# 1. MyBatis入门

## 1.1 传统JDBC带来的问题

（1）频繁创建和释放数据库连接资源，造成资源浪费，影响系统性能。（考虑采用数据库连接池）

（2）SQL语句硬编码在Java代码中，不利于系统维护：如果修改SQL语句，则源文件需要重新编译。

（3）设置Statement参数时，需要将参数一一对应，并且是硬编码。

（4）遍历ResultSet结果集时，也使用了硬编码。

对于第三和第四点，我们希望能够自动设置statement参数，并能将结果集自动映射成POJO对象输出。

## 1.2 Mybatis介绍

MyBatis本是apache的一个开源项目iBatis，2010年该项目迁移到了Google Code，并且改名为MyBatis。

MyBatis是一个优秀的持久层框架，它对JDBC进行了封装，使开发者只需要关注执行的SQL语句本身，而无需关注数据库连接的创建、设置Statement和检索结果集等繁琐的操作。

Mybatis通过XML（或注解）的方式配置将要执行的Statement，配置的Statement包含了带有占位符的SQL、输入和输出参数，这样Mybatis会通过输入和输出参数映射生成最终要执行的SQL，并得到结果。

说明：在Mybatis中一般不使用注解，因为如果使用注解配置SQL，那么SQL还是硬编码在Java代码中，不便维护。

## 1.3 Mybatis的架构

Mybatis的架构图如下所示：

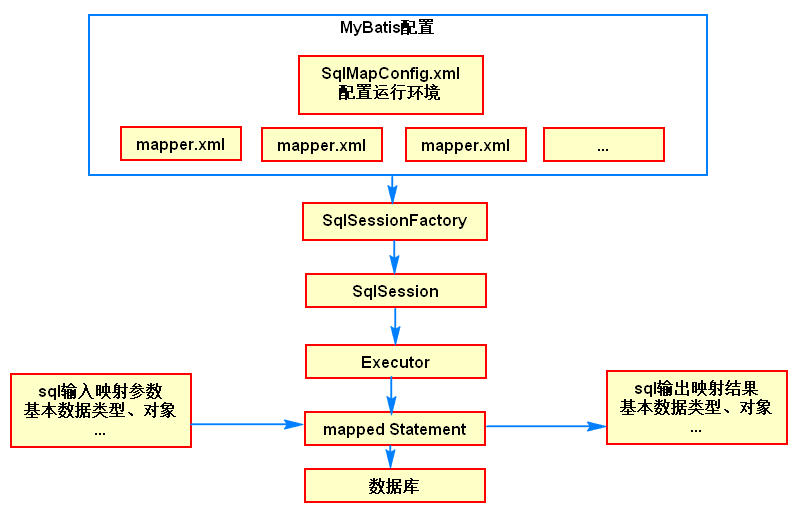


图1-1 Mybatis架构图

说明：

（1）MyBatis配置

一般用SqlMapConfig.xml作为Mybatis的全局配置文件，该文件中配置了运行环境等信息，包括数据库连接池和映射文件等。

mapper.xml文件是SQL映射文件，每个实体类都可对应一个mapper文件，文件名一般以“实体类名 + Mapper”命名。此文件需要在SqlMapConfig.xml中加载。

（2）SqlSessionFactory即会话工厂，可由Mybatis配置信息构建此对象。

（3）SqlSession即会话，由SqlSessionFactory产生。

（4）MyBatis使用Executor接口来执行Statement。Executor接口有两个实现，一个是基本实现，一个是缓存实现。

（5）Mapped Statement包装了Mybatis配置信息以及映射信息。Mapped Statement既可根据输入映射参数来设置Statement的参数，也可根据输出映射返回POJO对象。

下面进行详细地讲解。

## 1.4 第一个Mybatis示例

（1）创建Maven工程，pom.xml配置如下（Mybatis是持久层框架，使用普通Java项目即可）：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**groupId**>com.zhang</**groupId**>  <**artifactId**>mbdemo</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  <**dependencies**>  *<!-- Mybatis -->* <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>3.4.4</**version**>  <**scope**>compile</**scope**>  </**dependency**>  *<!-- MySQL的JDBC驱动 -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.42</**version**>  <**scope**>runtime</**scope**>  </**dependency**>  *<!-- 引入log4j2日志组件，以便查看运行日志 -->* <**dependency**>  <**groupId**>org.apache.logging.log4j</**groupId**>  <**artifactId**>log4j-core</**artifactId**>  <**version**>2.10.0</**version**>  </**dependency**>  </**dependencies**>  *<!-- 编码配置 -->* <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  </**properties**>  *<!-- 编译配置 -->* <**build**>  <**pluginManagement**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.6.1</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  <**encoding**>UTF-8</**encoding**>  </**configuration**>  </**plugin**>  </**plugins**>  </**pluginManagement**>  </**build**> </**project**> |

注意再在项目中引入log4j2.xml文件，内容和我们之前介绍的一样即可。这是为了记录日志用。

（2）在src/main/resources文件夹下创建SqlMapConfig.xml配置文件，内容如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"*>*** <**configuration**>  *<!--  environments节点下可配置多个environment环境，每个环境就是一种数据库配置。  配置好environment后，在environments中配置default属性值，该值就填写你想要默认用的那个数据库环境的id值。  -->* <**environments default="development"**>  *<!-- 这里id是environment 唯一标识 -->* <**environment id="development"**>  *<!-- 指明使用JDBC事务管理方式 -->* <**transactionManager type="JDBC"**/>  *<!-- 指明使用连接池方式获取连接 -->* <**dataSource type="POOLED"**>  *<!-- 以下是连接属性 -->* <**property name="driver" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://114.55.86.230:3306/mytest?useSSL=false&amp;useUnicode=true&amp;characterEncoding=UTF-8"**/>  <**property name="username" value="root"**/>  <**property name="password" value="123qwe!@#"**/>  *<!-- 这里，也能通过property配置一些数据库连接池的属性，这是Mybatis中默认集成的。  Mybatis有自己默认的设置，这里就不介绍了，实际中一般采用专门的数据库连接池，  并集成到Spring框架中。若有兴趣，自行查看Mybatis文档。 -->* </**dataSource**>  </**environment**>  *<!-- 也可再配置其他数据库连接... -->* </**environments**> </**configuration**> |

（3）编写实体类Employee，该类和数据库中的employee表对应：

|  |
| --- |
| **package** com.zhang.demo; **public class** Employee **implements** Serializable {  **private int eid**;  **private** String **ename**;   **public int** getEid() {  **return eid**;  }  **public void** setEid(**int** eid) {  **this**.**eid** = eid;  }  **public** String getEname() {  **return ename**;  }  **public void** setEname(String ename) {  **this**.**ename** = ename;  } } |

（4）可在src/main/resources下新建sqlmap文件夹，该文件夹专门用于存放映射文件。现在在此新建一个EmployeeMapper.xml映射文件（以实体类名开头命名映射文件），内容如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>*** *<!-- namespace中指明名称空间，这里先起名为test -->* <**mapper namespace="test"**>  *<!-- 该映射文件中就用来根据需求编写映射输入输出的Statement。该文件就专门处理Employee实体相关数据库操作 -->  <!-- 下面的select标签就是写了一个根据ID获取员工的映射 -->  <!-- id是映射的唯一标识，parameterType是传入SQL的参数的类型，resultType就是SQL的输出类型，就是实体类 -->  <!-- select中写了一个查询语句，#{value}就用来接收传递进来的int类型参数。 -->* <**select id="selectEmployeeById" parameterType="int" resultType="com.zhang.demo.Employee"**>  select *\** from employee where eid = #{value}  </**select**> </**mapper**> |

（5）在SqlMapConfig.xml文件中加载上述我们写好的映射文件。在SqlMapConfig.xml的根节点configuration中，再进行如下配置：

|  |
| --- |
| <**mappers**>  *<!-- 加载映射资源文件 -->* <**mapper resource="sqlmap/EmployeeMapper.xml"** /> </**mappers**> |

（6）编写程序：

|  |
| --- |
| **package** com.zhang.demo;  **import** org.apache.ibatis.io.Resources; **import** org.apache.ibatis.session.SqlSession; **import** org.apache.ibatis.session.SqlSessionFactory; **import** org.apache.ibatis.session.SqlSessionFactoryBuilder; **import** java.io.IOException; **import** java.io.InputStream;  **public class** MBTest {  **public static void** main(String[] args) **throws** IOException {  *// 加载主配置文件SqlMapConfig.xml* String resource = **"SqlMapConfig.xml"**;  InputStream inputStream = Resources.*getResourceAsStream*(resource);  *// 由配置文件构建得到SqlSessionFactory.* SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);  *// 得到sqlSessionFactory后即可关闭inputStream流* inputStream.close();  *// 再得到SqlSession。* SqlSession sqlSession = sqlSessionFactory.openSession();  *// 调用SqlSession的方法对数据库进行操作。  // 由于根据主键查询只会查出一条数据，这里选择selectOne方法。方法参数是：  // 参数1：定义的statement的id，就是“名称空间.statement的id”  // 参数2：向statement中传递的参数值。  // 可直接用对应的实体类接收返回结果。* Employee e = sqlSession.selectOne(**"test.selectEmployeeById"**, 1);  System.***out***.println(e.getEid());  System.***out***.println(e.getEname());  *// 关闭sqlSession*  sqlSession.close();  } } |

这样第一个案例就完成了，运行程序可正常执行。

说明：

（1）我们在代码中是通过文件名来加载主配置文件的，因此主配置文件的名称是可自定义的，不一定叫做“SqlMapConfig”。

（2）编写SQL时，其中的表名应该是数据库的表名，而不是对应的实体类名。SQL的查询结果能自动映射成POJO类型，前提是POJO类的属性名和数据库表的字段名一致，否则出错。

（3）在SQL中，用#{}作为占位符，当传递进来的类型是简单数据类型的时候，#{}里面的名称可以是任意名称，因此上述配置中可使用#{value}作为占位符来接收参数值，当然也可自定义名称。但为了直观，一般就使用方法的参数名作为占位符名。

（4）映射的输入参数和输出参数只能设置一个，不能设置多个，也就是说，调用sqlSession的相关方法时，只能传递一个输入参数，并返回一个输出结果，不存在多个的情况。那么如果有多个参数条件要输入来查询，也不能设置多个传递参数，而是需要用下面要讲的方法（利用实体类或者Map）。同样，有多个输出数据也需要用实体对象或者Map，下面会详细说。

## 1.5 基于示例1做常用CRUD操作

下面讲增加、删除和修改操作。因为增删改操作没有实体数据返回，因此不用写输出参数。

（1）插入操作：插入一个Employee对象到表中。

分析：既然插入的是Employee对象数据，那么Statement中传入的参数类型就是Employee，此时在占位符中，应该写Employee的属性。如：

|  |
| --- |
| *<!-- 插入使用insert标签，同样要指定statement的id。 -->* <**insert id="insertEmployee" parameterType="com.zhang.demo.Employee"**>  *<!-- insert into后面写的是表名，#{}占位符中写的是类的属性名 -->* insert into employee values(#{eid}, #{ename}) </**insert**> |

那么执行时只需要使用insert方法即可，并且需要手动提交事务：

|  |
| --- |
| Employee employee = **new** Employee(); employee.setEid(20); employee.setEname(**"王五"**); **int** row = sqlSession.insert(**"test.insertEmployee"**, employee); sqlSession.commit(); *// 必须要提交事务* |

如果主键是自增的，并且插入数据时由MySQL生成主键，那么可以设置selectKey将主键返回到实体对象中：

|  |
| --- |
| <**insert id="insertEmployee" parameterType="com.zhang.demo.Employee"**>  <**selectKey keyProperty="eid" order="AFTER" resultType="java.lang.Integer"**>  select LAST\_INSERT\_ID()  </**selectKey**>  insert into employee values(#{ename}) </**insert**> |

keyProperty指明返回到Employee实体对象中的属性名；order表示返回主键的时机，在MySQL自增中这里写AFTER；resultType表示返回的主键类型，这里是Integer类型。然后其中写查询刚刚插入的主键数据。

（2）删除操作和更新操作。这里只展示配置文件，代码和之前是类似的，只要把调用的方法改为delete或者update即可。

注意这些增删改都要提交事务，如果是查询的话，此时就不需要提交事务了。

配置示例：

|  |
| --- |
| *<!-- 按主键删除。使用delete标签。传入的参数是整型 -->* <**delete id="deleteEmployeeById" parameterType="java.lang.Integer"**>  delete from employee where eid = #{id} </**delete**> *<!-- 根据主键更新。parameterType传入的类类型，因为不仅要传递实体id，也要传递修改的数据 -->* <**update id="updateEmployeeById" parameterType="com.zhang.demo.Employee"**>  update employee set ename=#{ename} where eid=#{eid} </**update**> |

## 1.6 入门总结

MyBatis解决了单纯使用JDBC的一些弊端。这里再强调下面两点：

（1）配置statement时，需要关注的是输入映射和输出映射。输入映射通过parameterType配置，输出映射通过resultType配置。

（2）在代码中使用SqlSession对象操作数据库。需要注意，当增删改数据时，要调用commit方法手动提交事务。

## 1.7 Hibernate与Mybatis

Mybatis简单易学，开发者直接编写SQL语句，灵活度高。Mybatis适合对关系数据模型要求不高的软件开发，例如互联网软件等，因为这类软件需求变化比较频繁。但Mybatis无法做到数据库无关性，如果需要支持多种数据库，则需要定义多套SQL映射文件，工作量大。

Hibernate的对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件），如果用Hibernate开发可以节省很多代码，提高效率。但是Hibernate的学习门槛高，要精通则门槛更高，而且需要考虑如何设计O/R映射、要权衡软件性能和开发效率、需要团队有较强的Hibernate经验和能力。

# 2. namespace的作用

## 2.1 给应用程序加入DAO层

实际开发中，我们会在DAO层编写接口和实现。比如我们在com.zhang.dao中定义了IEmployeeDao接口，然后使用EmployeeDaoImpl实现IEmployeeDao接口。这里以按主键查询为例：

工具类MBHelper：

|  |
| --- |
| **package** com.zhang.dao;  **import** org.apache.ibatis.io.Resources; **import** org.apache.ibatis.session.SqlSession; **import** org.apache.ibatis.session.SqlSessionFactory; **import** org.apache.ibatis.session.SqlSessionFactoryBuilder; **import** java.io.IOException; **import** java.io.InputStream;  **public class** MBHelper {  *// 禁止创建本类实例* **private** MBHelper() {}   **private static** SqlSessionFactory *sqlSessionFactory* = **null**;  **static** {  InputStream inputStream = **null**;  **try** {  inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *sqlSessionFactory* = **new** SqlSessionFactoryBuilder().build(inputStream);  } **catch** (IOException e) {  System.***out***.println(**"加载SqlMapConfig.xml配置文件失败"**);  **throw new** ExceptionInInitializerError(e);  } **finally** {  **if** (inputStream != **null**) {  **try** {  inputStream.close(); *// 关闭inputStream* } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  }  }  }   *// 提供获得sqlSessionFactory的方法* **public static** SqlSessionFactory getSqlSessionFactory() {  **return** *sqlSessionFactory*;  }   *// 提供获得SqlSession的方法* **public static** SqlSession getSqlSession() {  **return** *sqlSessionFactory*.openSession();  } } |

IEmployeeDao接口：

|  |
| --- |
| **package** com.zhang.dao;  **import** com.zhang.demo.Employee;  **public interface** IEmployeeDao {  *// 按主键查询* **public** Employee queryEmployeeById(Integer eid); } |

EmployeeDaoImpl实现：

|  |
| --- |
| **package** com.zhang.dao;  **import** com.zhang.demo.Employee; **import** org.apache.ibatis.session.SqlSession;  **public class** EmployeeDaoImpl **implements** IEmployeeDao {  @Override  **public** Employee queryEmployeeById(Integer eid) {  SqlSession sqlSession = MBHelper.*getSqlSession*();  Employee emp =sqlSession.selectOne(**"test.selectEmployeeById"**, eid);  sqlSession.close();  **return** emp;  } } |

statement的配置还是之前的配置：

|  |
| --- |
| <**select id="selectEmployeeById" parameterType="int" resultType="com.zhang.demo.Employee"**>  select *\** from employee where eid = #{value} </**select**> |

主类测试代码：

|  |
| --- |
| **package** com.zhang.demo;  **import** com.zhang.dao.EmployeeDaoImpl; **import** com.zhang.dao.IEmployeeDao;  **public class** Demo {  **public static void** main(String[] args) {  IEmployeeDao employeeDao = **new** EmployeeDaoImpl();  Employee employee = employeeDao.queryEmployeeById(12);  System.***out***.println(employee);  } } |

## 2.2 使用namespace改进DAO

分析以上代码，我们发现，我们编写的DAO接口实现类代码是“多余”的，因为我们配置的statement已经表明了该DAO方法应该执行的操作：传入主键，通过SQL按主键查询并返回Employee对象。

我们希望：Mybatis能根据配置的statement自动选择执行的方法和返回实体对象。比如该案例希望：Mybatis会根据配置的select节点、输入类型参数和输出类型参数自动调用selectOne方法，并返回Employee对象。这样，我们就不需要写实现类了。

Mybatis是可以实现上述期望的。首先，我们需要将该实体的DAO接口（mapper.java）和该实体对应的映射文件（mapper.xml）联系起来，这就是通过mapper.xml的namespace进行关联的。我们需要将namespace配置为该实体DAO接口的完全类名。其次，需要将接口的方法和配置的statement联系起来，这要求接口的方法名和statement的id一致。

除此之外，statement中parameterType类型要和mapper接口中的方法参数类型保持一致；statement中resultType的类型要和mapper 接口中的方法返回值类型保持一致。这样，mapper接口和mapper.xml就完全匹配起来了，当需要调用DAO方法时，只需要先调用SqlSession对象的getMapper()方法获取一下Mybatis为接口生成的代理对象，再通过该代理对象调用DAO方法即可。

我们将第一章中的示例程序用接口的方式改进：

（1）EmployeeMapper.xml：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>*** *<!-- 将namespace指明为mapper接口全路径 -->* <**mapper namespace="com.zhang.dao.IEmployeeDao"**>  *<!-- 按主键查询 -->* <**select id="selectEmployeeById" parameterType="int" resultType="com.zhang.demo.Employee"**>  select *\** from employee where eid = #{value}  </**select**>  *<!-- 查询全部 -->* <**select id="selectAll" resultType="com.zhang.demo.Employee"**>  select *\** from employee  </**select**>  *<!-- 插入 -->* <**insert id="insertEmployee" parameterType="com.zhang.demo.Employee"**>  insert into employee values(#{eid}, #{ename})  </**insert**>  *<!-- 按主键删除。 -->* <**delete id="deleteEmployeeById" parameterType="java.lang.Integer"**>  delete from employee where eid = #{id}  </**delete**>  *<!-- 根据主键更新。 -->* <**update id="updateEmployeeById" parameterType="com.zhang.demo.Employee"**>  update employee set ename=#{ename} where eid=#{eid}  </**update**> </**mapper**> |

（2）IEmployeeDao.java接口：

|  |
| --- |
| **package** com.zhang.dao;  **import** com.zhang.demo.Employee; **import** java.util.List;  **public interface** IEmployeeDao {  *// 按主键查询* Employee selectEmployeeById(**int** eid);  *// 查询全部* List<Employee> selectAll();  *// 插入* **void** insertEmployee(Employee employee);  *// 按主键删除* **void** deleteEmployeeById(**int** id);  *// 按主键更新* **void** updateEmployeeById(Employee employee); } |

（3）主类调用示例：

|  |
| --- |
| **package** com.zhang.demo;  **public class** Demo {  **public static void** main(String[] args) {  *// 获得SqlSession* SqlSession sqlSession = MBHelper.*getSqlSession*();  *// 根据mapper接口获得代理对象，并且用该接口接收对象* IEmployeeDao employeeDao = sqlSession.getMapper(IEmployeeDao.**class**);  *// 以下调用dao方法* Employee employee = employeeDao.selectEmployeeById(12);  System.***out***.println(employee);  List<Employee> employees = employeeDao.selectAll();  System.***out***.println(employees);  Employee empAdd = **new** Employee(13, **"李四"**);  employeeDao.insertEmployee(empAdd);  sqlSession.commit(); *// 提交事务  // 以下方法自行测试。* sqlSession.close();  } } |

从上述示例可以看出，Mybatis会自动根据配置选择执行selectOne还是selectList（他们的resultType都是Employee）。DAO的方法只能接收一个参数，因为parameterType只能指定一个参数，如果要传入多个参数，只能封装在POJO对象中传递。

# 3. 全局配置文件SqlMapConfig.xml

在全局配置文件中可进行多项配置，一般需要按如下顺序配置：

properties（属性）

settings（全局配置参数）

typeAliases（类型别名）

typeHandlers（类型处理器）

objectFactory（对象工厂）

plugins（插件）

environments（环境集合属性对象）

mappers（映射器）

## 3.1 属性properties配置

SqlMapConfig.xml可以引用其他配置文件中的信息。例如我们可以把基本的JDBC连接信息单独配置在db.properties文件中，内容如下：

|  |
| --- |
| **jdbc.driver**=**com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://localhost:3306/mytest?useSSL=false&amp;useUnicode=true&amp;characterEncoding=UTF-8 jdbc.username**=**root jdbc.password**=**123456** |

那么SqlMapConfig中通过properties节点引入上述配置文件，这样文件中就能通过“${属性名}”方式引用db.properties的配置了：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"*>*** <**configuration**>  *<!-- 引入配置文件 -->* <**properties resource="db.properties"** />  <**environments default="development"**>  <**environment id="development"**>  <**transactionManager type="JDBC"**/>  <**dataSource type="POOLED"**>  *<!-- 在文件中使用配置文件的属性 -->* <**property name="driver" value="${jdbc.driver}"**/>  <**property name="url" value="${jdbc.url}"**/>  <**property name="username" value="${jdbc.username}"**/>  <**property name="password" value="${jdbc.password}"**/>  </**dataSource**>  </**environment**>  </**environments**>  <**mappers**>  <**mapper resource="sqlmap/EmployeeMapper.xml"** />  </**mappers**> </**configuration**> |

## 3.2 全局参数配置

全局参数配置在settings节点中配置，这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。暂时先了解一下可配置的参数，见网站“http://www.mybatis.org/mybatis-3/zh/configuration.html ”，具体需要用到的后面专门讲。

## 3.3 类型别名typeAliases

在前面的案例中，我们设置parameterType和resultType时都需要用到类的全路径。有时类的全路径较长，为了简化书写，可以使用别名功能，就是用简短的别名来代替类的全路径。

Mybatis已经为许多常见的 Java 类型内建了相应的类型别名。它们都是大小写不敏感的。详情还是参见http://www.mybatis.org/mybatis-3/zh/configuration.html 网址中的内容。

除此之外，Mybatis也允许用户自定义别名。我们在typeAliases节点中配置别名，有两种方式配置：

（1）使用typeAlias节点定义一个别名。如：

|  |
| --- |
| <**typeAliases**>  <**typeAlias type="com.zhang.demo.Employee" alias="employee"** /> </**typeAliases**> |

这样，我们在所有的映射文件mapper.xml中，就能在parameterType和resultType中直接用“employee”替代“com.zhang.demo.Employee”。

（2）使用package节点批量定义别名。如：

|  |
| --- |
| <**typeAliases**>  <**package name="com.zhang.demo"** /> </**typeAliases**> |

这样配置后，Mybatis将会扫描指定的包，将包下的所有类自动起一个别名，别名就是类名，且第一个字母小写。这样使用比较方便。

## 3.4 类型处理器typeHandlers

类型处理器的作用是处理SQL和对象之间的映射。

例如，按主键查询操作的输入和输出类型分别是int和Employee，这些是Java类型。那么类型处理器的作用就是：（1）将int类型转换输入到SQL中以支持SQL的执行；（2）从结果集中取出数据，并把数据转换为Employee类型对象。

由于Mybatis自带了大部分的类型处理器，基本上满足了需求，开发者不需要再单独定义类型处理器。这些处理器包括Integer、Double、Date等类型处理器。

## 3.5 映射器mappers

我们在mappers节点中加载映射文件的资源。之前我们直接用：

|  |
| --- |
| <**mappers**>  <**mapper resource="sqlmap/EmployeeMapper.xml"** /> </**mappers**> |

这种方式指定每个映射文件的位置（相对于类路径的资源）。

另外，还可使用mapper接口类路径进行加载，即例如：

|  |
| --- |
| <**mappers**>  <**mapper class="com.zhang.mapper.EmployeeMapper"** /> </**mappers**> |

使用此方法，要求mapper接口和mapper映射文件放在一个目录下（即一个包中），并且接口名称和mapper映射文件名称相同，这样才能通过配置的Mapper接口找到映射文件。【上述的EmployeeMapper就是以前的IEmployeeDao接口，为了和EmployeeMapper.xml同名，这里改名了，并且放入com.zhang.mapper包中，这是常见的做法。注意同时将EmployeeMapper.xml的namespace修改正确】

但是运行程序，会报错“org.apache.ibatis.binding.BindingException: Invalid bound statement (not found)”，这是什么原因呢？

我观察发现，由于我们采用的是Maven项目构建应用的，Maven项目默认不会将Java包中的配置文件放入编译目录即target文件夹中，此时，编译后的EmployeeMapper同级目录中没有“EmployeeMapper.xml”文件，因此导致程序运行失败，抛出“绑定映射文件”异常的错误。

解决办法：在pom.xml的build节点中加入如下配置：

|  |
| --- |
| <**resources**>  <**resource**>  <**directory**>src/main/java</**directory**>  <**includes**>  *<!-- 包括src目录下的xml配置文件 -->* <**include**>\*\*/\*.xml</**include**>  </**includes**>  </**resource**>  *<!-- 既然配置了resources，则还要说明包含的文件夹 -->* <**resource**>  <**directory**>src/main/resources</**directory**>  </**resource**>  <**resource**>  <**directory**>src/main/webapp</**directory**>  </**resource**> </**resources**> |

即指示Maven编译时包含src/main/java中各级目录的xml配置文件，这样，再运行程序就成功了。

在实际开发中，我们会将所有Mapper接口和配置文件都放在mapper包中，如果每个接口类都要使用mapper节点引入一下就不方便。在Mybatis中，可“批量加载接口”，即通过package节点指明mapper所在包，Mabatis会加载其中所有的接口及配置文件。例如：

|  |
| --- |
| <**mappers**>  <**package name="com.zhang.mapper"** /> </**mappers**> |

# 4. 映射文件（\*.mapper.xml）

映射文件中定义了操作数据库的SQL，映射文件是Mybatis的核心。

## 4.1 #{}与${}的区别

#{}表示占位符，在执行SQL时类似于效果“select \* from employee where eid = ?”。其中?就是占位符替代的内容。

使用#{}时不用考虑传入参数的类型，Mybatis的类型处理器会自动将Java类型映射成JDBC类型。比如，当输入参数是字符串类型时，使用#{}会自动在参数两边加上单引号。

而${}只是取出传入参数的值，会将取出的值直接拼接到SQL中。效果类似于“select \* from user where name like ‘张三%’”，其中“张三”就是${value}代替的内容。

使用${}容易导致SQL注入的问题，但有时使用${}还是比较方便的。如上面的like语句，如果一定要在like语句中使用#{}，那么只能传递参数的时候使用“张三%”这样的字符串了。

解释：

|  |
| --- |
| <**select id="query" parameterType="String" resultType="employee"**>  select *\** from user where name like '${value}%' </**select**> |

那么query方法只要传递参数“张三”即可，如果上述不用'${value}%'，而是用#{value}，那么query方法需要传递的参数为“张三%”。

## 4.2 parameterType可传入的类型

从前面的例子中我们知道，parameterType中能传入简单类型，也能传入POJO，在传入POJO时，只要通过POJO的属性名引用其中数据即可。

parameterType传入参数基本上是用来当做查询条件，为了灵活性，Mybatis支持将查询参数封装在HashMap集合中，即Mybatis的parameterType也支持HashMap类型。

例子：现在我们通过eid和ename两个条件查询合适的Employee，使用HashMap作为传入类型。

新增的接口方法：

|  |
| --- |
| *// 返回值是否是集合是自己按照实际情况决定的。 // 如果你返回的是Employee，那么Mybatis将调用selectOne方法。 // 如果返回的是List<>，那么Mybatis将会调用selectList方法。* List<Employee> selectByIdAndName(Map<String, Object> map); |

新增的配置：

|  |
| --- |
| *<!-- 这里直接使用“hashmap”，这是Mybatis为HashMap内置的别名 -->* <**select id="selectByIdAndName" parameterType="hashmap" resultType="employee"**>  select *\** from employee where eid = #{id} and ename = #{name} </**select**> |

上述的“id”和“name”将来都要作为Map的键（key）存储在map参数中。那么测试的主类代码为：

|  |
| --- |
| *// 构建map，将查询参数放入其中。 // 由于映射文件中配置的就是"id"和"name"，这里也要对应。* Map<String, Object> map = **new** HashMap<>(); map.put(**"id"**, 13); map.put(**"name"**, **"李四"**); *// 进行查询* List<Employee> employees = employeeMapper.selectByIdAndName(map); |

这样就能成功执行查询。

## 4.3 resultType可输出的类型

和parameterType一样，resultType可输出的类型有简单类型、POJO类型和HashMap类型。

（1）简单类型的示例。

使用简单类型要求查询出的结果集只能有一条、并且只有一列输出，否则无法映射。例如，查询聚合函数的结果，则输出类型就是int。

|  |
| --- |
| <**select id="selectCount" resultType="int"**>  select *count*(*\**) from employee </**select**> |

那么对应的接口方法为：

|  |
| --- |
| **int** selectCount(); |

（2）输出POJO类型和HashMap类型。

输出POJO类型只要在resultType中写上可匹配的实体类型即可（要求字段名称和属性名称对应），而输出HashMap类型，只要在resultType中写上“hashmap”类型即可，Mybatis会将结果集数据封装到Map对象中（即字段名是Map的键，字段值是Map的键）。如果查询到的是多条数据，则方法中使用List<HashMap>作为返回值类型。

例子：

|  |
| --- |
| <**select id="selectIdAndName" resultType="hashmap"**>  select eid, ename from employee </**select**> |

则对应的接口方法为：

|  |
| --- |
| *// 返回的是多条数据，因此使用List* List<HashMap<String, Object>> selectIdAndName(); |

Demo中获得数据代码示例：

|  |
| --- |
| List<HashMap<String, Object>> list = employeeMapper.selectIdAndName(); **for** (HashMap<String, Object> map : list) {  *// 再遍历Map* **for** (Map.Entry<String, Object> entry : map.entrySet()) {  System.***out***.print(entry.getKey() + **" --> "** + entry.getValue());  }  System.***out***.println(); } |

输出的结果是（示例数据）：

|  |
| --- |
| eid --> 12 ename --> 张三  eid --> 13 ename --> 李四 |

## 4.4 动态SQL

Mybatis支持动态SQL，这是Mybatis的使用重点之一。

### 4.4.1 if和where标签

动态 SQL 通常要做的事情是有条件地包含 where 子句的一部分。比如：

|  |
| --- |
| <**select id="selectWithCondition" parameterType="employee" resultType="employee"**>  select *\** from employee where  <**if test="ename != null and ename != ''"**>  and ename = #{ename}  </**if**> </**select**> |

那么当方法传入的employee参数的ename属性不为空时，就将查询if中的条件。

但是如果ename为空，if标签中的语句会忽略后，整个SQL会出错（因为最后带有where关键字），Mybatis提供了where关键字来解决这个问题。我们可把所有的查询条件放入where标签中。并且其中的每个if标签中都可带有“and”关键字，Mybatis在执行时会自动忽略第一个and关键字。如改进为：

|  |
| --- |
| <**select id="selectWithCondition" parameterType="employee" resultType="employee"**>  select *\** from employee  <**where**>  <**if test="eid != null and ename != ''"**>  and eid = #{eid}  </**if**>  <**if test="ename != null and ename != ''"**>  and ename = #{ename}  </**if**>  </**where**> </**select**> |

对应接口方法：

|  |
| --- |
| *// 查询条件封装在employee对象中* List<Employee> selectWithCondition(Employee employee); |

查询示例：

|  |
| --- |
| Employee employee = **new** Employee(); employee.setEid(13); *// 设置了eid查询参数。用户并没有填写ename查询参数 // 则只会根据eid进行查询* List<Employee> list = employeeMapper.selectWithCondition(employee); |

除此之外，Mybatis还提供了choose、when和otherwise标签来进行多重选择，详见“http://www.mybatis.org/mybatis-3/zh/dynamic-sql.html ”。

### 4.4.2 foreach标签

foreach可对一个集合进行遍历，通常是在构建带有in的条件语句的时候使用。例如我们需要查eid为12和13的员工，那么SQL可为：

|  |
| --- |
| select \* from employee where eid in (12, 13) |

如果我们在parameterType中传入了一个数组（或集合，或数组/集合是包含在对象中传递的），那么我们需要使用foreach遍历该集合来构建SQL。示例：

|  |
| --- |
| <**select id="selectMany" parameterType="int[]" resultType="employee"**>  select *\** from employee  <**where**>  <**foreach collection="array" item="item" index="index" open=" and eid in ( " close=" )" separator=","**>  #{item}  </**foreach**>  </**where**> </**select**> |

正如上述所示，使用foreach遍历数组元素。foreach中提供了很多属性：

（1）collection：要遍历的集合或数组。如果parameterType传入的是数组，那么这里直接写“array”；如果传入的是集合，那么直接写“list”；如果传入的是对象，且其中某属性的类型是数组或集合，那么这里就写该属性名。

（2）item：表示遍历的每个元素的名称，使用#{item名}引用元素。

（3）index：表示索引的名称，使用#{index名}引用索引。

（4）open：表示“前缀”；close：表示“后缀”。用于拼接在遍历的SQL外面。这里为了防止以后有多个查询参数，open中还是携带了“and”关键字，不影响执行。

（5）separator：表示分隔符。

则上述代码能很好地按照需求遍历，接口方法为：

|  |
| --- |
| List<Employee> selectMany(**int**[] array); |

使用示例：

|  |
| --- |
| **int**[] array = **new int**[2]; array[0] = 12; array[1] = 13; List<Employee> employeeList = employeeMapper.selectMany(array); |

则MyBatis最终生成的预编译SQL为：

|  |
| --- |
| select \* from employee WHERE eid in ( ? , ? ) |

如果你想用下面的方式：

|  |
| --- |
| select \* from employee where eid = 12 or eid = 13 |

也可用foreach自行编写，也很方便，可尝试。

## 4.5 SQL片段

我们可以将一些通用的SQL片段提取出来以便共用。比如可以将查询条件（不包含where标签，因为引用方可能有自己的where标签）提取出来。提取使用sql标签，并给个id作为唯一标记。例如：

|  |
| --- |
| <**sql id="employeeCondition"**>  <**if test="eid != null and ename != ''"**>  and eid = #{eid}  </**if**>  <**if test="ename != null and ename != ''"**>  and ename = #{ename}  </**if**> </**sql**> |

如果在本映射文件中引用上面的片段，则使用include标签和refid来引用：

|  |
| --- |
| <**select id="selectWithCondition" parameterType="employee" resultType="employee"**>  select *\** from employee  <**where**>  <**include refid="employeeCondition"** />  </**where**> </**select**> |

如果其他映射文件要引用SQL片段，则在refid中要加上所引用的映射文件的namespace作为前缀，即：

|  |
| --- |
| <**include refid="com.zhang.mapper.EmployeeMapper.employeeCondition"** /> |

## 4.6 resultMap

前面所做的示例中，之所以能够使用resultType映射结果集和POJO类，那是因为POJO类属性名和数据库表的字段名称相同。如果两者不同，就不能通过resultType进行映射了，而需要通过resultMap进行映射。

举例：为了使POJO类属性和结果集的字段名称不匹配，我们使用别名，例如：

|  |
| --- |
| select eid myid, ename myname from employee |

这样结果集中的字段名就是“myid”和“myname”了，其中，“myid”是“主键”。

首先在映射文件中定义一个resultMap，用于自定义结果集和POJO类之间的映射。使用resultMap节点，并用id属性给一个唯一标记，通过type属性指定要映射的POJO类：

|  |
| --- |
| *<!-- resultMap定义。这里的type也可以使用别名，这里就使用别名 -->* <**resultMap id="myResultMap" type="employee"**>  *<!-- 映射时，通过column指定要映射的结果集字段，通过property指定映射到的PIJO属性 -->  <!-- 主键使用id标签映射。如果是复合主键，则使用多个id标签 -->* <**id column="myid" property="eid"** />  *<!-- 其余键使用result映射，也可有多个result标签 -->* <**result column="myname" property="ename"** /> </**resultMap**> |

则在statement配置中，使用resultMap属性引用上面定义的myResultMap：

|  |
| --- |
| <**select id="selectAll" parameterType="String" resultMap="myResultMap"**>  select eid myid, ename myname from employee </**select**> |

至于对应的接口方法，还是用：

|  |
| --- |
| List<Employee> selectAll(); |

因为映射的结果类型还是Employee，只是中间借助resultMap进行了映射。

对于简单的操作，一般就使用resultType，而对于复杂的操作，有时还是要使用resultMap，下面介绍的关联查询就需要使用resultMap。

# 5. Mybatis的关联查询

下面我们在Mybatis中做一对一、一对多和多对对的关联查询，了解Mybatis是如何做关联映射的。

我们先设计数据库，数据有以下几张表：

（1）部门表dept：有id和name字段；

（2）员工表employee：有id、name和deptId字段；

（3）项目表project：有id和name字段；

（4）员工项目表emp\_project：有eid和pid字段。

上述中，部门和员工是一对多关系；员工和项目是多对多关系（所以有一张员工项目表），这是数据库层面的关系。

在MyBatis中，当我们进行关联查询时，是根据具体的业务来确定是“几对几”关联查询的。比如一个员工只有一个部门，则要查询“员工及其部门”时，就是“一对一”关联查询；而查询“一个员工所做的项目”时，就是“一对多”查询；而查询“所有项目对应的所有员工”时，就是“多对多”查询了。

下面附上创建表的SQL代码：

|  |
| --- |
| **create table** dept(  **id varchar**(20) **primary key comment '部门编号'**,  **name varchar**(50) **comment '部门名称'** );  **create table** employee(  **id varchar**(20) **primary key comment '员工编号'**,  **name varchar**(50) **comment '员工姓名'**,  **deptId varchar**(20) **comment '部门编号'**,  **constraint** deptId\_fk **foreign key**(**deptId**) **references** dept(**id**) );  **create table** project(  **id varchar**(20) **primary key comment '项目编号'**,  **name varchar**(50) **comment '项目名称'** );  **create table** emp\_project(  **eid varchar**(20) **comment '员工编号'**,  **pid varchar**(20) **comment '项目编号'**,  *# 复合主键* **constraint** ep\_pk **primary key**(**eid**, **pid**),  *# 员工编号外键* **constraint** emp\_fk **foreign key**(**eid**) **references** employee(**id**),  *# 项目编号外键* **constraint** project\_fk **foreign key**(**pid**) **references** project(**id**) ); |

预插入的数据：

|  |
| --- |
| *# 部门* **insert into** dept(id, **name**) **values**(**'DE01'**, **'信息部'**); **insert into** dept(id, **name**) **values**(**'DE02'**, **'开发部'**); **insert into** dept(id, **name**) **values**(**'DE03'**, **'人事部'**); *# 人员。有三个“开发者”* **insert into** employee(id, **name**, deptId) **values**(**'EM01'**, **'张三'**, **'DE01'**); **insert into** employee(id, **name**, deptId) **values**(**'EM02'**, **'李四'**, **'DE02'**); **insert into** employee(id, **name**, deptId) **values**(**'EM03'**, **'王五'**, **'DE02'**); **insert into** employee(id, **name**, deptId) **values**(**'EM04'**, **'赵六'**, **'DE02'**); *# 项目* **insert into** project(id, **name**) **values**(**'XP01'**, **'ERP系统'**); **insert into** project(id, **name**) **values**(**'XP02'**, **'手机商城'**); **insert into** project(id, **name**) **values**(**'XP03'**, **'微信服务'**); *# 项目和人员的关系* **insert into** emp\_project(eid, pid) **values**(**'EM02'**, **'XP01'**); **insert into** emp\_project(eid, pid) **values**(**'EM02'**, **'XP02'**); **insert into** emp\_project(eid, pid) **values**(**'EM03'**, **'XP01'**); **insert into** emp\_project(eid, pid) **values**(**'EM03'**, **'XP02'**); **insert into** emp\_project(eid, pid) **values**(**'EM04'**, **'XP03'**); |

现在，把数据库表、POJO类、以及项目配置自行完成。说明的是，POJO类现在只要写三个实体类（Dept、Employee和Project）和表中对应的属性即可（只写自己的属性，不写关联的属性，例如Employee中只写id和name属性，不写deptId属性）。

## 5.1 一对一查询

这里以查询“员工及其部门”为例。一对一查询一般就是查询“主表”中没有的但是唯一关联的额外信息，再比如说，查询一个订单的快递信息。

在本例中，查询的是员工信息和对应的部门名称，则对应的SQL语句可为：

|  |
| --- |
| **select** employee.*\**, dept.**name** deptName **from** employee, dept **where** employee.**deptId** = dept.**id** |

上述关联查询了employee中所有的属性和dept中的name属性，因为employee和dept都有name属性，会导致结果集无法映射，所以这里为dept的name起了别名叫“deptName”。如果不想查询employee的deptId，那么就显式地查询列。下面的示例中，虽然不会使用employee的deptId，但是我们还是会采用“employee.\*”的形式。

方法一：直接使用resultType设置输出参数。

那么此时resultType是什么类型呢？由于查询的是员工信息，干脆直接就用“employee”（别名）作为resultType，由于是一对一关系，我们只需要在Employee类中添加一个属性“deptName”即可。（如果不想为Employee增加额外的属性，那么可新建一个EmployeeInfo类继承Employee类，再新增deptName属性，那么下面的resultType就应设置为EmployeeInfo）。

接口方法：

|  |
| --- |
| List<Employee> queryEmpsWithDept(); |

SQL配置：

|  |
| --- |
| <**select id="queryEmpsWithDept" resultType="employee"**>  select employee.*\**, dept.name deptname from employee, dept  <**where**>  employee.deptId = dept.id  </**where**> </**select**> |

这样查询到的结果中就包含了部门名称，实现了“一对一”查询。这种方式比较简单，在企业中较常用。

方法二：使用resultMap定义映射。

首先在Employee中定义一个dept属性，注意这时，将该属性的类型定义为Dept，这种方式更加符合“ORM”。而之前的方法一就是你需要什么属性字段，就在原来类上新增哪个属性字段。当然，如果不想在Employee中增加属性，也可定义一个类继承Employee再添加。

这时要先在映射文件中定义resultMap，例如：

|  |
| --- |
| <**resultMap id="employeeInfoMap" type="employee"**>  *<!-- 先映射好Employee原有的字段 -->* <**id column="id" property="id"** />  <**result column="name" property="name"** />  *<!-- 再映射关联的字段，使用association标签 -->  <!-- property表示映射的一对一属性，这里就是dept，javaType表示该属性的类型，就是Dept -->* <**association property="dept" javaType="com.emp.entity.Dept"**>  *<!-- 在这其中再处理关联的映射。 -->  <!-- 这里可以在id标签中，使用查出的deptId给dept的id属性赋值。如果不写id标签，就不会赋值 -->* <**id column="deptId" property="id"** />  <**result column="deptName" property="name"** />  </**association**> </**resultMap**> |

然后只要将之前的statement配置改为使用resultMap即可，resultMap就使用上面的配置。其他的无需变动。这时再进行查询，得到的结果结构就是employee对象中包含了dept对象。

## 5.2 一对多查询

我们以查询一个部门所有的员工为例。查询的SQL示例：

|  |
| --- |
| **select** dept.*\**, employee.**id** eid, employee.**name** ename **from** dept, employee **where** dept.**id** = employee.**deptId** |

其中也使用了eid和ename作为别名。

首先要说明resultType是不支持一对多映射的。比如此时要实现一对多映射，需要在Dept类中添加List<Employee> employeeList属性，而这样的方式resultType是不支持的。因此这时，我们只能通过resultMap来进行映射。

首先在Dept类中添加List<Employee> employeeList属性，然后编写接口方法为：

|  |
| --- |
| List<Dept> queryDeptEmpList(); |

最后在映射文件中配置resultMap和statement：

|  |
| --- |
| *<!-- 配置resultMap -->* <**resultMap id="deptEmployeesMap" type="dept"**>  *<!-- 同样先映射dept原有的属性 -->* <**id column="id" property="id"** />  <**result column="name" property="name"** />  *<!-- 使用collection映射list。使用property表示映射的属性，ofType表示List的参数化类型 -->* <**collection property="employeeList" ofType="employee"**>  *<!-- 下面的column也使用别名，即结果集中显示的列名 -->* <**id column="eid" property="id"** />  <**result column="ename" property="name"** />  </**collection**> </**resultMap**> *<!-- 配置statement -->* <**select id="queryDeptEmpList" resultMap="deptEmployeesMap"**>  select dept.*\**, employee.id eid, employee.name ename from dept, employee  <**where**>  dept.id = employee.deptId  </**where**> </**select**> |

这样查询的结果就是dept对象中包含employeeList集合对象了，实现了一对多查询。我们发现，employeeList中每个employee的“信息”是不全的，那是因为我们之前给Employee类添加了dept属性，我们此时并没有映射employee中的dept属性！如果想实现这样“比较全面”的映射怎么办呢？方法就是在collection中再嵌套association，即：

|  |
| --- |
| <**resultMap id="deptEmployeesMap" type="dept"**>  <**id column="id" property="id"** />  <**result column="name" property="name"** />  <**collection property="employeeList" ofType="employee"**>  <**id column="eid" property="id"** />  <**result column="ename" property="name"** />  <**association property="dept" javaType="com.emp.entity.Dept"**>  <**id column="id" property="id"** />  <**result column="name" property="name"** />  </**association**>  </**collection**> </**resultMap**> |

发现association是和前面写的“一样”，只是因为别名不同。如果在系统中使用的别名都一样，那么我们可以通过“继承”的手段实现不写重复的配置。例如在resultMap节点中配置“extends”属性，将extends属性指向别的resultMap，那么该resultMap就不需要重复写“父resultMap”的配置。

其实上面的“嵌套”和“继承”的使用是可选的，当真正需要用到这些数据的时候再考虑。

## 5.3 多对多映射

这里以查出“所有的人员和对应的项目”为业务需求，该业务的SQL为：

|  |
| --- |
| **select** emp\_project.*\**, employee.**name** ename, project.**name** pname **from** employee, project, emp\_project **where** employee.**id** = emp\_project.**eid and** project.**id** = emp\_project.**pid** |

这是一个多对多的映射，我们还是需要定义resultMap来进行映射。不过在实际操作过程中，和前面的“一对多”很类似。

首先我们在Employee中加上一个类型为“List<Project>”的projectList属性，用于实现“项目”信息的映射。

这里，我们将返回信息的主体放在Employee中，因此接口方法是：

|  |
| --- |
| List<Employee> selectEmpProject(); |

你也可以根据业务需要的方便，将返回值类型规定为“List<Project>”，甚至也可自行创建新的类。

映射文件内容是：

|  |
| --- |
| *<!-- 配置resultMap -->* <**resultMap id="empProjectsMap" type="employee"**>  *<!-- 员工编号 -->* <**result column="eid" property="id"** />  *<!-- 员工姓名 -->* <**result column="ename" property="name"** />  *<!-- project列表还是使用collection配置 -->* <**collection property="projectList" ofType="project"**>  *<!-- 项目编号 -->* <**id column="pid" property="id"** />  *<!-- 项目名称 -->* <**result column="pname" property="name"** />  </**collection**> </**resultMap**> *<!-- 配置statement -->* <**select id="selectEmpProject" resultMap="empProjectsMap"**>  select emp\_project.*\**, employee.name ename, project.name pname from employee, project, emp\_project  <**where**>  employee.id = emp\_project.eid and project.id = emp\_project.pid  </**where**> </**select**> |

这样实现即可，和之前的“一对多”感觉是一样的。那么能否再在Project类中维护一个“List<Employee> employeeList”属性，然后在resultMap的collection节点中再嵌套一个collection实现“双向多对多”映射呢？我经过测试是不行的，因此现在感觉一对多和多对多配置和效果相同，从某个业务来讲，没有“多对多”的存在（欢迎指正和说明）。

总之，我们通过上面的几个映射关系例子，讲解了association和collection的用法，在Mybatis中使用这两个可解决大部分映射关系问题。

# 6. 延迟加载

在前面进行的关联映射中，Mybatis默认会自动加载映射的数据（这也跟我们的SQL有关，我们一次SQL就查询出了所有的关联数据）。

延迟加载就是“懒加载”，通过使用懒加载，我们希望当需要使用关联的数据时，Mybatis才会查询关联的数据，以便提高系统的性能。

首先在SqlMapConfig.xml全局配置文件中进行懒加载的配置：

|  |
| --- |
| <**settings**>  *<!-- 1. 开启全局性懒加载（延迟加载），默认该项值为false -->* <**setting name="lazyLoadingEnabled" value="true"**/>  *<!-- 2. 实现对象属性的按需加载，默认就为false，因此可省略该设置 -->* <**setting name="aggressiveLazyLoading" value="false"** /> </**settings**> |

下面我们以“一对一”的“员工及其部门”查询为例（一对一的第二种方式），进行懒加载查询。

首先查询的SQL要就要改动，一开始查询时，只需要查询“员工信息”，当我们调用员工的“getDept()”方法时，才会进行“部门”信息的查询。

同样接口方法为：

|  |
| --- |
| List<Employee> queryEmpsWithDept(); |

映射配置文件为：

|  |
| --- |
| *<!-- 先写对应的statement。这里先直接查询员工信息。resultMap的empsDept稍后写 -->* <**select id="queryEmpsWithDept" resultMap="empsDept"**>  select *\** from employee </**select**> *<!-- 再写一个statement，用于“调用getDept时进行部门的查询” -->* <**select id="queryDeptById" parameterType="string" resultType="dept"**>  select *\** from dept where id = #{id} </**select**> *<!-- 最后这里写empsDept -->* <**resultMap id="empsDept" type="employee"**>  *<!-- 其中的写法和之前相同 -->* <**id property="id" column="id"** />  <**result property="name" column="name"** />  *<!-- 只是在association节点中，需要写select和column两个属性，然后其中就不要再写映射了 -->  <!-- select表示懒加载时要执行的select，column中写关联的外键即可。注意select中要执行的statement如果在别的mapper中，则要写全namespace，还要保证其中的结果集和关联对象有正确的映射关系 -->* <**association property="dept" javaType="dept" select="queryDeptById" column="deptId"** /> </**resultMap**> |

最后我们的示例代码为：

|  |
| --- |
| List<Employee> employees= employeeMapper.queryEmpsWithDept(); System.***out***.println(employees.size()); employees.get(0).getDept(); |

我们在上述语句中加上断点，发现：

执行完第二行语句后，执行的SQL为：

|  |
| --- |
| select \* from employee |

当执行完第三行语句后，新执行的SQL为：

|  |
| --- |
| select \* from dept where id = ? |

这就说明我们成功地使用了懒加载（以上查看执行的SQL是在日志中查看的）。

# 7. 缓存

## 7.1 一级缓存

Mybatis的一级缓存作用域是同一个SqlSession。比如我们使用同一个SqlSession查询两次员工信息，第一次查询时Mybatis会从数据库中查询数据，并把数据保存在缓存中。当第二次查询员工信息时，Mybatis就会直接使用缓存中的数据。

Mybatis内部使用HashMap存储缓存数据，key为statementid与SQL的hashCode，value为查询映射出的Java对象。查询数据时，Mybatis会首先去缓存中查询，如果没有则查询数据库。这会在一定程度上减轻数据库的负担。

大家可自行测试上述说明的正确性（使用两次相同查询并查看执行的SQL）。

使用缓存可能会出现脏读，如果在两次相同的查询之前，员工表数据发生了变化，那么缓存数据还是旧的。Mybatis为了避免这种情况，会有如下操作：如果一个SqlSession中对数据进行了修改、添加、删除操作（即进行了数据的提交），那么该SQLSession的一级缓存将会被清空。

Mybatis的一级缓存一般无需开发者控制，开发者无需关心，直接“使用”即可。

## 7.2 二级缓存

### 7.2.1 使用二级缓存

Mybatis二级缓存作用域是一个mapper的namespace，二级缓存可以跨SqlSession使用，相同的namespace使用一个二级缓存结构。下面设置二级缓存。

（1）首先在核心配置文件中开启二级缓存：

|  |
| --- |
| <**settings**>  <**setting name="cacheEnabled" value="true"** /> </**settings**> |

在文档中可查到，该设置默认为true，即默认开启二级缓存，所以实际上我们可无需在核心配置文件中配置。

（2）在需要进行二级缓存的mapper.xml中加上一个cache标签，表示该mapper（即namespace）使用二级缓存：

|  |
| --- |
| <**cache** /> |

（3）在需要缓存查询的select标签中，使用useCache来指明是否使用缓存，默认为true使用缓存，无需配置。若不想某个select使用二级缓存，则需要使用false进行配置。

|  |
| --- |
| <**select id="" useCache="true"** /> |

（4）需要将查询结果的pojo类实现序列化接口Serializable。

这样，二级缓存就生效了。即使在两个SqlSession中查询相同的statement，也不会查询两次数据库，而会采用缓存。例如：

|  |
| --- |
| **public static void** main(String[] args) {  *// 得到SqlSession1。* SqlSession sqlSession1 = MBHelper.*getSqlSession*();  Employee e = sqlSession1.selectOne(**"test.selectEmployeeById"**, 1);  System.***out***.println(e.getEname());  sqlSession1.close();  *// 再用新的sqlSession2，进行同样的查询* SqlSession sqlSession2 = MBHelper.*getSqlSession*();  Employee e1 = sqlSession2.selectOne(**"test.selectEmployeeById"**, 1);  System.***out***.println(e1.getEname());  sqlSession2.close(); } |

运行上述程序发现第二次新的session查询并不会再发出SQL，而是会直接命中缓存，有这样的日志输出：

|  |
| --- |
| 21:22:49.014 DEBUG org.apache.ibatis.cache.decorators.LoggingCache 62 getObject - Cache Hit Ratio [test]: 0.5 |

“Cache Hit Ratio”是Mybatis自己得到的“命中率”，不是0代表命中了。

当然，如果在mapper.xml中将该statement的useCache属性设置为false，那么就不会使用二级缓存，命中率为0，每次都会去查询数据库。

二级缓存的原理：如果二缓存开启，首先从二级缓存查询数据，如果二级缓存有则从二级缓存中获取数据，如果二级缓存没有，从一级缓存找是否有缓存数据，如果一级缓存没有，查询数据库。

### 7.2.2 flushCache和useCache属性的使用

一般，二级缓存也会在数据进行提交后进行清除。增删改操作后都应该将二级缓存的数据清除。我们可以在statement中，用flushCache明确地指示该statement执行后刷新缓存，例如：

|  |  |
| --- | --- |
| <**update id="" flushCache="true"** /> |  |

这样执行操作后二级缓存被刷新（清除），再次查询一般就会再请求数据库。

一般：

查询时会把useCache设置成true，而把flushCache设置成false，这样下次查询相同数据时能使用缓存，这也是select标签中这些属性的默认设置。

相反，增删改时会把flushCache设置成true，因为增删改操作没有必要再进行缓存了，而应该清除缓存。这也是update等标签flushCache的默认值，而update等标签压根就没有useCache属性的设置，根本不允许缓存，没有意义。

### 7.2.3 二级缓存的应用场景

（1）一些复杂的查询或统计功能，本身查询就耗时，但实时性要求不是很高，可使用二级缓存，通过设置二级缓存的刷新时间来达到优化性能的效果，例如设置15分钟、半小时等。具体的设置自行学习。

（2）针对信息变化频率高的，需要显示最新的信息，也能使用二级缓存，但是刷新的间隔时间要短一些。

### 7.2.4 使用EhCache作为二级缓存的存储介质

Mybatis中的二级缓存也可使用EhCache等缓存框架。这时，二级缓存的策略还是由Mybatis来控制（即什么时候保存缓存，什么时候取得缓存数据等），而EhCache等只是Mybatis二级缓存使用的存储介质！

下面就介绍EhCache和Mybatis的整合使用。

（1）首先在项目中引入Mybatis对EhCache的支持：

|  |
| --- |
| <**dependency**>  <**groupId**>org.mybatis.caches</**groupId**>  <**artifactId**>mybatis-ehcache</**artifactId**>  <**version**>1.1.0</**version**> </**dependency**> |

上述Maven会自动引入Ehcache包。但是上述mybatis-ehcache还会引用slf4j这个通用的日志系统，而Mybatis中并没有使用slf4j而是配置了log4j。Mybatis会先监测项目是否依赖了slf4j，这样的话我们没配置slf4j就会导致日志系统不可用，因此还要引入slf4j和log4j的整合包：

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.logging.log4j</**groupId**>  <**artifactId**>log4j-slf4j-impl</**artifactId**>  <**version**>2.10.0</**version**> </**dependency**> |

这样就全部还会采用log4j日志系统。关于复杂的日志系统，下面会做点解释说明。

（2）在classpath目录下（即src/main/resource）新建ehcache.xml文件，内容如下：

|  |
| --- |
| <**ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../config/ehcache.xsd" updateCheck="false"**>   <**diskStore path="p:/ehcacheTemp"**/>   <**defaultCache  maxElementsInMemory="200"  maxElementsOnDisk="10000000"  overflowToDisk="true"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  diskPersistent="true"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"**>  </**defaultCache**>   <**cache  name="com.zhang.Employee"  maxElementsInMemory="1"  overflowToDisk="true"  eternal="false"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  diskPersistent="true"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU"** /> </**ehcache**> |

其实上面两步骤和Hibernate中使用ehcache一样。不再做解释了。注意一点：上面的根标签中我使用了一个“updateCheck”属性，把它设置为false，即不让Ehcache检查更新版本。EhCache默认会检查更新版本，如果不是最新版本的话，Ehcache会试图从网络上获取最新版本并在日志中显示，如果获取出错，就会有异常信息。我们不希望Ehcache自行检查，因此设置为false。

（3）在Mybatis中开启二级缓存，这时注意的是，在mapper.xml的cache标签中，要指明使用Ehcache，即这样写：

|  |
| --- |
| <**cache type="org.mybatis.caches.ehcache.EhcacheCache"**/> |

这样再启动系统，Mybatis就会采用Ehcache作为二级缓存。

# 8. 日志系统解释

Java的日志系统还是比较混乱的，有JDK的Logging，有Commons Logging、Log4J、logback等。我们之前讲过log4j，但是这么多的日志系统，如果某个项目及其依赖只使用了同一个日志系统还好，如果使用了多个日志系统，就产生了混乱，产生了冲突。

为了解决冲突的问题，又出现了slf4j通用日志系统，即提供了一个接口，使用了slf4j相当于只使用slf4j的日志接口，而针对slf4j可以配置各种各样的具体日志实现，例如Log4j等，这样就是Log4j来作为实际的日志系统了。

因此现在很多供别人使用的框架都是采用slf4j这种日志系统，或者框架内部自动判断你配置了什么日志系统（通过依赖的包来判断等），然后就使用这个系统。因为框架并不能决定开发者的系统究竟是采用哪个日志系统，因此只用通用的日志接口。

上面的mybatis-ehcache就是依赖使用了slf4j，但是Mybatis本身会先检测你依赖了哪个日志系统，然后采用某个日志系统。这样由于我们之前配置了Log4j，而mybatis-ehcache依赖了slf4j，而Mybatis会先监测slf4j，这样也会导致冲突，所以直接引入slf4j对Log4j的支持，让slf4j也采用我们的Log4j，这样就没有问题了。

总之日志系统现在看起来是比较复杂的，后续可能还会碰到类似的问题，这个以后肯定也是要再整理的。