**Hibernate总结**

**1.**[**hibernate**](http://lib.csdn.net/base/javaee)**框架作用，优点**

(1)、什么是Hibernate

       Hibernate是一个数据访问框架(持久层框架),在项目中利用Hibernate框

       架可以实现对[**数据库**](http://lib.csdn.net/base/mysql)的增删改查操作，为业务层构建一个持久层。

(2)、Hibernate框架主要用于对数据库的操作。

    使用该框架可以简化数据操作代码，程序员可以将更多地精力放在业务

       编写上。Hibernate经典总结

     Hibernate本质上是对JDBC技术的封装。

    Hibernate和JDBC的关系类似于[**jQuery**](http://lib.csdn.net/base/jquery)和[**JavaScript**](http://lib.csdn.net/base/javascript)的关系。

   (3)、原有JDBC方式访问数据库，有以下几点不足：

   a.需要编写大量复杂的SQL语句

   b.需要做大量的对象和记录的转换

       c.数据库移植时，需要修改SQL语句。（非标准SQL语句：如分页语句，

使用数据库函数的语句）

使用Hibernate框架后，可以解决上述问题

   (4)、Hibernate框架的优点：

a.无需编写大量复杂的SQL语句

b.程序中的实体对象和数据库中的数据实现自动映射转换

c.方便数据库的移植

**2.Hibernate设计原理**

   Hibernate框架是一款ORM工具。基于ORM设计思想开发出来的。

   ORM：Object--Relation--Mapping对象关系映射

         对象指的就是[**Java**](http://lib.csdn.net/base/javase)的实体对象;

         关系指的是关系型数据库。（[**Oracle**](http://lib.csdn.net/base/oracle)、DB2、[**MySQL**](http://lib.csdn.net/base/mysql)、SqlServer）

         ORM的主要思想就是将程序中的对象和数据库中的数据实现自动映射

         转换。利用ORM工具，在查询时，可以自动将记录封装成Java对象返

         回。在更新、插入操作时，可以将对象自动写入数据表。对于中间的

         SQL+JDBC操作细节，完全封装在工具底层

  基于ORM思想设计的框架有很多，例如Hibernate，iBATIS，JPA等.

**\*3.Hibernate框架主要结构**

    Hibernate主要由以下几部分构成：

   a.Java实体类（1-n个）

     与数据表对应，用于封装数据表的一行记录。

   b.hibernate.cfg.xml（1个）

      Hibernate主配置文件，里面主要定义连接数据库的参数、框架参数等。

<hibernate-configuration>

<session-factory>

<!-- 方言设置,hibernate根据方言类型自动生成相应数据库的sql语句

<property name="dialect">org.hibernate.dialect.OracleDialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1522:xe

        </property>

<property name="connection.username">system</property>

<property name="connection.password">zhuxun</property>

<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver

</property>

<!-- 将sql语句从控制台打印出来 -->

<property name="show\_sql">true</property>

<property name="format\_sql">true</property><!-- 格式化sql语句 -->

<!-- 加载映射描述文件 -->

<mapping resource="entity/Cost.hbm.xml" />

</session-factory>

</hibernate-configuration>

   c. 文件名.hbm.xml（1-n个）

      Hibernate映射描述文件，里面定义了实体类和数据库之间的对应关

      系。例如定义了java实体类和数据表，属性和表字段之间的对应关系。

<hibernate-mapping>

<!-- Hibernate映射描述文件定义了实体类和数据库之间的对应关 系。例

         如定义了java类和数据表， 属性和表字段之间的对应关系。

 -->

<class name="entity.Cost" table="Cost">

<!-- 定义主键映射 -->Hibernate经典总结

<id name="id" column="ID" type="integer">

            <!-- 也可写成type="java.lang.Integer"-->

<!-- 指定主键的生成方式为序列方式 -->

<generator class="sequence">

<param name="sequence">cost\_seq</param>

</generator>

</id>

<!-- 非主键映射 -->

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

        </property>

<property name="baseDuration" column="BASE\_DURATION"

           type="integer"></property>

<property name="baseCost" column="BASE\_COST"

           type="double"></property>

<property name="unitCost" column="UNIT\_COST"

           type="double"></property>

</class>

</hibernate-mapping>

   d.Hibernate API

   在使用时，需要使用Hibernate提供的API， 它们将SQL+JDBC操作细节

        封装起来了。

**\*4.Hibernate主要API**

 Configuration：用于加载主配置文件(hibernte.cfg.xml)和映射文件。

 SessionFactory：用于创建Session对象。

 Session：将原Connection对象进行了封装，代表Hibernate与数据库之

       间的一次连接，负责执行增删改查操作。

       save(),update(),delete(),load(),get()方法.

 Transaction：用于进行事务管理。

             注意：由于关闭了JDBC中的自动提交功能

                   (setAutoCommit(false)),所以使用时，必须显示地

                   执行commit 操作。

Query：负责执行各种查询。

注：此处的Session区别于JavaWeb中的Session。前者表示Hibernate与数据库之间的

    连接，后者表示客户端与服务器端的一次会话。

**\*5.Hibernate使用步骤**

   a.创建工程，引入hibernate和驱动开发包

   b.在src下追加hibernate.cfg.xml主配置

   c.根据数据表创建Entity实体类

   d.编写实体类和数据表的映射文件xxx.hbm.xml

   e.利用Hibernate API实现增删改查操作

**6.Hibernate中的增改删查操作：**

/\*\*

 \* [**测试**](http://lib.csdn.net/base/softwaretest)添加操作

 \*/

public static void testSave() {

Cost cost = new Cost();

// cost.setId();//由hbm.xml定义的序列负责

cost.setName("400元套餐");

cost.setBaseDuration(200);

cost.setBaseCost(400.0);

cost.setUnitCost(2.0);

cost.setCostType("1");

cost.setDescr("400特惠大套餐");

cost.setCreateTime(new Date(System.currentTimeMillis()));

cost.setStatus("0");

Session session = HibernateUtil.getSession();

// 开启事务

Transaction tx = session.beginTransaction();

session.save(cost);// 保存

tx.commit();// 提交事务

session.close();// 是否session

}

 Hibernate经典总结

/\*\*

 \* 测试更新操作

 \*/

public static void testUpdate() {

// 开启资费处理，更新status=1，startTime时间

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

// 按ID=189主键做条件查询

Cost cost = (Cost) session.load(Cost.class, 189);

cost.setStatus("1");// 设置为暂停状态

cost.setStartTime(new Date(System.currentTimeMillis()));// 设置为开

                    启时间

session.update(cost);

tx.commit();

session.close();

}

/\*\*

 \* 测试删除操作

 \*/

public static void testDelete() {

Cost cost = new Cost();

cost.setId(189);

// Cost cost = (Cost)session.get(Cost.class, 189);

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

session.delete(cost);// 执行一个delete语句,按ID做条件删除

tx.commit();

session.close();

}

/\*\*

 \* 测试查询操作

 \*/

public static void testFindAll() {

Session session = HibernateUtil.getSession();

// 利用Query执行一个HQL查询语句

Query query = session.createQuery("from Cost");

List<Cost> list = query.list();

for (Cost c : list) {

System.out.println(c.getId() + "  " + c.getName() + "  " +

              c.getBaseDuration());

}

session.close();

}

**7、Hibernate封装JDBC+SQL语句过程：**

Java对象—>Cost—>SQL+JDBC—>DB

insert into cost (NAME, BASE\_DURATION, BASE\_COST, UNIT\_COST, STATUS.) values(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

//java反射技术

Class costClass = Cost.class;

控制台输出的SQL语句：

Hibernate:

    select cost\_seq.nextval from dual

Hibernate:

    insert into Cost

        (NAME, BASE\_DURATION, BASE\_COST, UNIT\_COST, STATUS........)

    values

        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

**8、注意：**

  (1).Hibernate在底层对JDBC语句进行了封装，用rs.getObject(“some”)方式

获取数据。

eg:"BASE\_DURATION"为数据库中表的字段名，类型为Number(5)。如果在表中该字段有null值，那么对于如下两种获取方式将出现不同的结果:

对于rs.getObject("BASE\_DURATION")方式，得到的值为null

对于rs.getInt("BASE\_DURATION")方式，得到的值为0

所以在Hibernate中，根据数据表创建的实体类中不能有基本数据类型，如果为基本类型，rs.getObject(“some”)得到的null值赋给基本类型会发生异常，所以要定义成相应的包装类。

  (2).创建Session对象耗费时间比较多，session对象里面绑定了大量的预编译

     的SQL语句。所以不能每次使用都新建一个对象。

**9、Hibernate数据映射类型：**

  在hbm.xml中，描述属性和字段之间映射的时候，可以使用type属性指定数据

  映射类型。

  type属性可以指定Java类型和Hibernate类型。主要作用是指定属性值和字段

  值之间转换时(即在底层执行rs.getXXX(),stmt.setXXX()时XXX的类型)，采用的

  转换类型。（建议采用Hibernate映射类型）

  Hibernate数据映射类型主要有以下几种:

  a.整数类型

     byte,short,integer,long

  b.浮点数类型

     float,double

  c.字符串类型

     string

  d.日期时间类型

     date,time,timestamp

其中：

date只取年月日，

time只取小时分钟秒，

timestamp取完整的时间年月日小时分钟秒

  e.boolean类型

     该类型可以实现boolean属性和字符之间的转换。

     yes\_no：将true|false转换成Y|N

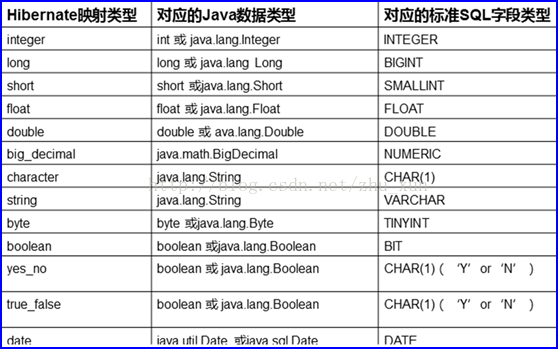
    true\_false：将true|false转换成T|F

    底层实现方法：

stmt.setChar(cost.getBoolean().toString())

cost.setBoolean(Boolean.parseBoolean(rs.getChar("\*\*\*")))

  f.其他big\_decimal,big\_integer,clob,blob



-----------案例-----------------

create table t\_foo(

  t\_id number primary key,

  t\_name varchar2(50),

  t\_salary number(8,2),

  t\_marry char(1),

  t\_hiredate date,

  t\_last\_login\_time date

)

-------------mysql----------------

create table t\_foo(

  t\_id int auto\_increment primary key,

  t\_name varchar(50),

  t\_salary double,

  t\_marry char(1),

  t\_hiredate date,

  t\_last\_login\_time timestamp

) default charset=utf8;

**10.Hibernate主键生成方式**

  在hbm.xml中，可以为主键指定生成方式。具体如下：

  \*a. sequence ：采用指定序列生成，适用于Oracle数据库。使用格式

<generator class="sequence">

    <param name="sequence">foo\_seq</param>

</generator>

 \*b.identity : 采用数据库自增长机制生成。适用于MySQL,SQLServer，DB2数

              据库。

   <generator class="identity">

   </generator>

 \*c.native : 由hibernate决定，hibernate会根据配置文件hibernate.cfg.xml中

             方言<property name="dialect">决定，如果方言是Mysql，相当

             于identity，如果方言是Oracle，相当于是sequence

   <generator class="native">

   </generator>

 \*d.increment : 首先获取最大主键值，然后加1，再执行插入操作。适用于各

                 种数据库。

 先执行select max(id) from t\_foo;

 再执行insert into...

    <generator class="increment">

   </generator>

 e.assigned : Hibernate忽略主键生成，不负责管理。需要程序员在程序中指定

             主键值，不常用

   <generator class="assigned">

   </generator>

  f.其他

 uuid：采用UUID[**算法**](http://lib.csdn.net/base/datastructure)生成一个字符串主键值

 hilo：采用高地位算法生成一个数值主键值

**11.Hibernate框架基本特性**

   1)一级缓存（Session级别缓存,默认开启）

    a.每次创建一个Session对象，会为这个Session对象提供一个缓存区，用于

      缓存该Session查询出来的单个对象。当使用该Session再次查询同一个对

      象时，从缓存取出，避免对数据库的查询。

   \*b.一级缓存区管理的方法：

    session.evict(obj):将obj从一级缓存移除

    session.clear():清除一级缓存所有对象

    c.一级缓存好处

     当利用同一个Session对象多次查询同一个数据对象时，仅第一次从数据库

     查询，后续几次从缓存获取。

-------------------------------------

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

for(int i=1;i<100000;i++){

     User user = new User();

     //......

     session.save(user);

     //20条调用一次flush

     if(i%20==0){

        session.flush();//同步到数据库

        session.clear();//清空缓存

      }

}

tx.commit;

HibernateUtil.close();

对于上面列子，如果不及时清理缓存，将会发生内存溢出异常。

------------------------------------

注：每个session只能访问自己的缓存区，在session1中缓存了一个对象，

   在session2中查询相同的对象，仍然要从数据库中重写查找

/\*\*

 \* 使用两个不同的Session查询同一个对象 结果：去数据库查询2两次

 \*/

public static void test3() {

Session session = HibernateUtil.getSession();

// 第一次查询

Foo foo = (Foo) session.get(Foo.class, 1);

System.out.println(foo.getName() + " " + foo.getSalary());

session.close();

session = HibernateUtil.getSession();

// 第二次查询

Foo foo1 = (Foo) session.get(Foo.class, 1);

System.out.println(foo1.getHireDate() + " " + foo1.getLastLoginTime());

session.close();

}

/\*\*

 \* 同一个session两次查询不同对象 结果：去DB查两次

 \*/

public static void test2() {

Session session = HibernateUtil.getSession();

// 第一次查询

Foo foo = (Foo) session.get(Foo.class, 1);

System.out.println(foo.getName() + " " + foo.getSalary());

// 第二次查询

Foo foo1 = (Foo) session.get(Foo.class, 22);

System.out.println(foo1.getHireDate() + " " + foo1.getLastLoginTime());

session.close();

}

/\*\*

 \* 同一个session两次查询相同对象 结果：去DB查一次

 \*/

public static void test1() {

Session session = HibernateUtil.getSession();

// 第一次查询

Foo foo = (Foo) session.get(Foo.class, 1);

System.out.println(foo.getName() + " " + foo.getSalary());

// session.evict(foo);//从一级缓存移除foo对象

session.clear();

// 第二次查询

Foo foo1 = (Foo) session.get(Foo.class, 1);

System.out.println(foo1.getHireDate() + " " + foo1.getLastLoginTime());

session.close();

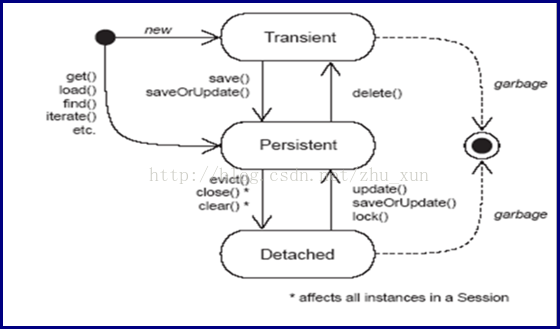
}

   2)对象持久性

     Hibernate框架用于实现对数据库的操作，为应用程序构建一个持久层。

    （由持久对象组成）

     Hibernate使用中，实体对象可以具有以下3种状态。



     a.临时状态（暂时态，Transient）

        采用new方式构建的对象的状态是暂时的，如果没有跟数据库表相关联

        的行为，只要应用程序不再引用这些对象，他们的状态将会丢失，并由

        垃圾回收机制回收。

     \*b.持久状态（Persistent）

如果内存中的对象和数据库的记录有对应关系，即和session对象相关，

则此对象处于Persistent状态，在当事务提交时它们的状态和数据库进行

同步。

采用Session对象查询出来的，受Session对象管理的对象。例如调

load,get,save,update方法后的对象。

        处于持久性的对象特点如下：

      --当事务提交(commit())时，或执行session.flush()方法时，对象的数

           据状态可以更新到数据库。

           commit()前会自动调用session.flush(),

 commit()=session.flush()+commit()

         --对象不能被垃圾回收器回收

        --对象存储在一级缓存中，由Session负责管理

     c.游离/脱管状态（Detached）

Session关闭(session.close())之后，持久化对象就变为detached对象。表

示这个对象不能再与数据库保持同步，它们不再受Hibernate(确切地说

是Session)管理。另外，调用了session.evict(),session.clear()方法后，

一级缓存中的对象就变成了游离状态。

session.close()前会自动调用session.clear(),close()=clear()+close()。

public static void test1() {

Foo foo = new Foo();

foo.setName("scott");

foo.setSalary(2000);

foo.setMarry(true);

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

session.save(foo);// foo变为持久对象

foo.setName("tom");// 修改持久对象name属性

foo.setSalary(3000);// 修改持久对象salary属性

// session.flush();//将缓存中对象数据与数据库同步

tx.commit();// 默认调用session.flush,然后提交事务

session.close();

      }

  \*3)延迟加载（默认启用）

     Hibernate提供一些方法，利用这些方法返回的对象，并没有立刻加载数据

    库的数据。而是在调用对象的getXxx()方法时才触发数据库查询，加载数据

    记录。

其一，如果通过session查询某对象，session将先到缓存中查找是否有查询

的对象，找到则直接取出，否则才查询数据库。

其二，session需要负责实时维护在缓存中的数据，保护缓存中的数据与数据

库中数据的一致性，一旦用户对缓存中的数据做了修改，session负责将数据

更新到数据库中(前提是调用了commit()或flush()方法)。

\*a)延迟加载机制的基本原理：

  当访问实体对象时，并不是立即到数据库中查找。而是在真正要使用实体

 对象的时候，才去数据库查询数据。有部分方法具备这种功能，比如

 session.load(),query.iterator()。

注意：这些方法返回的对象，只有id属性有值，其他属性数据在使用

时候（调用getXxx()方法时）才去获取。

b)延迟加载优点：

使用延迟加载，可以降低用户并发量，减少服务器资源占用。

\*c)哪些方法具有延迟加载

     session.load()

     query.iterator()

     关联属性

   \*d)使用延迟加载方法时，避免出现下面异常LazyInitializationException:

       could not initialize proxy - no Session

     原因：session关闭过早

  session.close()要放在查询之后，否则出现异常

   \*e)get()和load()区别

      相同点：按照主键ID做条件查询某个对象

      不同点如下：

       --load采用了延迟加载机制,get为立刻加载。

       --load如果没有符合记录，会抛出ObjectNotFoundException; 而get方法

         返回的是null,不会出现异常。

         session.load(Cost.class,11235)如果id不存在，则抛出异常

         session.get(Cost.class,11235)为null

       --load返回的是一个动态生成一个类型，get方法返回的是实体类型。

     f)延迟加载实现原理（动态代理技术）

      --采用延迟加载方法后，Hibernate会在底层动态的创建一个新的实体类，

        动态编译成class。比如load方法返回的是动态生成的一个类型（如Cost

        的子类），在这个类里面重写了Cost类的getter方法

  public class Cost$$CGLIB... extends Cost{

      //重写Cost中属性的getter方法

      public String getName(){

           //判断是否有name值，没有查询DB

           return name;

       }

   }

   --Hibernate采用了cglib.jar和asm.jar两个开发包。实现了动态生成新类和编

    译操作。

public static void test3() {

Session session = HibernateUtil.getSession();

Foo foo = (Foo) session.load(Foo.class, 29);

System.out.println(foo.getClass().getName());

session.close();

}

 Hibernate经典总结

public static void test2() {

Session session = HibernateUtil.getSession();

// 延迟加载查询

Foo foo = (Foo) session.load(Foo.class, 29);

session.close();// session关闭过早导致异常

System.out.println("-------------");

System.out.println(foo.getName());// 触发查询

System.out.println(foo.getSalary());

}

public static void test1() {

Session session = HibernateUtil.getSession();

// 延迟加载查询

Foo foo = (Foo) session.load(Foo.class, 1);

System.out.println("-------------");

System.out.println(foo.getName());// 触发查询

System.out.println(foo.getSalary());

session.close();

}

12、OpenSessionInView与ThreadLocal：

(1)、OpenSessionInView

如果通过Hibernate查询到了一条记录(此时并未执行getter方法，由于

延迟加载原因，此时的记录除了主键并不包含任何数据，即没有真正执行查

询)，该记录须在Jsp页面显示（本质上是间接执行了getter方法），那么此时

就需要一种技术实现把Session的关闭延迟到View组件运行完之后。

OpenSessionInView技术可以把Session的关闭延迟到View组件运行完之

  后。

 如果用延迟加载必须使用OpenSessionInView技术，否则在取数据时，

session已经关闭了。

实现OpenSessionInView可以采用的技术如下：

Servlet------过滤器

Struts2------拦截器

[**spring**](http://lib.csdn.net/base/javaee)--------AOP

(2)、ThreadLocal：使用OpenSessionInView必须满足Session的线程单例。

一个线程分配一个Session，在该线程的方法中可以获得该Session，具体使用ThreadLoad，ThreadLocal是一个Map结构，线程ID作为key，拿要存的对象当做value，这样就可以将要存储的数据和线程绑定。

(3)、如何实现OpenSessionInView

方案一:利用ThreadLocal机制手动实现Session与当前线程绑定

  public class HibernateUtil1 {

private static SessionFactory sf;

private static ThreadLocal<Session> sessionLocal = new

ThreadLocal<Session>();

static {

// 创建配置对象

Configuration conf = new Configuration();

// 加载指定的hibernate.cfg.xml配置

conf.configure("/hibernate.cfg.xml");

// 获取SessionFactory

sf = conf.buildSessionFactory();

}

public static Session getSession() {

Session session = sessionLocal.get();

if (session == null) {

session = sf.openSession();

sessionLocal.set(session);

}

return session;

}

public static void closeSession() {

Session session = sessionLocal.get();

sessionLocal.set(null);

if (session != null && session.isOpen()) {

session.close();

}

}

   }

public class OpenSessionInViewInterceptor extends AbstractInterceptor {

@Override

public String intercept(ActionInvocation in) throws Exception {

// 开启事务

Session session = HibernateUtil1.getSession();

Transaction tx = session.beginTransaction();

try {

System.out.println("----开启事务---");

in.invoke();// 执行Action,DAO和Result-->JSP

// 提交事务

tx.commit();

System.out.println("----提交事务---");

return null;

} catch (Exception ex) {

// 回滚事务

tx.rollback();

System.out.println("----回滚事务---");

throw ex;

} finally {

// 关闭session

 HibernateUtil1.closeSession();

System.out.println("----关闭事务---");

}

}

  }

 方案二:利用Hibernate框架封装的机制实现Session与当前线程绑定

public class HibernateUtil2 {

private static SessionFactory sf;

static {

//创建配置对象

Configuration conf = new Configuration();

//加载指定的hibernate.cfg.xml配置

conf.configure("/hibernate.cfg.xml");

//获取SessionFactory

sf = conf.buildSessionFactory();

}

public static Session getSession(){

//返回当前线程绑定的session,需要在hibernate.cfg.xml增加配置

//如果没有的新建一个,然后和线程绑定

//该Session对象在事务结束自动关闭

//该Session对象必须在一个事务中使用

return sf.getCurrentSession();

//创建一个新的Session对象,

//必须手动关闭

//return sf.openSession();

}

}

public class OpenSessionInViewInterceptor extends AbstractInterceptor {

@Override

public String intercept(ActionInvocation in) throws Exception {

// 开启事务

Session session = HibernateUtil1.getSession();

Transaction tx = session.beginTransaction();

try {

System.out.println("----开启事务---");

in.invoke();// 执行Action,DAO和Result-->JSP

// 提交事务

tx.commit();

System.out.println("----提交事务---");

return null;

} catch (Exception ex) {

// 回滚事务

tx.rollback();

System.out.println("----回滚事务---");

throw ex;

} finally {

// Hibernate会自动关闭session

System.out.println("----关闭事务---");

}

}

}

另外还需要在hibernate.cfg.xml中加入如下配置：

<!-- 支持sf.getCurrentSession()方法,将session与线程绑定 -->

<property name="current\_session\_context\_class">thread</property>

注：对于sf.openSession()和sf.getCurrentSession()，sf.getCurrentSession()

获得的session对象实现了和ThreaLocal的自动绑定以及在事务结束后的自

动关闭功能。而sf.openSession()则要显示地绑定和关闭，该

sf.getCurrentSession()对象必须在一个事务中使用。

=========案例练习==========

1.重构资费列表显示示例

为了支持延迟加载方法，需要在项目中采用OpenSessionInView思想。可以利用Filter和Interceptor技术实现。

\*.action-->拦截器前期（无）

-->Action-->DAO-->Result

-->JSP(通过标签和EL获取对象数据)

-->拦截器后期（关闭Session）

-->将HTML结果响应

**13.Hibernate关系映射的作用**

   利用关联映射，Hibernate可以帮助查询加载对象相关的数据，可以简化查询

   代码。

   此外还可以基于关系进行增删改操作。

   Hibernate关系映射分类： 一对一、一对多、多对一、多对多、继承

注：Hibernate关系映射区别于Hibernate数据映射。

**14.一对多关系映射**

   a.确定两个对象哪个是1方和n方

   b.需求：由1方对象查询n方记录，可以采用一对多关系映射

   c.首先在1方实体类中添加集合属性Set

   d.然后再1方的hbm.xml中定义一对多关系映射的描述

    <set name="关系属性">

        <key column="关联条件的外键字段"/>

        <one-to-many class="关联的另一方类型,即n方类型"/>

    </set>

   e.使用时，通过1方对象.关系属性获取n方数据

    eg:测试一对多关系：account--->service

public class TestOneToMany {

/\*\*

 \* 测试一对多

 \*/

public static void main(String[] args) {

test2();

test2();

}

/\*\*

 \* 显示帐务帐号信息(关联操作)

 \*/

public static void test2() {

int id = 1011;

Session session = HibernateUtil.getSession();

Account account = (Account) session.load(Account.class, id);

System.out.println("--------显示帐务帐号基本信息--------");

System.out.println("登录名:" + account.getLoginName());

System.out.println("身份证号:" + account.getIdcardNo());

System.out.println("状态:" + account.getStatus());

System.out.println("创建日期" + account.getCreateDate());

System.out.println("------包含的业务帐号信息-------");

Set<Service> list = account.getServices();

for (Service s : list) {

System.out.println(s.getId() + " " + s.getUnixHost() + " "

+ s.getOsUsername() + "" + s.getStatus());

}

HibernateUtil.closeSession();

}

/\*\*

 \* 显示帐务帐号信息(单表操作)

        如果用单表操作的话，要手动的编写两次查询语句，而关联操作只需一

         次

 \*/

public static void test1() {

int id = 1011;

Session session = HibernateUtil.getSession();

Account account = (Account) session.load(Account.class, id);

System.out.println("--------显示帐务帐号基本信息--------");

System.out.println("登录名:" + account.getLoginName());

System.out.println("身份证号:" + account.getIdcardNo());

System.out.println("状态:" + account.getStatus());

System.out.println("创建日期" + account.getCreateDate());

System.out.println("------包含的业务帐号信息-------");

String hql = "from Service where accountId=?";

Query query = session.createQuery(hql);

query.setInteger(0, id);

List<Service> list = query.list();

for (Service s : list) {

System.out.println(s.getId() + " " + s.getUnixHost() + " "

+ s.getOsUsername() + "" + s.getStatus());

}

HibernateUtil.closeSession();

}

}

<!-- 描述services属性,采用一对多关系加载Service记录 -->

       <set name="services">

        <!-- 指定关联条件,写外键字段 -->

        <key column="ACCOUNT\_ID"></key>

        <!-- 指定采用一对多关系, class指定关联的另一方-->

        <one-to-many class="org.tarena.netctoss.entity.Service"/>

 </set>

test2()输出：

--------显示帐务帐号基本信息--------

Hibernate:

    select

        account0\_.ID as ID0\_0\_,

        account0\_.RECOMMENDER\_ID as RECOMMEN2\_0\_0\_,

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

        account0\_.LAST\_LOGIN\_IP as LAST20\_0\_0\_

    from

        ACCOUNT account0\_

    where

        account0\_.ID=?

登录名:dgbf70

身份证号:330902197108270429

状态:1

创建日期2009-03-01

------包含的业务帐号信息-------

Hibernate:

    select

        services0\_.ACCOUNT\_ID as ACCOUNT4\_1\_,

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

        services0\_.CLOSE\_DATE as CLOSE10\_1\_0\_

    from

        SERVICE services0\_

    where

        services0\_.ACCOUNT\_ID=?

2002 192.168.0.26 huangr2

2004 192.168.0.23 huangr0

2003 192.168.0.20 huangr0

test1()输出：

--------显示帐务帐号基本信息--------

Hibernate:

    select

        account0\_.ID as ID0\_0\_,

        ...........................

        account0\_.LAST\_LOGIN\_IP as LAST20\_0\_0\_

    from

        ACCOUNT account0\_

    where

        account0\_.ID=?

登录名:dgbf70

身份证号:330902197108270429

状态:1

创建日期2009-03-01

------包含的业务帐号信息-------

Hibernate:

    select

        service0\_.ID as ID1\_,

        .....................

        service0\_.CLOSE\_DATE as CLOSE10\_1\_

    from

        SERVICE service0\_

    where

        accountId=?

2002 192.168.0.26 huangr2

2004 192.168.0.23 huangr0

2003 192.168.0.20 huangr0

**15.多对一关系映射**

   a.需要由n方对象查询1方对象信息

   b.在n方实体类中添加属性，属性类为1方类型

   c.在n方hbm.xml文件中，添加属性的映射描述信息

<many-to-one name="关系属性"

  column="关联条件的外键字段" class="关联的另一方类型,即1方类型"/>

   d.清除n方实体类中外键字段描述信息和属性

   e.使用时，通过n方对象.关联属性获取相关的1方记录信息

测试多对一关系：service-->account

public class TestManyToOne {

/\*\*

 \* 测试多对一：

 \*/

public static void main(String[] args) {

Session session = HibernateUtil.getSession();

Service service = (Service) session.load(Service.class, 2002);

System.out.println("业务帐号:" + service.getId());

System.out.println("业务服务器:" + service.getUnixHost());

System.out.println("业务用户名" + service.getOsUsername());

System.out.println("所属帐务帐号:" + service.getAccount().getId());

System.out.println("开通身份证:" + service.getAccount().getIdcardNo());

System.out.println("真实姓名:" + service.getAccount().getRealName());

HibernateUtil.closeSession();

}

     }

输出：

业务帐号:2002

Hibernate:

    select

        service0\_.ID as ID1\_0\_,

        service0\_.UNIX\_HOST as UNIX2\_1\_0\_,

        ........................

        service0\_.CLOSE\_DATE as CLOSE10\_1\_0\_

    from

        SERVICE service0\_

    where

        service0\_.ID=?

业务服务器:192.168.0.26

业务用户名huangr

所属帐务帐号:1011

Hibernate:

    select

        account0\_.ID as ID0\_0\_,

        account0\_.RECOMMENDER\_ID as RECOMMEN2\_0\_0\_,

        ........................

        account0\_.LAST\_LOGIN\_IP as LAST20\_0\_0\_

    from

        ACCOUNT account0\_

    where

        account0\_.ID=?

开通身份证:330902197108270429

  真实姓名:huangrong

**16.关联操作注意事项**

 级联操作也即关联操作。

 注意区分：关联属性和主对象的关系

 不要忘了在hbm.xml的关联属性定义中添加cascade属性，具体参见下面

        的”\*2)级联添加”

 \*1)关联(级联)查询

     默认情况下，关联属性数据的加载采用了延迟加载机制，当调用属性getter

    方法才出发查询。在使用时，如果需要将关联属性和主对象一起加载，可以

    通过以下方法改变加载机制

     a.(1)在hbm.xml中为该属性添加lazy="false" (关联属性数据在主对象

        加载时加载)

       为该属性追加fetch="join"

       (2).(指定关联属性加载方式，可以指定为select,join,subselect值)

<many-to-one name="account" column="ACCOUNT\_ID"

lazy="false" fetch="join"

class="org.tarena.netctoss.entity.Account">

  </many-to-one>

     b.写一个HQL，采用join fetch关键字加载关联属性。

    （注意：a方案会影响所有Service对象操作，不推荐，如果需要一起加载关

     联属性，建议采用b方案）

public class TestJoinFetch {

/\*\*

 \* 用join fetch关键字加载关联属性,取消延迟加载

 \*/

public static void main(String[] args) {

test2();

}

/\*\*

 \* 实例化Account对象时一起实例化关联属性services

 \*/

public static void test2(){

String hql = "from Account a " +

"join fetch a.services " +

"where a.id=?";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

query.setInteger(0, 1011);

Account account =

(Account)query.uniqueResult();

System.out.println("--------显示帐务帐号基本信息--------");

System.out.println("登录名:"+account.getLoginName());

System.out.println("身份证号:"+account.getIdcardNo());

System.out.println("状态:"+account.getStatus());

System.out.println("创建日期"+account.getCreateDate());

System.out.println("------包含的业务帐号信息-------");

Set<Service> list = account.getServices();

for(Service s : list){

System.out.println(s.getId()+" "

+s.getUnixHost() +" "

+s.getOsUsername() +""

+ s.getStatus());

}

HibernateUtil.closeSession();

}

/\*\*

 \* 实例化Service对象时一起实例化关联属性account

 \*/

public static void test1(){

String hql = "from Service s " +

"join fetch s.account " +

"where s.id=?";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

query.setInteger(0, 2002);

Service service  = (Service)query.uniqueResult();

System.out.println("业务帐号:"+service.getId());

System.out.println("业务服务器:"+service.getUnixHost());

System.out.println("业务用户名"+service.getOsUsername());

System.out.println("所属帐务帐号:"+service.getAccount().getId());

System.out.println("开通身份

          证:"+service.getAccount().getIdcardNo());

System.out.println("真实姓名:"+service.getAccount().getRealName());

HibernateUtil.closeSession();

}

}

   2)级联添加

     对主对象做添加操作时，关联属性的数据也做相应的添加操作。

     a.需要在hbm.xml的关联属性定义中添加cascade属性，属性值可以为

       none(默认),all,delete,save-update等。

       其中：none为默认，表示不进行级联操作。

            save-update:级联添加或者级联更新。

            delete：级联删除

               all：包括级联添加、删除、更新

     b.在执行session.save()之前，将关联数据对象添加到关联属性中。

<!-- 描述services属性,采用一对多关系加载Service记录 -->

       <set name="services" cascade="all" inverse="true">

/\*\*

 \* 级联添加操作 同时向数据库添加一个Account和两个Service记录

 \*/

public static void testAdd() {

// 创建一个Account

Account account = new Account();

account.setLoginName("tiggy41");

account.setLoginPasswd("123456");

account.setRealName("老虎");

account.setIdcardNo("33033234324393");

account.setTelephone("13234232");

// 创建两个Service

Service service1 = new Service();

service1.setUnixHost("192.168.0.23");

service1.setOsUsername("hooqt1");

service1.setLoginPasswd("111111");

service1.setCostId(1);

service1.setAccount(account);

// --

Service service2 = new Service();

service2.setUnixHost("192.168.0.20");

service2.setOsUsername("hilot");

service2.setLoginPasswd("222222");

service2.setCostId(3);

service2.setAccount(account);

// 将service1和service2给account对象的services赋值

account.getServices().add(service1);

account.getServices().add(service2);

// 添加

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

session.save(account);

tx.commit();

HibernateUtil.closeSession();

}

   3)级联删除

     对主对象做删除操作时，关联属性的数据也做相应的删除操作。

      a.需要在hbm.xml中的关联属性开启级联删除操作。

     在执行session.delete(obj)操作时，删除的obj对象时利用session查询出

        来的。注意不要使用new方式，因为不具有关联数据

      （级联删除采用n+1个delete清除数据，因此关联数据对象n过多时，不推

       荐使用，而是采用HQL语句进行删除）

Hibernate在做级联删除操作时，比如删除service表中相关5个

accountId=1010相关字段时，其在底层执行的sql语句并不是delete from

service where accountId=1010 而是5条delete from service where

id=?,所以。在做批量删除时效率较低

 //批量清除Service关系表记录

delete from Service where account.id=?

Query query = session.createQuery(hql);

     //...设置参数

query.executeUpdate();

/\*\*

 \* 级联删除操作 删除account记录的同时， 也删除services属性中关联

   \*的servcie记录，关联数据对象n过多时，不推荐使用

 \*/

public static void testDelete() {

Session session = HibernateUtil.getSession();

Transaction tx = session.beginTransaction();

Account account = (Account) session.load(Account.class, 2);

session.delete(account);

tx.commit();

HibernateUtil.closeSession();

}

   4)inverse属性

     a.inverse可以控制关系字段值维护的操作由哪一方负责。默认情况下由具

      有关系的对象双方负责。

     b.Hibernate在做增删改操作时默认分成对（非关联字段）的增删改和关联

      字段更新两个步骤

     c.Hibernate中对关系的维护默认为有关系的多表共同维护，在一对多关系

     中，通常在多方中设置inverse="true"，意思要多方放弃关系维护

      操作，当程序对1方对象做级联操作时，不会再出现update维护关系字段

      的语句。这样可以提高执行效率。

 inverse="true"在添加的时候可以省略Hibernate对关联字段的更新

（account\_Id）步骤，这样就可以解除一方解除关系维护操作

eg:

inverse=”true”时的添加操作：

Hibernate:

    insert into SERVICE

     (UNIX\_HOST, COST\_ID, ACCOUNT\_ID, OS\_USERNAME, LOGIN\_PASSWD, STATUS, CREATE\_DATE, PAUSE\_DATE, CLOSE\_DATE, ID)

    values

        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

默认情况下的添加操作（inverse=”false”）：

insert  into SERVICE

     (UNIX\_HOST, COST\_ID, ACCOUNT\_ID, OS\_USERNAME, LOGIN\_PASSWD,        STATUS, CREATE\_DATE, PAUSE\_DATE, CLOSE\_DATE, ID)

    values

        (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)

   \*\*\* update SERVICE  set ACCOUNT\_ID=? where ID=?

--------------------------案例-----------------------------------

ACCOUNT ： 账务账号表(1方)

SERVICE ：业务账号表(n方)

案例1：查看账务账号及其包含的业务账号信息

案例2：查看某一个业务账号及其账务账号信息

**17.如何利用MyEclipse根据数据库生成实体类和映射描述文件**

 1)在DB Browser中建立一个与数据库的连接

 2)新建一个Web Project工程

 3)为工程添加Hibernate开发框架

   （导入包，添加主配置及其参数设置）

     选中工程-->右键-->MyEclipse-->Add Hibernate Capabilities...

  4)按MyEclipse向导添加Hibernate框架开发包，添加主配置文件，设置连接参

    数，创建一个HibernateUtil工具类  .

    ----------生成实体类和hbm.xml-------------

  5)进入DB Browser，选中要操作的数据表，右键-->Hibernate Reverse

     Engineering.按向导生成实体类和hbm.xml.

  6)向导界面1：选择存放实体类和hbm文件的工程和package。

     选择要生成文件:hbm.xml，pojo，dao

  7)向导界面2: 将Type Mapping选中为Hibernate Types

  8)向导界面3:点击Finish完成

**18.多对多关系映射**

   多对多关系在数据库中需要3张表表示。

  例如AdminInfo-->Admin\_Role<--Role

  如果需要根据Admin查找Role，可以建立Admin到Role的多对多关系映射。具

  体过程如下：

   a.在Admin实体类中追加一个集合属性，用于存储相关联的Role对象信息

   b.在Admin的hbm.xml中描述集合属性映射

<set name="关系属性" table="关系表">

   <key column="与当前类型联的关系表字段">

   </key>

   <many-to-many  class="关联的另一方类型"

      column="与另一方类型关联的关系表字段">

   </many-to-many>

 </set>

eg:

    <!-- 采用多对多关系加载roles信息 -->

        <set name="roles" order-by="ROLE\_ID" table="ADMIN\_ROLE">

         <!-- 指定ADMIN\_ROLE表的ADMIN\_ID与当前AdminInfo主键关联 -->

         <key column="ADMIN\_ID"></key>

         <!-- 指定ADMIN\_ROLE表的ROLE\_ID与Role主键关联 -->

         <many-to-many class="org.tarena.entity.Role" column="ROLE\_ID">

         </many-to-many>

    </set>

public class TestManyToMany {

/\*\*

 \*基于多对多关系映射的查询、添加、删除

 \*/

public static void main(String[] args) {

 testFindAdmin();

// testAdminAddRole();

// testAdminRemoveRole();

//testFindRole();

 testAddAdmin();

}

/\*\*查询具有某个角色的用户 \*/

public static void testFindRole() {

Session session = HibernateSessionFactory.getSession();

Role role = (Role) session.get(Role.class, 1);

System.out.println(role.getName());

System.out.println("-----具有该角色的用户----");

for (AdminInfo admin : role.getAdmins()) {

System.out.println(admin.getId() + " " + admin.getAdminCode());

}

HibernateSessionFactory.closeSession();

}

/\*\*给管理员删除某个角色\*/

public static void testAdminRemoveRole() {

Session session = HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

// 分配角色

// 获取管理员

AdminInfo admin = (AdminInfo) session.get(AdminInfo.class, 1);

// 获取角色

Role role1 = (Role) session.get(Role.class, 10);

// 将角色对象在管理员对象的roles中移除

admin.getRoles().remove(role1);

// 更新admin

session.update(admin);

tx.commit();

HibernateSessionFactory.closeSession();

}

/\*\* 利用多对多查询管理员和角色信息\*/

public static void testFindAdmin() {

Session session = HibernateSessionFactory.getSession();

AdminInfo admin = (AdminInfo) session.load(AdminInfo.class, 1);

System.out.println(admin.getAdminCode() + " " + admin.getName());

System.out.println("---具有以下角色---");

for (Role role : admin.getRoles()) {

System.out.println(role.getId() + " " + role.getName());

}

HibernateSessionFactory.closeSession();

}

/\*\*给管理员追加角色\*/

public static void testAdminAddRole() {

Session session = HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

// 分配角色

// 获取管理员

AdminInfo admin = (AdminInfo) session.get(AdminInfo.class, 1);

// 获取角色

Role role = (Role) session.get(Role.class, 9);

Role role1 = (Role) session.get(Role.class, 10);

// 将角色对象给管理员对象的roles赋值

admin.getRoles().add(role);

admin.getRoles().add(role1);

// 更新admin对象

session.update(admin);

tx.commit();

HibernateSessionFactory.closeSession();

}

/\*\*给管理员添加角色\*/

public static void testAddAdmin(){

Session session = HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

//添加管理员

AdminInfo admin = new AdminInfo();

admin.setAdminCode("Scofield");

admin.setName("斯科菲尔德");

admin.setPassword("12364");

admin.setEnrolldate(new Date(System.currentTimeMillis()));

//获取角色

Role role1 = (Role) session.get(Role.class, 2);

Role role2 = (Role) session.get(Role.class, 7);

// 将角色对象给管理员对象的roles赋值

admin.getRoles().add(role1);

admin.getRoles().add(role2);

//给管理员添加角色

session.save(admin);

tx.commit();

HibernateSessionFactory.closeSession();

}

}

 Hibernate经典总结

**19.继承关系映射**

   a.父类一张表，每个子类一个表，主键对等

   b.可以采用<joined-subclass>进行继承关系映射，具体如下

    --在子类追加extends 父类

    --在子类hbm.xml中定义

 <joined-subclass name="子类类型" extends="父类类型" table="子类表">

   <key column="子类哪个字段与父类关联">

   </key>

   //子类中属性的property映射

</joined-subclass>

     eg:参见Hibernate05\_1

hbm.xml:

<hibernate-mapping>

    <joined-subclass name="org.tarena.entity.Book" table="BOOK"

        extends="org.tarena.entity.Product" schema="SYSTEM">

         <!-- 指定父子之间的关联条件，写子类的关联字段 -->

         <key column="ID"></key>

         <!-- 由于映射类型为继承类型，所以把子类的主键映射描述去掉 -->

        <!--

        <id name="id" type="integer">

            <column name="ID" precision="5" scale="0" />

            <generator class="assigned" />

        </id>

         -->

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

            <column name="AUTHOR" length="20" />

        </property>

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

            <column name="PUBLISHING" length="50" />

        </property>

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

            <column name="WORD\_NUMBER" length="20" />

        </property>

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

            <column name="TOTAL\_PAGE" length="20" />

        </property>

    </joined-subclass>

</hibernate-mapping>

test.java:

public class TestExtends {

/\*\*

 \* 多对多关系映射：

 \*/

public static void main(String[] args) {

addBook1();

//addBook2();

//findBook();

//deleteBook();

//addCar();

//findAllBook();

}

public static void findAllBook(){

String hql = "from Book";

Session session =

HibernateSessionFactory.getSession();

Query query = session.createQuery(hql);

List<Product> list = query.list();

for(Product p:list){

System.out.println(p.getId()+" "+p.getName()+" "+p.getPrice());

}

HibernateSessionFactory.closeSession();

}

public static void addCar(){

Session session =

HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

//添加汽车

Car car = new Car();

car.setName("Q5");

car.setPrice(150000);

car.setProductPic("4.jpg");

car.setType("J");

car.setBrand("宝马");

car.setColor("红色");

car.setDisplacement("2.0排量");

session.save(car);//保存

tx.commit();

HibernateSessionFactory.closeSession();

}

public static void deleteBook(){

Session session =

HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

Book book = (Book)session.load(Book.class, 3);

session.delete(book);

tx.commit();

HibernateSessionFactory.closeSession();

}

public static void findBook(){

Session session =

HibernateSessionFactory.getSession();

Book book = (Book)session.load(Book.class, 1);

System.out.println(book.getName()+" "+book.getPrice());

System.out.println(book.getAuthor()+" "+book.getPublishing());

HibernateSessionFactory.closeSession();

}

/\*\* 基于继承关系映射添加\*/

public static void addBook2(){

Session session =

HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

//添加图书

Book book = new Book();

book.setName("Struts2框架使用");

book.setPrice(90);

book.setProductPic("2.jpg");

book.setAuthor("张三的书");

book.setPublishing("菜鸟出版社");

book.setTotalPage("100");

book.setWordNumber("100个字");

session.save(book);//添加图书

tx.commit();

HibernateSessionFactory.closeSession();

}

/\*\* 没有关系映射添加图书\*/

public static void addBook1(){

Session session =

HibernateSessionFactory.getSession();

Transaction tx = session.beginTransaction();

//添加图书

//product

Product pro = new Product();

pro.setName("Java语言基础");

pro.setPrice(100);

pro.setProductPic("1.jpg");

session.save(pro);//保存product信息

//book

Book book = new Book();

book.setId(pro.getId());

book.setAuthor("我写的");

book.setPublishing("我家出版社");

book.setWordNumber("10个字");

book.setTotalPage("2");

session.save(book);//保存book信息

tx.commit();

HibernateSessionFactory.closeSession();

}

}

######示例表Oracle######

CREATE TABLE PRODUCT

(

ID  NUMBER(5) CONSTRAINT PRODUCT\_ID\_PK PRIMARY KEY,

NAME  VARCHAR2(20),

PRICE  NUMBER(15,2),

PRODUCT\_PIC  VARCHAR2(100)

);

CREATE SEQUENCE product\_seq;

CREATE TABLE BOOK

(

ID  NUMBER(5) CONSTRAINT BOOK\_ID\_PK PRIMARY KEY,

AUTHOR  VARCHAR2(20),

PUBLISHING  VARCHAR2(50),

WORD\_NUMBER VARCHAR2(20),

TOTAL\_PAGE VARCHAR2(20)

);

CREATE TABLE CAR

(

ID  NUMBER(5) CONSTRAINT CAR\_ID\_PK PRIMARY KEY,

BRAND VARCHAR2(20),

TYPE VARCHAR2(1),

COLOR  VARCHAR2(50),

DISPLACEMENT VARCHAR2(20)

);

##############################

**20、Hibernate查询**

  \*a.HQL查询

    Hibernate Query Language

    HQL与SQL语句结构相似，SQL语句是面向数据表和字段进行查询，而HQL

    语句是面向Hibernate映射过来的对象进行查询，因此HQL被称为是

    一种面向对象查询语言

    HQL和SQL共同点：

      --都支持select,from,where,order by,having,group by等子句。

      --都支持运算符表达式，例如+,-,\*,/,>,<等

      --都支持in,not in,between and,like等过滤条件关键字

      --都支持分组函max,min,sum,avg,count

    HQL和SQL不同点：

      --HQL是大小写敏感的，类名和属性名严格区分大小写

      --HQL不支持select \* 写法

      --HQL不支持join...on...中的on子句，因为join...on发生在多表操作，而

         Hibernate中对于有关系的多张表已将将关联映射写在了xxx.hbm.xml

         中，在查询时会自动加上join..on

      --HQL不支持表名和字段名

 HQL案例：

--查询所有:

from Account

/\*\*查询所有的Account select \* from account \*/

public static void test1() {

 String hql = "from Account"; //也可写成：String hql = "select a from

                                  Account a";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

List<Account> list = query.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName() + " "+

                            a.getIdcardNo());

}

HibernateUtil.closeSession();

     }

--参数查询:

from Account where status=?

    也可以写成：from Account where status=:s

   eg1:使用默认参数：?

   /\*\* 测试带参数的查询\*/

public static void test3() {

// 查询所有status=0的account记录

// String hql = "from Account where status=?";

// 查询status=1 并且 realname姓guo的

String hql = "from Account " + "where status=? and realName like ?";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

// 设置?参数

query.setString(0, "1");

query.setString(1, "guo%");

List<Account> list = query.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName() + " "

+ a.getIdcardNo());

 }

HibernateUtil.closeSession();

}

       eg2:使用Hibernate中的参数： :Xxx

/\*\*测试带参数的查询\*/

public static void test4() {

// 查询status=1 并且 realname姓guo的

String hql = "from Account " + "where status=:s and realName like :r";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

// 设置:s和:r参数

query.setString("r", "guo%");

query.setString("s", "1");

List<Account> list = query.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName() + " "

+ a.getIdcardNo());

}

HibernateUtil.closeSession();

}

--查询部分字段:

    select id,osUsername from Service

    返回List<Object[]>

    select new Service(id,osUsername) from Service

    返回List<Service>

注意：查询部分字段，也包括select id,osUsername,.......(Account中的

      所有字段)，情况；非部分字段查询只限于”from account ”。

    eg1:返回List<Object[]>

   /\*\* 测试查询部分字段值 查询部分字段，hibernate会利用Object[]封装一条记

       录信息\*/

public static void test5() {

String hql = "select id,osUsername,unixHost "

+ "from Service where account.id=?";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

query.setInteger(0, 1011);

// 部分字段用Object[]封装一行结果

List<Object[]> list = query.list();

for (Object[] objs : list) {

System.out.println(objs[0] + " " + objs[1] + " " + objs[2]);

}

HibernateUtil.closeSession();

}

  eg2:返回List<Service>

     要先在Service实体类中添加如下构造器：

public Service(){}

public Service(int id,String osUsername,String unixHost){

this.id = id;

this.osUsername = osUsername;

this.unixHost = unixHost;

}

   /\*\* 测试查询部分字段值查询部分字段，通过追加构造器使hibernate利用

 \* List<Service>封装一条记录信息\*/

public static void test6() {

// 需要在Service类中添加相应的构造方法

String hql = "select " + "new Service(id,osUsername,unixHost) "

+ "from Service where account.id=?";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

query.setInteger(0, 1011);

// 部分字段用Service封装一行结果

List<Service> list = query.list();

for (Service s : list) {

System.out.println(s.getId() + " " + s.getOsUsername() + " "

+ s.getUnixHost());

}

HibernateUtil.closeSession();

}

--在hbm.xml中定义hql：

   Query query =  session.getNamedQuery("标识符")

eg:

  先在Service.hbm.xml中加入如下配置：（与class平级）

  <!-- 在TestHQL.java中被读取用于做查询语句 -->

          <query name="findAll"><![CDATA[ from Account  ]]></query>

   /\*\*在hbm.xml中定义hql\*/

public static void test2() {

// String hql = "from Account";

Session session = HibernateUtil.getSession();

// Query query = session.createQuery(hql);

// 获取hbm.xml定义的hql

Query query = session.getNamedQuery("findAll");

List<Account> list = query.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName() + " "

+ a.getIdcardNo());

 }

HibernateUtil.closeSession();

}

--分页查询用法：

    query.setFirstResult(抓取起点从0开始计算)

    query.setMaxResult(抓取最大数量);

    List list = query.list();

/\*\* 分页查询 \*/

public static void test7() {

Session session = HibernateUtil.getSession();

Query query = session.getNamedQuery("findAll");

// 设置分页抓取参数,从第1条抓，最多抓取3个

query.setFirstResult(0);// 设置抓取起点(从0开始)

query.setMaxResults(3);// 设置最大抓取数量

List<Account> list = query.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName());

 }

HibernateUtil.closeSession();

}

  b.Criteria查询（QBC）

只需了解，具体参考示例和Hibernate帮助文档

public static void test2() {

// 按status=0并且accountId=1011查询Service

Session session = HibernateUtil.getSession();

Criteria c = // from Service

session.createCriteria(Service.class);

// 添加查询条件

c.add(Restrictions.and(Restrictions.eq("status", "0"), Restrictions.eq(

"account.id", 1011)));

// 执行查询

List<Service> list = c.list();

for (Service s : list) {

System.out.println(s.getId() + " " + s.getOsUsername());

}

HibernateUtil.closeSession();

}

public static void test1() {

Session session = HibernateUtil.getSession();

// 设置查询的数据源

Criteria c = session.createCriteria(Account.class);

// 执行查询

c.setFirstResult(0);

c.setMaxResults(5);

List<Account> list = c.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName());

}

HibernateUtil.closeSession();

}

  c.Native SQL查询

只需了解，具体参考示例和Hibernate帮助文档

/\*\*

 \* 默认采用Object[]封装一行结果

 \*/

public static void test1() {

String sql = "select id,unix\_host,os\_username from SERVICE";

Session session = HibernateUtil.getSession();

SQLQuery query = session.createSQLQuery(sql);

// 默认采用Object[]封装一行记录

List<Object[]> list = query.list();

for (Object[] objs : list) {

System.out.println(objs[0] + " " + objs[1] + " " + objs[2]);

}

HibernateUtil.closeSession();

}

/\*\*

 \* 利用指定的类型封装记录结果

 \*/

public static void test2() {

String sql = "select \* from SERVICE";

Session session = HibernateUtil.getSession();

SQLQuery query = session.createSQLQuery(sql);

// 指定采用哪种类型将一行记录封装,仅限于单表使用

query.addEntity(Service.class);

List<Service> list = query.list();

for (Service s : list) {

System.out.println(s.getId() + " " + s.getOsUsername());

}

HibernateUtil.closeSession();

}

**21.Hibernate高级特性**

 1)二级缓存技术

    SessionFactory级别的缓存，受SessionFactory管理，可以被不同

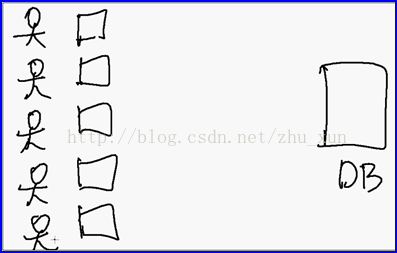
    Session访问和操作。默认是关闭。一般在使用时需要利用

    SessionFactory.evict()等方法显式的管理该缓存。

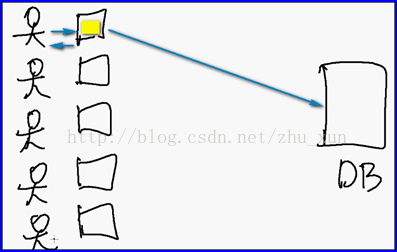
二级缓存和一级缓存的区别：

Hibernate提供了二级缓存机制。

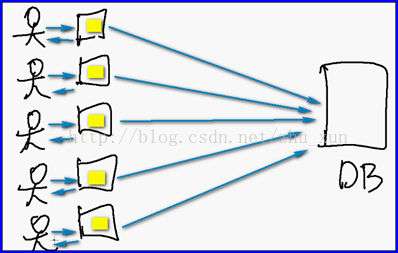
首先，Hibernate中的一级缓存机制（也叫做事务内的缓存）是与Session绑定在一起的。当一个Session开启，一级缓存创 建；当一个Session关闭，一级缓存销毁。若使用一级缓存机制（Session的缓存，每个用户线程对应一块Session缓存）现在有5个用户（5 个线程）访问Hibernate，那么Hibernate会为5个用户创建5个不同的Session（一个线程分配一个Session）。



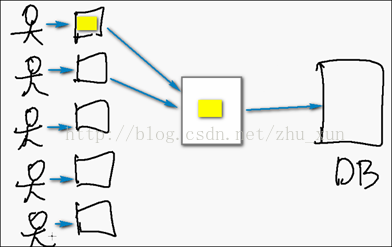
假设用户1调用getId("1")方法查找id=1的Emp对象，Session会首先查找内部有没有id=1的Emp对象，如果有，则返回给用户；没有则去数据库中查找，并保存到该Session中，当用户第二次访问时，就不用去数据库中取数据了。



一级缓存提高了效率，减少了访问数据库的压力。如果5个用户都调用getId("1")方法查找id=1的Emp对象，那么在这5个session中就分别保存着5个id=1的Emp对象，这样显然重复。



由此，我们引入了二级缓存机制（SessionFactory中的缓存，同一个项目中只有一份，所有用户共用）当用户1第一次调用getId("1")方法时，会到数据库中查找出Emp对象，保存到一级缓存中的同时，也在二级缓存中保存一份。这样，当其他用户也需要id=1的Emp对象时，只需要到二级缓存中查找即可，就不用连接到数据库了。



一级缓存是用户线程专用的，二级缓存是大家共用的。

我们通过配置一些现成的缓存组件（如ehcache）来实现，同时我们还可以控制哪些对象需要放入二级缓存，哪些对象不需要做二级缓存。

   a.什么情况可以考虑使用二级缓存

     --该对象被多个不同用户频繁使用

     --该对象更新操作不频繁

   b.如何使用二级缓存

     --添加ehcache.jar开发包和src/ehcache.xml配置

     --在hibernate.cfg.xml中开启二级缓存，指定采用哪种二级

       缓存组件

    hibernate.cfg.xml：

<!-- 指定二级缓存组件 -->

  <property name="hibernate.cache.provider\_class">

net.sf.ehcache.hibernate.EhCacheProvider

  </property>

       ehcache.xml:

     <!--

b.eternal表示是否设置这些放入二级缓存的数据对象为永久的

         （即放入即保存，不再清除）一般都为false

c.timeToIdleSeconds=120表示如果120秒内，放入的对象没有被

          再次访问到，就清除出去

d.timeToLiveSeconds=120表示对象在缓存中存活的时间，一个

           对象进入到本缓存中120秒后，就会自动被清除（一般 设置

           的时间会比timeToIdleSeconds时间长），设置此属性是为了

           让更多活跃的对象进入到缓存中来。

e.overflowToDisk="true"表示如果活跃对象已经超出

          maxElementInMemory设置的最大值时，

  超出的对象要被写入到硬盘上保存下来，用于缓解活跃用户较

          多的情况。

        -->

    <defaultCache

        maxElementsInMemory="10000"

        eternal="false"

        timeToIdleSeconds="30"

        timeToLiveSeconds="240"

        overflowToDisk="true"

                      />Hibernate经典总结

<cache name="sampleCache1"

        maxElementsInMemory="10000"

        eternal="false"

        timeToIdleSeconds="300"

        timeToLiveSeconds="600"

        overflowToDisk="true"

  />

     --需要缓存哪个对象，就在hbm.xml中添加<cache>元素配

       置。

      <cache usage="read-only或read-write"

        region="采用ehcache.xml哪组参数缓存该对象"/>

Account.hbm.xml：

<!-- 指定采用二级缓存缓存Account对象 -->

    <cache usage="read-only" region="sampleCache1"/>

    注：上面的配置要写在配置文件的顶端

其中：

region属性表示指定使用哪个二级缓存。

usage属性表示二级缓存的使用方式。有两种：read-only

和read-write、read-only 表示如果值为read-only，那么就

不能修改，这样ehcache就不用考虑修改和更新的操作。

read-write 设置为read-write，ehcache还需要考虑更新和

修改，这样会降低效率。

所以，设置usage属性是很重要的，需要根据实际情况判

断存入的对象使用二级缓存的方式。

     c.二级缓存管理

         sessionFactory.evict方法

eg:

public class TestSecondCache {

/\*\*

 \* 二级缓存

 \*/

public static void main(String[] args) {

findAccount();

findAccount();

}

/\*\*

 \* 为使用二级缓存时，两次调用findAccount()方法，会执行两次查询

 \* 使用二级缓存后，两次调用findAccount()方法，只执行一次查询

 \* \*/

public static void findAccount() {

Session session = HibernateUtil.getSession();

Account account = (Account) session.get(Account.class, 1011);

System.out.println(account.getRealName() + " " +

          account.getIdcardNo());

HibernateUtil.closeSession();

}

}

 2)查询缓存技术

查询缓存基于二级缓存

二级缓存只保存单个对象，而不会管诸如取字符串、取数组、取集合等的

操作，但是查询缓存支持这些。

    a.查询缓存的使用

       --要对查询的目标对象开启二级缓存

  <!-- 设置开启二级缓存 -->

<property name="hibernate.cache.use\_second\_level\_cache">true

      </property>

<!-- 指定二级缓存组件 -->

<property name="hibernate.cache.provider\_class">

net.sf.ehcache.hibernate.EhCacheProvider

 </property>

       --在hibernate.cfg.xml中开启查询缓存设置

<!-- 开启查询缓存 -->

<property name="hibernate.cache.use\_query\_cache">

true

</property>

       --在执行query.list()查询之前，

         调用query.setCacheable(true);

    b.适合使用查询缓存的情况

       --不同用户都执行相同的SQL查询和相同结果

       --查询结果集不发生改变

   注意：在二级缓存和查询缓存使用关联映射时，关系属性数据

         默认不参与缓存，如果需要缓存关联属性数据，需要

         对关联属性和hbm.xml（Account.hbm.xml和

         Service.hbm.xml）都设置<cache>元素

    查询缓存是在二级缓存的基础之上的，所以关于二级缓

    存的在hibernate.cfg.xml中的配置要保留。

   在Account.hbm.xml和Service.hbm.xml中均添加：

 <!--指定该关联属性也参与缓存操作，

  在需要查询缓存时才需要添加-->

 <cache usage="read-only" region="sampleCache1"/>

   eg:

public class TestQueryCache {

/\*\*

 \* 查询缓存

 \*/

public static void main(String[] args) {

findAll();

System.out.println("-----------");

findAll();

}

public static void findAll() {

String hql = "select distinct a " + "from Account a "

+ "left outer join fetch a.services where a.realName

                     like ?";

Session session = HibernateUtil.getSession();

Query query = session.createQuery(hql);

// 设置采用查询缓存机制

query.setCacheable(true);

query.setString(0, "guo%");

List<Account> list = query.list();

for (Account a : list) {

System.out.println(a.getId() + " " + a.getRealName());

// 业务账号信息

for (Service s : a.getServices()) {

System.out.println("---" + s.getId() + " " +

                s.getOsUsername());

}

}

HibernateUtil.closeSession();

}

}

             hibernate缓存机制总结：

Hibernate中的一级缓存机制（也叫做事务内的缓存）是与Session绑定在一起的。

当一个Session开启，一级缓存创建；当一个Session关闭，一级缓存销毁。

Hibernate中的二级缓存特点如下:

a.二级缓存被称为SessionFactory级别缓存. 生命周期与SessionFactory对象相关

b. 二级缓存空间可以被不同的Session对象访问 共享.

c. 二级缓存默认是关闭状态.如果遇到某个数据对象被多个不同的Session频繁访 问,可以开启.

Hibernate中的查询缓存：

前面的一级和二级缓存,缓存的时load,get出来的数据对象.不能缓存一个结果集.查询缓存可以缓存查询语句和结果集， 当重复执， 同一个查询语句时,只取数据库查询一次,后续都是将缓存中的结果集取出。适用于频繁的执行同一个查询语句,而且查询结果集很少发生变化的情况。

3)悲观锁和乐观锁

 当出现多个用户同时执行更新等操作时，会出现事务交叉更新操作的冲突，会破

 坏业务和数据的完整性。可以使用悲观锁和乐观锁解决这类问题。

 悲观锁：

 a.悲观锁机制：在进行数据查询时追加一个锁机制，进行业务操作，此时其他用

               户不能进行增删改操作，在事务结束时会自动将锁释放，其他用

               户可以继续执行此类操作。

悲观锁的处理方式是当前线程的事务没有结束前，其它事务都要

等着（事务串行化）。

  悲观锁特点：将用户操作一个一个处理，可以解决更新并发问题，缺点是处理效

              率比较低。

   Hibernate悲观锁机制一般是借助于数据库锁机制。

   eg:

 Train train =

(Train)session.load(Train.class, 1,LockMode.UPGRADE);

  乐观锁：

     b.乐观锁机制：多个不同用户都可以同时对数据库记录进行查看和更新操作，

       但是最先commit提交的用户会执行成功，后续用户会以异常形  式提示失败。

         乐观锁是借助于一个版本字段进行控制，当并发操作中一个用  户成功提交了，版本字段值会自动加1，后续提交的对象版本信

                 息小于数据库版本字段值会被hibernate阻止掉。

     乐观锁实现原理：

使用乐观锁，我们需要做数据库更改，为数据库增加一个字段version（版本号），当用户读取数据时，会将版本号version 一同读出，如果该用户修改了数据，会先将取出的版本号与数据库中的版本号做对比，如果相同，才能修改；修改完成后，会将版本号version+1如果不相 同，则不能修改，会抛出异常。

如果不使用框架技术，那么我们需要手工做对比，使用Hibernate框架后，Hibernate可以帮助我们做version对比的操作。

       乐观锁特点：允许多个用户同时操作，处理效率相对较高。

      乐观所使用步骤：

        --将原有数据表追加一列版本字段，初始值0

        --在实体类中添加版本属性

        --在映射描述文件中采用<version>元素定义版本属性和版本字段的映射

<!--optimistic-lock 指定采用版本字段形式实现乐观锁控制-->

<hibernate-mapping>

<class name="org.tarena.netctoss.entity.Train"

      optimistic-lock="version" table="TRAIN">

<id name="id" type="integer">

<column name="T\_ID" precision="22" scale="0" />

<generator class="assigned" />

</id>

<!-- 描述版本字段的映射 -->

<version name="version" type="integer">

<column name="T\_VERSION"></column>

</version>

<property name="value" type="integer">

<column name="T\_VALUE" precision="22" scale="0" />

</property>

</class>

</hibernate-mapping>

        --当发生多个事务并行交叉执行时，第一个提交的成功，后续提交的会抛

          出异常（org.hibernate.StaleObjectStateException）。可以异常捕

         获给用户一个友善的提示。

==============示例表=================

create table train(

t\_id number primary key,

t\_value number,

t\_version number);

insert into train values (1,100,0);