[Hibernate核心接口 2](#_Toc285005591)

[配置文件 2](#_Toc285005592)

[hibernate.cfg.xml 2](#_Toc285005593)

[映射文件 3](#_Toc285005594)

[ExportDB.java 4](#_Toc285005595)

[HibernateUtils.java 4](#_Toc285005596)

[log4j.properties 6](#_Toc285005597)

[Hibernate3 中 MySQL驱动导出 6](#_Toc285005598)

[java程序模板 7](#_Toc285005599)

[test程序 7](#_Toc285005600)

[lifecycle(生命周期) 12](#_Toc285005601)

[get和load的区别: 13](#_Toc285005602)

[session flush 13](#_Toc285005603)

[session的几个方法: 23](#_Toc285005604)

[锁 24](#_Toc285005605)

[悲观锁(并发性不好) 24](#_Toc285005606)

[乐观锁(并发性好, 用得最多) 26](#_Toc285005607)

[其它锁 29](#_Toc285005608)

[树 29](#_Toc285005609)

[HQL 30](#_Toc285005610)

[1、简单属性查询【重要】 32](#_Toc285005611)

[2、实体对象查询【重要】 38](#_Toc285005612)

[3、条件查询【重要】 49](#_Toc285005613)

[4、hibernate也支持直接使用sql进行查询, 适合批量更新 58](#_Toc285005614)

[5、外置命名查询, 适合HQL经常变的, 把HQL写在配置文件里. 60](#_Toc285005615)

[6、查询过滤器, 在这个session中每个sql语句上 加上过滤条件, 类似于Spring的AOP 62](#_Toc285005616)

[7、分页查询【重要】 64](#_Toc285005617)

[8、对象导航查询，在hql中采用 . 进行导航【重要】 66](#_Toc285005618)

[9、连接查询【重要】 67](#_Toc285005619)

[10、统计查询【重要】 71](#_Toc285005620)

[11、DML风格的操作（批量操作, 尽量少用，因为和缓存不同步） 74](#_Toc285005621)

[HQL对实体的更新和删除： 75](#_Toc285005622)

[缓存 76](#_Toc285005623)

[hibernate 一级缓存(session级的缓存, 缓存实体对象) 76](#_Toc285005624)

[hibernate 二级缓存(SessionFactory级的缓存, 只缓存实体对象) 85](#_Toc285005625)

[hibernate 查询缓存(缓存普通属性结果集, 对实体对象的结果集只缓存id) 96](#_Toc285005626)

[Hibernate 批量查询 或 批量更新 110](#_Toc285005627)

[lazy策略, 常用的三个地方 110](#_Toc285005628)

[\_\_\_\_\_\_\_\_\_\_\_\_\_\_未分类\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 117](#_Toc285005629)

[数据库隔离级别 117](#_Toc285005630)

[分组与排序 118](#_Toc285005631)

[hibernater基础操作 HQL常用语句示例 120](#_Toc285005632)

[HQL经典语句 122](#_Toc285005633)

# Hibernate核心接口

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | session | 负责执行被持久化对象的CRUD操作  非线程安全的. | | SessionFactory | 负责初始化Hibernate.  它充当数据存储源的代理, 并负责创建session对象  重量级的, 一般情况下一个项目只需要一个SessionFactory, 当需要操作多个数据库时, 可以为每个数据库指定一个SessionFactory | | Configuration | 负责配置并启动Hibernate, 创建SessionFactory对象,  在Hibernate启动过程中, Configuration类的实例首先定位映射文档位置, 读取配置, 然后创建SessionFactory对象. | | Transaction | 负责事务相关操作 | | query 和 Criteria | 负责执行各种数据库查询, 可以使用HQL和SQL两种方式 | | Lifecycle | 已不建议使用 | | Validatable | 对pojo验证, 已不建议使用 | | Interceptor(拦截器) | 没有侵入性 | | UserType | 用户自定义类型转换器, 例如用于一个人有3个邮箱地址, 用逗号表达式间隔, 自定义类型转换. | |

# 配置文件

## hibernate.cfg.xml

|  |
| --- |
| <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name="hibernate.connection.url">jdbc:mysql://localhost/bjsxt</property>  <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>  <property name="hibernate.connection.username">root</property>  <property name="hibernate.connection.password">root</property>  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  <property name="hibernate.show\_sql">true</property>  <property name="hibernate.hbm2ddl.auto">update</property>  <mapping resource="com/bjsxt/hibernate/User.hbm.xml" />  </session-factory>  </hibernate-configuration> |

## 映射文件

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| (映射文件,和实体类放在一个包内.)  <?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.bjsxt.hibernate”>  <class name="User" table=”t\_user” lazy=”true”>  <id name="id" column=”重命名” type=”手动指定类型” length=”20”>  <generator class="uuid" />  </id>  <property name="name" unique=” true” not-null=”yes” />  <property name="password"/>  <property name="createTime"/>  <property name="expireTime"/>  </class>  </hibernate-mapping>  映射文件中属性的配置:   |  |  | | --- | --- | | column | 自定义字段名 | | cascade=all/none/save-update/delete | 级联(操作上的连锁反应, 对[增], [删], [改]起作用) | | constrained=true/false | 外键约束 | | generator=uuid/native/assigned | 主键生成策略 | | inverse=true/false | 控制反转(关联关系的控制方向), 可以配置在一对多双向关联, 和 多对多双向关联. | | order-by=userid/… | 排序 | | property-ref="指定主键名称" | 指定主键 | | unique=true/false | 唯一性 | | <many-to-one name="classes" column="classesid" fetch="select"/> | hibernate会另外发送一条select语句抓取当前对象关联实体或集合 | | <many-to-one name="classes" column="classesid" fetch="join"/> | hibernate会通过select语句使用外连接来加载其关联实体或集合 | |

## ExportDB.java(导出表)

|  |
| --- |
| import org.hibernate.cfg.Configuration;  import org.hibernate.tool.hbm2ddl.SchemaExport;  public class ExportDB {  public static void main(String[] args) {  Configuration cfg = new Configuration().configure();  SchemaExport export = new SchemaExport(cfg);  export.create(true, true);  }  } |

## HibernateUtils.java

|  |
| --- |
| import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.cfg.Configuration;  public class HibernateUtils {  private static SessionFactory SessionFactory;  //静态带模块: 类加载的时候执行  static {  try {  Configuration cfg = new Configuration().configure();//读取Hibernate配置文件  SessionFactory = cfg.buildSessionFactory();//由配置文件创建SessionFactory  }catch(Exception e) {  e.printStackTrace();  }  }  public static SessionFactory getSessionFactory() { // SessionFactory是重量级的, 而且是线程安全的, 所以只需创建一次.  return SessionFactory;  }  public static Session getSession() { // Session是非线程安全的, 一个请求对应一个session. 用完必须关闭  return SessionFactory.openSession();  }  public static void closeSession(Session session) {  if (session != null) {//session是null的吗?  if (session.isOpen()) {//session是打开的吗?  session.close();  }  }  }  } |

## log4j.properties

|  |
| --- |
| log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.Target=System.out  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n  log4j.rootLogger=warn, stdout |

## Hibernate3 中 MySQL驱动导出

|  |
| --- |
| <!--  MyEclipse Database Explorer Driver Repository  -->  <Beans version="1.0">  <Bean Class="net.sourceforge.squirrel\_sql.fw.sql.SQLDriver">  <driverClassName>com.mysql.jdbc.Driver</driverClassName>  <identifier Class="net.sourceforge.squirrel\_sql.fw.id.UidIdentifier">  <string>-52fec7ee:1215ccacf46:-7f6d</string>  </identifier>  <jarFileName/>  <jarFileNames Indexed="true">  <Bean Class="net.sourceforge.squirrel\_sql.fw.util.beanwrapper.StringWrapper">  <string>D:\Tool\&#x7f16;&#x7a0b; &#x5de5;&#x5177;\JAR&#x5305;\mysql-connector-java-5.1.7-bin.jar</string>  </Bean>  </jarFileNames>  <name>MySQL Connector/J0</name>  <url>jdbc:mysql://&lt;hostname&gt;[&lt;:3306&gt;]/&lt;dbname&gt;</url>  </Bean>  <Bean Class="com.genuitec.eclipse.sqlexplorer.utils.ConnectionProfile">  <activate>false</activate>  <driverIdentifier Class="net.sourceforge.squirrel\_sql.fw.id.UidIdentifier">  <string>-52fec7ee:1215ccacf46:-7f6d</string>  </driverIdentifier>  <identifier Class="net.sourceforge.squirrel\_sql.fw.id.UidIdentifier">  <string>-52fec7ee:1215ccacf46:-7f90</string>  </identifier>  <name>com.mysql.jdbc.Driver</name>  <password encryption="true">SLHTjDlUMWTd0gpZj5ANNg==</password>  <promptForPassword>false</promptForPassword>  <schema Indexed="true"/>  <schemaDisplayPolicy>0</schemaDisplayPolicy>  <url>jdbc:mysql://localhost:3306/MySQL</url>  <userName>root</userName>  </Bean>  </Beans> |

# java程序模板

## test程序

|  |
| --- |
| import java.util.Date;  import org.hibernate.Session;  import org.hibernate.Transaction;  import junit.framework.TestCase;  public class SessionTest extends TestCase {  public void testHello1() {  System.out.println("-----------向上抛出异常------------");  //throw new java.lang.RuntimeException();//向上抛出异常  }  public void testHello2() {  System.out.println("-----------使用断言------------");  //this.assertEquals("hello", "hello111");//使用断言  }  public void save(){ **//添加数据**  Session session = null;  try{  session = HibernateUtils.getSession();  session.beginTransaction();    User user = new User();  user.setName("金万林");  user.setPassword("jinwanlin");  user.setCreateTime(new Date());  user.setExpireTime(new Date());    session.save(user);  session.beginTransaction().commit();  }catch(Exception e){  session.beginTransaction().rollback();  e.printStackTrace();  }finally{  HibernateUtils.closeSession(session);  }  }  public void get() { **//get查询**  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    //马上发出查询sql，加载User对象  //采用get加载数据，如果数据库中不存在相应的数据，返回null  User user = (User)session.get(User.class, "2c908e912264ad83012264ad86d70001");  System.out.println("user.name=" + user.getName());    //persistent状态,当属性发生改变的时候，hibernate会自动和数据库同步  user.setName("龙哥");  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  public void load() { **//load查询**  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    //不会立即发出查询sql，因为load方法实现了lazy（懒加载或延迟加载）  //延迟加载：只有真正使用这个对象的时候，才加载（发出sql语句）  //hibernate延迟加载实现原理是代理方式  //采用load加载数据，如果数据库中没有相应的数据  //那么抛出ObjectNotFoundException    User user = (User)session.load(User.class, "2c908e912264ad83012264ad86d70001");  System.out.println("user.name=" + user.getName());    //persistent状态,当属性发生改变的时候，hibernate会自动和数据库同步  user.setName("发哥");  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void update() { **//更新**  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    User user = (User)session.load(User.class, "2c908e9123b77f980123b77f9da60001");  user.setName("德华");    session.update(user);  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  public void delete() { **//删除**  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    User user = (User)session.load(User.class, "2c908e9123b77f980123b77f9da60001");  session.delete(user);  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

## lifecycle(生命周期)

|  |
| --- |
| transient  ['tran·sient ]状态的特征？  \* 在数据库中没有与之匹配的数据  \* 没有纳入session的管理    persistent [per'sist·ent]状态的特征？  \* persistent状态的对象在数据库中有与之匹配的数据  \* 纳入了session的管理  \* 在清理缓存（脏数据检查）的时候,会和数据库同步    detached  [de'tached]状态的特征？  \* 在数据库中有与之匹配的数据  \* 没有纳入session的管理 |

## get和load的区别:

|  |  |  |
| --- | --- | --- |
|  | get | load |
| 支持lazy吗? | 不支持 | 支持 |
| 如果没有匹配的数据 | 返回null | 抛出异常 |
| 备注:  //延迟加载：只有真正使用这个对象的时候，才加载（发出sql语句）  //hibernate延迟加载实现原理是代理方式  代理:系统将继承user类的CGLIB…,返回给你. | | |

## session flush

|  |
| --- |
| session flush主要做了两件事：   1. 清理缓存 2. 执行sql |
| session在什么情况下执行flush   1. 默认在事务提交时 2. 显示调用flush 3. 使用数据前，如iterate   hibernate按照下面的顺序发出sql语句  1. 所有对实体进行插入的语句, 其顺序按照对象执行session.save()的时间顺序.  2. 所有对实体进行更新的语句.  3. 所有进行集合删除的语句.  4. 所有对集合元素进行删除, 更新或者插入的语句.  5. 所有进行集合插入的语句.  6. 所有对实体进行删除的语句, 其顺序按照对象执行session.delete9)的时间顺序.  7. 有一个例外的是, 如果对象使用native方式来生成ID(持久化标识), 则他们一执行save就会被插入.  通过调用session.setFlashMode(FlushMode.COMMIT) 可以设置flash模式: 一般都不修改flash模式, 只有当使用触发器, 或把Hibernate和JDBC语句混合使用, 直接调用flush()才是很有用的.   |  |  | | --- | --- | | FlushMode.AUTO | Hibernate判断对象属性有没有更改, 如果被更改过变成了脏数据, 则在一个查询语句前将更新此改动一保证同步数据库;这是Hibernate的默认模式. | | FlushMode.COMMIT | 在事务结束之前清理Session的缓存, 其它任何时候都不清理缓存. 这样的设置将有可能使查询出来的数据是脏数据. | | FlushMode.NEVER | 除非强制执行session.flush(), 否则永不清理缓存. 这时对数据所做的修改只限于内存, 不会同步到数据库, 数据库的数据相当于只读数据. | | FlushMode.ALWAYS | 在每一个查询语句前都调用flush()进行缓存清理. 这种模式效率很低,不必要使用. | |
| import java.util.Date;  import org.hibernate.Session;  import org.hibernate.Transaction;  import junit.framework.TestCase;  public class SessionFlushTest extends TestCase {    /\*\*  \* 测试**uuid**主键生成策略  \*/  public void testSave1() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User1 user = new User1();  user.setName("李四");  user.setPassword("123");  user.setCreateTime(new Date());  user.setExpireTime(new Date());    //因为user的主键生成侧路采用的是uuid，所以调用完成save后，只是将user纳入到了session的管理  //不会发出insert语句，但是id已经生成，session中existsInDatebase状态为false  session.save(user);    //调用flush，hibernate会清理缓存，执行sql  //如果数据库的隔离级别设置为 未提交读，那么我们可以看到flush过的数据  //并且session中existsInDatebase状态为true  session.flush();    //提交事务  //默认情况下commit操作会先执行flush清理缓存，所以不用显示的调用flush  //commit后数据是无法回滚的  tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 测试**native**主键生成策略  \*/  public void testSave2() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User2 user = new User2();  user.setName("张三1");  user.setPassword("123");  user.setCreateTime(new Date());  user.setExpireTime(new Date());    //因为user的主键生成策略为native,所以调用session.save后，将发SQL,执行insert语句，返回由数据库生成的id  //纳入了session的管理，修改了session中existsInDatebase状态为true  //如果数据库的隔离级别设置为 未提交读，那么我们可以看到save过的数据  session.save(user);    tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }      /\*\*  \* 测试**uuid**主键生成策略  \*/  public void testSave3() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User1 user = new User1();  user.setName("王五");  user.setPassword("123");  user.setCreateTime(new Date());  user.setExpireTime(new Date());    //因为user的主键生成策略采用的是uuid，所以调用完成save后，只是将user纳入到了session的管理  //不会发出insert语句，但是id已经生成，session中existsInDatebase状态为false  session.save(user);    //将user对象从session中逐出，即session的EntityEntries属性中逐出  session.evict(user);  //无法成功提交，因为hibernate在清理缓存时，在session的insertions集合中取出user对象进行insert操作后  //需要更新entityEntries属性中的existsInDatabase为true，而我们采用evict已经将user从session的entityEntries  //中逐出了，所以找不到相关数据，无法更新，抛出异常  tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 测试**uuid**主键生成策略  \*/  public void testSave4() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User1 user = new User1();  user.setName("王五");  user.setPassword("123");  user.setCreateTime(new Date());  user.setExpireTime(new Date());    //因为user的主键生成策略采用的是uuid，所以调用完成save后，只是将user纳入到了session的管理  //不会发出insert语句，但是id已经生成，session中existsInDatebase状态为false  session.save(user);    //flush后hibernate会清理缓存，会将user对象保存到数据库中，将session中的insertions中的user对象  //清除，并且设置session中existsInDatebase的状态为true  session.flush();    //将user对象从session中逐出，即session的EntityEntries属性中逐出  session.evict(user);    //可以成功提交，因为hibernate在清理缓存时，在session的insertions集合中无法找到user对象  //所以就不会发出insert语句，也不会更新session中的existsInDatabase的状态  tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 测试native主键生成策略  \*/  public void testSave5() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User2 user = new User2();  user.setName("张三11");  user.setPassword("123");  user.setCreateTime(new Date());  user.setExpireTime(new Date());    //因为user的主键生成策略为native,所以调用session.save后，将执行insert语句，返回由数据库生成的id  //纳入了session的管理，修改了session中existsInDatebase状态为true  //如果数据库的隔离级别设置为为提交读，那么我们可以看到save过的数据  session.save(user);    //将user对象从session中逐出，即session的EntityEntries属性中逐出  session.evict(user);    //可以成功提交，因为hibernate在清理缓存时，在session的insertions集合中无法找到user对象  //所以就不会发出insert语句，也不会更新session中的existsInDatabase的状态  tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 测试assigned主键生成策略  \*  \*/  public void testSave6() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User3 user = new User3();  user.setId("001");  user.setName("张三");    session.save(user);    user.setName("王五");  session.update(user);    User3 user3 = new User3();  user3.setId("002");  user3.setName("李四");  session.save(user3);    //Hibernate: insert into t\_user3 (name, password, create\_time, expire\_time, user\_id) values (?, ?, ?, ?, ?)  //Hibernate: insert into t\_user3 (name, password, create\_time, expire\_time, user\_id) values (?, ?, ?, ?, ?)  //Hibernate: update t\_user3 set name=?, password=?, create\_time=?, expire\_time=? where user\_id=?  //hibernate按照save(insert),update、delete顺序提交相关操作    tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 测试assigned主键生成策略  \*  \*/  public void testSave7() {  Session session = null;  Transaction tx = null;  try {  session = HibernateUtils.getSession();  tx = session.beginTransaction();  User3 user = new User3();  user.setId("003");  user.setName("张三");    session.save(user);    user.setName("王五");  session.update(user);    session.flush();    User3 user3 = new User3();  user3.setId("004");  user3.setName("李四");  session.save(user3);    //Hibernate: insert into t\_user3 (name, password, create\_time, expire\_time, user\_id) values (?, ?, ?, ?, ?)  //Hibernate: update t\_user3 set name=?, password=?, create\_time=?, expire\_time=? where user\_id=?  //Hibernate: insert into t\_user3 (name, password, create\_time, expire\_time, user\_id) values (?, ?, ?, ?, ?)  //因为我们在session.udpate(user)后执行了flush，所以在清理缓存时执行flush前的sql不会生成  //sql会按照我们的意愿执行  tx.commit();  }catch(Exception e) {  e.printStackTrace();  tx.rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

## session的几个方法:

|  |
| --- |
| 1. session.save(user); 将user纳入到了session的管理 设置session的缓存entityEntries属性中existsInDatabase的状态为false  2. session.flash();  查找session的临时集合insertions中的user对象, 做脏数据检查 发出sql语句 清除session的临时集合insertions中的user对象 更新session的缓存entityEntries属性中existsInDatabase的状态为true  3. session.evict(user);  逐出session的缓存entityEntries属性中的user对象  4. session.commit()会隐式调用session.flush(); |

# 锁

悲观锁(并发性不好)

|  |
| --- |
| 悲观锁的实现，通常依赖于数据库机制，在整个过程中将数据锁定，其它任何用户都不能读取或修改 |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Inventory" table="t\_inventory">  <id name="itemNo">  <generator class="native"/>  </id>    <property name="itemName"/>  <property name="quantity"/>  </class>  </hibernate-mapping> |
| import org.hibernate.LockMode;  import org.hibernate.Session;  import junit.framework.TestCase;  public class PessimisticLockingTest extends TestCase {  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();  //下一行, 设置了**悲观锁**: LockMode.UPGRADE  Inventory inv = (Inventory)session.load(Inventory.class, 1, **LockMode.UPGRADE**);  System.out.println("itemName=" + inv.getItemName());  System.out.println("quantity=" + inv.getQuantity());  inv.setQuantity(inv.getQuantity() - 200);  session.update(inv);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    //下一行, 设置了**悲观锁**: LockMode.UPGRADE  Inventory inv = (Inventory)session.load(Inventory.class, 1, **LockMode.UPGRADE**);  System.out.println("itemName=" + inv.getItemName());  System.out.println("quantity=" + inv.getQuantity());  inv.setQuantity(inv.getQuantity() - 200);  session.update(inv);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

乐观锁(并发性好, 用得最多)

|  |
| --- |
| 大多数基于数据版本记录机制（version）实现，一般是在数据库表中加入一个version字段读取数据时将版本号一同读出，  之后更新数据时版本号加一，如果提交数据时版本号小于或等于数据表中的版本号，则认为数据是过期的，只当没过期才给予数据更新  乐观锁业可以使用**时间戳**. |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Inventory" table="t\_inventory" optimistic-lock="version">  <id name="itemNo">  <generator class="native"/>  </id>    <**version** name="**version**"/> <!-- 用于**乐观锁**的标识字段, **这个version字段是数据库专用于设置乐观锁的字段**, **不需要用户作标识**, **当用户每次修改, 它都会自动加1**. -->  <property name="itemName"/>  <property name="quantity"/>  </class>  </hibernate-mapping> |
| import org.hibernate.LockMode;  import org.hibernate.Session;  import junit.framework.TestCase;  public class OptimisticLockingTest extends TestCase {  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Inventory inv = (Inventory)session.load(Inventory.class, 1);    System.out.println("itemName=" + inv.getItemName());  System.out.println("version=" + inv.getVersion());  System.out.println("quantity=" + inv.getQuantity());    inv.setQuantity(inv.getQuantity() - 200);  session.update(inv);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Inventory inv = (Inventory)session.load(Inventory.class, 1);    System.out.println("itemName=" + inv.getItemName());  System.out.println("version=" + inv.getVersion());  System.out.println("quantity=" + inv.getQuantity());    inv.setQuantity(inv.getQuantity() - 200);  session.update(inv);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

## 其它锁

**对比所有字段**, 如果有一个字段不一样, 则被修改过了.

树

|  |
| --- |
| 比如说:bbs跟帖回复"树" , 文件目录"树" |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Node" table="t\_node">  <id name="id">  <generator class="native"/>  </id>    <property name="name"/>  <property name="level"/>  <property name="leaf"/>    <many-to-one name="parent" column="pid"/>    <set name="children" lazy="extra" inverse="true">  <key column="pid"/>  <one-to-many class="com.bjsxt.hibernate.Node"/>  </set>    </class>  </hibernate-mapping> |
|  |

# HQL

|  |
| --- |
| 在hql中关键字不区分大小写，但是属性和类名区分大小写 |
| <hibernate-mapping package="com.bjsxt.hibernate">  <class name="Classes" table="t\_classes">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <set name="students" inverse="true" cascade="all">  <key column="classesid"/>  <one-to-many class="Student"/>  </set>  </class>  </hibernate-mapping> |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Student" table="t\_student">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <property name="createTime"/>  <many-to-one name="classes" column="classesid"/>  <filter name="filtertest" condition="id &lt; :myid"/>  </class>    <!-- **外置命名查询: 在映射文件中采用<query>标签来定义hql** -->  <**query** name="searchStudents">  <![CDATA[  SELECT s FROM Student s where s.id<?  ]]>  </query>    <!-- **查询过滤器: 在映射文件中定义过滤器参数** -->  <**filter-def** name="filtertest">  <**filter-param** name="myid" type="integer"/>  </filter-def>  </hibernate-mapping> |

## 1、简单属性查询【重要】

\* 单一属性查询，返回结果集属性列表，元素类型和实体类中相应的属性类型一致

\* 多个属性查询,返回的集合元素是对象数组,数组元素的类型和对应的属性在实体类中的类型一致

数组的长度取决与select中属性的个数

\* 如果认为返回数组不够对象化，可以采用hql动态实例化Student对象

参见：SimplePropertyQueryTest.java

|  |
| --- |
| **import** java.util.Iterator;  **import** java.util.List;  **import** org.hibernate.Session;  **import** junit.framework.TestCase;  /\*\*  \* 简单属性查询  \* **@author** Administrator  \*  \*/  **public** **class** Test\_01\_SimplePropertyQueryTest **extends** TestCase {    /\*\*  \* 单一属性查询  \*/  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //返回结果集属性列表，元素类型和实体类中相应的属性类型一致  List students = session.createQuery("select name from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  String name = (String)iter.next();  System.*out*.println(name);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 多个属性查询  \*/  **public** **void** testQuery2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //查询多个属性，其集合元素是对象数组  //数组元素的类型和对应的属性在实体类中的类型一致  //数组的长度取决与select中属性的个数  List students = session.createQuery("select id, name from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 返回Student实体对象:select new Student(id, name)  \*/  **public** **void** testQuery3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //如果认为返回数组不够对象化，可以采用hql动态实例化Student对象, 需要在实体类中提供代参数的构造函数  //此时list中为Student对象集合  List students = session.createQuery("select new Student(id, name) from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getId() + "," + student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 使用别名, 没有as标识  \*/  **public** **void** testQuery4() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //可以使用没有as命名的别名  List students = session.createQuery("select s.id, s.name from Student s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 使用别名, 有as标识  \*/  **public** **void** testQuery5() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //可以使用as命名别名  List students = session.createQuery("select s.id, s.name from Student as s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 2、实体对象查询【重要】

\* N + 1问题，在默认情况下，使用query.iterate查询，有可以能出现N+1问题

所谓的N+1是在查询的时候发出了N+1条sql语句

1: 首先发出一条查询对象id列表的sql

N: 根据id列表到缓存中查询，如果缓存中不存在与之匹配的数据，那么会根据id发出相应的sql语句

\* list和iterate的区别？

\* list每次都会发出sql语句，list会向缓存中放入数据，而不利用缓存中的数据

\* iterate：在默认情况下iterate利用缓存数据，但如果缓存中不存在数据有可以能出现N+1问题

参见：SimpleObjectQueryTest1.java/SimpleObjectQueryTest2.java

|  |
| --- |
| **import** java.util.Iterator;  **import** java.util.List;  **import** org.hibernate.Session;  **import** junit.framework.TestCase;  /\*\*  \* 实体对象查询  \* **@author** Administrator  \*  \*/  **public** **class** Test\_02\_1\_SimpleObjectQueryTest1 **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //返回Student对象的集合  //可以忽略select  List students = session.createQuery("from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    **public** **void** testQuery2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //返回Student对象的集合  //可以忽略select,表可以使用别名  List students = session.createQuery("from Student s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  **public** **void** testQuery3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //返回Student对象的集合  //可以忽略select,表可以使用as命名别名  List students = session.createQuery("from Student as s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    **public** **void** testQuery4() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //返回Student对象的集合  //如果使用select查询实体对象，则必须采用别名  List students = session.createQuery("select s from Student as s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }      **public** **void** testQuery5() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //不支持select \* from .....这样的查询语句  List students = session.createQuery("select \* from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    /\*\*  \* 采用**list查询**  \* 发出一条查询语句，取得Student对象数据、  \*  \* Hibernate: select student0\_.id as id1\_, student0\_.name as name1\_,  \* student0\_.createTime as createTime1\_, student0\_.classesid as classesid1\_  \* from t\_student student0\_  \*  \*/  List students = session.createQuery("from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    **public** **void** testQuery2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    /\*\*  \* **迭代查询:**  \* 出现N+1问题  \*  \* 1：发出查询id列表的sql  \* Hibernate: select student0\_.id as col\_0\_0\_ from t\_student student0\_  \*  \* N：再依次发出根据id查询Student对象的sql  \* Hibernate: select student0\_.id as id1\_0\_, student0\_.name as name1\_0\_,  \* student0\_.createTime as createTime1\_0\_, student0\_.classesid as classesid1\_0\_  \* from t\_student student0\_ where student0\_.id=?  \*  \*/  Iterator iter = session.createQuery("from Student").iterate();  **while**(iter.hasNext()) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 当session缓存里 没有数据的时候,建议使用session.createQuery("from Student").list()  \* 当session缓存里 有数据的时候,建议使用session.createQuery("from Student").iterate()  \*  \* 向缓存里放数据的有:  \* 1. session.createQuery("from Student").list();  \* 2.  \*/  **public** **void** testQuery3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List students = session.createQuery("from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  System.*out*.println("---------------------------------------------");    /\*\*  \* 不会出现N+1问题  \*  \* 因为list操作已经将Student对象放到了一级缓存中，所以再次使用iterate操作的时候  \* 它首先发出一条查询id列表的sql，在根据id到缓存中去数据，只有在缓存中找不到相应的  \* 数据时，才会发出sql到数据库中查询  \*  \*/  Iterator iter = session.createQuery("from Student").iterate();  **while**(iter.hasNext()) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    **public** **void** testQuery4() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List students = session.createQuery("from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  System.*out*.println("---------------------------------------------");    /\*\*  \* 再次发出查询sql  \*  \* 在默认情况下list每次都会向数据库发出查询对象的sql，除非配置查询缓存，所以下面的list操作  \* 虽然在一级缓存中已经有了对象数据，但list默认情况下不会利用缓存，而再次发出sql  \*  \* 默认情况下，list会向缓存中放入数据，但不会利用数据  \*  \*/  students = session.createQuery("from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 3、条件查询【重要】

\* 可以采用拼字符串的方式传递参数

\* 可以采用 ? 来传递参数（索引从0开始）

\* 可以采用 :参数名 来传递参数, 如果传递多个参数，可以采用setParamterList方法

\* 在hql中可以使用数据库的函数，如：date\_format, [最好不要用, 因为换了数据库可能就不能用了]

参见：SimpleConditionQueryTest.java

|  |
| --- |
| **package** com.bjsxt.hibernate;  **import** java.text.SimpleDateFormat;  **import** java.util.Iterator;  **import** java.util.List;  **import** org.hibernate.Query;  **import** org.hibernate.Session;  **import** junit.framework.TestCase;  /\*\*  \* 条件查询  \* **@author** Administrator  \*/  **public** **class** Test\_03\_SimpleConditionQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //可以拼字符串  List students = session.createQuery("select s.id, s.name from Student s where s.name like '%1%'").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 条件查询, 使用占位符: 使用 ? 传递参数 方式  \*/  **public** **void** testQuery2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    // Query query = session.createQuery("select s.id, s.name from Student s where s.name like ?");  // query.setParameter(0, "%1%");  // List students = query.list();    //可以使用？方式传递参数  //参数的索引从0开始  //传递的参数值，不用单引号引起来  //下面使用方法链编程  List students = session.createQuery("select s.id, s.name from Student s where s.name like ?")  .setParameter(0, "%1%")  .list();    **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 条件查询,使用 :参数名称 的方式传递参数值  \*/  **public** **void** testQuery3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //使用 :参数名称 的方式传递参数值  List students = session.createQuery("select s.id, s.name from Student s where s.name like :myname")  .setParameter("myname", "%1%")  .list();    **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 条件查询,使用 :参数名称 的方式传递参数值  \* 并使用as取别名  \*/  **public** **void** testQuery4() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //使用 :参数名称 的方式传递参数值  List students = session.createQuery("select s.id, s.name from Student s where s.name like :myname and s.id = :myid")  .setParameter("myname", "%1%")  .setParameter("myid", 12)  .list();    **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 条件查询,使用 :参数名称 的方式传递数组类型参数值  \* 支持in，需要使用setParameterList进行数组类型参数值传递  \*/  **public** **void** testQuery5() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //支持in，需要使用setParameterList进行参数传递  List students = session.createQuery("select s.id, s.name from Student s where s.id in(:myids)")  .setParameterList("myids", **new** Object[]{1, 2, 3, 4, 5})  .list();    **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 在HQL中使用和数据库相关的函数  \*/  **public** **void** testQuery6() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //查询2008年2月创建的学生  List students = session.createQuery("select s.id, s.name from Student s where date\_format(s.createTime, '%Y-%m') = ?")  .setParameter(0, "2008-02")  .list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 查询某个日期段的学生  \*/  **public** **void** testQuery7() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");    //查询2008-01-10到2008-02-15创建的学生 between ? and ? 会包含前后两个数值在内.  List students = session.createQuery("select s.id, s.name from Student s where s.createTime between ? and ?")  .setParameter(0, sdf.parse("2008-01-10 00:00:00")) //构造Date对象  .setParameter(1, sdf.parse("2008-02-15 23:59:59"))  .list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 4、hibernate也支持直接使用sql进行查询, 适合批量更新

|  |
| --- |
| /\*\*  \* 使用sql: session.createSQLQuery("select \* from t\_student").list()  \* 区别于  \* 使用hql: session.createQuery("from t\_student").list()  \*/  **public** **class** Test\_04\_SqlQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List students = session.createSQLQuery("select \* from t\_student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 5、外置命名查询, 适合HQL经常变的, 把HQL写在配置文件里.

第一步: 在任何一个映射文件中采用<query>标签来定义hql, 注意名字不要重复

第二步: 在程序中采用session.getNamedQuery()方法得到hql查询串

|  |
| --- |
| <!-- 外置命名查询: 在映射文件中采用<query>标签来定义hql -->  <query name=*"searchStudents"*>  <![CDATA[  SELECT s FROM Student s where s.id < ?  ]]>  </query>  **import** java.util.Iterator;  **import** java.util.List;  **import** org.hibernate.Session;  **import** junit.framework.TestCase;  /\*\*  \* 外置命名查询:  \* 在映射文件中采用<query>标签来定义hql  \* 在程序中采用session.getNamedQuery()方法得到hql查询串  \* **@author** Administrator  \*  \*/  **public** **class** Test\_05\_NameQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List students = session.getNamedQuery("searchStudents")  .setParameter(0, 10) // sql语句使用的是 ? 传参数  .list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 6、查询过滤器, 在这个session中每个sql语句上 加上过滤条件, 类似于Spring的AOP

\* 在任何一个映射文件中定义过滤器参数

\* 在类的映射中使用这些参数

\* 在程序中启用过滤器

|  |
| --- |
| <hibernate-mapping>  <class name=*"com.bjsxt.hibernate.Student"* table=*"t\_student"*>  <id name=*"id"*>  <generator class=*"native"*/>  </id>  <property name=*"name"*/>  <property name=*"createTime"*/>  <many-to-one name=*"classes"* column=*"classesid"*/>  <filter name=*"filtertest"* condition=*"id &lt; :myid"*/> <!-- &lt; 是 小于 的意思 -->  </class>  <!-- 查询过滤器: 在映射文件中定义过滤器参数 -->  <filter-def name=*"filtertest"*>  <filter-param name=*"myid"* type=*"integer"*/>  </filter-def>  </hibernate-mapping>    **public** **class** Test\_06\_FilterQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    session.enableFilter("filtertest")  .setParameter("myid", 10);    List students = session.createQuery("from Student").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 7、分页查询【重要】

\* setFirstResult()，从0开始

\* setMaxResults, 每页显示多少条数据

|  |
| --- |
| **public** **class** Test\_07\_PageQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();  //采用方法链编程风格  List students = session.createQuery("from Student")  .setFirstResult(1)  .setMaxResults(2)  .list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 8、对象导航查询，在hql中采用 . 进行导航【重要】

|  |
| --- |
| **public** **class** Test\_08\_ObjectNavQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List students = session.createQuery("select s.name from Student s where s.classes.name like '%1%'").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  String name = (String)iter.next();  System.*out*.println(name);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 9、连接查询【重要】

|  |
| --- |
| \* 内连接: 相等的显示出来  \* 外连接（左连接/右连接）  左连接: 把左边那个表列出来, 和右边表不相等的也列出来  右连接: 把右边那个表列出来, 和左边表不相等的也列出来  **public** **class** Test\_09\_JoinQueryTest **extends** TestCase {  /\*\*  \* 内连接  \*/  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();  //Hibernate会自动把条件(student0\_.classesid = classes1\_.id)加上  //List students = session.createQuery("select c.name, s.name from Student s join s.classes c").list(); 可以省略inner为内连接,  List students = session.createQuery("select c.name, s.name from Student s inner join s.classes c").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \*左连接  \*/  **public** **void** testQuery2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();  List students = session.createQuery("select c.name, s.name from Classes c left join c.students s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \*右连接  \*/  **public** **void** testQuery3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();  List students = session.createQuery("select c.name, s.name from Classes c right join c.students s").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + "," + obj[1]);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    } |

## 10、统计查询【重要】

|  |
| --- |
| 取最大, 最小, 统计, 分组  **public** **class** Test\_10\_StatQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    // List students =session.createQuery("select count(\*) from Student").list();  // Long count = (Long)students.get(0);  // System.out.println(count);  Long count = (Long)session.createQuery("select count(\*) from Student").uniqueResult(); // uniqueResult返回唯一值, 如果有问题, 则返回null.  System.*out*.println(count);    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    **public** **void** testQuery2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //Hibernate: select classes1\_.name as col\_0\_0\_,  //count(student0\_.id) as col\_1\_0\_  //from t\_student student0\_ inner join t\_classes classes1\_ on student0\_.classesid=classes1\_.id group by classes1\_.name order by classes1\_.name    List students =session.createQuery("select c.name, count(s) from Student s join s.classes c " +  "group by c.name order by c.name").list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Object[] obj = (Object[])iter.next();  System.*out*.println(obj[0] + ", " + obj[1]);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 11、DML风格的操作（批量操作, 尽量少用，因为和缓存不同步）

|  |
| --- |
| /\*\*  \* DML风格的操作  \* DML是data manipulation language的缩写，是数据操纵语言的意思。主要用于对数据库表和视图进行操作。  \* 一般来说，DML是指批量select，insert，update，delete。  \* 不太建议使用, 因为它不会更新缓存, 容易造成缓存里的数据与数据库里的数据不一致,  \* 如果你使用了DML, 马上又拿数据, 拿到的只是缓存中的数据, 而不是数据库中真实的数据,  \* 除非这些数据变化不是很大, 不是经常被访问  \* **@author** Administrator  \*/  **public** **class** Test\_11\_DMLQueryTest **extends** TestCase {  **public** **void** testQuery1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    session.createQuery("update Student s set s.name= ? where s.id < ?")  .setParameter(0, "李四")  .setParameter(1, 5)  .executeUpdate();    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## HQL对实体的更新和删除：

|  |
| --- |
| 在Hibernate3中HQL对实体的更新  Transaction trans=session.beginTransaction();  update User as user set user.age=20 where user.id=18  　　String hql="update User as user set user.age=20 where user.id=18";  　　Query queryupdate=session.createQuery(hql);  int ret=queryupdate.executeUpdate();  trans.commit(); |
| //使用HQL语句删除数据      public void delete(int id) {          Transaction tran = this.session.beginTransaction() ;           String hql = "Delete FROM Member Where id=?" ;           Query q = this.session.createQuery(hql) ;          q.setInteger(0, id) ;          q.executeUpdate() ;           tran.commit() ;      } |

# 缓存

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | 默认打开或关闭 | 缓存对象 | get() | load() | iterate() | save | | 一级缓存(session级的缓存 或 事务级缓存) | 打开 | 实体对象 | 读写 | 读写 | 只读 | 只写 | | 二级缓存(进程级的缓存 或 SessionFactory级的缓存) | 打开 | 实体对象 | 读写 | 读写 | ? | 只写 | | 查询缓存 | 关闭 | 普通属性结果集,  对实体对象的结果集只缓存id | ? | 读写 | / | ? | |

hibernate 一级缓存(session级的缓存, 缓存实体对象)

|  |
| --- |
| 一级缓存很短和session的生命周期一致，一级缓存也叫 session级的缓存 或 事务级缓存  一级缓存是缓存实体对象的, 不会缓存普通属性.  那些方法支持一级缓存：  \* get() 会写入一级缓存, 也会使用一级缓存  \* load() 会写入一级缓存, 也会使用一级缓存  \* iterate 不写入缓存, 只使用缓存（查询实体对象）  \* save 会写入一级缓存    如何管理一级缓存：一级缓存无法取消.  \* session.clear(),session.evict()    如何避免一次性大量的实体数据入库导致内存溢出  \* 先flush，再clear    如果数据量特别大，考虑采用jdbc实现，如果jdbc也不能满足要求可以考虑采用数据本身的特定导入工具 |
| **package** com.bjsxt.hibernate;  **import** java.io.Serializable;  **import** org.hibernate.Session;  **import** junit.framework.TestCase;  **public** **class** CacheLevel1Test **extends** TestCase {  /\*\*  \* 在同一个session中发出两次load查询  \*/  **public** **void** testCache1() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student = (Student)session.load(Student.**class**, 1);  System.out.println("student.name=" + student.getName());    //不会发出sql，因为load使用缓存  student = (Student)session.load(Student.**class**, 1);  System.out.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }  }  /\*\*  \* 在同一个session中发出两次get查询  \*/  **public** **void** testCache2() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student = (Student)session.get(Student.**class**, 1);  System.out.println("student.name=" + student.getName());    //不会发出sql，因为get使用缓存  student = (Student)session.get(Student.**class**, 1);  System.out.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 在同一个session中发出两次iterate查询实体对象  \*/  **public** **void** testCache3() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student = (Student)session.createQuery("from Student s where s.id=1").iterate().next();  System.out.println("student.name=" + student.getName());    //会发出查询id的sql，不会发出查询实体对象的sql，因为iterate使用缓存  student = (Student)session.createQuery("from Student s where s.id=1").iterate().next();  System.out.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 在同一个session中发出两次iterate查询普通属性  \*/  **public** **void** testCache4() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    String name = (String)session.createQuery("select s.name from Student s where s.id=1").iterate().next();  System.out.println("student.name=" + name);    //iterate查询普通属性，一级缓存不会缓存，所以发出sql  //一级缓存是缓存实体对象的  name = (String)session.createQuery("select s.name from Student s where s.id=1").iterate().next();  System.out.println("student.name=" + name);    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 开启两个session中发出load查询  \*/  **public** **void** testCache5() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student = (Student)session.load(Student.**class**, 1);  System.out.println("student.name=" + student.getName());  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }    **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    //会发出查询语句，session间不能共享一级缓存的数据  //因为它会伴随session的生命周期存在和消亡  Student student = (Student)session.load(Student.**class**, 1);  System.out.println("student.name=" + student.getName());  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }    }    /\*\*  \* 在同一个session中先save，在发出load查询save过的数据  \*/  **public** **void** testCache6() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    Student stu = **new** Student();  stu.setName("王五");    Serializable id = session.save(stu);    //不会发出sql，因为save是使用缓存的  Student student = (Student)session.load(Student.**class**, id);  System.out.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }  }  /\*\*  \* 向数据库中批量加入1000条数据  \*/  **public** **void** testCache7() {  Session session = **null**;  **try** {  session = HibernateUtils.getSession();  session.beginTransaction();    **for** (**int** i=0; i<1000; i++) {  Student student = **new** Student();  student.setName("s\_" + i);  session.save(student);  **if** ( i % 20 == 0) {//每20条数据  session.flush();//强制session将数据持久化  session.clear();//清除缓存，避免大量数据造成内存溢出  }  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.closeSession(session);  }  }    } |

hibernate 二级缓存(SessionFactory级的缓存, 只缓存实体对象)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hibernate的二级缓存默认是打开的.  二级缓存也称 进程级的缓存 或 SessionFactory级的缓存, 一个数据库对应一个Session ，二级缓存 可以被所有的session共享  二级缓存的生命周期和SessionFactory的生命周期一致， SessionFactory可以管理二级缓存  二级缓存的配置和使用：  \* 第1步: 将echcache.xml文件拷贝到src下  \* 第2步: 开启二级缓存，修改hibernate.cfg.xml文件  <property name="hibernate.cache.use\_second\_level\_cache">true</property>  \* 第3步: 指定缓存产品提供商，修改hibernate.cfg.xml文件  <property name="hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider</property>  \* 第4步: 指定那些实体类使用二级缓存（两种方法）  \* 在映射文件中采用<cache>标签  \* 在hibernate.cfg.xml文件中，采用<class-cache>标签  二级缓存和一级缓存一样, 只缓存实体对象.  了解一级缓存和二级缓存的交互  **缓存策略提供商（Cache Providers）**   | **Cache** | **Provider class** | **Type** | **Cluster Safe (分布式缓存, 集群中热拷贝)** | **Query Cache Supported** | | --- | --- | --- | --- | --- | | Hashtable (not intended for production use) | org.hibernate.cache.HashtableCacheProvider | memory |  | yes | | EHCache | org.hibernate.cache.EhCacheProvider | memory, disk |  | yes | | OSCache | org.hibernate.cache.OSCacheProvider | memory, disk |  | yes | | SwarmCache | org.hibernate.cache.SwarmCacheProvider | clustered (ip multicast) | yes (clustered invalidation) 当缓存数据过期后, 他不会修改, 而是删除缓存, 再次访问时, 重新加载到缓存里. |  | | JBoss TreeCache | org.hibernate.cache.TreeCacheProvider | clustered (ip multicast), transactional | yes (replication) 当这个缓存中的数据修改了,他会通知其他缓存, 作相应修改. | yes (clock sync req.) | |
| <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name="hibernate.connection.url">jdbc:mysql://localhost/hibernate\_cache</property>  <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>  <property name="hibernate.connection.username">root</property>  <property name="hibernate.connection.password">bjsxt</property>  <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>  <property name="hibernate.show\_sql">true</property>    <!-- **开启二级缓存** -->  <property name="hibernate.cache.use\_second\_level\_cache">true</property>    <!-- **指定缓存产品提供商** -->  <property name="hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider</property>    <mapping resource="com/bjsxt/hibernate/Classes.hbm.xml"/>  <mapping resource="com/bjsxt/hibernate/Student.hbm.xml"/>    <!-- **指定哪些实体类使用二级缓存**（两种方法）  \* 在**映射文件中**采用<cache>标签  \* 在**hibernate.cfg.xml文件中[配置在这里, 可以一次看出所有使用二级缓存的类, 推荐使用]**，采用<class-cache>标签 usage="read-only"只读, 有时会不一致, 但效率好, 所以优先选择只读-->  <class-cache class="com.bjsxt.hibernate.Student" usage="read-only"/>  </session-factory>  </hibernate-configuration> |
| <?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>  <class name="com.bjsxt.hibernate.Student" table="t\_student">    <!--**指定哪些实体类使用二级缓存, 放在id标签前面.**  **<cache usage="read-only"/>**-->    <id name="id">  <generator class="native"/>  </id>    <property name="name"/>  <many-to-one name="classes" column="classesid"/>  </class>  </hibernate-mapping> |
| <ehcache>  <diskStore path="java.io.tmpdir 保存在磁盘上的临时目录"/>  <!--  <defaultCache  maxElementsInMemory="10000 缓存对象个数"  overflowToDisk="true or false 当它发生益处时,该怎么办? 是否保存 在磁盘上, yes"  eternal=" true or false 设置成false, 可以失效; 当设置成true时, 永远都不过期, 下面的配置就无意义了"  timeToIdleSeconds="1000 空闲: 多长秒没访问就失效."  timeToLiveSeconds="1000 能活多少秒"/>  <diskStore path="java.io.tmpdir 保存在磁盘上的临时目录"/>  -->  <defaultCache  maxElementsInMemory="10000"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  overflowToDisk="true"  />  <cache name="sampleCache1"  maxElementsInMemory="10000"  eternal="false"  timeToIdleSeconds="300"  timeToLiveSeconds="600"  overflowToDisk="true"  />  <cache name="sampleCache2"  maxElementsInMemory="1000"  eternal="true"  timeToIdleSeconds="0"  timeToLiveSeconds="0"  overflowToDisk="false"  />  </ehcache> |
| **package** com.bjsxt.hibernate;  **import** org.hibernate.CacheMode;  **import** org.hibernate.Session;  **import** org.hibernate.SessionFactory;  **import** junit.framework.TestCase;  **public** **class** CacheLevel2Test **extends** TestCase {  /\*\*  \* 开启两个session，分别调用load  \*/  **public** **void** testCache1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //不会发出sql，因为开启了二级缓存，session是共享二级缓存的  Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    }    /\*\*  \* 开启两个session，分别调用get  \*/  **public** **void** testCache2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Student student = (Student)session.get(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //不会发出sql，因为开启了二级缓存，session是共享二级缓存的  Student student = (Student)session.get(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 开启两个session，分别调用load, 再使用SessionFactory清除二级缓存  \*/  **public** **void** testCache3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    //管理二级缓存  SessionFactory factory = HibernateUtils.*getSessionFactory*();  //factory.evict(Student.class); //Student的所有对象都消失.  factory.evict(Student.**class**, 1); //第二个参数是主键, 清理指定对象.    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //会发出查询sql，因为二级缓存中的数据被清除了  Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }      /\*\*  \* 一级缓存 和 二级缓存 的交互  \*/  **public** **void** testCache4() {  Session session = **null**;    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //设置: 只向二级缓存读数据，而不向二级缓存写数据  session.setCacheMode(CacheMode.*GET*);    Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //发出sql语句，因为session设置了CacheMode为GET，所以二级缓存中没有数据  Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //设置: 只向二级缓存写数据，而不从二级缓存读数据  session.setCacheMode(CacheMode.*PUT*);    //会发出查询sql，因为session将CacheMode设置成了PUT  Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    }  } |

hibernate 查询缓存(缓存普通属性结果集, 对实体对象的结果集只缓存id)

|  |
| --- |
| Hibernate的查询缓存默认是关闭的.  缓存对象: 缓存普通属性结果集, 对实体对象的结果集只缓存id  生命周期: 当前关联的表内容发生修改，那么查询缓存生命周期结束  使用查询缓存的方法: 只对query.list()起作用，query.iterate不使用查询缓存  查询缓存的配置和使用：  第1步: 在hibernate.cfg.xml文件中启用查询缓存，如：  <property name="hibernate.cache.use\_query\_cache">true</property>  第2步: 还要在程序中, 必须手动启用查询缓存，如：  query.setCacheable(true); |
| <hibernate-configuration>  <session-factory>  <property name=*"hibernate.connection.url"*>jdbc:mysql://localhost/hibernate</property>  <property name=*"hibernate.connection.driver\_class"*>com.mysql.jdbc.Driver</property>  <property name=*"hibernate.connection.username"*>root</property>  <property name=*"hibernate.connection.password"*>root</property>  <property name=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</property>  <property name=*"hibernate.show\_sql"*>true</property>    <!-- 开启二级缓存 -->  <property name=*"hibernate.cache.use\_second\_level\_cache"*>true</property>    <!-- 指定缓存产品提供商 -->  <property name=*"hibernate.cache.provider\_class"*>org.hibernate.cache.EhCacheProvider</property>    <!-- 启用查询缓存 -->  <property name=*"hibernate.cache.use\_query\_cache"*>true</property>    <mapping resource=*"com/bjsxt/hibernate/Classes.hbm.xml"*/>  <mapping resource=*"com/bjsxt/hibernate/Student.hbm.xml"*/>    <class-cache class=*"com.bjsxt.hibernate.Student"* usage=*"read-only"*/>  </session-factory>  </hibernate-configuration> |
| **package** com.bjsxt.hibernate;  **import** java.util.Iterator;  **import** java.util.List;  **import** org.hibernate.Query;  **import** org.hibernate.Session;  **import** junit.framework.TestCase;  **public** **class** CacheLevel2Test **extends** TestCase {  /\*\*  \* 开启查询缓存，关闭二级缓存  \*  \* 开启一个session，分别调用query.list  \*/  **public** **void** testCache1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s.name from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    List names = query.list();  **for** (Iterator iter=names.iterator();iter.hasNext(); ) {  String name = (String)iter.next();  System.*out*.println(name);  }    System.*out*.println("-------------------------------------");    query = session.createQuery("select s.name from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    //没有发出查询sql，因为启用了查询缓存, 缓存了普通属性.  names = query.list();  **for** (Iterator iter=names.iterator();iter.hasNext(); ) {  String name = (String)iter.next();  System.*out*.println(name);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 开启查询缓存，关闭二级缓存  \*  \* 开启两个session，分别调用query.list  \*/  **public** **void** testCache2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s.name from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    List names = query.list();  **for** (Iterator iter=names.iterator();iter.hasNext(); ) {  String name = (String)iter.next();  System.*out*.println(name);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    System.*out*.println("-------------------------------------");    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s.name from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    //不会发出查询sql，因为**查询缓存的生命周期和session无关**  List names = query.list();  **for** (Iterator iter=names.iterator();iter.hasNext(); ) {  String name = (String)iter.next();  System.*out*.println(name);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 开启查询缓存，关闭二级缓存  \*  \* 开启两个session，分别调用query.iterate  \*/  **public** **void** testCache3() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s.name from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    **for** (Iterator iter=query.iterate();iter.hasNext(); ) {  String name = (String)iter.next();  System.*out*.println(name);  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    System.*out*.println("-------------------------------------");    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s.name from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    //**查询缓存只对query.list()起作用， query.iterate不使用查询缓存**  **for** (Iterator iter=query.iterate();iter.hasNext(); ) {  String name = (String)iter.next();  System.*out*.println(name);  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 关闭查询缓存，关闭二级缓存  \*  \* 开启两个session，分别调用query.list查询实体对象  \*/  **public** **void** testCache4() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s from Student s");    //启用查询查询缓存  //query.setCacheable(true);    List students = query.list();  **for** (Iterator iter=students.iterator();iter.hasNext(); ) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    System.*out*.println("-------------------------------------");    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s from Student s");    //启用查询查询缓存  //query.setCacheable(true);    //会发出查询sql，因为**list默认每次都会发出查询sql**  List students = query.list();  **for** (Iterator iter=students.iterator();iter.hasNext(); ) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  /\*\*  \* 开启查询缓存，关闭二级缓存  \*  \* 开启两个session，分别调用query.list查询实体对象  \*/  **public** **void** testCache5() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    List students = query.list();  **for** (Iterator iter=students.iterator();iter.hasNext(); ) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    System.*out*.println("-------------------------------------");    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    //会**发出n条查询语句**，因为**开启了查询缓存，关闭了二级缓存，那么查询缓存会缓存实体对象的id**  //所以hibernate会根据实体对象的id去查询相应的实体，**如果缓存中不存在相应的实体那**么**将发出根据实体id查询的sql语句**，否则不会发出sql使用缓存中的数据  List students = query.list();  **for** (Iterator iter=students.iterator();iter.hasNext(); ) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* 开启查询缓存，开启二级缓存  \*  \* 开启两个session，分别调用query.list查询实体对象  \*/  **public** **void** testCache6() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    List students = query.list();  **for** (Iterator iter=students.iterator();iter.hasNext(); ) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }    System.*out*.println("-------------------------------------");    **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Query query = session.createQuery("select s from Student s");    //启用查询查询缓存  query.setCacheable(**true**);    //不会发出查询sql，因为开启了二级缓存和查询缓存，**查询缓存缓存了实体对象的id列表**  //**hibernate会根据实体对象的id列表到二级缓存中取得相应的数据**  List students = query.list();  **for** (Iterator iter=students.iterator();iter.hasNext(); ) {  Student student = (Student)iter.next();  System.*out*.println(student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

# lazy策略, 常用的三个地方

|  |
| --- |
| Hibernate采用CGlib, 继承了原先的这个类, 用子类作为代理类返回给你.  JDK的动态代理, 只能对实现了接口进行代理.  lazy概念：只有真正使用该对象时，才会创建，对于hibernate而言，真正使用的时候才会发出sql  hibernate支持lazy策略只有在session打开状态下有效  Hibernate 3默认是打开的. |
| 第1种: class上的lazy, 只对普通属性起作用, 不会影响到集合<set/list> 和单端关联上的lazy特性 [常用]  <class lazy=true/false> <class>标签的lazy默认等于"true"  当class标签上的lazy=false时, 一load, 就发出sql, 查询所有普通属性, 但不查询集合属性(集合属性仍然为空), 所以说class标签上的lazy只对普通属性起作用, 不会影响到集合<set/list> 和单端关联上的lazy特性  /\*\*  \* 本测试条件: <class>标签的lazy为默认值, 等于"true"  \*/  public class ClassLazyTest extends TestCase {  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    //不会发出sql  Group group = (Group)session.load(Group.class, 1);    //不会发出sql  System.out.println("group.id=" + group.getId());    //会发出sql  System.out.println("group.name=" + group.getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  /\*\*  \* 本测试条件: <class>标签的lazy为默认值, 等于"true"  \*/  public void testLoad2() {  Session session = null;  Group group = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    group = (Group)session.load(Group.class, 1);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  //不能正确输出，抛出LazyInitializationException 异常，因为session已经关闭  //hibernate支持lazy策略只有在session打开状态下有效  System.out.println("group.name=" + group.getName());  } |
| 第2种: property上的lazy, 当property是大对象时才需要配置lazy. 需要类增强工具, 通过修改字节码达到代理功能. [不常用]  <property lazy=true/false> |
| 第3种: 集合上的lazy, 最好设置成extra, 会发送比较精巧的sql语句 [常用]  <set/list> lazy=true/false/extra 集合标签的lazy默认等于"true"  设置class上的lazy =true; 集合上的lazy =false时: 使用普通属性, 就会一并把集合上的所有数据查上来. 效率低.  设置lazy =true时, 当统计符合查询条件的条数, 发出查询所有符合条件的sql, 效率低.  设置lazy =extra时, 当统计符合查询条件的条数, 发出count统计函数sql. 效率高. |
| 第4种: 单端关联上的lazy [常用]  <one-to-one > <many-to-one > lazy=false/proxy代理/noproxy不使用代理(修改字节码, 需要增强工具)>  /\*\*  \* 所有lazy属性默认: class, 集合, 单端关联 上的lazy都等于true  \* **@author** Administrator  \*  \*/  **public** **class** SingleEndTest1 **extends** TestCase {    **public** **void** testLoad1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //不会发出sql  User user = (User)session.load(User.**class**, 1);    //会发出sql  System.*out*.println("user.name=" + user.getName());    //不会发出sql  Group group = user.getGroup();    //会发出sql  System.*out*.println("group.name=" + group.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    }  /\*\*  \* 将<many-to-one>中的lazy设置为false,其它默认  \* **@author** Administrator  \*  \*/  **public** **class** SingleEndTest2 **extends** TestCase {    **public** **void** testLoad1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //不会发出sql  User user = (User)session.load(User.**class**, 1);    //会发出sql，发出两条sql分别加载User和Group  System.*out*.println("user.name=" + user.getName());    //不会发出sql  Group group = user.getGroup();    //不会发出sql  System.*out*.println("group.name=" + group.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    }  /\*\*  \* <class>标签上的lazy=false,其它默认  \* **@author** Administrator  \*  \*/  **public** **class** SingleEndTest3 **extends** TestCase {    **public** **void** testLoad1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //会发出sql  User user = (User)session.load(User.**class**, 1);    //不会发出sql  System.*out*.println("user.name=" + user.getName());    //不会发出sql  Group group = user.getGroup();    //会发出sql  System.*out*.println("group.name=" + group.getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |
|  |

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_未分类\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 数据库隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻读 |
| Read Uncommited(未提交) | Y | Y | Y |
| Read Commited(提交读) | N | Y | Y |
| Repeateble read(可重复读,MySQL默认级别) | N | N | Y |
| Serialiazble(序列化读, 串行操作, 不能并行运行) | N | N | N |

|  |  |
| --- | --- |
| JNDI(Java Name Datebase Interface) | java名称和目录接口 |
| JDBC(本地事务) | 不同数据库,不通用 |
| JTA(Java 事务 API)(全局事务) | 两阶段提交协议(跨数据库操作) |

|  |  |
| --- | --- |
| Interceptor | 拦截器 |
| UsreType | 转换器 |
| LIfecycle |  |
| Validatable |  |

类似于Hibernate的工具:

|  |
| --- |
| Apache OJB  JDO  Toplink  EJB(CMP), JPA  IBatis |

POJO对象(最纯粹的java对象):没有继承,或实现第三方.

# 分组与排序

|  |
| --- |
| A、Order by子句：  　　与SQL语句相似，HQL查询也可以通过order by子句对查询结果集进行排序，并且可以通过asc或者desc关键字指定排序方式，如下面的代码：  　　from User user order by user.name asc,user.age desc;  　　上面HQL查询语句，会以name属性进行升序排序，以age属性进行降序排序，而且与SQL语句一样，默认的排序方式为asc,即升序排序。  　　B、Group by子句与统计查询：  　　在HQL语句中同样支持使用group by子句分组查询，还支持group by子句结合聚集函数的分组统计查询，大部分标准的SQL聚集函数都可以在HQL语句中使用，比如：count(),sum(),max(),min(),avg()等。如下面的程序代码：  　　String hql=”select count(user),user.age from User user group by user.age having count(user)>10 ”;  　　List list=session.createQuery(hql).list();  　　C、优化统计查询：  　　假设我们现在有两张数据库表，分别是customer表和order表，它们的结构如下：  　　customer  　　ID varchar2(14)  　　age number(10)  　　name varchar2(20)  　　order  　　ID varchar2(14)  　　order\_number number(10)  　　customer\_ID varchar2(14)  　　现在有两条HQL查询语句，分别如下：  1.　　from Customer c inner join c.orders o group by c.age;)  2. 　select c.ID,c.name,c.age,o.ID,o.order\_number,o.customer\_ID from Customer c inner join c.orders c group by c.age;(2)  　　这两条语句使用了HQL语句的内连接查询（我们将在HQL语句的连接查询部分专门讨论），现在我们可以看出这两条查询语句最后所返回的结果是一样的，但是它们其实是有明显区别的语句  （1）检索的结果会返回Customer与Order持久化对象，而且它们会被置于Hibernate的Session缓存之中，并且Session会负责它们在缓存中的唯一性以及与后台数据库数据的同步，只有事务提交后它们才会从缓存中被清除；而语句  （2）返回的是关系数据而并非是持久化对象，因此它们不会占用Hibernate的Session缓存，只要在检索之后应用程序不在访问它们，它们所占用的内存就有可能被JVM的垃圾回收器回收，而且Hibernate不会同步对它们的修改。  　　在我们的系统开发中，尤其是Mis系统，不可避免的要进行统计查询的开发，这类功能有两个特点：第一数据量大；第二一般情况下都是只读操作而不会涉及到对统计数据进行修改，那么如果采用第一种查询方式，必然会导致大量持久化对象位于Hibernate的Session缓存中，而且Hibernate的Session缓存还要负责它们与数据库数据的同步。而如果采用第二种查询方式，显然就会提高查询性能，因为不需要Hibernate的Session缓存的管理开销，而且只要应用程序不在使用这些数据，它们所占用的内存空间就会被回收释放。  　　因此在开发统计查询系统时，尽量使用通过select语句写出需要查询的属性的方式来返回关系数据，而避免使用第一种查询方式返回持久化对象（这种方式是在有修改需求时使用比较适合），这样可以提高运行效率并且减少内存消耗。真正的高手并不是精通一切，而是精通在合适的场合使用合适的手段。 |

# hibernater基础操作 HQL常用语句示例

|  |
| --- |
| /查询  public testbasequery(){  Session session =this.getSession();  String sql="FROM user";  Query query=session.createQuery(sql);  List list=query.list();  this.closeSession();  printlist(list);  }  //打印list  public printlist(List list){  Iterator it=list.iterator();  while(it.hasNext()){  User user=(User)it.next();  }  }  //HQl语句  String hql="from User us where us.username like ?";  Query query=session.createQuery(hql);  query.setString(0,"%itworld%");  List list=query.list();  printlist(list);  setInTeger(2,int);  setLong(3,long);  setDate(4,date);  String hql="from User us where us.username=:name and us.userpassoword=:password";  setString("username",String);  setString("userpassword",String);  String hql="select user from User user,UserInfo userinfo where user.username= usname";  //分页查询  public page(int pageNo,int pageSize){  Session session=this.getSession();  String hql="from User user order by user.id asc";  Query query=session.createQuery(hql);  int first=pageSize\*(pageNo-1);  query.setFirstResult(first);  query.setMaxResults(pageSize);  return query.list();  }  List list=page(1,15);  //统计总条数  public int getTotalCount(){  Session session=this.getSession();  String hql="select count(user) from User user";  Query query=session.createQuery(hql);  int count=(int)query.uniqueResult();  return.count;  }  //条件查询  public List search(User user){  Session session=this.getSession();  String hql="select user from User user where i=1";  if(null!=condition){  if(user.getusername()!=null && !user.getusername().equals("")){  hql += "and user.username like '%"+user.getusername+"%' ";  }  }  Query query =session.createQuery(hql);  List list= query.list();  return list;  } |

# HQL经典语句

|  |
| --- |
| Hib的检索方式  1'导航对象图检索方式。通过已经加载的对象，调用.iterator()方法可以得到order对象  如果是首次执行此方法，Hib会从数据库加载关联的order对象，否则就从缓存中得到。  2'OID检索方式。通过session的get，load方法知道了OID的情况下可以使用  3'HQL检索方式。使用面向对象的HQL查询语句session的find方法利用HQL来查询  4'QBC检索方式。利用QBCAPI来检索它是封装了基于字符串的查询语句  5'本地的SQL检索方式。使用本地数据库的SQL查询语句Hib会负责把检索到的JDBC结果集映射为持久化对象图。  五种检索方式的使用场合和特点：  HQL ： 是面向对象的查询语言，同SQL有些相似是Hib中最常用的方式。  查询设定各种查询条件。  支持投影查询，检索出对象的部分属性。  支持分页查询，允许使用having和group by  提供内制的聚集函数，sum()，min()，max()  能调用用户的自定义SQL  支持子查询，嵌入式查询  支持动态绑定参数  建议使用Query接口替换session的find方法。  Query Q = session.createQuery("from customer as c where c.name = :customerName" + "and c.age = :customerAge");  query.setString ("customerName" , "tom");  query.setInteger("customerAge" , "21");  list result = query.list();  QBC : QBCAPI提供了另一种方式，主要是Criteria接口、Criterion接口和Expression类  Criteria criteria = session.createCriteria(customer.class);  Criterion criterion1 = Expression.like("name","t%");  Criterion criterion2 = Expression.eq("age",new Integer(21));  Critera = criteria.add(criterion1) ;  Critera = criteria.add(criterion2) ;  list result = criteria.list();  或是： list result = session.createCriteria(Customer.class).add(Expression.eq("this.name","tom")).list();  SQL : 采用HQL和QBC检索时，Hib生成SQL语句适用所有数据库。  Query query = session.createSQLQuery("select {c.\*} from customers c where c.name like : customername " + "and c.age = :customerage","c",customer.calss);  query.setString("customername","tom");  query.setInteger("customerage","21");  list result = query.list();  /////////////多态查询  HQL ：session.createQuery("from employee");  QBC ：session.createCriteria(employee.class);  HQL : session.createQuery("from hourlyEmployee");  QBC : session.createCriteria(hourlyEmployee.class);  下面的HQL查询语句将检索出所有的持久化对象：  from java .lang.Object ;  from java .io.serializable ;  ////////////查询的排序  1'查询结果按照客户姓名升序排列：  HQL ：  Query query = session.createQuery ("from customer c order by c.name");  QBC ：  Criteria criteria = session.createCriteria(customer.class);  criteria.addOrder(order.asc("name"));  HQL :  Query query = session.createQuery ("from customer c order by c.name asc , c.age desc");  QBC :  Criteria criteria = session.createCriteria(customer.class);  criteria.addOrder(order.asc ("name"));  criteria.addOrder(order.desc("age"));  import net.sf.hibernate.pression.Order  import mypack.Order  ...........  Criteria criteria = session.createCritria (mypack.Order.class);  criteria.addOrder(net.sf.hibernate.Order.asc("name"));  ///////////HQL语句的参数绑定Query接口提供了绑定各种Hib映射类型的方法。  setBinary()  setString()  setBoolean()  setByte()  setCalendar()  setCharacter()  setDate()  setDouble()  setText()  setTime()  setTimestamp()  setEntity()//把参数与一个持久化类的事例绑定lsit result = session.createQuery("from order o where o.customer = :customer").setEntity("customer" , customer).list ;  setParameter()//绑定任意类型的参数  setProperties()//把命名参数与一个对象的属性值绑定 Query query = session.createQuery("from customer c where c.name =: name " + "and c.age =:age" );  Query.setProperties(customer);  过滤查询结果中的重复元素  使用Set集合来去除重复元素；或是使用distinct元素  Iterator iterator = session.createQuery("select distinct c.name from customer ").list().iterator();  while(iterator.hasnext()){  String name = (String) it.next() ;  }  ///////////使用聚集函数  count(); 记录的条数  min(); 求最小值  max(); 求最大值  avg(); 求平均值  sum(); 求和  1'查询customer中的所有记录条数  integer i = (Integer) session.createQuery("select count(\*) from customer").uniqueResult();  2'查询customer中的所有客户的平均年龄  integer i = (Integer) session.createQuery("select avg(c.age) from customer c ").uniqueResult();  3'查询customer中的客户年龄的最大值、最小值  object [] i = (Integer) session.createQuery("select max(c.age),min(c.age) from customer c ").uniqueResult();  Integer Maxage = (Integer) i [0];  Integer Minage = (Integer) i [1];  4'统计customer中的客户的名称数目，忽略重复的姓名  Integer cout = (Integer) session.createQuery("select count(distinct c.name) from customer c").uniqueResult();  ////////////使用分组查询  1'按姓名分组，统计customer中的相同姓名的记录数目  Iterator iterator = (Integer) session.createQuery("select c.name ,count(c) from customer c group by c.name").list.iterator();  while(iterator.hasnext()){  object[] p = (objcet[])iterator.next();  String name = p[0];  Integer cout = p[1];  }  2'按客户分组，统计每个客户的订单数量  Iterator iterator = session.crateQuery("select c.id ,c.name , count(o) from customer c join c.order o group by c.id ").list().iterator;  while(iterator.hasnext()){  object[] p = (objcet[])iterator.next();  Integer id = p[0]  String name = p[1];  Integer cout = p[2];  }  3'统计每个客户的订单总价  Iterator iterator = session.crateQuery("select c.id ,c.name,sum(o.price) from customer c join c.order o group by c.id").list.iterator();  while(iterator.hasnext()){  object[] p = (objcet[])iterator.next();  Integer id = p[0]  String name = p[1];  Double cout = p[2];  }  //////////报表的优化  使用HQL时如果只查询对象的一部分属性不会返回持久化对象  from customer c join c.order o group by c.age;//返回持久化对象，占用缓存。  select c.id , c.name , c.age ,o.id ,o.price from customer c join c.order o group by c.age //返回关系性数据，不占用session的缓存，可以为JVM回收  /////////HQL子查询的集合函数属性  size() 返回集合中元素的数目  minIndex() 建立索引的集合获得最小的索引  maxIndex() 建立索引的集合获得最大的索引  minElement() 对于包含基本元素的集合获得集合最小的元素  maxElement() 对于包含基本元素的集合获得集合最大的元素  element() 获得集合中的所有元素  ///////////Hibernate的缓存管理  管理一级缓存，不建议使用下列方法来管理一级缓存  evict(Object o)从缓存中清除参数指定的持久化对象  clear()清空缓存中所有的持久化对象  ///////////批量更新和批量删除  批量更新customer表中的年龄大于零的所有记录的AGE字段：  tx = session.beginTransaction();  Iterator customer = session.find("from customer c where c.age > 0").iterator();  while(iterator.hasnext()){  customer customer = (customer) customer.next();  customer.setAge(customer.getAge()+1);  }  tx.commit();  session.close();//上述代码对数据库操作效果不好可以使用下列的JDBCAPI直连    tx = session.beginTranscation();  Connection conn = session.connection();  Preparedstatement ps = conn.preparedstatment("update customer set age = age + 1 where age > 0 ");  ps.executeupdate();  tx.commit(); |