hibernate

# 基础部分

## hibernate与jdbc的优缺点对比

### jdbc缺点

1、编程的时候很繁琐，用的try和catch比较多

2、jdbc没有做数据的缓存

3、没有做到面向对象编程

4、sql语句的跨平台性很差

### jdbc的优点

效率比较高

### hibernate的优点

1、完全的面向对象编程

2、hibernate的缓存很牛的，一级缓存，二级缓存，查询缓存 重点

3、编程的时候就比较简单了

4、跨平台性很强

5、使用场合就是企业内部的系统

### hibernate的缺点

1、效率比较低

2、表中的数据如果在千万级别，则hibernate不适合

3、如果表与表之间的关系特别复杂，则hibernate也不适合

## 利用hibernate实现crud的操作

### 步骤

#### 1、创建一个工程

#### 2、导入jar包

说明：

javassist包是用来创建代理对象的

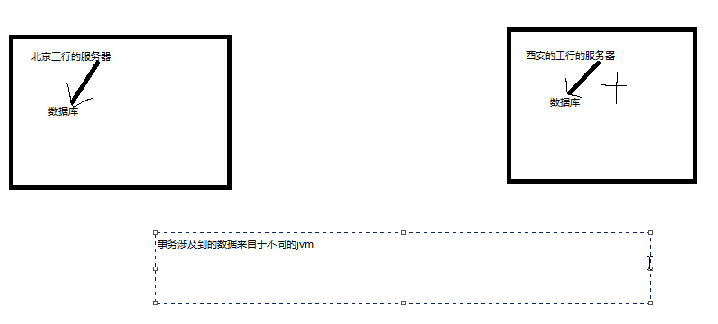
代理对象的三种创建方式：

1、jdkproxy

2、cglib

3、javassist

jta: Java Transaction API，JTA允许应用程序执行分布式事务处理，是sun公司给分布式事务处理出来的规范



#### 3、hibernate.cfg.xml

主要的用途：

告诉hibernate连接数据库的信息，用的什么样的数据库(方言)

根据持久化类和映射文件生成表的策略

<?xml version=*'1.0'* encoding=*'utf-8'*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<!--

一个sessionFactory代表数据库的一个连接

-->

<session-factory>

<!-- 链接数据库的用户名 -->

<property name=*"connection.username"*>root</property>

<!-- 链接数据库的密码 -->

<property name=*"connection.password"*>root</property>

<!-- 链接数据库的驱动 -->

<property name=*"connection.driver\_class"*>

com.mysql.jdbc.Driver

</property>

<!-- 链接数据库的url -->

<property name=*"connection.url"*>

jdbc:mysql://localhost:3306/itheima12\_hibernate

</property>

<!--

方言

告诉hibernate使用什么样的数据库，hibernate就会在底层拼接什么样的sql语句

-->

<property name=*"dialect"*>

org.hibernate.dialect.MySQLDialect

</property>

<!--

根据持久化类生成表的策略

validate 通过映射文件检查持久化类与表的匹配

update 每次hibernate启动的时候，检查表是否存在，如果不存在，则创建，如果存在，则什么都不做了

create 每一次hibernate启动的时候，根据持久化类和映射文件生成表

create-drop

-->

<property name=*"hbm2ddl.auto"*>update</property>

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

<mapping

resource=*"com/itheima12/hibernate/domain/Person.hbm.xml"* />

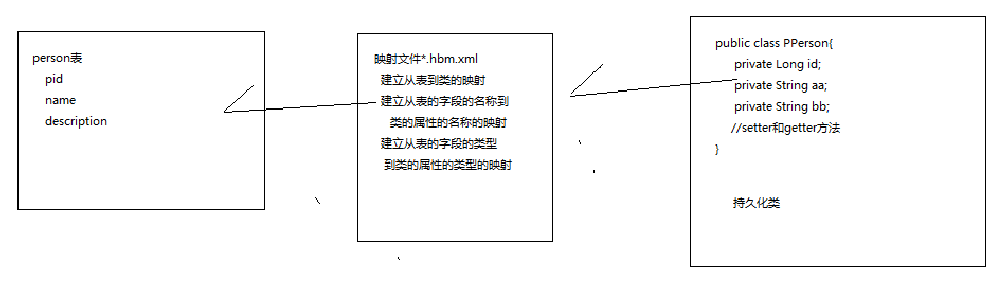
<mapping

resource=*"com/itheima12/hibernate/utils/Person.hbm.xml"* />

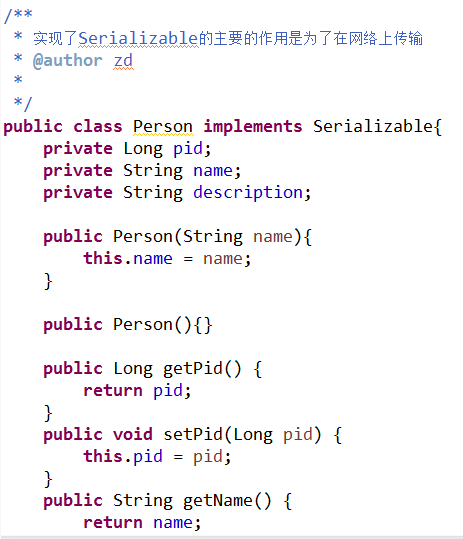
</session-factory>

</hibernate-configuration>

#### 4、持久化类和映射文件



持久化类：



映射文件：

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<!--

class用来描述一个类

name 类的全名

table 该持久化类对应的表名 可以不写，默认值为类名

catalog 数据库的名称

-->

<class name=*"com.itheima12.hibernate.domain.Person"* table=*"person"*>

<!--

用来描述主键

name 属性的名称

column 属性的名称对应的表的字段 可以不写 默认值就是属性的名称

length 属性的名称对应的表的字段的长度 如果不写，默认是最大的长度

-->

<id name=*"pid"* column=*"pid"* length=*"5"*>

<!--

主键的产生器

-->

<generator class=*"increment"*></generator>

</id>

<property name=*"name"* length=*"20"* type=*"string"*></property>

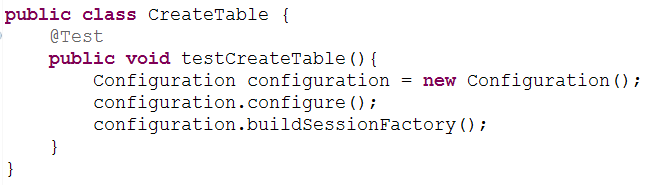
<property name=*"description"* length=*"50"* type=*"java.lang.String"*></property>

</class>

</hibernate-mapping>

#### 5、生成表

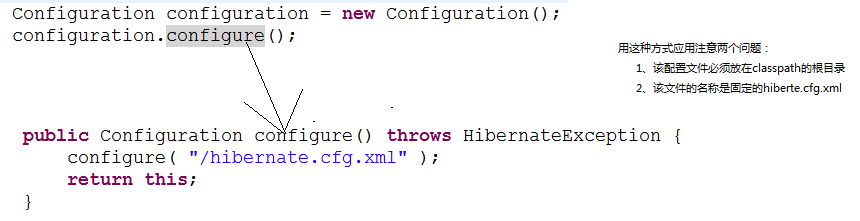
欧美国家的公司一般是先根据需求建立e-r图，再建类，再根据类生成表， hibernate配置文件中的*hbm2ddl.auto*就是为了解决这种需求的：根据类生成表。 而在中国则正好相反



#### 6、客户端

##### 说明

hibernate.cfg.xml文件的加载

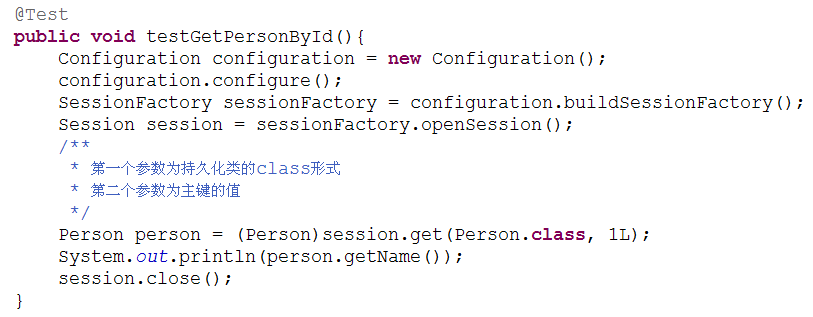




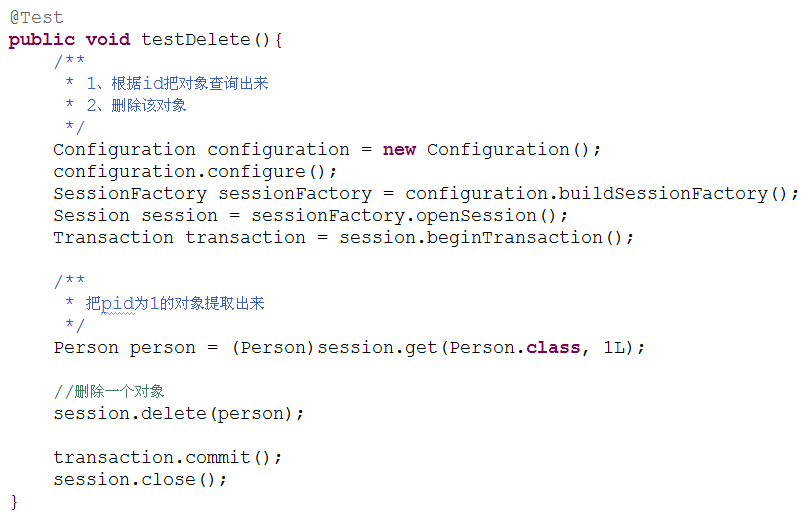
可以指定配置文件的路径，配置文件可以随意放

##### 增删改查操作









##### sessionFactory

1、hibernate中的配置文件、映射文件、持久化类的信息都在sessionFactory中

2、sessionFactory中存放的信息都是共享的信息

3、sessionFactory本身就是线程安全的，因为sessionFactory只有在服务器启动的时候将信息加载进来，只加载一次

4、一个hibernate框架sessionFactory只有一个

5、sessionFactory是一个重量级别的类，因为sessionFactory中存放了配置文件、映射文件、持久化类的信息

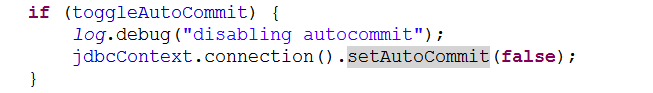
##### session

1、得到了一个session，相当于打开了一次数据库的连接

2、在hibernate中，对数据的crud操作都是由session来完成的

##### transaction

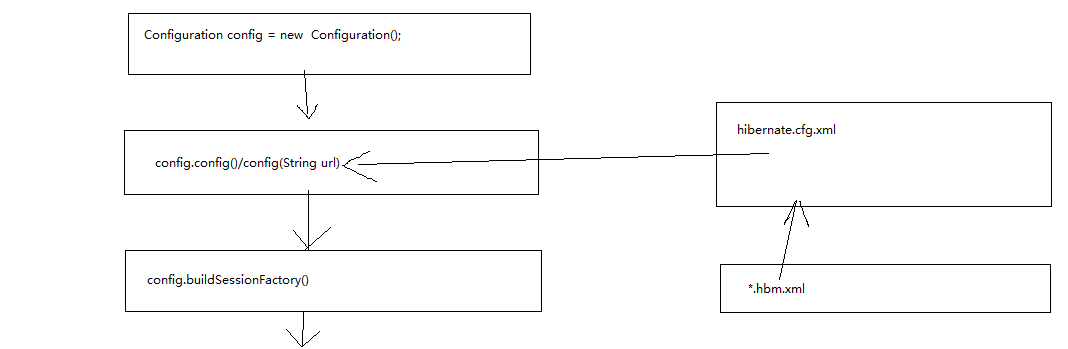
hibernate中的事务默认不是自动提交的

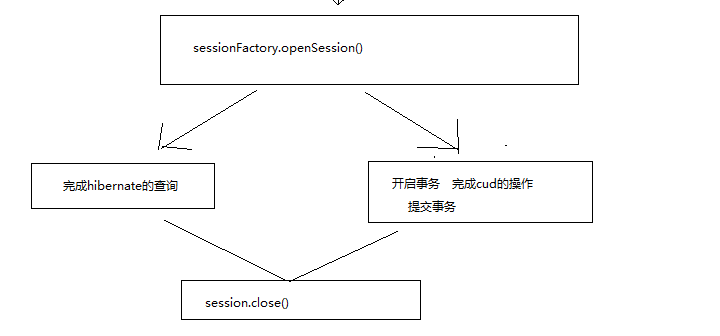


设置了connection的setAutoCommit为false

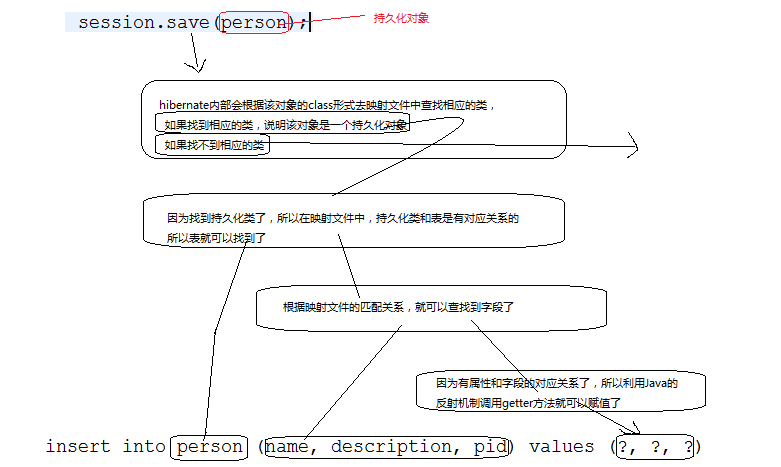
只有产生了连接，才能进行事务的操作。所以只有有了session以后，才能有transaction

##### 流程



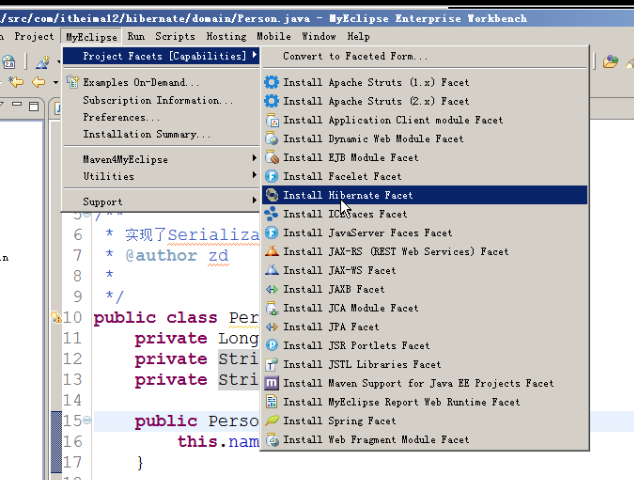


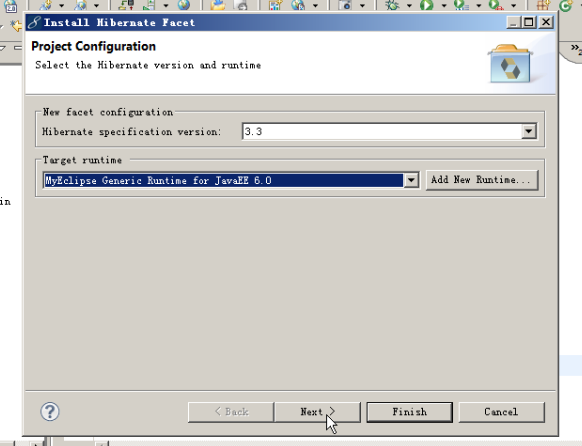
##### 内部执行原理

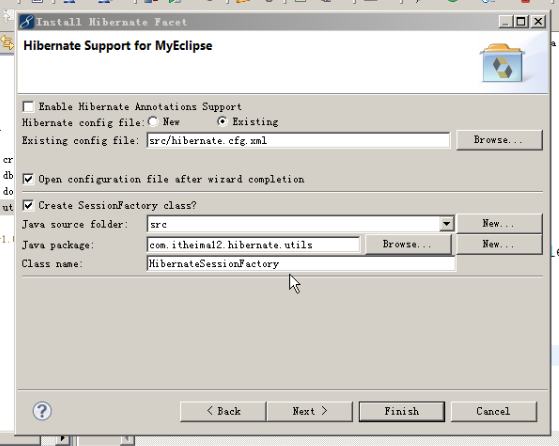


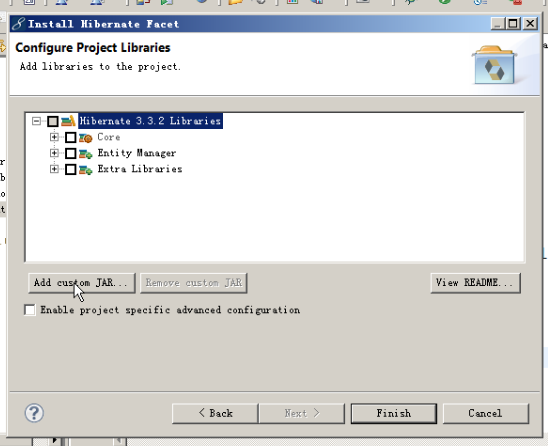
## 根据表生成持久化类和映射文件

### 把一个工程变成hibernate工程

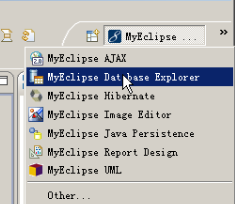


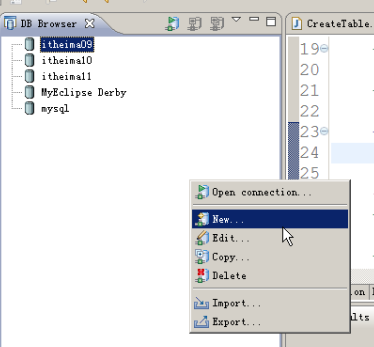


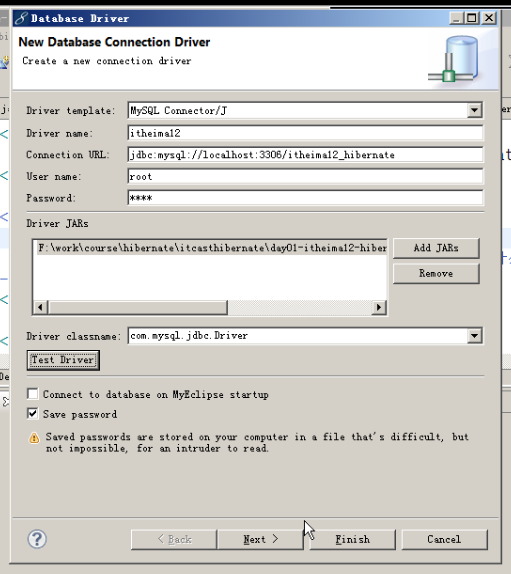


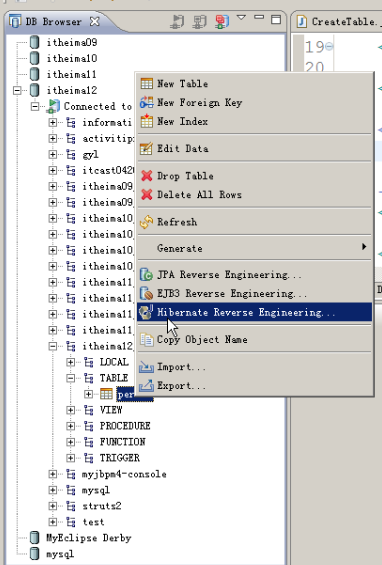


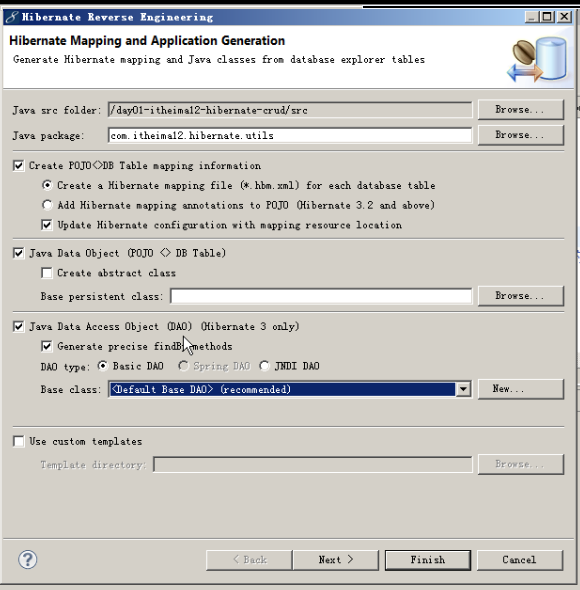
### 把数据库引入到myeclipse中

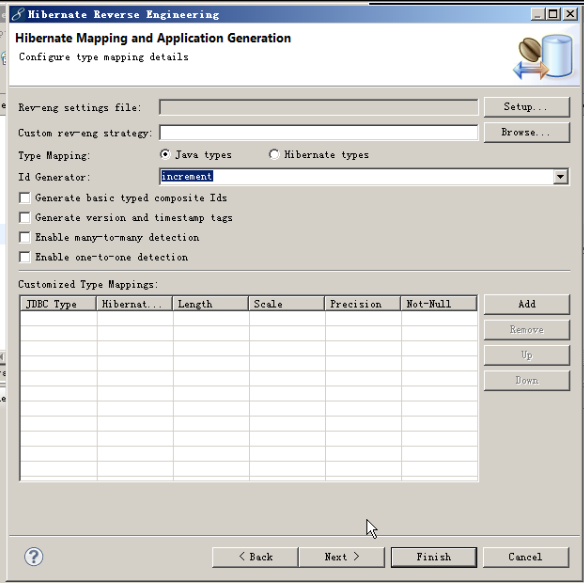












## 类型

### hibernate类型

这个效率比较低（要先找到映射关系）

### java类型

该种类型的效率比较高（直接映射了）

在hibernate内部，维护了一张表，这张表中有如下的映射关系：

hibernate类型 java类型 数据库的字段的类型

string java.lang.String varchar/varchar2



## 主键生成器

### increment

/\*\*

\* select max(pid) from person

\* 先找到主键的最大值，在最大值基础上加1

\* 所以该主键必须是数字类型

\*/

@Test

**public** **void** testIncrement(){

Session session = HibernateUtils.*sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Person person = **new** Person();

person.setName("aaa");

session.save(person);

transaction.commit();

session.close();

}

### Assigned

/\*\*

\* 在程序中手动的设置主键的值

\*/

@Test

**public** **void** testAssigned(){

Session session = HibernateUtils.*sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Person person = **new** Person();

person.setName("aaa");

person.setPid(2L);

session.save(person);

transaction.commit();

session.close();

}

### identity

/\*\*

\* 表的自动增长机制

\* 主键必须是数字类型

\* 该效率比increment要高，但是id值不连续

\*/

@Test

**public** **void** testIdentity(){

Session session = HibernateUtils.*sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Person person = **new** Person();

person.setName("aaa");

session.save(person);

transaction.commit();

session.close();

}

### uuid

/\*\*

\* uuid的字符串是由hibernate内部生成的

\* 要求主键必须是字符串类型

\*/

@Test

**public** **void** testUUID(){

Session session = HibernateUtils.*sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Person person = **new** Person();

person.setName("aaa");

session.save(person);

transaction.commit();

session.close();

}

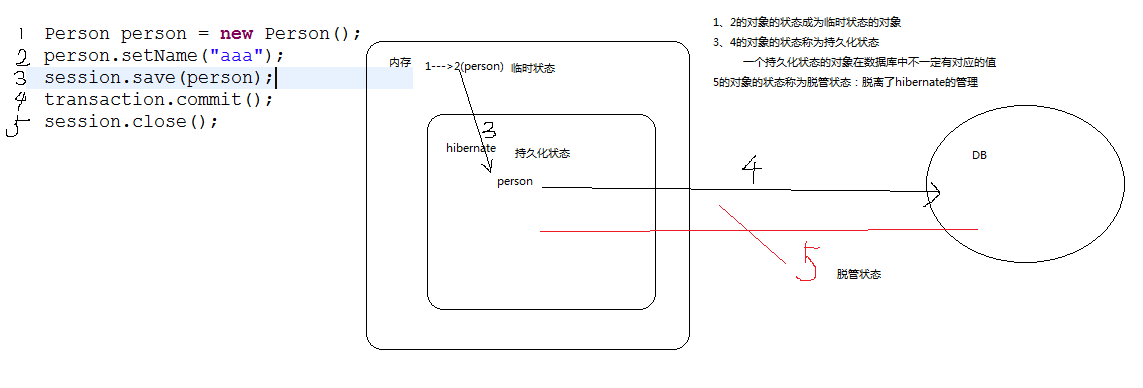
### sequence

/\*\*

\* sequence 支持oracle的数据库，oracle自己生成，效率高，是oracle自己维护的序列

\*/

## 对象的状态



第1步，第2步只是在内存中创建了对象，和hibernate没有发生关系，第3步是将person对象放到hibernate区域中，第4步是与数据库进行交互，第5步将连接关闭

第1,2步的对象的状态称为临时状态的对象

第3,4步的对象的状态称为持久化状态

一个持久化状态的对象在数据库中不一定有对应的值

第5步的对象的状态称为托管状态

临时状态的对象和托管状态的对象的区别在于托管状态的对象和数据库交互过

### save

该方法可以把一个对象从临时状态转换成持久化状态

### get

从数据库中根据主键提取出一个对象，该对象就是一个持久化状态的对象

@Test

**public** **void** testGet(){

SessionFactory sessionFactory = HibernateUtils.*sessionFactory*;

Session session = sessionFactory.openSession();

Person person = (Person)session.get(Person.**class**, 1L);//持久化状态

session.close();

}

### update

把一个对象变成持久化状态

@Test

**public** **void** testUpdate(){

SessionFactory sessionFactory = HibernateUtils.*sessionFactory*;

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Person person = (Person)session.get(Person.**class**, 1L);//持久化状态

person.setDescription("111");

//session.update(person);//update方法就是把一个对象的状态变成持久化状态，person对象已经是持久化状态了,所以这行代码不需要写

transaction.commit();

session.close();

}

### evict

把一个对象从持久化状态转化为脱管状态

/\*\*

\* evict方法使得某一个对象变成脱管状态了

\*/

@Test

**public** **void** testEvict(){

SessionFactory sessionFactory = HibernateUtils.*sessionFactory*;

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Person person = (Person)session.get(Person.**class**, 1L);//持久化状态

person.setDescription("111");

session.evict(person);//该方法可以把一个持久化状态变成脱管状态

session.update(person);//由脱管状态的对象变成持久化状态

transaction.commit();

session.close();

}

### clear

把所有的hibernate中的持久化对象都转换成脱管状态的对象

@Test

**public** **void** testClear(){

SessionFactory sessionFactory = HibernateUtils.*sessionFactory*;

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Person person = (Person)session.get(Person.**class**, 1L);//持久化状态

person.setDescription("111");

Person person2 = **new** Person();

person2.setName("88");

session.save(person2);//person2是一个持久化状态的对象

session.clear();//把hibernate里面的所有的对象都变成脱管状态的对象

transaction.commit();

session.close();

}

### flush

在执行过程中发现在执行到transact.commit的时候才真正发出sql语句，但是真正发出sql语句是在执行session.flush方法的时候，因为在transact.commit方法的内部做了判断，如果没flush就执行flush方法

如果transact.commit之前的代码有session.flush，则事务提交的时候不再flush

如果transact.commit之前的代码没有session.flush，则在transact.commit方法中会执行session.flush

flush做的事情：

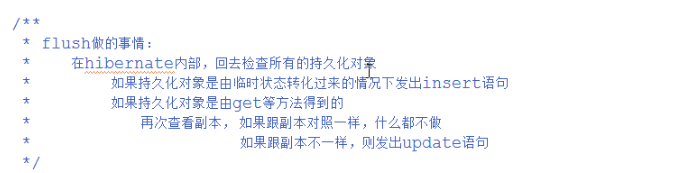
在hibernate内部，会去检查所有的持久化对象

如果持久化对象是由临时状态转化过来的，则发出insert语句

如果持久化对象是由get等方法得到的

再次查看副本，如果跟副本对照一样，什么都不做

如果跟副本不一样，则发出update语句

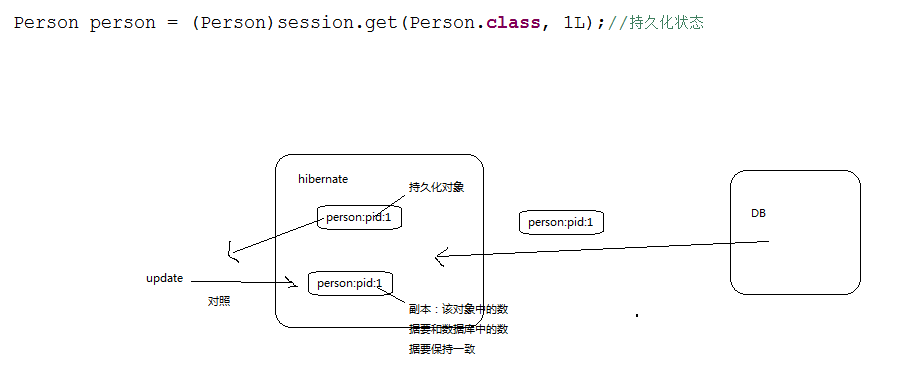






## 副本

hibernate在与数据库交互时，会对持久化对象克隆一份生成一个副本，副本的数据和数据库中的数据一致，副本的作用就是在持久化对象进行update操作时，hibernate会先与副本进行对照，如果持久化对象与副本完全一样，就不执行update操作了，不一样再执行update操作，这样可以提高效率。否则每次从数据库中读取出来的对象，即使没有修改数据最后也要执行update操作，效率太低了

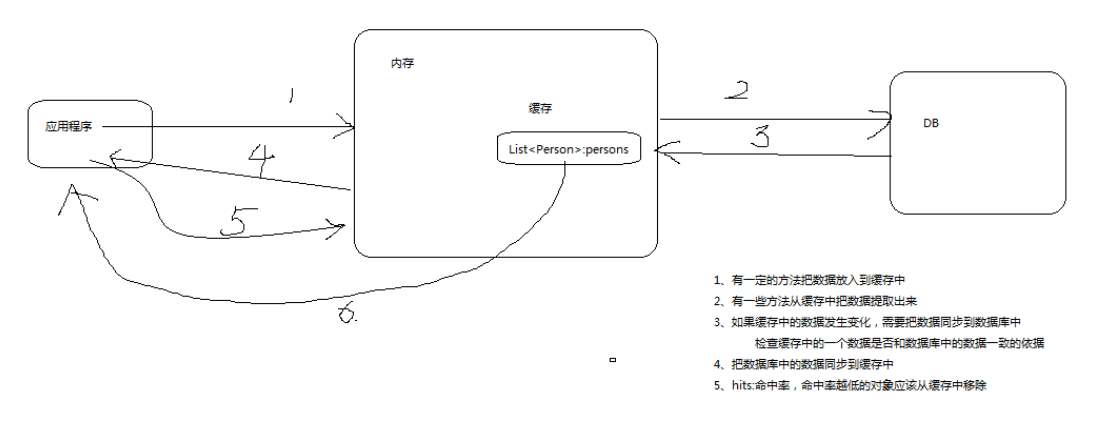


# 缓存

oscache,ehcache 小型的应用

memory cache,redis,hbase 分布式的应用

缓存简单的用一句话说就是将数据临时的放在内存区域中，这样可以直接从内存中将数据读取出来

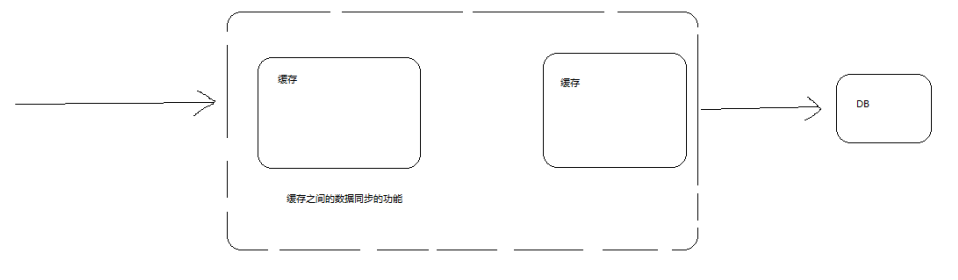


第3步在返回数据的时候，会将数据在内存中放一份，这就是缓存数据

分布式缓存：把缓存放到好几台机器上,分布式缓存相当于在几台机器外面套了一层”黑匣子”，客户端对于要访问哪台机器上的缓存是不可见的

分布式缓存的目的:分担高并发的压力

分布式缓存的应用场景：当前的服务器的并发量太大，需要增加服务器，那么这些服务器上面的缓存就是分布式缓存，这些服务器上面的缓存数据应该是一样的



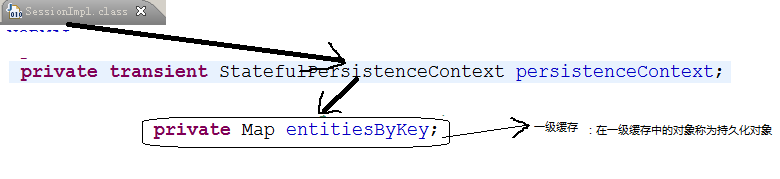
## 一级缓存

### 概念

为session级别的缓存（也就是说他是在session里边），一级缓存的生命周期和session的生命周期保持一致。（当session开启就有了，当session关闭就结束了）

### 位置

在SessionImpl类中的StatefulPersistenceContext中的map中



之前说的对象的状态从临时状态变为持久化状态，持久化状态对象是进入到hibernate中的说法不太准确，正确的说法应该是这个对象进入到了一级缓存中，也可以说一个对象如果是持久化状态的对象，那么这个对象在一级缓存中

### 操作

#### get方法

可以把对象放入到一级缓存中，也可以从一级缓存中把对象提取出来

#### save方法

该方法可以把一个对象放入到一级缓存中

#### evit方法

可以把一个对象从session的缓存中清空

#### update方法

可以把一个对象放入到一级缓存(session)中

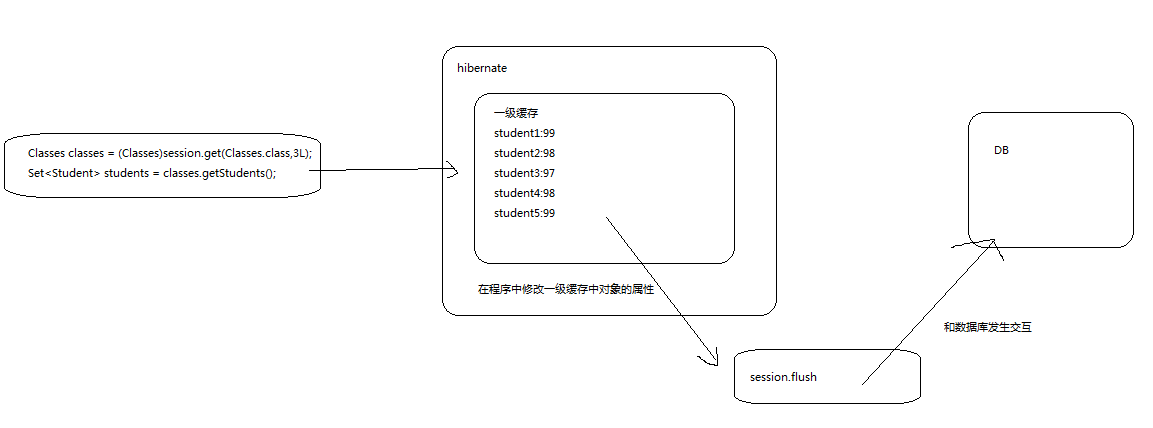
#### clear方法

清空一级缓存中所有的数据

#### close方法

当调用session.close方法的时候，一级缓存的生命周期就结束了

### 一级缓存的意义

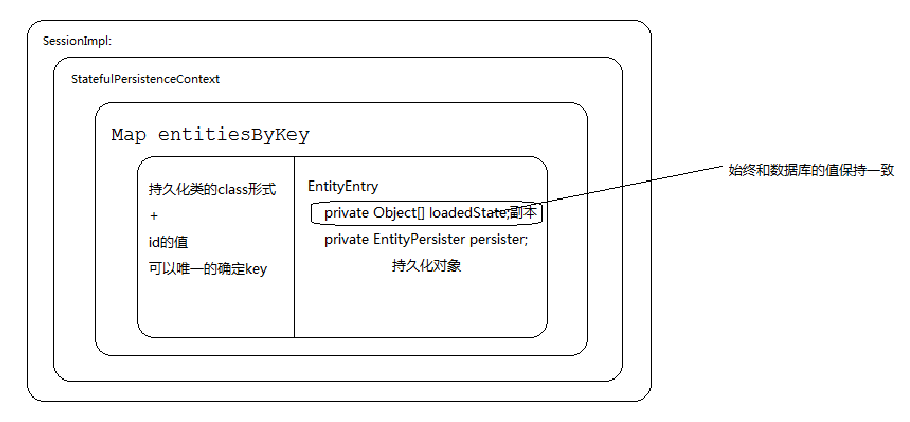


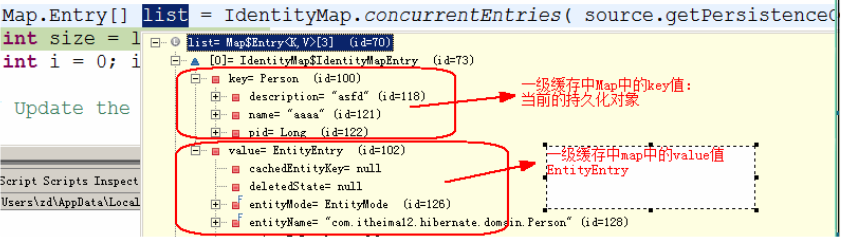
一级缓存的意义：在session的生命周期内无论修改多少数据（修改数据也只是在缓存中对该对象的属性值进行修改而并没有与数据库交互），最后在执行session.flush的时候只与数据库发生一次交互，提高了效率（否则只要修改一个对象的数据就要update一次数据库，效率太低）

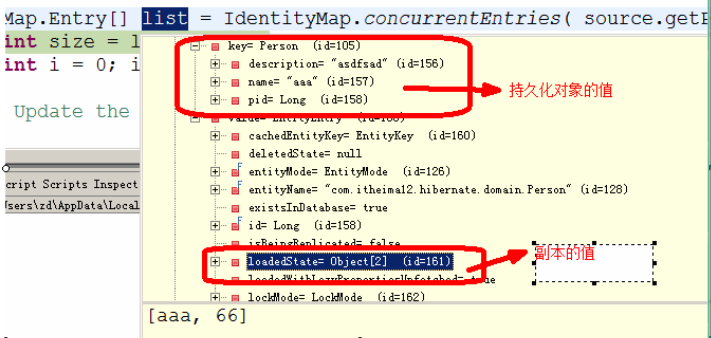
hibernate给持久化对象放了一个内存区域，由hibernate管理，当执行session.flush的时候，hibernate只检查一级缓存里面对象的状态，从而决定发出insert语句还是update语句，不在一级缓存中的对象就不管了

一级缓存的意义不是为了读取一个对象之后再次读取就不用读数据库了，因为不会连续执行get方法两次，也不是为了读取对象之后下次再次访问时可以不用读取数据库了，因为方法执行完之后session就关闭了，一级缓存的生命周期就结束了

### 一级缓存的内存结构







## 二级缓存

### 适用的场合

公开的数据

数据基本上不发生变化

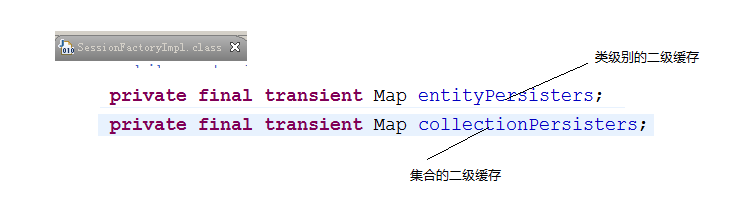
该数据保密性不是很强

说明：如果一个数据一直在改变，不适合用缓存。

### 生命周期

二级缓存的生命周期和sessionFactory是一致的。

### 位置



### 设置二级缓存

利用的是ehcache实现的二级缓存（要加入ehcache的jar包）

1、在hibernate的配置文件中配置

<!--

二级缓存的供应商

-->

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

org.hibernate.cache.EhCacheProvider

</property>

<!--

开启二级缓存

-->

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

2、指定哪个类开启二级缓存

两种方式：

1.在类的映射文件中设置：



2.在hibernate配置文件中设置：



3、在hibernate的配置文件中开启二级缓存的统计机制(可选，用于二级缓存的统计)

<!--

开启二级缓存的统计机制

-->

<property name=*"generate\_statistics"*>true</property>

### 操作

#### get方法

/\*\*

\* session.get方法不仅要把数据放入到一级缓存，而且要放入到二级缓存

\* 该方法在提取数据的时候，先从一级缓存中查找，再从二级缓存中查找，如果找不到，则查询数据库

\*/

@Test

**public** **void** testGet(){

Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

System.***out***.println(*sessionFactory*.getStatistics()

.getEntityLoadCount());

session.close();

}

#### save方法

/\*\*

\* session.save方法不操作二级缓存

\*/

@Test

**public** **void** testSave(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Classes classes = **new** Classes();

classes.setName("a");

session.save(classes);

System.***out***.println(*sessionFactory*.getStatistics()

.getEntityLoadCount()); //输出0

transaction.commit();

session.close();

}

#### update方法

/\*\*

\* session.update不操作二级缓存

\*/

@Test

**public** **void** testUpdate(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Classes classes = **new** Classes();

classes.setCid(3L);

classes.setName("afds");

session.update(classes);

System.***out***.println(*sessionFactory*.getStatistics().getEntityInsertCount());

transaction.commit();

session.close();

}

#### hql操作

list方法可以让hql语句指定的对象进入到二级缓存中，但是list方法不利用二级缓存查询数据

iterate方法的查询策略：

1、先查找该表中所有的id的值

2、再根据id值从二级缓存中查找对象，如果有，则利用二级缓存，如果没有则根据id查询该表中的所有的属性的值

/\*\*

\* 利用查询让对象进入到二级缓存中

\*/

@Test

**public** **void** testQuery() **throws** Exception{

/\*\*

\* list方法可以让hql语句指定的对象进入到二级缓存中，但是list方法不利用二级缓存查询数据

\*/

Session session = *sessionFactory*.openSession();

List<Classes> classes = session.createQuery("from Classes").list(); //hql:hibernate query language

System.***out***.println(*sessionFactory*.getStatistics().getEntityLoadCount());

session.close();

Thread.*sleep*(1000L);

/\*\*

\* iterate方法的查询策略：

\* 1、先查找该表中所有的id的值

\* 2、再根据id值从二级缓存中查找对象，如果有，则利用二级缓存，如果没有则根据id查询该表中的所有的属性的值

\*/

session = *sessionFactory*.openSession();

Iterator<Classes> iterator = session.createQuery("from Classes").iterate();

**while**(iterator.hasNext()){

Classes classes2 = iterator.next();

System.***out***.println(classes2.getName());

}

session.close();

}

### 集合的二级缓存

#### 设置

在映射文件中的集合下面添加二级缓存



#### 操作

/\*\*

\* 集合的二级缓存

\*/

@Test

**public** **void** testCollection\_SecondLevel(){

Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

Set<Student> students = classes.getStudents();

**for** (Student student : students) {

System.***out***.println(student.getName());

}

//输出的值为1,说明有一个集合进入了二级缓存中

System.***out***.println(*sessionFactory*.getStatistics().getCollectionLoadCount());

session.close();

}

### 总结

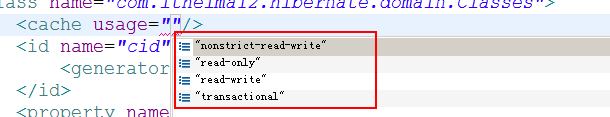
#### 哪些方法可以把对象放入到二级缓存

get方法,list方法可以把一个或者一些对象放入到二级缓存中

#### 哪些方法可以把对象从二级缓存中提取出来

get方法,iterate方法可以提取

### 策略



read-only：对象只要加载到二级缓存以后，就只能读取了，不能进行修改。

read-write：对二级缓存中的对象能够进行读和写的操作

### 缓存到磁盘

一般不使用这种方式，可以使用分布式缓存的方式

添加ehcache.xml文件：



文件中的内容：

<ehcache xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:noNamespaceSchemaLocation=*"../config/ehcache.xsd"*>

<diskStore path=*"e:\\TEMP1"*/>

<defaultCache

maxElementsInMemory=*"12"*

eternal=*"false"*

timeToIdleSeconds=*"1200"*

timeToLiveSeconds=*"1200"*

overflowToDisk=*"false"*

maxElementsOnDisk=*"10000000"*

diskPersistent=*"false"*

diskExpiryThreadIntervalSeconds=*"120"*

memoryStoreEvictionPolicy=*"LRU"*

/>

<Cache

name=*"com.itheima12.hibernate.domain.Classes"*

maxElementsInMemory=*"3"*

eternal=*"false"*

timeToIdleSeconds=*"120"*

timeToLiveSeconds=*"120"*

overflowToDisk=*"true"*

maxElementsOnDisk=*"10000000"*

diskPersistent=*"false"*

diskExpiryThreadIntervalSeconds=*"120"*

memoryStoreEvictionPolicy=*"LRU"*

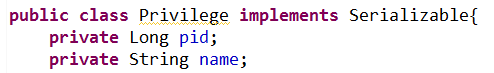
/>

</ehcache>

### 应用

使用监听器将权限的数据放入到二级缓存中,以后访问数据从二级缓存中读取

权限类：



映射文件:

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.servlet.hibernate.sessionfactory.domain.Privilege"*>

<cache usage=*"read-only"*/>

<id name=*"pid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"name"* length=*"20"*></property>

</class>

</hibernate-mapping>

配置文件：

<?xml version=*'1.0'* encoding=*'utf-8'*?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<!--

一个sessionFactory代表数据库的一个连接

-->

<session-factory>

<!-- 链接数据库的用户名 -->

<property name=*"connection.username"*>root</property>

<!-- 链接数据库的密码 -->

<property name=*"connection.password"*>root</property>

<!-- 链接数据库的驱动 -->

<property name=*"connection.driver\_class"*>

com.mysql.jdbc.Driver

</property>

<!-- 链接数据库的url -->

<property name=*"connection.url"*>

jdbc:mysql://localhost:3306/itheima12\_hibernate

</property>

<!--

方言

告诉hibernate使用什么样的数据库，hibernate就会在底层拼接什么样的sql语句

-->

<property name=*"dialect"*>

org.hibernate.dialect.MySQLDialect

</property>

<!--

二级缓存的供应商

-->

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

org.hibernate.cache.EhCacheProvider

</property>

<!--

开启二级缓存

-->

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

<!--

<class-cache usage="read-only" class=""/>

-->

<!--

根据持久化类生成表的策略

validate 通过映射文件检查持久化类与表的匹配

update 每次hibernate启动的时候，检查表是否存在，如果不存在，则创建，如果存在，则什么都不做了

create 每一次hibernate启动的时候，根据持久化类和映射文件生成表

create-drop

-->

<property name=*"hbm2ddl.auto"*>update</property>

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

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

<property name=*"format\_sql"*>true</property>

<!--

开启二级缓存的统计机制

-->

<property name=*"generate\_statistics"*>true</property>

<mapping

resource=*"com/itheima12/servlet/hibernate/sessionfactory/domain/Privilege.hbm.xml"* />

</session-factory>

</hibernate-configuration>

监听器：

**public** **class** PrivilegeListener **implements** ServletContextListener{

@Override

**public** **void** contextDestroyed(ServletContextEvent arg0) {

}

@Override

**public** **void** contextInitialized(ServletContextEvent arg0) {

System.***out***.println("++++++++++++++");

PrivilegeDao privilegeDao = **new** PrivilegeDao();

privilegeDao.putPrivilegesToSecondLevel();

}

}

Dao：

**public** **class** PrivilegeDao {

/\*\*

\* 在监听器启动的时候,把所有的权限放入到二级缓存中

\*/

**public** **void** putPrivilegesToSecondLevel(){

Session session = HibernateUtils.*sessionFactory*.openSession();

session.createQuery("from Privilege").list();//把所有的privilege对象放入到了二级缓存中

}

}

Servlet：

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

Session session = HibernateUtils.*sessionFactory*.openSession();

Iterator<Privilege> iterator = session.createQuery("from Privilege").iterate();

**while**(iterator.hasNext()){

Privilege privilege = iterator.next();

System.***out***.println(privilege.getName()+"-----------------------------------------------");

}

}

web.xml:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xmlns:web=*"http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* id=*"WebApp\_ID"* version=*"3.0"*>

<display-name>hibernate\_servlet\_sessionfactory</display-name>

<listener>

<listener-class>com.itheima12.servlet.hibernate.sessionfactory.listener.PrivilegeListener</listener-class>

</listener>

<servlet>

<servlet-name>SecondLevelServlet</servlet-name>

<servlet-class>com.itheima12.servlet.hibernate.sessionfactory.servlet.SecondLevelServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>SecondLevelServlet</servlet-name>

<url-pattern>/SecondLevelServlet</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

## 查询缓存

### 概念

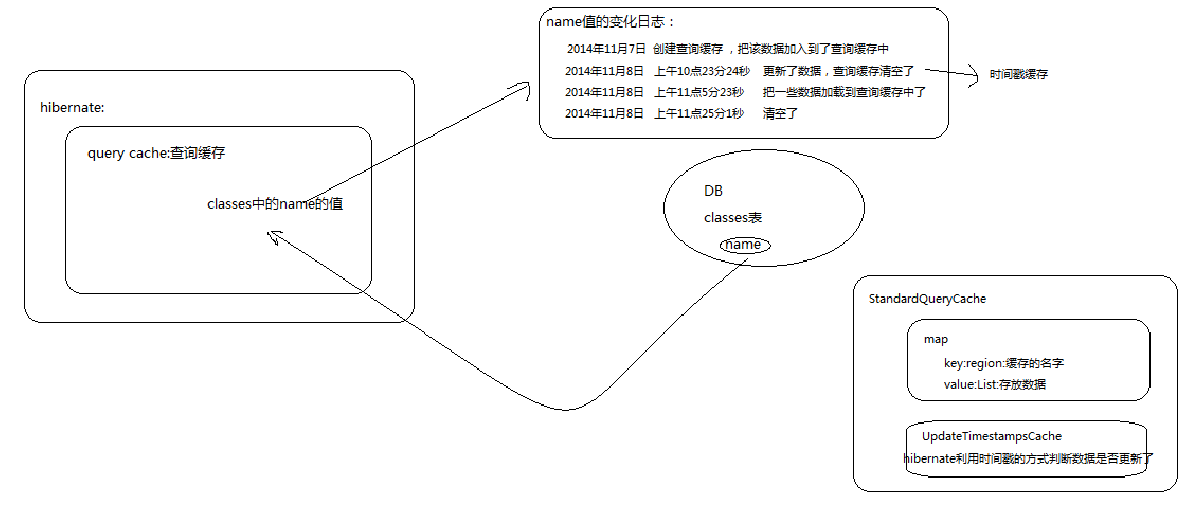
一级缓存和二级缓存都是对象缓存:就是把该对象对应的数据库表中的所有的字段全部查询出来了，这种查询在某些场合下会让效率降低。例如：表中的字段特别多，但是程序中所需要的字段却很少。

查询缓存也叫数据缓存：内存(页面)中需要多少数据就把多少数据放入到查询缓存中。

### 生命周期

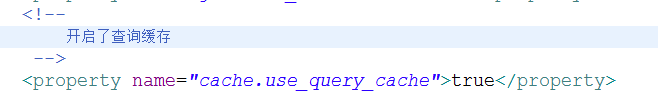
只要一些数据放入到查询缓存中，该缓存会一直存在，直到缓存中的数据被修改了，该缓存的生命周期就结束了。

### 内存结构



### 操作步骤

#### 1、在hibernate的配置文件中，开启查询缓存



#### 2、使用查询缓存

/\*\*

\* "from Classes"查询出来的数据能够放入到查询缓存中

\*/

@Test

**public** **void** testList\_1(){

Session session = *sessionFactory*.openSession();

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

query.setCacheable(**true**);//query要使用查询缓存了

query.list();//把数据放入到查询缓存中

session.close();

session = *sessionFactory*.openSession();

query = session.createQuery("from Classes");

query.setCacheable(**true**);

query.list();

session.close();

}

/\*\*

\* "select name from Classes"

\* 查询出来的数据能够放入到查询缓存中

\* 但是不能放入到二级缓存中，因为不是对象

\*/

@Test

**public** **void** testList\_2(){

Session session = *sessionFactory*.openSession();

Query query = session.createQuery("select name from Classes");

query.setCacheable(**true**);//query要使用查询缓存了

query.list();//把数据放入到查询缓存中

System.***out***.println(*sessionFactory*.getStatistics().getEntityLoadCount());

session.close();

session = *sessionFactory*.openSession();

query = session.createQuery("select name from Classes");

query.setCacheable(**true**);

query.list();

session.close();

}

/\*\*

\* 如果两个hql一样，则可以利用查询缓存，如果不一样，哪怕有一点不一样，就不能够利用了。

\*/

@Test

**public** **void** testList\_3(){

Session session = *sessionFactory*.openSession();

Query query = session.createQuery("select name from Classes");

query.setCacheable(**true**);//query要使用查询缓存了

query.list();//把数据放入到查询缓存中

session.close();

session = *sessionFactory*.openSession();

query = session.createQuery("select name from Classes where cid=1");

query.setCacheable(**true**);

query.list();

session.close();

}

/\*\*

\* 先把一些数据放入到查询缓存中，修改一些数据，看生命周期

\*/

@Test

**public** **void** testList\_4(){

/\*\*

\* 把name放入到了查询缓存中

\*/

Session session = *sessionFactory*.openSession();

Query query = session.createQuery("select name from Classes");

query.setCacheable(**true**);//query要使用查询缓存了

query.list();//把数据放入到查询缓存中

session.close();

/\*\*

\* 修改name属性的值

\* 修改了查询缓存的时间戳缓存，从而知道了该数据已经被修改了,查询缓存中的数据就被清空了

\*/

session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

classes.setName("adsfasfd");

transaction.commit();

session.close();

/\*\*

\* 再次查询name属性的值,查询缓存中name属性的值清空了

\*/

session = *sessionFactory*.openSession();

query = session.createQuery("select name from Classes");

query.setCacheable(**true**);//query要使用查询缓存了

query.list();//把数据放入到查询缓存中

session.close();

}

## 总结

hibernate总共有三种缓存：

一级缓存解决的问题是在一次请求中，尽量减少和数据库交互的次数，在session.flush之前，改变的是一级缓存的对象的属性。当session.flush的时候才要跟数据库交互，一级缓存解决不了重复查询的问题。一级缓存是对象缓存

二级缓存可以把经常不改变、常用的公共的数据放入进来，可以重复查询，利用get方法和iterator方法可以把二级缓存中的数据得到。

查询缓存可以缓存数据或者对象，可以利用list方法把查询缓存中的数据放入到缓存中。查询缓存中存放的是数据，是数据缓存。

# session

## 创建session的方式

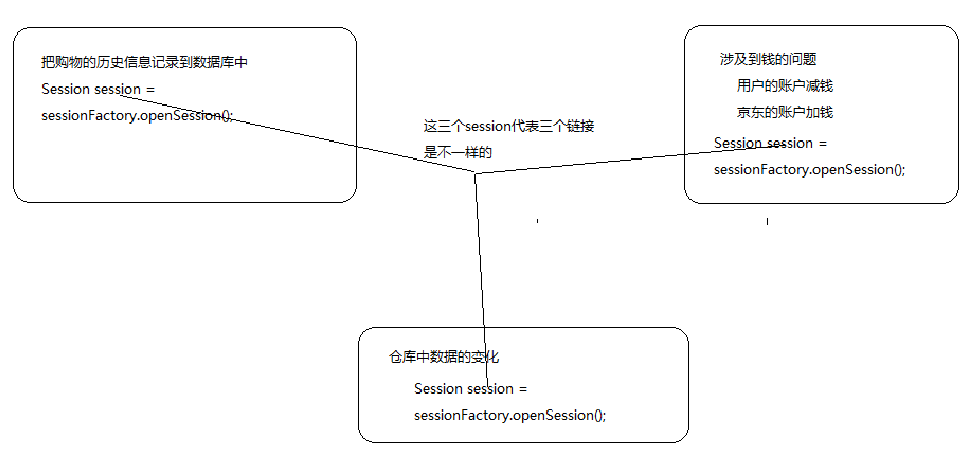
### sessionFactory.openSession



每次当客户端调用sessionFactory.openSession方法的时候，都会创建一个新的Session, 相当于一个新的连接

每次都要创建一个新的session,相当于一个新的连接

#### sessionFactory.openSession的弊端

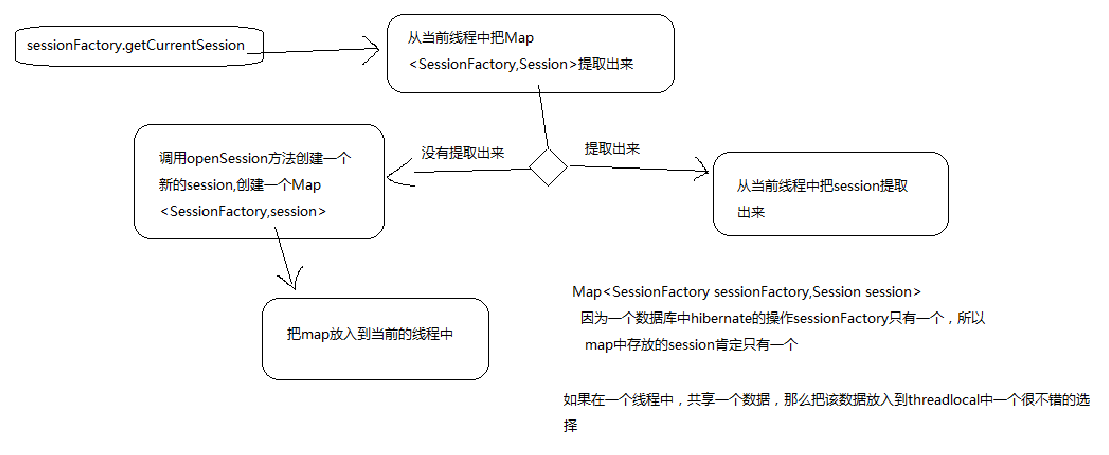


比如京东这种的购物网站会有自己很多的系统，那么在购物支付的时候，会在这些系统之间进行数据的操作，这些操作就需要创建session，并且这些操作需要在同一个事务之中，那么使用sessionFactory.openSession的方式就无法实现了（因为sessionFactory.openSession每次都会创建一个新的session，是不同的session，无法让他们处于同一个事务之中）

### sessionFactory.getCurrentSession

#### 原理

相当于把session放到了当前线程中，放到了ThreadLocal中（这样可以解决同一个事务中处理多个系统创建session的问题），ThreadLocal中放了一个Map，key是SessionFactory,value是Session,这么做是为了保证一个数据库中只有一个session



#### 用法

@Test

**public** **void** testSession\_getCurrentSession(){

Session session = HibernateUtils.*sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Person person = (Person)session.get(Person.**class**, 1L);

transaction.commit();//当事务提交的时候，session自动关闭

}

##### 在hibernate.cfg.xml文件中加入下面的配置



##### 如果用该方法产生session,则crud操作必须在事务的环境下运行

##### 当执行transaction.commit的方法的时候，session自动关闭（不用执行session.close方法了）

# 关系操作

## 一对多的单向

在Classes中能访问到Student，在Student中访问不到Classes，这就是一对多的单向关联

在Classes中使用集合时，Set比List好，因为Set可以去重

在一的一方维护关系的时候，总会发出维护关系的update语句，该update语句就是更新外键,所以一的一方维护关系效率并不高

### 配置、建表

Classes:

**public** **class** Classes **implements** Serializable{

**private** Long cid; //标示符属性

**private** String name; //一般属性

**private** String description;

**private** Set<Student> students; //关联对象

映射文件：

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Classes"*>

<id name=*"cid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

set元素针对的就是Classes类中的Set属性

-->

<set name=*"students"*>

<!--

外键

告诉hibernate,通过cid就可以建立classes与student之间的关联（告诉hibernate，外键关联关系）

-->

<key>

<column name=*"cid"*></column>

</key>

<!--

告诉hibernate，Classes类中的set集合中存放的是哪个元素

-->

<one-to-many class=*"com.itheima12.hibernate.domain.Student"*/>

</set>

</class>

</hibernate-mapping>

Student:

**public** **class** Student **implements** Serializable{

**private** Long sid;

**private** String name;

**private** String description;

映射文件：

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Student"*>

<id name=*"sid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

</class>

</hibernate-mapping>

### 一般操作（保存的数据没有关联关系）

#### 1、保存班级

@Test

**public** **void** testSaveClasses(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Classes classes = **new** Classes();

classes.setName("黑马JavaEE+hadoop的12期");

classes.setDescription("牛");

session.save(classes);

transaction.commit();

}

#### 2、保存学生

@Test

**public** **void** testSaveStudent(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Student student = **new** Student();

student.setName("班长");

student.setDescription("带头大哥");

session.save(student);

transaction.commit();

}

#### 3、保存班级，保存学生

@Test

**public** **void** testSaveClassesAndSaveStudent(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Classes classes = **new** Classes();

classes.setName("黑马视频班");

classes.setDescription("野牛");

Student student = **new** Student();

student.setName("班秘");

student.setDescription("凤姐");

session.save(classes);

session.save(student);

transaction.commit();

}

### 级联操作

save-update:在保存或者更新Classes的时候，根据Student对象的状态进行级联保存或者更新操作

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Classes"*>

<id name=*"cid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

set元素针对的就是Classes类中的Set属性

cascade 级联操作

null 默认值

save-update

在保存classes对象的时候，针对student进行保存或者更新的操作

在更新classes对象的时候，针对student进行保存或者更新的操作

all

delete

-->

<set name=*"students"* cascade=*"save-update"*>

<!--

外键

告诉hibernate,通过cid就可以建立classes与student之间的关联

-->

<key>

<column name=*"cid"*></column>

</key>

<!--

告诉hibernate，Classes类中的set集合中存放的是哪个元素

-->

<one-to-many class=*"com.itheima12.hibernate.domain.Student"*/>

</set>

</class>

</hibernate-mapping>

#### 4、保存班级级联保存学生

/\*\*

\* 在保存班级的时候，级联保存学生

\*/

@Test

**public** **void** testSaveClasses\_Cascade\_Save\_Student(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Classes classes = **new** Classes();

classes.setName("黑马视频班");

classes.setDescription("野牛");

Student student = **new** Student();

student.setName("班秘");

student.setDescription("凤姐");

//建立classes与student之间的关联

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

students.add(student);

classes.setStudents(students);

session.save(classes); //显示保存，把保存student称为隐式保存

transaction.commit();

}

#### 7、更新班级级联更新学生

/\*\*

\* 在更新班级的时候，级联更新学生

\* sessin.flush的时候

\* 1、检查一级缓存中所有的持久化状态的对象

\* 判断发出insert语句或者update语句

\* 2、检查所有的持久化对象的关联对象

\* 如果关联对象是由临时状态转化过来的，则对关联对象发出insert语句

\* 如果关联对象是从数据库中提取出来的，则对照副本，决定是否发出update语句

\*/

@Test

**public** **void** testUpdateClasses\_Cascade\_Update\_Student(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Classes classes = (Classes)session.get(Classes.**class**, 2L);//持久化类

Set<Student> students = classes.getStudents();//持久化类

**for** (Student student : students) {//把每一个student对象(关联对象)也放入到了一级缓存中

student.setDescription("bb");

}

transaction.commit();

}

#### 6、更新班级级联保存学生

/\*\*

\* 在更新班级的时候，添加学生

\*/

@Test

**public** **void** testUpdateClasses\_Cascade\_Save\_Student(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

/\*

\* 给cid为2的班级添加一个学生

\*/

Classes classes = (Classes)session.get(Classes.**class**, 2L);

Student student = **new** Student();

student.setName("美女1");

student.setDescription("小林志玲");

/\*\*

\* 建立班级与学生的关联

\*/

classes.getStudents().add(student);

transaction.commit();

}

#### 8、删除班级级联删除学生

/\*\*

\* 在 Classes.hbm.xml文件中

\* <set name="students" cascade="all">

\* 在删除班级的时候，级联删除学生

\*/

@Test

**public** **void** testDeleteClasses\_2(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

session.delete(classes);

transaction.commit();

}

### 关系操作

在hibernate中，维护关系的操作是由inverse配置的，inverse设置了default或false表示维护关系，就会发出更新关系的update语句

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Classes"*>

<id name=*"cid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

set元素针对的就是Classes类中的Set属性

cascade 级联操作

null 默认值

save-update

在保存classes对象的时候，针对student进行保存或者更新的操作

在更新classes对象的时候，针对student进行保存或者更新的操作

all

delete

inverse 关系操作

default:classes维护classes与student之间的关系

true: classes不维护classes与student之间的关系

false: classes维护classes与student之间的关系

-->

<set name=*"students"* cascade=*"save-update"* inverse=*"true"*>

<!--

外键

告诉hibernate,通过cid就可以建立classes与student之间的关联

-->

<key>

<column name=*"cid"*></column>

</key>

<!--

告诉hibernate，Classes类中的set集合中存放的是哪个元素

-->

<one-to-many class=*"com.itheima12.hibernate.domain.Student"*/>

</set>

</class>

</hibernate-mapping>

#### 8、已经存在一个班级，新建一个学生，把该学生加入到该班级(建立关系操作)

/\*\*

\* 在更新班级的时候，级联保存学生，并且维护关系

\* Hibernate:

select

classes0\_.cid as cid0\_0\_,

classes0\_.description as descript2\_0\_0\_,

classes0\_.name as name0\_0\_

from

Classes classes0\_

where

classes0\_.cid=?

Hibernate:

select

students0\_.cid as cid0\_1\_,

students0\_.sid as sid1\_,

students0\_.sid as sid1\_0\_,

students0\_.description as descript2\_1\_0\_,

students0\_.name as name1\_0\_

from

Student students0\_

where

students0\_.cid=?

Hibernate:

select

max(sid)

from

Student

Hibernate:

因为在Classes.hbm.xml文件中设置了级联

<set name="students" cascade="save-update">

insert

into

Student

(description, name, sid)

values

(?, ?, ?)

Hibernate:

因为在Classes.hbm.xml文件中，inverse没有写，默认classes维护classes与student之间的关系

所以发出了更新关系的update语句

update

Student

set

cid=?

where

sid=?

\*/

@Test

**public** **void** testUpdateClasses\_Cascade\_Save\_Student\_Inverse(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

/\*

\* 给cid为2的班级添加一个学生

\*/

Classes classes = (Classes)session.get(Classes.**class**, 2L);

Student student = **new** Student();

student.setName("美女1");

student.setDescription("小林志玲");

/\*\*

\* 建立班级与学生的关联

\*/

classes.getStudents().add(student);

transaction.commit();

}

#### 11、已经存在一个学生，把一个学生从一个班级转移到另外一个班级

/\*\*

\* 已经存在一个班级cid为1，已经存在一个学生，已经存在另外一个班级cid为2，该学生从cid为1的班级转到cid为2的班级

\*/

@Test

**public** **void** testTransform(){

/\*\*

\* 1、把cid为1，2和sid为1的对象提取出来

\*/

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

//Classes classes1 = (Classes)session.get(Classes.class, 1L);

Classes classes2 = (Classes)session.get(Classes.**class**, 2L);//发出查询的sql语句

Student student = (Student)session.get(Student.**class**, 1L);//发出查询的sql语句

//解除 classes1与student之间的关系

//classes1.getStudents().remove(student);

//建立classes2与student之间的关系

classes2.getStudents().add(student);

transaction.commit();

}

#### 解除班级和该班级中的所有的学生之间的关系

/\*\*

\* 解除该班级和该班级中的所有的学生之间的关系

\*/

@Test

**public** **void** testRealseR(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

/\*\*

\* 解除cid为2的班级和所有的学生之间的关系

\*/

Classes classes = (Classes)session.get(Classes.**class**,2L);

classes.setStudents(**null**);

transaction.commit();

}

#### 解除该班级和所有的学生之间的关系，再建立该班级和一些学生之间的关系

/\*\*

\* 解除该班级和所有的学生之间的关系，再建立该班级和一些学生之间的关系

\*/

@Test

**public** **void** testRealseAllR\_BuildSomeR(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

//解除cid为2的班级和所有的学生之间的关系

Classes classes = (Classes)session.get(Classes.**class**, 2L);

classes.setStudents(**null**);

Student student1 = **new** Student();

student1.setName("王二麻子");

Student student2 = **new** Student();

student2.setName("隔壁老李");

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

students.add(student2);

students.add(student1);

classes.setStudents(students);

transaction.commit();

}

#### 12、在删除班级的时候，解除班级和学生之间的关系

/\*\*

\* 删除一个班级

\* 在删除班级之前，解除班级和学生之间的关系

\* Hibernate:

select

classes0\_.cid as cid0\_0\_,

classes0\_.description as descript2\_0\_0\_,

classes0\_.name as name0\_0\_

from

Classes classes0\_

where

classes0\_.cid=?

Hibernate:

因为classes负责维护关系，所以该语句就是解除关系的sql语句

update

Student

set

cid=null

where

cid=?

Hibernate:

delete

from

Classes

where

cid=?

\*/

@Test

**public** **void** testDeleteClasses\_1(){

Session session = *sessionFactory*.getCurrentSession();

Transaction transaction = session.beginTransaction();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

session.delete(classes);

transaction.commit();

}

### inverse与cascade的关系

cascade指的是级联操作，操作的是一般属性，指的是对象与对象的操作（save或update Classes对象的时候级联save或update Student对象）

inverse指的是关系操作，针对的就是外键

## 一对多的双向

### 总结

在一对多中，当多的一方维护关系时，不会发出更新关系的update语句，而一的一方维护关系时需要发出维护关系的update语句，

所以在这里，一般情况下，多的一方维护关系效率比较高。（但也有例外，比如解除一个班级和所有学生之间的关系，就在Classes中设置学生集合为null的效率比较高）

在多的一方维护关系时，将操作多的一方的对象的除了主键以外的所有属性，没有更新外键这么一说了，因为只要是在多的一方维护关系，那么多的一方的对象中就肯定有值了，所以直接操作就行

### 配置、建表

Classes:

**public** **class** Classes **implements** Serializable{

**private** Long cid; //标示符属性

**private** String name; //一般属性

**private** String description;

**private** Set<Student> students; //关联对象

映射文件：

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Classes"*>

<id name=*"cid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

set元素针对的就是Classes类中的Set属性

cascade 级联操作

null 默认值

save-update

在保存classes对象的时候，针对student进行保存或者更新的操作

在更新classes对象的时候，针对student进行保存或者更新的操作

all

delete

inverse 关系操作

default:classes维护classes与student之间的关系

true: classes不维护classes与student之间的关系

false: classes维护classes与student之间的关系

-->

<set name=*"students"* cascade=*"save-update"* inverse=*"true"*>

<!--

外键

告诉hibernate,通过cid就可以建立classes与student之间的关联

-->

<key>

<column name=*"cid"*></column>

</key>

<!--

告诉hibernate，Classes类中的set集合中存放的是哪个元素

-->

<one-to-many class=*"com.itheima12.hibernate.domain.Student"*/>

</set>

</class>

</hibernate-mapping>

Student:

**public** **class** Student **implements** Serializable{

**private** Long sid;

**private** String name;

**private** String description;

**private** Classes classes;//关联对象

映射文件：

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Student"*>

<id name=*"sid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

外键

column

-->

<many-to-one name=*"classes"* column=*"cid"* class=*"com.itheima12.hibernate.domain.Classes"*></many-to-one>

</class>

</hibernate-mapping>

### 级联操作

#### 4、保存学生级联保存班级

/\*\*

\* Hibernate:

select

max(sid)

from

Student

Hibernate:

select

max(cid)

from

Classes

Hibernate:

insert

into

Classes

(description, name, cid)

values

(?, ?, ?)

Hibernate:

外键直接插入进去了

insert

into

Student

(description, name, cid, sid)

values

(?, ?, ?, ?)

\*/

@Test

**public** **void** testSaveStudent\_Cascade\_Save\_Classes(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = **new** Student();

student.setName("asfd");

Classes classes = **new** Classes();

classes.setName("12期");

//创建班级与学生之间的关联

student.setClasses(classes); //通过学生建立的关联

session.save(student);

transaction.commit();

session.close();

}

### 关系操作

#### 11、已经存在一个学生，把一个学生从一个班级转移到另外一个班级

/\*\*

\* update

Student

set

description=?,

name=?,

cid=?

where

sid=?

\*/

@Test

**public** **void** testTransform(){

/\*\*

\* 把sid为11的学生从cid为2的班级转移到cid为1的班级

\*/

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = (Student)session.get(Student.**class**, 11L);

Classes classes = (Classes)session.get(Classes.**class**, 1L);

//建立关系

student.setClasses(classes);

transaction.commit();

session.close();

}

#### 把一个学生从一个班级中移除

/\*\*

\* 把一个学生从一个班级中移除

\* update

Student

set

description=?,

name=?,

cid=?

where

sid=?

\*/

@Test

**public** **void** testRemoveFromClasses(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = (Student)session.get(Student.**class**, 11L);

student.setClasses(**null**);

transaction.commit();

session.close();

}

## 多对多

多对多内连接查询的sql语句：

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\ScreenClip.png

### 维护关系

多对多建立关系相当于在第三张表中插入一行数据

多对多解除关系相当于在第三张表中删除一行数据

多对多修改关系相当于先删除后增加

结论: 多对多谁维护效率都一样。看需求

### 配置、建表

Course:

**public** **class** Course **implements** Serializable{

**private** Long cid;

**private** String name;

**private** String description;

**private** Set<Student> students;

映射文件：

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Course"*>

<id name=*"cid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

table

第三张表

-->

<set name=*"students"* table=*"student\_course"*>

<key>

<!--

外键

-->

<column name=*"cid"*></column>

</key>

<!--

column外键

-->

<many-to-many class=*"com.itheima12.hibernate.domain.Student"* column=*"sid"*></many-to-many>

</set>

</class>

</hibernate-mapping>

Student:

**public** **class** Student **implements** Serializable{

**private** Long sid;

**private** String name;

**private** String description;

**private** Set<Course> courses;

映射文件：

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Student"*>

<id name=*"sid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

table

第三张表

-->

<set name=*"courses"* table=*"student\_course"* cascade=*"save-update"*>

<key>

<!--

外键

-->

<column name=*"sid"*></column>

</key>

<!--

column外键

-->

<many-to-many class=*"com.itheima12.hibernate.domain.Course"* column=*"cid"*></many-to-many>

</set>

</class>

</hibernate-mapping>

### 关系操作

#### 保存学生的时候，同时保存课程

/\*\*

\* 保存学生的时候，同时保存课程

\* Hibernate:

select

max(sid)

from

Student

Hibernate:

select

max(cid)

from

Course

Hibernate:

insert

into

Student

(description, name, sid)

values

(?, ?, ?)

Hibernate:

在Student.hbm.xml文件中

<set cascade="save-update">

insert

into

Course

(description, name, cid)

values

(?, ?, ?)

Hibernate:

建立关系的sql语句

insert

into

student\_course

(sid, cid)

values

(?, ?)

\*/

@Test

**public** **void** testSaveStudent\_Cascade\_Save\_Course(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = **new** Student();

student.setName("aa");

Course course = **new** Course();

course.setName("java基础课程");

//建立学生与课程之间的关系

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

courses.add(course);

student.setCourses(courses);

session.save(student);

transaction.commit();

session.close();

}

#### 已经存在一个课程，已经存在一个学生，建立该课程和该学生之间的关系

/\*\*

\* 已经存在一个课程，已经存在一个学生，建立该课程和该学生之间的关系

\*/

@Test

**public** **void** testBuildR(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

//把cid为2的课程和sid为1的学生提取出来

Course course = (Course)session.get(Course.**class**, 2L);

Student student = (Student)session.get(Student.**class**, 1L);

student.getCourses().add(course);

transaction.commit();

session.close();

}

#### 已经存在一个课程，新建一个学生，建立该课程和该学生之间的关系

/\*\*

\* 已经存在一个课程，新建一个学生，建立该课程和该学生之间的关系

\*/

@Test

**public** **void** testSaveStudent\_BuildR(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = **new** Student();

student.setName("student1");

Course course = (Course)session.get(Course.**class**, 1L);

/\*\*

\* 建立了新的学生与该课程之间的关系

\*/

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

courses.add(course);

student.setCourses(courses);

session.save(student);

transaction.commit();

session.close();

}

#### 一个学生从一门课程转到另外一门课程

/\*\*

\* 一个学生从一门课程转到另外一门课程

\*/

@Test

**public** **void** testTranform(){

/\*\*

\* sid为1的学生从课程2转到课程1

\*/

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

//提取sid为1的学生

Student student = (Student)session.get(Student.**class**, 1L);

//把课程1和课程2提取出来

Course course1 = (Course)session.get(Course.**class**, 1L);

Course course2 = (Course)session.get(Course.**class**, 2L);

student.getCourses().remove(course2);//解除sid为1的学生和cid为2的课程

student.getCourses().add(course1);//建立sid为1的学生和cid为1的课程

transaction.commit();

session.close();

}

#### 解除一个学生和该学生所学的所有的课程之间的关系

/\*\*

\* 解除一个学生和该学生所学的所有的课程之间的关系

\*/

@Test

**public** **void** testRealseAllR(){

/\*\*

\* 解除sid为1的学生和所有的课程之间的关系

\*/

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = (Student)session.get(Student.**class**, 1L);

student.setCourses(**null**);

transaction.commit();

session.close();

}

#### 解除一个学生和所有的课程之间的关系，再建立一些课程之间的关系

/\*\*

\* 解除一个学生和所有的课程之间的关系，再建立一些课程之间的关系

\*/

@Test

**public** **void** testRealseAllRAndBuildR(){

Session session = *sessionFactory*.openSession();

Transaction transaction = session.beginTransaction();

Student student = (Student)session.get(Student.**class**, 1L);

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

Course course = (Course)session.get(Course.**class**, 3L);

courses.add(course);

//把原来的student中的courses给覆盖掉了

student.setCourses(courses);

transaction.commit();

session.close();

}

## 一对一

一对一就是在一对多的基础上，给外键加一个唯一约束，其他都不变

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.domain.Student"*>

<id name=*"sid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

外键

column

unique外键只能出现一次

-->

<many-to-one name=*"classes"* column=*"cid"* class=*"com.itheima12.hibernate.domain.Classes"*

unique=*"true"*

cascade=*"save-update"*></many-to-one>

</class>

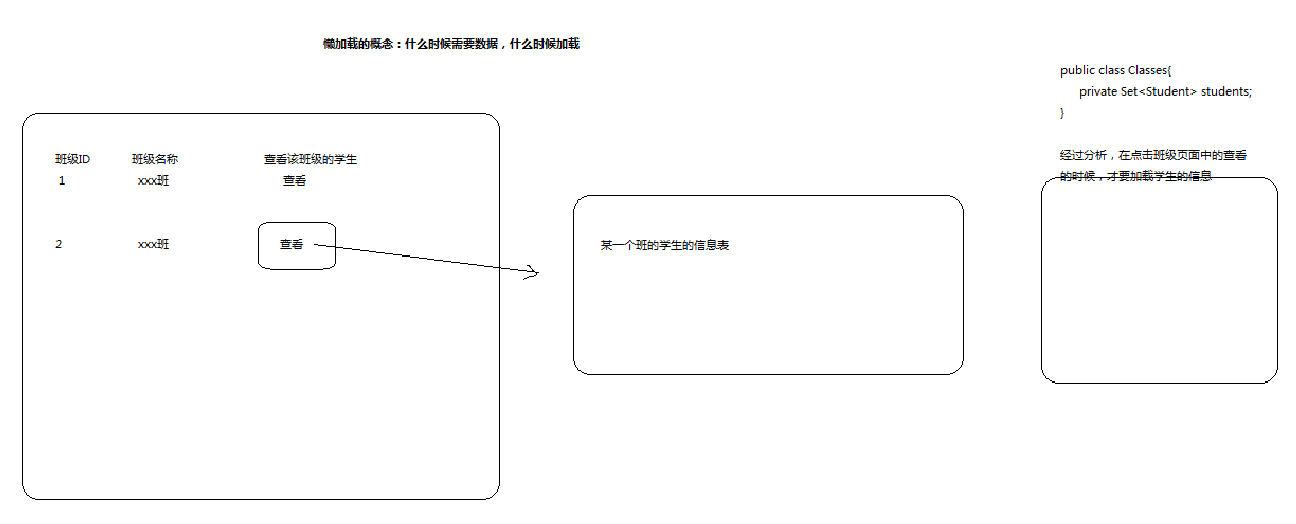
</hibernate-mapping>

# 性能

## lazy（懒加载）

需要数据的时候才要加载

### 概念



### 分类

#### 类的懒加载

/\*\*

\* 类的延迟加载

\*/

@Test

**public** **void** testClass\_Lazy(){

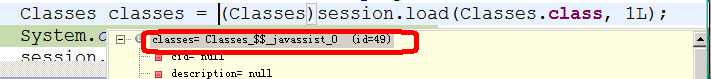
Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.load(Classes.**class**, 1L);

System.***out***.println(classes.getName());//发出sql语句

session.close();

}



session.load方法当加载类的属性时才发出sql语句

session.load方法产生的是代理对象，该代理类是持久化类的子类

#### 集合的懒加载

set的延迟加载：

true：延迟加载

false：不延迟加载

extra：进一步的延迟加载

**<set name="students" cascade="save-update" inverse="true" lazy=" true">**

/\*\*

\* 集合的延迟加载

\* <set lazy="true">

\*/

@Test

**public** **void** testCollection\_Lazy\_true(){

Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

Set<Student> students = classes.getStudents();//没有发出sql语句

**for** (Student student : students) {//发出加载学生的sql语句

System.*out*.println(student.getName());

}

session.close();

}

/\*\*

\* 集合的延迟加载

\* <set lazy="false">

\*/

@Test

**public** **void** testCollection\_Lazy\_false(){

Session session = *sessionFactory*.openSession();

/\*\*

\* 加载了classes又加载了student

\*/

Classes classes = (Classes)session.get(Classes.**class**, 1L);

Set<Student> students = classes.getStudents();

**for** (Student student : students) {

System.*out*.println(student.getName());

}

session.close();

}

/\*\*

\* 集合的延迟加载

\* <set lazy="extra">

\*

\* select

count(sid)

from

Student

where

cid =?

2

\*/

@Test

**public** **void** testCollection\_Lazy\_extra(){

Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

Set<Student> students = classes.getStudents();

System.*out*.println(students.size());

session.close();

}

说明：

1、因为延迟加载在映射文件设置，而映射文件一旦确定，不能修改了。

2、延迟加载是通过控制sql语句的发出时间来提高效率的。

#### manytoone的懒加载

对于性能来说，没有什么影响，所以随便怎么样都行

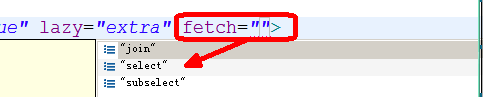
## 抓取策略

### 目的

默认的策略是：先查询一的一方的所有的对象(Classes)，再根据每一个对象的id值查询关联对象(Student)，这样会发出n+1条sql语句，效率很低，使用iterate方法同样会有这样的问题

抓取策略解决的是如何，使用什么样的sql语句来加载set集合中的数据的问题

### 取值



join:左外连接

select:默认的值

subselect:子查询

#### subselect:子查询

/\*\*

\* 加载所有的班级的所有的学生

\* n+1条sql语句

\* n 表示 classes表中的数据条数

\* 1 表示classes本身的表

\*

\* 因为根据需求分析，判断出来该需求分析中含有子查询，所以抓取策略应该用"subselect"

\*/

@Test

**public** **void** testQueryAllStudent\_SubSelect(){

Session session = *sessionFactory*.openSession();

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

**for** (Classes classes2 : classes) {

System.*out*.println(classes2.getName());

Set<Student> students = classes2.getStudents();

**for** (Student student : students) {

System.*out*.println(student.getName());

}

}

session.close();

}

#### select:默认的值

先查询一的一方的所有的对象(Classes)，再根据每一个对象的id值查询关联对象(Student)

/\*\*

\* select

\* 先查询一的一方的所有的对象(Classes)，再根据每一个对象的id值查询关联对象(Student)

\*/

@Test

**public** **void** testQueryClassesByCidAndStudents\_Select(){

Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

System.*out*.println(classes.getName());

Set<Student> students = classes.getStudents();

**for** (Student student : students) {

System.*out*.println(student.getName());

}

session.close();

}

#### join:左外连接

利用左外连接一条SQL语句把classes和student表全部查询出来了

在含有子查询的需求分析中，利用join的抓取策略是不取的

/\*\*

\* join 利用左外连接一条SQL语句把classes和student表全部查询出来了

\*

\* 在含有子查询的需求分析中，利用join的抓取策略是不取的

\*/

@Test

**public** **void** testQueryClassesByCidAndStudents\_Join(){

Session session = *sessionFactory*.openSession();

Classes classes = (Classes)session.get(Classes.**class**, 1L);

System.*out*.println(classes.getName());

Set<Student> students = classes.getStudents();

**for** (Student student : students) {

System.*out*.println(student.getName());

}

session.close();

}

### 说明

1、因为抓取策略的设置在映射文件中，所以一旦映射文件生成就不能改变了。

2、通过发出怎么样的SQL语句加载集合，从而优化效率的。

# 查询

## hql语句

### 单表

#### 查询全部数据

**public** **void** testQueryAllClasses(){

Session session = *sessionFactory*.openSession();

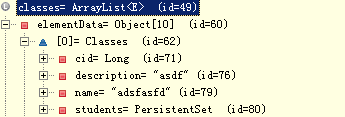
List<Classes> classes = session.createQuery("from Classes").list();

session.close();

}



内存结构：List中是持久化对象



#### 查询部分字段（投影查询）

**public** **void** testQueryProperties(){

Session session = *sessionFactory*.openSession();

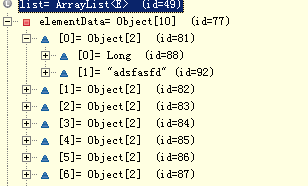
List list = session.createQuery("select cid,name from Classes").list();

session.close();

}



内存结构：List中是object类型的数组



#### 带构造函数的投影查询

**public** **void** testQueryProperties\_Constructor(){

Session session = *sessionFactory*.openSession();

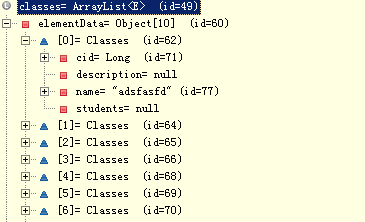
List<Classes> classes = session.createQuery("select new com.itheima12.hibernate.domain.Classes(cid,name) from Classes").list();

session.close();

}



内存结构：List中是持久化对象



#### 单一结果查询

**public** **void** testQueryCount(){

Session session = *sessionFactory*.openSession();

Long count = (Long)session.createQuery("select count(cid) from Classes")

.uniqueResult();

System.*out*.println(count);

session.close();

}



结果为Long类型的数据，该表的总的行数

### 预编译查询

#### 第一种

**public** **void** testQueryPrepare\_1(){

Session session = *sessionFactory*.openSession();

Query query = session.createQuery("from Classes where cid=? and name=?");

query.setParameter(0, 1L);

query.setParameter(1, "adsfasfd");

Classes classes = (Classes)query.uniqueResult();

System.*out*.println(classes.getName());

session.close();

}

#### 第二种

**public** **void** testQueryPrepare\_2(){

Session session = *sessionFactory*.openSession();

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

query.setParameter("cid", 1L);

query.setParameter("name", "adsfasfd");

Classes classes = (Classes)query.uniqueResult();

System.*out*.println(classes.getName());

session.close();

}

### 分页

**public** **void** testDispage(){

Session session = *sessionFactory*.openSession();

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

query.setFirstResult(3);//表示在集合中索引的位置

query.setMaxResults(3);//表示设置一页显示多少个

List<Classes> classes = query.list();

**for** (Classes classes2 : classes) {

System.***out***.println(classes2.getCid());

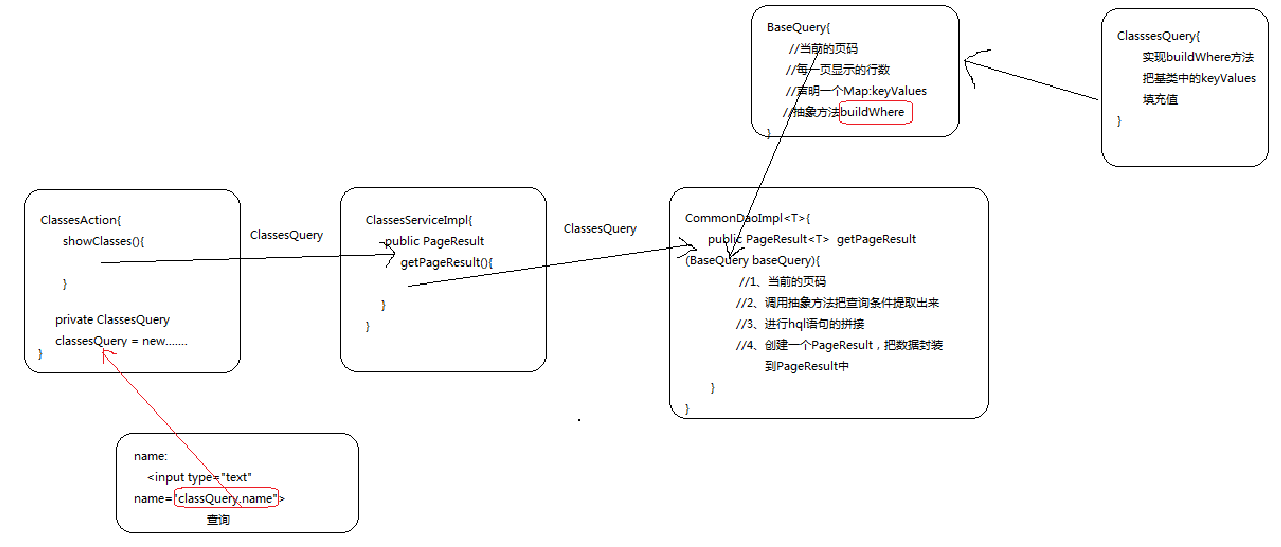
}

session.close();

}

### 公用的查询的封装

#### 图示



#### 代码

CommonDao接口：

**public** **interface** CommonDao<T>{

**public** List<T> getAllEntry();

**public** PageResult<T> getPageResult(BaseQuery baseQuery);

**public** List<T> getEntriesByCondition(Map<String, Object> keyValues);

**public** Long getCount();

}

CommonDao接口实现类:

**public** **class** CommonDaoImpl<T> **implements** CommonDao<T>{

**private** Class classt;

/\*\*

\* 元数据，用来描述持久化类的

\*/

**private** ClassMetadata classMetadata;

**public** CommonDaoImpl(){

ParameterizedType type = (ParameterizedType)**this**.getClass().getGenericSuperclass();

**this**.classt = (Class)type.getActualTypeArguments()[0];

/\*\*

\* clast就是代表持久化类的class形式

\*/

**this**.classMetadata = HibernateUtils.*sessionFactory*.getClassMetadata(classt);

}

@Override

**public** List<T> getAllEntry() {

Session session = HibernateUtils.*sessionFactory*.openSession();

List<T> list = session.createQuery("from "+**this**.classt.getName()).list();

**return** list;

}

@Override

**public** Long getCount() {

Session session = HibernateUtils.*sessionFactory*.openSession();

Long count = (Long)session.createQuery("select count("+**this**.classMetadata.getIdentifierPropertyName()+") from "+**this**.classt.getName()).uniqueResult();

**return** count;

}

@Override

**public** List<T> getEntriesByCondition(Map<String, Object> keyValues) {

StringBuffer buffer = **new** StringBuffer();

buffer.append("from "+**this**.classt.getName());

buffer.append(" where 1=1 ");

**for** (Entry<String, Object> entry : keyValues.entrySet()) {

/\*\*

\* from Classes where cid=:cid and name=:name

\*/

buffer.append("and "+entry.getKey()+"=:"+entry.getKey());

}

Session session = HibernateUtils.*sessionFactory*.openSession();

Query query = session.createQuery(buffer.toString());

**for** (Entry<String, Object> entry : keyValues.entrySet()) {

query.setParameter(entry.getKey(), entry.getValue());

}

**return** query.list();

}

@Override

**public** PageResult<T> getPageResult(BaseQuery baseQuery) {

PageResult<T> pageResult = **new** PageResult<T>();

pageResult.setCurrentPage(baseQuery.getCurrentPage());

pageResult.setPageSize(baseQuery.getPageSize());

StringBuffer buffer = **new** StringBuffer();

buffer.append("from "+**this**.classt.getName());//持久化类的名称

buffer.append(" where 1=1 ");

/\*\*

\* baseQuery.buildWhere()方法返回一个Map<String,Object>封装了查询条件

\*/

**for** (Entry<String, Object> entry : baseQuery.buildWhere().entrySet()) {

buffer.append(" and "+entry.getKey()+"=:"+entry.getKey());

}

System.***out***.println(buffer.toString());

Session session = HibernateUtils.*sessionFactory*.openSession();

Query query = session.createQuery(buffer.toString());

**for** (Entry<String, Object> entry : baseQuery.buildWhere().entrySet()) {

query.setParameter(entry.getKey(), entry.getValue());

}

List<T> list = query.setFirstResult(baseQuery.getCurrentPage()).setMaxResults(baseQuery.getPageSize()).list();

pageResult.setRows(list);

**return** pageResult;

}

}

ClassesDao:

**public** **class** ClassesDao **extends** CommonDaoImpl<Classes>{

}

PageResult:

**public** **class** PageResult<T>{

**private** List<T> rows;//属性代表数据

**private** **int** currentPage;//当前的页数

**private** **int** pageSize;//一页显示多少行

**public** List<T> getRows() {

**return** rows;

}

**public** **void** setRows(List<T> rows) {

**this**.rows = rows;

}

**public** **int** getCurrentPage() {

**return** currentPage;

}

**public** **void** setCurrentPage(**int** currentPage) {

**this**.currentPage = currentPage;

}

**public** **int** getPageSize() {

**return** pageSize;

}

**public** **void** setPageSize(**int** pageSize) {

**this**.pageSize = pageSize;

}

}

BaseQuery:

**public** **abstract** **class** BaseQuery {

**private** **int** currentPage;

**private** **int** pageSize;

**private** Map<String, Object> keyValues = **new** HashMap<String, Object>();

**public** Map<String, Object> getKeyValues() {

**return** keyValues;

}

**public** **void** setKeyValues(Map<String, Object> keyValues) {

**this**.keyValues = keyValues;

}

**public** **int** getCurrentPage() {

**return** currentPage;

}

**public** **void** setCurrentPage(**int** currentPage) {

**this**.currentPage = currentPage;

}

**public** **int** getPageSize() {

**return** pageSize;

}

**public** **void** setPageSize(**int** pageSize) {

**this**.pageSize = pageSize;

}

/\*\*

\* 子类继承该类，实现该抽象方法,把keyValues填充内容

\* **@return**

\*/

**public** **abstract** Map<String, Object> buildWhere();

}

ClassesQuery:

**public** **class** ClassesQuery **extends** BaseQuery{

**private** String name;

**private** String description;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getDescription() {

**return** description;

}

**public** **void** setDescription(String description) {

**this**.description = description;

}

@Override

**public** Map<String, Object> buildWhere() {

**if**(!StringUtils.*isNullOrEmpty*(name)){//如果不为null或者不为""

**this**.getKeyValues().put("name", name);

}

**if**(!StringUtils.*isNullOrEmpty*(description)){

**this**.getKeyValues().put("description", description);

}

**return** **this**.getKeyValues();

}

}

#### 查询全部

**public** **void** testQueryAll(){

ClassesDao classesDao = **new** ClassesDao();

List<Classes> classes = classesDao.getAllEntry();

System.***out***.println(classes.size());

}

#### 查询总数

**public** **void** testQueryCount(){

ClassesDao classesDao = **new** ClassesDao();

System.***out***.println(classesDao.getCount());

}

#### 条件查询

**public** **void** testQueryByCondition(){

Map<String, Object> keyValues = **new** HashMap<String, Object>();

keyValues.put("description", "asd");

ClassesDao classesDao = **new** ClassesDao();

List<Classes> classes = classesDao.getEntriesByCondition(keyValues);

System.***out***.println(classes.size());

}

#### 分页查询

**public** **void** testQueryDispage(){

ClassesDao classesDao = **new** ClassesDao();

ClassesQuery baseQuery = **new** ClassesQuery();

baseQuery.setCurrentPage(4);

baseQuery.setPageSize(3);

baseQuery.setName("asdf");

PageResult<Classes> pageResult = classesDao.getPageResult(baseQuery);

**for** (Classes classes : pageResult.getRows()) {

System.***out***.print("name:"+classes.getName()+" ");

System.***out***.println("cid:"+classes.getCid());

}

}

### 一对多

#### 内连接

/\*\*

\* 实现classes与student的内连接

\* select s.\*,c.\*

from student s inner join classes c on(s.cid=c.cid)

\*/

@Test

**public** **void** testInnerJoin(){

Session session = *sessionFactory*.openSession();

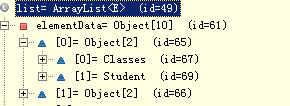
List list = session.createQuery("from Classes c inner join c.students").list();

session.close();

}



内存结构：List中是Object类型的数组



#### 迫切内连接

/\*\*

\* 迫切内连接

\*/

@Test

**public** **void** testInnerJoin\_fetch(){

Session session = *sessionFactory*.openSession();

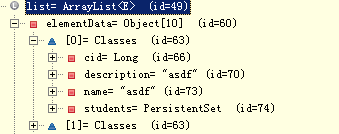
List list = session.createQuery("from Classes c inner join fetch c.students").list();

session.close();

}



内存结构：List中是持久化对象



#### 左外连接

/\*\*

\* 左外连接

\* select c.\*,s.\*

from classes c left outer join student s on(s.cid=c.cid)

\*/

@Test

**public** **void** testLeftJoin(){

Session session = *sessionFactory*.openSession();

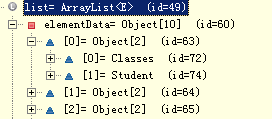
List list = session.createQuery("from Classes c left outer join c.students").list();

session.close();

}



内存结构：List中是Object类型的数组



#### 迫切左外连接

/\*\*

\* 迫切左外连接

\*/

@Test

**public** **void** testLeftJoin\_fetch(){

Session session = *sessionFactory*.openSession();

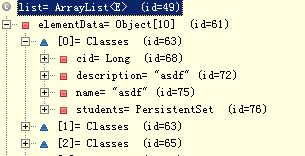
List list = session.createQuery("from Classes c left outer join fetch c.students").list();

session.close();

}



内存结构：List中是持久化对象



#### 要查询的属性来自于两个持久化类

/\*\*

\* 要查询的属性来自于两个持久化类

\* 带构造函数的查询和fetch的查询不能同时存在

\*/

@Test

**public** **void** testQueryPropertiesFromClassesAndStudent(){

Session session = *sessionFactory*.openSession();

List<ClassesView> classesViews = session.createQuery("select new com.itheima12.hibernate.domain.ClassesView (c.name,s.name) from Classes c inner join c.students s").list();

session.close();

}

ClassesView:

**public** **class** ClassesView {

**private** String cname;

**private** String sname;

**public** ClassesView(String cname, String sname) {

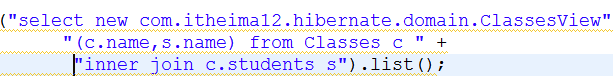
**super**();

**this**.cname = cname;

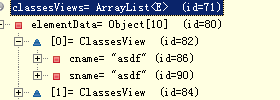
**this**.sname = sname;

}

}



内存结构：List中是持久化对象



### 多对多

#### 迫切内连接

**public** **void** testInnerJoin\_Fetch(){

Session session = *sessionFactory*.openSession();

session.createQuery("from Student s inner join fetch s.courses c").list();

session.close();

}

#### 迫切左外连接

**public** **void** testLeftOuterJoin\_Fetch(){

Session session = *sessionFactory*.openSession();

session.createQuery("from Student s left outer join fetch s.courses c").list();

session.close();

}

### 一对多结合多对多

#### 实体类

Student:

**public** **class** Student **implements** Serializable{

**private** Long sid;

**private** String name;

**private** String description;

**private** Set<Course> courses;

**private** Classes classes;

#### 映射文件

Student:

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<!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.itheima12.hibernate.onetomanytomany.domain.Student"*>

<id name=*"sid"* length=*"5"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"description"* length=*"50"*></property>

<property name=*"name"* length=*"20"*></property>

<!--

table

第三张表

-->

<set name=*"courses"* table=*"student\_course"* cascade=*"save-update"*>

<key>

<!--

外键

-->

<column name=*"sid"*></column>

</key>

<!--

column外键

-->

<many-to-many class=*"com.itheima12.hibernate.onetomanytomany.domain.Course"* column=*"cid"*></many-to-many>

</set>

<many-to-one name=*"classes"* class=*"com.itheima12.hibernate.onetomanytomany.domain.Classes"* column=*"cid"*></many-to-one>

</class>

</hibernate-mapping>

#### 方式一：

/\*\*

\* 查询所有的班级的所有的学生的所有的课程

\*/

@Test

**public** **void** testQuery\_1(){

Session session = *sessionFactory*.openSession();

List<Classes> classes = session.createQuery("from Classes c inner join " +

"fetch c.students s inner join " +

"fetch s.courses cc").list();

session.close();

}

#### 方式二：

**public** **void** testQuery\_2(){

Session session = *sessionFactory*.openSession();

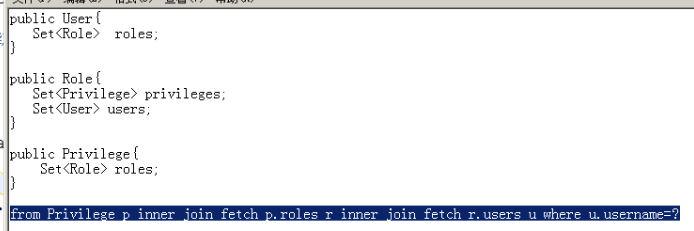
List<Student> classes = session.createQuery("from Student s inner join fetch s.classes c inner join fetch s.courses cc").list();

session.close();

}

### 多对多结合多对多

用户、角色、权限：



## Criteria查询

## sql查询

# 错误分析

## 错误1



该Person在hibernate的映射文件中没有找到:

1、在hibernate的配置文件中没有加载相应的映射文件

2、也有可能是客户端的session的操作的对象在映射文件中找不到

## 错误2



在映射文件中找不到该类，因为该类写错了。

## 错误3



当在hibernate内部生成sql语句的时候，要给insert语句赋值，值从属性的getter方法中得到，在拼接getter方法的时候，出错了。

## 错误4



当在执行session.get方法的时候，会利用Java的反射机制创建一个对象，利用的是默认的构造函数。**所以一个持久化类中必须有一个默认的构造函数**

## 错误5



要求主键必须在程序中进行设置

## 错误6



主键的生成策略支持表的主键的值的自动增长，但是该表本身没有这样的机制。

## 错误7

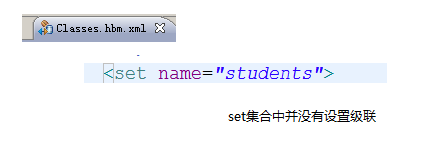


错误解析：

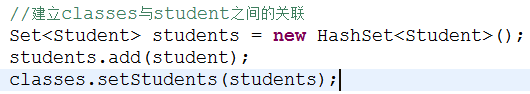
在flushing之前，保存一个临时状态的对象

前提条件：

在Classes.hbm.xml文件中，set集合中没有设置级联



在客户端：建立了关联关系



所以hibernate在保存classes的时候，就不知道该怎么办了。

## 错误8



## 错误9



错误原因：当一个对象从数据库中加载数据的时候，session已经关闭了

所有的数据在数据库中加载出来以后，session才能关闭。