# 1. 缓存

## 1.1 缓存简介

缓存就是把一些常用数据临时存放在内存区域中，应用会优先从缓存中查找数据，以便提高速度和效率。

现在常用的缓存框架有ehcache，memcached，redis，hbase等。一个简单的缓存框架基本上要实现以下功能：

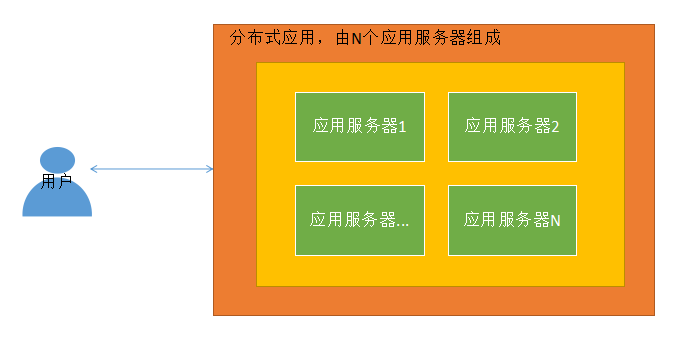
1. 提供操作缓存的API，能够将数据放入缓存中、能够从缓存中把数据提取出来等。

2. 解决数据同步问题。如果缓存中的数据发生变化，需要将数据同步到数据库，同样，也需要把数据库中的数据同步到缓存中。这时就需要有一个判断缓存数据和数据库中数据是否一致的依据，如使用副本等。另外如果是分布式缓存，还要处理缓存服务器间的数据同步，这个在下面再说明。

3. 将命中率（hits）越低的对象从缓存中移除。

关于分布式缓存：

在大型应用中，一般要使用分布式，就是把应用部署到多个服务器上。如果把Java Web应用部署成分布式应用，则把应用部署到不同机器的不同JVM上。但是当用户访问应用时，分布式对用户来说是透明的，用户只要能进行请求和获得响应即可，而具体访问哪个服务器是不知道的。如图：



在这种分布式应用中，要解决的一个问题就是各个服务器之间的session同步问题（session一致性问题）。因为用户可能第一次请求时访问应用服务器1，第二次请求时访问服务器2，为了记住用户的session信息，必须要同步session。分布式应用的具体情况在以后的高级内容会涉及。

同样，大型应用也会做分布式缓存，分布式缓存是提供给应用服务器使用的。同样对于应用开发者来说，使用分布式缓存也是透明的，开发者好像是操作同一个缓存服务器，这时分布式缓存中的各个服务器之间也要做数据同步，使得开发者拿到的数据一致。其实上述的session一致性问题也是通过分布式缓存解决的，从缓存中拿到session。

说明的是，目前不涉及到分布式的问题，这里只是一些概念。

## 1.2 Hibernate的一级缓存

Hibernate的一级缓存，也叫做session级别的缓存。一级缓存的生命周期和session的生命周期保持一致，即：获得sesison时创建缓存，session关闭时缓存就失效。

Hibernate中的一级缓存其实就是一个内存中的Map对象，一次Session会话中的持久化对象就是保存在这个Map对象中。实际上我们前面讲的get、save、evict等方法，就是把对象放入（或移出）这个Map对象中。

（1）get方法：用于把查询的对象到一级缓存中，也可以从一级缓存中把对象提取出来。

例如在Session中使用get两次查询同一个对象数据，则只会发送一次查询SQL：

|  |
| --- |
| Session session = **sessionFactory**.openSession(); Transaction transaction = session.beginTransaction(); Student student = session.get(Student.**class**, 1); *// 获得主键为1的对象数据* Student student1 = session.get(Student.**class**, 1); *// 再获取一次* transaction.commit(); *// 提交事务，最终只会有一次查询* session.close(); |

（2）save方法：该方法可以把一个对象放入到一级缓存中，即变成持久化对象。我们同统计session中实体个数验证一级缓存中的对象个数：

|  |
| --- |
| Student student = **new** Student(); student.setName(**"Tom"**); session.save(student) System.***out***.println(session.getStatistics().getEntityCount()); *// 将输出1* |

（3）evict方法：把一个对象从session的缓存中清除。

（4）update方法：把一个对象放入到一级缓存中。（把一个持久化对象先evict再update，并不会再发出查询SQL，因为对象没有改变。）

（5）clear方法：清空一级缓存中所有的数据。

（6）close方法：session关闭，该session中一级缓存的生命周期结束。

设计一级缓存的意义是使在一次会话中，更新或保存这些操作在最后只与数据库交互一次。一级缓存中的持久化数据是先通过save或者get方法得到的，在这之后，一级缓存中的持久化数据可以根据业务要求任意操作，都不会和数据库交互，最终在事务提交前执行flush方法时，最终决定需要一次性执行的insert或者update语句，实现与数据库进行本会话的最后一次交互。

特别注意一级缓存的有效期就是本次会话，本次session会话的缓存数据是不能保存到被下次会话所使用的。

# 2. Hibernate的关联映射

要想充分使用Hibernate的强大功能，必须先设计好数据库，分析实体关系。然后根据数据库表的关系写好Hibernate的映射文件。

数据库实体之间的常用关系有一对多、多对多和一对一，在Hibernate映射时，也能体现实体间的关系，这样使用Hibernate效率较高。下面通过例子来讲解使用关联映射。

## 2.1 一对多单向映射

例子：部门与员工之间的关系就是一对多的关系，即一个部门下有多个员工。我们需要在映射文件中体现这一关系。

我们可以不自己建立数据库表，因为Hibernate有自动建表功能。我们按照逻辑写好映射文件后，直接使用自动建表，也正好可以通过查看建好的表来验证映射的正确性。

下面是开发的步骤。

（1）先设计好实体类。基于部门与员工的关系，那么部门实体类中应该有维护所有员工的属性，员工实体类中也应该有维护一个部门的属性。例如：

部门实体类：

|  |
| --- |
| **package** com.hb.domain;  **import** java.io.Serializable; **import** java.util.LinkedHashSet; **import** java.util.Set;  **public class** Dept **implements** Serializable {  **private** Integer **deptId**; *// 编号* **private** String **name**; *// 部门名称* **private** Set<Employee> **employeeSet** = **new** LinkedHashSet<>(); *// 维护的员工。  // 用set集合表示，便于关联，也可去除重复元素。用LinkedHashSet保证元素有序   // 自行写setter/setter/toString()等方法* } |

员工实体类：

|  |
| --- |
| **package** com.hb.domain;  **import** java.io.Serializable;  **public class** Employee **implements** Serializable {  **private** Integer **eid**; *// 编号* **private** String **name**; *// 姓名  // 现在做单向映射，这里无需维护员工部门信息   // 自行写getter/setter/toString()等方法* } |

注意实体类都应该写上无参数的构造方法，否则主键查询会出错。

（2）配置映射文件

首先，编写Dept.hbm.xml：

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.hb.domain"**>  <**class name="Dept" table="dept"**>  *<!-- 主键 -->* <**id name="deptId" column="id"**>  <**generator class="increment"** />  </**id**>  *<!-- 部门名称 -->* <**property name="name" column="name" length="20"** />   *<!-- 一对多关系配置 -->  <!-- set中，指定employeeSet关联的表是employee。 -->* <**set name="employeeSet" table="employee"**>  *<!-- key标签中，用column属性指定employee表的外键是deptId -->* <**key column="deptId"** />  *<!--one-to-many表明一对多映射，多的一方类型是Employee类型-->* <**one-to-many class="Employee"** />  </**set**>  </**class**> </**hibernate-mapping**> |

这样配置“一”的一方的映射文件，就能让Hibernate知道如何进行部门和员工的关联查询，只要通过employee表的deptId外键进行相关查询即可。

其次，编写Employee.hbm.xml。由于我们做的是一对多的单向关联，并且在Dept.hbm.xml中进行了单向关联配置，因此这里Employee.hbm.xml中并不要做什么特别的配置：

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.hb.domain"**>  <**class name="Employee" table="employee"**>  *<!-- 主键 -->* <**id name="eid" column="eid"**>  <**generator class="increment"** />  </**id**>  *<!-- 员工姓名 -->* <**property name="name" column="name" length="20"** />  </**class**> </**hibernate-mapping**> |

（3）配置Hibernate主配置文件。注意加上自动建表的配置，并且引入上述两个映射文件，即：

|  |
| --- |
| <**property name="hbm2ddl.auto"**>update</**property**> <**mapping resource="com/hb/domain/Dept.hbm.xml"**/> <**mapping resource="com/hb/domain/Employee.hbm.xml"**/> |

至此，一对多的单向配置就已经完成了。

现在我们进行正常的保存操作，先测试无关联的部门和员工的数据的添加：

|  |
| --- |
| **public static void** main(String[] args) {  Configuration cfg = **new** Configuration();  SessionFactory sessionFactory = cfg.configure().buildSessionFactory();  Session session = sessionFactory.openSession();  Transaction transaction = session.beginTransaction();  *// 下面创建无关联的Dept和Employee* Dept dept = **new** Dept();  dept.setName(**"开发部"**);  Employee employee = **new** Employee();  employee.setName(**"王五"**);  *// 分别进行保存* session.save(dept);  session.save(employee);  transaction.commit();  session.close();  sessionFactory.close(); } |

初次运行程序时，Hibernate会自动创建表。查看数据库，发现已经有了dept和employee两张表，表结构正确，外键关系正确。并且，dept表中有了“开发部”的记录数据，employee表中有了“王五”的记录数据，此时“王五”的dept外键值为NULL，因为此时没有维护任何关系（外键字段默认是可以为NULL的）。

如果想要维护部门与员工之间的关系，使deptId中有正确的值，只要通过dept对象的setEmployeeSet方法进行关系的维护即可，例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 下面创建无关联的Dept和Employee* Dept dept = **new** Dept(); dept.setName(**"人力资源部"**); Employee employee = **new** Employee(); employee.setName(**"张三"**); dept.getEmployeeSet().add(employee); // 维护关系 *// 分别进行保存* session.save(dept); session.save(employee); transaction.commit(); session.close(); |

只是加了“dept.getEmployeeSet().add(employee);”这样一行，运行后数据库中deptId外键就会被正确地维护。

## 2.2 关于cascade

cascade是级联操作的意思。可以在上述“一”的一方（即dept）的映射文件的set节点中，通过cascade属性设置级联操作。cascade的默认值是none，可以设置为“save-update”、“delete”和“all”。

默认值null表示无级联操作；

save-update表示级联保存和更新，即当保存或更新本对象时，其关联的对象也会进行保存或更新的操作。

例如，先在dept.hbm.xml中设置级联操作：

|  |
| --- |
| <**set name="employeeSet" table="employee" cascade="save-update"**>  ...... </**set**> |

这样，通过Dept维护与Employee的关系后，直接保存Dept对象即可，而不需要保存Employee对象：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 下面创建无关联的Dept和Employee* Dept dept = **new** Dept(); dept.setName(**"系统部"**); Employee employee = **new** Employee(); employee.setName(**"小李"**); dept.getEmployeeSet().add(employee); *// 维护关系 // 仅保存dept对象即可* session.save(dept); transaction.commit(); session.close(); |

执行后数据库中既有系统部，也有小李的记录，且有外键数据。这证明了级联保存在起作用。调用了save方法后，dept变为持久化类，由于设置了级联保存和更新，employee是持久化类的关联对象，而session缓存也包括了持久化类的关联对象，因此提交事务后，它们的数据和关系被保存到数据库中。这里，session.save(dept)称为显式地保存dept对象（而student对象的保存是隐式的）。

如果不设置cascade为save-update，则必须要用save方法保存employee（和以前一样），否则程序运行会出现错误：

|  |
| --- |
| Error during managed flush [org.hibernate.TransientObjectException: object references an unsaved transient instance - save the transient instance beforeQuery flushing: com.hb.domain.Employee] |

原因就是“在flushing之前保存一个临时状态的对象”。因为不设置级联的话，employee对象不会被加入到session缓存中（即成为持久化对象）。

上述是在保存部门时级联保存员工。同样的道理，设置了cascade为save-update后，在更新部门时，如果要更新和保存员工，只要维护好关系即可，连dept对象的update方法都不需要调用，直接提交事务即可将数据保存到数据库中。

为什么呢？这是因为更新部门之前用get方法获得了dept对象，此时dept对象是持久化对象，这时通过dept对象维护employee关系，对应的关联对象也会被放入一级缓存中。因此正如我们之前所说，只要是持久化对象，直接提交事务即可保存到数据库，而无需调用update等方法。

例如级联更新员工：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 根据主键查询获得dept* Dept dept = session.get(Dept.**class**, 1); *// 遍历Employee* **for** (Employee e : dept.getEmployeeSet()) { *// 其实会把每个Employee对象放入session缓存中  // 把员工的姓名改为赵六* e.setName(**"赵六"**); } *// 最后直接提交事务即可。* transaction.commit(); session.close(); |

因此session.flush方法其实也会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。同样，如果在没有改变数据的情况下两次调用get方法获得dept数据，其实只会查询一次。

再比如，在更新部门的时候添加员工：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 根据主键查询获得dept* Dept dept = session.get(Dept.**class**, 1); Employee employee = **new** Employee(); employee.setName(**"小王"**); dept.getEmployeeSet().add(employee); *// 同样，也不用调用update方法，直接提交事务即可。* transaction.commit(); session.close(); |

注意的是这里不能直接dept.setEmployeeSet()，否则会把员工数据覆盖了。

## 2.3 关于inverse

上述的例子中，在dept中设置了dept与employee的关系，那么最终数据库中employee表的deptId会成功保存。实际上，deptId有值的原因不是因为设置了级联操作，而是因为一个叫做“inverse”的属性。

同样在“一”的一方（即dept）的映射文件的set节点中，通过inverse属性可以设置是否由本方维护外键关系。inverse的属性默认为false，表示关系由dept进行维护，如果设置为true，表示dept不进行关系的维护。

因此，如果设置了inverse为true，执行2.2节中的代码时，数据能够保存，但并不会维护外键关系，即deptId为NULL。相反，inverse默认为false时，我们观察Hibernate发送的SQL会发现，执行完查询、更新或保存的SQL后，Hibernate总是在最后再发送一条这样的update语句：

|  |
| --- |
| update employee set deptId=? where eid=? |

其实这条语句就是更新外键关系的，这是inverse为false起的作用。因此后面会说，在一对多关系中，利用“一”方维护关系是效率不高的，因为每次维护都会发出update语句来更新外键的值。而cascade设置为update-save的作用就是当通过dept对象添加或修改关联的员工时，会执行insert或update方法保存员工数据（这时保存时设置的员工deptId属性为NULL，最后通过上述的update语句设置好deptId）。

也就是说，cascade控制关联的对象是否进行级联操作；而inverse控制是否维护外键关系。

下面通过例子演示一些操作。

现有数据：

dept：

|  |  |
| --- | --- |
| id | name |
| 1 | 人事部 |
| 2 | 开发部 |

employee：

|  |  |  |
| --- | --- | --- |
| eid | name | deptId |
| 1 | 张三 | 1 |
| 2 | 李四 | 1 |
| 3 | 王五 | 1 |

下面的每个例子都在该数据基础上重新操作。

例1：将1号员工“张三”调整到“开发部”。

基于“面向对象”的操作方式，按步骤则书写以下代码：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 1. 找到人事部* Dept resourceDept = session.get(Dept.**class**, 1); *// 2. 找到张三* Employee zs = session.get(Employee.**class**, 1); *// 3. 将张三从人事部移除（注意这样的操作只是解除关系，并不会移除张三这个记录）* resourceDept.getEmployeeSet().remove(zs); *// 4. 找到开发部* Dept developDept = session.get(Dept.**class**, 2); *// 5. 将张三添加到开发部* developDept.getEmployeeSet().add(zs);  transaction.commit(); session.close(); |

上面的代码能成功更改张三的deptId为2。从上面的过程中也可看出，Hibernate不适合大数据，因为通过这样面向对象的方式操作数据还是比较繁琐的，如果直接使用JDBC发送SQL，我们只要更改张三的deptId为2即可。

但上述Hibernate代码却会执行7条SQL：

|  |
| --- |
| select dept0\_.id as id1\_0\_0\_, dept0\_.name as name2\_0\_0\_ from dept dept0\_ where dept0\_.id=?  select employee0\_.eid as eid1\_1\_0\_, employee0\_.name as name2\_1\_0\_ from employee employee0\_ where employee0\_.eid=?  select employeese0\_.deptId as deptId3\_1\_0\_, employeese0\_.eid as eid1\_1\_0\_, employeese0\_.eid as eid1\_1\_1\_, employeese0\_.name as name2\_1\_1\_ from employee employeese0\_ where employeese0\_.deptId=?  select dept0\_.id as id1\_0\_0\_, dept0\_.name as name2\_0\_0\_ from dept dept0\_ where dept0\_.id=?  select employeese0\_.deptId as deptId3\_1\_0\_, employeese0\_.eid as eid1\_1\_0\_, employeese0\_.eid as eid1\_1\_1\_, employeese0\_.name as name2\_1\_1\_ from employee employeese0\_ where employeese0\_.deptId=?  update employee set deptId=null where deptId=? and eid=?  update employee set deptId=? where eid=? |

前5条查询SQL分别是：查询人事部、查询张三、查询人事部员工、查询开发部、查询开发部员工。

后2条SQL分别是先设置张三的deptId为NULL（先解除关系），再更新deptId为2（设置关系）。

实际上这些操作最终就是设置张三的deptId为开发部的编号2，因此代码可以直接写成这样：

|  |
| --- |
| *// 1. 找到张三* Employee zs = session.get(Employee.**class**, 1); *// 2. 找到开发部* Dept developDept = session.get(Dept.**class**, 2); *// 3. 将张三添加到开发部* developDept.getEmployeeSet().add(zs); |

运行程序也会成功，即直接将张三添加到开发部。为什么不用解除之前的关系呢？其实很好理解，因为一对多的关系，一个员工只能属于一个部门，因此只要新建立了关系，原来的关系自动解除。并且此时的Hibernate执行SQL条数只有4条：

|  |
| --- |
| select employee0\_.eid as eid1\_1\_0\_, employee0\_.name as name2\_1\_0\_ from employee employee0\_ where employee0\_.eid=?  select dept0\_.id as id1\_0\_0\_, dept0\_.name as name2\_0\_0\_ from dept dept0\_ where dept0\_.id=?  select employeese0\_.deptId as deptId3\_1\_0\_, employeese0\_.eid as eid1\_1\_0\_, employeese0\_.eid as eid1\_1\_1\_, employeese0\_.name as name2\_1\_1\_ from employee employeese0\_ where employeese0\_.deptId=?  update employee set deptId=? where eid=? |

最后直接使用update更改了deptId，而不是先变成NULL再设置，这样做效率较高，推荐使用。

例2：解除人事部与所有员工的关系。

这里只要设置dept.setEmployeeSet(null)：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 1. 找到人事部* Dept resourceDept = session.get(Dept.**class**, 1); *// 2. 设置员工集合为null即可* resourceDept.setEmployeeSet(**null**);  transaction.commit(); session.close(); |

注意不要这样写：

|  |
| --- |
| resourceDept.getEmployeeSet().removeAll(resourceDept.getEmployeeSet()); |

这样的效率会较低。

## 2.4 删除操作

还是以2.3节中的例子数据作为基础做删除操作。先设置cascade为save-update和inverse为false。

例1：删除人事部：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 查到后删除* Dept resourceDept = session.get(Dept.**class**, 1); session.delete(resourceDept);  transaction.commit(); session.close(); |

程序执行成功，人事部数据被删除了，且员工中相关的deptId变为了NULL。这是因为由于inverse为false，则关联会被自动解除，因此设置成了NULL，但是不会删除员工数据。

但是如果设置为inverse为true，此时关联关系不会被自动解除，则上述的代码会出现错误:

|  |
| --- |
| Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Cannot delete or update a parent row: a foreign key constraint fails (`mytest`.`employee`, CONSTRAINT `FKkuqqg198bwpd26abp41s6ppf0` FOREIGN KEY (`deptId`) REFERENCES `dept` (`id`)) |

就是出现了外键约束错误，因为直接删除部门而不管dept表中的deptId的话，就可能出现外键约束错误。此时，就算先调用dept.setEmployeeSet(null)解除关系也不行，因为inverse为true，Hibernate根本无法发送更新deptId的SQL（inverse就是用来控制这个的）。

另外，如果想要删除部门时，级联删除部门下的员工，则此时可以将cascade属性设置成delete，delete就表示设置成级联删除。也可以设置成all，表示所有操作都进行级联，包括保存、更新和删除等。

这样将cascade设置成all的话，上述代码执行后，与部门相关的员工也会被删除。这种情况下，inverse是不是true都是一样的效果了，也不会报错，因为会级联删掉关联对象，不会有外键约束错误。

这里需要注意下面的一个问题。例如有如下代码，功能是通过dept找到关联的employee对象，再把employee对象删除（cascade属性设置为save-update）：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 1.查到dept对象* Dept dept = session.get(Dept.**class**, 2); **for** (Employee e : dept.getEmployeeSet()) {  session.delete(e); *// 删除关联对象* }  transaction.commit(); session.close(); |

会出现下述错误：

|  |
| --- |
| javax.persistence.EntityNotFoundException: deleted object would be re-saved by cascade (remove deleted object from associations): [com.hb.domain.Employee#1] |

出现这种情况的原因是：在有级联操作关系的情况下，hibernate不允许删除从对象中提取出来的关联对象。因为cascade是save-update，因此dept和employee有级联关系，这时不允许删除得到的关联对象employee。但如果开发者单独查询出employee再删除，是可以的。

如果设置cascade="none"，即无级联操作，上述代码也能成功删除employee。如果cascade就需要设置成save-update，如何删除提取出来的关联对象呢？因为实际中是有这种需求的。那么就要先通过dept解除它与empoyee的关系：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  Dept dept = session.get(Dept.**class**, 2); **for** (Employee e : dept.getEmployeeSet()) {  dept.getEmployeeSet().remove(e); *// 则需要先解除关系* session.delete(e); *// 删除关联对象* }  transaction.commit(); session.close(); |

## 2.2 一对多的双向映射

还以上述的部门与员工为例，只是此时实现的是一对多双向映射，即再能实现由员工能关联到部门即可。只要做如下改动：

（1）Employee类中要有dept的关联字段：

|  |
| --- |
| **public class** Employee **implements** Serializable {  **private** Integer **eid**; *// 编号* **private** String **name**; *// 姓名* **private** Dept **dept**; *// 部门   // 自行写getter/setter/toString()等方法* } |

（2）在Employee.hbm.xml中进行“多对一”的配置（使用many-to-one标签）：

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.hb.domain"**>  <**class name="Employee" table="employee"**>  *<!-- 主键 -->* <**id name="eid" column="eid"**>  <**generator class="increment"** />  </**id**>  *<!-- 员工姓名 -->* <**property name="name" column="name" length="20"** />  *<!--多对一外键配置 -->  <!--  name：表示本对象对应的属性  class：表示该属性对应的类型  column：指明本表中的外键字段  -->* <**many-to-one name="dept" class="Dept" column="deptId"**></**many-to-one**>  </**class**> </**hibernate-mapping**> |

即在一对多关联映射中，一的一方使用set标签，多的一方使用many-to-one标签，目的都是使得某一方通过指定的外键能关联到另一方。但最终由Hibernate生成的表的结构和单向关联是一样的。

双向关联就意味着双方都可维护关系。例如此时要同时添加员工和对应的部门，则可由员工进行关系的维护，代码为：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 添加员工“赵六”和对应的“研发部”部门 // 1. 创建员工和部门对象* Employee zl = **new** Employee(); zl.setName(**"赵六"**); Dept dept = **new** Dept(); dept.setName(**"研发部"**); *// 2. 通过员工维护与部门的关系* zl.setDept(dept); *// 3. 进行保存* session.save(zl);  transaction.commit(); session.close(); |

运行上述代码会报错：

|  |
| --- |
| TransientObjectException: object references an unsaved transient instance - save the transient instance beforeQuery flushing: com.hb.domain.Dept |

这是因为代码只保存了“zl”，而没有保存“dept”，加上“session.save(dept)”即可。当然，这个问题也能通过对Employee.hbm.xml中设置级联操作来解决。可在many-to-one中设置cascade为save-update，说明操作employee也会级联添加和更新dept，这样也解决了问题（即设置了save-update就可隐式保存dept了）。

另外我们通过执行的SQL会发现，进行级联操作，并通过多的一方（即员工）来维护关系时，最后并不会产生额外的更新SQL来单独地设置deptId外键，这是因为在添加或更新员工时就直接设置好了deptId。因此在一对多关系中，一般推荐使用多的一方维护关系，避免不需要的update语句。而且，由于外键关系（deptId）本身就会在保存或更新员工时进行维护，因此员工中不存在不维护deptId的说法，所以many-to-one中也没有提供inverse的设置。

下面再举两个一对多双向关联通过多的一方维护关系的例子。数据还是使用2.3节的数据。

例1：将1号员工“张三”调整到“开发部”。

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 直接拿到员工和部门，通过员工维护关系* Employee zs = session.get(Employee.**class**, 1); Dept developDept = session.get(Dept.**class**, 2); zs.setDept(developDept); *// 无需update，直接提交事务即可* transaction.commit(); session.close(); |

例2：把学生从班级中移除：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction();  *// 直接拿到员工和部门，通过员工维护关系* Employee zs = session.get(Employee.**class**, 1); zs.setDept(**null**); *// 设置为null即可 // 无需update，直接提交事务即可* transaction.commit(); session.close(); |

用法和之前是类似的，注意cascade设置即可。

## 2.2 关于使用Set集合

我们在配置一对多（包括下面的多对多）关系时，使用的是Set集合，Set集合是无序的，但是我们可以在“set”标签中配置“order-by”属性，以便对集合进行排序，这样拿到的Set集合就是“有序”的。例如：

|  |
| --- |
| <**set name="employeeSet" table="employee" order-by="eid"**>  <**key column="deptId"** />  <**one-to-many class="Employee"** /> </**set**> |

order-by中指定的是“employee”表中的字段，如果“employee”中有age字段，还能按照年龄排序，即order-by="age"，这样从Dept中取到的Set<Employee>就是年龄从小到大排序了。默认是升序排序的，也可设置为降序排序，即设置desc，比如：order-by="age desc"。实现的原理就是通过SQL的“order by”查询的。

## 2.4 多对多映射

例子：项目（Project）与开发人员（Developer）之间的关系就是多对多关系：

一个项目，可有多个开发人员；一个开发人员，可参与多个项目。因此，项目与开发人员之间还有一张表，即“项目人员表”。

映射数据库表时，我们使用peoject、developer和relation三张表，其中relation采用复合主键。

（1）写Project和Developer两个实体类

Project：

|  |
| --- |
| **package** com.hb.domain;  **import** java.io.Serializable; **import** java.util.LinkedHashSet; **import** java.util.Set;  **public class** Project **implements** Serializable {  **private** Integer **pid**; *// 项目主键* **private** String **pname**; *// 项目名称* **private** Set<Developer> **developerSet** = **new** LinkedHashSet<>();  *// setter/getter...* } |

Developer：

|  |
| --- |
| **package** com.hb.domain;  **import** java.io.Serializable; **import** java.util.LinkedHashSet; **import** java.util.Set;  **public class** Developer **implements** Serializable {  **private** Integer **devId**; *// 开发者编号* **private** String **devName**; *// 开发者姓名* **private** Set<Project> **projectSet** = **new** LinkedHashSet<>();  *// getter/setter...* } |

两个实体类中的Set集合分别表示项目对应的人员和人员对应的项目。

（2）映射文件

Project.hbm.xml：

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.hb.domain"**>  <**class name="Project" table="project"**>  <**id name="pid"**>  <**generator class="increment"** />  </**id**>  <**property name="pname" length="20" column="pname"** />  *<!-- 多对多映射 -->  <!-- 还是用set标签指明属性和关系表。这时关系表叫relation，用table指定-->* <**set name="developerSet" table="relation"**>  *<!--key中指明本表的外键-->* <**key column="pid"** />  *<!-- 这时应该用many-to-many指明要关联的列名和类型-->* <**many-to-many column="devId" class="Developer"** />  </**set**>  </**class**> </**hibernate-mapping**> |

Developer.hbm.xml：也是用多对多，和上述“相反”。

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.hb.domain"**>  <**class name="Developer" table="developer"**>  <**id name="devId"**>  <**generator class="increment"** />  </**id**>  <**property name="devName" length="20" column="devName"** />  *<!-- 多对多映射 -->  <!-- 关系表还是relation-->* <**set name="projectSet" table="relation"**>  *<!--key中指明本表的外键-->* <**key column="devId"** />  *<!-- many-to-many指明要关联的列名和类型-->* <**many-to-many column="pid" class="Project"**/>  </**set**>  </**class**> </**hibernate-mapping**> |

（3）在主配置文件中加载上述的映射文件。

这样，多对多的映射关系就配置好了。下面通过案例来说明多对多的操作。

首先，在每个映射文件的set标签中设置好cascade为save-update级联操作，否则下面保存或更新时，无法进行隐式地保存和更新，在开发中要注意这个区别。

例1：保存一个“OA系统”项目和它的一个“张三”开发者，且维护它们的关系。

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 1. OA系统* Project oa = **new** Project(); oa.setPname(**"OA系统"**); *// 2. 开发者张三* Developer zs = **new** Developer(); zs.setDevName(**"张三"**); *// 3. 维护关系* oa.getDeveloperSet().add(zs); *// 通过zs维护关系也是一样的。 // 4. 保存* session.save(oa); transaction.commit();  session.close(); |

上述代码执行时，Hibernate共会发出下述语句：

|  |
| --- |
| select max(pid) from project  select max(devId) from developer  insert into project (pname, pid) values (?, ?)  insert into developer (devName, devId) values (?, ?)  insert into relation (pid, devId) values (?, ?) |

前两句SQL不用说了，就是插入前先获取主键的最大值。第三句是插入项目数据，第四句是插入开发者数据，这是因为设置了cascade为save-update，能够进行隐式保存关联数据。第五句插入关系表数据就是维护项目与开发者之间的关系，这是因为inverse默认为false，因此会根据代码维护外键关系。

例2：将一个新的开发者“李四”加入到上述的“OA系统”项目中。

解析：这时用“李四”来维护关系较好，因为“李四”只是新增与项目的关系，只要调用set设置一下即可。但如果是用“OA系统”维护关系，则使用“oa.getDeveloperSet().add()”时，还要先查出OA系统的开发者信息，这是不必要的。因此代码如下：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 1. 李四* Developer ls = **new** Developer(); ls.setDevName(**"李四"**); *// 2. 获得OA系统 项目* Project oa = session.get(Project.**class**, 1); *// 3. 通过ls维护关系* Set<Project> projects = **new** LinkedHashSet<>(); projects.add(oa); ls.setProjectSet(projects); *// 4. 保存ls* session.save(ls); transaction.commit(); session.close(); |

例3：将张三由OA系统转移到图书管理系统的开发。（自行先在数据库中创建图书管理系统的记录）。

解析：将已有的开发者进行项目的调整，根据Hibernate面向对象思想，需要先解除之前与OA的关系，再新建现在的关系。则代码为：

|  |
| --- |
| Session session = sessionFactory.openSession(); Transaction transaction = session.beginTransaction(); *// 将张三更改为图书管理系统 // 1. 拿到张三* Developer zs = session.get(Developer.**class**, 1); *// 2. 拿到OA系统* Project oa = session.get(Project.**class**, 1); *// 3. 拿到图书管理系统* Project lib = session.get(Project.**class**, 2); *// 4. 先解除与OA的关系再新建与lib的关系* zs.getProjectSet().remove(oa); zs.getProjectSet().add(lib); *// 直接提交事务即可* transaction.commit(); session.close(); |

注意：这里不能像一对多那样直接设置张三的新的关系，因为在多对多这样操作的话，只会新增张三的关系（而不是修改），多对多是允许多个关系存在的。如果单纯使用SQL操作，直接调用update语句更新关系表即可，但这里在Hibernate中无优化办法，这是无可避免的，必须先解除关系再建立关系。

上述代码中，“zs.getProjectSet().remove(oa);”不能改成“zs.setProjectSet(null)”，这样会导致张三与所有的项目都解除关系。

总结：relation就是生成的一张关系表。其实：

多对多关系的建立相当于在关系表中插入一行数据；

多对多关系的解除相当于在关系表中删除一行数据；

多对多关系的修改相当于先删除记录，再增加新的记录。

因此，多对多关系的维护使用哪一方进行维护，效率都是一样的，但要看具体的需求，例如上述的例2通过开发者一方维护关系就避免了查询语句，因此写Hibernate代码时要好好考虑执行的效率。

## 2.5 一对一映射

例如一个用户和他的身份证信息就是一对一的关系。一般是用户表只存储基本信息，然后身份证表存储身份证信息，并且身份证表有外键关联用户表。

（1）实体类：

User：

|  |
| --- |
| **package** com.zhang.entity;  **public class** User {  **private** String **userId**; *// 用户编号* **private** String **userName**; *// 用户姓名* **private** IDCard **idCard**; *// 对应的身份证* } |

IDCard：

|  |
| --- |
| **package** com.zhang.entity; **public class** IDCard {  **private** String **cardNum**; *// 身份证号（主键）* **private** String **address**; *// 地址* **private** User **user**; *// 对应的用户* } |

（2）映射文件

User.hbm.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.zhang.entity"**>  <**class name="User" table="user"**>  <**id name="userId"**>  <**generator class="assigned"** />  </**id**>  <**property name="userName" length="20" column="userName"** />  *<!-- 一对一映射，这是主表，直接写类型即可。关联在外键一方配置 -->* <**one-to-one name="idCard" class="IDCard"** />  </**class**> </**hibernate-mapping**> |

IDCard.hbm.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.zhang.entity"**>  <**class name="IDCard" table="idCard"**>  <**id name="cardNum"**>  *<!-- 主键映射。 -->* <**generator class="assigned"** />  </**id**>  <**property name="address"** />  *<!-- 一对一映射。但是用many-to-one标签，因为此标签可维护外键信息。 -->  <!-- column指明外键的列名，由于是一对一，这里用unique为true表示唯一，即外键只能出现一次。就是把一对多改造成一对一 -->* <**many-to-one name="user" class="User" column="user\_Id" unique="true"**></**many-to-one**>  </**class**> </**hibernate-mapping**> |

（3）测试。注意，由于外键关系是通过IDCard维护的，因此代码中，也要通过IDCard维护关联。即用IDCard对象设置其用户，而不能通过用户设置其IDCard信息。而获取也是单向获取（下述代码没有使用级联操作）。

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Session session = **new** Configuration().configure().buildSessionFactory().openSession();  session.beginTransaction();  User user = **new** User(**"U001"**, **"张三"**);  IDCard idCard = **new** IDCard(**"321322"**, **"江苏"**);  idCard.setUser(user);  session.save(user);  session.save(idCard);  session.getTransaction().commit();  session.close();  session.getSessionFactory().close();  } } |

# 3. Hibernate的二级缓存

## 3.1 Hibernate二级缓存的适用场景

在Hibernate中除了一级缓存，还可使用二级缓存。二级缓存的生命周期是在sessionFactory范围中，当sessionFactory关闭时二级缓存会被销毁。

Hibernate二级缓存适合存储以下类型的数据：

（1）公开的数据。一些保密性强的数据不适合在缓存中存储。

（2）数据基本上不发生变化的数据。例如，省份和城市的数据、应用功能配置的数据等，这些数据通常不会发生频繁变动，可以放入二级缓存中。但一些时刻变化的数据（例如实时股票数据）就不适合存储在二级缓存中。

## 3.2 设置Hibernate二级缓存

Hibernate并没有自己实现具体的二级缓存，一般用插件的方式给Hibernate配置具体的二级缓存实现。我们这里使用ehcache实现Hibernate的二级缓存。

在Hibernate中使用ehcache作为二级缓存的步骤如下：

（1）在Hibernate的lib目录中，再进入optional/ehcache文件夹，其中的jar包就用于在Hibernate中配置ehcache作为二级缓存的实现。因此先在项目中引入该目录下的包。

（2）在hibernate.cfg.xml主配置文件中，配置二级缓存的具体实现框架，并开启Hibernate的二级缓存（默认未开启）：

|  |
| --- |
| *<!-- 指定使用的具体的缓存框架 -->* <**property name="cache.region.factory\_class"**>org.hibernate.cache.ehcache.EhCacheRegionFactory</**property**> *<!-- 开启Hibernate二级缓存 -->* <**property name="cache.use\_second\_level\_cache"**>true</**property**> |

（3）指定哪些类需要开启二级缓存。

这有两种方式，一种是直接在主配置文件中进行配置，例如：

|  |
| --- |
| *<!-- 对Project类应用二级缓存 -->* <**class-cache class="com.hb.domain.Project" usage="read-write"**/> |

usage="read-write"含义下面讲。

还有一种就是在类的映射文件中进行配置，例如：

|  |
| --- |
| <**hibernate-mapping package="com.hb.domain"**>  <**class name="Project" table="project"**>  *<!-- cache标签用于配置本类应用二级缓存 -->* <**cache usage="read-only"** />  *<!-- 主键配置.... -->* <**id name="pid"**>  <**generator class="increment"** />  </**id**>  ......  </**class**> </**hibernate-mapping**> |

我们这里使用这个第二种方式。

## 3.3 操作二级缓存

操作二级缓存，就是研究将对象放入二级缓存和将对象从二级缓存中提取出来的方法。

设置好Hibernate后，可以通过下述方法操作二级缓存：

（1）get、list方法可以把对象放入到二级缓存中；

（2）get、iterate方法可以把对象从二级缓存中提取出来。

### 3.3.1 get等方法

下面验证一下get方法可操作二级缓存：

|  |
| --- |
| **public static void** main(String[] args) {  Configuration cfg = **new** Configuration();  SessionFactory sessionFactory = cfg.configure().buildSessionFactory();  *// 创建session* Session session = sessionFactory.openSession();  *// 拿到OA系统* Project oa = session.get(Project.**class**, 1);  System.***out***.println(oa);  *// 关闭本次session* session.close();  *// 创建新的session* Session newSession = sessionFactory.openSession();  Project newOa = newSession.get(Project.**class**, 1);  System.***out***.println(newOa);  *// 关闭新的session* newSession.close();  sessionFactory.close(); } |

上述为了验证get是操作二级缓存的，期间关闭了原来的session，创建了新的session，使一级缓存不可用，但可使用sessionFactory范围的二级缓存。执行程序发现，上述代码只会产生一条查询SQL，这证明了：

（1）调用session.get方法时，该方法不仅把查询到的对象放入了一级缓存中，还把对象放到了二级缓存中（前提是开启了二级缓存）；

（2）在调用get方法提取数据时，先从一缓存中查找，如果找不到，再从二级缓存中查找（如果开启了二级缓存），如果还是找不到，则通过数据源查询数据库来查找。

那么session.save方法会不会操作二级缓存呢？我们可以通过调用sessionFactory.getStatistics().getEntityLoadCount()方法查看二级缓存中加载的实体个数。但前提是要在Hibernate主配置文件中开启统计功能：

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

验证代码如下：

|  |
| --- |
| Session session = sessionFactory.openSession(); Project project = **new** Project(); project.setPname(**"能耗监测系统"**); session.save(project); System.***out***.println(sessionFactory.getStatistics().getEntityLoadCount()); session.close(); |

输出为0，表示save方法并不会操作二级缓存。同样，update等操作也不会操作二级缓存。

### 3.3.2 list和iterate

本节开头提到的list和iterator方法实际上是在HQL中使用的，可将查询出来的对象放入二级缓存并利用。例如：

|  |
| --- |
| Session session = sessionFactory.openSession(); List<Project> projects = session.createQuery(**"from Project"**).list(); *// 查询所有的项目* System.***out***.println(session.getStatistics().getEntityCount()); *// 检查是否放入对象到一级缓存* System.***out***.println(sessionFactory.getStatistics().getEntityLoadCount()); *// 检查是否放入对象到二级缓存* session.close(); |

上述代码（list方法）会把projects保存到一级和二级缓存中（通过输出结果可证明，这里输出为2，因为数据库中存储了两个项目）。但是，如果在list方法执行后，再调用list方法查询同样的SQL和数据（HQL最终还是会转化为SQL），那么Hibernate还是会发送SQL语句。这说明，list方法只能把HQL查询结果放入一级和二级缓存中，但是不能从一级和二级缓存中提取数据。

这时就需要使用iterator方法了。例如如下代码：

|  |
| --- |
| Session session = sessionFactory.openSession(); *// 先调用list，这样对象也会被缓存到一级和二级缓存中。* List<Project> projects = session.createQuery(**"from Project"**).list(); session.close(); *// 新的session* Session newSession = sessionFactory.openSession(); *// 某段时间，用iterator查询相同的数据* Iterator<Project> newProjects = newSession.createQuery(**"from Project"**).iterate(); *// 遍历newProjects* **while** (newProjects.hasNext()) {  Project project = newProjects.next();  System.***out***.println(project.getPname()); } newSession.close(); |

上述代码执行发出的只有两条SQL：

|  |
| --- |
| select project0\_.pid as pid1\_1\_, project0\_.pname as pname2\_1\_ from project project0\_  select project0\_.pid as col\_0\_0\_ from project project0\_ |

第一条SQL就是第一个session中调用list方法时进行的查询，而第二个SQL就是新的session中调用iterate方法时进行的查询，发现该查询只是查询了主键数据。当新的session中获取每个project数据时，Hibernate并没有发送额外的查询语句，这说明此时就利用了之前session中通过list方法设置的二级缓存数据。如果说直接使用iterate，之前没有使用过list方法缓存过数据，那么每查询一个project数据就会发出一条SQL（可自行验证）。

因此对iterate方法的查询策略总结如下：

（1）先查找该表中所有的主键；

（2）再根据主键从二级缓存中查找对象，如果有，则利用二级缓存；如果没有则根据主键再查询该表中对应的数据。

Hibernate这样进行处理还是比较恶心的，但是利用好list和iterate也是能提高效率的。例如，可以在系统启动时，通过list方法加载好系统的权限数据，并操作到二级缓存，随后不同用户登录时，再通过iterate方法查询权限数据，这样就减少了对权限数据的读取。

## 3.4 把集合设置到二级缓存

在一级缓存中，Hibernate会把关联对象一同设置到一级缓存中，即调用getDeveloperSet()等方法时，关联对象的集合数据（相关的开发者Developer数据）也会被放入一级缓存中。

但在二级缓存中，如果想要关联对象（集合）也随着调用getDeveloperSet()等方法被放入到二级缓存中，则要先在映射文件的set标签中设置cache标签：

|  |
| --- |
| <**hibernate-mapping package="com.hb.domain"**>  <**class name="Project" table="project"**>  *<!-- cache标签用于配置本类应用二级缓存 -->* <**cache usage="read-only"** /><**id name="pid"**>  <**generator class="increment"** />  </**id**>  <**property name="pname" length="20" column="pname"** />  <**set name="developerSet" table="relation" cascade="save-update"**>  *<!-- 这里要配置下cache，才能把集合设置到二级缓存 -->* <**cache usage="read-only"** />  <**key column="pid"** /><**many-to-many column="devId" class="Developer"** />  </**set**>  </**class**> </**hibernate-mapping**> |

测试代码如下：

|  |
| --- |
| Project project = session.get(Project.**class**, 1); Set<Developer> developerSet = project.getDeveloperSet(); System.***out***.println(developerSet); *// 需要用一下developerSet，否则会出现懒加载，这个后面说。* System.***out***.println(sessionFactory.getStatistics().getEntityLoadCount()); *// 统计二级缓存总共保存的实体数* System.***out***.println(sessionFactory.getStatistics().getCollectionLoadCount()); *// 统计二级缓存总共保存的集合数* |

## 3.5 二级缓存的策略

一般，二级缓存中的数据通常是很少改变的，因此之前我们在cache标签中都这样设置：

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

这里的usage为read-only表示该对象可以加载到二级缓存，但是是只读的，无法修改。如果代码中对二级缓存进行修改并且提交事务的话，则程序会出现以下异常：

|  |
| --- |
| UnsupportedOperationException: Can't write to a readonly object |

如果想要能够修改该对象的数据，则需要将usage设置为“read-write”：

|  |
| --- |
| <**cache usage="read-write"** /> |

这就是二级缓存的只读策略和读写策略。

## 3.6 允许二级缓存使用磁盘空间

有时，需要进行二级缓存的数据较多，为了避免占用大量的内存空间，可以设置将二级缓存部分数据暂存到磁盘上。这实际是ehcache框架本身提供的功能，如果单独地使用ehcache也适用这样的功能。

在“ehcache-2.10.3.jar”这个我们引入的包中，其中有一个“ehcache-failsafe.xml”文件，这是ehcache的默认配置文件。我们可以把ehcache-failsafe.xml复制一份到src目录下，并改名为“ehcache.xml”，项目启动后，ehcache缓存框架就会默认加载这个配置文件，我们基于这个文件进行修改使用即可。

下面我们简单地配置ehcache，并进行解释：

|  |
| --- |
| <**ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../config/ehcache.xsd"**>  *<!-- 设置磁盘的临时缓存文件目录 -->* <**diskStore path="p:/ehcacheTemp"**/>  *<!-- 在defaultCache标签中设置默认的缓存策略 -->  <!-- 其中属性解释：  name：缓存的名字，可取值为类的全限定名。在defaultCache中无需设置。  maxElementsInMemory：可在内存中存放的最大的缓存对象个数  maxElementsOnDisk: 可在磁盘中存放的最大的缓存对象个数  overflowToDisk：当内存中缓存达到最大值时，是否将缓存放到磁盘  eternal：缓存是否永不过期，默认为false。  如果设置为true，将忽略下面的timeToIdleSeconds和timeToLiveSeconds。使用默认false即可。  timeToIdleSeconds：对象过期之前的空闲时间（秒）。即此时间内缓存未被使用，则被销毁  timeToLiveSeconds：对象过期之前的生存时间（秒）。即缓存超过此时间，则被销毁  diskPersistent：当JVM结束时，是否持久化对象。默认为false，则程序运行结束后，临时文件被自动删除。  为了验证临时文件的存在，我们这里设置成true，这样当程序执行结束后，临时文件目录中会有相关文件信息，可以验证ehcache在生效。  diskExpiryThreadIntervalSeconds：指定用于清除过期对象的线程的轮询执行时间。  该值默认是120秒。不宜设置过小，否则容易消耗计算机资源，也不宜设置过大，否则导致过期对象不能被及时清除。按照默认的120即可。  memoryStoreEvictionPolicy：当内存缓存达到最大，有新的对象加入的时候，移除缓存中元素的策略。  默认是LRU（最近最少使用），也可以设置为FIFO（先入先出）和LFU（最少使用）的策略。  -->* <**defaultCache  maxElementsInMemory="10000"  maxElementsOnDisk="10000000"  overflowToDisk="true"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  diskPersistent="true"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"**>  </**defaultCache**>   *<!-- 下面用cache节点设置具体的命名缓存的策略  为了能看到缓存到磁盘上的效果，这里设置的少一些。  其中的属性和上面的defaultCache使用相同，只要复制过来修改即可。  -->* <**cache  name="com.hb.domain.Project"  maxElementsInMemory="1"  overflowToDisk="true"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  diskPersistent="true"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"** /> </**ehcache**> |

配置后，运行一个能操作二级缓存的代码即可看到临时文件目录中发生的变化。

ehcache还有很多配置，可以等到专门研究缓存时再看。这里还有一个小问题，就是在IDE中，编写这个XML时没有提示。这是因为我们拷贝的ehcache-failsafe.xml中，根标签中指定的xsd文件路径不正确。这里应该把“../config/ehcache.xsd”改成“http://ehcache.org/ehcache.xsd”，即：

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

在IDEA中，可能还会让你注册该URL，照做即可（或者按Alt + Enter提示操作），这样就能出现提示了。

在实际项目中，一般会使用分布式缓存来解决内存缓存不足的情况，因为内存毕竟比较快。

# 4. 查询缓存

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字段的数据都查询出来，这种查询在某些场合下（如表中字段过多）会导致效率不高。

有时，程序中进行的查询可能只查询某几个字段，这时可以使用查询缓存。查询缓存也叫数据缓存，即可以把对象的部分数据放到缓存中。

查询缓存的生命周期：只要数据放入了查询缓存中，该缓存就会一直存在，直到缓存中的数据被修改了，则该缓存的生命周期就结束了。

## 4.1 使用查询缓存

首先，在主配置文件中开启Hibernate的查询缓存（注意，也要开启二级缓存，且相关的类的映射文件需要使用cache标签来指定加入二级缓存，查询缓存依赖二级缓存）：

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

一般，使用HQL语法将数据加入到查询缓存中，因为get()等方法查询到的直接是对象，它们会被放入一级或者二级缓存中，而HQL可只查询某些字段。

例1：第一个使用查询缓存的案例。见代码：

|  |
| --- |
| **public static void** main(String[] args) **throws** InterruptedException {  Configuration cfg = **new** Configuration();  SessionFactory sessionFactory = cfg.configure().buildSessionFactory();  Session session = sessionFactory.openSession();  Query query = session.createQuery(**"from Project "**);  query.setCacheable(**true**); *// 设置query使用查询缓存* query.list(); *// 会把数据放到查询缓存中* session.close();  *// 在新的session中通过同样的方式拿到数据* Session newSession = sessionFactory.openSession();  Query newQuery = newSession.createQuery(**"from Project "**);  newQuery.setCacheable(**true**); *// 也要设置query使用查询缓存* newQuery.list();  newSession.close();  sessionFactory.close(); } |

上述代码在两个session中用同样的方法进行了HQL查询，且设置了查询缓存的使用。程序执行完发现虽然调用了两次list方法，但只执行了一次SQL查询。这证明了用list进行HQL查询的数据先被放入了查询缓存中。为什么说这使用了查询缓存呢？这是因为list不会放到一级缓存中，虽然list会放入数据到二级缓存中，但list无法从二级缓存中提取数据（需要使用iterate方法），因此排除了一级缓存和二级缓存的使用，只能是查询缓存。

再比如，查询特定字段数据放入到查询缓存中：

|  |
| --- |
| Session session = sessionFactory.openSession(); Query query = session.createQuery(**"select pname from Project "**); query.setCacheable(**true**); query.list(); session.close(); |

使用查询缓存时需要知道：能够成功使用查询缓存的前提是两个HQL是一模一样的，哪怕有一点不一样，第二次执行的HQL都不能利用之前的查询缓存！例如上面的HQL是“select pname from Project”，如果以后又有session执行“select pname from Project”这个HQL，那么这次查询是能够利用查询缓存的，但是如果这次执行的HQL是“select pname from Project where pid=1”，就不能利用查询缓存，虽然这个HQL的结果包含在了上次的查询缓存中。这是Hibernate比较“恶心”的地方。

## 4.2 Hibernate中缓存的总结

hibernate中总共有三种缓存：

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方法之前，改变的是一级缓存的对象的属性。当session.flush执行时才要跟数据库交互。但一级缓存解决不了重复查询的问题（因为查询一次后session就关闭了，一级缓存随之消失）。

二级缓存可以把较少变动的、常用的公共的数据存放进来，减少重复查询。利用get和iterate方法可以得到二级缓存中的数据。

一级缓存和二级缓存都是对象缓存。

查询缓存可以缓存数据或者对象，可以利用list方法把数据放入查询缓存中，也可利用该方法得到查询缓存中的数据。查询缓存中存放的是数据，是数据缓存。