java，C# 等面向对象的编程语言中转换数据的技术。一个 ORM 系统相比于普通的 JDBC 有以下的优点。

序号	优点
1	使用业务代码访问对象而不是数据库中的表
2	从面向对象逻辑中隐藏 SQL 查询的细节

序号	优点
3	基于 JDBC 的 'under the hood'
4	没有必要去处理数据库实现
5	实体是基于业务的概念而不是数据库的结构
6	事务管理和键的自动生成
7	应用程序的快速开发

一个 ORM 解决方案由以下四个实体组成：

序号	优点
1	一个 API 来在持久类的对象上实现基本的 CRUD 操作
2	一个语言或 API 来指定引用类和属性的查询
3	一个可配置的服务用来指定映射元数据
4	一个技术和事务对象交互来执行 dirty checking, lazy association fetching 和其它优化的功能

Java ORM 框架

在 Java 中有几个持久化的框架和 ORM 选项。一个持久化的框架是 ORM 存储和索引对象到关系型数据库的服务。

- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate
- And many more



简介



Hibernate 是由 Gavin King 于 2001 年创建的开放源代码的对象关系框架。它强大且高效的构建具有关系对象持久性和查询服务的 Java 应用程序。

Hibernate 将 Java 类映射到数据库表中，从 Java 数据类型中映射到 SQL 数据类型中，并把开发人员从 95% 的公共数据持续性编程工作中解放出来。

Hibernate 是传统 Java 对象和数据库服务器之间的桥梁，用来处理基于 O/R 映射机制和模式的那些对象。



图片 2.1 image

Hibernate 优势

- Hibernate 使用 XML 文件来处理映射 Java 类别到数据库表格中，并且不用编写任何代码。
- 为在数据库中直接储存和检索 Java 对象提供简单的 APIs。
- 如果在数据库中或任何其它表格中出现变化，那么仅需要改变 XML 文件属性。
- 抽象不熟悉的 SQL 类型，并为我们提供工作中所熟悉的 Java 对象。
- Hibernate 不需要应用程序服务器来操作。
- 操控你数据库中对象复杂的关联。
- 最小化与访问数据库的智能提取策略。
- 提供简单的数据询问。

支持的数据库

Hibernate 支持几乎所有的主要 RDBMS。以下是一些由 Hibernate 所支持的数据库引擎。

- HSQL Database Engine
- DB2/NT
- MySQL
- PostgreSQL

- FrontBase
- Oracle
- Microsoft SQL Server Database
- Sybase SQL Server
- Informix Dynamic Server

支持的技术

Hibernate 支持多种多样的其它技术，包括以下：

- XDoclet Spring
- J2EE
- Eclipse plug-ins
- Maven



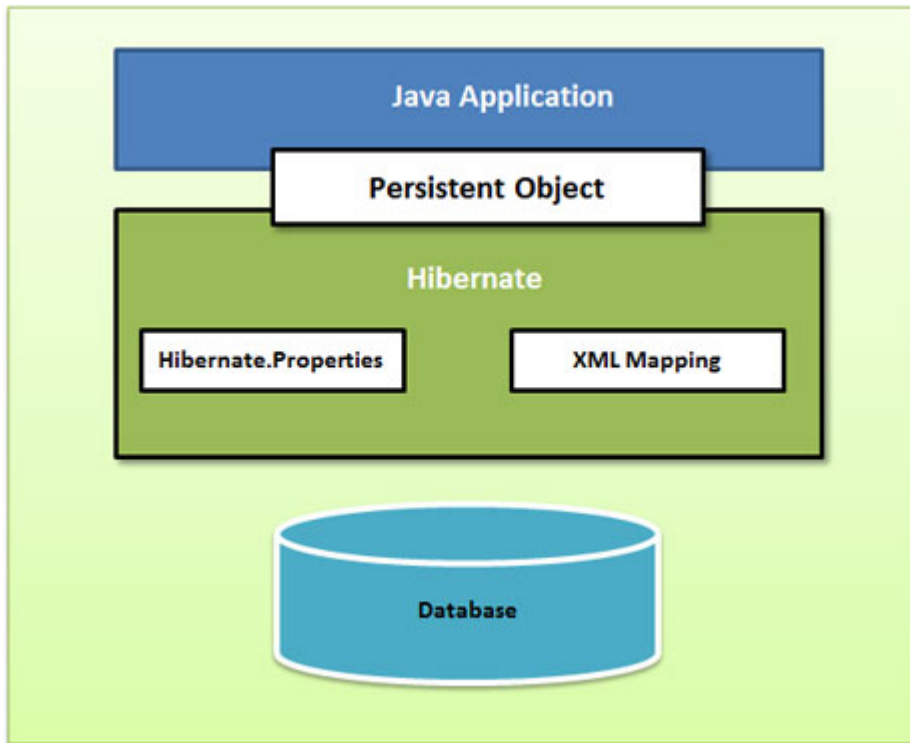
3

架构



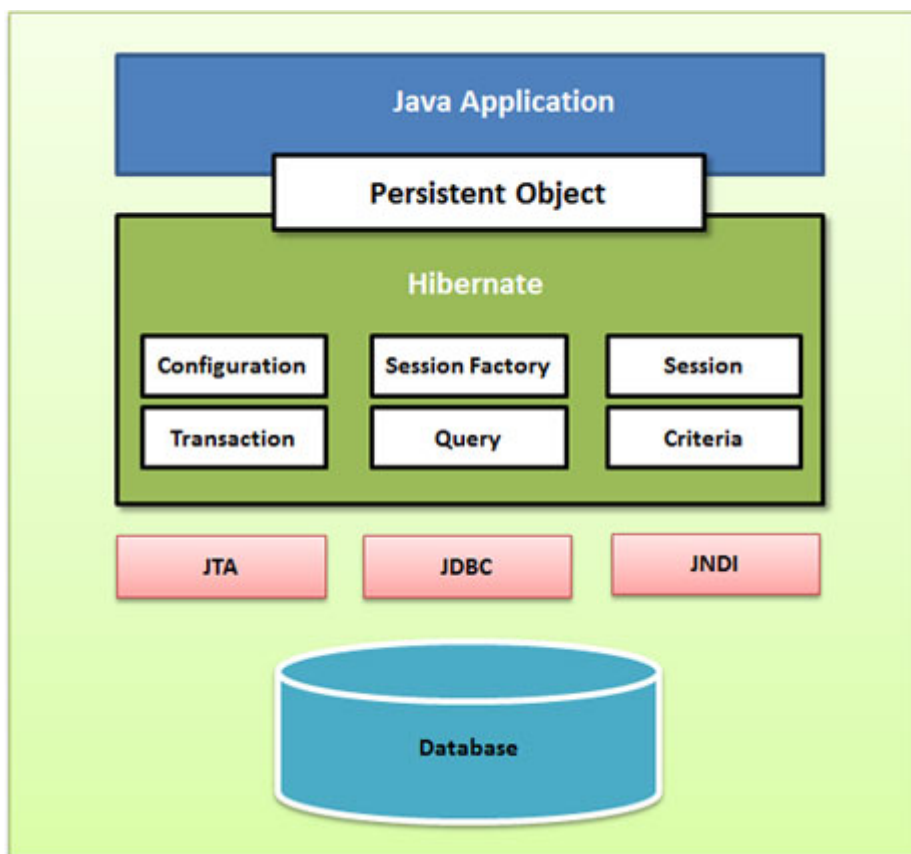
Hibernate 架构是分层的，作为数据访问层，你不必知道底层 API。Hibernate 利用数据库以及配置数据来为应用程序提供持续性服务（以及持续性对象）。

下面是一个非常高水平的 Hibernate 应用程序架构视图。



图片 3.1 image

下面是一个详细的 Hibernate 应用程序体系结构视图以及一些重要的类。



图片 3.2 image

Hibernate 使用不同的现存 Java API，比如 JDBC，Java 事务 API（JTA），以及 Java 命名和目录界面（JNDI）。JDBC 提供了一个基本的抽象级别的通用关系数据库的功能，Hibernate 支持几乎所有带有 JDBC 驱动的数据库。JNDI 和 JTA 允许 Hibernate 与 J2EE 应用程序服务器相集成。

下面的部分简要地描述了在 Hibernate 应用程序架构所涉及的每一个类对象。

配置对象

配置对象是你在任何 Hibernate 应用程序中创造的第一个 Hibernate 对象，并且经常只在应用程序初始化期间创造。它代表了 Hibernate 所需一个配置或属性文件。配置对象提供了两种基础组件。

- **数据库连接：**由 Hibernate 支持的一个或多个配置文件处理。这些文件是 `hibernate.properties` 和 `hibernate.cfg.xml`。
- **类映射设置：**这个组件创造了 Java 类和数据库表格之间的联系。

SessionFactory 对象

配置对象被用于创建一个 SessionFactory 对象，使用提供的配置文件为应用程序依次配置 Hibernate，并允许实例化一个会话对象。SessionFactory 是一个线程安全对象并由应用程序所有的线程所使用。

SessionFactory 是一个重量级对象所以通常它都是在应用程序启动时创造然后留存为以后使用。每个数据库需要一个 SessionFactory 对象使用一个单独的配置文件。所以如果你使用多种数据库那么你要创造多种 Session Factory 对象。

Session 对象

一个会话被用于与数据库的物理连接。Session 对象是轻量级的，并被设计为每次实例化都需要与数据库的交互。持久对象通过 Session 对象保存和检索。

Session 对象不应该长时间保持开启状态因为它们通常情况下并非线程安全，并且它们应该按照所需创造和销毁。

Transaction 对象

一个事务代表了与数据库工作的一个单元并且大部分 RDBMS 支持事务功能。在 Hibernate 中事务由底层事务管理器和事务（来自 JDBC 或者 JTA）处理。

这是一个选择性对象，Hibernate 应用程序可能不选择使用这个接口，而是在自己应用程序代码中管理事务。

Query 对象

Query 对象使用 SQL 或者 Hibernate 查询语言（HQL）字符串在数据库中来检索数据并创造对象。一个查询的实例被用于连结查询参数，限制由查询返回的结果数量，并最终执行查询。

Criteria 对象

Criteria 对象被用于创造和执行面向规则查询的对象来检索对象。



环境

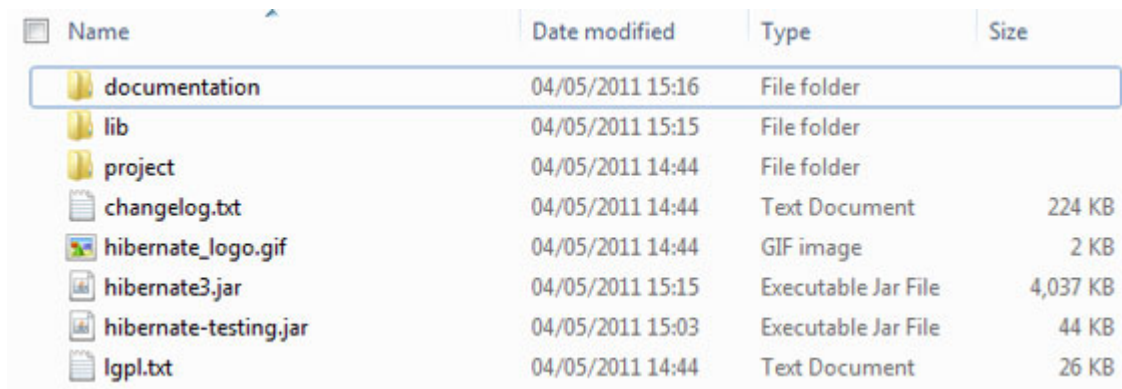


这个章节会告诉你为了给 Hibernate 应用准备需要的开发环境，该怎样安装 Hibernate 应用和一些其它相关的包。我们会用 MySQL 数据库来对一些 Hibernate 应用的例子进行试验，所以先要确保你已经安装过了 MySQL 数据库。想了解更多的关于 MySQL 数据库的详情的话，你可以搜索我们的 [MySQL教程 \(http://wiki.jikexueyu.com/list/mysql/\)](http://wiki.jikexueyu.com/list/mysql/)。

下载 Hibernate

如果你已经在你的机器上安装了 Java 的最新版本，那么按照以下这些简易的步骤来下载并安装 Hibernate 在你的机器上就可以了。

- 首先要在想要把 Hibernate 应用安装在 Windows 系统或是 Unix 系统这两者之间做出选择，之后继续到下一个步骤去下载与 Windows 系统对应的 .zip 文件或是与 Unix 系统对应的 .tar.gz 文件。
- 之后到 <http://www.hibernate.org/downloads> 这个网址来下载最新版本的 Hibernate 应用。
- 在写这个教程时我下载的是 hibernate-distribution-3.6.4.Final 这个版本的应用，在这个版本的应用下当我们解压下载的文件时会显示以下的目录结构。



Name	Date modified	Type	Size
documentation	04/05/2011 15:16	File folder	
lib	04/05/2011 15:15	File folder	
project	04/05/2011 14:44	File folder	
changelog.txt	04/05/2011 14:44	Text Document	224 KB
hibernate_logo.gif	04/05/2011 14:44	GIF image	2 KB
hibernate3.jar	04/05/2011 15:15	Executable Jar File	4,037 KB
hibernate-testing.jar	04/05/2011 15:03	Executable Jar File	44 KB
lgpl.txt	04/05/2011 14:44	Text Document	26 KB

图片 4.1 image

安装 Hibernate

一旦你下好并且解压了 Hibernate 应用最新版本的安装文件，你需要执行以下两个简单的步骤。一定要确保你把你的 CLASSPATH 变量设置合理，否则当你编译你的应用时可能会遇到问题。

- 首先把从 /lib 复制来的所有库文件拷贝到 CLASSPATH 里，并且改变你的 CLASSPATH 变量来涵盖所有的 JAR。
- 最后复制 hibernate3.jar 这个文件到 CLASSPATH 里。这个文件位于安装文件的根目录里，它是 Hibernate 应用针对的主要 JAR。

Hibernate 的前提

以下是一个 Hibernate 应用需要的有关包/库的表格，在安装 Hibernate 应用之前你需要先安装它们。为了安装这些包你必须把来自 `lib` 的库文件拷贝到 CLASSPATH，并按以下说明相应地改变 CLASSPATH 变量。

S.N.	包/库
1	dom4j – XML 解析 www.dom4j.org/
2	Xalan – XSLT 处理器 http://xml.apache.org/xalan-j/
3	Xerces – The Xerces Java 解析器 http://xml.apache.org/xerces-j/
4	cglib – Java 类生成库 http://cglib.sourceforge.net/
5	log4j – 日志控制 http://logging.apache.org/log4j
6	Commons – 日志，邮件等 http://jakarta.apache.org/commons
7	SLF4J – 简单日志门面 http://www.slf4j.org



T



配置



Hibernate 需要事先知道在哪里找到映射信息，这些映射信息定义了 Java 类怎样关联到数据库表。Hibernate 也需要一套相关数据库和其它相关参数的配置设置。所有这些信息通常是作为一个标准的 Java 属性文件提供的，名叫 `hibernate.properties`。又或者是作为 XML 文件提供的，名叫 `hibernate.cfg.xml`。

我们将考虑 `hibernate.cfg.xml` 这个 XML 格式文件，来决定在我的例子里指定需要的 Hibernate 应用属性。这个 XML 文件中大多数的属性是不需要修改的。这个文件保存在应用程序的类路径的根目录里。

■ Hibernate 属性

下面是一个重要的属性列表，您可能需要表中的属性来在单独的情况下配置数据库。

S.N.	属性和描述
1	<code>hibernate.dialect</code> 这个属性使 Hibernate 应用为被选择的数据库生成适当的 SQL。
2	<code>hibernate.connection.driver_class</code> JDBC 驱动程序类。
3	<code>hibernate.connection.url</code> 数据库实例的 JDBC URL。
4	<code>hibernate.connection.username</code> 数据库用户名。
5	<code>hibernate.connection.password</code> 数据库密码。
6	<code>hibernate.connection.pool_size</code> 限制在 Hibernate 应用数据库连接池中连接的数量。
7	<code>hibernate.connection.autocommit</code> 允许在 JDBC 连接中使用自动提交模式。

如果您正在使用 JNDI 和数据库应用程序服务器然后您必须配置以下属性:

S.N.	属性和描述
1	<code>hibernate.connection.datasource</code> 在应用程序服务器环境中您正在使用的应用程序 JNDI 名。
2	<code>hibernate.jndi.class</code> JNDI 的 <code>InitialContext</code> 类。
3	<code>hibernate.jndi.<JNDIpropertyname></code> 在 JNDI 的 <code>InitialContext</code> 类中通过任何你想要的 Java 命名和目录接口属性。
4	<code>hibernate.jndi.url</code> 为 JNDI 提供 URL。
5	<code>hibernate.connection.username</code> 数据库用户名。
6	<code>hibernate.connection.password</code> 数据库密码。

Hibernate 和 MySQL 数据库

MySQL 数据库是目前可用的开源数据库系统中最受欢迎的数据库之一。我们要创建 `hibernate.cfg.xml` 配置文件并将其放置在应用程序的 CLASSPATH 的根目录里。你要确保在你的 MySQL 数据库中 `testdb` 数据库是可用的，而且你要有一个用户 `test` 可用来访问数据库。

XML 配置文件一定要遵守 Hibernate 3 Configuration DTD，在 <http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd> 这个网址中是可以找到的。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>

    <!-- Assume test is the database name -->
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
    </property>
    <property name="hibernate.connection.username">
      root
    </property>
    <property name="hibernate.connection.password">
      root123
    </property>

    <!-- List of XML mapping files -->
    <mapping resource="Employee.hbm.xml"/>

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

上面的配置文件包含与 `hibernate-mapping` 文件相关的 `<mapping>` 标签，我们将在下章看看 `hibernate mapping` 文件到底是什么并且要知道为什么用它，怎样用它。以下是各种重要数据库同源语属性类型的列表：

数据库	方言属性
DB2	org.hibernate.dialect.DB2Dialect
HSQLDB	org.hibernate.dialect.HSQLDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Ingres	org.hibernate.dialect.IngresDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
MySQL	org.hibernate.dialect.MySQLDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect



T



会话



Session 用于获取与数据库的物理连接。Session 对象是轻量级的，并且设计为在每次需要与数据库进行交互时被实例化。持久态对象被保存，并通过 Session 对象检索找回。

该 Session 对象不应该长时间保持开放状态，因为它们通常不能保证线程安全，而应该根据需求被创建和销毁。Session 的主要功能是为映射实体类的实例提供创建，读取和删除操作。这些实例可能在给定时间点时存在于以下三种状态之一：

- **瞬时状态**: 一种新的持久性实例，被 Hibernate 认为是瞬时的，它不与 Session 相关联，在数据库中没有与之关联的记录且无标识符值。
- **持久状态**: 可以将一个瞬时状态实例通过与一个 Session 关联的方式将其转化为持久状态实例。持久状态实例在数据库中没有与之关联的记录，有标识符值，并与一个 Session 关联。
- **脱管状态**: 一旦关闭 Hibernate Session，持久状态实例将会成为脱管状态实例。

若 Session 实例的持久态类别是序列化的，则该 Session 实例是序列化的。一个典型的事务应该使用以下语法：

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

如果 Session 引发异常，则事务必须被回滚，该 session 必须被丢弃。

Session 接口方法

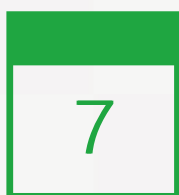
Session 接口提供了很多方法，但在以下讲解中我将仅列出几个我们会在本教程中应用的重要方法。您可以查看 Hibernate 文件，查询与 Session 及 SessionFactory 相关的完整方法目录。

序号	Session 方法及说明
1	Transaction beginTransaction() 开始工作单位，并返回关联事务对象。

序号	Session 方法及说明
2	<code>void cancelQuery()</code> 取消当前的查询执行。
3	<code>void clear()</code> 完全清除该会话。
4	<code>Connection close()</code> 通过释放和清理 JDBC 连接以结束该会话。
5	<code>Criteria createCriteria(Class persistentClass)</code> 为给定的实体类或实体类的超类创建一个新的 Criteria 实例。
6	<code>Criteria createCriteria(String entityName)</code> 为给定的实体名称创建一个新的 Criteria 实例。
7	<code>Serializable getIdentifier(Object object)</code> 返回与给定实体相关联的会话的标识符值。
8	<code>Query createFilter(Object collection, String queryString)</code> 为给定的集合和过滤字符创建查询的新实例。
9	<code>Query createQuery(String queryString)</code> 为给定的 HQL 查询字符创建查询的新实例。
10	<code>SQLQuery createSQLQuery(String queryString)</code> 为给定的 SQL 查询字符串创建 SQLQuery 的新实例。
11	<code>void delete(Object object)</code> 从数据存储中删除持久化实例。
12	<code>void delete(String entityName, Object object)</code> 从数据存储中删除持久化实例。
13	<code>Session get(String entityName, Serializable id)</code> 返回给定命名的且带有给定标识符或 null 的持久化实例（若无该种持久化实例）。
14	<code>SessionFactory getSessionFactory()</code> 获取创建该会话的 session 工厂。
15	<code>void refresh(Object object)</code> 从基本数据库中重新读取给定实例的状态。
16	<code>Transaction getTransaction()</code> 获取与该 session 关联的事务实例。
17	<code>boolean isConnected()</code> 检查当前 session 是否连接。
18	<code>boolean isDirty()</code> 该 session 中是否包含必须与数据库同步的变化？
19	<code>boolean isOpen()</code> 检查该 session 是否仍处于开启状态。
20	<code>Serializable save(Object object)</code> 先分配一个生成的标识，以保持给定的瞬时状态实例。
21	<code>void saveOrUpdate(Object object)</code> 保存（对象）或更新（对象）给定的实例。
22	<code>void update(Object object)</code> 更新带有标识符且是给定的处于脱管状态的实例的持久化实例。

序号	Session 方法及说明
----	---------------

23	<code>void update(String entityName, Object object)</code> 更新带有标识符且是给定的处于脱管状态的实例的持久化实例。
----	--



持久化类



Hibernate 的完整概念是提取 Java 类属性中的值，并且将它们保存到数据库表中。映射文件能够帮助 Hibernate 确定如何从该类中提取值，并将它们映射在表格和相关域中。

在 Hibernate 中，其对象或实例将会被存储在数据库表中的 Java 类被称为持久化类。若该类遵循一些简单的规则或者被大家所熟知的 Plain Old Java Object (POJO) 编程模型，Hibernate 将会处于其最佳运行状态。以下所列就是持久化类的主要规则，然而，在这些规则中，没有一条是硬性要求。

- 所有将被持久化的 Java 类都需要一个默认的构造函数。
- 为了使对象能够在 Hibernate 和数据库中容易识别，所有类都需要包含一个 ID。此属性映射到数据库表的主键列。
- 所有将被持久化的属性都应该声明为 private，并具有由 JavaBean 风格定义的 getXXX 和 setXXX 方法。
- Hibernate 的一个重要特征为代理，它取决于该持久化类是处于非 final 的，还是处于一个所有方法都声明为 public 的接口。
- 所有的类是不可扩展或按 EJB 要求实现的一些特殊的类和接口。

POJO 的名称用于强调一个给定的对象是普通的 Java 对象，而不是特殊的对象，尤其不是一个 Enterprise JavaBean。

一个简单的 POJO 的例子

基于以上所述规则，我们能够定义如下 POJO 类：

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
}
```

```
}  
public String getFirstName() {  
    return firstName;  
}  
public void setFirstName( String first_name ) {  
    this.firstName = first_name;  
}  
public String getLastName() {  
    return lastName;  
}  
public void setLastName( String last_name ) {  
    this.lastName = last_name;  
}  
public int getSalary() {  
    return salary;  
}  
public void setSalary( int salary ) {  
    this.salary = salary;  
}  
}
```



映射文件



一个对象/关系型映射一般定义在 XML 文件中。映射文件指示 Hibernate 如何将已经定义的类或类组与数据库中的表对应起来。

尽管有些 Hibernate 用户选择手写 XML 文件，但是有很多工具可以用来给先进的 Hibernate 用户生成映射文件。这样的工具包括 XDoclet, Middlegen 和 AndroMDA。

让我们来考虑我们之前定义的 POJO 类，它的对象将延续到下一部分定义的表中。

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

对于每一个你想要提供持久性的对象都需要一个表与之保持一致。考虑上述对象需要存储和检索到下列 RDBMS 表中：

```
create table EMPLOYEE (
  id INT NOT NULL auto_increment,
  first_name VARCHAR(20) default NULL,
  last_name VARCHAR(20) default NULL,
  salary INT default NULL,
  PRIMARY KEY (id)
);
```

基于这两个实体之上，我们可以定义下列映射文件来指示 Hibernate 如何将已定义的类或类组与数据库表匹配。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

你需要以格式 `<classname>.hbm.xml` 保存映射文件。我们保存映射文件在 `Employee.hbm.xml` 中。让我们来详细地看一下在映射文件中使用的一些标签：

- 映射文件是一个以 `<hibernate-mapping>` 为根元素的 XML 文件，里面包含所有 `<class>` 标签。
- `<class>` 标签是用来定义从一个 Java 类到数据库表的特定映射。Java 的类名使用 **name** 属性来表示，数据库表明用 **table** 属性来表示。
- `<meta>` 标签是一个可选元素，可以被用来修饰类。
- `<id>` 标签将类中独一无二的 ID 属性与数据库表中的主键关联起来。id 元素中的 **name** 属性引用类的性质，**column** 属性引用数据库表的列。**type** 属性保存 Hibernate 映射的类型，这个类型会将 Java 转换成 SQL 数据类型。

- 在 id 元素中的 `<generator>` 标签用来自动生成主键值。设置 generator 标签中的 `class` 属性可以设置 `native` 使 Hibernate 可以使用 `identity`, `sequence` 或 `hilo` 算法根据底层数据库的情况来创建主键。
- `<property>` 标签用来将 Java 类的属性与数据库表的列匹配。标签中 `name` 属性引用的是类的性质, `column` 属性引用的是数据库表的列。`type` 属性保存 Hibernate 映射的类型, 这个类型会将 Java 转换成 SQL 数据类型。

还有一些其它属性和元素可用在映射文件中, 我会在其它讨论 Hibernate 相关的主题中尽可能得涉及更多。



T



映射类型



当你准备一个 Hibernate 映射文件时，我们已经看到你把 Java 数据类型映射到了 RDBMS 数据格式。在映射文件中已经声明被使用的 `types` 不是 Java 数据类型；它们也不是 SQL 数据库类型。这种类型被称为 Hibernate 映射类型，可以从 Java 翻译成 SQL，反之亦然。

在这一章中列举出所有的基础，日期和时间，大型数据对象，和其它内嵌的映射数据类型。

原始类型

映射类型	Java 类型	ANSI SQL 类型
integer	int 或 java.lang.Integer	INTEGER
long	long 或 java.lang.Long	BIGINT
short	short 或 java.lang.Short	SMALLINT
float	float 或 java.lang.Float	FLOAT
double	double 或 java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte 或 java.lang.Byte	TINYINT
boolean	boolean 或 java.lang.Boolean	BIT
yes/no	boolean 或 java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean 或 java.lang.Boolean	CHAR(1) ('T' or 'F')

日期和时间类型

映射类型	Java 类型	ANSI SQL 类型
date	java.util.Date 或 java.sql.Date	DATE
time	java.util.Date 或 java.sql.Time	TIME
timestamp	java.util.Date 或 java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

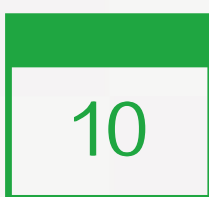
二进制和大型数据对象

映射类型	Java 类型	ANSI SQL 类型
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB

映射类型	Java 类型	ANSI SQL 类型
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

JDK 相关类型

映射类型	Java 类型	ANSI SQL 类型
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR



例子



让我们看一个独立应用程序利用 Hibernate 提供 Java 持久性的例子。我们将通过不同的步骤使用 Hibernate 技术创建 Java 应用程序。

创建 POJO 类

创建应用程序的第一步就是建立 Java 的 POJO 类或者其它类，这取决于即将要存放在数据库中的应用程序。我们可以考虑一下让我们的 `Employee` 类使用 `getXXX` 和 `setXXX` 方法从而使它们变成符合 JavaBeans 的类。

POJO (Plain Old Java Object) 是 Java 的一个对象，这种对象不会扩展或者执行一些特殊的类并且它的接口都是分别在 EJB 框架的要求下的。所有正常的 Java 对象都是 POJO。

当你设计一个存放在 Hibernate 中的类时，最重要的是提供支持 JavaBeans 的代码和在 `Employee` 类中像 `id` 属性一样可以当做索引的属性。

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
}
```

```

}
public int getSalary() {
    return salary;
}
public void setSalary( int salary ) {
    this.salary = salary;
}
}

```

创建数据库表

第二步就是在你的数据库中创建表格。每一个你所愿意提供长期留存的对象都会有一个对应的表。上述的对象需要在下列的 RDBMS 表中存储和被检索到：

```

create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name VARCHAR(20) default NULL,
    salary INT default NULL,
    PRIMARY KEY (id)
);

```

创建映射配置文件

这一步是创建一个映射文件从而指导 Hibernate 如何对数据库的表映射定义的类。

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="salary" column="salary" type="int"/>
    </class>
</hibernate-mapping>

```

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

你需要将映射文档以 `<classname>.hbm.xml` 的格式保存在一个文件中。我们将映射文档保存在 `Employee.hbm.xml` 文件中。下面让我们看看映射文档相关的一些小细节：

- 映射文档是一个 XML 格式的文档，它拥有 `<hibernate-mapping>` 作为根元素，这个元素包含了所有的 `<class>` 元素。
- `<class>` 元素被用来定义从 Java 类到数据库表的特定的映射。Java 类的名称是特定的，它使用的是类元素的 `name` 属性，数据库表的名称也是特定的，它使用的是 `table` 属性。
- `<meta>` 元素是一个可选元素，它可以用来创建类的描述。
- `<id>` 元素向数据库的主要关键字表映射类中的特定的 ID 属性。id 元素的 `name` 属性涉及到了类中的属性同时 `column` 属性涉及到了数据库表中的列。`type` 属性掌握了 hibernate 的映射类型，这种映射类型将会从 Java 转到 SQL 数据类型。
- id 元素中的 `<generator>` 元素是用来自动产生主要关键字的值的。将 generator 元素的 `class` 属性设置成 `native` 从而使 Hibernate 运用 `identity`, `sequence` 或者 `hilo` 算法依靠基础数据库的性能来创建主要关键字。
- `<property>` 元素是用来映射一个 Java 类的属性到数据库的表中的列中。这个元素的 `name` 属性涉及到类中的属性，`column` 属性涉及到数据表中的列。`type` 属性控制 Hibernate 的映射类型，这种映射类型将会从 Java 转到 SQL 数据类型。

映射文档中还有许多其它的属性和元素，在探讨其它的 Hibernate 相关的话题时我将会详细进行讲解。

创建应用程序类

最后，我们将要使用 `main()` 方法创建应用程序类来运行应用程序。我们将用这个程序来保存一些 Employee 的记录，然后我们将在这些记录上应用 CRUD 操作。

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
```

```

private static SessionFactory factory;
public static void main(String[] args) {
    try{
        factory = new Configuration().configure().buildSessionFactory();
    }catch (Throwable ex) {
        System.err.println("Failed to create sessionFactory object." + ex);
        throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();

    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 10000);

    /* List down all the employees */
    ME.listEmployees();

    /* Update employee's records */
    ME.updateEmployee(empID1, 5000);

    /* Delete an employee from the database */
    ME.deleteEmployee(empID2);

    /* List down new list of the employees */
    ME.listEmployees();
}

/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee(fname, lname, salary);
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}
}

```



```

/* Method to READ all the employees */
public void listEmployees() {
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator =
            employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print(" Last Name: " + employee.getLastName());
            System.out.println(" Salary: " + employee.getSalary());
        }
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
}

/* Method to UPDATE salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        employee.setSalary( salary );
        session.update(employee);
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
}

/* Method to DELETE an employee from the records */
public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession();
    Transaction tx = null;
    try {

```

```

    tx = session.beginTransaction();
    Employee employee =
        (Employee)session.get(Employee.class, EmployeeID);
    session.delete(employee);
    tx.commit();
} catch (HibernateException e) {
    if (tx!=null) tx.rollback();
    e.printStackTrace();
} finally {
    session.close();
}
}
}

```

编译和执行

下面是编译和运行上述提到的应用程序的步骤。在编译和执行应用程序之前确保你已经设置好了 PATH 和 CLASSPATH。

- 创建设置章节中所讲的 hibernate.cfg.xml 配置文件。
- 创建上文所述的 Employee.hbm.xml 映射文件。
- 创建上文所述的 Employee.java 源文件并且进行编译。
- 创建上文所述的 ManageEmployee.java 源文件并且进行编译。
- 执行二进制的 ManageEmployee 来运行程序。

你将会得到如下结果，记录将会在 EMPLOYEE 表中建立。

```

$java ManageEmployee
.....VARIOUS LOG MESSAGES WILL DISPLAY HERE.....

First Name: Zara Last Name: Ali Salary: 1000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 10000
First Name: Zara Last Name: Ali Salary: 5000
First Name: John Last Name: Paul Salary: 10000

```

如果你检查你的 EMPLOYEE 表，它将会有如下记录：

```

mysql> select * from EMPLOYEE;
+----+-----+-----+-----+
| id | first_name | last_name | salary |
+----+-----+-----+-----+

```

```
| 29 | Zara   | Ali   | 5000 |  
| 31 | John   | Paul  | 10000 |  
+-----+-----+-----+-----+  
2 rows in set (0.00 sec  
  
mysql>
```



O/R 映射



目前为止我们已经通过应用 Hibernate 见识过十分基础的 O/R 映射了，但是还有三个更加重要的有关映射的话题需要我们更详细的探讨。这三个话题是集合的映射，实体类之间的关联映射以及组件映射。

集合映射

如果一个实例或者类中有特定变量的值的集合，那么我们可以应用 Java 中的任何的可用的接口来映射这些值。Hibernate 可以保存 `java.util.Map`, `java.util.Set`, `java.util.SortedMap`, `java.util.SortedSet`, `java.util.List` 和其它持续的实例或者值的任何数组的实例。

集合类型	映射和描述
<code>java.util.Set</code>	它和 <code><set></code> 元素匹配并且用 <code>java.util.HashSet</code> 初始化。
<code>java.util.SortedSet</code>	它和 <code><set></code> 元素匹配并且用 <code>java.util.TreeSet</code> 初始化。 <code>sort</code> 属性可以设置成比较器或者自然排序。
<code>java.util.List</code>	它和 <code><list></code> 元素匹配并且用 <code>java.util.ArrayList</code> 初始化。
<code>java.util.Collection</code>	它和 <code><bag></code> 或者 <code><ibag></code> 元素匹配以及用 <code>java.util.ArrayList</code> 初始化。
<code>java.util.Map</code>	它和 <code><map></code> 元素匹配并且用 <code>java.util.HashMap</code> 初始化。
<code>java.util.SortedMap</code>	它和 <code><map></code> 元素匹配并且用 <code>java.util.TreeMap</code> 初始化。 <code>sort</code> 属性可以设置成比较器或者自然排序。

对于 Java 的原始数值 Hibernate 采用 `<primitive-array>` 支持数组，对于 Java 的其它数值 Hibernate 采用 `<array>` 支持数组。然而它们很少被应用，因此我也就不在本指导中讨论它们。

如果你想要映射一个用户定义的集合接口而这个接口不是 Hibernate 直接支持的话，那么你需要告诉 Hibernate 你定义的这个集合的语法，这个很难操作而且不推荐使用。

关联映射

实体类之间的关联映射以及表之间的关系是 ORM 的灵魂之处。对象间的关系的子集可以用下列四种方式解释。关联映射可以是单向的也可以是双向的。

映射类型	描述
Many-to-One	使用 Hibernate 映射多对一关系
One-to-One	使用 Hibernate 映射一对一关系
One-to-Many	使用 Hibernate 映射一对多关系
Many-to-Many	使用 Hibernate 映射多对多关系

组件映射

作为变量的一员实体类很可能和其它类具有相关关系。如果引用的类没有自己的生命周期并且完全依靠于拥有它的那个实体类的生命周期的话，那么这个引用类因此就可以叫做组件类。

组件集合的映射很可能和正常集合的映射相似，只会有很少的设置上的不同。我们可以在例子中看看这两种映射。

映射类型	描述
Component Mappings	类的映射对于作为变量的一员的另外的类具有参考作用。



T



12

注释



到现在为止，你已经看到 Hibernate 如何使用 XML 映射文件来完成从 POJO 到数据库表的数据转换的，反之亦然。Hibernate 注释是无需使用 XML 文件来定义映射的最新方法。你可以额外使用注释或直接代替 XML 映射元数据。

Hibernate 注释是一种强大的来给对象和关系映射表提供元数据的方法。所有的元数据被添加到 POJO java 文件代码中，这有利于用户在开发时更好的理解表的结构和 POJO。

如果你想让你的应用程序移植到其它 EJB 3 的 ORM 应用程序中,您必须使用注释来表示映射信息，但是如果想要得到更大的灵活性,那么你应该使用基于 XML 的映射。

Hibernate 注释的环境设置

首先你必须确定你使用的是 JDK 5.0，否则你需要升级你的 JDK 至 JDK 5.0，来使你的主机能够支持注释。

其次，你需要安装 Hibernate 3.x 注释包，可以从 sourceforge 行下载：[（下载 Hibernate 注释）](http://sourceforge.net/projects/hibernate/files/hibernate-annotations/) (<http://sourceforge.net/projects/hibernate/files/hibernate-annotations/>) 并且从 Hibernate 注释发布中拷贝 `hibernate-annotations.jar`, `lib/hibernate-comons-annotations.jar` 和 `lib/ejb3-persistence.jar` 到你的 CLASSPATH。

注释类示例

正如我上面所提到的，所有的元数据被添加到 POJO java 文件代码中，这有利于用户在开发时更好的理解表的结构和 POJO。

下面我们将使用 EMPLOYEE 表来存储对象:

```
create table EMPLOYEE (
  id INT NOT NULL auto_increment,
  first_name VARCHAR(20) default NULL,
  last_name VARCHAR(20) default NULL,
  salary INT default NULL,
  PRIMARY KEY (id)
);
```

以下是用带有注释的 Employee 类来映射使用定义好的 Employee 表的对象:

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
```



```

@Id @GeneratedValue
@Column(name = "id")
private int id;

@Column(name = "first_name")
private String firstName;

@Column(name = "last_name")
private String lastName;

@Column(name = "salary")
private int salary;

public Employee() {}
public int getId() {
    return id;
}
public void setId( int id ) {
    this.id = id;
}
public String getFirstName() {
    return firstName;
}
public void setFirstName( String first_name ) {
    this.firstName = first_name;
}
public String getLastName() {
    return lastName;
}
public void setLastName( String last_name ) {
    this.lastName = last_name;
}
public int getSalary() {
    return salary;
}
public void setSalary( int salary ) {
    this.salary = salary;
}
}

```

Hibernate 检测到 @Id 注释字段并且认定它应该在运行时通过字段直接访问一个对象上的属性。如果你将 @Id 注释放在 getId() 方法中，你可以通过默认的 getter 和 setter 方法来访问属性。因此，所有其它注释也放在字段或是 getter 方法中，决定于选择的策略。下一节将解释上面的类中使用的注释。

@Entity 注释

EJB 3 标准的注释包含在 `javax.persistence` 包，所以我们第一步需要导入这个包。第二步我们对 `Employee` 类使用 `@Entity` 注释，标志着这个类为一个实体 bean，所以它必须含有一个没有参数的构造函数并且在可保护范围是可见的。

@Table 注释

`@table` 注释允许您明确表的详细信息保证实体在数据库中持续存在。

`@table` 注释提供了四个属性，允许您覆盖的表的名称，目录及其模式,在表中可以对列制定独特的约束。现在我们使用的是表名为 `EMPLOYEE`。

@Id 和 @GeneratedValue 注释

每一个实体 bean 都有一个主键，你在类中可以用 `@Id` 来进行注释。主键可以是一个字段或者是多个字段的组合，这取决于你的表的结构。

默认情况下，`@Id` 注释将自动确定最合适的主键生成策略，但是您可以通过使用 `@GeneratedValue` 注释来覆盖掉它。`strategy` 和 `generator` 这两个参数我不打算在这里讨论，所以我们只使用默认键生成策略。让 `Hibernate` 确定使用哪些生成器类型来使代码移植于不同的数据库之间。

@Column Annotation

`@Column` 注释用于指定某一列与某一个字段或是属性映射的细节信息。您可以使用下列注释的最常用的属性：

- `name` 属性允许显式地指定列的名称。
- `length` 属性为用于映射一个值，特别为一个字符串值的列的大小。
- `nullable` 属性允许当生成模式时，一个列可以被标记为非空。
- `unique` 属性允许列中只能含有唯一的内容

创建应用类

最后,我们将创建应用程序类,并使用 main() 方法来运行应用程序。我们将使用此应用程序来保存一些员工的记录,然后我们对这些记录进行 CRUD 操作。

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new AnnotationConfiguration().
                configure().
                //addPackage("com.xyz") //add package if used.
                addAnnotatedClass(Employee.class).
                buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 10000);

        /* List down all the employees */
        ME.listEmployees();

        /* Update employee's records */
        ME.updateEmployee(empID1, 5000);

        /* Delete an employee from the database */
    }
}
```

```

ME.deleteEmployee(empID2);

/* List down new list of the employees */
ME.listEmployees();
}
/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee();
        employee.setFirstName(fname);
        employee.setLastName(lname);
        employee.setSalary(salary);
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}
/* Method to READ all the employees */
public void listEmployees() {
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator =
            employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print(" Last Name: " + employee.getLastName());
            System.out.println(" Salary: " + employee.getSalary());
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {

```

```

        session.close();
    }
}

/* Method to UPDATE salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        employee.setSalary( salary );
        session.update(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}

/* Method to DELETE an employee from the records */
public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        session.delete(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
}

```

数据库配置

现在，让我们创建 `hibernate.cfg.xml` 配置文件来定义数据库相关参数。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>

    <!-- Assume students is the database name -->
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
    </property>
    <property name="hibernate.connection.username">
      root
    </property>
    <property name="hibernate.connection.password">
      cohondob
    </property>

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

编译和执行

这里是编译并运行以上提到的应用程序的步骤。再继续编译和运行之前需要确保你正确设置路径和类路径。

- 从目录中删除 Employee.hbm.xml 映射文件。
- 创建上述 Employee.java 源文件并编译。
- 创建上述 ManageEmployee.java 源文件并编译。
- 执行 ManageEmployee 二进制程序。

你将得到如下结果，并且会在 EMPLOYEE 表中记录。

```
$java ManageEmployee
.....VARIOUS LOG MESSAGES WILL DISPLAY HERE.....

First Name: Zara Last Name: Ali Salary: 1000
```

First Name: Daisy Last Name: Das Salary: 5000
 First Name: John Last Name: Paul Salary: 10000
 First Name: Zara Last Name: Ali Salary: 5000
 First Name: John Last Name: Paul Salary: 10000

如果你查看 EMPLOYEE 表，它将有如下记录：

```
mysql> select * from EMPLOYEE;
+----+-----+-----+-----+
| id | first_name | last_name | salary |
+----+-----+-----+-----+
| 29 | Zara      | Ali      | 5000   |
| 31 | John      | Paul     | 10000  |
+----+-----+-----+-----+
2 rows in set (0.00 sec

mysql>
```



查询语言



Hibernate 查询语言（HQL）是一种面向对象的查询语言，类似于 SQL，但不是去对表和列进行操作，而是面向对象和它们的属性。HQL 查询被 Hibernate 翻译为传统的 SQL 查询从而对数据库进行操作。

尽管你能直接使用本地 SQL 语句，但我还是建议你尽可能的使用 HQL 语句，以避免数据库关于可移植性的麻烦，并且体现了 Hibernate 的 SQL 生成和缓存策略。

在 HQL 中一些关键字比如 SELECT，FROM 和 WHERE 等，是不区分大小写的，但是一些属性比如表名和列名是区分大小写的。

FROM 语句

如果你想要在存储中加载一个完整并持久的对象,你将使用 FROM 语句。以下是 FROM 语句的一些简单的语法:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

如果你需要在 HQL 中完全限定类名，只需要指定包和类名，如下：

```
String hql = "FROM com.hibernatebook.criteria.Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

AS 语句

在 HQL 中 AS 语句能够用来给你的类分配别名，尤其是在长查询的情况下。例如，我们之前的例子，可以用如下方式展示：

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

关键字 AS 是可选择的并且你也可以在类名后直接指定一个别名，如下：

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

SELECT 语句

SELECT 语句比 from 语句提供了更多的对结果集的控制。如果你只想得到对象的几个属性而不是整个对象你需要使用 SELECT 语句。下面是一个 SELECT 语句的简单语法示例，这个例子是为了得到 Employee 对象的 first_name 字段：

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

值得注意的是 Employee.firstName 是 Employee 对象的属性，而不是一个 EMPLOYEE 表的字段。

WHERE 语句

如果你想要精确地从数据库存储中返回特定对象，你需要使用 WHERE 语句。下面是 WHERE 语句的简单语法例子：

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

ORDER BY 语句

为了给 HSQ 查询结果进行排序，你将需要使用 ORDER BY 语句。你能利用任意一个属性给你的结果进行排序，包括升序或降序排序。下面是一个使用 ORDER BY 语句的简单示例：

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

如果你想要给多个属性进行排序，你只需要在 ORDER BY 语句后面添加你要进行排序的属性即可，并且用逗号进行分割：

```
String hql = "FROM Employee E WHERE E.id > 10 " +
    "ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

GROUP BY 语句

这一语句允许 Hibernate 将信息从数据库中提取出来，并且基于某种属性的值将信息进行编组，通常而言，该语句会使用得到的结果来包含一个聚合值。下面是一个简单的使用 GROUP BY 语句的语法：

```
String hql = "SELECT SUM(E.salary), E.firstName FROM Employee E " +
    "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

使用命名参数

Hibernate 的 HQL 查询功能支持命名参数。这使得 HQL 查询功能既能接受来自用户的简单输入，又无需防御 SQL 注入攻击。下面是使用命名参数的简单的语法：

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
List results = query.list();
```

UPDATE 语句

HQL Hibernate 3 较 HQL Hibernate 2，新增了批量更新功能和选择性删除工作的功能。查询接口包含一个 `executeUpdate()` 方法，可以执行 HQL 的 UPDATE 或 DELETE 语句。

UPDATE 语句能够更新一个或多个对象的一个或多个属性。下面是使用 UPDATE 语句的简单的语法：

```
String hql = "UPDATE Employee set salary = :salary " +
    "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE 语句

DELETE 语句可以用来删除一个或多个对象。以下是使用 DELETE 语句的简单语法：

```
String hql = "DELETE FROM Employee " +
    "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT 语句

HQL 只有当记录从一个对象插入到另一个对象时才支持 INSERT INTO 语句。下面是使用 INSERT INTO 语句的简单的语法:

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)" +
    "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

聚合方法

HQL 类似于 SQL，支持一系列的聚合方法，它们以同样的方式在 HQL 和 SQL 中工作，以下列出了几种可用方法:

S.N.	方法	描述
1	avg(property name)	属性的平均值
2	count(property name or *)	属性在结果中出现的次数
3	max(property name)	属性值的最大值
4	min(property name)	属性值的最小值
5	sum(property name)	属性值的总和

distinct 关键字表示只计算行集中的唯一值。下面的查询只计算唯一的值:

```
String hql = "SELECT count(distinct E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

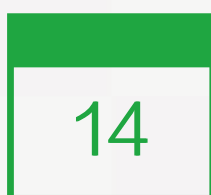
使用分页查询

以下为两种分页查询界面的方法:

S.N.	方法&描述
1	Query setFirstResult(int startPosition) 该方法以一个整数表示结果中的第一行,从 0 行开始。
2	Query setMaxResults(int maxResult) 这个方法告诉 Hibernate 来检索固定数量,即 maxResults 个对象。

使用以上两种方法,我们可以在我们的 web 或 Swing 应用程序中构造一个分页组件。下面是示例,您可以扩展到每次取 10 行:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```



标准查询



Hibernate 提供了操纵对象和相应的 RDBMS 表中可用的数据的替代方法。一种方法是标准的 API，它允许你建立一个标准的可编程查询对象来应用过滤规则和逻辑条件。

Hibernate `Session` 接口提供了 `createCriteria()` 方法，可用于创建一个 `Criteria` 对象，使当您的应用程序执行一个标准查询时返回一个持久化对象的类的实例。

以下是一个最简单的标准查询的例子，它只是简单地返回对应于员工类的每个对象：

```
Criteria cr = session.createCriteria(Employee.class);
List results = cr.list();
```

对标准的限制

你可以使用 `Criteria` 对象可用的 `add()` 方法去添加一个标准查询的限制。

以下是一个示例，它实现了添加一个限制，令返回工资等于 2000 的记录：

```
Criteria cr = session.createCriteria(Employee.class);
cr.add(Restrictions.eq("salary", 2000));
List results = cr.list();
```

以下是几个例子，涵盖了不同的情况，可按要求进行使用：

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having firstName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
```

```
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

你可以模仿以下示例，使用逻辑表达式创建 AND 或 OR 的条件组合：

```
Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt("salary", 2000);
Criterion name = Restrictions.ilike("firstName", "zara%");

// To get records matching with OR conditions
LogicalExpression orExp = Restrictions.or(salary, name);
cr.add( orExp );

// To get records matching with AND conditions
LogicalExpression andExp = Restrictions.and(salary, name);
cr.add( andExp );

List results = cr.list();
```

另外，上述所有的条件都可按之前的教程中解释的那样与 HQL 直接使用。

分页使用标准

这里有两种分页标准接口方法：

序号	方法描述
1	public Criteria setFirstResult(int firstResult)，这种方法需要一个代表你的结果集的第一行的整数，以第 0 行为开始。
2	public Criteria setMaxResults(int maxResults)，这个方法设置了 Hibernate 检索对象的 maxResults。

利用上述两种方法结合在一起，我们可以在我们的 Web 或 Swing 应用程序构建一个分页组件。以下是一个例子，利用它你可以一次取出 10 行：

```
Criteria cr = session.createCriteria(Employee.class);
cr.setFirstResult(1);
```



```
cr.setMaxResults(10);
List results = cr.list();
```

排序结果

标准 API 提供了 `org.hibernate.criterion.order` 类可以根据你的一个对象的属性把你的排序结果集按升序或降序排列。这个例子演示了如何使用 `Order` 类对结果集进行排序：

```
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order
crit.addOrder(Order.desc("salary"));

// To sort records in ascending order
crit.addOrder(Order.asc("salary"));

List results = cr.list();
```

预测与聚合

标准 API 提供了 `org.hibernate.criterion.projections` 类可得到各属性值的平均值，最大值或最小值。`Projections` 类与 `Restrictions` 类相似，均提供了几个获取预测实例的静态工厂方法。

以下是几个例子，涵盖了不同的情况，可按要求进行使用：

```
Criteria cr = session.createCriteria(Employee.class);

// To get total row count.
cr.setProjection(Projections.rowCount());

// To get average of a property.
cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.
cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.
cr.setProjection(Projections.max("salary"));

// To get minimum of a property.
```

```
cr.setProjection(Projections.min("salary"));

// To get sum of a property.
cr.setProjection(Projections.sum("salary"));
```

标准查询示例

考虑下面的 POJO 类：

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

```
}
}
```

让我们创建以下员工表来存储 Employee 对象：

```
create table EMPLOYEE (
  id INT NOT NULL auto_increment,
  first_name VARCHAR(20) default NULL,
  last_name VARCHAR(20) default NULL,
  salary INT default NULL,
  PRIMARY KEY (id)
);
```

以下是映射文件：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

最后，我们将用 main() 方法创建应用程序类来运行应用程序，我们将使用 Criteria 查询：

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.Criteria;
import org.hibernate.criterion.Restrictions;
```

```

import org.hibernate.criterion.Projections;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 5000);
        Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);

        /* List down all the employees */
        ME.listEmployees();

        /* Print Total employee's count */
        ME.countEmployee();

        /* Print Toatl salary */
        ME.totalSalary();
    }
    /* Method to CREATE an employee in the database */
    public Integer addEmployee(String fname, String lname, int salary){
        Session session = factory.openSession();
        Transaction tx = null;
        Integer employeeID = null;
        try{
            tx = session.beginTransaction();
            Employee employee = new Employee(fname, lname, salary);
            employeeID = (Integer) session.save(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
    }
}

```

```

    return employeeID;
}

/* Method to READ all the employees having salary more than 2000 */
public void listEmployees() {
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Criteria cr = session.createCriteria(Employee.class);
        // Add restriction.
        cr.add(Restrictions.gt("salary", 2000));
        List employees = cr.list();

        for (Iterator iterator =
            employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print(" Last Name: " + employee.getLastName());
            System.out.println(" Salary: " + employee.getSalary());
        }
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
}

/* Method to print total number of records */
public void countEmployee(){
    Session session = factory.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        Criteria cr = session.createCriteria(Employee.class);

        // To get total row count.
        cr.setProjection(Projections.rowCount());
        List rowCount = cr.list();

        System.out.println("Total Count: " + rowCount.get(0) );
        tx.commit();
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
    }
}

```

```

        e.printStackTrace();
    }finally {
        session.close();
    }
}
/* Method to print sum of salaries */
public void totalSalary(){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Criteria cr = session.createCriteria(Employee.class);

        // To get total salary.
        cr.setProjection(Projections.sum("salary"));
        List totalSalary = cr.list();

        System.out.println("Total Salary: " + totalSalary.get(0) );
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
}

```

编译和执行

这是编译并运行上述应用程序的步骤。确保你有适当的 PATH 和 CLASSPATH，然后执行编译程序。

- 按照在配置一章讲述的方法创建 hibernate.cfg.xml 配置文件。
- 如上述所示创建 employee.hbm.xml 映射文件。
- 如上述所示创建 employee.java 源文件并编译。
- 如上述所示创建 manageemployee.java 源文件并编译。
- 执行 manageemployee 二进制代码运行程序。

你会得到下面的结果，并且记录将会在 EMPLOYEE 表创建。

```

$java ManageEmployee
.....VARIOUS LOG MESSAGES WILL DISPLAY HERE.....

```

First Name: Daisy Last Name: Das Salary: 5000
 First Name: John Last Name: Paul Salary: 5000
 First Name: Mohd Last Name: Yasee Salary: 3000
 Total Count: 4
 Total Salary: 15000

如果你检查你的 EMPLOYEE 表，它应该有如下记录：

```
mysql> select * from EMPLOYEE;
+----+-----+-----+-----+
| id | first_name | last_name | salary |
+----+-----+-----+-----+
| 14 | Zara      | Ali      | 2000   |
| 15 | Daisy     | Das      | 5000   |
| 16 | John      | Paul     | 5000   |
| 17 | Mohd      | Yasee    | 3000   |
+----+-----+-----+-----+
4 rows in set (0.00 sec)
mysql>
```



原生 SQL



如果你想使用数据库特定的功能如查询提示或 Oracle 中的 CONNECT 关键字的话，你可以使用原生 SQL 数据库来表达查询。Hibernate 3.x 允许您为所有的创建，更新，删除，和加载操作指定手写 SQL，包括存储过程。您的应用程序会在会话界面用 `createSQLQuery()` 方法创建一个原生 SQL 查询：

```
public SQLQuery createSQLQuery(String sqlString) throws HibernateException
```

当你通过一个包含 SQL 查询的 `createsqlquery()` 方法的字符串时，你可以将 SQL 的结果与现有的 Hibernate 实体，一个连接，或一个标量结果分别使用 `addEntity()`, `addJoin()`, 和 `addScalar()` 方法进行关联。

标量查询

最基本的 SQL 查询是从一个或多个列表中获取一个标量（值）列表。以下是使用原生 SQL 进行获取标量的值的语法：

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List results = query.list();
```

实体查询

以上的查询都是关于返回标量值的查询，只是基础性地返回结果集中的“原始”值。以下是从原生 SQL 查询中通过 `addEntity()` 方法获取实体对象整体的语法：

```
String sql = "SELECT * FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List results = query.list();
```

指定 SQL 查询

以下是从原生 SQL 查询中通过 `addEntity()` 方法和使用指定 SQL 查询来获取实体对象整体的语法：

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list();
```

原生 SQL 的例子

考虑下面的 POJO 类：

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

让我们创建以下 EMPLOYEE 表来存储 Employee 对象：

```
create table EMPLOYEE (
  id INT NOT NULL auto_increment,
  first_name VARCHAR(20) default NULL,
  last_name VARCHAR(20) default NULL,
  salary INT default NULL,
  PRIMARY KEY (id)
);
```

以下是映射文件：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

最后，我们将用 main() 方法创建应用程序类来运行应用程序，我们将使用原生 SQL 查询：

```
import java.util.*;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.SQLQuery;
import org.hibernate.Criteria;
import org.hibernate.Hibernate;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
  private static SessionFactory factory;
  public static void main(String[] args) {
    try{
```

```

        factory = new Configuration().configure().buildSessionFactory();
    }catch (Throwable ex) {
        System.err.println("Failed to create sessionFactory object." + ex);
        throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();

    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 5000);
    Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);

    /* List down employees and their salary using Scalar Query */
    ME.listEmployeesScalar();

    /* List down complete employees information using Entity Query */
    ME.listEmployeesEntity();
}

/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee(fname, lname, salary);
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}

/* Method to READ all the employees using Scalar Query */
public void listEmployeesScalar( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        String sql = "SELECT first_name, salary FROM EMPLOYEE";

```

```

SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List data = query.list();

for(Object object : data)
{
    Map row = (Map)object;
    System.out.print("First Name: " + row.get("first_name"));
    System.out.println(", Salary: " + row.get("salary"));
}
tx.commit();
}catch (HibernateException e) {
    if (tx!=null) tx.rollback();
    e.printStackTrace();
}finally {
    session.close();
}
}

/* Method to READ all the employees using Entity Query */
public void listEmployeesEntity( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        String sql = "SELECT * FROM EMPLOYEE";
        SQLQuery query = session.createSQLQuery(sql);
        query.addEntity(Employee.class);
        List employees = query.list();

        for (Iterator iterator =
            employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print(" Last Name: " + employee.getLastName());
            System.out.println(" Salary: " + employee.getSalary());
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}

```

```
}
}
```

编译和执行

这是编译并运行上述应用程序的步骤。确保你有适当的 PATH 和 CLASSPATH，然后执行编译程序。

- 按照在配置一章讲述的方法创建 hibernate.cfg.xml 配置文件。
- 如上述所示创建 employee.hbm.xml 映射文件。
- 如上述所示创建 employee.java 源文件并编译。
- 如上述所示创建 manageemployee.java 源文件并编译。
- 执行 manageemployee 二进制代码运行程序。

你会得到下面的结果，并且记录将会在 EMPLOYEE 表创建。

```
$java ManageEmployee
.....VARIOUS LOG MESSAGES WILL DISPLAY HERE.....
```

```
First Name: Zara, Salary: 2000
First Name: Daisy, Salary: 5000
First Name: John, Salary: 5000
First Name: Mohd, Salary: 3000
First Name: Zara Last Name: Ali Salary: 2000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 5000
First Name: Mohd Last Name: Yasee Salary: 3000
```

如果你检查你的 EMPLOYEE 表，它应该有如下记录：

```
mysql> select * from EMPLOYEE;
+----+-----+-----+-----+
| id | first_name | last_name | salary |
+----+-----+-----+-----+
| 26 | Zara      | Ali      | 2000   |
| 27 | Daisy     | Das      | 5000   |
| 28 | John      | Paul     | 5000   |
| 29 | Mohd      | Yasee    | 3000   |
+----+-----+-----+-----+
4 rows in set (0.00 sec)
mysql>
```



T



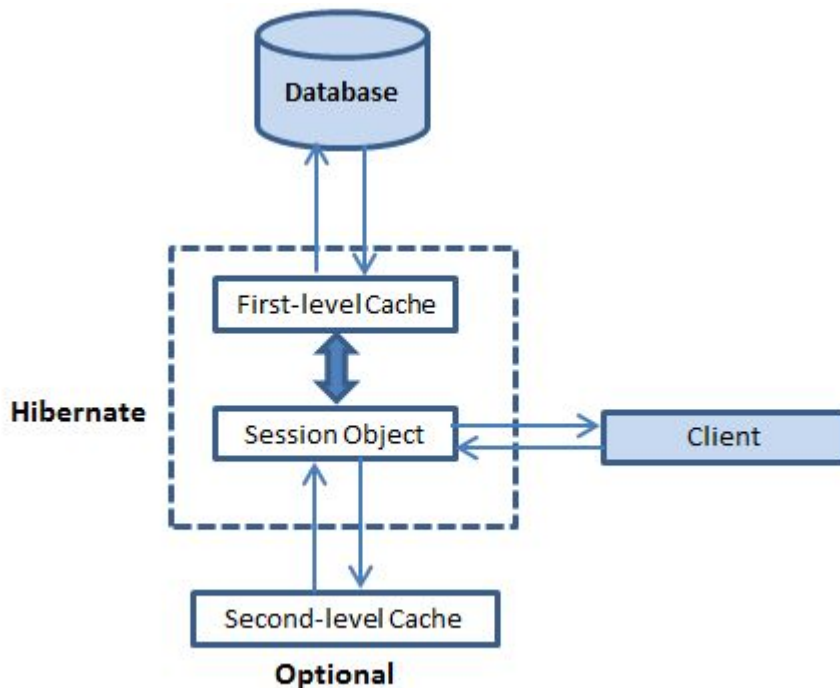
16

缓存



缓存是关于应用程序性能的优化，降低了应用程序对物理数据源访问的频次，从而提高应用程序的运行性能。

缓存对 Hibernate 来说也是重要的，它使用了如下解释的多级缓存方案：



图片 16.1 image

一级缓存

第一级缓存是 Session 缓存并且是一种强制性的缓存，所有的要求都必须通过它。Session 对象在它自己的权利之下，在将它提交给数据库之前保存一个对象。

如果你对一个对象发出多个更新，Hibernate 会尝试尽可能长地延迟更新来减少发出的 SQL 更新语句的数目。如果你关闭 session, 所有缓存的对象丢失，或是存留，或是在数据库中被更新。

二级缓存

第二级缓存是一种可选择的缓存并且第一级缓存在任何想要在第二级缓存中找到一个对象前将总是被询问。第二级缓存可以在每一个类和每一个集合的基础上被安装，并且它主要负责跨会话缓存对象。

任何第三方缓存可以和 Hibernate 一起使用。org.hibernate.cache.CacheProvider 接口被提供，它必须实现来给 Hibernate 提供一个缓存实现的解决方法。

查询层次缓存

Hibernate 也实现了一个和第二级缓存密切集成的查询结果集缓存。

这是一个可选择的特点并且需要两个额外的物理缓存区域，它们保存着缓存的查询结果和表单上一次更新时的时间戳。这仅对以同一个参数频繁运行的查询来说是有用的。

第二级缓存

Hibernate 使用默认的一级缓存并且你不用使用一级缓存。让我们直接看向可选的二级缓存。不是所有的类从缓存中获益，所以能关闭二级缓存是重要的。

Hibernate 的二级缓存通过两步设置。第一，你必须决定好使用哪个并发策略。之后，你使用缓存提供程序来配置缓存到期时间和物理缓存属性。

并发策略

一个并发策略是一个中介，它负责保存缓存中的数据项和从缓存中检索它们。如果你将使用一个二级缓存，你必须决定，对于每一个持久类和集合，使用哪一个并发策略。

- **Transactional**: 为主读数据使用这个策略，在一次更新的罕见状况下并发事务阻止过期数据是关键。
- **Read-write**: 为主读数据再一次使用这个策略，在一次更新的罕见状况下并发事务阻止过期数据是关键。
- **Nonstrict-read-write**: 这个策略不保证缓存和数据库之间的一致性。如果数据几乎不改变并且过期数据不是很重要，使用这个策略。
- **Read-only**: 一个适合永不改变数据的并发策略。只为参考数据使用它。

如果我们将为我们的 `Employee` 类使用二级缓存，让我们使用 `read-write` 策略来添加需要告诉 Hibernate 来缓存 `Employee` 实例的映射元素。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
```

```

    This class contains the employee detail.
</meta>
<cache usage="read-write"/>
<id name="id" type="int" column="id">
    <generator class="native"/>
</id>
<property name="firstName" column="first_name" type="string"/>
<property name="lastName" column="last_name" type="string"/>
<property name="salary" column="salary" type="int"/>
</class>
</hibernate-mapping>

```

usage="read-write" 参数告诉 Hibernate 为定义的缓存使用 read-write 并发策略。

缓存提供者

在考虑你将为你的缓存候选类所使用的并发策略后你的下一步是挑选一个缓存提供者。Hibernate 让你为整个应用程序选择一个单独的缓存提供者。

S.N.	缓存名	描述
1	EHCac he	它能在内存或硬盘上缓存并且集群缓存，而且它支持可选的 Hibernate 查询结果缓存。
2	OSCac he	支持在一个单独的 JVM 中缓存到内存和硬盘，同时有丰富的过期策略和查询缓存支持。
3	warmC ache	一个基于 JGroups 的聚集缓存。它使用集群失效但是不支持 Hibernate 查询缓存。
4	JBoss Cache	一个也基于 JGroups 多播库的完全事务性的复制集群缓存。它支持复制或者失效，同步或异步通信，乐观和悲观锁定。Hibernate 查询缓存被支持。

每一个缓存提供者都不和每个并发策略兼容。以下的兼容性矩阵将帮助你选择一个合适的组合。

策略/提供者	Read-only	Nonstrictread-write	Read-write	Transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBoss Cache	X			X

你将在 hibernate.cfg.xml 配置文件中指定一个缓存提供者。我们选择 EHCache 作为我们的二级缓存提供者：

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

```

```

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>

    <!-- Assume students is the database name -->
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
    </property>
    <property name="hibernate.connection.username">
      root
    </property>
    <property name="hibernate.connection.password">
      root123
    </property>
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.EhCacheProvider
    </property>

    <!-- List of XML mapping files -->
    <mapping resource="Employee.hbm.xml"/>

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

```

现在，你需要指定缓存区域的属性。EhCache 有它自己的配置文件，**ehcache.xml**，它应该在应用程序的 CLASSPATH 中。Employee 类的 ehcache.xml 缓存配置像如下这样：

```

<diskStore path="java.io.tmpdir"/>
<defaultCache
maxElementsInMemory="1000"
eternal="false"
timeToIdleSeconds="120"
timeToLiveSeconds="120"
overflowToDisk="true"
/>

<cache name="Employee"
maxElementsInMemory="500"
eternal="true"
timeToIdleSeconds="0"

```

```
timeToLiveSeconds="0"
overflowToDisk="false"
/>
```

就是这样，现在我们有 Employee 类的二级缓存并且 Hibernate 现在能命中缓存无论是你导航到 Employee 时或是当你通过标识符上传 Employee 时。

你应该为每个类分析你所有的类并选择合适的缓存策略。有时候，二级缓存可能使应用程序的表现下降。所以首先不允许缓存用基准程序测试你的应用程序，然后开启合适的缓存，之后检测表现是推荐的。如果缓存不提升系统表现那么支持任何类型的缓存都是没有意义的。

查询层次缓存

为了使用查询缓存，你必须首先使用配置文件中的 `hibernate.cache.use_query_cache="true"` 属性激活它。通过设置这个属性为真，你使得 Hibernate 创建内存中必要的缓存来保存查询和标识符集。

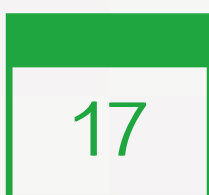
然后，为了使用查询缓存，你使用 Query 类的 `setCacheable(Boolean)` 方法。例如：

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
List users = query.list();
SessionFactory.closeSession();
```

Hibernate 通过缓存区域的概念也支持非常细粒度的缓存支持。一个缓存区域是被给予名字的缓存部分。

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
query.setCacheRegion("employee");
List users = query.list();
SessionFactory.closeSession();
```

这段代码使用方法来告诉 Hibernate 存储和寻找缓存 employee 区域的查询。



批处理



考虑一种情况，你需要使用 Hibernate 将大量的数据上传到你的数据库中。以下是使用 Hibernate 来达到这个的代码片段：

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee employee = new Employee(.....);
    session.save(employee);
}
tx.commit();
session.close();
```

因为默认下，Hibernate 将缓存所有的在会话层缓存中的持久的对象并且最终你的应用程序将和 `OutOfMemoryException` 在第 50000 行的某处相遇。你可以解决这个问题，如果你在 Hibernate 使用批处理。

为了使用批处理这个特性，首先设置 `hibernate.jdbc.batch_size` 作为批处理的尺寸，取一个依赖于对象尺寸的值 20 或 50。这将告诉 hibernate 容器每 X 行为一批插入。为了在你的代码中实现这个我们将需要像以下这样做一些修改：

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee employee = new Employee(.....);
    session.save(employee);
    if( i % 50 == 0 ) { // Same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}
tx.commit();
session.close();
```

上面的代码将使 INSERT 操作良好运行，但是如果你愿意进行 UPDATE 操作那么你可以使用以下代码达到这一点：

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

ScrollableResults employeeCursor = session.createQuery("FROM EMPLOYEE")
    .scroll();

int count = 0;

while ( employeeCursor.next() ) {
```

```

Employee employee = (Employee) employeeCursor.get(0);
employee.updateEmployee();
session.update(employee);
if ( ++count % 50 == 0 ) {
    session.flush();
    session.clear();
}
}
tx.commit();
session.close();

```

批处理样例

让我们通过添加 `hibernate.jdbc.batch_size` 属性来修改配置文件：

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>

        <!-- Assume students is the database name -->
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/test
        </property>
        <property name="hibernate.connection.username">
            root
        </property>
        <property name="hibernate.connection.password">
            root123
        </property>
        <property name="hibernate.jdbc.batch_size">
            50
        </property>

        <!-- List of XML mapping files -->
        <mapping resource="Employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>

```

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

考虑以下 POJO Employee 类：

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

让我们创建以下的 EMPLOYEE 表单来存储 Employee 对象：


```
create table EMPLOYEE (
  id INT NOT NULL auto_increment,
  first_name VARCHAR(20) default NULL,
  last_name VARCHAR(20) default NULL,
  salary INT default NULL,
  PRIMARY KEY (id)
);
```

以下是将 Employee 对象和 EMPLOYEE 表单配对的映射文件。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name" type="string"/>
    <property name="lastName" column="last_name" type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

最后，我们将用 main() 方法来创建我们的应用程序类以运行应用程序，我们将使用 Session 对象和可用的 flush() 和 clear() 方法以让 Hibernate 持续将这些记录写入数据库而不是在内存中缓存它们。

```
import java.util.*;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
  private static SessionFactory factory;
  public static void main(String[] args) {
    try{
      factory = new Configuration().configure().buildSessionFactory();
    }catch (Throwable ex) {
```

```

        System.err.println("Failed to create sessionFactory object." + ex);
        throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();

    /* Add employee records in batches */
    ME.addEmployees( );
}
/* Method to create employee records in batches */
public void addEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        for ( int i=0; i<100000; i++ ) {
            String fname = "First Name " + i;
            String lname = "Last Name " + i;
            Integer salary = i;
            Employee employee = new Employee(fname, lname, salary);
            session.save(employee);
            if( i % 50 == 0 ) {
                session.flush();
                session.clear();
            }
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return ;
}
}

```

编译和执行

这里是编译和运行以上提及的应用程序的步骤。确保你已经在处理编译和运行前已经正确设置了 PATH 和 CLASSPATH。

- 如上面解释的那样创建 hibernate.cfg.xml 配置文件。

- 如上面显示的那样创建 Employee.hbm.xml 映射文件。
- 如上面显示的那样创建 Employee.java 源文件并编译。
- 如上面显示的那样创建 ManageEmployee.java 源文件并编译。
- 执行 ManageEmployee 二进制代码来运行可以在 EMPLOYEE 表单中创建 100000 个记录的程序。



18

拦截器



你已经学到，在 Hibernate 中，一个对象将被创建和保持。一旦对象已经被修改，它必须被保存到数据库里。这个过程持续直到下一次对象被需要，它将被从持久的存储中加载。

因此一个对象通过它生命周期中的不同阶段，并且 `Interceptor` 接口提供了在不同阶段能被调用来进行一些所需要的任务的方法。这些方法是从会话到应用程序的回调函数，允许应用程序检查或操作一个持续对象的属性，在它被保存，更新，删除或上传之前。以下是在 `Interceptor` 接口中可用的所有方法的列表。

S.N.	方法和描述
1	<code>findDirty()</code> 这个方法在当 <code>flush()</code> 方法在一个 <code>Session</code> 对象上被调用时被调用。
2	<code>instantiate()</code> 这个方法在一个持续的类被实例化时被调用。
3	<code>isUnsaved()</code> 这个方法在当一个对象被传到 <code>saveOrUpdate()</code> 方法时被调用。
4	<code>onDelete()</code> 这个方法在一个对象被删除前被调用。
5	<code>onFlushDirty()</code> 这个方法在当 Hibernate 探测到一个对象在一次 <code>flush</code> （例如，更新操作）中是脏的（例如，被修改）时被调用。
6	<code>onLoad()</code> 这个方法在一个对象被初始化之前被调用。
7	<code>onSave()</code> 这个方法在一个对象被保存前被调用。
8	<code>postFlush()</code> 这个方法在一次 <code>flush</code> 已经发生并且一个对象已经在内存中被更新后被调用。
9	<code>preFlush()</code> 这个方法在一次 <code>flush</code> 前被调用。

Hibernate 拦截器给予了我们一个对象如何应用到应用程序和数据库的总控制。

如何使用拦截器？

为了创建一个拦截器你可以直接实现 `Interceptor` 类或者继承 `EmptyInterceptor` 类。以下是简单的使用 Hibernate 拦截器功能的步骤。

创建拦截器

我们将在例子中继承 `EmptyInterceptor`，当 `Employee` 对象被创建和更新时拦截器的方法将自动被调用。你可以根据你的需求实现更多的方法。

```

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 MyInterceptor 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
    }

    // This method is called when Employee object gets updated.
    public boolean onFlushDirty(Object entity,
        Serializable id,
        Object[] currentState,
        Object[] previousState,
        String[] propertyNames,
        Type[] types) {
        if ( entity instanceof Employee ) {
            System.out.println("Update Operation");
            return true;
        }
        return false;
    }

    public boolean onLoad(Object entity,
        Serializable id,
        Object[] state,
        String[] propertyNames,
        Type[] types) {
        // do nothing
        return true;
    }

    // This method is called when Employee object gets created.
    public boolean onSave(Object entity,
        Serializable id,

```

```

        Object[] state,
        String[] propertyNames,
        Type[] types) {
    if ( entity instanceof Employee ) {
        System.out.println("Create Operation");
        return true;
    }
    return false;
}
//called before commit into database
public void preFlush(Iterator iterator) {
    System.out.println("preFlush");
}
//called after committed into database
public void postFlush(Iterator iterator) {
    System.out.println("postFlush");
}
}

```

创建 POJO 类

现在让我们稍微修改我们的第一个例子，我们使用 EMPLOYEE 表单和 Employee 类：

```

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
}

```

```

public void setFirstName( String first_name ) {
    this.firstName = first_name;
}
public String getLastName() {
    return lastName;
}
public void setLastName( String last_name ) {
    this.lastName = last_name;
}
public int getSalary() {
    return salary;
}
public void setSalary( int salary ) {
    this.salary = salary;
}
}

```

创建数据库表

第二步将是在你的数据库中创建表。一张表对应每个你提供持久性的对象。考虑以上的对象需要被存储和检索到以下的 RDBM 表中：

```

create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name VARCHAR(20) default NULL,
    salary INT default NULL,
    PRIMARY KEY (id)
);

```

创建 Mapping 配置文件

这个步骤是来创建一个指导 Hibernate 如何将定义类或者多个类映射到数据库表单中的映射文件。

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">

```



```

    This class contains the employee detail.
</meta>
<id name="id" type="int" column="id">
    <generator class="native"/>
</id>
<property name="firstName" column="first_name" type="string"/>
<property name="lastName" column="last_name" type="string"/>
<property name="salary" column="salary" type="int"/>
</class>
</hibernate-mapping>

```

创建 Application 类

最后，我们将用 main() 创建 application 类来运行应用程序。这里应该注意当创建 session 对象时我们使用 Interceptor 类作为参数。

```

import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    ManageEmployee ME = new ManageEmployee();

    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 10000);

    /* List down all the employees */

```

```

ME.listEmployees();

/* Update employee's records */
ME.updateEmployee(empID1, 5000);

/* Delete an employee from the database */
ME.deleteEmployee(empID2);

/* List down new list of the employees */
ME.listEmployees();
}

/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession( new MyInterceptor() );
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee(fname, lname, salary);
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}

/* Method to READ all the employees */
public void listEmployees() {
    Session session = factory.openSession( new MyInterceptor() );
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator =
            employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print(" Last Name: " + employee.getLastName());
            System.out.println(" Salary: " + employee.getSalary());
        }
        tx.commit();
    }catch (HibernateException e) {

```

```

        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
/* Method to UPDATE salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession( new MyInterceptor() );
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        employee.setSalary( salary );
        session.update(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
/* Method to DELETE an employee from the records */
public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession( new MyInterceptor() );
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        session.delete(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
}
}
}

```

编译和执行

这里是编译和运行上面提及的应用程序的步骤。确保你已经在处理编译和执行前正确设置了 PATH 和 CLASSPATH。

- 创建在 configuration 章节中解释的 hibernate.cfg.xml 配置文件。
- 创建如上所示的 Employee.hbm.xml 映射文件。
- 创建如上所示的 Employee.java 源文件并编译。
- 创建如上所示的 MyInterceptor.java 源文件并编译。
- 创建如上所示的 ManageEmployee.java 源文件并编译。
- 执行 ManageEmployee 来运行程序。

你将得到以下结果，而且记录将在 EMPLOYEE 表单中被创建。

```
$java ManageEmployee
.....VARIOUS LOG MESSAGES WILL DISPLAY HERE.....

Create Operation
preFlush
postFlush
Create Operation
preFlush
postFlush
Create Operation
preFlush
postFlush
First Name: Zara Last Name: Ali Salary: 1000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 10000
preFlush
postFlush
preFlush
Update Operation
postFlush
preFlush
postFlush
First Name: Zara Last Name: Ali Salary: 5000
First Name: John Last Name: Paul Salary: 10000
preFlush
postFlush
```

如果你检查你的 EMPLOYEE 表单，它应该有如下结果：

```
mysql> select * from EMPLOYEE;
+-----+-----+-----+-----+
| id | first_name | last_name | salary |
+-----+-----+-----+-----+
| 29 | Zara      | Ali      | 5000   |
| 31 | John      | Paul     | 10000  |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
mysql>
```

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/hibernate/>