Hibernate\_day03总结

今日内容

* Hibernate的检索方式
* Hibernate的抓取策略
* Hibernate的批量检索
* 事务的并发问题
* Hibernate的二级缓存
* Hibernate的查询缓存

## 上次课内容回顾:

1.Hibernate一级缓存:

\* 缓存:优化手段.

\* Hibernate中两个基本缓存:

\* 一级缓存:与session的生命周期一致.

\* 二级缓存:与sessionFactory的生命周期一致.

\* 证明一级缓存的存在:

\* session的生命周期开始

\* 查询了一下Book.//发送SQL去查询

\* 查询了一个Book.//不发送SQL.

\* session的生命周期结束

\* 一级缓存的管理:

\* clear();

\* evict(Object obj);

\* flush();

\* refresh(Object obj);

\* 一级缓存的刷出时机:

\* ALWAYS :

\* AUTO :默认值

\* COMMIT :事务提交/session.flush()时候

\* MANUAL :手动调用session.flush()刷出缓存

\* 持久化对象的操作的常用方法:

1. Hibernate为了管理持久化对象:

\* 将持久化对象分成三种状态:

\* 瞬时态:

\* 没有唯一标识OID,且没有与session关联.

\* 持久态:

\* 有唯一的标识OID,且与session关联.

\* 脱管态:

\* 有唯一的标识OID,没有与session关联.

\* 瞬时态对象:

\* Book book = new Book();

\* 瞬时-->持久:

\* save()/saveOrUpdate();

\* 瞬时-->脱管:

\* book.setBid(1);

\* 持久态对象:

\* get()/load()/find()/lock();

\* 持久态-->瞬时:

\* delete();

\* 持久态-->脱管:

\* close()/clear()/evict();

\* 脱管态对象:

\* Book book = new Book();

\* book.setBid(1);

\* 脱管-->瞬时:

\* book.setBid(null);

\* 脱管-->持久:

\* update();/saveOrUpdate();

1. Hibernate的关联关系配置:

\* 一对多:

\* 多对多:

\* 一对一:

一对多:

\* 实体:

private class Customer{

private Set<Order> orders = new HashSet<Order>;

}

private class Order{

private Customer customer;

}

\* 映射文件:

\* 在多的一方Order.hbm.xml

<many-to-one />

\* 在一的一方Customer.hbm.xml

<set>

<key />

<one-to-many />

</set>

\* cascade:级联操作:

\* save-update:

\* delete

\* all

\* delete-orphan

\* all-delete-orphan

\* inverse:放弃外键维护权

\* 通常在一的一方放弃.

多对多:

实体:

private class Student{

private Set<Course> courses = new HashSet<Course>();

}

private class Course{

private Set<Student> students = new HashSet<Student>();

}

映射文件:

在多对多的双方需要配置:

<set>

<key />

<many-to-many />

</set>

\*\*\*\*\* 保存多对多双方的时候,需要有一方放弃外键维护权.

## Hibernate的检索方式:

### Hibernate的检索方式:

检索方式:查询的方式:

**导航对象图检索方式: 根据已经加载的对象导航到其他对象**

**\* Customer customer = (Customer)session.get(Customer.class,1);**

**\* customer.getOrders();// 获得到客户的订单**

**OID 检索方式: 按照对象的 OID 来检索对象**

\* get()/load();方法进行检索.

**HQL 检索方式: 使用面向对象的 HQL 查询语言**

**\* Query query = session.createQuery(“HQL”);**

**QBC 检索方式: 使用 QBC(Query By Criteria) API 来检索对象. 这种 API 封装了基于字符串形式的查询语句, 提供了更加面向对象的查询接口.**

**\* Criteria criteria = session.createCriteria(Customer.class);**

**本地 SQL 检索方式: 使用本地数据库的 SQL 查询语句**

**\* SQLQuery query = session.createSQLQuery(“SQL”);**

### HQL:

HQL:Hibernate Query Language:

\* 特点:

\* 面向对象的查询:

\* 支持方法链编程:

\* 使用:

1. 创建Query接口.
2. 查询所有记录:

List<Customer> list = session.createQuery("from Customer").list();

for (Customer customer : list) {

System.out.println(customer);

}

1. 查询使用别名:

// 使用别名

// 别名as可以省略

/\* List<Customer> list =

session.createQuery("from Customer c").list();

System.out.println(list);\*/

// 使用别名:带参数

/\*List<Customer> list = session

.createQuery("from Customer as c where c.cname = ?")

.setString(0, "小沈").list();

System.out.println(list);\*/

// 不支持 select \* from Customer写法.可以写成 select 别名 from Customer as 别名;

List<Customer> list = session.createQuery("select c from Customer c").list();

System.out.println(list);

1. 排序:

List<Customer> list = session.createQuery(

"from Customer c order by c.id desc").list();

for (Customer customer : list) {

System.out.println(customer);

}

1. 分页查询:

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

query.setFirstResult(20);

query.setMaxResults(10);

List<Order> list = query.list();

for (Order order : list) {

System.out.println(order);

}

1. 单个对象查询:

Customer customer = (Customer) session

.createQuery("from Customer where cname = ?")

.setString(0, "小明").uniqueResult();

System.out.println(customer);

1. 参数绑定:

// 1.使用?号方式绑定

/\*Query query = session.createQuery("from Customer where cname = ?");

query.setString(0, "小沈");

List<Customer> list = query.list();

System.out.println(list);\*/

/\*Query query = session.createQuery("from Customer where cname = ? and cid =?");

query.setString(0, "小沈");

query.setInteger(1,3);

List<Customer> list = query.list();

System.out.println(list);\*/

// 2.使用名称的方式绑定

Query query = session.createQuery("from Customer where cname=:name and cid=:id");

/\*query.setString("name", "小沈");

query.setInteger("id", 3);\*/

query.setParameter("name", "小沈");

query.setParameter("id", 3);

List<Customer> list = query.list();

System.out.println(list);

// 3.绑定实体

List<Order> list = session

.createQuery("from Order o where o.customer = ?")

.setEntity(0, customer).list();

for (Order order : list) {

System.out.println(order);

}

1. 投影操作:

// 查询客户的名称:

/\*

\* List<Object> list = session.createQuery(

\* "select c.cname from Customer c").list();

\* System.out.println(list);

\*/

/\*

\* List<Object[]> list = session.createQuery(

\* "select c.cid,c.cname from Customer c").list();

\* for (Object[] objects: list) {

\* System.out.println(Arrays.toString(objects));

\* }

\*/

List<Customer> list = session.createQuery(

"select new Customer(cname) from Customer").list();

System.out.println(list);

1. 模糊查询:

Query query = session.createQuery("from Customer where cname like ?");

query.setParameter(0, "小%");

List<Customer> list = query.list();

System.out.println(list);

1. 使用聚集函数:

Long count = (Long) session.createQuery("select count(\*) from Order")

.uniqueResult();

System.out.println(count);

1. 命名查询:

List<Customer> list = session.getNamedQuery("findAll").list();

System.out.println(list);

SQL多表查询:

\* 连接:

\* 交叉连接:

\* select \* from A,B;

\* 内连接:查询的是两个表的交集!

\* select \* from A inner join B on A.字段 = B.字段;

\* 隐式内连接:

\* select \* from A,B where A.字段 = B.字段;

\* 外连接:

\* 左外连接:

\* select \* from A left outer join B on A.字段 = B.字段;

\* 右外连接:

\* select \* from A right outer join B on A.字段 = B.字段;

HQL多表的查询:

\* 连接:

\* 交叉连接:

\* 内连接:

\* 隐式内连接:

**\* 迫切内连接:**

\* 左外连接:

**\* 迫切左外连接:**

\* 右外连接:

\* HQL的内连接和迫切内连接区别:

\* 内连接查询 :将数据封装一个List<Object[]>中.

\* 迫切内连接 :将数据封装一个List<Customer>中.但是迫切内连接,得到会有重复记录 ,需要使用distinct排重.

### QBC:

1. 查询所有记录:

List<Customer> list = session.createCriteria(Customer.class).list();

for (Customer customer : list) {

System.out.println(customer);

}

1. 排序:

List<Customer> list = session.createCriteria(Customer.class)

.addOrder(org.hibernate.criterion.Order.desc("id")).list();

for (Customer customer : list) {

System.out.println(customer);

}

1. 分页:

Criteria criteria = session.createCriteria(Order.class);

criteria.setFirstResult(10);

criteria.setMaxResults(10);

List<Order> list = criteria.list();

for (Order order : list) {

System.out.println(order);

}

1. 获取单个对象:

Customer customer = (Customer) session.createCriteria(Customer.class)

.add(Restrictions.eq("cname", "小明")).uniqueResult();

System.out.println(customer);

1. 带参数的查询:

/\*

\* List<Customer> list = session.createCriteria(Customer.class)

\* .add(Restrictions.eq("cname", "小明")).list();

\* System.out.println(list);

\*/

List<Customer> list = session.createCriteria(Customer.class)

.add(Restrictions.eq("cname", "小明"))

.add(Restrictions.eq("cid", 2)).list();

System.out.println(list);

1. 模糊查询:

Criteria criteria = session.createCriteria(Customer.class);

criteria.add(Restrictions.like("cname", "大%"));

List<Customer> list = criteria.list();

System.out.println(list);

### SQL:

1. SQL语句查询所有记录:

List<Object[]> list = session.createSQLQuery("select \* from customer").list();

for (Object[] objects : list) {

System.out.println(Arrays.toString(objects));

}

List<Customer> list = session.createSQLQuery("select \* from customer")

.addEntity(Customer.class).list();

for (Customer customer : list) {

System.out.println(customer);

}

## Hibernate的抓取策略

### 区分延迟和立即检索:

立即检索:

\* 当执行某行代码的时候,马上发出SQL语句进行查询.

\* get()

延迟检索:

\* 当执行某行代码的时候,不会马上发出SQL语句进行查询.当真正使用这个对象的时候才会发送SQL语句.

\* load();

类级别检索和关联级别检索:

\* 类级别的检索:

\* ＜class>标签上配置lazy.

\* 关联级别的检索:

\* <set>、<many-to-one>上面的lazy.

\* 查询某个对象的时候,是否需要查询关联对象?

\* 查询关联对象的时候是否采用延迟检索?

**从一的一方关联多的一方:**

**\* <set>**

**\* fetch:控制sql语句的类型**

**\* join :发送迫切左外连接的SQL查询关联对象.fetch=”join”那么lazy被忽略了.**

**\* select :默认值,发送多条SQL查询关联对象.**

**\* subselect :发送子查询查询关联对象.(需要使用Query接口测试)**

**\* lazy:控制关联对象的检索是否采用延迟.**

**\* true :默认值, 查询关联对象的时候使用延迟检索**

**\* false :查询关联对象的时候不使用延迟检索.**

**\* extra :及其懒惰.**

**\*\*\*\*\* 如果fetch是join的情况,lazy属性将会忽略.**

**在多的一方关联一的一方:**

**\* <many-to-one>**

**\* fetch:控制SQL语句发送格式**

**\* join :发送一个迫切左外连接查询关联对象.fetch=”join”,lay属性会被忽略.**

**\* select :发送多条SQL检索关联对象.**

**\* lazy:关联对象检索的时候,是否采用延迟**

**\* false :不延迟**

**\* proxy :使用代理.检索订单额时候,是否马上检索客户由Customer对象的映射文件中<class>上lazy属性来决定.**

**\* no-proxy :不使用代理**

### 比较三种检索策略

|  |  |  |  |
| --- | --- | --- | --- |
| **检索策略** | **优点** | **缺点** | **优先考虑使用的场合** |
| 立即检索 | 对应用程序完全透明，不管对象处于持久化状态还是游离状态，应用程序都可以从一个对象导航到关联的对象 | (1)select语句多  (2)可能会加载应用程序不需要访问的对象，浪费许多内存空间。 | (1)类级别  (2)应用程序需要立即访问的对象  (3)使用了二级缓存 |
| 延迟检索 | 由应用程序决定需要加载哪些对象，可以避免执行多余的select语句，以及避免加载应用程序不需要访问的对象。因此能提高检索性能，并节省内存空间。 | 应用程序如果希望访问游离状态的代理类实例，必须保证她在持久化状态时已经被初始化。 | (1)一对多或者多对多关联  (2)应用程序不需要立即访问或者根本不会访问的对象 |
| 迫切左外连接检索 | (1)对应用程序完全透明，不管对象处于持久化状态还是游离状态，都可从一个对象导航到另一个对象。  (2)使用了外连接，select语句少 | (1)可能会加载应用程序不需要访问的对象，浪费内存。  (2)复杂的数据库表连接也会影响检索性能。 | (1)多对一或一对一关联  (2)需要立即访问的对象  (3)数据库有良好的表连接性能。 |

### 批量抓取

都在一的一方配置

一对多：通过客户批量抓取订单

客户一端配置

<set>集合上配置batch-size="3",该值代表每次批量抓取客户的个数.

List<Order> list = session.createQuery("from Order").list();

for (Order order : list) {

System.out.println(order.getCustomer().getCname());

}

多对一：通过订单批量抓取客户:

需要在客户一端<class>标签上配置batch-size，

List<Customer> list = session.createQuery("from Customer").list();

for (Customer customer : list) {

for (Order order : customer.getOrders()) {

System.out.println(order.getAddr());

}

}

## Hibernate的事务处理:

事务:

\* 事务就是逻辑上的一组操作,要么全都成功,要么全都失败!!!

事务特性:

\* 原子性:事务一组操作不可分割.

\* 一致性:事务的执行前后,数据完整性要保持一致.

\* 隔离性:一个事务在执行的过程中不应该受到其他事务的干扰.

\* 持久性:一旦事务结束,数据就永久保存数据库.

如果不考虑事务的隔离性引发一些安全性问题:

\* 5大类问题:3类读问题 2类写问题.

\* 读问题:

\* 脏读 :一个事务读到另一个事务未提交数据.

\* 不可重复读 :一个事务读到另一个事务已经提交数据(update),导致查询结果不一致.

\* 虚读 :一个事务读到另一个事务已经提交的数据(insert),导致查询结果不一致

\* 避免三种读的问题:

\* 设置事务的隔离级别:

\* 未提交读:以上三种读问题 都有可能发生.

\* 已提交读:避免脏读,但是不可重复读和虚读有可能发生.

\* 重复读:避免脏读和不可重复读,但是虚读是有可能发生.

\* 串行的:可以避免以上三种读问题.

\* 在Hibernate中设置事务的隔离级别:

\* 在核心配置文件中:

<property name="hibernate.connection.isolation">4</property>

MYSQL ---- 4

Orical ---- 2

\* 写问题:丢失更新

\* 解决;

\* 悲观锁:

\*

\* 乐观锁;

\*

线程绑定的session:

\* 在Hibernate.cfg.xml中配置一个:

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

\* 使用SessionFactory中的getCurrentSession();方法.

\* 底层就是ThreadLocal.

**\*\*\*\*\* 当前线程中的session不需要进行关闭,线程结束后自动关闭!!!**