1 Hibernate 入门

我们平常做开发的时候经常需要使用到 JdbC 连接来操作数据库，常用的操作数据库的步骤如下：

1. 指定数据库连接参数
2. 打开数据库连接
3. 声明 SQL 语句
4. 预编译并执行 SQL 语句
5. 遍历查询结果（如果需要的话）
6. 处理每一次遍历操作
7. 处理抛出的任何异常
8. 处理事务
9. 关闭数据库连接

这是一个非常繁琐的过程。具体代码如下：

```
Connection conn=null;
try {
    //1.指定数据库连接参数
    Class.forName("oracle.jdbc.driver.OracleDriver");
    String url="jdbc:oracle:thin:@localhost:1521:orcl";
    //2.打开数据库连接
    conn=DriverManager.getConnection(url,"javakc","javakc");
    //手动提交事务
    conn.setAutoCommit(false);
    //3.声明SQL语句
    String sql="select id,name,tel,address,birthday from student where
name like '李%'";
    //4.预编译并执行SQL语句
    PreparedStatement pstmt=conn.prepareStatement(sql);
    ResultSet rs=pstmt.executeQuery();
    //5.遍历查询结果（如果需要的话）
    List list=new ArrayList();
    while(rs.next()){
        //6.处理每一次遍历操作
        StudentModel sm=new StudentModel();
        sm.setId(rs.getString("id"));
        sm.setName(rs.getString("name"));
        sm.setAddress(rs.getString("address"));
        sm.setTel(rs.getString("tel"));
        java.sql.Date d=rs.getDate("birthday");
        if(d!=null){
            sm.setBirthday(new Date(d.getTime()));
        }
```

```
            list.add(sm);
        }
        //8.处理事务
        conn.commit();
    } catch (Exception e) {
        //7.处理抛出的任何异常
        try {
            conn.rollback();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
        e.printStackTrace();
    }finally{
        //9.关闭数据库连接
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

随着面向对象的软件开发方法发展而产生的。面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系数据库是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，对象-关系映射(ORM)系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。面向对象是从软件工程基本原则(如耦合、聚合、封装)的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象,对象关系映射技术应运而生。

Hibernate 是一个免费的开源 Java 包，是目前最流行的 ORM 框架，它是一个面向 Java 环境的对象/关系数据库映射工具。也是一个轻量级的 O/R Mapping 框架，它问世的时间并不长，但已经成为目前最流行的持久层解决方案。

它使得程序与数据库的交互变得十分容易，更加符合面向对象的设计思想，像数据库中包含普通 Java 对象一样，而不必考虑如何把它们从数据库表中取出。使开发者可以专注于应用程序的对象和功能，而不必关心如何保存它们或查找这些对象。甚至在对 SQL 语句完全不了解的情况下，使用 hibernate 仍然可以开发出优秀的包含数据库访问的应用程序。

1.1 Hibernate 的诞生

让时间回到 2001 年，地点是澳大利亚悉尼的 Clarence Street 有一家叫做 Cirrus Technologies 的公司，这是一家做 J2EE 企业级应用开发和咨询的公司，在会议桌上一个小伙子和老板正在进行着激烈的讨论。

小伙子："老板，我总觉得现在开发的效率太低了，我用了 EJB 的 Entity bean 1.1 时，我总觉得我浪费了好多时间在处理 Entity Bean 的体系架构上，却没有花时间在核心业务逻辑的开发上，而且 CMP 给我们的限制太多了"。

老板："Gavin，别傻了，EJB 是业界的标准，也是最流行的技术，而且我们公司是 IBM 的合作伙伴。如果有问题，问题就是我们还没有适应这样的开发模式"。

小伙子："不，我觉得肯定有更好的解决的方案。我们可以设计出比 Entity Bean 更好的方案"。

老板："哦，Gavin，我知道你很聪明，开发水平也不错。但是开发这样的系统太难了，而且你根本就没有用 SQL 开发过任何数据库系统。不要想这样一个不现实的目标啦！"

小伙子皱了皱眉，说道："不，我相信我有能力开发出这个系统。我的想法绝对是可行的。"

（注：以上场景纯属虚构，但至少以下内容完全属实：Gavin King 开发 hibernate 的动机有两个：发现 CMP 太烂；赢得对老板的争执。Gavin King 当时没有任何用 SQL 开发数据库的经验，Gavin King 开发 hibernate 的第一件事是去街上买了本 SQL 基础的书）

也许 Cirrus Technologies 的老板做梦也想不到两年以后，这个小伙子开发出的那个产品会成为全世界最流行的 O/R Mapping 工具，而那个对 SQL 和数据库一窍不通的小伙子居然会成为全世界 J2EE 数据库解决方案的领导者。

Gavin King(hibernate 创始人)

日期	事件
2001 年末	Hibernate 的第一个版本发布
2003 年 6 月	Hibernate2 发布，并于年末获得 Jolt2004 大奖，后被 JBOSS 收纳为其子项目之一
2005 年 3 月	Hibernate3 正式发布
2011 年 12 月	Hibernate4 正式发布
2013 年 5 月	hibernate4.2.2 最新版本发布

1.2Hibernate 框架简介

1.2.1 Hibernate 初识

如果想快速了解 Hibernate，我们必须记住下面的黑体字部分：

1．Hibernate 是轻量级的 ORM 框架、开源的持久层框架，对 JDBC 的 API 进行封装，负责 java 对象的持久化。

2．ORM(Object/Relational Mapping)映射工具，建立面向对象的域模型和关系数据模型之间的映射。

3．Hibernate 是 Java 应用和关系数据库之间的桥梁，它负责 Java 对象和关系数据之间的映射。是连接 java 应用和数据库的中间件.

4.在分层结构中处于持久化层，封装对数据库的访问细节，使业务逻辑层更专注于实现.

1.2.2 ORM 详解

ORM 的全称是 Object/Relation Mapping，即对象/关系映射。ORM 也可理解是一种规范，具体的 ORM 框架可作为应用程序和数据库的桥梁。

面向对象程序设计语言与关系数据库发展不同步时，需要一种中间解决方案，ORM 框架就是这样的解决方案。

ORM 并不是一种具体的产品，而是一类框架的总称，它概述了这类框架的基本特征：完成面向对象的程序设计语言到关系数据库的映射。基于 ORM 框架完成映射后，既可利用面向对象程序设计语言的简单易用性，又可利用关系数据库的技术优势。

目前 ORM 的产品非常多，比如 Apache 组织下的 OJB，Oracle 的 TopLink，JDO，JPA 等等。

对象关系映射（ORM）提供了概念性的、易于理解的模型化数据的方法。

ORM 方法论基于三个核心原则：

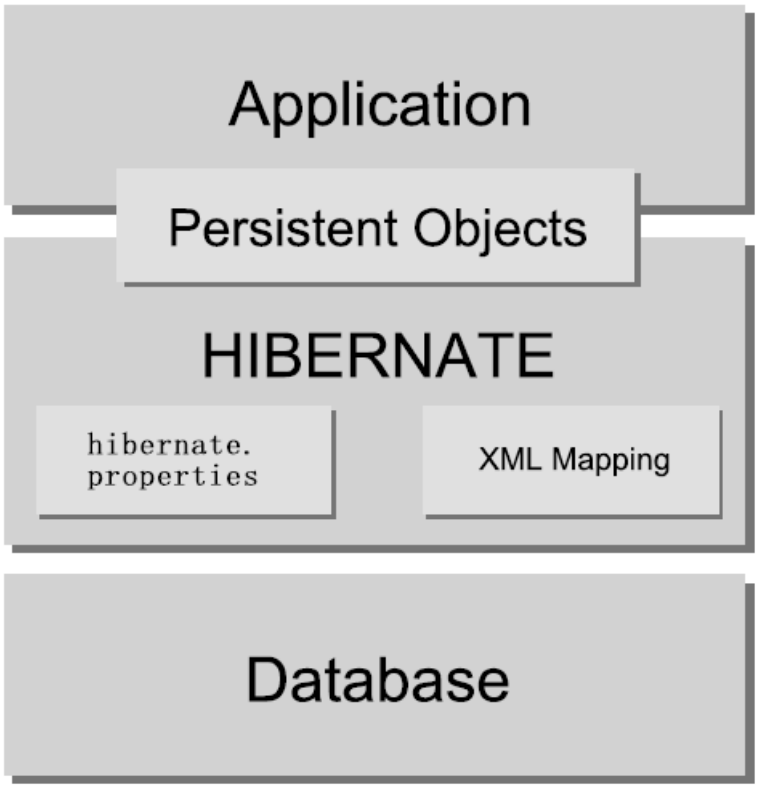
简单性：以最基本的形式建模数据。

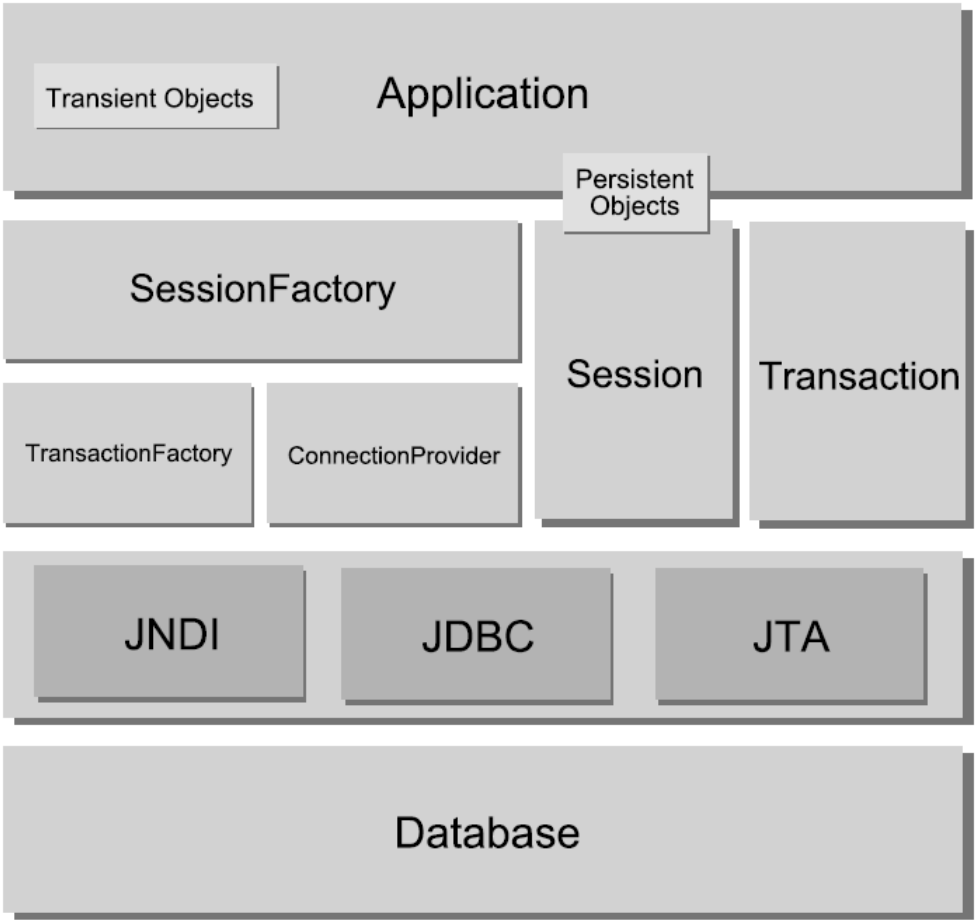
传达性：数据库结构被任何人都能理解的语言文档化。

精确性：基于数据模型创建正确标准化了的结构。

建模者通过收集来自那些熟悉应用程序但不熟练的数据建模者的人的信息开发信息模型。建模者必须能够用非技术企业专家可以理解的术语在概念层次上与数据结构进行通讯。建模者也必须能以简单的单元分析信息，对样本数据进行处理。ORM 专门被设计为改进这种联系。

1.2.3 体系结构概要图





1.3 目前流行的 ORM 持久层可选方案

	优点	缺点
SQL/JDBC	成熟，流行，使用 DAO 模式	代码烦杂，可读性差，维护困难，移植困难
Entity Bean	CMP（EJB1.1 之后），未来的 EJB3	错误的设计。不可移植，依赖性强，不可序列化，不支持多态的关联查询
JDO	简单、透明、标准	不够成熟
Apache OJB	性能、稳定性，属于 Apache 基金组织	文档资源太少，支持标准太多成了负担
iBATIS	可以控制更多的数据库操作细节。实用于遗留系统的改造和对既有数据库的复用	持久层封装不够彻底，只是一个 DBHelper
Hibernate	成熟、流行、功能强大。并逐渐发展成 Java 持久层事实上的标准。	不够透明

2 第一个 Hibernate 应用程序

2.1开发步骤

1.搭建 hibernate 运行环境，把 hibernate 相关 jar 包放到 lib 目录中。

2.搭建包结构，并创建持久化类。

3.在数据库创建持久化类对应的表。

4.配置文件

4.1 cfg.xml 放在 classes 根目录下，默认名称为 hibernate.cfg.xml

<1>添加数据库的链接

<2>可选的编程配置 例如：方言、缓存、事务、连接池

<3>映射资源的注册

4.2 hbm.xml 文件与类名相同，且放在同一目录下

<1>对象与表之间的映射

<2>对象的属性与表字段的映射

<3>组件之间的映射

<4>对象与对象之间的映射

5.客户端

5.1 读取并解析配置文件

5.2 创建 sessionFactory 对象，并创建 session 实例

5.3 开启 transaction（事务）

5.4 执行数据库操作

5.5 提交事务、关闭连接

2.2Hibernate 运行过程

1、应用程序先调用 Configuration 类,该类读取 Hibernate 配置文件及映射文件中的信息,

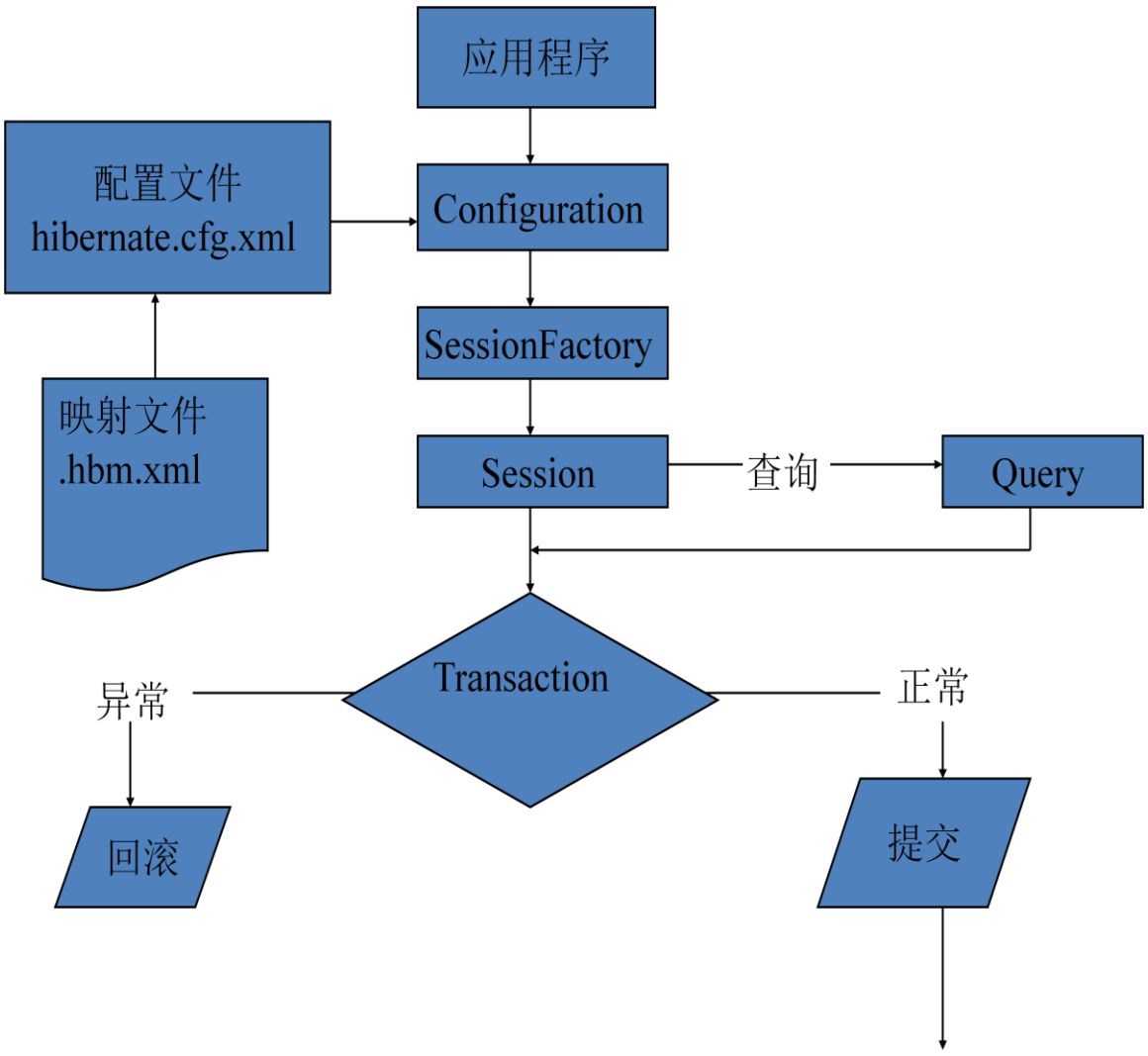
2、并用这些信息生成一个 SessionFactory 对象,

3、然后从 SessionFactory 对象生成一个 Session 对象,

4、并用 Session 对象生成 Transaction 对象;

A、可通过 Session 对象的 get(),load(),save(),update(),delete()和 saveOrUpdate()等方法对 PO 进行加载、保存、更新、删除、等操作;

B、在查询的情况下,可通过 Session 对象生成一个 Query 对象,然后利用 Query 对象执行查询操作;如果没有异常,Transaction 对象将提交这些操作到数据库中。



2.3Hibernatejar 包认识

包名	说明
hibernate3.jar	包含 Hibernate3 的基础框架和核心 API 类库,是必须使用的 jar 包
cglib-2.1.2.jar	CGLIB 库，Hibernate 用它来实现 PO 字节码的动态生成，它是非常核心的库，是必须使用的 jar 包
dom4j-1.6.1.jar	dom4j 是一个 Java 的 XML API，类似于 jdom，用来读写 XML 文件
commons-collections.jar	Apache Commons 包中的一个，包含了一些 Apache 开发的集合类，功能比 java.util.*强大。必须使用的 jar 包
commons-logging.jar	Apache Commons 包中的一个，包含了日至功能，必须使用的 jar 包
ant-1.6.5.jar	Ant 编译工具的 jar 包，用来编译 Hibernate 源代码的。它是可选包
c3po-0.9.0.jar	C3PO 是一个数据库连接池，Hibernate 可以配置为使用 C3PO 的连接池，如果准备用这个连接池，就需要这个 jar 包
connector.jar	JCA(Java Cryptography Architecture，Java 加密架构，java 平台中用于访问和开发加密功能的框架)规范，如果在 App Server 上把 Hibernate 配置为 Connector,就需要这个 jar。一般 App Server 都会带上这个包
jaas.jar	JAAS 是用来进行权限验证的，已经包含在 JDK1.4 里面了。所以它的实际上是多余的包
jdbc2_0-stdext.jar	JDBC2.0 的扩展包，一般来说数据库连接池会用上它，不过 App Server 都会带上它，所以也是多余的
jta.jar	JTA(java 事务处理的机制)规范，当 Hibernate 使用 JTA 的时候需要，不过 App Server 都会带上它，所以也是多余的
junit-3.8.1.jar	Junit 包，当运行 Hibernate 自带的测试代码的时候需要，否则就不用
xerces-2.6.2.jar xml-apis.jar	是 XML 解析器，xml-apis 实际上是 JAXP。也是多余的包

2.4Hibernate 核心接口

接口	描述
Configuration	配置 hibernate，根启动 hibernate，创建 sessionFactory 对象。
SessionFactory	负责创建 Session 对象，可以通过 Configuration 对象创建 SessionFactory 对象。 SessionFactory 对象中保存了当前的数据库配置信息和所有映射关系以及预定义的 SQL 语句。同时，SessionFactory 还负责维护 Hibernate 的二级缓存。 SessionFactory 对象的创建会有较大的开销，而 SessionFactory 对象采取了线程安全的设计方式，因此在实际中 SessionFactory 对象可以尽量的共享，在大多数情况下，一个应用中针对一个数据库可以共享一个 SessionFactory 实例
Session	使用最广泛，也被称为持久化管理器，它提供和持久化相关的操作。增、删、改、查等。 Session 不是线程安全的，它代表与数据库之间的一次操作，因此，一个 Session 对象只可以由一个线程使用。避免多个线程共享。轻量级的，创建和销毁不需要消耗太多资源。Session 通过 SessionFactory 打开，在所有的工作完成后，需要关闭。 Session 中有一个缓存，称为一级缓存。 存放当前工作单元加载的对象。 它与 Web 层的 HttpSession 没有任何关系。
Transaction	Transaction 是 Hibernate 中进行事务操作的接口，将应用代码从底层的事务实现中抽象出来，这可能是一个 JDBC 事务，一个 JTA 用户事务或者甚至是一个公共对象请求代理结构（CORBA）。允许应用通过一组一致的 API 控制事务边界。 使用 Hibernate 进行操作时（增、删、改）必须显示的调用 Transaction（默认：autoCommit=false）
Query 和 Criteria	执行数据库查询对象，以及控制执行查询过程

2.5为什么使用 hibernate

Hibernate 能在众多的 ORM 框架中脱颖而出，因为 Hibernate 与其他 ORM 框架对比具有如下优势：

1、开源和免费的 License，方便需要时研究源代码、改写源代码、进行功能定制。

2、轻量级封装，避免引入过多复杂的问题，调试容易，可减轻程序员负担。

3、具有可扩展性，API 开放。功能不够用时，可自己编码进行扩展。

4、开发者活跃，产品有稳定的发展保障。Hibernate 的工作方式灵巧的设计，出色的性能表现。

2.6要求与目标

要求：熟悉 Java、SQL、JDBC，掌握面向对象的开发方法。

课程目标：理解 O/R Mapping 原理，掌握 Hibernate 开发的相关知识，并能使用 Hibernate 进行实际项目开发。

3 Hibernate 的配置

3.1 Hibernate.cfg.xml

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

<hibernate-configuration>

   <session-factory>

      <!-- 1. Database connection settings -->
      <property name="connection.driver_class">
         oracle.jdbc.driver.OracleDriver
      </property>
      <property name="connection.url">
         jdbc:oracle:thin:@127.0.0.1:1521:orcl
      </property>
      <property name="connection.username">javakc24</property>
      <property name="connection.password">javakc24</property>

      <!-- SQL dialect anyversion -->
      <property name="hibernate.dialect">
         org.hibernate.dialect.OracleDialect
      </property>
      <property name="hibernate.connection.isolation">2</property>
      <property name="hibernate.show_sql">true</property>
      <property name="hibernate.format_sql">true</property>
      <property name="hibernate.use_sql_comments">true</property>

      <!-- Names the annotated entity class -->
      <mapping resource="com/javakc/hibernate/Cat.hbm.xml" />

   </session-factory>

</hibernate-configuration>
```

3.2 Configuration 接口

Configuration 负责管理 Hibernate 的配置信息。

Hibernate 运行时需要一些底层实现的基本信息。

这些信息包括：数据库 URL、数据库用户名、数据库用户密码、数据库 JDBC 驱动类、数据库 dialect。

用于对特定数据库提供支持，其中包含了针对特定数据库特性的实现，如 Hibernate 数据库类型到特定数据库数据类型的映射等。

使用 Hibernate 必须首先提供这些基础信息以完成初始化工作，为后续操作做好准备。

这些属性在 Hibernate 配置文件 hibernate.cfg.xml 中加以设定，

当调用：Configuration config=new Configuration().configure();时，

Hibernate 会自动在目录下搜索 hibernate.cfg.xml 文件，并将其读取到内存中作为后续操作的基础配置。

3.3Configuration 加载方式

3.3.1 读取 hibernate.properties

```
    //通过configuration直接读取配置文件
    //默认读取src目录下的hibernate.properties
    Configuration config = new Configuration();
```

3.3.2 读取 hibernate.cfg.xml

```
    //通过configuration().configure().读取hibernate.cfg.xml配置文件
    Configuration config = new Configuration().configure();
```

3.3.3 读取指定位置 hibernate.cfg.xml

```
    //调用有参的 configure方法传入cfg文件的路径加载配置文件
    Configuration config = new
Configuation().configure("com/javakc/hibernate/hibernate.cfg.xml");
```

3.3.4 通过编程方式配置 Hibernate

```
    Configuration config = new Configuration().configure();
    // 加载连接数据库配置
    config.setProperty("connection.driver_class",
           "oracle.jdbc.driver.OracleDriver");
```

```
    config.setProperty("connection.url",
            "jdbc:oracle:thin:@127.0.0.1:1521:orcl");
    config.setProperty("connection.username", "javakc");
    config.setProperty("connection.password", "javakc");
    // 加载可选编程配置
    config.setProperty("hibernate.dialect",
            "org.hibernate.dialect.OracleDialect");
    // 加载映射资源文件/类
    config.addClass(StudentModel.class);
    config.addResource("com/javakc/hibernate/Cat.hbm.xml");
```

3.4获取 sessionFactory

```
    //当所有的映射文件被Configuration解析后，应用程序必须要获取一个构建session实例的工厂类，这个工厂将被应用程序所有线程共享。
    SessionFactory sessionFactory = config.buildSessionFactory();
```

config 会根据当前的数据库配置信息，构造 SessionFacory 实例并返回。SessionFactory 一旦构造完毕，即被赋予特定的配置信息。也就是说，之后 config 的任何变更将不会影响到已经创建的 SessionFactory 实例 sessionFactory。如果需要使用基于变更后的 config 实例的 SessionFactory，需要从 config 重新构建一个 SessionFactory 实例。SessionFactory 保存了对应当前数据库配置的所有映射关系，同时也负责维护当前的二级数据缓存和 Statement Pool。由此可见，SessionFactory 的创建过程非常复杂、代价高昂。这也意味着，在系统设计中充分考虑到 SessionFactory 的重用策略。由于 SessionFactory 采用了线程安全的设计，可由多个线程并发调用。

3.5JDBC 连接属性

hibernate.connection.driver	Jdbc 驱动类
hibernate.connection.url	Jdbc URL
hibernate.connection.username	数据库用户名
hibernate.connection.password	数据库用户密码
hibernate.connection.pool_size	连接池容量上限数目

```
hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql://localhost/hibernate
hibernate.connection.username root
hibernate.connection.password root
hibernate.jdbc.fetch_size 50
```

设定JDBC的Statement读取数据的时候每次从数据库中取出的记录条数

```
hibernate.jdbc.batch_size 25
```

是设定对数据库进行批量删除，批量更新和批量插入的时候的批次大小，有点相当于设置Buffer缓冲区大小的意思。Batch Size越大，批量操作的向数据库发送sql的次数越少，速度就越快。我做的一个测试结果是当Batch Size=0 的时候，使用Hibernate对Oracle数据库删除 1 万条记录需要 25 秒，Batch Size = 50 的时候，删除仅仅需要 5 秒！

3.6hibernate 可选配置属性

属性名称	用途
dialect	一个 hibernate dialect 类允许 hibernate 针对特定的关系型数据库生成优化的 sql 语句.
show_sql	输出所有的 sql 语句到控制台.
format_sql	在 log 和 console 输出更漂亮的 sql 语句.
use_sql_comments	在输出的 sql 语句中添加注释
default_schema	在生成的 sql 语句中,将给定的 schema 附加于非全限定名的表名上.
default_catalog	在生成的 sql 语句中，将给定的 catalog 附加于非全限定名的表名上.
cache.use_query_cache	是否启用二级缓存
cache.provider_class	二级缓存 class org.hibernate.cache.NoCacheProvider org.hibernate.cache.EhCacheProvider
cache.provider_configuration_file_resource_path	加载二级缓存配置文件 /cache/ehcache.xml
hbm2ddl.auto	是否同步数据库结构 取值：none、create、create-drop、update、validate
max_fetch_depth	设置对单个表的外连接数最大深度。0 是屏蔽默认的外连接设置。推荐

	设置为 0 到 3 之间。
default_batch_fetch_size	设置 Hibernate 批量联合查询的尺度。强烈建议。推荐设置为 4、8、16。
default_entity_mode	默认的实体表现模式，通过 SessionFactory 打开的所有的 Session。取值，dynamic-map、dom4j、pojo。
order_updates	强迫 Hibernate 通过被更新项的主键值排序 SQL 更新。这样可以在高并发时，减少事务死锁。取值 true|false。
connection.isolation	设置 JDBC 事务隔离的级别。检查 java.sql.Connection 的定义的常量值，但要注意大多数数据库不支持所有的隔离级别、一些附加的和非标准的隔离级别。取值，1、2、4、8。
connection.autocommit	JDBC 共享连接的自动提交。（不推荐）取值，true|false。

3.7 hibernate 方言

数据库都是支持 SQL 的，不过不同的数据库会存在一些 SQL 语法上面的差异，而方言则保证了 HQL 翻译成 SQL 的正确性。

数据库	方言
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2OS390	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/10g	org.hibernate.dialect.Oracle9Dialect
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

4 映射的配置

4.1 映射定义

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

请注意，虽然很多 Hibernate 用户选择手写 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。

让我们从一个映射的例子开始：

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

<hibernate-mapping package="com.javakc.hibernate">

    <class name="Cat" table="cats" >

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

        <property name="weight" />

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

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

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

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

    </class>

</hibernate-mapping>
```

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

4.2Doctype 文档类型

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/net/sf/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。

4.3hibernate-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"
        catalog="catalogName"
        default-cascade="cascade_style"
        default-lazy="true|false"
        auto-import="true|false"
        package="package.name"
    />
```

属性	默认	描述
schema	可选	数据库 schema 的名称
catalog	可选	数据库 catalog 的名称
default-cascade	可选 - 默认为 none	默认的级联风格
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
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）
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“import 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。

4.4 class

你可以使用 class 元素来定义一个持久化类：

```
<class
    name="ClassName"
    table="tableName"
    discriminator-value="discriminator_value"
    mutable="true|false"
    schema="owner"
    catalog="catalog"
    proxy="ProxyInterface"
    dynamic-update="true|false"
    dynamic-insert="true|false"
    select-before-update="true|false"
    polymorphism="implicit|explicit"
    where="arbitrary sql where condition"
    persister="PersisterClass"
    batch-size="N"
    optimistic-lock="none|version|dirty|all"
    lazy="true|false"
    entity-name="EntityName"
    check="arbitrary sql check condition"
    rowid="rowid"
    subselect="SQL expression"
    abstract="true|false"
    node="element-name"
/>
```

属性	默认	描述
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
table	可选 默认是类的非全限定名	对应的数据库表名
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

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
schema	可选	覆盖在根<hibernate-mapping>元素中指定的 schema 名字
catalog	可选	覆盖在根<hibernate-mapping>元素中指定的 catalog 名字
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
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
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
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，来决定是否应该执行 UPDATE。
polymorphism（多态）	可选-默认值为 implicit (隐式)	界定是隐式还是显式的使用多态查询(这只在 Hibernate 的具体表继承策略中用到－译注)
where	可选	指定一个附加的 SQLWHERE 条件，在抓取这个类的对象时会一直增加这个条件
persister	可选	指定一个定制的 ClassPersister
optimistic-lock（乐观锁定）	可选-默认是 version	决定乐观锁定的策略
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）
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）
entity-name	可选-默认为类名	Hibernate3 允许一个类进行多次映射（前提是映射到不同的表），并且允许使用 Maps 或 XML 代替 Java 层次的实体映射（也就是实现动态领域模型，不用写持久化类－译注）
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
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
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
abstract	可选	用于在<union-subclass>的继承结构（hierarchies）中标识抽象超类

4.5 id

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

```
<id
    name="propertyName"
    type="typename"
    column="column_name"
    unsaved-value="null|any|none|undefined|id_value">
    <generator class="generatorClass" />
</id>
```

属性	默认	描述
name	可选	标识属性的名字
type	可选	标识 Hibernate 类型的名字
column	可选- 默认为属性名	主键字段的名字
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

4.6 generator

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

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

所有的生成器都实现 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：

属性名称	用途
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	用一个 128-bit 的 UUID 算法生成字符串类型的标识符，这在一个网络中是唯一的（使用了 IP 地址）。UUID 被编码为一个 32 位 16 进制数字的字符串。
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 save()之前为对象分配一个标示符。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。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,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。注意，因为 Oracle 驱动程序的一个 bug，这些插入语句的注释被关闭了。

4.6.1 程序分配的标识符（Assigned Identifiers）

Assigned 为数据库主键 ID 手动控制，所以在程序执行保存操作中，我们需要手动去控制 ID 生成方式!

```
<id name="id" column="cat_id" type="string">
    <generator class="assigned"/>
</id>
```

4.6.2 UUID 算法

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）。

```
<id name="id" column="cat_id" type="string">
    <generator class="uuid"/>
```

```
</id>
```

4.6.3 sequence 序列

hibernate 使用 oracle 中的 sequence 名称，作为主键的生成。

```
<id name="id" column="cat_id" type="string">
        <generator class="native">
                <param name="sequence">cat_sequence</param>
        </generator>
</id>
```

4.6.4 increment 自动增长

主键生成方式，不适合在集群范围内使用！

```
<id name="id" column="cat_id" type="string">
    <generator class="increment"/>
</id>
```

4.7property

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

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

属性	默认	描述
name		属性的名字,以小写字母开头
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
type	可选	一个 Hibernate 类型的名字
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
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）
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
not-null	可选	使用 DDL 为该字段添加可否为空（nullability）的约束
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

4.8 Hibernate 的内置映射类型

Hibernate 映射类型分为两种：内置映射类型和客户化映射类型。内置映射类型负责把一些常见的 Java 类型映射到相应的 SQL 类型；此外，Hibernate 还允许用户实现 UserType 或 CompositeUserType 接口，来灵活地定制客户化映射类型。客户化类型能够把用户定义的 Java 类型映射到数据库表的相应字段

4.8.1 Java 基本类型的 Hibernate 映射类型

Hibernate 映射类型	Java 类型	标准 SQL 类型	大小和取值范围
integer 或者 int	int 或者 java.lang.Integer	INTEGER	4 字节
long	long Long	BIGINT	8 字节
short	short Short	SMALLINT	2 字节
byte	byte Byte	TINYINT	1 字节
float	float Float	FLOAT	4 字节
double	double Double	DOUBLE	8 字节
big_decimal	java.math.BigDecimal	NUMERIC	NUMERIC(8,2)8 位
character	char Character String	CHAR(1)	定长字符
string	String	VARCHAR	变长字符串
boolean	boolean Boolean	BIT	布尔类型

yes_no	boolean Boolean	CHAR(1) (Y-N)	布尔类型
true_false	boolean Boolean	CHAR(1) (T-F)	布尔类型

4.8.2 Java 时间和日期类型的 Hibernate 映射类型

映射类型	Java 类型	标准 SQL 类型	描述
date	util.Date 或者 sql.Date	DATE	YYYY-MM-DD
time	Date Time	TIME	HH:MM:SS
timestamp	Date Timestamp	TIMESTAMP	YYYYMMDDHHMMSS
calendar	calendar	TIMESTAMP	YYYYMMDDHHMMSS
calendar_date	calendar	DATE	YYYY-MM-DD

4.8.3 Java 大对象类型的 Hibernate 映射类型

映射类型	Java 类型	标准 SQL 类型	MySQL 类型	Oracle 类型
binary	byte[]	VARBINARY(或 BLOB)	BLOB	BLOB
text	String	CLOB	TEXT	CLOB
serializable	Serializable 接口任意实现类	VARBINARY(或 BLOB)	BLOB	BLOB
clob	java.sql.Clob	CLOB	TEXT	CLOB
blob	java.sql.Blob	BLOB	BLOB	BLOB

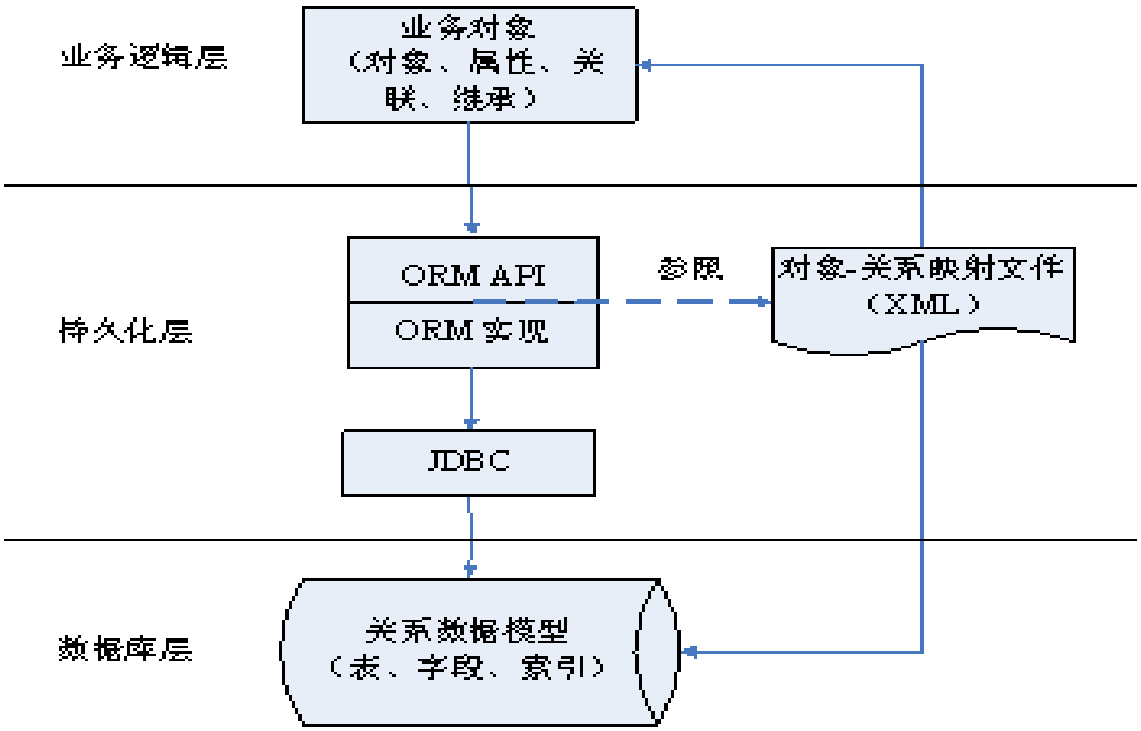
5 Hibernate 与对象共事

5.1 持久化层含义

访问数据库代码(Dao)与业务逻辑(Service)混杂在一起带来了很多问题，这样的程序设计严重限制了程序的可扩展性和适应性，所以有必要要把涉及数据库操作的代码分离出来与业务逻辑分离。就形成了所谓“持久化层”的概念。

持久化(Persistence)，即把数据(如内存中的对象)保存到可永久保存的存储设备中(如磁盘)。持久化的主要应用是将内存中的数据存储在关系型的数据库中，当然也可以存储在磁盘文件中、XML 数据文件中等等。

ORM 工具实现持久化示意图



ORM 工具实现持久化示意图

5.2 Session 接口

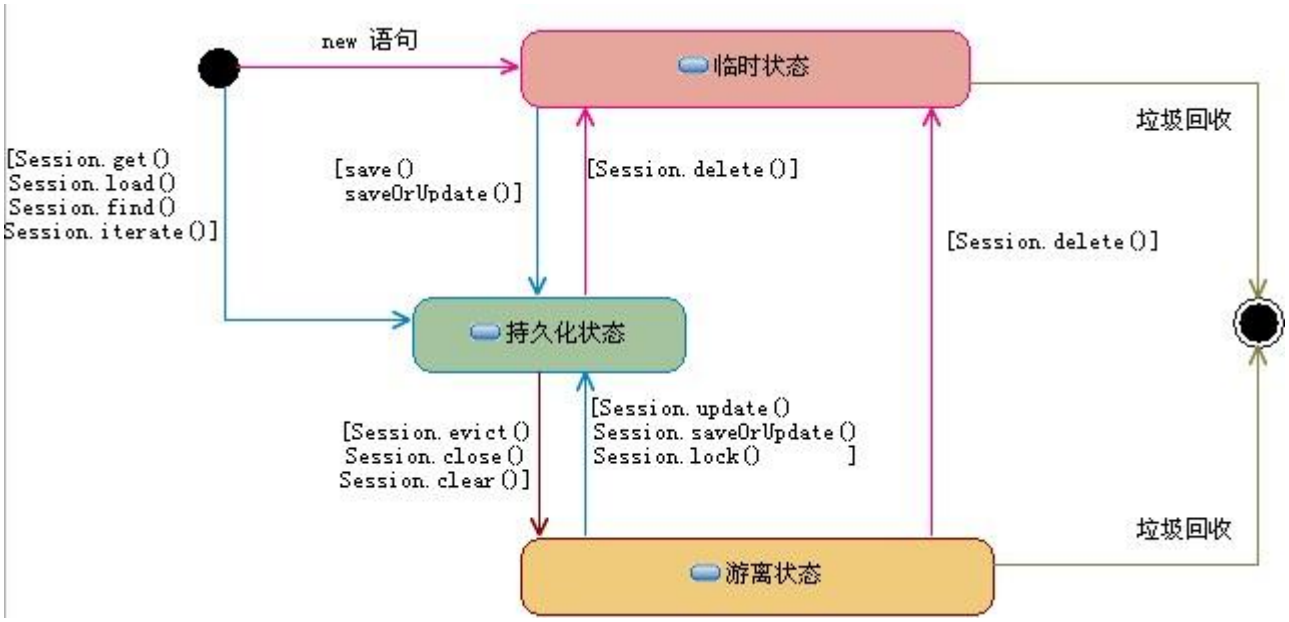
Session 接口对于 Hibernate 开发人员来说是一个最重要的接口。然而在 Hibernate 中，实例化的 Session 是一个轻量级的类，创建和销毁它都不会占用很多资源。这在实际项目中确实很重要，因为在客户程序中，可能会不断地创建以及销毁 Session 对象，如果 Session 的开销太大，会给系统带来不良影响。但值得注意的是 Session 对象是非线程安全的，因此在你的设计中，最好是一个线程只创建一个 Session 对象。

在 Hibernate 的设计者的头脑中，他们将 session 看作介于数据连接与事务管理一种中间接口。我们可以将 session 想象成一个持久对象的缓冲区，Hibernate 能检测到这些持久对象的改变，并及时刷新数据库。我们有时也称 Session 是一个持久层管理器，因为它包含这一些持久层相关的操作，诸如存储持久对象至数据库，以及从数据库从获得它们。

Java 对象在 Hibernate 持久化层的三种状态：

- 临时状态：刚用 new 语句创建，还没有被持久化，并且不处于 session 缓存中（处于临时状态的对象成为临时对象）
- 持久化状态：已经被持久化，并且加入到 session 缓存中。处于持久化状态的对象称为持久化对象
- 游离状态：已经被持久化，但不再处于 session 缓存中。处于游离状态的对象称为游离对象

通过研究 Sesison 的生命周期，可以更清楚的理解对象在 Session 中的三种状态。



记清楚这三种状态，对于理解 Hibernate 的运行原理至关重要。

5.3使对象持久化-save

Hibernate 认为持久化类(perssistent class)新实例化的对象是瞬时(Transient)的。我们可通过将瞬时对象(Transient)与 session 关联而把它变成持久(Persistent)的。

```
Cat cat=new Cat();
cat.setName("花花");
cat.setColor("黑色");
cat.setWeight(1.0);
cat.setBirthdate(new Date(2013,10,3));
Long generatedId = (Long)session.save(cat);
```

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 sava()被调用时产生并分配给 cat。如果 Cat 持久化标识(identifier)是 assigned 类型的，那么该标识(identifier)应当在调用 save()之前手动赋值给 cat。

5.4装载对象-load/get

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 class 对象。 load 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 cat=(Cat)session.load(Cat.class, generatedId);
```

或者这样写：

```
Cat cat=(Cat)session.load(Cat.class, new Long(1));
```

说明:从数据库加载一个实体对象后，返回的对象都位于 Session 缓存中，接下来修改了持久化对象的属性后，当 Session 清理缓存时，会根据持久化对象的属性变化来同步更新数据库。

```
Cat cat=(Cat)session.load(Cat.class, new Long(1));
cat.setWeight(2.0);
session.close();
```

Hibernate 会在后台先进行 select 查询操作，再进行 update 修改操作。

将上述代码中的 load 方法修改成 get 方法，也可以完成上述功能。

get/load 区别：

(1) 当数据库中不存在与 ID 对应的记录时，load()方法抛出 ObjectNotFoundException 异常，而 get()方法返回 null.

(2) 两者采用不同的检索策略。

默认情况下，load()方法采用延迟检索策略（Hibernate 不会执行 select 语句，仅返回实体类的代理类实例，占用内存很少）；

而 get()采用立即检索策略（Hibernate 会立即执行 select 语句）。

使用场合：

（1）如果加载一个对象的目的是为了访问它的各个属性，可以用 get()；

（2）如果加载一个对象的目的是为了删除它，或者建立与别的对象的关联关系，可以用 load()；

5.5 修改持久对象-update

update()方法把游离对象加入当前 Session 缓存中，计划执行 update 语句。

```
Cat cat=new Cat();
cat.setId(1L);
cat.setName("花花");
cat.setColor("灰色");
cat.setWeight(2.0);
cat.setBirthdate(new Date(2013,10,3));
session.update(cat);
```

当 update()方法关联一个游离对象时，如果 session 缓存中已经有一个同类型且 ID 相同的持久化对象，那么 update()方法会抛出 NonUniqueException 异常.

当 update()方法关联一个持久化对象时，该方法不起作用。

5.6 自动状态监测-saveOrUpdate

说明： 同时包含了 save()和 update()方法的功能。

如果传入的是临时对象，就调用 save()方法；

如果传入的是游离对象，就调用 update()方法

如果传入的是持久化对象，就直接返回。

示例： session.saveOrUpdate(model);

5.7删除持久对象-delete

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伤处对象的引用。所以，最好这样理解：delete()的用途是把对象从 Session 缓存中删除，也就是说把一个持久实例变成瞬时(transient)实例。

```
session.delete(cat);
```

5.8close()

说明： 清空 session 缓存。

示例： session.close();

5.9clear()

说明：清除 session 中的缓存数据（不管缓存与数据库的同步）。

示例：session.clear();

5.10 flush()

说明：将 session 的缓存中的数据与数据库同步。

示例：session.flush();

6 Hibernate 查询语言 HQL

HQL：Hibernate Qusery Language，如果你已经熟悉它，就会发现它跟 SQL 非常相像。不过 你不要被表面的假象迷惑，HQL 是面向对象的（OO，用生命的眼光看待每一个对象，他们是如此 鲜活）。如果你对 JAVA 和 SQL 语句有一定了解的话，那么 HQL 对你简直易如反掌，你完全可以利用在公车上的时间掌握它。

6.1 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。

本教程中的 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。

6.2 from 子句

Hibernate 中最简单的查询语句的形式如下：

```
from com.javakc.hibernate.Student
```

该子句简单的返回 com.javakc.hibernate.Student 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 Student
```

大多数情况下，你需要指定一个别名，原因是你可能需要在查询语句的其它部分引用到 Student。

```
from Student as s
```

这个语句把别名 s 指定给类 Student 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 Students
```

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 Student,Person
```

```
from Student as s, Person as p
```

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)。

6.3 select 子句

select 子句用于确定选择出的属性，当然 select 选择的属性必须是 from 后持久化类包含的属性。例如：

```
select p.name from Person as p
```

select 可以选择任意属性，不仅可以选择持久化类的直接属性，还可以选择组件属性包含的属性，例如：

```
select p.name.firstName from Person as p
```

查询语句可以返回多个对象和（或）属性，存放在 Object[]数组中：

```
select p.name,p.address from Person as p
```

select 也支持将选择出的属性存入一个 List 对象中，例如：

```
select new list(p.name , p.address) from Person as p
```

甚至可以将选择出的属性直接封装成对象，例如：

```
select new ClassTest(p.name , p.address) from Person as p
```

前提是 ClassTest 支持 p.name 和 p.address 的构造器，假如 p.name 的数据类型是 String，p.address 的数据类型是 String，则 ClassTest 必须有如下的构造器：

```
ClassTest(String s1, String s2)
```

select 还支持使用 as 给选中的表达式命名别名，例如：

```
select count(p.name) as num from Person as p
```

下面这种用法与 new map 结合使用更普遍。如：

```
select new map(p.name as personName) from Person as p
```

在这种情形下，选择出的是 Map 结构，以 personName 为 key，实际选出的值作为 value。

6.4 聚集函数

HQL 也支持在选出的属性上，使用聚集函数。HQL 支持的聚集函数与 SQL 完全相同，如下：

- avg，计算属性平均值。
- count，统计选择对象的数量。
- max，统计属性值的最大值
- min，统计属性值的最小值。

- sum，计算属性值的总和。

例如：

```
select count(name) from Person
select max(p.age) from Person as p
```

select 子句还支持字符串连接符、算术运算符以及 SQL 函数。如：

```
select p.age+1 from Person as p
select p.name || "" || p.address from Person as p
```

select 子句也支持使用 distinct 和 all 关键字，此时的效果与 SQL 中的效果完全相同。

6.5 where 子句

where 子句用于筛选选中的结果，缩小选择的范围。如果没有为持久化实例命名别名，可以直接使用属性名引用属性。

如下面的 HQL 语句：

```
from Person where name like 'tom%'
```

上面 HQL 语句与下面的语句效果相同：

```
from Person as p where p.name like "tom%"
```

在后面的 HQL 语句中，如果为持久化实例命名了别名，则应该使用完整的属性名。两个 HQL 语句都可返回 name 属性以 tom 开头的实例。

6.6 表达式

HQL 的功能非常丰富，where 子句后支持的运算符异常丰富，不仅包括 SQL 的运算符，还包括 EJB-QL 的运算符等。

where 子句中允许使用大部分 SQL 支持的表达式：

- 数学运算符+、-、*、/ 等。
- 二进制比较运算符=、>=、<=、<>、!=、like 等。
- 逻辑运算符 and、or、not 等。
- in、not in、between、is null、is not null、is empty、is not empty、member of 和 not member of 等。
- 简单的 case、case ... when ... then ... else ... end 和 case、case when ... then ... else ... end 等。
- 字符串连接符 value1 || value2 或使用字符串连接函数 concat(value1 , value2)。

- 时间操作函数 current_date()、current_time()、current_timestamp()
- second()、minute()、hour()、day()、month()、year()等。
- HQL 还支持 EJB-QL 3.0 所支持的函数或操作 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
- 还支持数据库的类型转换函数，如 cast(... as ...)，第二个参数是 Hibernate 的类型名，或者 extract(... from ...)，前提是底层数据库支持 ANSI cast()和 extract()。
- 如果底层数据库支持如下单行函数 sign()、trunc()、rtrim()、sin()。则 HQL 语句也完全可以支持。
- HQL 语句支持使用？作为参数占位符，这与 JDBC 的参数占位符一致
- 可以使用命名参数占位符号，方法是在参数名前加冒号 :，例如 :start_date 和:x1 等。
- SQL 常量，例如'foo'、69、'1970-01-01 10:00:01'等。
- Java public static final 类型的常量，例如 eg.Color.TABBY。

in 与 between...and 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')
```

当然，也支持 not in 和 not between...and 的使用，例如：

```
from DomesticCat cat where cat.name not between 'A' and 'B'
from DomesticCat cat where cat.name not in ( 'Foo','Bar','Baz' )
```

子句 is null 与 is not null 可以被用来测试空值，例如：

```
from DomesticCat cat where cat.name is null;
from Person as p where p.address is not null;
```

如果在 Hibernate 配置文件中进行如下声明：

```
<property name="hibernate.query.substitutions">true 1,false 0</property>
```

上面的声明表明，HQL 转换 SQL 语句时，将使用字符 1 和 0 来取代关键字 true 和 false。然后将可以在表达式中使用布尔表达式，例如：

```
from Cat cat where cat.alive = true
```

6.7 order by 子句

查询返回的列表(list)可以根据类或组件属性的任何属性进行排序，例如：

```
from Person as p order by p.name, p.age
```

还可使用 asc 或 desc 关键字指定升序或降序的排序规则，例如：

```
from Person as p order by p.name asc , p.age desc
```

如果没有指定排序规则，默认采用升序规则。即是否使用 asc 关键字是没有区别的，加 asc 是升序排序，不加 asc 也是升序排序。

6.8group by 子句

返回聚集值的查询可以对持久化类或组件属性的属性进行分组，分组所使用的 group by 子句。看下面的 HQL 查询语句：

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

类似于 SQL 的规则，出现在 select 后的属性，要么出现在聚集函数中，要么出现在 group by 的属性列表中。

having 子句用于对分组进行过滤，如下：

```
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)
```

注意：having 子句用于对分组进行过滤，因此 having 子句只能在有 group by 子句时才可以使用，没有 group by 子句，不能使用 having 子句。

Hibernate 的 HQL 语句会直接翻译成数据库 SQL 语句。因此，如果底层数据库支持的 having 子句和 group by 子句中出现一般函数或聚集函数，HQL 语句的 having 子句和 order by 子句中也可以出现一般函数和聚集函数。例如：

```
select cat
from Cat cat
join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

6.9子查询

如果底层数据库支持子查询，则可以在 HQL 语句中使用子查询。与 SQL 中子查询相似的是，HQL 中的子查询也需要使用()括起来。如：

```
from Cat as fatcat
where fatcat.weight > ( select avg(cat.weight) from DomesticCat cat )
```

如果 select 中包含多个属性，则应该使用元组构造符：

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

7 条件查询 Criteria Query

Criteria 是一种比 hql 更面向对象的查询方式，在查询方法设计上可以灵活的根据 Criteria 的特点来方便地进行查询条件的组装。应对简单查询，比使用 HQL 更简单。

7.1 创建 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。

下面代码完成了查询 Cat 类对应数据表的前 50 条数据：

```
Criteria crit = session.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

7.2 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.sql("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();
```

7.3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();
```

```
    }
```

7.4关联

你可以使用 createCriteria()非常容易的在互相关联的实体间建立约束。

```
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");
    }
```

8 Native Sql 查询

Hibernate 还支持使用 SQL 查询，使用 SQL 查询可以利用某些数据库的特性，或者用于将原有的 JDBC 应用迁移到 Hibernate 应用上。使用命名的 SQL 查询还可以将 SQL 语句放在配置文件中配置，从而提高程序的解耦，命名 SQL 查询还可以用于调用存储过程。如果是一个新的应用，通常不要使用 SQL 查询。

SQL 查询是通过 SQLQuery 接口来表示的，SQLQuery 接口是 Query 接口的子接口，因此完全可以调用 Query 接口的方法：

setFirstResult()，设置返回结果集的起始点。

setMaxResults()，设置查询获取的最大记录数。

list()，返回查询到的结果集。

但 SQLQuery 比 Query 多了两个重载的方法：

addEntity，将查询到的记录与特定的实体关联。

addScalar，将查询的记录关联成标量值。

8.1 使用 SQLQuery

hibernate 对原生 SQL 查询执行的控制是通过 SQLQuery 接口进行的.

```
session.createSQLQuery();
```

下面展示一下最基本的 SQL 查询：

```
session.createSQLQuery("SELECT * FROM CATS").list();
session.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

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。

8.2 标量查询

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()。

```
session.createSQLQuery("SELECT * FROM CATS")
        .addScalar("ID", Hibernate.LONG)
        .addScalar("NAME", Hibernate.STRING)
        .addScalar("BIRTHDATE", Hibernate.DATE);
```

这个查询指定了: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。

```
session.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.

8.3 实体查询

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。

```
session.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
session.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：

```
session.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS")
       .addEntity(Cat.class);
```

这样 cat.getDog()就能正常运行。

8.4 返回多个实体

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）：

```
    session.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：

```
session.createSQLQuery(
"SELECT {cat.*}, {mother.*}   FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
    .addEntity("cat", Cat.class).addEntity("mother", Cat.class);
```

这个查询指明：SQL 查询语句，其中包含占位符来让 Hibernate 标记字段别名,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 来为每个属性标记 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 = session.createSQLQuery(sql)
    .addEntity("cat", Cat.class).addEntity("mother", Cat.class)
    .list();
```

8.5返回非托管实体

可以对原生 sql 查询使用 ResultTransformer。这会返回不受 Hibernate 管理的实体。

```
session.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。

8.6使用存储过程来查询

1. 首先添加查询存储过程

```
DROP PROCEDURE IF EXISTS `getCatList`;

CREATE PROCEDURE `getCatList`()

begin

     select * from tbl_cat;

end;
```

2. Hbm.xml 文件的配置

```
<?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="com.javakc.cat.model">

    <class name="Cat" table="tbl_cat">

        <id name="catId" column="catid">
            <generator class="assigned"/>
        </id>

        <property name="catName" column="catname" type="string" />

        <property name="catWidth" column="catwidth" type="string" />

    </class>

    <sql-query name="getCatList" callable="true">

        <return alias="Cat" class="Cat">
            <return-property name="catId" column="catid"/>
            <return-property name="catName" column="catname"/>
            <return-property name="catWidth" column="catwidth" />
        </return>

        {call getCatList()}
```

```
    </sql-query>

</hibernate-mapping>
```

3. Hibernate 执行存储过程查询

```
List list = session.getNamedQuery("getCatList").list();
```

8.7访问 Oracle 中的序列

通过 generator 访问 Oracle 数据库中的序列,并添加到数据库中!

```
<id name="id" column="cat_id" type="string">
        <generator class="native">
                <param name="sequence">cat_sequence</param>
        </generator>
</id>
```

9 Hibernate 对象关系

Hibernate 中的关联映射主要有 3 种：

一对一关联 (one-to-one)

一对多（或多对一）关联 (many-to-one)

多对多关联。 (many-to-many)

每种关联都可以分为单向和双向两种。关联关系映射通常情况是最难配置正确的。在这个部分中，将从单向关系映射开始，然后考虑双向关系映射。

9.1 One To One

9.1.1 理解原理

一对一关系也是比较常见的一种关系，在 Hibernate 中可以分为单向一对一关系和双向一对一关系。分类的原因一般是由于需求决定的，单双向是站在不同的角度去看认为规定的。一对一关系相对来说比较少见。但是在某些时候也会用到。

比如：

学生和学生证、公民与省份证、会员与会员卡这些都是现实生活中一对一的例子。

一对一的关联分为主键关联和外键关联。

（1）主键关联（共享主键方式）：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。这里的关联关系是定义在类的映射文件中，在辅表的 one-to-one 的属性里要 constrained="true"表示受到约束。

（2）外键关联：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。

知识点：

（1）主控方是指拥有主动性，可以找到被控方，通过对主控方的操作能带动被控方的自动操作。

（2）共享主键是指两个表有共同的主键值，如果是单项关联，其中主控方主键不能设置为自动增长，只能依赖于被控方的主键，被控方主键设置为自动增长。（因为是通过主控方去找被控方，当主控方被实例化后，意味着主控方拥有了被控方对象的实例，因此被控方必须先拥有主键值）

（3）单向关联是指在主控方的持久类和映射文件中进行关联设置，被控方的持久类和映射文件保持原来的设置。

9.1.2 表结构

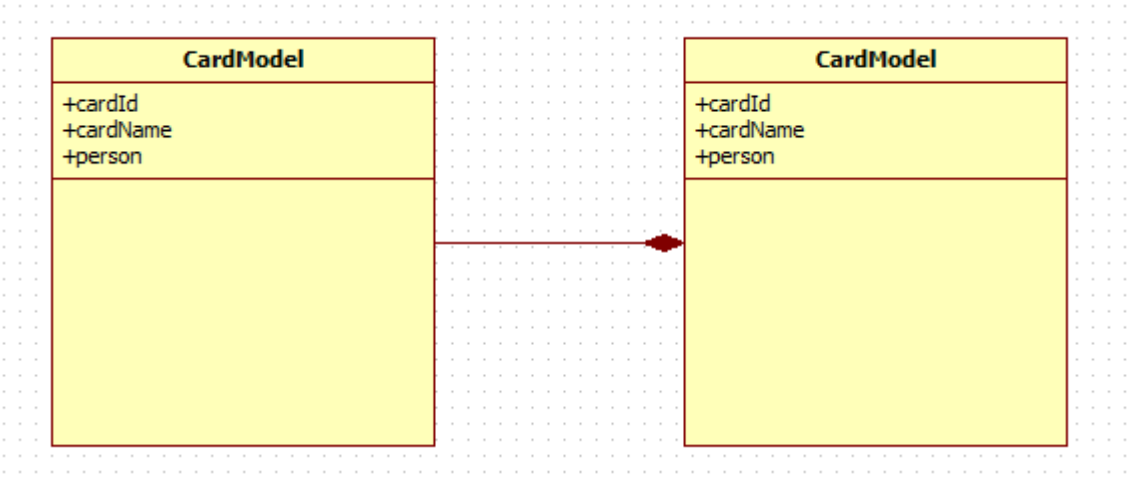
表 1：TBL_PERSON（主表）

字段名称	数据类型	主键	外键	描述
PID	NUMBER	是		人员表主键
PNAME	VARCHAR2(25)			人员名称

表 2：TBL_CARD（子表）

字段名称	数据类型	主键	外键	描述
CID	NUMBER	是	是	身份证表主键
CNUM	VARCHAR2(18)			身份证号码

9.1.3 类的关系



PersonModel.java

```
package com.javakc.hibernate.onetoone;

publicclass PersonModel {
    private String personId;
    private String personName;
    private CardModel card;

    public String getPersonId() {
        returnpersonId;
    }
    publicvoid setPersonId(String personId) {
        this.personId = personId;
    }
    public String getPersonName() {
        returnpersonName;
    }
    publicvoid setPersonName(String personName) {
        this.personName = personName;
    }
    public CardModel getCard() {
        returncard;
    }
    publicvoid setCard(CardModel card) {
        this.card = card;
    }
```

```
    }
```

CardModel.java

```
package com.javakc.hibernate.onetoone;

publicclass CardModel {

    private String cid;
    private String cnum;
    private PersonModel pm;

    public String getCid() {
        returncid;
    }
    publicvoid setCid(String cid) {
        this.cid = cid;
    }
    public String getCnum() {
        returncnum;
    }
    publicvoid setCnum(String cnum) {
        this.cnum = cnum;
    }
    public PersonModel getPm() {
        returnpm;
    }
    publicvoid setPm(PersonModel pm) {
        this.pm = pm;
    }
}
```

9.1.4 配置

9.1.4.1 配置文件

PersonModel.hbm.xml

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

```
<hibernate-mapping package="com.javakc.hibernate.onetoone">

    <class name="PersonModel" table="TBL_PERSON" lazy="false">

        <id name="personId" column="PID">
            <generator class="uuid"></generator>
        </id>

        <property name="personName" column="PNAME"></property>

        <one-to-one
            cascade="save-update"
            name="card"
            class="CardModel"></one-to-one>

    </class>

</hibernate-mapping>
```

CardModel.hbm.xml

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

<hibernate-mapping package="com.javakc.hibernate.onetoone">

    <class name="CardModel" table="TBL_CARD">

        <id name="cid" column="CID">
            <generator class="uuid"/>
        </id>

        <property name="cnum" column="CNUM"></property>

        <one-to-one cascade="all" name="pm" class="PersonModel">
        </one-to-one>

    </class>

</hibernate-mapping>
```

9.1.4.2 cascade 属性的作用

CASCADE 在 hibernate 中起到级联保存的作用!

1) none :在保存,更新或删除当前对象时,忽略其他关联的对象,它是 cascade 属性的默认值.

2)save-update : 当通过 Session 的 save() , update() 以及 saveOrUpdate()方法来保存或更新当时对象时,级联保存所有关联的新建的临时对象,并且级联更新所有关联的游离对象.

3)delete : 当通过 Session 的 delete()方法删除当前对象时,级联删除所有关联的对象.

4)all : 包含 save-update 以及 delete 的行文. 此外,对当前对象执行 evict()或 lock()操作时,也会对所关联的持久化对象执行 evict()和 lock()操作.

5)delete-orphan :删除所有和当前对象解除关联关系的对象.

6)all-delete-orphan :包含 all 和 delete-orphan 的行为

9.1.5 各种操作

9.1.5.1 单独新增主表

9.1.5.2 单独新增子表

9.1.5.3 主子表同增

9.1.5.4 单独删除子表

9.1.5.5 主子表同删

9.1.5.6 主子表同改

9.1.5.7 主子表关联查询

查询张三的会员卡信息：

9.2 One To Many

在学习 Hibernate 的时候，很大一部分任务量是配置实体映射关系，Hibernate 的映射关系的关键就是掌握面向对象的思想，搞清楚实体之间的关系。每一个实体关系都对应这 UML 中的对象关系。我们配置这些对象的关系模型。下面分节讲述 Hibernate 的关系映射。

9.2.1 理解原理

本节主要讲述 Hibernate 的一和多的关系。之所以称之为一和多的关系，是因为他包括三种关系：单向一对多，单向多对一，双向一对多。我这里统称为它一对多。一对多关系是非常重要的关系，也是现实世界中最多的关系。这三个关系对应的是 UML 中的关联关系，也可以分成聚合和组合。

比如：

学生和班级、员工和部门、人员和订单等等都是一对多的例子。

之所以把一对多分成三种类型，是因为它们站的角度不同。这里就拿学生和班级的关系举例，站在学生的角度看这个关系是多对一，站在班级的角度看是一对多。但是为了更好的使用这个关系，也考虑到实际操作的原因。一般把它们设置成双向一对多。

9.2.2 表结构

表 1：TBL_DEPT

字段名称	数据类型	主键	外键	描述

DID	VARCHAR2(32)	是		部门表主键
DNAME	VARCHAR2(25)			部门名称

表 2：TBL_EMP

字段名称	数据类型	主键	外键	描述
EID	VARCHAR2(32)	是		员工表主键
ENAME	VARCHAR2(18)			员工名称
DID	VARCHAR2(32)		是	部门表主键

9.2.3 类的关系

DeptModel.java

```
import java.util.HashSet;
import java.util.Set;

public class DeptModel {
    private String did;
    private String dname;
    private Set<EmpModel> emp = new HashSet<EmpModel>();

    public String getDid() {
        return did;
    }

    public void setDid(String did) {
        this.did = did;
    }

    public String getDname() {
        return dname;
    }

    public void setDname(String dname) {
        this.dname = dname;
    }

    public Set<EmpModel> getEmp() {
        return emp;
    }
```

```
    public void setEmp(Set<EmpModel> emp) {
        this.emp = emp;
    }
}
```

EmpModel.java

```
package com.javakc.hibernate.onetomany;

publicclass EmpModel {
    private String eid;
    private String ename;
    private DeptModel dept;

    public String getEid() {
        returneid;
    }
    publicvoid setEid(String eid) {
        this.eid = eid;
    }
    public String getEname() {
        returnename;
    }
    publicvoid setEname(String ename) {
        this.ename = ename;
    }
    public DeptModel getDept() {
        returndept;
    }
    publicvoid setDept(DeptModel dept) {
        this.dept = dept;
    }

}
```

9.2.4 配置

9.2.4.1 配置文件

DeptModel.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.javakc.hibernate.onetomany">

<class name="DeptModel" table="TBL_DEPT">

        <id name="did" column="DID">
            <generator class="uuid"></generator>
        </id>

        <property name="dname" column="DNAME"></property>

        <set name="emp" cascade="all" lazy="false">
            <key column="DID"></key>
            <one-to-many class="EmpModel"/>
        </set>

    </class>

</hibernate-mapping>
```

EmpModel.hbm.xml

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

<hibernate-mapping package="com.javakc.hibernate.onetomany">

<class name="EmpModel" table="TBL_EMP">

        <id name="eid" column="EID">
            <generator class="uuid"></generator>
        </id>

        <property name="ename" column="ENAME"></property>

        <many-to-one
            name="dept"
            lazy="false"
            cascade="save-update"
            column="DID"
            class="DeptModel"></many-to-one>

    </class>
```

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

9.2.5 各种操作

9.2.5.1 单独新增主表

9.2.5.2 单独新增子表

9.2.5.3 主子表同增

9.2.5.4 单独删除子表

9.2.5.5 主子表同删

9.2.5.6 主子表同改

9.2.5.7 主子表关联查询

9.3Many To Many

9.3.1 理解原理

Hibernate 多对多关联也是比较常见的一种。对于多对多关系，我们都是采用引入第三方表来描述它们之间的关联的。本节主要讲述一下 Hibernate 多对多关联。多对多关联根据需求也可以分为单向多对多和双向多对多。这里用比较常见的多对多关系用户与角色的关系来举例。

比如：

学生与课程、学生与教师等等这些都是多对多的列子。

9.3.2 表结构

表 1：TBL_STUDENT

字段名称	数据类型	主键	外键	描述
SID	VARCHAR2(32)	是		学生表主键
SNAME	VARCHAR2(25)			学生名称

表 2：TBL_COURSE

字段名称	数据类型	主键	外键	描述
CID	VARCHAR2(32)	是		课程表主键
CNAME	VARCHAR2(18)			课程名称

表 3：TBL_RELATION

字段名称	数据类型	主键	外键	描述
SID	VARCHAR2(32)	是	是	学生表主键
CID	VARCHAR2(18)	是	是	课程表主键

9.3.3 类的关系

StudentModel.java

```
package com.javakc.hibernate.manytomany;
```

```
import java.util.Set;

publicclass StudentModel {
    private String sid;
    private String sname;
    private Set<CourseModel>course;

    public String getSid() {
        returnsid;
    }
    publicvoid setSid(String sid) {
        this.sid = sid;
    }
    public String getSname() {
        returnsname;
    }
    publicvoid setSname(String sname) {
        this.sname = sname;
    }
    public Set<CourseModel> getCourse() {
        returncourse;
    }
    publicvoid setCourse(Set<CourseModel> course) {
        this.course = course;
    }

}
```

CourseModel.java

```
package com.javakc.hibernate.manytomany;

import java.util.Set;

publicclass CourseModel {
    private String cid;
    private String cname;
    private Set<StudentModel>student;

    public String getCid() {
        returncid;
    }
    publicvoid setCid(String cid) {
        this.cid = cid;
    }
    public String getCname() {
```

```
        returncname;
    }
    publicvoid setCname(String cname) {
        this.cname = cname;
    }
    public Set<StudentModel> getStudent() {
        returnstudent;
    }
    publicvoid setStudent(Set<StudentModel> student) {
        this.student = student;
    }

}
```

9.3.4 配置

9.3.4.1 配置文件

StudentModel.hbm.xml

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

<hibernate-mapping package="com.javakc.hibernate.manytomany">

<class name="StudentModel" table="TBL_STUDENT">

        <id name="sid" column="SID">
            <generator class="uuid"/>
        </id>

        <property name="sname" column="SNAME"></property>

        <set name="course" cascade="all" inverse="false" lazy="false"
table="TBL_SC">
            <key column="SID"></key>
            <many-to-many column="CID"
class="CourseModel"></many-to-many>
        </set>

    </class>

</hibernate-mapping>
```

CourseModel.hbm.xml

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

<hibernate-mapping package="com.javakc.hibernate.manytomany">

<class name="CourseModel" table="TBL_COURSE">

        <id name="cid" column="CID">
            <generator class="uuid"/>
        </id>

        <property name="cname" column="CNAME"></property>

        <set name="student" table="TBL_SC">
            <key column="CID"></key>
            <many-to-many column="SID"
class="StudentModel"></many-to-many>
        </set>

    </class>

</hibernate-mapping>
```

9.3.4.2 inverse 属性的作用

inverse 在 hibernate 中起到的作用是维护中间表关系!它是用来指定关联的控制方的。inverse 属性默认是 false，若为 false，则关联由自己控制，若为 true，则关联由对方控制。

9.3.5 各种操作

9.3.5.1 单独新增学生和课程

场景：

9.3.5.2 学生和课程同增

9.3.5.3 单独删除学生

考虑单独删除学生后，关系表的处理

9.3.5.4 学生和课程同删

9.3.5.5 单独修改学生

考虑单独修改学生后，关系表的数据

9.3.5.6 学生和课程同改

9.3.5.7 关联查询

查询张三选修的课程

查询选修计算机原理的学生

10 Hibernate 与事务

10.1 数据库事务

数据库事务是指由一个或多个 SQL 语句组成的工作单元，这个工作单元中的 SQL 语句相互依赖，如果有一个 SQL 语句执行失败，就必须撤销整个工作单元。

数据库事务必须具备 ACID 特征：

- Atomic 原子性，整个事务不可分割，要么都成功，要么都撤销。
- Consistency 一致性，事务不能破坏关系数据的完整性和业务逻辑的一致性。例如转账，应保证事务结束后两个账户的存款总额不变。
- Isolation 隔离性，多个事务同时操纵相同数据时，每个事务都有各自的完整数据空间。

- Durability 持久性，只要事务成功结束，对数据库的更新就必须永久保存下来，即使系统发生崩溃，重启数据库后，数据库还能恢复到事务成功结束时的状态。

只要声明了一个事务，数据库系统就会自动保证事务的 ACID 特性。

6.2 事务边界

事务的开始结束：

事务的正常结束边界(commit):提交事务，永久保存

事务的异常结束边界(rollback):撤销事务，数据库回退到执行事务前的状态

事务的两种模式：

自动提交模式：每个 SQL 语句都是一个独立的事务，数据库执行完一条 SQL 语句后，会自动提交事务。

手工提交模式：必须由数据库的客户程序显式指定事务的开始和结束边界。

一个 session 可以对应多个事务，但是应优先考虑让一个 session 只对应一个事务，当一个事务结束或撤销后，就关闭 session.

不管事务成功与否，最后都应调用 session 的 close 关闭 session

任何时候一个 session 只允许有一个未提交的事务，不能同时开始两个事务

6.3 多事务并发问题

同时运行多个事务访问相同数据时，可能会导致 5 类并发问题：

名称	出现问题
第一类丢失更新	撤销一个事务时，把其他事务已提交的更新覆盖
脏读	一个事务读到另一事务未提交的更新数据
虚读(幻读)	一个事务读到另一事务已提交的新插入的数据
不可重复读	一个事务读到另一事务已提交的更新数据
第二类丢失更新	一个事务覆盖另一事务已提交的更新数据，不可重复读的特例

第一类丢失更新:

第一类丢失更新（Lost Update）

时间	取款事务A	存款事务B
T1	**开始事务**	
T2		**开始事务**
T3	查询账户余额为1000元	
T4		查询账户余额为1000元
T5		汇入100元把余额改为1100元
T6		**提交事务**
T7	取出100元把余额改为900元	
T8	**撤销事务**	
T9	**余额恢复为1000元（丢失更新）**	

脏读：

脏读（dirty read）

时间	取款事务A	转账事务B
T1	**开始事务**	
T2		**开始事务**
T3	查	查询账户余额为1000元
T4		汇入100元把余额改为1100元
T5	查询账户余额为1100元（脏数据）	
T6		**回滚**
T7	取款1100	
T8	**提交事务失败**	

虚读：

幻读（phantom read）

时间	查询学生事务A	插入新学生事务B
T1	**开始事务**	
T2		**开始事务**
T3	查询学生为10人	
T4		插入一个新学生
T5	查询学生为11人	
T6		**提交事务**
T7	**提交事务**	

不可重复读：

不可重复读（non-repeatable read）

时间	取款事务A	转账事务B
T1	**开始事务**	
T2		**开始事务**
T3	查询账户余额为1000元	
T4		汇入100元把余额改为1100元
T5		**提交事务**
T6	查询账户余额为1100元	
T7	**提交事务**	

第二类丢失更新：

第二类丢失更新（不可重复读的特殊情况）

时间	转账事务A	取款事务B
T1		**开始事务**
T2	**开始事务**	
T3		查询账户余额为1000元
T4	查询账户余额为1000元	
T5		取出100元把余额改为900元
T6		**提交事务**
T7	汇入100元	
T8	**提交事务**	
T9	**把余额改为1100元（丢失更新）**	

6.4 数据库锁

数据库必须具有隔离并发运行的各个事务的能力，数据库采用锁来实现事务的隔离性。

根据数据库能够锁定的资源，可分为以下锁：

数据库级锁、区域级锁、表级锁、行级锁

锁的封锁粒度越大，隔离性越高，并发性越低

锁升级指调整锁的粒度，将多个低粒度锁替换成更高粒度的锁，以降低系统负荷。

按照封锁程度，锁可分为：

共享锁：用于读数据，非独占，允许其他事务同时读取锁定的资源，但不允许其他事务更新。

加锁：执行 select 语句时，数据库会为事务分配共享锁，锁定被查询的数据。

解锁：数据被读取后，立即解除

兼容性：还可再放置共享锁和更新锁

如果使用共享锁，更新数据的操作分为两步：

1.获得一个共享锁，读取一条记录

2.将共享锁升级为独占锁，再执行更新操作

如果同时多个事务同时更新该数据，每个事务都先获得一把共享锁，在更新数据的时候，这些事务都要先将共享锁升级为独占锁。由于独占锁不能与其他锁并存，因此每个事务都进入等待，等待其他事务释放共享锁，造成了死锁。

更新锁：在更新操作的初始化阶段用来锁定可能要修改的资源，避免使用共享锁造成的死锁现象。

加锁：一个事务执行 update 语句时，数据库先为事务分配一把更新锁

解锁：读取数据完毕，执行更新操作时，会把更新锁升级为独占锁。

兼容性：更新锁与共享锁是兼容的，可以同时放置，但是最多只能放一把更新锁，以确保多个事务更新数据时，只有一个事务能获得更新锁，再把更新锁升级为独占锁，其他事务必须等到前一个事务结束后，才能获得更新锁，这就避免了死锁。

如果使用更新锁，更新数据的操作分为以下两步：

1.获得一个更新锁，读取一条记录

2.将更新锁升级为独占锁，再执行更新操作

并发性能: 允许多个事务同时读锁定的资源，但不允许其他事务修改它。

独占锁：也叫排他锁，用于修改数据，锁定的资源不能被其他事务读取和修改。

加锁：一个事务执行 insert update delete 语句时，自动使用独占锁，若已有其他锁存在，无法再独占锁

解锁：一直事务结束才能被解除

兼容性：不能和其他锁兼容，不能再放置其他任何锁。

并发性能：较差，只允许有一个事务访问锁定的数据，其他事务要访问，必读等待，直到前一个事务结束，解除了独占锁。

死锁：

死锁是指多个事务分别锁定了一个资源，又试图请求锁定对方已经锁定的资源，这就产生了死锁一个锁定请求环，导致多个事务都处于等待对方释放锁定资源的状态。

预防死锁：

合理安排表访问顺序

使用短事务

允许脏读

错开执行时间

使用尽可能低的事务隔离级别

6.5 事务隔离级别

锁机制能解决各种并发问题，但是会影响并发性能，为了能让用户根据实际应用的需要，在

事务的隔离性和并发性之间做出合理权衡，数据库系统提供了四种事务隔离级别供用户选择：

隔离级别	编号	作用
Serializable	8	串行化 完全看不到其他事务的更新，串行等待
Repeatable Read	4	可重复读 事务可看到其他事务已提交的新插入记录，但是不能看到其他事务对已有记录的更新
Read Commited	2	读已提交数据 事务可看到其他事务已提交的新插入记录，还能看到其他事务已经提交的对已有记录的更新
Read Uncommited	1	读未提交数据 事务可看到其他事务没有提交的新插入记录，还能看到其他事务没有提交的对已有记录的更新

优先考虑隔离级别为 Read Commited，能避免脏读，有较好的并发性能，可能会导致不可重复读、虚读和第二类丢失更新，但是可以采用悲观锁或乐观锁来控制。

Hibernate 中设置事务隔离级别：

```
hibernate.connection.isolation = 2     (Read Commited)
```

6.6 乐观锁与悲观锁

悲观锁：先锁定资源，能防止丢失更新和不可重复读问题，但是影响并发性能

乐观锁：完全依靠数据库的隔离级别来自动管理锁的工作，采用版本控制可避免并发问题。

悲观锁的实现：

显式指定独占锁来锁定资源

select 语句默认采用共享锁

可采用 select .. for update 来显式指定采用独占锁来锁定查询的记录

Hibernate 采用 LockMode 来实现锁定模式

2. 增加一个 LOCK 字段

Hibernate 采用<version>和<timestamp>来实现版本控制

乐观锁的实现：三种方式

1）Version 版本号

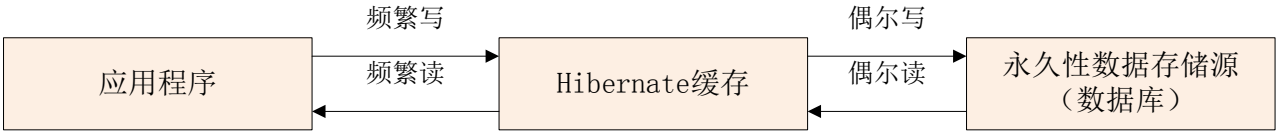
2）时间戳

3）自动版本控制。

11 Hibernate 缓存

11.1 Hibernate 缓存概述

Hibernate 缓存机制对 Hibernate 的性能发挥一直处于一个极其重要的作用，它是持久层性能提升的关键。



hibernate 缓存在应用系统中的位置

Hibernate 缓存介于 Hibernate 应用和数据库之间，缓存中存放了数据库数据的拷贝。其作用是减少访问数据库的频率，从而提高应用的运行性能。

Hibernate 在进行读取数据的时候，根据缓存机制在相应的缓存中查询，如果在缓存中找到

了需要的数据(我们把这称做“**缓存命中**”)，则就直接把命中的数据作为结果加以利用，避免的了建立数据库查询的性能损耗。

11.2 Hibernate 缓存分类

Hibernate 提供了两级缓存：

一级缓存： Session 级别的缓存

二级缓存： SessionFactory 级别的全局缓存

Hibernate 的这两级缓存都位于持久化层，存放的都是数据库数据的拷贝。

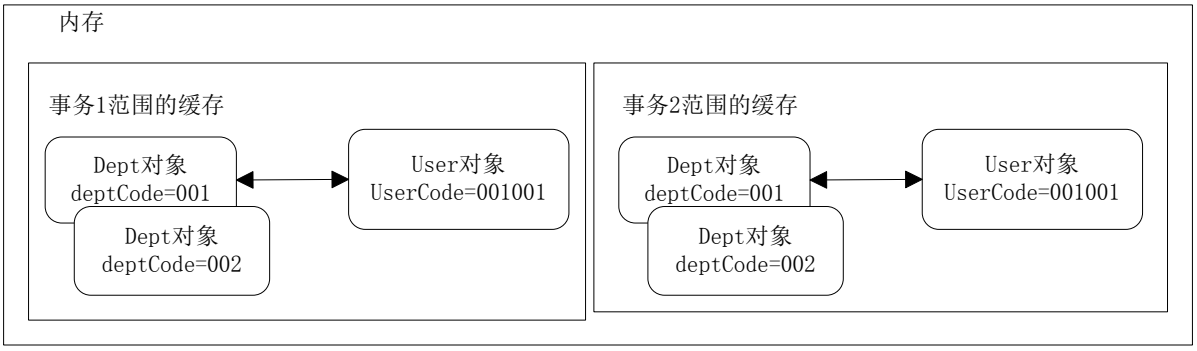
那么它们之间的区别是什么呢？

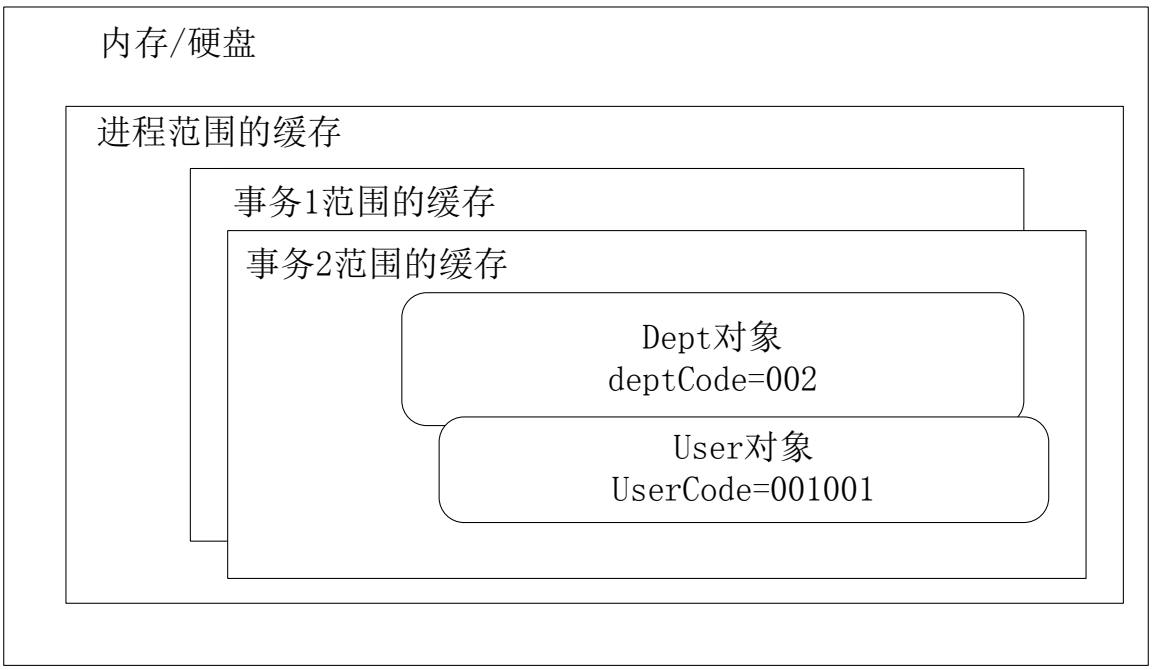
为了理解二者的区别，需要深入理解持久化层的缓存的一个特性：**缓存的范围**。

11.3 Hibernate 缓存范围

缓存的范围决定了缓存的生命周期以及可以被谁访问。缓存的范围分为三类。

事务范围 缓存只能被当前事务访问。缓存的生命周期依赖于事务的生命周期，当事务结束时，缓存也就结束生命周期。缓存的介质是内存。事务可以是数据库事务或者应用事务，每个事务都有独自的缓存，缓存内的数据通常采用相互关联的的对象形式。**一级缓存就属于事务范围。**





进程范围 缓存被进程范围内的所有事务共享。这些事务有可能并发访问缓存，因此必须对缓存采取必要的事务隔离机制。缓存的生命周期依赖于进程的生命周期，进程结束时，缓存也就结束了生命周期。它的物理介质可以是内存或硬盘。

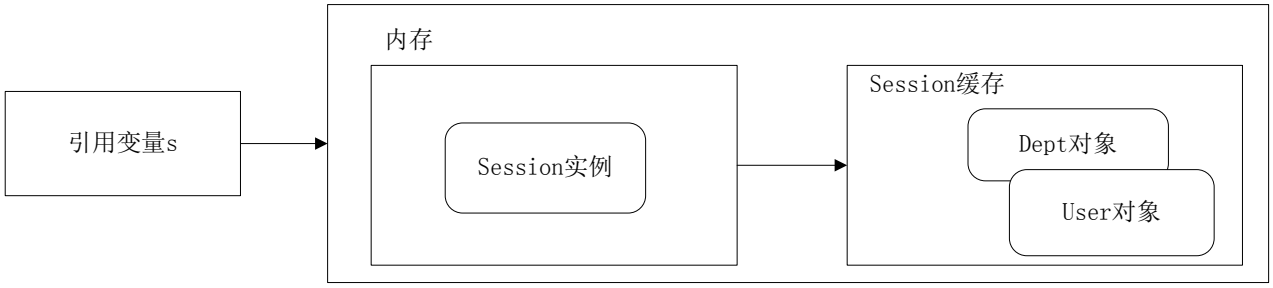
集群范围 在集群环境中，缓存被一个机器或者多个机器的进程共享。缓存中的数据被复制到集群环境中的每个进程节点，进程间通过远程通信来保证缓存中的数据的一致性，缓存中的数据通常采用对象的松散数据形式。

持久化层的第二级缓存就存在于进程范围或集群范围。

11.4 Hibernate 一级缓存

Session 具有一个缓存，是一块内存空间，在这个内存空间存放了相互关联的 java 对象，这种位于 Session 缓存内的对象也被称为持久化对象，Session 负责根据持久化对象的状态变化来同步更新数据库。

Session 的缓存是内置的，不能被卸除的，也被称为 Hibernate 的第一级缓存。在正常的情况下一级缓存是由 Hibernate 自动维护的，无需人工干预。

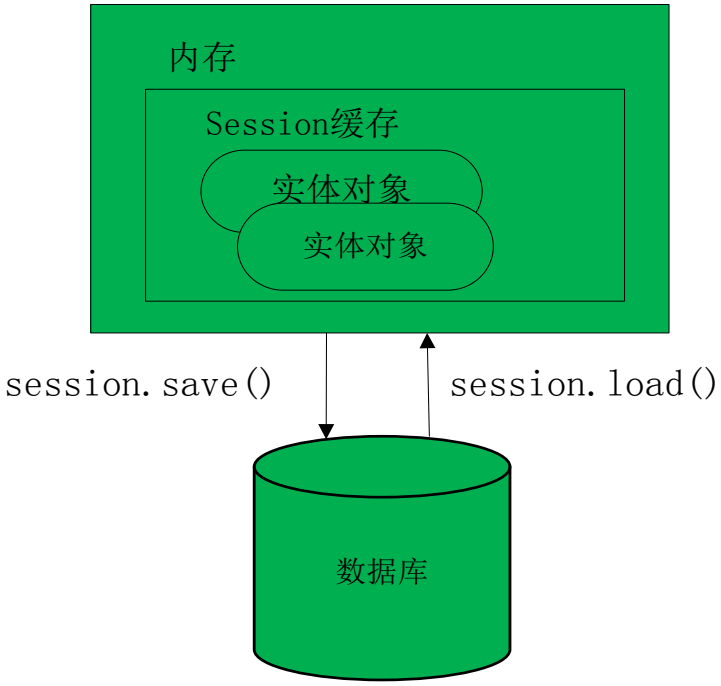


session 缓存中对象的生命周期依赖 session 实例

（1）当应用程序调用 Session 接口的 save()、update()、saveOrUpdate()时，如果 Session 缓存中还不存在相应的对象，Hibernate 就会把该对象加入到第一级缓存中。

（2)当调用 Session 接口的 load()、get()以及 Query 查询接口的 list()、iterator()方法时，如果 Session 缓存中存在相应的对象，就不需要到数据库中检索。

（3)当调用 Session 的 close()时， Session 缓存就被清空。



11.5 Hibernate 二级缓存

因为 Session 的生命期往往很短，存在于 Session 内部的第一级缓存的生命期当然也很短

，所以第一级缓存的命中率是很低的。其对系统性能的改善也是很有限的。当然，这个 Session 缓存的主要作用是保持 Session 内部数据状态同步，并非是 hibernate 为了大幅提高系统性能所提供的。

Hibernate 二级缓存特点：

SessionFactory 级别的缓存是全局缓存

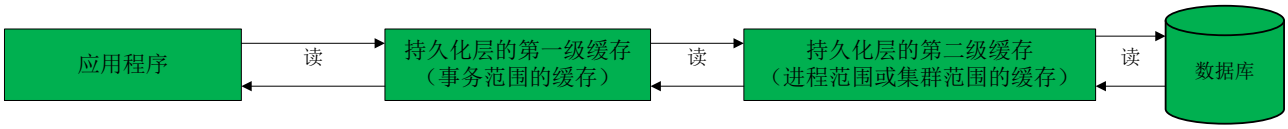
可配置可插拔的缓存插件

涵盖了进程范围与集群范围

物理介质：内存或硬盘

Hibernate 的两级缓存机制：

如果在一级缓存中没有查询到相应的数据，可以到二级缓存中查找，如果在二级缓存中也没有找到数据，那么就只好查询数据库了。



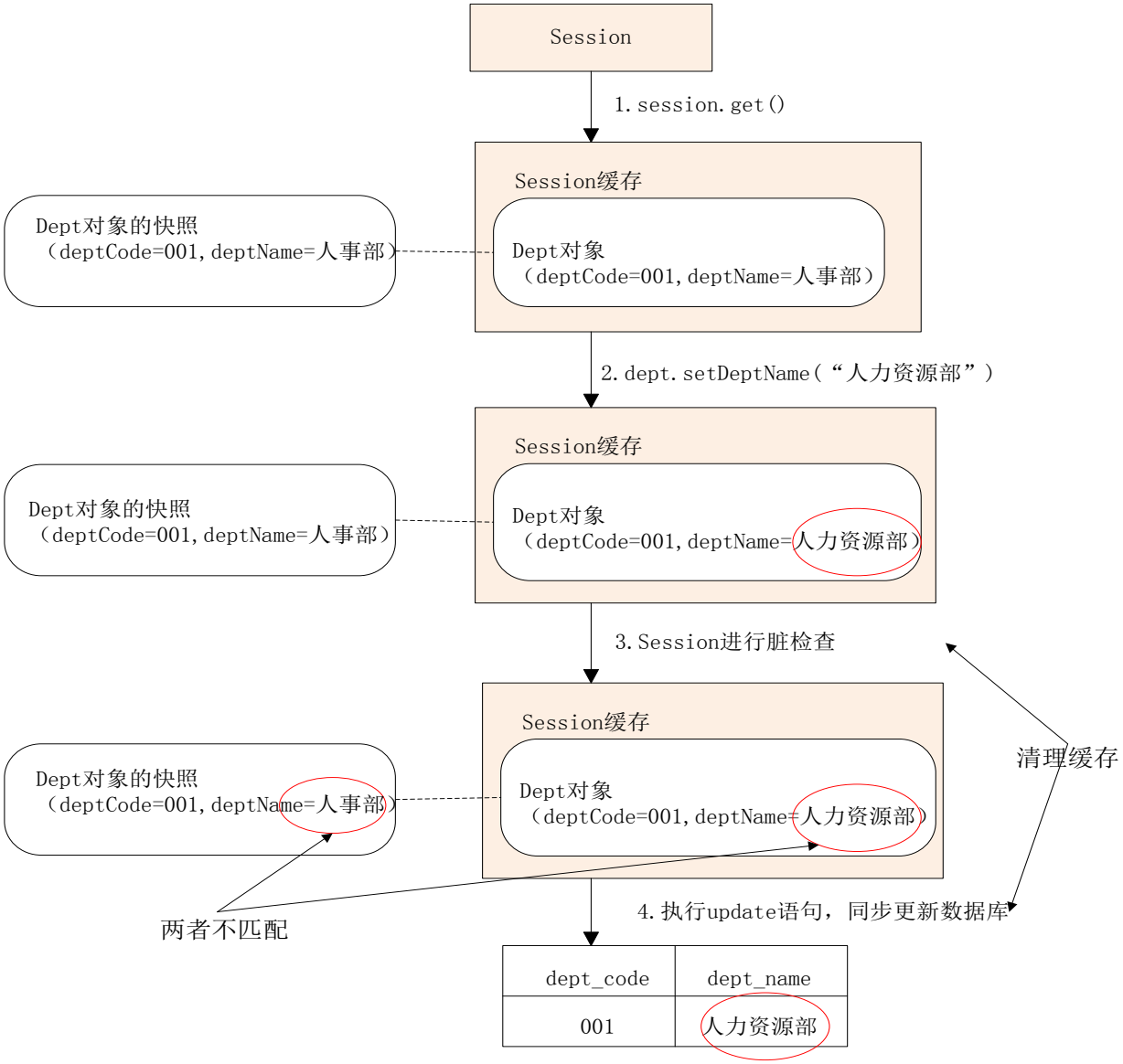
11.6 Hibernate 清理缓存

清理缓存是指 Session 按照缓存中对象的属性变化来同步更新数据库。

Session 在清理缓存的时候会自动进行**脏检查**（dirty-check），如果发现 Session 缓存中的对象与数据库中相应的记录不一致，就会同步数据库。

Session 是如何进行脏检查的呢？

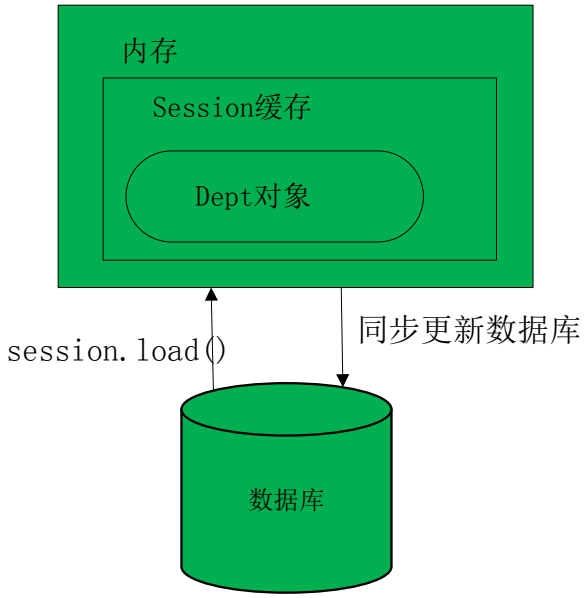
当一个对象被加入到 Session 缓存时，Session 会为该对象的值类型的属性复制一份快照。当 Session 清理缓存的时候，会进行脏检查，即比较对象的当前属性与它的快照，来判断对象的属性是否发生变化，如果发生变化，就称这个对象是“脏对象”，Session 会根据脏对象的最新属性来执行相关的 SQL 语句，从而同步更新数据库。



session 缓存中对象的属性每次发生变化，Session 不会立即清理缓存及执行相关的 update 语句，而是在特定的时间点才清理缓存，这使得 Session 能够把几条相关的 sql 语句合并为一条 sql 语句，以便减少访问数据库的次数。

以下代码对 dept 对象的 deptName 属性修改了两次：

```
Transaction tx = session.biginTransaction();
Dept   dept = (Dept)session.get(Dept.class,”001”);
dept.setDeptName(“人事部”);
dept.setDeptName(“人力资源部”);
tx.commit();
```



当 Session 清理缓存时，只会执行一条 update 语句

session 会在下面的时间点清理缓存：
(1) 当应用程序调用 org.hibernate.Transaction 的 commit()方法的时候 commit()方法先清理缓存，然后再向数据库提交事务。

(2) 当应用程序执行一些查询操作时，如果缓存中持久化对象的属性已经发生变化，就会先清理缓存，使得 Session 缓存与数据库进行了同步，从而保证查询结果返回的是正确的数据。

(3) 当应用程序显式调用 Session 的 flush()方法的时候

11.7 Sesion 缓存的管理

第一级缓存在正常的情况下是由 Hibernate 自动维护的。

在特殊的情况下需要我们进行手动维护，Hibernate 就提供了两个管理 Session 缓存的方法：

(1)Session.evict(Object o)

将某个特定的对象从缓存中清除，

使用此方法有两种适用情形，一是在特定的操作（如批量处理），需要及时释放对象占用的内

存。二是不希望当前 Session 继续运用此对象的状态变化来同步更新数据库。

(2)Session.clear()

清除缓存中的所有持久化对象。

1、在多数情况下并不提倡通过 EVIT()和 CLEAR()来管理一级缓存。

2、管理一级缓存最有效的方法是采用合理的检索策略和检索方式来节省内存的开销。

12 Hibernate 的优缺点

12.1 优点

生产性

与持续性有关的代码可能是 Java 应用中最乏味的代码。Hibernate 去掉了很多让人心烦的

工作（多于你的期望），让你可以集中到业务问题上。不论你喜欢哪种应用开发策略——自顶向下，从域模型开始；或者自底向上，从一个现有的数据库模式开始——使用 Hibernate 和适当的工具将会减少大量的开发时间。

可维护性

更少的代码行数（LOC）使系统更容易理解因为它们强调了业务逻辑而不是管道设备。更重要的，一个系统包含的代码越少则越容易重构。自动的对象-关系持续性充分地减少了 LOC。当然，统计代码行数是度量应用复杂性的值得争议的方式。然而，Hibernate 应用更容易维护有其它方面的原因。在手工编码的持续性系统中，关系表示和对象模型之间存在一种不可避免的紧张。改变一个几乎总是包含改变其它的。并且一种表示设计经常需要妥协来适应其它的存在（实际上几乎总是发生的是域对象模型进行妥协）。ORM 在这两种模型之间提供了一个缓冲，允许 Java 一方更优雅地进行面向对象的使用，并且每个模型都对其它模型的轻微改动进行了绝缘。

性能

一个共同的断言是手工编码的持续性与自动的持续性相比可能至少一样快，并且经常更快一些。在相同的意义上：汇编代码至少与 Java 代码一样快，或者手工编写的解析器至少与 YACC 或 ANTLR 生成的一样快，这的确是真的——换句话说，这有点离题了。这种断言未明确说明的含意是在实际的应用中手工编码的持续性至少应该完成得一样好。但这种含意只有在实现至少一样快的手工编码的持续性所需的努力与利用自动的解决方案包含的努力相似的情况下才是对的。实际令人感兴趣的问题是，当我们考虑时间和预算的限制时会发生什么？对一项给定的持续性任务，可以进行许多优化。许多（例如查询提示）通过手工编码的 SQL/JDBC 也很容易完成，然而，使用自动的 ORM 完成会更简单。在有时间限制的项目中，手工编码的持续层通常允许你利用一点时间做一些优化。Hibernate 则允许你在全部的时间内做更多的优化。另外，自动的持续性给开发者提供了如此高的生产性因此你可以花更多的时间手工优化一些其余的瓶颈。

厂商独立性

ORM 抽象了你的应用使用下层 SQL 数据库和 SQL 方言的方式。如果工具支持许多不同的数据库（大部分如此），那么这会给你的应用带来一定程度的可移植性。你不必期望可以达到“一次编写，到处运行”，因为数据库的性能不同并且达到完全的可移植性需要牺牲更强大的平台的更多的力气。然而，使用 ORM 开发跨平台的应用通常更容易。即使你不需要跨平台操作，ORM 依然可以帮你减小被厂商锁定的风险。另外，数据库独立性对这种开发情景也有帮助：开发者使用一个轻量级的本地数据库进行开发但实际产品需要配置在一个不同的数据库上。

提高生产力，代码更加简练了，不用再写枯燥的 jdbc 语句了。

开发更对象化了，只需操作对象就 ok 了，抛弃了数据库中心的思想，完全的面向对象思想。

移植性，不涉及 sql 语句，sql 虽然是标准的，但是各个数据库厂商会有不同，使用 hibernate 标准 API,会自动转换成相应的数据库 sql 语句，关键是 hibernate 配置文件中方言的配置。

支持透明持久化，hibernate 的 API 没有侵入性，对于实体类来说，既可用于 JDBC，也可

用于 hibernate。

事务

应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短。它通过抽象将应用从底层具体的 JDBC、JTA 以及 CORBA 事务隔离开。某些情况下，一个 Session 之内可能包含多个 Transaction 对象。尽管是否使用该对象是可选的，但无论是使用底层的 API 还是使用 Transaction 对象，事务边界的开启与关闭是必不可少的。

缓存机制，提供一级缓存和二级缓存

简洁的 HQL 编程

12.2 缺点

Hibernate 在批量数据处理时有弱势

针对单一对象简单的增删查改，适合于 Hibernate,而对于批量的修改，删除，不适合用 Hibernate,这也是 OR 框架的弱点；要使用数据库的特定优化机制的时候，不适合用 Hibernate