**Hibernate查询以及优化策略**

# QBC查询(了解)

## 1.1 QBC查询介绍

如果对SQL不熟悉，那么HQL语句可能不会写，Hibernate提供了一种面向对象的查询方式，不需要有SQL的基础！是另外的一种查询的方式！(不同HQL)

Query By Criteria 使用 Criteria 对象进行查询

特点：**面向对象方式的查询，不用SQL HQL！**

注意：5.2版本以后被废弃，推荐使用 **JPA** Criteria 注解

## 1.2 QBC查询测试

* 全部查询

|  |
| --- |
| /\*\*  \* 全表查询  \*/  @Test  **public** **void** test1() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Customer.**class**);  List<Customer> list = ce.list();  **for** (Customer customer : list) {  System.***out***.println(customer.getName());  }  tx.commit();  session.close();  } |

* 条件查询

|  |
| --- |
| /\*\*  \* 条件查询  \*/  @Test  **public** **void** test21() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Order.**class**);  // 添加查询条件 orderno = '201709070002'  ce.add(Restrictions.*eq*("orderno", "201809120256"));  List<Order> list = ce.list();  **for** (Order order : list) {  System.***out***.println(order.getProductName());  }  tx.commit();  session.close();  } |

* 条件查询

|  |
| --- |
| /\*\*  \* 条件查询2(多条件)  \*/  @Test  **public** **void** test22(){  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Order.**class**);  //添加查询条件 orderno like '%2017%' and productName like '%JavaWeb%'  ce.add( Restrictions.*and*( Restrictions.*like*("orderno", "%2018%") , Restrictions.*like*("productName", "%iphone%") ) );  List<Order> list = ce.list();  **for** (Order order : list) {  System.***out***.println(order.getProductName());  }  tx.commit();  session.close();  } |

* 分页查询

|  |
| --- |
| /\*\*  \* 分页查询  \*/  @Test  **public** **void** test3(){  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Order.**class**);  //分页查询  ce.**setFirstResult**(2);//起始行  ce.**setMaxResults**(2);//查询行数  List<Order> list = ce.list();  **for** (Order order : list) {  System.***out***.println(order.getProductName());  }  tx.commit();  session.close();  } |

* 查询排序

|  |
| --- |
| /\*\*  \* 查询排序  \*/  @Test  **public** **void** test4(){  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Order.**class**);  //排序 order by id desc  //因为我们的项目中也定义了Order类，所以这里的Order使用全名  ce.addOrder(org.hibernate.criterion.Order.*desc*("id"));  List<Order> list = ce.list();  **for** (Order order : list) {  System.***out***.println(order);  }  tx.commit();  session.close();  } |

* 聚合查询

|  |
| --- |
| /\*\*  \* 聚合查询  \*/  @Test  **public** **void** test5(){  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Order.**class**);    //查询总记录数 select count(id)  //ce.setProjection(Projections.rowCount());  ce.setProjection(Projections.*count*("id"));  Long count = (Long)ce.uniqueResult();    //查询id的最大值  //ce.setProjection(Projections.max("id"));  //Integer count = (Integer)ce.uniqueResult();    System.***out***.println(count);  tx.commit();  session.close();  } |

* 聚合查询

|  |
| --- |
| /\*\*  \* 投影查询  \*/  @Test  **public** **void** test6(){  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Criteria ce = session.createCriteria(Order.**class**);  //投影操作  ProjectionList pList = Projections.*projectionList*();  pList.add(Property.*forName*("orderno"));  pList.add(Property.*forName*("productName"));  ce.setProjection(pList);    List<Object[]> list = ce.list();  **for** (Object[] objects : list) {  **for** (Object object : objects) {  System.***out***.print(object+"\t");  }  System.***out***.println();  }    tx.commit();  session.close();  } |

# 2.原生SQL查询（了解）

## 2.1 Hibernate使用SQL语句介绍

本地sql查询可以直接执行 SQL语句

- 5.2以后推荐使用 createNativeQuery

- 5.2之前使用 **createSQLQuery 可以执行原生的SQL**

## 2.2 Hibernate使用SQL语句

* 查询所有

|  |
| --- |
| /\*\*  \* 以JavaBean对象封装  \*/  @Test  **public** **void** test2() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  SQLQuery sqlQuery = session.createSQLQuery("select \* from t\_order");  sqlQuery.addEntity(Order.**class**);  List<Order> list = sqlQuery.list();  **for** (Order order : list) {  System.***out***.println(order);  }  tx.commit();  session.close();  } |

* 查询所有

|  |
| --- |
| /\*\*  \* 以对象数组封装  \*/  @Test  **public** **void** test3() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  SQLQuery sqlQuery = session.createSQLQuery("select \* from t\_order");  List<Object[]> list = sqlQuery.list();  **for** (Object[] order : list) {  **for** (Object column : order) {  System.***out***.print(column);  System.***out***.print(" ");  }  System.***out***.println();  }  tx.commit();  session.close();  } |

# 延迟加载策略

延迟加载是为了减少程序和数据库的访问次数，提供程序的执行性能。

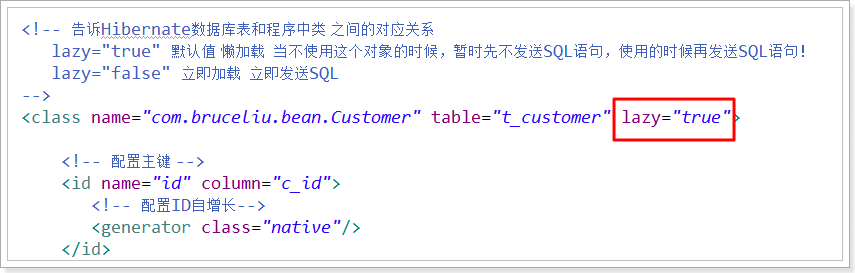
延迟加载的执行机制：

1）在查询一个对象的时候，不会到数据库查询对象的属性或者其关联的数据

2）在需要使用到对象的属性或关联数据的才会去查询数据库！

**按需加载！使用的时候就去加载 不使用的时候暂时不加载！**

## 3.1 类级别的(属性)延迟加载



|  |
| --- |
| /\*\*  \* 类级别 延迟加载  \*/  @Test  **public** **void** test1() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  // get()：get方法不支持类级别的延迟加载  /\*  \* Customer cust = session.get(Customer.class, 8L);//debug：此行查询了数据库  \* System.out.println(cust.getName());  \*/  // load()：load方法支持类级别的延迟加载  Customer cust = session.load(Customer.**class**, 8L);// debug：此行没有查询数据库  System.***out***.println(cust.getName());// debug：此行查询了数据库  tx.commit();  session.close();  } |

* 结论：

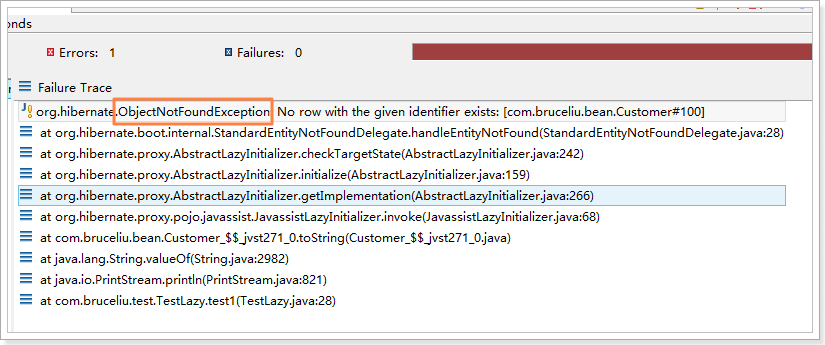
load()：只有 load 方法才支持类级别的延迟加载

get()：get 方法不支持类级别的延迟加载

使用 load() 方法的默认延迟加载策略是延迟加载，可以在配置文件中修改延迟加载策略



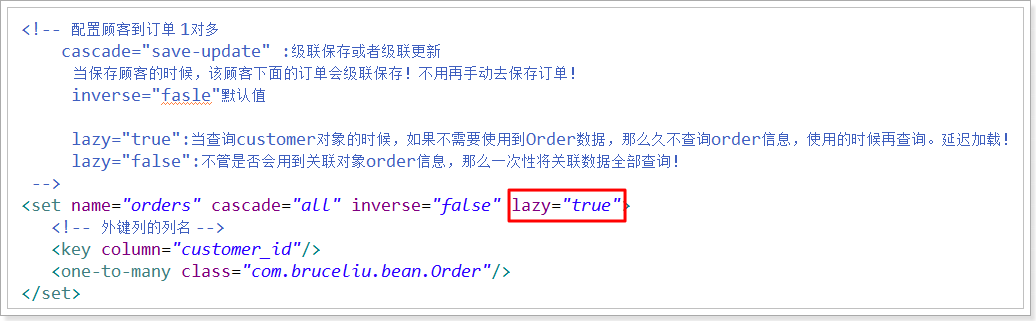
* Get和load之间区别？
* **延迟加载功能**：
  + Load在lazy=“ture”的时候具备延迟加载的功能，但是get()不具备
* **查询不到值**
  + Load()查询不到对象的时候，直接报异常！Get()返回一个null值！



## 3.2关联级别的

注意: 测试前先删除前面配置的lazy = "false"

以一对多为例



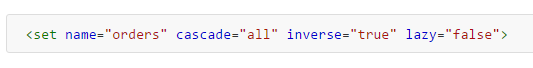
1）一方：<set/>

测试：

|  |
| --- |
| /\*\*  \* 关联级别 延迟加载（一方：<set/>）  \* 修改一对多的延迟加载配置：<set name="orders" inverse="true" lazy="false">  \*/  @Test  **public** **void** test2() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Customer cust = session.load(Customer.**class**, 8L);  // 关联订单  System.***out***.println(cust.getOrders().size()); // 延迟加载的  tx.commit();  session.close();  } |

结论：

类级别默认使用延迟加载策略，如果不想使用延迟加载策略，那么可以在配置文件中修改延迟加载策略：Customer.hbm.xml



2）多方 <many-to-one/>

测试:

|  |
| --- |
| /\*\*  \* 关联级别 延迟加载（多对一：<many-to-one/>）  \* 修改多对一延迟加载配置：  \* <many-to-one name="customer" class="Customer" column="customer\_id" cascade="all" lazy="false"/>  \*/    @Test  **public** **void** test3(){    Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();    Order order = session.load(Order.**class**, 13L);  System.***out***.println(order.getCustomer().getName()); // 延迟加载    tx.commit();  session.close();  } |

结论：

类级别默认使用延迟加载策略，如果不想使用延迟加载策略，那么可以在配置文件中修改延迟加载策略：Order.hbm.xml



|  |
| --- |
| <!-- 多对一配置  name javaBean中的属性  class 属性的全路径  colunm 对应的列名  lazy="proxy" 默认 懒加载  lazy="false" **立即加载**  lazy="no-proxy" 懒加载 (别的容器中使用)  -->  <many-to-one name=*"customer"* class=*"com.bruceliu.bean.Customer"* column=*"customer\_id"* cascade=*"all"* lazy=*"proxy"*/> |

# 抓取策略

抓取策略，是为了改变 SQL 语句查询的方式，从而提高 SQL 语句查询的效率（优化 SQL 语句）

可以设置以下三个值：

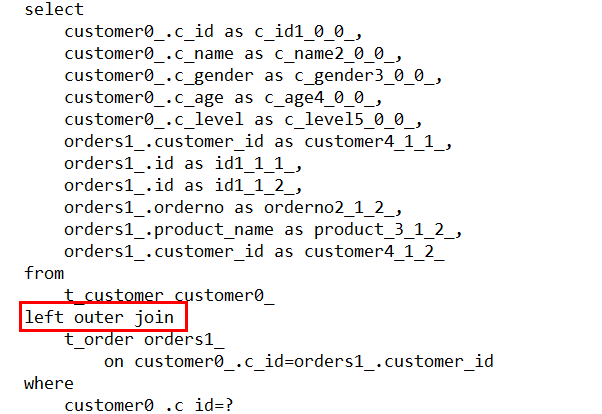
**fetch="select(默认值)|join|subselect"**

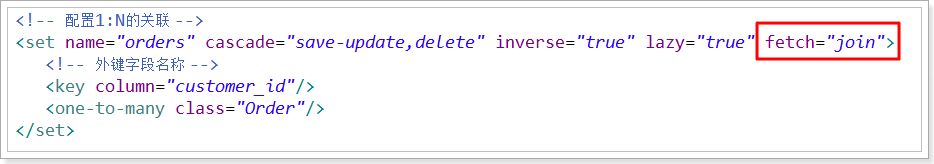
## 4.1查询一方单条记录:<set/>

在Customer.hbm.xml中配置 fetch="select" 会执行两条sql 查询多次！



配置 fetch="**join**" 会执行一条左外连接的sql语句。查询一遍得到结果！但是延迟加载的功能失效！







注意：如果配置了join，那么延迟加载就会失效！

测试

|  |
| --- |
| /\*\*  \* 一方：<set/>  \* fetch="select" : 默认情况，执行两条sql语句  \* fetch="join"： 把两条sql合并成左外连接查询（效率更高）  \* 注意：如果配置了join，那么延迟加载就会失效！  \*/  @Test  **public** **void** test2(){  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();    Customer cust = session.get(Customer.**class**, 8L);  System.***out***.println(cust.getOrders());    tx.commit();  session.close();  } |

## 4.2 查询一方多条记录: <set/>

此时无论设置 fetch="select" 还是 fetch="join" ，都会执行多条sql语句（n+1）

可以设置 fetch="subselect" ，会执行一条带有子查询的sql语句：



测试

|  |
| --- |
| /\*\*  \* 一方：<set/>  \* 需求：在查询多个一方（客户列表）的数据，关联查询多方（订单）的数据  \* 如果fetch的配置是select或join的时候，一共发出n+1条sql语句  \* fetch="subselect"： 使用子查询进行关联查询  \*/  @Test  **public** **void** test3(){    Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();    Query query = session.createQuery("from Customer");  List<Customer> list = query.list();  **for** (Customer customer : list) {  **System.*out*.println(customer.getOrders().size());**  }    tx.commit();  session.close();  } |

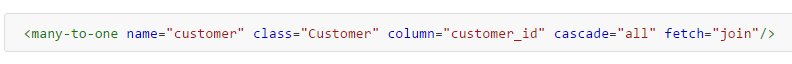
## 5.3 查询多方的记录: <many-to-one/>

Order.hbm.xml中配置 fetch="select" 会执行两条sql



配置 fetch="join" 会执行一条左外连接的sql语句。

注意：如果配置了join，那么延迟加载就会失效！



测试:

|  |
| --- |
| /\*\*  \* 多方：<many-to-one/>  \* fetch="select" : 默认情况，执行两条sql语句（支持延迟加载）  \* fetch="join"： 把两条sql合并成左外连接查询（效率更高）  \* 注意：如果配置了join，那么延迟加载就会失效！  \*/  @Test  **public** **void** test4(){    Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();    Order order = session.get(Order.**class**, 14L);  System.***out***.println(order.getCustomer());    tx.commit();  session.close();  } |

# Hibernate一级缓存(Cache) （面试题）

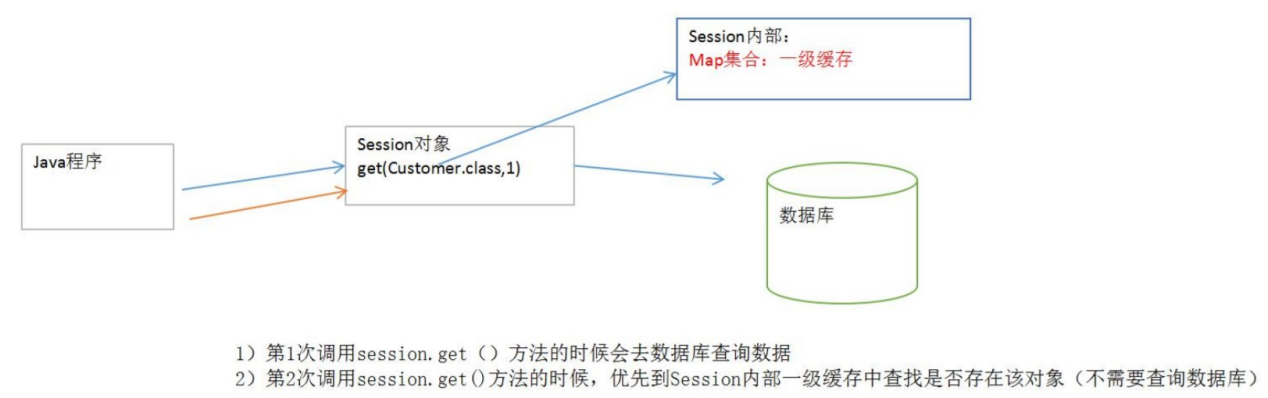
## 5.1 一级缓存介绍

一级缓存是Session级别的缓存。同一个session可以共享缓存数据！Session一旦失效，那么缓存消失！

一级缓存是默认开启的，不能手动关闭！

Hibernate的一级缓存是指**Session**（属于事务范围的缓存，由Hibernate管理，无需干预），它是一块内存空间，用来存放从数据库查询出的java对象，有了一级缓存，应用程序可以减少访问数据库的次数，提高了性能。

在使用Hibernate查询对象的时候，首先会使用对象属性的OID值（对应表中的主键）在Hibernate的一级缓存进行查找，如果找到，则取出返回，不会再查询数据库，如果没有找到，再到数据库中进行查询操作。然后将查询结果存放到Session一级缓存中。



* **一级缓存演示**

|  |
| --- |
| /\*\*  \* 使用代码来证明Hibernate的一级缓存是存在的！  \*/  @Test  **public** **void** testCache() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  // 第1次查询  Customer c1 = session.get(Customer.**class**, 10L);  System.***out***.println(c1);  // 第2次查询  Customer c2 = session.get(Customer.**class**, 10L);  System.***out***.println(c2);  // 第二次查询不触发sql语句,直接获取缓存中的结果!  tx.commit();  session.close();  } |

## 5.2 Hibernate 的快照机制

当执行 commit() 时，Hibernate同时会执行 flush() 方法，hibernate会清理session的一级缓存（flush），也就是将堆内存中的数据与快照中的数据进行对比，如果不一致，则会执行同步（update）操作，若相同，则不执行update。

1、快照（缓存）是数据的副本

2、快照属于一级缓存

3、快照是在堆内存中的

4、快照的作用：保证缓存中的数据和数据库中表数据一致性！！！

|  |
| --- |
| /\*\*  \* 说明持久态对象可以直接更新数据库的数据！  \*/  @Test  **public** **void** testAutoUpdate() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  // 获取到一个持久态对象  Customer cust = session.get(Customer.**class**, 10L);  // 修改cust的数据  cust.setName("汤姆1");  // 没有 必要执行update语句，因为现在持久态对象已经能够更新数据库的数据啦！  // session.update(cust);  tx.commit();  session.close();  } |

## 5.3 一级缓存管理

Q：如果持久态对象不在一级缓存中，可以更新数据库吗？

A：不能！

把对象移出一级缓存的方法：

session.**evict**(object) : 把一个对象移出一级缓存

session.clear() ： 把一级缓存的所有对象移出

测试：以下测试数据不会被更新

|  |
| --- |
| /\*\*  \* 一级缓存的管理  \*/  @Test  **public** **void** testEvictAndClear() {  Session session = HibernateUtil.*openSession*();  Transaction tx = session.beginTransaction();  Customer cust = session.get(Customer.**class**, 10L); // cust是持久态对象，在一级缓存  cust.setName("老王1");  // 把cust对象移出一级缓存  session.evict(cust);  // 清空一级缓存  // session.clear();  tx.commit();  session.close();  } |

# Hibernate二级缓存

Hibernate的一级缓存：就是 Session对象的缓存，而Session对象在每次操作之 后都会关闭，那么一级缓存就丢失！

结论 ：一级缓存只用于一次业务操作内的缓存。

Hibernate 的二级缓存：就是**SessionFactory 的缓存**，二级缓存和SessionFactory 对象的生命周期是一致的，SessionFactory 不消失，那么二级缓存的数据就不会丢失！

结论：二级缓存可以用于多次业务操作的缓存。

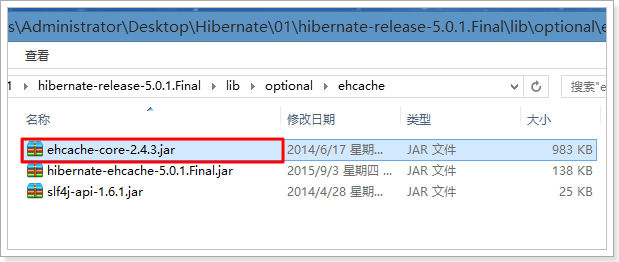
注意的问题：

1）Hibernate 一级缓存默认是开启的，而且无法关闭。

2）Hibernate 二级缓存默认是关闭的，如果使用需要开启，而且需要引入第三方的缓存工具，例如 EhCache 等。

## 6.1 添加二级缓存需要实现jar包

jar包位置: hibernate解压目录下 lib/optional/ehcache下找到相关包!



maven项目,pom文件添加

|  |
| --- |
| <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-ehcache</artifactId>  <version>5.0.1.Final</version>  </dependency> |

## 6.2 配置二级缓存

在hibernate.cfg.xml中配置以下节点:

1. property 节点 要放在 mapping 节点的上方

|  |
| --- |
| <!-- 开启 Hibernate 的二级缓存 -->  <property name=*"hibernate.cache.use\_second\_level\_cache"*>true</property>    <!-- 引入 EhCache 的工具 -->  <property name=*"hibernate.cache.region.factory\_class"*>  org.hibernate.cache.ehcache.EhCacheRegionFactory  </property> |

1. class-cache 节点 要放在 mapping 节点的下方

|  |
| --- |
| <!-- 需要缓存哪个类 -->  <class-cache usage=*"read-only"* class=*"com.bruceliu.bean.Customer"* /> |

## 6.3测试

使用二级缓存只打印1条sql，不使用二级缓存会打印2条sql

|  |
| --- |
| /\*\*  \* 演示二级缓存  \*/  @Test  **public** **void** test1() {  SessionFactory factory = **new** Configuration().configure().buildSessionFactory();  // 第1次操作  Session session1 = factory.openSession();  Customer cust1 = session1.get(Customer.**class**, 10L);  System.***out***.println(cust1.getName());  // 关闭session  session1.close();  // 第2次操作  Session session2 = factory.openSession();  Customer cust2 = session2.get(Customer.**class**, 10L);  System.***out***.println(cust2.getName());  session2.close();  } |

* 管理二级缓存

Cache cache = factory.getCache();

//cache.evictEntityRegion(Customer.class);

//cache.evictEntity(Customer.class, 10L);

cache.evictAllRegions(); //清空所有二级缓存

# 查询缓存(三级缓存)

对于经常用到的查询语句，如果其中的查询参数是固定的，则可以将这样的查询结果也存储到Hibernate的二级缓存中。

**查询缓存依赖二级缓存！！**

应用中设置了查询缓存后，查询的结果集将被存储到缓存中，查询缓存往往代价很大，如果缓存的查询不当，性能有很大影响。

**Query对象**默认情况下不读缓存，如果要使其支持缓存Query对象默认情况下不读缓存，如果要使其支持缓存.

1. 开启查询缓存的步骤：

1).配置文件中，设置配置参数hibernate.cache.use\_query\_cache=true

|  |
| --- |
| <!-- 开启查询缓存 -->  <property name=*"hibernate.cache.use\_query\_cache"*>true</property> |

2).应用代码中，调用对应的方法

* 代码测试1

|  |
| --- |
| @Test  **public** **void** test1(){    Session session = HibernateUtil.*openSession*();    Customer c1 = (Customer) session.createQuery("from Customer where name='小沈阳'").setCacheable(**true**).uniqueResult();  System.***out***.println(c1);    Customer c2 = (Customer) session.createQuery("from Customer where name='小沈阳'").setCacheable(**true**).uniqueResult();  System.***out***.println(c2);    session.close();  } |

* 代码测试2

|  |
| --- |
| @Test  **public** **void** test1(){    Session session = HibernateUtil.*openSession*();    List<Customer> list1 = session.createQuery("from Customer").setCacheable(**true**).list();  System.***out***.println(list1);    List<Customer> list2 = session.createQuery("from Customer").setCacheable(**true**).list();  System.***out***.println(list2);    session.close();  } |

# 事务（transaction）

## 8.1事务的概念

**事务指逻辑上的一组操作，组成这组操作的各个单元，要不全部成功，要不全部不成功**。  
　　 例如：A——>B转帐，对应于如下两条sql语句

  　　update from account set money=money+100 where name='B';  
 　　 update from account set money=money-100 where name='A';

## 8.2 MySQL数据库中操作事务命令

1. 编写测试SQL脚本，如下：

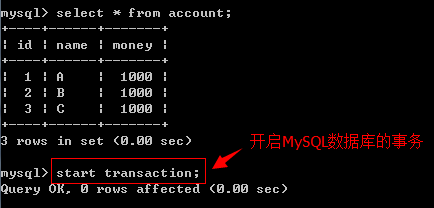
|  |
| --- |
| /\*创建账户表\*/  create table account(  id int primary key auto\_increment,  name varchar(40),  money float  );  /\*插入测试数据\*/  insert into account(name,money) values('A',1000);  insert into account(name,money) values('B',1000);  insert into account(name,money) values('C',1000); |

下面我们在MySQL数据库中模拟A——B转帐这个业务场景

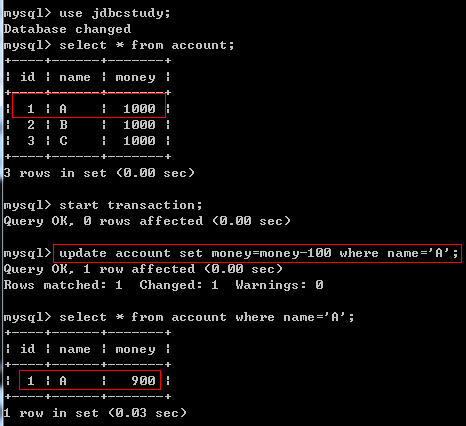
## 8.3开启事务(start transaction)

Mysql默认是自动提交提交！

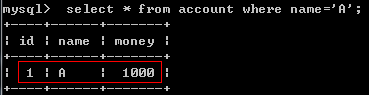
使用"start transaction"开启MySQL数据库的事务，如下所示：



我们首先在数据库中模拟转账失败的场景，首先执行update语句让A用户的money减少100块钱，如下图所示：

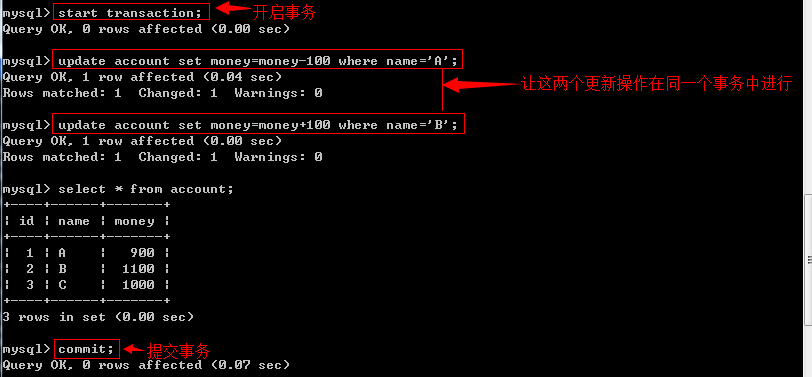


然后我们关闭当前操作的dos命令行窗口，这样就导致了刚才执行的update语句的数据库的事务没有被提交，那么我们对A用户的修改就不算是是真正的修改了，下次在查询A用户的money时，依然还是之前的1000，如下图所示：



## **9.4提交事务(commit)**

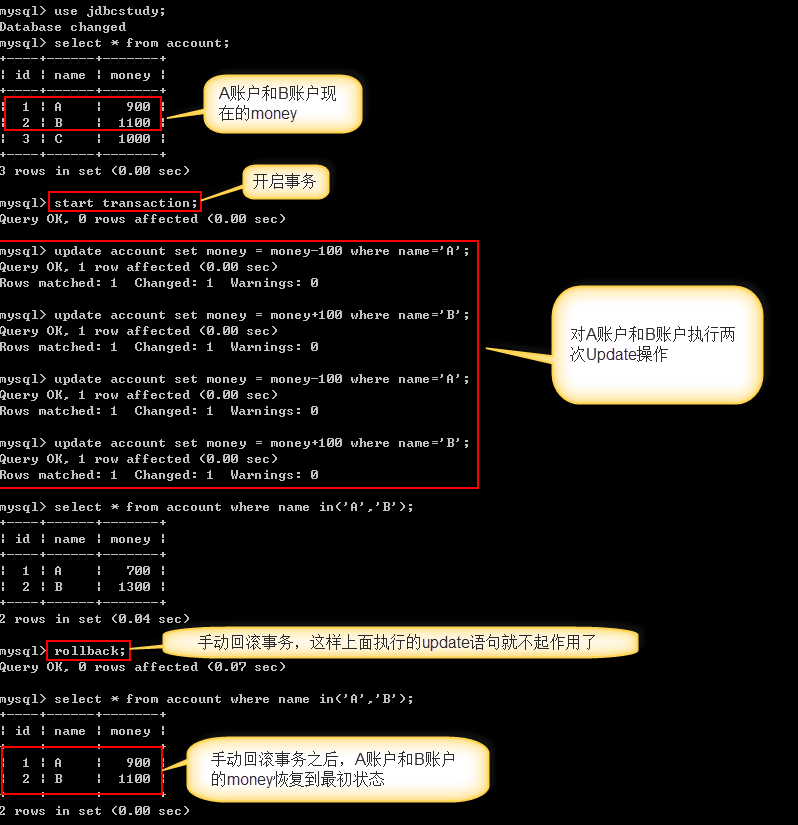
　下面我们在数据库模拟A——B转账成功的场景



我们手动提交(commit)数据库事务之后，A——B转账100块钱的这个业务操作算是真正成功了，A账户中少了100，B账户中多了100。

## **9.5回滚事务(rollback)**

* 当开启事务之后，执行相关的SQL操作，但是没有commit(提交)，那么MySQL数据库会自动的回滚rollback(撤销), 也可以手动的回滚事务！



通过手动回滚事务，让所有的操作都失效，这样数据就会回到最初的初始状态！

# 事务的并发问题

- **脏读（Dirty Read）**

一个事务读取到了另一个事务未提交的数据操作结果。这是相当危险的，因为很可能所有的操作都被回滚。

**- 不可重复读（虚读）（NonRepeatable Read）**

一个事务对同一行数据重复读取两次，但是却得到了不同的结果。例如事务T1读取某一数据后，事务T2对其做了修改，当事务T1再次读该数据时得到与前一次不同的值。

**- 幻读（Phantom Read）**

事务在操作过程中进行两次查询，第二次查询的结果包含了第一次查询中未出现的数据或者缺少了第一次查询中出现的数据，这是因为在两次查询过程中有另外一个事务插入数据造成的

# 事务的隔离级别

- 1- 读未提交

Read uncommitted：最低级别，以上情况均无法保证。

- 2- 读已提交

Read committed：可避免脏读情况发生。（Oracle默认）

- 4- 可重复读

Repeatable read：可避免脏读、不可重复读情况的发生。不可以避免虚读。（MySQl默认）

- 8- 串行化读

Serializable：事务只能一个一个执行，避免了脏读、不可重复读、幻读。执行效率慢，使用时慎重.

# Hibernate的事务隔离级别

## 12.1 配置

在Hibernate.cfg.xml中进行配置

|  |
| --- |
| <!-- 修复 hibernate 的隔离级别 -->  <property name=*"hibernate.connection.isolation"*>4</property> |

可以配置四个值:

1: read uncommited

2: read commited

4: repeatable read

8: serializeable