# 1. Hibernate懒加载

懒加载（lazy load）也叫延迟加载，延迟加载的意思是只有在用到数据的时候才进行加载。即只有在用到关联对象的数据时，才会发送SQL查询这些关联数据，而并不是先把所有的关联数据都查询出来，否则效率很低。

Hibernate中有三类懒加载，分别是类的懒加载、集合的懒加载和单端关联的懒加载（即many-to-one的懒加载）。下面进行具体介绍并举例体验懒加载的作用。

## 1.1 类的懒加载

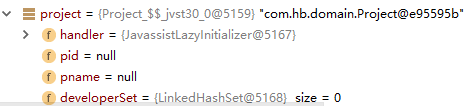
使用load方法可以实现类的懒加载。例如下面的代码：

|  |
| --- |
| Session session = sessionFactory.openSession(); Project project = session.load(Project.**class**, 1); session.close(); |

执行上述的代码发现程序从运行到结束没有发出一条SQL查询语句。这就是懒加载，因为我们没有使用project对象的数据，因此不会发送SQL加载数据。下面的代码：

|  |
| --- |
| Session session = sessionFactory.openSession(); Project project = session.load(Project.**class**, 2); System.***out***.println(project); session.close(); |

上面的代码由于会使用project，因此会加载数据。通过打断点我们发现，从load方法得到的对象其实是代理对象，这个代理类就是持久化对象的子类。这是利用javassist实现的。例如这里的project类型：



就是“Project\_$$\_jvst30\_0”。

可以在映射文件中，设置class节点的lazy属性值，默认为true，即使用懒加载。如果设置为false，就是关闭本类的懒加载功能，这样调用load方法就不会懒加载，而首先执行SQL查询数据。

使用懒加载时一个典型的错误就是session被关闭后才使用懒加载数据。例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); Project project = session.load(Project.**class**, 3); session.close(); *// 关闭了session* System.***out***.println(project.getPname()); *// 使用懒加载的数据* |

运行程序则会出现下述异常：

|  |
| --- |
| org.hibernate.LazyInitializationException: could not initialize proxy - no Session |

也就是说，如果由于进行了懒加载，数据还未被加载前session就被提前关闭了，那么就不能在以后使用对象的数据，否则报错。

那么如何解决这个问题呢，有如下两种方法：

（1）在session关闭之前使用一下懒加载的数据；

（2）在映射文件中关闭懒加载，设置lazy为false。

## 1.2 集合的懒加载

当获得关联对象的集合数据时，就会进行集合的懒加载功能。例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); Project project = session.get(Project.**class**, 1); Set<Developer> developerSet = project.getDeveloperSet(); *// 遍历开发者* **for**(Developer developer : developerSet) {  System.***out***.println(developer.getDevName()); } session.close(); |

上述代码中，通过get方法得到project后，又获得关联的集合对象。由于懒加载，获得关联对象的集合时才会进行关联对象的加载，而调用get时并不会加载全部数据。

在映射文件的set标签中，也可通过lazy属性设置关联对象集合是否进行懒加载。在这里，lazy的属性值可以设置为以下三种：

（1）lazy设置为true：这是默认值，即使用集合的懒加载。

（2）lazy设置为false：不使用懒加载，即会先把所有关联数据都查询出来（比如加载项目的同时加载开发者信息）。

（3）lazy设置为extra：这是“更进一步的懒加载”，比设置为true还“厉害”。什么意思呢？例如如下代码：

|  |
| --- |
| Session session = sessionFactory.openSession(); Project project = session.get(Project.**class**, 1); System.***out***.println(project.getDeveloperSet().size()); session.close(); |

如果在set标签中设置了lazy为true或false，程序最终都会执行两条查询SQL：

|  |
| --- |
| select project0\_.pid as pid1\_1\_0\_, project0\_.pname as pname2\_1\_0\_ from project project0\_ where project0\_.pid=?  select developers0\_.pid as pid1\_2\_0\_, developers0\_.devId as devId2\_2\_0\_, developer1\_.devId as devId1\_0\_1\_, developer1\_.devName as devName2\_0\_1\_ from relation developers0\_ inner join developer developer1\_ on developers0\_.devId=developer1\_.devId where developers0\_.pid=? |

但如果lazy设置为extra，上述代码最终执行这样的两条SQL：

|  |
| --- |
| select project0\_.pid as pid1\_1\_0\_, project0\_.pname as pname2\_1\_0\_ from project project0\_ where project0\_.pid=?  select count(devId) from relation where pid =? |

也就是说设置为extra后，当你只查询数据集合大小时，Hibernate只通过count()查询个数，而不是查询所有数据。

同样，集合的懒加载也要注意session提前关闭的异常。

## 1.3 单端关联的懒加载

单端关联的懒加载其实就是many-to-one的懒加载，即通过多的一方数据懒加载一的一方数据。这类懒加载我们就不研究了，因为一的一方只会关联到一个数据，不管是否使用懒加载，都对性能没什么影响，想怎么做就怎么做。

总结说明：

（1）延迟加载是在映射文件中配置的，而不是由各自的代码决定的。在实际的团队开发中，映射文件一旦确定就不能随意修改，这是需要注意的。

（2）延迟加载是通过控制SQL语句发出的时间来提高效率的。

## 1.4 总结

懒加载就是在真正使用数据时才加载数据。例如以后在Web开发中，使用懒加载在DAO中查询数据，但并不是在调用DAO时就查询数据，而有可能在访问JSP时，访问到持久化对象的属性时才加载数据。

之所以使用懒加载，是因为有时候某些关联对象的信息可能不会被使用。例如查询部门时不需要查询员工信息，除非用户确实点击了按钮查看部门下的员工信息。

还有一个需要注意，双向关联时不要产生调用关联对象的“死循环”：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Session session = **new** Configuration().configure().buildSessionFactory().openSession();  session.beginTransaction();  *// 得到员工* Employee employee = session.get(Employee.**class**, **"YG001"**);  *// 员工姓名* System.***out***.println(employee.getName());  *// 部门名称* System.***out***.println(employee.getDept().getName());  *// 员工该部门下所有的员工* Set<Employee> employeeSet = employee.getDept().getEmployeeSet();  **for** (Employee emp : employeeSet) {  System.***out***.println(emp.getName());  }  *// 不要用system.out.println(employee)直接输出一个对象。  // 因为他们会相互调用，出现死循环而出错。* session.getTransaction().commit();  session.close();  session.getSessionFactory().close();  } } |

最后，在映射文件的package节点中，设置default-lazy值为false，这样就会针对本持久化类关闭所有的懒加载。

# 2. 抓取策略

先看一个查询所有项目和所有开发者的例子：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 加载所有项目的所有开发者* List<Project> projectList = session.createQuery(**"from Project "**).list(); *// 遍历项目* **for** (Project project : projectList) {  *// 再遍历开发者* **for** (Developer dev : project.getDeveloperSet()) {  System.***out***.println(project.getPname() + **"的开发者有："** + dev.getDevName());  } } session.close(); |

运行发现，Hibernate是这样查询的：先查询所有的项目，然后查询开发者时，每查询一个项目的开发者，就按照项目的pid来查询开发者信息。这样就会产生“N+1”条查询SQL，N就是查询关联表中数据所用的SQL，1就是查询本表所有数据的SQL。同样，使用iterate方法也会先查询本表数据，再根据本表的主键去匹配查询关联表的相关数据，也会产生N + 1条SQL。实际上如果我们自行使用SQL查询项目和对应开发者的所有数据的话，直接使用一条子查询就可解决问题。

所以上述的Hibernate做法是不推荐的，特别是数据量大的时候。而抓取策略主要解决的问题就是如何加载Set集合中数据的问题。这里还是不研究many-to-one的抓取策略，因为一方只有一条数据，相比多的一方的大数据量可以忽略。

在set标签中，fetch属性可以设置如下值：

select：默认的值。查询时先查询本表数据，再通过主键查询关联表数据（N+1）

join：查询时使用左外连接

subselect：使用子查询

例如经过分析，上述例子中应该使用子查询，因此我们设置fetch为join，再运行程序，则执行的SQL变为（格式化了）：

|  |
| --- |
| Hibernate: （第一条，和之前一样）  select  project0\_.pid as pid1\_1\_,  project0\_.pname as pname2\_1\_  from  project project0\_  Hibernate: （第二条，使用了子查询）  select  developers0\_.pid as pid1\_2\_1\_,  developers0\_.devId as devId2\_2\_1\_,  developer1\_.devId as devId1\_0\_0\_,  developer1\_.devName as devName2\_0\_0\_  from  relation developers0\_  inner join  developer developer1\_  on developers0\_.devId=developer1\_.devId  where  developers0\_.pid in (  select  project0\_.pid  from  project project0\_  ) |

上述的例子中，抓取策略不能设置为join，因为是包含子查询的，设置为join也会使用默认select的抓取策略。但下面的例子可使用join抓取策略（查询一个项目的所有开发者）：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 加载所有项目的所有开发者* Project project = session.get(Project.**class**, 1); *// 遍历开发者* **for** (Developer dev : project.getDeveloperSet()) {  System.***out***.println(project.getPname() + **"的开发者有："** + dev.getDevName()); } session.close(); |

最终执行的一条SQL是：

|  |
| --- |
| Hibernate:  select  project0\_.pid as pid1\_1\_0\_,  project0\_.pname as pname2\_1\_0\_,  developers1\_.pid as pid1\_2\_1\_,  developer2\_.devId as devId2\_2\_1\_,  developer2\_.devId as devId1\_0\_2\_,  developer2\_.devName as devName2\_0\_2\_  from  project project0\_  left outer join  relation developers1\_  on project0\_.pid=developers1\_.pid  left outer join  developer developer2\_  on developers1\_.devId=developer2\_.devId  where  project0\_.pid=? |

总结：如果加载集合的时候，通过分析需要使用子查询，则使用subselect效率最高；如果想把一的一方和多的一方一次性加载出来，就是用左外链接；如果想先加载一的一方，再加载多的一方就使用默认的select。

这里是通过在映射文件中设置的。正如我们之前所说，映射文件中是不好随意更改的，因此在下面，Hibernate也提供了在代码中的抓取策略操作方式。

# 3. Hibernate的查询

在Hibernate中，除了使用主键查询外，还支持其他三种查询，分别是HQL查询、Criteria查询（条件查询）和原生SQL查询。

## 3.1 HQL查询

HQL查询就是使用Hibernate提供的查询语言进行查询，比较常用。

### 3.1.1 单表查询

最简单的单表查询就是查询出表中所有数据，例如（之前写过）：

|  |
| --- |
| Session session = sessionFactory.openSession(); List<Project> projects = session.createQuery(**"from Project "**).list(); System.***out***.println(projects); session.close(); |

也可以查询特定的属性：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 不能再用泛型接收* List list = session.createQuery(**"select pid, pname from Project "**).list(); *// 遍历list* **for** (**int** i = 0; i < list.size(); i++) {  *// list中的每个元素是一个Object数组，每个数组元素才是查询出来的一项纪录* Object[] objArr = (Object[]) list.get(i);  System.***out***.println(**"编号："** + objArr[0]);  System.***out***.println(**"姓名："** + objArr[1]); } session.close(); |

从上述代码中看出，此时list方法返回的list集合中，存储的每个元素是Object数组，而Object数组中每个元素才是查询出的数据，数组的长度就是查询字段的个数。

上述的结果不便使用，实际上也可以把上述的结果封装成持久化对象集合。首先，在Project持久化对象中提供pid和pname的构造函数：

|  |
| --- |
| **public** Project(Integer pid, String pname) {  **this**.**pid** = pid;  **this**.**pname** = pname; } |

这样在查询时，就可使用下面的语法封装成Project对象集合：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 使用泛型接收* List<Project> projects = session.createQuery(**"select new Project(pid, pname) from Project "**).list(); *// 遍历projects* **for** (Project project : projects) {  System.***out***.println(project.getPid());  System.***out***.println(project.getPname()); } session.close(); |

当然，对于没有进行查询的字段（属性），值是没有的。这就是带有构造函数的查询，注意先提供相应的构造方法。

如果查询结果只有一行，可以使用uniqueResult方法得到结果数据。例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); Long count = (Long) session.createQuery(**"select count(pid) from Project "**).uniqueResult(); System.***out***.println(count); session.close(); |

上述就得到Project表的总行数。

另外，HQL中也可以设置预编译参数。设置预编译参数有两种方式。

方式一：使用问号“?”，并调用query.setParameter设置参数，索引从0开始。例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); Query query = session.createQuery(**"from Project where pid = ? and pname = ? "**); query.setParameter(0, 2); *// 设置参数1。注意类型要匹配，pid是Integer类型。因此这里传递2，而不是字符串2。* query.setParameter(1, **"图书管理系统"**); *// 设置参数2，类型就是字符串* Project project = (Project) query.uniqueResult(); System.***out***.println(project.getPname()); session.close(); |

方式二：在参数名前使用冒号“:”，调用query.setParameter设置参数时，需要设置相应的参数名。示例代码：

|  |
| --- |
| Session session = sessionFactory.openSession(); Query query = session.createQuery(**"from Project where pid = :pid and pname = :pname"**); query.setParameter(**"pid"**, 2); *// 设置pid参数* query.setParameter(**"pname"**, **"图书管理系统"**); *// 设置pname参数* Project project = (Project) query.uniqueResult(); System.***out***.println(project.getPname()); session.close(); |

需要注意的是，上述两种方法在设置参数值时，必须使用匹配的类型，否则Hibernate会报类型转换异常的错误。

在query对象中，提供了setFirstResult(int index)和setMaxResults(int num)方法用于分页查询。其中setFirstResult用于设置集合索引的位置（即从哪个索引开始取集合的数据），setMaxResults用于设置取出数据的个数（即一页显示多少个）。例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); Query query = session.createQuery(**"from Project "**); query.setFirstResult(2); query.setMaxResults(3); List<Project> projectList = query.list(); System.***out***.println(projectList.size()); session.close(); |

在进行分页时，直接使用setFirstResult(（当前页-1） \* 每页条数)；setMaxResults(每页条数)即可。

### 3.1.2 一对多和多对多查询

例1：实现一对多的内连接查询。例如内连接查询部门和员工数据的SQL语句是：

|  |
| --- |
| select d.\*, e.\* from dept d inner join employee e on e.deptId = d.id; |

如果要在HQL中进行这样的内连接查询，则为：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 使用别名。且HQL中能直接用“关联的属性”进行连接，比较方便。* List list = session.createQuery(**"from Dept d inner join d.employeeSet "**).list(); session.close(); |

通过断点我们发现此时list结构是这样的：



即list中的元素是Object数组，Object数组中的元素才是Dept和Employee对象。这样的结构使用比较麻烦，因此Hibernate提供了“迫切内连接”的使用方法。

迫切内连接就是在join后使用“fetch”关键字，这样返回的list数据中，元素就是本持久类对象，然后关联的对象中也保存了内连接查询到的数据。代码示例如下：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 使用fetch，这样可使用泛型了* List<Dept> depts = session.createQuery(**"from Dept d inner join fetch d.employeeSet "**).list(); **for** (Dept dept : depts) {  *// 再遍历Employee* **for** (Employee e : dept.getEmployeeSet()) {  System.***out***.println(e.getName() + **"所在的部门是："** + dept.getName());  } } session.close(); |

同样，可以使用“左外连接”和HQL中的“迫切左外连接”，这里以使用“迫切左外连接”为例，使用“outer left join fetch”即可：

|  |
| --- |
| Session session = sessionFactory.openSession(); List<Dept> depts = session.createQuery(**"from Dept d left outer join fetch d.employeeSet "**).list(); **for** (Dept dept : depts) {  **for** (Employee e : dept.getEmployeeSet()) {  System.***out***.println(e.getName() + **"所在的部门是："** + dept.getName());  } } session.close(); |

凡是使用“迫切”的，都能得到较好使用的数据结构。

下面将查询两个持久化类连接中得到的某些属性（不是全部属性），例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 注意在下面的HQL中，最后的“d.employeeSet e”相当于是给employee起了别名e，因此select时可以使用别名e。* List list = session.createQuery(**"select d.name, e.name from Dept d left outer join d.employeeSet e "**).list(); *// 都仅查询名称* session.close(); |

上述的list的结构也是其中是Object数组，不太好用，但是这种只查询某些属性的HQL是无法使用fetch的，因为无法转换成合适的对象数据。如果使用fetch会报错。为了解决这个问题，可以单独新建一个类用于存放这样的数据便于使用。例如先新建一个DeptEmployeeView类：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 下面HQL中使用DeptEmployeeView需要写全限定名，因为该类并不是Hibernate映射文件中指定的持久化类，无法直接识别，因此要使用全限定名* List<DeptEmployeeView> list = session.createQuery(**"select new com.hb.domain.DeptEmployeeView(d.name, e.name) from Dept d left outer join d.employeeSet e "**).list(); *// 都仅查询名称* **for** (DeptEmployeeView dev : list) {  System.***out***.println(dev.getDeptName());  System.***out***.println(dev.getEmployeeName()); } session.close(); |

注意上面要使用全限定名（因为映射文件并不能指定该类），而且该HQL中也不能使用fetch，因为带构造函数的查询和fetch的查询不能同时存在，带构造方法的HQL已经知道要封装成什么样的数据结构了。

上述都是在一对多中操作的，实际上多对多中进行这样的操作和上述基本是完全一样的，使用HQL进行多对多查询甚至比HQL使用的更简单。比如这里就举出一个迫切内连接的例子：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 多对多迫切内连接(注意加上fetch)* List<Project> projects = session.createQuery(**"from Project p inner join fetch p.developerSet"**).list(); **for** (Project project : projects) {  **for** (Developer developer : project.getDeveloperSet()) {  System.***out***.println(project.getPname() + **"的开发者是："** + developer.getDevName());  } } session.close(); |

为什么说简单呢？这是因为根本不用关心第三张关系表，而自己写SQL要两个inner join。迫切左外连接只是使用“left outer join”罢了。

### 3.1.3 一对多+多对多查询

在做该案例之前，我们先把Hibernate项目改造成“一对多 + 多对多”的结构。即：部门和开发者（员工）是一对多关系，然后开发者和项目是多对多关系。这样就去掉原来的员工Employee类，将Dept、Developer和Project改造成这样的关系。

熟悉关联映射的话，进行改造还是比较容易的，只是要在Developer中既配置与部门的一对多关系，又要配置与项目的对多对关系。这里就不贴相关的代码和配置文件内容了，读者可以在“project-demos”目录下找到HibernateSrc.zip文件，其中就是我已写好的代码和配置，使用即可，注意修改主配置文件中的数据库连接信息。如果要阅读这个代码和配置文件，按照Dept-Developer-Project的顺序看就可以了。

程序运行会创建4张表，分别是dept、developer、project和relation。读者可自行向表中添加一些数据，以便进行下面的“一对多+多对多”查询操作。

例1：查询所有部门下所有开发者和对应的项目。

|  |
| --- |
| Session session = sessionFactory.openSession(); List<Dept> depts = session.createQuery(**"from Dept d inner join fetch d.developerSet dev inner join fetch dev.projectSet "**).list(); *// 遍历depts* **for** (Dept dept : depts) {  *// 遍历developer* **for** (Developer developer : dept.getDeveloperSet()) {  *// 遍历project* **for** (Project project : developer.getProjectSet()) {  System.***out***.println(**"开发者："** + developer.getDevName() + **"，开发："** + project.getPname());  }  } } session.close(); |

注意不要这样写HQL：

|  |
| --- |
| session.createQuery(**"from Dept d inner join fetch Developer dev inner join fetch Project p "**).list(); |

是会出错的。

通过上面的例子我们发现使用HQL是“通用”的思路和写法，较SQL要简单，“傻瓜化”。

这里要知道，为什么用List<Dept>接收返回的结果呢？这是因为from后面全都使用Dept进行“关联查询”的，因此可以用List<Dept>接收。如果是“from Project ....关联查询”，那么接收的数据类型就应该是List<Project>。

学会了一对多+对多对的操作，我们就能使用这样的方式比较简单地开发“权限管理系统”了。因为权限管理系统就是类似这样的关系。这样的关系Hibernate还能较好的应付，但是表的关系一旦非常复杂，就不适合用Hibernate了，开发会异常恶心。所以Hibernate的应用场合实际上很有限。

## 3.2 Criteria查询和SQL查询

Criteria查询就是Hibernate提供的完全的以面向对象的方式查询数据的方式。有兴趣可以了解。SQL查询就是在Hibernate中利用原生SQL进行的查询，以前讲过，不再赘述。但这两种方式都不太常用。

# 4. Hibernate常用技术

## 4.1 创建session的方式

之前我们是通过“sessionFactory.openSession()”的方式获得session对象的。使用这种方式创建session时，每次都会创建一个新的session对象，相当于是创建了一个新的连接，这是比较耗费资源的。并且如果想在业务层控制事务，这样是不易控制的，因为每访问一次DAO方法，就会创建一个新的session对象。而事务一般需要在同一个会话中才能控制。

Hibernate通过调用“sessionFactory.getCurrentSession()”方法来解决上述的问题。getCurrentSession会得到一个与当前线程绑定的Session对象，这样在一个线程中，通过该方法就能拿到同一个session对象，解决了耗费资源和事务处理的问题。

一次请求的处理一般都是在同一个线程中进行流转，因此从当前线程拿到session是没有问题的。这种方式的实现也是通过ThreadLocal实现的，和我们之前在JDBC中做的大同小异。在Hibernate中调用getCurrentSession，就是从当前线程拿到Session，如果session不存在，内部就会通过openSession来创建一个session并绑定到当前线程。

如果需要在一个线程共享数据，那么把该数据放入到ThreadLocal中是一个很好的选择。现在很多框架都会这样使用ThreadLocal。利用ThreadLocal存储数据，那么就能在任何地方拿到该数据，只要代码是在这个线程中运行的即可。

要想使用“sessionFactory.getCurrentSession()”这种方式得到session，还需要在Hibernate主配置文件中加上如下配置：

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

使用该方式获得session进行使用时，需要注意以下两点：

（1）crud操作进行事务的控制，包括查询的操作也要在事务控制中，否则程序会出现如下异常：

|  |
| --- |
| org.hibernate.HibernateException: createQuery is not valid without active transaction |

（2）使用这种方式，当执行transaction.commit()方法提交事务后，session会自动关闭。因此不要手动关闭，否则出现“session is already closed”的异常。而且，这也导致事务提交后不能再使用该session。在学习后面的Spring框架后，实际上是把事务的提交放在Service层中控制，也就是执行完业务方法后session才关闭，因此一般是这样结合Hibernate使用。在实际开发中，一般都使用sessionFactory.getCurrentSession这种方式。

## 4.2 在Hibernate中使用连接池

Hibernate自带就有一个连接池，不过此连接池比较简陋，不适合在生产环境中使用。

我们这里使用一个名为“druid”的数据库连接池，它是支持与Hibernate集成的。druid是阿里巴巴开发的数据库连接池，号称是Java最好的数据库连接池。

（1）首先，下载druid的jar包（下载地址：

http://central.maven.org/maven2/com/alibaba/druid/，使用1.1.6版本的即可），并将该包引入到项目中。

（2）在主文件中配置连接池（取代之前的数据库连接信息）：

|  |
| --- |
| *<?***xml version='1.0' encoding='utf-8'***?>* **<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"*>*** <**hibernate-configuration**>  <**session-factory**>  *<!-- 1. 先配置数据库连接池 -->  <!-- 配置连接池驱动管理类。设置为druid的支持类。这是Hibernate的属性 -->* <**property name="connection.provider\_class"**>com.alibaba.druid.support.hibernate.DruidConnectionProvider</**property**>  *<!-- 下面的属性不是hibernate提供的属性，而是druid需要的属性 -->  <!-- 连接数据库的url -->* <**property name="url"**>jdbc:mysql://114.55.86.230:3306/mytest?useSSL=false**&amp;**useUnicode=true**&amp;**characterEncoding=utf-8</**property**>  *<!-- 用户名 -->* <**property name="username"**>root</**property**>  *<!-- 密码 -->* <**property name="password"**>123qwe!@#</**property**>  *<!-- 驱动类 -->* <**property name="driverClassName"**>com.mysql.jdbc.Driver</**property**>  *<!-- 初始化时建立物理连接的个数。 -->* <**property name="initialSize"**>2</**property**>  *<!-- 最大连接池数量 -->* <**property name="maxActive"**>8</**property**>  *<!-- 最小连接池数量 -->* <**property name="minIdle"**>1</**property**>  *<!-- 2. 配置其他的Hibernate的必要属性 -->  <!-- 设置数据库方言 -->* <**property name="dialect"**>org.hibernate.dialect.MySQLDialect</**property**>  *<!-- 显示执行的SQL -->* <**property name="show\_sql"**>true</**property**>  *<!-- 格式化显示的SQL -->* <**property name="format\_sql"**>true</**property**>  *<!-- 其他配置... -->  <!-- 3. 加载映射文件 -->* <**mapping resource="com/hb/domain/Dept.hbm.xml"**/>  <**mapping resource="com/hb/domain/Developer.hbm.xml"**/>  <**mapping resource="com/hb/domain/Project.hbm.xml"**/>  </**session-factory**> </**hibernate-configuration**> |

上述很多druid属性就是用于数据库连接的，因此原本的hibernate连接信息不用重复配置。更多druid属性参见网址：

“https://github.com/alibaba/druid/wiki/DruidDataSource%E9%85%8D%E7%BD%AE%E5%B1%9E%E6%80%A7%E5%88%97%E8%A1%A8”

更多druid信息也可在github上查看：https://github.com/alibaba/druid。

Hibernate也支持其他连接池，配置方法大同小异，遇到问题自行搜索（不推荐使用C3P0）。

# 5. Hibernate逆向工程

Hibernate是JBoss负责维护的，JBoss提供了一个名叫“Hibernate Tool”的Eclipse插件，该插件能根据我们在数据库中设计好的数据库表，自动生成实体类和映射文件！这种根据数据库自动生成Hibernate相关文件的方法，我们称之为Hibernate的“逆向工程”。

为何生成Hibernate相关文件是可行的呢？因为我们手动编写Hibernate配置文件也大部分是根据设计的表来进行的。对于较复杂的表结构，使用Hibernate Tool能大幅减少开发者工作，但前提是表的设计必须合理且适合使用Hibernate框架。当然，利用Hibernate Tool生成的文件也不是100%满足要求，有时我们还需根据实际情况进行微调，如懒加载配置、级联设置等。

## 5.1 环境配置

Hibernate Tools包含在JBoss Tools工具包中，因此在Eclipse中安装JBoss Tools插件中的Hibernate Tool即可。如果安装不能成功，可以直接使用“JBoss Developer Studio”开发工具，该开发工具是JBoss基于Eclipse EE开发的集成开发环境，其中已经集成了JBoss Tools全部工具。

## 5.2 使用Hibernate Tool

进行Hibernate的逆向工程，需要做如下操作：

（1）在Eclipse中新建数据源。

在“Window – Show View”中选择打开“Data Source Explorer”。在打开的选项卡中右击“Database Connections”，点击“New…”菜单项以便连接数据库。

在弹出的框中选择好使用的数据库（如MySQL），可输入名称Name和描述Description，点击Next。Drivers中选择合适的驱动。初次使用时可能没有合适的驱动，需要自己点击选择框右边的图标按钮“New Driver Definition”进行添加。添加时，我们可选择最新的5.1版本驱动，当然此时也要在“JAR List”选项卡中选择好对应的5.1版本的JDBC驱动包。

驱动加载好后，再设置MySQL数据库配置，输入数据库名、URL等，然后就可点击Finish完成了。完成后，Database Connections中就能看到连接到的数据库和表（Schemas）。

（2）配置Hibernate Configurations

在“Window – Show View”中选择打开Hibernate Configurations。在该标签中，右键“Add Configuration…”，可以给该配置起个名字，默认为“hibernate”。选择“Type”为“Core”。选择好Hibernate版和Project项目，并且设置“Database connection”为刚刚设置的MySQL数据源连接后，就可点击OK完成配置了。

（3）进行逆向工程

点击“Run – Hibernate Code Generation… - Hibernate Code Generation Configurations…”。选中其中的“Hibernate Code Generation”，再点击左上角的“新建（New launch configuration）”图标新建一个配置，可起名为“hibernate逆向工程”，在Main选项卡中，选择“Console configuration”为刚设置的“hibernate”，设置Output directory为本项目的src路径；勾选“Reverse engineer from JDBC Connection”，并设置包名（如com.entity）。除此之外，Main选项卡无需再手动配置（如果想指定某几张数据库表来生成Hibernate映射，则可设置reveng.xml）。

在“Exporters”选项卡中，勾选“Use Java 5 syntax”，表示会使用Java 5的语法，即会使用泛型，推荐使用。若勾选“Generate EJB3 annotations”，表示是使用注解方式，此时就不用生成映射文件了。

在“Reporters”中，一般会勾选“Domain code”、“Hibernate XML Mappings”和“DAO code”，分别表示生成实体类、映射文件和基本的DAO层代码。按需选择即可。完毕后，点击“Run”即可生成代码和文件，逆向工程完毕。