# Mybatis入门

## 单独使用jdbc编程问题总结

### jdbc程序

**Public static void** main(String[] args) {

Connection connection = **null**;

PreparedStatement preparedStatement = **null**;

ResultSet resultSet = **null**;

**try** {

//加载数据库驱动

Class.*forName*("com.mysql.jdbc.Driver");

//通过驱动管理类获取数据库链接

connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root", "mysql");

//定义sql语句 ?表示占位符

String sql = "select \* from user where username = ?";

//获取预处理statement

preparedStatement = connection.prepareStatement(sql);

//设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值

preparedStatement.setString(1, "王五");

//向数据库发出sql执行查询，查询出结果集

resultSet = preparedStatement.executeQuery();

//遍历查询结果集

**while**(resultSet.next()){

System.*out*.println(resultSet.getString("id")+" "+resultSet.getString("username"));

}

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

//释放资源，倒着释放

**if**(resultSet!=**null**){

**try** {

resultSet.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**if**(preparedStatement!=**null**){

**try** {

preparedStatement.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**if**(connection!=**null**){

**try** {

connection.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

上边使用jdbc的原始方法（未经封装）实现了查询数据库表记录的操作。

### jdbc编程步骤：

1、加载数据库驱动

2、创建并获取数据库链接

3、创建jdbc statement对象

4、设置sql语句

5、设置sql语句中的参数(使用preparedStatement)

6、通过statement执行sql并获取结果

7、对sql执行结果进行解析处理

8释放资源(resultSet、preparedstatement、connection)

### jdbc问题总结如下：

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
2. Sql语句在代码中硬编码java代码，需要重新编译java代码，造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

设想：将sql语句配置在xml文件中，即使sql变化，不需要对java代码重新编译。

1. 使用preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不一定，可能多也可能少，修改sql还要修改代码，系统不易维护。

设想:将sql语句及占位符号和参数全部配置在xml中。

1. 对结果集解析存在硬编码（查询列名），sql变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成pojo对象解析比较方便。

设想：将查询的结果集，自动的映射成java对象。

## MyBatis介绍

MyBatis 本是[apache](http://baike.baidu.com/view/28283.htm)的一个开源项目[iBatis](http://baike.baidu.com/view/628102.htm), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis，实质上Mybatis对ibatis进行一些改进。

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。（半自动化可以实现sql语句效率的问题）

Mybatis通过xml或注解的方式将要执行的各种statement（statement、preparedStatemnt、CallableStatement）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

主要作用：

1、封装jdbc繁琐的过程代码。

2、输入映射：将preparedStatment输入的参数自动映射。

3、输出映射：查询的结果作为对象返回。

## Mybatis架构



### mybatis配置

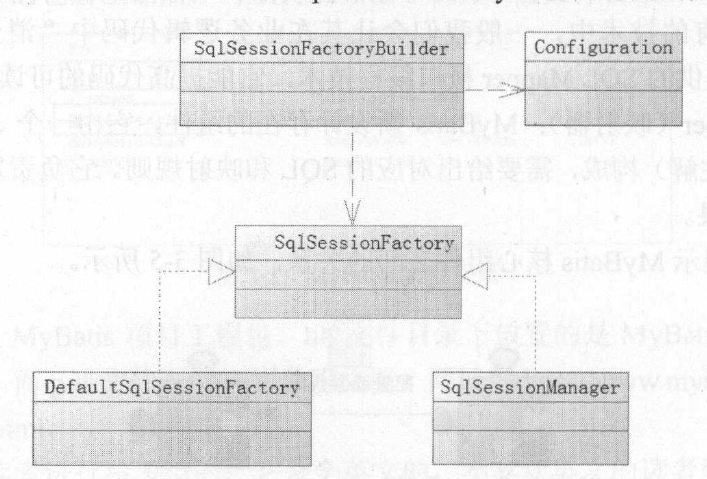
SqlMapConfig.xml:此文件作为mybatis的全局配置文件，主要是配置一些最基本的上下文参数和运行环境.

mapper.xml另一类是映射文件，它可以配置映射关系、SQL 、参数等信息。

### SqlSessionFactory

使用MyBatis 首先是使用配置或者代码去生产SqlSessionFactory，而MyBatis 提供了构造器SqlSessionFactoryBuiIder它提供了一个类。.apache.ibatis.session.Configuration 作为引导，采用的是Builder 模式。具体的分步则是在Configuration 类里面完成的。

在MyBatis中，既可以通过读取配置的XML文件的形式生成SqlSessionFactory，也可

以通过Java代码的形式去生成SqlSessionFactory。笔者强烈推荐采用XML的形式，因为代码的方式在需要修改的时候会比较麻烦。当配置了XML或者提供代码后，MyBatis会读取配置文件，通过Configuration类对象构建整个MyBatis的上下文。注意，SqlSessionFactory是一个接口，在MyBatis中它存在两个实现类：SqlSessionManager和DefaultSqlSessionFactory。一般而言，具体是由DefaultSqlSessionFactory去实现的，SqlSessionManager使用在多线程的环境中，它的具体实现依DefaultSqlSessionFactory。 

每个基于MyBatis的应用都是以一个SqlSessionFactory的实例为中心的，而SqlSessionFactory唯一的作用就是生产MyBatis的核心接口对象SqlSlession，所以它的责任

是唯一的。我们往往会采用单例模式处理它，下面讨论使用配置文件和Java代码两种形式

去生成SqlSessionFactory的方法。

首先读取mybatis-config.xml，然后通过SqlSessionFactoryBuilder的Builder方法去创建SqlSessionFactory。整个过程比较简单，而里面的步骤还是比较烦琐的，只是MyBatis采用了Builder模式为开发者隐藏了这些细节。这样一个SqlSessionFactory 就被创建出来了。

### SqlSession

在MyBatis 中，SqlSession是其核心接口。在MyBatis 中有两个实现类，DefaultSqlSession和SqlSessionManager。DefaultSqlSession是单线程使用的，而SqlSessionManager在多线程环境下使用。SqlSession的作用类似于一个JDBC中的Connection对象，代表着一个连接资源的启用。具体而言，它的作用有3个：

1、获取mapper接口。

2、发送sql给数据库。

3、控制数据库事务。

### 映射器

映射器是MyBatis中最重要、最复杂的组件，它由一个接口和对应的XML文件（或注解）组成。它可以配置以下内容：

1、描述映射规则。

2、提供SQL语句，并可以配置SQL参数类型、返回类型、缓存刷新等信息。

3、配置缓存。

4、提供动态SQL。

映射器的主要作用就是将SQL查询到的结果映射为一个POJO，或者将POJO的数据插入到数据库中，并定义一些关于缓存等的重要内容。

注意，开发只是一个接口，而不是一个实现类。初学者可能会产生一个很大的疑问，那就是接口不是不能运行吗?是的，接口不能直接运行。MyBatis 运用了动态代理技术使得接口能运行起来，入门阶段只要懂得MyBatis会为这个接口生成一个代理对象，代理对象会去处理相关的逻辑即可。

xml实现映射器。

1、定义映射器接口

|  |
| --- |
| **public** **interface** EmployeeMapper {  //添加  **int** insert(Employee emp);  } |

2、mabatis.cfg.xml配置文件

|  |
| --- |
| <mappers>  //引入xml文件，用xml方式创建映射器。  <mapper resource="cn/sxt/mapping/EmployeeMapper.xml" />  </mappers> |

|  |
| --- |
| <mapper namespace="cn.sxt.mapper.EmployeeMapper">  <insert id=*"insert"* parameterType=*"cn.sxt.domain.Employee"* useGeneratedKeys=*"true"*>  <!-- insert into t\_employee(ename,sex,salary) values('${ename}','${sex}',${salary}) -->  insert into t\_employee(ename,sex,salary) values(#{ename},#{sex},#{salary})  </insert>  </mapper> |

<mapper>元素中的属性namespace所对应的是一个接口的全限定名，于是MyBatis上下文就可以通过它找到对应的接口。

<selec>元素表明这是一条查询语句，而属性id标识了这条SQL，属性parameterType="long"说明传递给SQL的是一个long型的参数，而 resultType="role"表示返回的是一个role类型的返回值。而role是之前配置文件mybatis-config.xml配置的别名，指代的是*cn.sxt.domain.Employee*。

#{sex}：表示传递进去的参数。

注意，我们并没有配置SQL执行后和role的对应关系，它是如何映射的呢？其实这里

采用的是一种被称为自动映射的功能，MyBatis 在默认情况下提供自动映射，只要SQL返回的列名能和POJO对应起来即可。这里SQL返回的列名id和note是可以和之前定义的POJO的属性对应起来的，而表里的列role\_name通过SQL别名的改写，使其成为roleName，

也是和POJO对应起来的，所以此时MyBatis就可以把SQL查询的结果通过自动映射的功能映射成为一个POJO。

注解实现映射器

除XML方式定义映射器外，还可以采用注解方式定义映射器，它只需要一个接口就

可以通过MyBatis的注解来注入SQL。

这完全等同于XML方式创建映射器。也许你会觉得使用注解的方式比XML方式要简

单得多。如果它和XML方式同时定义时，XML方式将覆盖掉注解方式，所以MyBatis官

方推荐使用的是XML方式。

### 发送sql

SqlSession 发送SQL

用mapper接口发送sql

|  |
| --- |
| EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  Employee emp = **new** Employee("张无忌", "男", 8250);  **int** i = mapper.insert(emp); |

通过SalSession的getMapper方法来获取一个Mapper接口，就可以调用它的方法了。因为XML文件或者接口注解定义的SQL都可以通过“类的全限定名+方法名”查找，所以MyBatis会启用对应的SQL进行运行，并返回结果。

两种方式的比较

一种用SqlSesion直接发送，另外一种通过SqlSession获取Mapper 接口再发送。笔者建议采用SqlSession获取

Mapper的方式，理由如下：

·使用Mapper 接口编程可以消除SqlSession带来的功能性代码，提高可读性，而

SqlSession 发送SQL，需要一个SQLid去匹配SQL，比较晦涩难懂。使用Mapper

接口，类似roleMapper.getRole(IL)则是完全面向对象的语言，更能体现业务的逻辑。

·使用Mapper.getRole(1L)方式，IDE会提示错误和校验，而使用sqlSesion.selectOne(“getRole'”,1L)语法，只有在运行中才能知道是否会产生错误。目前使用Mapper 接口编程已成为主流，尤其在Spring中运用MyBatis时，Mapper接口的使用就更为简单。

## Mybatis的生命周期

所谓生命周期就是每一个对象应该存活的时间，比如一些对象一次用完后就要关闭，使它们被Java虚拟机（JVM）销毁，以避免继续占用资源，所以我们会根据每一个组件的作用去确定其生命周期。

### SqlSessionFactoryBuilder

SqlSesionFactoryBuilder的作用在于创建SqlSessionFactory，创建成功后，SqlSessionFactoryBuilder就失去了作用，所以它只能存在于创建SqlSessionFactory的方法中，而不要让其长期存在。

### SqlSessionFactory

SqlSessionFactory可以被认为是一个数据库连接池，它的作用是创建 SqlSession接口对象。因为MyBatis的本质就是Java对数据库的操作，所以SqlSessionFactory的生命周期存在于整个MyBatis的应用之中，所以一旦创建了SqlSessionFactory，就要长期保存它，直至不再使用MyBatis应用，所以可以认为SqlSessionFactory的生命周期就等同于MyBatis的应用周期。

### SqlSession

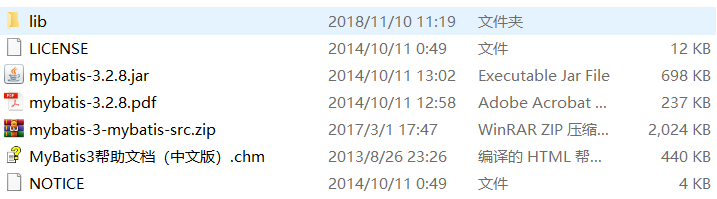
如果说SqlSessionFactory 相当于数据库连接池，那么SqlSession 就相当于一个数据库连接（Connection对象），你可以在一个事务里面执行多条SQL，然后通过它的commit、rollback等方法，提交或者回滚事务。所以它应该存活在一个业务请求中，处理完整个请求后，应该关闭这条连接，让它归还给SqlSessionFactory，否则数据库资源就很快被耗费精光，系统就会瘫痪，所以用try..catch..finally..语句来保证其正确关闭。

### Mapper

Mapper是一个接口，它由SqlSession所创建，所以它的最大生命周期至多和SqlSession保持一致，尽管它很好用，但是由于SqlSession的关闭，它的数据库连接资源也会消失，所以它的生命周期应该小于等于SqlSesion的生命周期。Mapper代表的是一个请求中的业务处理，所以它应该在一个请求中，一旦处理完了相关的业务，就应该废弃它。

## mybatis下载

mybaits的代码由github.com管理，地址：https://github.com/mybatis/mybatis-3/releases



mybatis-3.2.7.jar----mybatis的核心包

lib----mybatis的依赖包

mybatis-3.2.7.pdf----mybatis使用手册

## mybatis入门程序

### 需求

实现以下功能：

根据用户id查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户

更新用户

删除用户

### 第一步：创建java工程

创建maven工程

### 第二步：加入jar包

加入mybatis核心包、依赖包、数据驱动包。

|  |
| --- |
| <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.5</version> </dependency> <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.6</version> </dependency> <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>1.2.12</version> </dependency> <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.10</version> </dependency> |

### 第三步：log4j.properties

在classpath下创建log4j.properties如下：

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=ERROR, stdout  # MyBatis logging configuration...  log4j.logger.com.sx.mapper.EmployeeMapper=TRACE  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

mybatis默认使用log4j作为输出日志信息。

### 第四步：SqlMapConfig.xml(文件的名字不是固定的)

在classpath下创建SqlMapConfig.xml，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"*>*** *<!--mybaits的主配置文件-->* <**configuration**>  *<!--配置环境-->* <**environments default="mysql"**>  *<!--配置mysql环境-->* <**environment id="mysql"**>  *<!--配置事务类型-->* <**transactionManager type="JDBC"**></**transactionManager**>  *<!--配置数据源（连接池）-->* <**dataSource type="POOLED"**>  *<!--配置数据源（连接池）的基本信息-->* <**property name="driver" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://localhost:3306/car?characterEncoding=UTF-8"**/>  <**property name="username" value="root"**/>  <**property name="password" value="root"**/>  </**dataSource**>  </**environment**>  </**environments**>  *<!--指定映射配置文件的位置，映射配置文件指的是每个dao配置的文件-->* <**mappers**>  *<!--<mapper resource="com/daniel/dao/IUserDao.xml"/> 使用的是xml-->* <**mapper class="com.daniel.dao.IUserDao"**/> *<!--使用的是注解 annotations-->* </**mappers**> </**configuration**> |

SqlMapConfig.xml是mybatis核心配置文件，上边文件的配置内容为数据源、事务管理。

### 第五步：po类

Po类作为mybatis进行sql映射使用，po类通常与数据库表对应，User.java如下：

|  |
| --- |
| **private** Integer **id**; **private** String **username**; **private** Date **birthday**; **private** String **sex**; **private** String **address**; |

### IuserDao 接口

类路劲下:com.sx.mapper. IUserDao

|  |
| --- |
| **public interface** IUserDao {  *//查询所有* List<User> queryAll();  *//查询一个* User findById(Integer id);  *//添加用户* **void** addUser(User user);  *//更新用户* Integer updateUser(User user);  *//删除用户* Integer deleteUser(Integer id);  *//模式查询* List<User> findByLike(String name);  *//查询数量* **int** findTotal(); } |

### IuserDao.xml (mapper.xml)

类路劲下:com/sx/mapper/IuserDao.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"*>*** <**mapper namespace="com.sx.mapper.IUserDao"**>  <**select id="queryAll" resultType="com.sx.domain.User"**>  select *\** from user  </**select**>  *<!--只有一个参数的时候可以随意写占位符-->* <**select id="findById" parameterType="java.lang.Integer" resultType="com.sx.domain.User"**>  select *\** from user where id = #{id}  </**select**>  *<!--模糊查询-->* <**select id="findByLike" parameterType="java.lang.String" resultType="com.sx.domain.User"**>  select *\** from user where username like "%"#{username}"%"  </**select**>  *<!--查询全部数量-->* <**select id="findTotal" resultType="Integer"**>  select *count*(id) from user  </**select**>  <**insert id="addUser" parameterType="com.sx.domain.User"**>  *<!--配置插入成功后，返回插入数据的id-->* <**selectKey keyProperty="id" keyColumn="id" resultType="int" order="AFTER"**>  select last\_insert\_id()  </**selectKey**>  insert into user(username,address,sex,birthday)  values(#{username},#{address},#{sex},#{birthday})  </**insert**>  <**update id="updateUser" parameterType="com.sx.domain.User"**>  update user set username=#{username}, address=#{address}, sex=#{sex}, birthday=#{birthday}  where id=#{id}  </**update**>  *<!--只有一个参数的时候可以随意写占位符-->* <**delete id="deleteUser" parameterType="java.lang.Integer"** >  delete from user where id = #{id}  </**delete**> </**mapper**> |

### 映射文件：

在classpath下的sqlmap目录下创建sql映射文件Users.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">

<mapper namespace=*"test"*>

</mapper>

namespace ：命名空间，用于隔离sql语句，后面会讲另一层非常重要的作用。

在SqlMapConfig.xml中添加：

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user where id = #{id}

</select>

<!-- 自定义条件查询用户列表 -->

<select id=*"findUserByUsername"* parameterType=*"java.lang.String"*

resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user where username like '%${value}%'

</select>

#{id}：id是接入输入的采参数。如果输入的参数是简单类型，#{}里面的参数值可以是任意的。

parameterType：定义输入到sql中的映射类型，#{id}表示使用preparedstatement设置占位符号并将输入变量id传到sql。

resultType：指定sql输出结果的所映射的java对象类型，select指定resultType表示将单条记录映射成的java对象。

### 测试

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception {  *//1.读取配置文件* InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.创建SqlSessionFactory工厂，使用工厂模式，输入为配置文件内容。* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(inputStream);  *//3.使用工厂创建SqlSession* SqlSession session = factory.openSession();  *//4.使用SqlSession创建dao接口的代理对象* IUserDao userDao = session.getMapper(IUserDao.**class**);  *//5.使用代理对象的方法* List<User> users = userDao.queryAll();  **for** (User user : users) {  System.***out***.println(user);  }  *//6.释放资源* session.close();  inputStream.close(); } |

mybatis的入门案例

第一步：读取配置文件

第二步：创建SqlSessionFactory工厂

第三步：创建SqlSession

第四步：创建Dao接口的代理对象

第五步：执行dao中的方法

第六步：释放资源

mybatis使用的设计模式:工厂模式，代理模式

### 环境搭建的事项

第一个：创建IUserDao.xml 和 IUserDao.java时名称是为了和我们之前的知识保持一致。

在Mybatis中它把持久层的操作接口名称和映射文件也叫做：Mapper

所以：IUserDao 和 IUserMapper是一样的

第二个：在idea中创建目录的时候，它和包是不一样的

包在创建时：com.itheima.dao它是三级结构

目录在创建时：com.itheima.dao是一级目录

第三个：mybatis的映射配置文件位置必须和dao接口的包结构相同

第四个：映射配置文件的mapper标签namespace属性的取值必须是dao接口的全限定类名

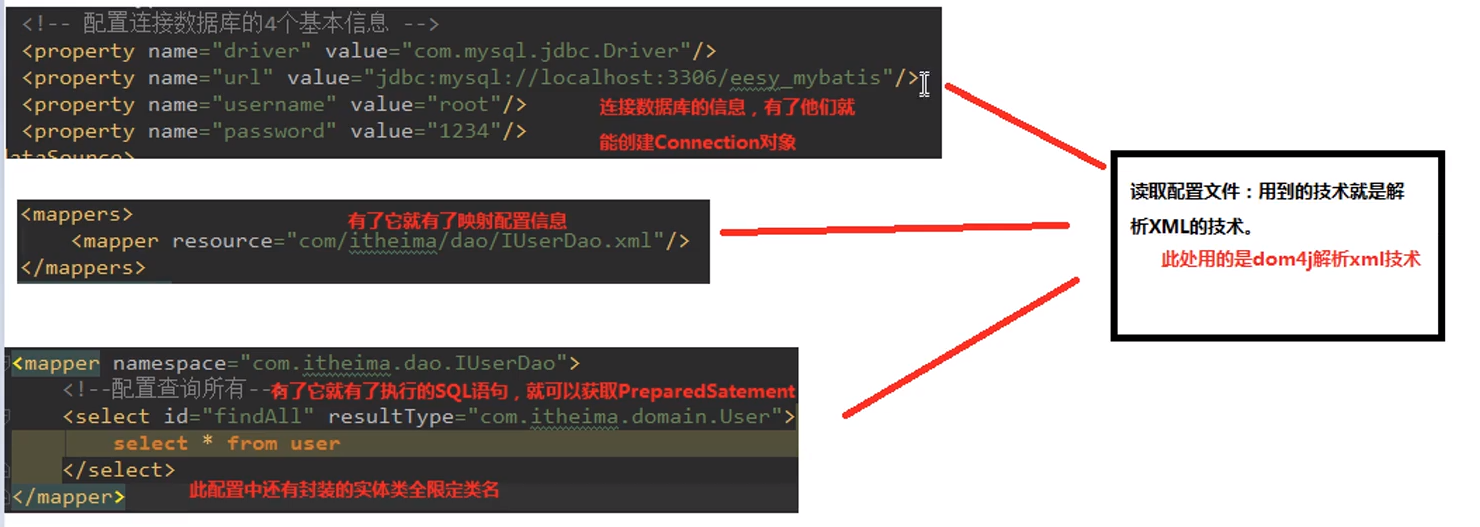
第五个：映射配置文件的操作配置（select），id属性的取值必须是dao接口的方法名

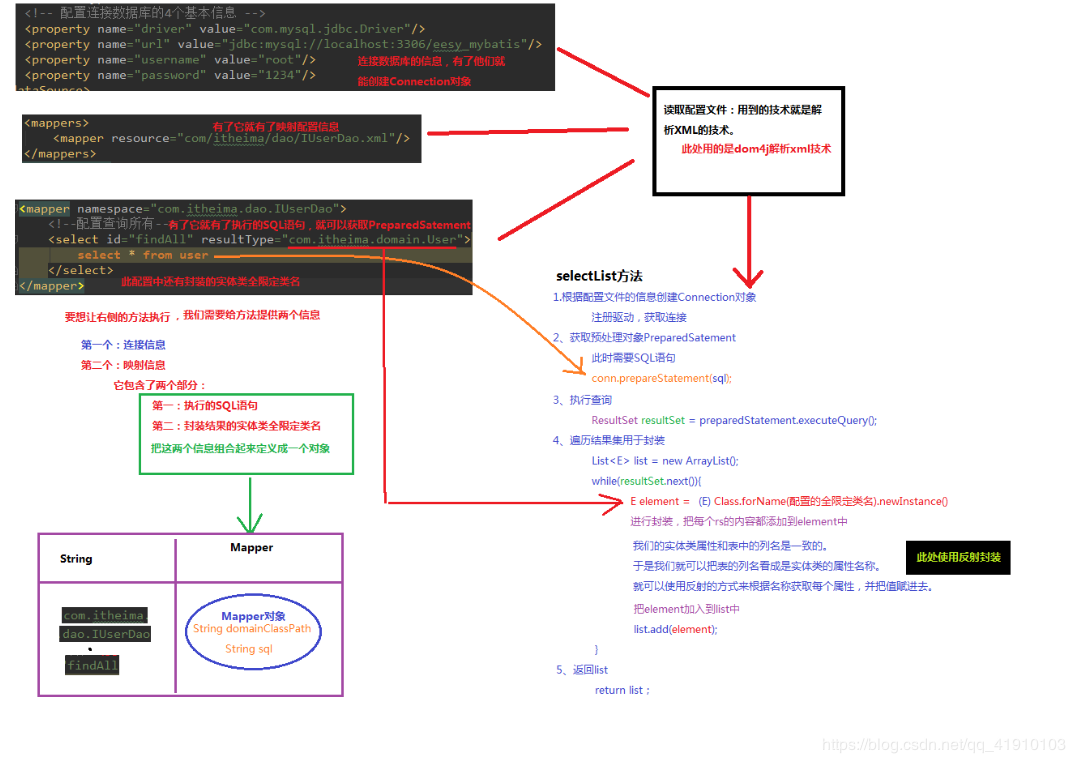
### mysql文件创建

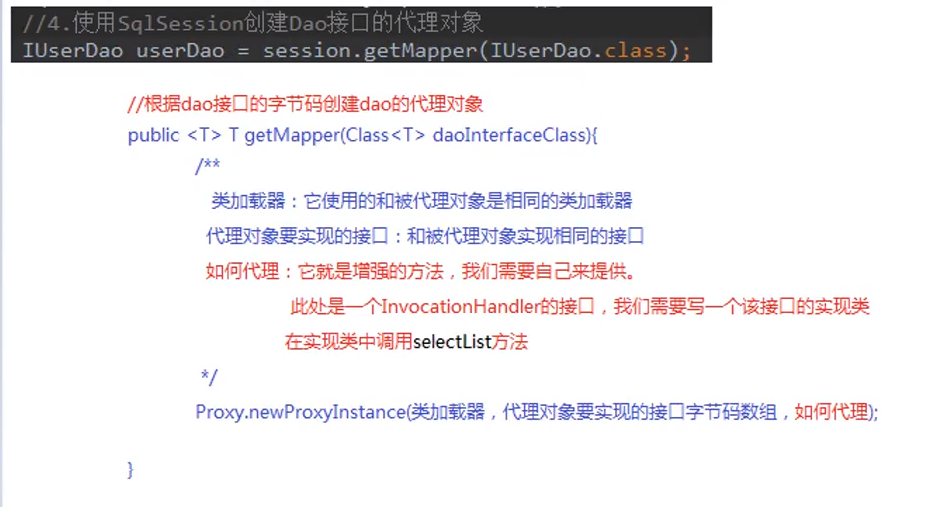
user表

|  |
| --- |
| DROP TABLE IF EXISTS `user`;  CREATE TABLE `user` (  `id` int(11) NOT NULL auto\_increment,  `username` varchar(32) NOT NULL COMMENT '用户名称',  `birthday` datetime default NULL COMMENT '生日',  `sex` char(1) default NULL COMMENT '性别',  `address` varchar(256) default NULL COMMENT '地址',  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `user`(`id`,`username`,`birthday`,`sex`,`address`) values (41,'老王','2018-02-27 17:47:08','男','北京'),(42,'小二王','2018-03-02 15:09:37','女','北京金燕龙'),(43,'小二王','2018-03-04 11:34:34','女','北京金燕龙'),(45,'传智播客','2018-03-04 12:04:06','男','北京金燕龙'),(46,'老王','2018-03-07 17:37:26','男','北京'),(48,'小马宝莉','2018-03-08 11:44:00','女','北京修正'); |

### 入门程序分析







### #{}和${}

#{}表示一个占位符号，通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换，#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

### parameterType和resultType

parameterType：指定输入参数类型，mybatis通过ognl从输入对象中获取参数值拼接在sql中。

resultType：指定输出结果类型，mybatis将sql查询结果的一行记录数据映射为resultType指定类型的对象。

### selectOne和selectList

selectOne查询一条记录，如果使用selectOne查询多条记录则抛出异常：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

selectList可以查询一条或多条记录。

## 添加

### 映射文件：

在SqlMapConfig.xml中添加：

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

//如果数据库有自增，可以不用写。

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address})

</insert>

### 测试程序：

// 添加用户信息

@Test

**public** **void** testInsert() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 添加用户信息

User user = **new User()**;

user.setUsername("张小明");

user.setAddress("河南郑州");

user.setSex("1");

user.setPrice(1999.9f);

sqlSession.insert("test.insertUser", user);

//提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

### mysql自增主键返回

mysql自增主键，执行insert之前自动生成一个自增主键。可以通过mysql函数获取刚插入记录的自增主键。

通过修改sql映射文件，可以将mysql自增主键返回：

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<!-- selectKey将主键返回，需要再返回 -->

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address});

</insert>

添加selectKey实现将主键返回

keyProperty:将返回的主键值设置到parameterType的那个属性值。

order：selectKey的执行顺序，是相对与insert语句来说，由于mysql的自增原理执行完insert语句之后才将主键生成，所以这里selectKey的执行顺序为after

resultType:返回的主键是什么类型

LAST\_INSERT\_ID():是mysql的函数，返回auto\_increment自增列新记录id值。得到刚插入记录的主键值，只使用自增主键。

注：如果没有设置selectKey，返回的是0，即int的默认值。

### Mysql使用 uuid实现主键(非自增主键的返回)

使用mysql的uuid()函数生成主键，需要修改表中id字段类型为string，长度设置为35位。

首先通过uuid()得到主键，将主键设置到user对象的id属性中，其次，在insert执行时，从user对象中取出id值

|  |
| --- |
| <insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>  <selectKey resultType=*"java.lang.String"* order=*"BEFORE"*  keyProperty=*"id"*>  select uuid()  </selectKey>  //这里要将插入id  insert into user(id,username,birthday,sex,address)  values(#{id},#{username},#{birthday},#{sex},#{address})  </insert> |

注意这里使用的order是“BEFORE”

### Oracle使用序列生成主键

首先自定义一个序列且用于生成主键，selectKey使用如下：

|  |
| --- |
| <insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>  <selectKey resultType=*"java.lang.Integer"* order=*"BEFORE"*  keyProperty=*"id"*>  SELECT 自定义序列.NEXTVAL FROM DUAL  </selectKey>  insert into user(id,username,birthday,sex,address)  values(#{id},#{username},#{birthday},#{sex},#{address})  </insert> |

注意这里使用的order是“BEFORE”

## 删除

### 映射文件：

<!-- 删除用户 -->

<delete id=*"deleteUserById"* parameterType=*"int"*>

delete from user where id=#{id}

</delete>

### 测试程序：

// 根据id删除用户

@Test

**public** **void** testDelete() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 删除用户

sqlSession.delete("test.deleteUserById",18);

// 提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

## 修改

### 映射文件

<!-- 更新用户 -->

<update id=*"updateUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address}

where id=#{id}

</update>

### 测试程序

// 更新用户信息

@Test

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

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 添加用户信息

User user = **new User()**;

user.setId(16);

user.setUsername("张小明");

user.setAddress("河南郑州");

user.setSex("1");

user.setPrice(1999.9f);

sqlSession.update("test.updateUser", user);

// 提交事务

sqlSession.commit();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (sqlSession != **null**) {

sqlSession.close();

}

}

}

## Mybatis解决jdbc编程的问题

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

1. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

1. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis自动将java对象映射至sql语句，通过statement中的parameterType定义输入参数的类型。

1. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

## 与hibernate不同

Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句，不过mybatis可以通过XML或注解方式灵活配置要运行的sql语句，并将java对象和sql语句映射生成最终执行的sql，最后将sql执行的结果再映射生成java对象。

Mybatis学习门槛低，简单易学，程序员直接编写原生态sql，可严格控制sql执行性能，灵活度高，非常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、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套sql映射文件，工作量大。

Hibernate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，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

# Dao开发方法

使用Mybatis开发Dao，通常有两个方法，即原始Dao开发方法和Mapper接口开发方法。

## 前提

### 需求

将下边的功能实现Dao：

根据用户id查询一个用户信息

根据用户名称模糊查询用户信息列表

添加用户信息

### SqlSession的使用范围

SqlSession中封装了对数据库的操作，如：查询、插入、更新、删除等。

通过SqlSessionFactory创建SqlSession，而SqlSessionFactory是通过SqlSessionFactoryBuilder进行创建。

### SqlSessionFactoryBuilder

SqlSessionFactoryBuilder用于创建SqlSessionFacoty，SqlSessionFacoty一旦创建完成就不需要SqlSessionFactoryBuilder了，因为SqlSession是通过SqlSessionFactory生产，所以可以将SqlSessionFactoryBuilder当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

### SqlSessionFactory

SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

### SqlSession

SqlSession是一个面向用户的接口， sqlSession中定义了数据库操作，默认使用DefaultSqlSession实现类。线程不安全，在sqlsesison中除了有接口的方法，这些方法除了有接口中的方法，还有数据域的属性。由于这些的存在，导致线程不安全。sqlsesison最佳应用场合为方法体呢。定义成局部变量使用

执行过程如下：

1. 加载数据源等配置信息

Environment environment = configuration.getEnvironment();

1. 创建数据库链接
2. 创建事务对象
3. 创建Executor，SqlSession所有操作都是通过Executor完成，mybatis源码如下：

**if** (ExecutorType.*BATCH* == executorType) {

executor = **new**BatchExecutor(**this**, transaction);

} **elseif** (ExecutorType.*REUSE* == executorType) {

executor = **new** ReuseExecutor(**this**, transaction);

} **else** {

executor = **new** SimpleExecutor(**this**, transaction);

}

**if** (cacheEnabled) {

executor = **new** CachingExecutor(executor, autoCommit);

}

1. SqlSession的实现类即DefaultSqlSession，此对象中对操作数据库实质上用的是Executor

结论：

每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能共享使用，它也是线程不安全的。因此最佳的范围是请求或方法范围。绝对不能将SqlSession实例的引用放在一个类的静态字段或实例字段中。

打开一个 SqlSession；使用完毕就要关闭它。通常把这个关闭操作放到 finally 块中以确保每次都能执行关闭。如下：

SqlSession session = sqlSessionFactory.openSession();

try {

// do work

} finally {

session.close();

}

## 原始Dao开发方式

原始Dao开发方法需要程序员编写Dao接口和Dao实现类。

### 映射文件（底层实现sql操作）

<?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">

<mapper namespace=*"test"*>

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"cn.itcast.mybatis.po.User"*>

select \* from user where id = #{id}

</select>

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"cn.itcast.mybatis.po.User"*>

<selectKey keyProperty=*"id"* order=*"AFTER"* resultType=*"java.lang.Integer"*>

select LAST\_INSERT\_ID()

</selectKey>

insert into user(username,birthday,sex,address)

values(#{username},#{birthday},#{sex},#{address})

</insert>

</mapper>

### Dao接口(定义方法)

**Public interface** UserDao {

**public** User getUserById(**int** id) **throws** Exception;

**public void** insertUser(User user) **throws** Exception;

}

### Dao实现类

**Public class** UserDaoImpl **implements** UserDao {

//一般是spring整合后注入SqlSessionFactory

//这例不能写Sqlsession，sqlsesison里有公有的数据，线程不安全。

**public** UserDaoImpl(SqlSessionFactory sqlSessionFactory){

**this**.setSqlSessionFactory(sqlSessionFactory);

}

**private** SqlSessionFactory sqlSessionFactory;

@Override

**public User** getUserById(**int** id) **throws** Exception {

SqlSession session = sqlSessionFactory.openSession();

User user = **null**;

**try** {

//通过sqlsession调用selectOne方法获取一条结果集

//参数1：指定定义的statement的id,参数2：指定向statement中传递的参数

user = session.selectOne("test.findUserById", 1);

System.out.println(user);

} **finally**{

session.close();

}

**return** user;

}

@Override

**Public void** insertUser(User user) **throws** Exception {

SqlSession sqlSession = sqlSessionFactory.openSession();

**try** {

sqlSession.insert("insertUser", user);

sqlSession.commit();

} **finally**{

session.close();

}

}

}

### 问题

原始Dao开发中存在以下问题：

* Dao方法体存在重复代码：通过SqlSessionFactory创建SqlSession，调用SqlSession的数据库操作方法。

设想：将这些公有的代码提取出来，减轻程序员工作量。

* 调用sqlSession的数据库操作方法需要指定statement的id，这里存在硬编码，不得于开发维护。
* 调用sqlsession方法时传入的变量，由与sqlsession方法使用泛型，即使类型传入错误。在编译阶段也不报错，不利于程序员开发。设想：

# Mapper动态代理方式

## 实现原理

Mapper接口开发方法只需要程序员编写Mapper接口（相当于Dao接口），由Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper接口实现下列规范，mybats底层能够生成mapper接口实现类的代理对象那。

也就是实现了mapper接口中的方法。

Mapper接口开发需要遵循以下规范()：

1. Mapper.xml文件中的namespace与mapper接口的类路径相同。
2. Mapper接口方法名和Mapper.xml中定义的每个statement的id相同
3. Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同
4. Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

接口名+方法名+参数类型和数量+返回值类型，能够准确的定位方法。方法里面的具体想要获取的数据在映射文件中配置。有底层代码去实现编写的sql，并执行增删改查。

## 增

### 接口

|  |
| --- |
| //添加  **int** insert(Employee emp); |

### mapper.xm

|  |
| --- |
| <insert id=*"insert"* parameterType=*"cn.sxt.domain.Employee"* useGeneratedKeys=*"true"*>  <!-- insert into t\_employee(ename,sex,salary) values('${ename}','${sex}',${salary}) -->  insert into t\_employee(ename,sex,salary) values(#{ename},#{sex},#{salary})  </insert> |

### 测试

|  |
| --- |
| **public** **void** test1() {  SqlSession session = MyBatisUtil.*getSession*();  // 获取Mapper接口的实现类对象(Mapper接口.class)  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  // EmployeeDao dao=new EmployeeDaoImpl();  // 调用Mapper接口中的方法  Employee emp = **new** Employee("张无忌", "男", 8250);  **int** i = mapper.insert(emp);  **if** (i > 0) {  System.*out*.println("添加成功");  } **else** {  System.*out*.println("添加失败");  }  session.commit();  session.close();  } |

## 修改

### 接口

|  |
| --- |
| //修改  **int** update(Employee emp); |

### mapper.xm

|  |
| --- |
| <!-- 执行Dao里修改 -->  <update id=*"update"* parameterType=*"cn.sxt.domain.Employee"*>  update t\_employee set ename=#{ename},sex=#{sex},salary=#{salary} where eid=#{eid}  </update> |

### 测试

|  |
| --- |
| @Test  **public** **void** test2() {  SqlSession session = MyBatisUtil.*getSession*();  // 获取Mapper接口的实现类对象(Mapper接口.class)  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  // EmployeeDao dao=new EmployeeDaoImpl();  // 调用Mapper接口中的方法  Employee emp = **new** Employee("纪晓芙", "女", 4350);  emp.setEid(9);  **int** i = mapper.update(emp);  **if** (i > 0) {  System.*out*.println("修改成功");  } **else** {  System.*out*.println("修改失败");  }  session.commit();  session.close();  } |

## 删除

### 接口

|  |
| --- |
| //删除  **int** delete(Integer eid)； |

### mapper.xm

|  |
| --- |
| <!-- 执行删除,如果只有1个参数且参数是简单类型的，可以省略parameterType不写 -->  <delete id=*"delete"* parameterType=*"java.lang.Integer"* >  delete from t\_employee where eid=#{eid}  </delete> |

### 测试

|  |
| --- |
| // 测试删除  @Test  **public** **void** test5() {  SqlSession session = MyBatisUtil.*getSession*();  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  **int** i = mapper.delete(5);  **if** (i > 0) {  System.*out*.println("删除成功");  } **else** {  System.*out*.println("删除失败");  }  session.commit();  session.close();  } |

## 查全部

### 接口

|  |
| --- |
| //全查询  List<Employee> queryAll(); |

### mapper.xm

|  |
| --- |
| <!-- 执行查询 ,resultType:指定返回值类型，如果返回类型是List集合，则resultType  只用写集合中元素的类型，如果返回值是简单类型，可以省略resultType -->  <select id=*"queryAll"* resultType=*"Employee"*>  <!-- select \* from t\_employee -->  <!-- 引用sql片段 -->  <include refid=*"empSql"*></include>  </select> |

### 测试

|  |
| --- |
| // 测试全查询  @Test  **public** **void** test3() {  SqlSession session = MyBatisUtil.*getSession*();  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  List<Employee> list = mapper.queryAll();  **for** (Employee employee : list) {  System.*out*.println(employee);  }  session.close();  } |

## 查单个

### 接口

|  |
| --- |
| //查询单对象  Employee queryById(Integer eid); |

### mapper.xm

|  |
| --- |
| <select id=*"queryById"* resultType=*"employee"*>  <include refid=*"empSql"*></include> where eid=${value}  </select> |

### 测试

|  |
| --- |
| // 测试查询单个对象  @Test  **public** **void** test4() {  SqlSession session = MyBatisUtil.*getSession*();  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  Employee emp = mapper.queryById(7);  Employee emp2 = mapper.queryById(7);  **if** (emp != **null**) {  System.*out*.println(emp);  } **else** {  System.*out*.println("查无此人");  }  session.close();  } |

## 多个参数

### 接口

|  |
| --- |
| //根据性别和工资查询  List<Employee> queryByConditon(String sex,Integer salary); |

### mapper.xm

|  |
| --- |
| <!-- MyBatis处理多个简单参数的规则：  将多个简单参数通过Map来存储，Map中的key默认是param1,parm2...paramN(在mybatis3.3以前版本中,key还可以是0,1...N)，  如果我们自定义了Map来存储参数，那么key就是我们定义的key ,在 SQL语句中通过#{key}来拿到参数的值 -->  <select id=*"queryByConditon"* resultType=*"cn.sxt.domain.Employee"*>  select \* from t\_employee where sex=#{param1} and salary>#{param2}  </select> |

### 测试

|  |
| --- |
| // 测试按条件查询  @Test  **public** **void** test6() {  SqlSession session = MyBatisUtil.*getSession*();  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  List<Employee> list = mapper.queryByConditon("女", 4300);  **for** (Employee employee : list) {  System.*out*.println(employee);  }  session.close();  } |

## 多个参数(map)

### 接口

|  |
| --- |
| //根据性别和工资查询  List<Employee> queryByConditonWithMap(Map<String, Object> map); |

### mapper.xm

|  |
| --- |
| <!-- MyBatis处理多个简单参数的规则：  将多个简单参数通过Map来存储，Map中的key默认是param1,parm2...paramN(在mybatis3.3以前版本中,key还可以是0,1...N)，  如果我们自定义了Map来存储参数，那么key就是我们定义的key ,在 SQL语句中通过#{key}来拿到参数的值 -->  <select id=*"queryByConditonWithMap"* resultType=*"cn.sxt.domain.Employee"* parameterType=*"java.util.Map"*>  select \* from t\_employee where sex=#{sex} and salary>#{rmb}  </select> |

### 测试

|  |
| --- |
| // 测试按条件查询，参数是Map  @Test  **public** **void** test7() {  SqlSession session = MyBatisUtil.*getSession*();  EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);  //把sql语句的参数保存在Map中  Map<String, Object> map = **new** HashMap<String, Object>();  map.put("sex", "女");  map.put("rmb", 4300);  List<Employee> list = mapper.queryByConditonWithMap(map);  **for** (Employee employee : list) {  System.*out*.println(employee);  }  session.close();  } |

## 总结

### 接口定义有如下特点：

1. Mapper接口方法名和Mapper.xml中定义的statement的id相同
2. Mapper接口方法的输入参数类型和mapper.xml中定义的statement的parameterType的类型相同
3. Mapper接口方法的输出参数类型和mapper.xml中定义的statement的resultType的类型相同

## 返回值总结

1、selectOne和selectList

动态代理对象调用sqlSession.selectOne()和sqlSession.selectList()是根据mapper接口方法的返回值决定，如果返回list则调用selectList方法，如果返回单个对象则调用selectOne方法。如果定义的接口方法中的返回值是list，但是接收使用单值，会报错。

2、namespace

mybatis官方推荐使用mapper代理方法开发mapper接口，程序员不用编写mapper接口实现类，使用mapper代理方法时，输入参数可以使用pojo包装对象或map对象，保证dao的通用性。

# mybatis.cfg.xml配置文件

## 配置内容

mybatis.cfg.xml中配置的内容和顺序如下：

|  |
| --- |
| <?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>  <!--属性 -->  <properties></properties>  <!--设置 -->  <settings></settings>  <!--类型别名 -->  <typeAliases></typeAliases>  <!--类型处理器 -->  <typeHandlers></typeHandlers>  <!--对象工厂 -->  <objectFactory type=*""*></objectFactory>  <!--插件 -->  <plugins></plugins>  <!--配置环境 -->  <environments default=*""*>  <!--环境变量 -->  <environment id=*""*>  <!--事务管理器 -->  <transactionManager type=*""*></transactionManager>  <!--数据源 -->  <dataSource type=*""*></dataSource>  </environment>  </environments>  <!--数据库厂商标识 -->  <databaseIdProvider type=*""*></databaseIdProvider>  <!--映射器 -->  <mappers></mappers>  </configuration> |

但是需要注意的是，MyBatis配置项的顺序不能颠倒。如果颠倒了它们的顺序，那么在MyBatis启动阶段就会发生异常，导致程序无法运行。

本章的任务是了解MyBatis配置项的作用，其中properties、settings、typeAliases、

typeHandler、plugin、environments、mappers是常用的内容。本章不讨论plugin（插件）元素的使用，在进一步学习MyBatis的许多底层内容和设计后我们才会学习它。objectFactory和databaseldProvider不常用。

## properties（属性）

properties 属性可以给系统配置一些运行参数，可以放在XML文件或者properties文件中，而不是放在Java编码中，这样的好处在于方便参数修改，而不会引起代码的重新编译。

一般而言，MyBatis 提供了3种方式让我们使用properties，它们是：

property子元素。

properties文件。

程序代码传递。

### properties

使用properties文件是比较普遍的方法，一方面这个文件十分简单，其逻辑就是键值对

应，我们可以配置多个键值放在一个properties文件中，也可以把多个键值放到多个properties文件中，这些都是允许的，它方便日后维护和修改。

在数据库连接阐述单独配置db.properties，只需要在mybatis配置加载db.properties的属性在mybatis的配置文件中就不用硬编码。

为什么配置该属性？

本省的mybatis.cfg.xml，里面的东西过多，需要修改或者是加载数据库信息，可能时间过长。

SqlMapConfig.xml可以引用java属性文件中的配置信息如下：

在classpath下定义db.properties文件，

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://localhost:3306/mybatis  jdbc.username=root  jdbc.password=mysql |

SqlMapConfig.xml引用如下：

|  |
| --- |
| <properties resource="db.properties"/>//2然后读取这里的属性。  <property name="name" value=""/>//1这里定义的属性先被读取。  <properties >  <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <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> |

注意： MyBatis 将按照下面的顺序来加载属性：

* 在 properties 元素体内定义的属性首先被读取。
* 然后会读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。
* 最后读取parameterType传递的属性，它会覆盖已读取的同名属性。如果在sql语句中通过${name}获取properties中的属性，获取不到。

因此，通过parameterType传递的属性具有最高优先级，resource或 url 加载的属性次之，最低优先级的是 properties 元素体内定义的属性。

建议：不要早properties元素内添加任何属性值，只将属性值定义properties文件中，子啊properties中定义的属性要既有一定的命名特殊性。

测试是否成功：

## settings（配置）

mybatis全局配置参数，全局参数将会影响mybatis的运行行为。

详细参见“学习资料/mybatis-settings.xlsx”文件







### mybatis运行日志

和其它框架整合时，使用log4j来记录mybatis运行日志

|  |
| --- |
| <settings>  <!-- 和其它框架整合时，使用log4j来记录mybatis运行日志 -->  <setting name=*"logImpl"* value=*"LOG4J"*/>  </settings> |

## typeAliases（类型别名）

由于类的全限定名称很长，需要大量使用的时候，总写哪么长的名称不方便。在MyBatis中允许定义一个简写来代表这个类，这就是别名，别名分为系统定义别名和自定义别名。

在MyBatis 中别名由类TypeAliasRegistry （org.apache.ibatis.type.TypeAliasRegistry)去定义。

注意，在MyBatis中别名不区分大小写。

在mapperxml中，定义很多的statement，statement 需要parameterType.指定输入参数的类型、需要resultType指定输出结果的映射类型。如果在指定类型时输入类型全路径，不方便进行开发，可以针对parameterype或 resultType.指定的类型定义一些别名，在mapperxml中通过别名定义，方便开发。

### mybatis支持别名：

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |

### 自定义别名：

在SqlMapConfig.xml中配置：

|  |
| --- |
| <typeAliases>  <!-- 用typeAlias元素指定类型的别名，从而简化Mapper.xml中的类型引用 type：指定类型的全限定名 alias:别名 -->  <typeAlias type="cn.sxt.domain.Employee" alias="emp"/>  <!-- package：是为某个包下的所有类批量配置别名，默认别名就是类名，首字母不区分大小写 -->  <package name=*"cn.sxt.domain"* />  </typeAliases> |

这样MyBatis 将扫描这个包里面的类，将其第一个字母变为小写作为其别名，比如类Role的别名会变为role，而User的别名会变为user。使用这样的规则，有时候会出现重名，比如com.learn.ssm.chapter3.pojo.User 这个类，MyBatis 还增加了对包com.learn.ssm.chapter3.pojo的扫描，那么就会出现异常，这个时候可以使用MyBatis提供的注解@Alias("user3"）进行区分。

|  |
| --- |
| @Alias("employee")  **public** **class** Employee {…} |

## typeHandlers（类型处理器）

mybatis中通过 typeHandlers完成jdbc和java类型的转换。

类型处理器用于java类型和jdbc类型映射，如下：

<select id=*"findUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

mybatis自带的类型处理器基本上满足日常需求，不需要单独定义。

### mybatis支持类型处理器：

|  |  |  |
| --- | --- | --- |
| 类型处理器 | **Java**类型 | **JDBC**类型 |
| BooleanTypeHandler | Boolean，boolean | 任何兼容的布尔值 |
| ByteTypeHandler | Byte，byte | 任何兼容的数字或字节类型 |
| ShortTypeHandler | Short，short | 任何兼容的数字或短整型 |
| IntegerTypeHandler | Integer，int | 任何兼容的数字和整型 |
| LongTypeHandler | Long，long | 任何兼容的数字或长整型 |
| FloatTypeHandler | Float，float | 任何兼容的数字或单精度浮点型 |
| DoubleTypeHandler | Double，double | 任何兼容的数字或双精度浮点型 |
| BigDecimalTypeHandler | BigDecimal | 任何兼容的数字或十进制小数类型 |
| StringTypeHandler | String | CHAR和VARCHAR类型 |
| ClobTypeHandler | String | CLOB和LONGVARCHAR类型 |
| NStringTypeHandler | String | NVARCHAR和NCHAR类型 |
| NClobTypeHandler | String | NCLOB类型 |
| ByteArrayTypeHandler | byte[] | 任何兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB和LONGVARBINARY类型 |
| DateTypeHandler | Date（java.util） | TIMESTAMP类型 |
| DateOnlyTypeHandler | Date（java.util） | DATE类型 |
| TimeOnlyTypeHandler | Date（java.util） | TIME类型 |
| SqlTimestampTypeHandler | Timestamp（java.sql） | TIMESTAMP类型 |
| SqlDateTypeHandler | Date（java.sql） | DATE类型 |
| SqlTimeTypeHandler | Time（java.sql） | TIME类型 |
| ObjectTypeHandler | 任意 | 其他或未指定类型 |
| EnumTypeHandler | Enumeration类型 | VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。 |

## environments （运行环境）

在MyBatis中，运行环境主要的作用是配置数据库信息，它可以配置多个数据库，一

般而言只需要配置其中的一个就可以了。它下面又分为两个可配置的元素：事务管理器

（transactionManager)、数据源（dataSource)。在实际的工作中，大部分情况下会采用Spring对数据源和数据库的事务进行管理。

|  |
| --- |
| <environments default=*"development"*>  <environment id=*"development"*>  <!-- 使用jdbc事务管理器控制事务 -->  <transactionManager type=*"JDBC"* />  <!-- 配置数据库连接池(又叫数据源) -->  <dataSource type=*"POOLED"*>  <!-- 使用${key}去获取Properties文件中的value -->  <property name=*"driver"* value=*"${driver}"* />  <property name=*"url"* value=*"${url}"* />  <property name=*"username"* value=*"${username}"* />  <property name=*"password"* value=*"${password}"* />  </dataSource>  </environment>  </environments> |

## mappers（映射器）

Mapper配置的几种方法：

### <mapper resource=" " />：加载单个映射文件。

使用相对于类路径的资源，

如：<mapper resource="sqlmap/User.xml" />

注：mapper.xml和mapper接口可以不再同一个包。

### <mapper url=" " />：加载的那个文件加载

使用完全限定路径

如：<mapper url="file:///D:\workspace\_spingmvc\mybatis\_01\config\sqlmap\User.xml" />

### <mapper class=" " />

使用mapper接口类全限定名，mybatis会自动加载namespace与这个接口名相同的，mapper.xml

如：<mapper class="cn.itcast.mybatis.mapper.UserMapper"/>

//类路径用点分割

**注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。**

**前提：使用mapper代理的方法。**

### <package name=""/>

注册指定包下的所有mapper接口

如：<package name="cn.itcast.mybatis.mapper"/>

**注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。**

**自动扫描包下面的所有接口。**

**前提：使用mapper代理的方法。**

## include

### 首先定义一个sql标签，一定要定义唯一id

<sql id="Base\_Column\_List" >name,age</sql>

### 然后通过id引用

<select id="selectAll">

select

<include refid="Base\_Column\_List" />

from student

</select>

这个<include refid="Base\_Column\_List" />会自动把上面的代码贴过来。

## 连接池

### mybatis连接池

mybatis连接池提供了3种方式的配置:

配置的位置:主配置文件SqIMapConfig. xml中的dataSource标签, type属性就是表示采用何种连接池方式。

POOLED 采用传统的*javax. sql. DataSource*规范中的连接池, mybatis中有针对规范的实现.

UNPOOLED采用传统的获取连接的方式,虽然也实现javax. sql. DataSource，但是并没有使用池的思想。

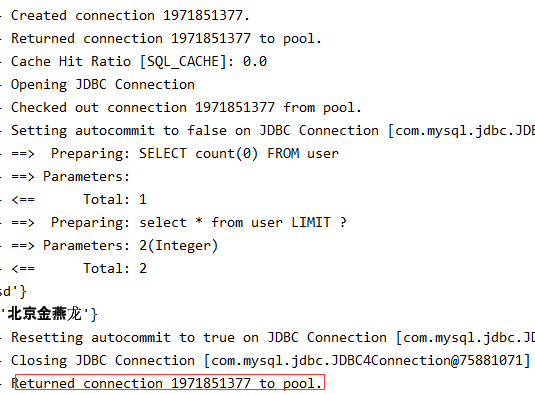
JNDI: 采用服务器提供的JNDI技术实现

### 配置连接池

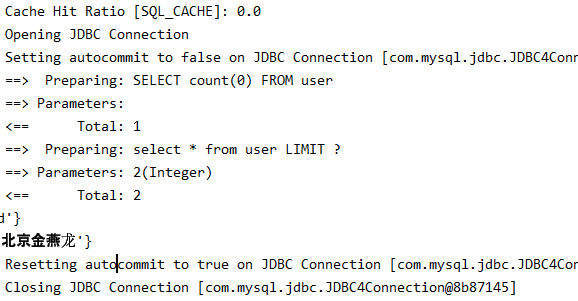


### 对比

使用连接池后，有归还连接池的操作



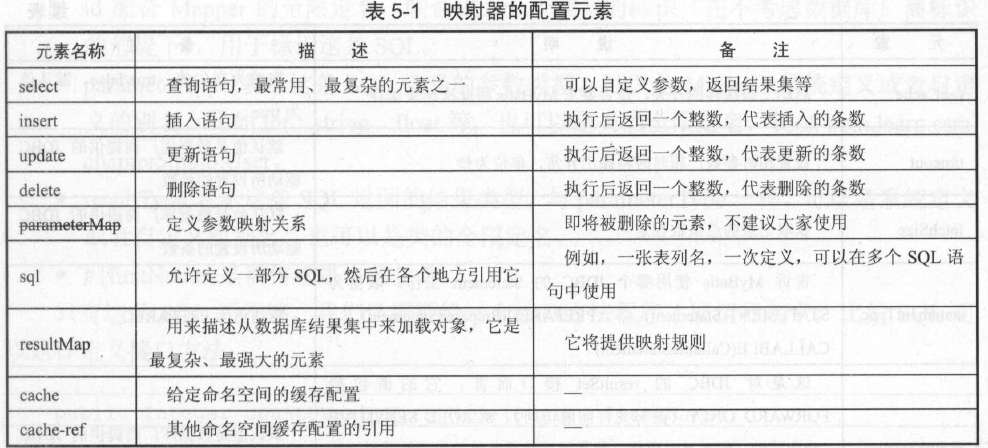
未使用连接池:直接关闭了连接



# Mapper.xml映射文件

映射器是MyBatis最复杂且最重要的组件。它由一个接口加上XML文件（或者注解）

组成。在映射器中可以配置参数、各类的SQL语句、存储过程、缓存、级联等复杂的内容，并且通过简易的映射规则映射到指定的POJO或者其他对象上，映射器能有效消除JDBC底层的代码。



## select

在映射器中select元素代表SQL的select语句，用于查询。在SQL中，select 语句是

用得最多的语句，在MyBatis中select 元素也是用得最多的元素，使用的多就意味着强大和复杂。先来看看select元素的配置。

### 常用的属性

|  |  |  |
| --- | --- | --- |
|  |  |  |
| id | 它和Mapp er 的命名空间组合起来是唯一的，供MyBati s 调用 | 如果命名空间和id不唯一，mybatis将抛异常。 |
| parameterType | 可以给出类的全命名，也可以给出别名，但是别名必须是MyBatis内部定义或者自定义的 | 可以选择javabean，map等简单那元素传递参数。 |
| resultType | 定义类的全路径，在允许自动匹配的情况下，结果集将通过JavaBean的规范映射；  或定义为int、double、float、map等参数；  也可以使用别名，但是要符合别名规范，且不能和resultMap同时使用 | 常用的参数之一，比如统计总条数时可以把它的值设置为int |
| resultMap | 它是映射集的引用，将执行强大的映射功能。我们可以使用resultType 和 resultMap其中的一个，resultMap能提供自定义映射规则的机会 | MyBati s最复杂的元素，可以配置决射规则、级联等 |

### 简单的select使用

mapper接口

|  |
| --- |
| **public** **interface** UserMapper {  //1、查询用户的数量  **int** queryALLCount();  } |

mapper.xml

|  |
| --- |
| <select id=*"queryALLCount"* resultType=*"int"*>  select count(\*) from t\_user  </select> |

测试文件

|  |
| --- |
| **public** **void** test11(){  SqlSession session = MyBatisUtil.*getSession*();  UserMapper userMapper = session.getMapper(UserMapper.**class**);  **int** count = userMapper.queryALLCount();  System.*out*.println(count);  } |

id 配合Mapper: 的全限定名，联合成为一个唯一的标识

parameterType :表示这条SQL 接受的参数类型，可以是MyBatis 系统定义或者自定

义的别名，比如int 、string 、float 等，也可以是类的全限定名，’

resultType 表示这条SQL 运回的结果类型，与parameterType 一样，可以是系统定义

或者自定义的别名，也可以是类的全限定名。

# { }是被传递进去的参数。

## insert

MyBatis在执行完一条insert语句后，会返回一个整数表示其影响记录数

|  |
| --- |
| <insert id=*"insert"* parameterType=*"cn.sxt.domain.Employee"* useGeneratedKeys=*"true"*>  <!-- insert into t\_employee(ename,sex,salary) values('${ename}','${sex}',${salary}) -->  insert into t\_employee(ename,sex,salary) values(#{ename},#{sex},#{salary})  </insert> |

id标识出这条SQL，结合命名空间让MyBatis 能够找到它。

parameterType代表传入参数类型。

### 主键回填

上述插入语句，但是它并没有插入id列，因为MySQL中的表格采用了自增主键，MySQL数据库会为该记录生成对应的主键。有时候还可能需要继续使用这个主键，用以关联其他业务，因此有时候把它取到是十分必要的，比如新增用户时，

首先会插入用户表的记录，然后插入用户和角色关系表，插入用户时如果没有办法取到用户的主键，那么就没有办法插入用户和角色关系表了，因此在这个时候要拿到对应的主键，

以便后面的操作，MyBatis提供了这样的支持。

JDBC中的Statement对象在执行插入的SQL后，可以通过getGeneratedKeys方法获得

数据库生成的主键（需要数据库驱动支持），这样便能达到获取主键的功能。在insert 语句

中有一个开关属性useGeneratedKeys，用来控制是否打开这个功能，它的默认值为false。

当打开了这个开关，还要配置其属性keyProperty或keyColumn，告诉系统把生成的主键放

入哪个属性中，如果存在多个主键，就要用逗号（，）将它们隔开。

|  |
| --- |
| <insert id=*"insertUser"* parameterType=*"user"* useGeneratedKeys=*"true"* keyProperty=*"uid"*>  insert into t\_user(uname,uage,address) values(#{uname},#{uage},#{address})  </insert> |

测试

|  |
| --- |
| @Test  **public** **void** test5(){  SqlSession session = MyBatisUtil.*getSession*();  UserMapper userMapper = session.getMapper(UserMapper.**class**);  User user = **new** User();  user.setUname("ss");  user.setUage(13);  user.setAddress("aa");  **int** i = userMapper.insertUser(user);  System.*out*.println(user.getUid());//返回用户的id。  System.*out*.println(i);//返回影响的行数。  session.commit();  } |

## sql元素

sql元素的作用在于可以定义一条SQL的一部分，方便后面的SQL引用它，比如最典型的列名。通常情况下要在select、insert等语句中反复编写它们，特别是那些字段较多的表更是如此，而在MyBatis中，只需要使用sql元素编写一次便能在其他元素中引用它了。

### 定义

|  |
| --- |
| <!-- 定义sql片段：即抽取sql语句中重复的部分 -->  <sql id=*"empSql"*>  select \* from t\_employee  </sql> |

### 使用

|  |
| --- |
| <select id=*"queryAll"* resultType=*"Employee"*>  <!-- select \* from t\_employee -->  <!-- 引用sql片段 -->  <include refid=*"empSql"*></include>  </select> |

## parameterType(输入类型)

通过parameterType指定输入参数的类型，类型可以是简单类型、hashmap、pojo的包装类型。

MyBatis的传入参数parameterType类型分两种：

基本数据类型：int,string,long,Date;

复杂数据类型：类和Map

如何获取参数中的值:

基本数据类型：#{参数} 获取参数中的值

复杂数据类型：#{属性名} ，map中则是#{key}

注：

1、MyBatis处理多个简单参数的规则：将多个简单参数通过Map来存储，Map中的key默认是param1,parm2...paramN(在mybatis3.3以前版本中,key还可以是0,1...N)，

如果我们自定义了Map来存储参数，那么key就是我们定义的key ,在 SQL语句中通过#{key}来拿到参数的值-->

### #{}与${}

#{}实现的是向prepareStatement中的预处理语句中设置参数值，sql语句中#{}表示一个占位符即?。

<!-- 根据id查询用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

使用占位符#{}可以有效防止sql注入，在使用时不需要关心参数值的类型，mybatis会自动进行java类型和jdbc类型的转换。#{}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}和#{}不同，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。使用${}不能防止sql注入，但是有时用${}会非常方便，如下的例子：

<!-- 根据名称模糊查询用户信息 -->

<select id=*"selectUserByName"* parameterType=*"string"* resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

如果本例子使用#{}则传入的字符串中必须有%号，而%是人为拼接在参数中，显然有点麻烦，如果采用${}在sql中拼接为%的方式则在调用mapper接口传递参数就方便很多。

//如果使用占位符号则必须人为在传参数中加%

List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%

List<User>list = userMapper.selectUserByName("管理员");

再比如order by排序，如果将列名通过参数传入sql，根据传的列名进行排序，应该写为：

ORDER BY ${columnName}

如果使用#{}将无法实现此功能。

### 传递简单类型

1、如果传递的是一个参数

|  |
| --- |
| 方法：  User getUserByName(String username);  mapper：  <select id="getUserByName" parameterType="String" resultMap="user">  select \* from d\_user where user\_name =#{1234567890} //（或者#{\_parameter}）  </select> |

2、如果传递的是两个参数

|  |
| --- |
| 方法：  User getUserByNameAndPwd(String username, String password);  mapper：  <select id="getUserByNameAndPwd" parameterType="String" resultMap="user">  select \* from d\_user where user\_name =#{param1} and user\_password=#{param2}  </select>  或者是  User getUserByNameAndPwd(@param("username")String username, @param("password")String password);  <select id="getUserByNameAndPwd" parameterType="String" resultMap="user">  select \* from d\_user where user\_name =#{username} and user\_password=#{password}  </select> |

总结

针对一个基本类型参数，mybatis可以准确的获取这个参数的数值，不论你在mapper中用#{XXX}怎么获取都可以，

针对多个基本参数类型，mybatis需要给他提供一个标识，你可以在方法上添加@param注解 ，当然是，mybatis的包org.apache.ibatis.annotations.Param

参考上边的例子。

### 传递pojo对象

ognl表达式:通过对象的取值方法来获取数据。在写法上把get省略了。

Mybatis使用ognl表达式解析对象字段的值，如下例子：

<!—传递pojo对象综合查询用户信息 -->

<select id=*"findUserByUser"* parameterType=*"user"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是user对象中的字段名称。

测试：

**Public void** testFindUserByUser()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件user对象

User user = **new User()**;

user.setId(1);

user.setUsername("管理员");

//传递user对象查询用户列表

List<User>list = userMapper.findUserByUser(user);

//关闭session

session.close();

}

异常测试：

Sql中字段名输入错误后测试，username输入dusername测试结果报错：

org.apache.ibatis.exceptions.PersistenceException:

### Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class cn.itcast.mybatis.po.User'

### Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property named 'dusername' in 'class cn.itcast.mybatis.po.User'

### 传递pojo包装对象

开发中通过pojo传递查询条件 ，查询条件是综合的查询条件，不仅包括用户查询条件还包括其它的查询条件（比如将用户购买商品信息也作为查询条件），这时可以使用包装对象传递输入参数。

1、定义包装对象

定义包装对象将复杂的查询条件(pojo)以类组合的方式包装起来。

**public** **class** QueryVo {

//自定义用户扩展类、这个类继承了user类。

**private** UserCustom userCustom;/这个类中分装了类的属性。

}

2、mapper.xml映射文件

|  |
| --- |
| <!--用戶信息綜合查詢 -->根据多个条件进行查询。  <select id=*"queryByConditonWithMap"* resultType=*"cn.sxt.domain.queryVo "* parameterType=*" cn.sxt.domain.queryVo "*>//使用自定义的类来接收这个对象。  select \* from t\_employee where sex=#{ UserCustom .sex} and salary>#{salary}//这里查询的结果可能在user表中没有值。此时可以使用自定义的类来接收  </select> |

说明：mybatis底层通过ognl从pojo中获取属性值：#{user.username}，user即是传入的包装对象的属性。queryVo是别名，即上边定义的包装对象类型。

### 传递hashmap

Sql映射文件定义如下：

<!-- 传递hashmap综合查询用户信息 -->

<select id=*"findUserByHashmap"* parameterType=*"hashmap"* resultType=*"user"*>

select \* from user where id=#{id} and username like '%${username}%'

</select>

上边红色标注的是hashmap的key。

测试：

**Public void** testFindUserByHashmap()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件Hashmap对象

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

map.put("id", 1);

map.put("username", "管理员");

//传递Hashmap对象查询用户列表

List<User>list = userMapper.findUserByHashmap(map);

//关闭session

session.close();

}

异常测试：

传递的map中的key和sql中解析的key不一致。

测试结果没有报错，只是通过key获取值为空。

## resultType(输出类型)

### 输出简单类型

参考getnow输出日期类型，看下边的例子输出整型：

Mapper.xml文件

<!-- 获取用户列表总数 -->

<select id=*"findUserCount"* parameterType=*"user"* resultType=*"int"*>

select count(1) from user

</select>

Mapper接口

public int findUserCount(User user) throws Exception;

调用：

**Public void** testFindUserCount() **throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

User user = **new User()**;

user.setUsername("管理员");

//传递Hashmap对象查询用户列表

**int** count = userMapper.findUserCount(user);

//关闭session

session.close();

}

**总结：**

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。使用session的selectOne可查询单条记录。

### 输出JavaBean对象

1、封装的javaBean对象的属性名字必须和数据库查询中的字段名相同，否则不会将属性封装到对象，并且在windows系统下可以忽略数据库和属性的大小写，但是在linux系统中是不行的。

参考*findUserById*的定义：

Mapper.xml

<!-- 根据id查询用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

Mapper接口：

public User findUserById(int id) throws Exception;

测试：

**Public void** testFindUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//通过mapper接口调用statement

User user = userMapper.findUserById(1);

System.out.println(user);

//关闭session

session.close();

}

使用session调用selectOne查询单条记录。

### 输出List列表

参考selectUserByName的定义：

Mapper.xml

<!-- 根据名称模糊查询用户信息 -->

<select id=*"findUserByUsername"* parameterType=*"string"* resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

Mapper接口：

public List<User> findUserByUsername(String username) throws Exception;

测试：

**Public void** testFindUserByUsername()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//如果使用占位符号则必须人为在传参数中加%

//List<User> list = userMapper.selectUserByName("%管理员%");

//如果使用${}原始符号则不用人为在参数中加%

List<User> list = userMapper.findUserByUsername("管理员");

//关闭session

session.close();

}

使用session的selectList方法获取pojo列表。

输出pojo对象可以改用hashmap输出类型，将输出的字段名称作为map的key，value为字段值。

### 返回map类型

1如果查询的结果是一条，我们可以把查询的数据以{表字段名， 对应的值}方式存入到Map中。

mapper 接口：

|  |
| --- |
| // 根据 id 查询信息，并把结果信息封装成 Map  Map<String, Object> getEmpAsMapById(Integer id); |

SQL 映射文件：

|  |
| --- |
| <!--  注意这里的 resultType 返回值类型是 'map'  -->  <select id="getEmpAsMapById" resultType="map">  select \* from t\_employee where id = #{id}  </select> |

2. 如果查询的结果是多条数据，我们也可以把查询的数据以{表中某一字段名, JavaBean}方式来封装成Map。

|  |
| --- |
| // 查询所有员工的信息，把数据库中的 'id' 字段作为 key,对应的 value 封装成 Employee 对象  // @MapKey 中的值表示用数据库中的哪个字段名作 key  @MapKey("id")  Map<Integer, Employee> getAllEmpsAsMap(); |
| <!--  注意 resultType 返回值类型，不再是 'map'，而是 Map 的 value 对应的 JavaBean 类型  -->  <select id="getAllEmpsAsMap" resultType="employee">  select \* from t\_employee  </select> |

### resultType总结：

1、如果想要使用resultType进行输出映射，只有查询出来的列名，和pojo中的属性名一致，该列才映射成功。

如果查询出来的列名和pojo中的属性名全部不一致，没有创建 pojo对象。

只要查询出来的列名和pojo中的属性有一个一致，就会创建 pojo对象。

2、输出pojo对象和输出pojo列表在sql中定义的resultType是一样的。

3、返回单个pojo对象要保证sql查询出来的结果集为单条，内部使用session.selectOne方法调用，mapper接口使用pojo对象作为方法返回值。

4、返回pojo列表表示查询出来的结果集可能为多条，内部使用session.selectList方法，mapper接口使用List<pojo>对象作为方法返回值。

### 数据库和pojo字段名对应解决方法

1、通过给查询的字段起别名

|  |
| --- |
| <**select id="queryAll" resultType="com.sx.domain.User"**>  select username as username,sex as usersex,address as address from user </**select**> |

2、配置，查询结果的列名和实体类的属性名的对应关系

|  |
| --- |
| <**resultMap id="userMap" type="com.sx.domain.User"**>  *<!--实体类与数据库中对应-->  <!--主键字段对应-->* <**id property="userId" column="id"**></**id**>  *<!--非主键字段的对应-->* <**result property="userName" column="username"**></**result**>  <**result property="userAddress" column="address"**></**result**>  <**result property="userSex" column="sex"**></**result**> </**resultMap**> |

## resultMap

resultMap的作用是定义映射规则、级联的更新、定制类型转化器等。resultMap定义的主要是一个结果集的映射关系，也就是SQL到Java Bean的映射关系定义，它也支持级联等特性。只是MyBatis现有的版本只支持resultMap查询，不支持更新或者保存，更不必说级联的更新、删除和修改了。

### resultMap的元素构成

|  |
| --- |
| <resultMap type=*""* id=*""*>  <constructor>  <arg/>  <idArg/>  </constructor>  <id/>  <result/>  <association property=*""*></association>  <collection property=*""*></collection>  <discriminator javaType=*""*>  <case value=*""*></case>  </discriminator>  </resultMap> |

### 元素内容

|  |
| --- |
| <!--column不做限制，可以为任意表的字段，而property须为type 定义的pojo属性-->  <resultMap id="唯一的标识" type="映射的pojo对象">  <id column="表的主键字段，或者可以为查询语句中的别名字段" jdbcType="字段类型" property="映射pojo对象的主键属性" />  <result column="表的一个字段（可以为任意表的一个字段）" jdbcType="字段类型" property="映射到pojo对象的一个属性（须为type定义的pojo对象中的一个属性）"/>  <association property="pojo的一个对象属性" javaType="pojo关联的pojo对象">  <id column="关联pojo对象对应表的主键字段" jdbcType="字段类型" property="关联pojo对象的主席属性"/>  <result column="任意表的字段" jdbcType="字段类型" property="关联pojo对象的属性"/>  </association>  <!-- 集合中的property须为oftype定义的pojo对象的属性-->  <collection property="pojo的集合属性" ofType="集合中的pojo对象">  <id column="集合中pojo对象对应的表的主键字段" jdbcType="字段类型" property="集合中pojo对象的主键属性" />  <result column="可以为任意表的字段" jdbcType="字段类型" property="集合中的pojo对象的属性" />  </collection>  </resultMap> |

### 别名操作

resultType可以指定pojo将查询结果映射为pojo，但需要pojo的属性名和sql查询的列名一致方可映射成功。

如果sql查询字段名和pojo的属性名不一致，可以通过resultMap将字段名和属性名作一个对应关系 ，resultMap实质上还需要将查询结果映射到pojo对象中。

resultMap可以实现将查询结果映射为复杂类型的pojo，比如在查询结果映射对象中包括pojo和list实现一对一查询和一对多查询。

|  |
| --- |
| <!--resultType:指定定义的resultmap的id，如果这个result是别的mapper文件中的，需要在前面加namespace -->  <select id=*"queryAll"* resultMap=*"empMap"*>//使用resultmap指定要返回的類型。  select eid e\_id,ename e\_name from t\_employee  </select>  定义返回的类型：使别名和类的属性建立联系。  <!--select eid e\_id,ename e\_name from t\_employee 完成别名与employee中该属性的映射 -->  <!--type:resultmap最终映射成的java类型 -->  <!--id:对resultMap的唯一标识-->  <resultMap type=*"com.sx.domain.Employee"* id=*"empMap"*>  <!--id:表示查询结果集中的唯一标识  属性:column:查询出来的列名。  property:type中类的属性名。  result:对普通列的定义  完成employee与查询结果的一一对应。  -->  <id column=*"e\_id"* property=*"eid"*/>  <result column=*"e\_name"* property=*"ename"*/>  </resultMap> |

# 关联查询

## 一对一

|  |
| --- |
| <!-- 订单及订单明细的resultMap 使用extends继承，不用在中配置订单信息和用户信息的映射 -->  <resultMap type=*"cn.itcast.mybatis.po.Orders"* id=*"OrdersAndOrderDetailResultMap"*  extends=*"OrdersUserResultMap"*>  <!-- 订单信息 -->  <!-- 用户信息 -->  <!-- 使用extends继承，不用在中配置订单信息和用户信息的映射 -->  <!-- 订单明细信息 一个订单关联查询出了多条明细，要使用collection进行映射  collection：对关联查询到多条记录映射到集合对象中  property：将关联查询到多条记录映射到cn.itcast.mybatis.po.Orders哪个属性  ofType：指定映射到list集合属性中pojo的类型 -->  <collection property=*"orderdetails"* ofType=*"cn.itcast.mybatis.po.Orderdetail"*>  <!-- id：订单明细唯 一标识 property:要将订单明细的唯 一标识  映射到cn.itcast.mybatis.po.Orderdetail的哪个属性 -->  <id column=*"orderdetail\_id"* property=*"id"* />  <result column=*"items\_id"* property=*"itemsId"* />  <result column=*"items\_num"* property=*"itemsNum"* />  <result column=*"orders\_id"* property=*"ordersId"* />  </collection>  </resultMap> |

## 多对一

### 结果嵌套

mapper接口

|  |
| --- |
| //查询员工时查他所在部门，使用Domian类封装--结果嵌套  Employee queryEmp(Integer eid); |

mapper配置文件

|  |
| --- |
| <!-- 自定义ResultMap来封装连接查询的结果集  结果嵌套: 使用association标签把关联对象的查询结果集嵌套在resultMap中  -->  <resultMap type=*"Employee"* id=*"empMap"*>  <普通的数据类型的映射>  <id column=*"eid"* property=*"eid"* />  <result column=*"ename"* property=*"ename"* />  <result column=*"salary"* property=*"salary"* />  <!-- association：配置关联的一方对象，相当于hibernate里面的many-to-one property:指定关联的一方的属性名  javaType:关联对象的java类型,不能省略; 在association指定一方对象的属性及字段的映射  -->  <javabean类型的映射>  <association property=*"dept"* javaType=*"Dept"*>  <主键映射>  <id column=*"deptId"* property=*"did"* />  <非属性映射>  <result column=*"dname"* property=*"dname"* />  </association>  </resultMap> |

### 查询嵌套

mapper接口

|  |
| --- |
| //查询员工时查他所在部门，使用Domian类封装--结果嵌套  Employee queryEmp(Integer eid); |

mapper配置文件

|  |
| --- |
| 查询语句  <!-- 查询员工，返回ResultMap，记得一定要查外键 ，需要根据外键的值，来查询其他相关联的表的信息。-->  <select id=*"queryEmpPlus"* resultMap=*"empStepMap"* >  select eid,ename,salary,d\_id from t\_employee where eid=#{id}  </select>  映射结果类型  <resultMap type=*"Employee"* id=*"empStepMap"*>  <id column=*"eid"* property=*"eid"* />  <result column=*"ename"* property=*"ename"* />  <result column=*"salary"* property=*"salary"* />  <!-- association：配置关联的一方对象  select:指定用哪个satementment去查询关联对象，  值就是 satementment完整id:namespace+sql标签id  column:指定用哪个列的值传给查询关联对象的方法，通常就是外键字段  简单理解:使用column列的值传到select所指定的satementment里面  去执行查询关联对象的方法(定位查询关联对象的SQL语句)  select的值通常是引用另一个Mapper.xml中的sql标签(statement)  -->  <association property=*"dept"* javaType=*"Dept"*  select=*"cn.sxt.mapper.DeptMapper.queryById"* column=*"d\_id"*></association>  </resultMap> |

注意， association 元素代表着一对一级联的开始。property 属性代表映射到POJO 属性上。select 配置是命名空间+SQL id 的形式，这样便可以指向对应Mapper 的SQL,MyBatis 就会通过对应的SQL 将数据查询回来。column 代表SQL 的列，用作参数传递给select 属性制定的SQL ，如果是多个参数，则需要用逗号隔开。

## 一对多

部门表

|  |
| --- |
| **private** Integer did;  **private** String dname;  **private** List<Employee> empList；//定义员工的list集合，实现一个部门多个员工 |

### 查询嵌套

部门mapper接口

|  |
| --- |
| // 查询一个部门时查询它的所有员工--查询嵌套  Dept queryDeptPlus(Integer did); |

部门mapper配置

|  |
| --- |
| <!-- 查询嵌套的查询:sql只查主对象,查到的id是员工表的外键，并传递给employee中的statement -->  <select id=*"queryDeptPlus"* resultMap=*"deptStepMap"*>  select did,dname from t\_dept where did=#{id}  </select>  //查询结果映射到dept对象中。  <!-- 自定义resultMap，封装主对象查询结果，  查询嵌套：用collection的select属性去发起一个sql查询 -->  <resultMap type=*"Dept"* id=*"deptStepMap"*>  <id column=*"did"* property=*"did"* />  <result column=*"dname"* property=*"dname"* />  <!--collection配置关联的多方集合 ,  property：集合属性名  javaType:集合的java类型  ofType:集合中元素的类型  select:指定一个statement的id,去定位查询sql，去查询关联的多方对象  column:用哪个列的值传到查多方对象的方法里  -->  <collection  property=*"emplist"*  javaType=*"java.util.List"*  ofType=*"Employee"*  //这里会自动调用员工表中的statement  select=*"cn.sxt.mapper.EmployeeMapper.queryEmpBydid"*  column=*"did"*>  </collection>  </resultMap> |

员工的mapper配置。

|  |
| --- |
| <!—根据不们表中的mapper传过来的id查询员工的信息，并返回调用它的statement -->  <select id=*"queryEmpBydid"* resultType=*"Employee"*>  select eid,ename,salary from t\_employee where d\_id=#{id}  </select> |

### 结果嵌套

mapper接口

|  |
| --- |
| // 查询一个部门时查询它的所有员工--结果嵌套  Dept queryDept(Integer did); |

mapper配置

|  |
| --- |
| <!-- 结果嵌套的查询 -->  <select id=*"queryDept"* resultMap=*"deptMap"*>  select did,dname,  eid,ename,salary from t\_dept t1 LEFT JOIN t\_employee t2 on  t1.did=t2.d\_id  WHERE did=1  </select>  <!-- 自定义resultMap，封装连接查询结果集 结果嵌套：  把关联的多方集合使用collection嵌套在resultMap中 -->  <resultMap type=*"Dept"* id=*"deptMap"*>  <id column=*"did"* property=*"did"* />  <result column=*"dname"* property=*"dname"* />  <!--collection配置关联的多方集合 ,  property：集合属性名  javaType:集合的java类型  ofType:集合中元素的类型 -->  <collection property=*"emplist"* javaType=*"java.util.List"*  ofType=*"Employee"*>  <id column=*"eid"* property=*"eid"* />  <result column=*"ename"* property=*"ename"* />  <result column=*"salary"* property=*"salary"* />  </collection>  </resultMap> |

### 打开延迟加载开关

只对分布查询有用

在mybatis核心配置文件中配置：

lazyLoadingEnabled、aggressiveLazyLoading

|  |  |  |  |
| --- | --- | --- | --- |
| 设置项 | 描述 | 允许值 | 默认值 |
| lazyLoadingEnabled | 全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。 | true | false | false |
| aggressiveLazyLoading | 当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。 | true | false | true |

# 动态sql(重点)

作用：通过mybatis提供的各种标签方法实现动态拼接sql。

## If标签

|  |
| --- |
| <!-- 根据条件查询员工 如果eid不为空，就根据id查询；如果ename不为名，按ename做模糊查询.. if:测试条件，相当于jstl的if  where:将所有的查询条件嵌套起来，mybatis会把拼接后SQL的首个and去掉  choose:只会拼接第1次满足test条件的sql，后面的测试条件就不看了 -->  <select id=*"queryEmpList"* parameterType=*"Employee"* resultType=*"Employee"*>  select eid,ename,sex,salary from t\_employee  <where>//这里mybatis会自动去添加where，并自动去掉条件中的第一个and  <choose>  <when test=*"eid!=null"*>  and eid=#{eid}  </when>  <when test=*"ename!=null"*>  and ename like '%${ename}%'  </when>  <otherwise>  and sex=#{sex}  </otherwise>  </choose>  </where>  </select> |

**注意要做不等于空字符串校验。**

## set

|  |
| --- |
| <!-- set标签：包裹修改时的条件,会把多余的","去掉 -->  <update id=*"updateEmp"* parameterType=*"Employee"*>  update t\_employee  <set>  <if test=*"ename!=null"*> ename=#{ename},</if>  <if test=*"sex!=null"*> sex=#{sex},</if>  <if test=*"salary!=null"*> salary=#{salary},</if>//这里会自动去掉最后一个的逗号  </set>  where eid=#{eid}  </update> |

## Where

上边的sql也可以改为：

<select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</where>

</select>

<where />可以自动处理第一个and。

## foreach

向sql传递数组或List，mybatis使用foreach解析，如下：

### 通过pojo传递list

* 需求

传入多个id查询用户信息，用下边两个sql实现：

SELECT \* FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)

SELECT \* FROM USERS WHERE username LIKE '%张%' id IN (10,89,16)

* 在pojo中定义list属性ids存储多个用户id，并添加getter/setter方法



* mapper.xml

<if test=*"ids!=null and ids.size>0"*>

<foreach collection=*"ids"* open=*" and id in("* close=*")"* item=*"id"* separator=*","* >

#{id}

</foreach>

</if>

* 测试代码：

List<Integer> ids = **new** ArrayList<Integer>();

ids.add(1);//查询id为1的用户

ids.add(10); //查询id为10的用户

queryVo.setIds(ids);

List<User> list = userMapper.findUserList(queryVo);

### 传递单个List

传递List类型在编写mapper.xml没有区别，唯一不同的是只有一个List参数时它的参数名为list。

如下：

* Mapper.xml

<select id=*"selectUserByList"* parameterType=*"java.util.List"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递List，List中是pojo -->

<if test=*"list!=null"*>

<foreach collection=*"list"* item=*"item"* open=*"and id in("*separator=*","*close=*")"*>

#{item.id}

</foreach>

</if>

</where>

</select>

* Mapper接口

public List<User> selectUserByList(List userlist) throws Exception;

* 测试：

**Public void** testselectUserByList()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

List<User> userlist = **new** ArrayList<User>();

User user = **new User()**;

user.setId(1);

userlist.add(user);

user = **new User()**;

user.setId(2);

userlist.add(user);

//传递userlist列表查询用户列表

List<User>list = userMapper.selectUserByList(userlist);

//关闭session

session.close();

}

### 传递单个数组（数组中是pojo）：

请阅读文档学习。

* Mapper.xml

<!-- 传递数组综合查询用户信息 -->

<select id=*"selectUserByArray"* parameterType=*"Object[]"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递数组 -->

<if test=*"array!=null"*>

<foreach collection=*"array"* index=*"index"* item=*"item"* open=*"and id in("*separator=*","*close=*")"*>

#{item.id}

</foreach>

</if>

</where>

</select>

sql只接收一个数组参数，这时sql解析参数的名称mybatis固定为array，如果数组是通过一个pojo传递到sql则参数的名称为pojo中的属性名。

index：为数组的下标。

item：为数组每个元素的名称，名称随意定义

open：循环开始

close：循环结束

separator：中间分隔输出

* Mapper接口：

public List<User> selectUserByArray(Object[] userlist) throws Exception;

* 测试：

**Public void** testselectUserByArray()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

Object[] userlist = **new** Object[2];

User user = **new User()**;

user.setId(1);

userlist[0]=user;

user = **new User()**;

user.setId(2);

userlist[1]=user;

//传递user对象查询用户列表

List<User>list = userMapper.selectUserByArray(userlist);

//关闭session

session.close();

}

### 传递单个数组（数组中是字符串类型）：

请阅读文档学习。

* Mapper.xml

<!-- 传递数组综合查询用户信息 -->

<select id=*"selectUserByArray"* parameterType=*"Object[]"* resultType=*"user"*>

select \* from user

<where>

<!-- 传递数组 -->

<if test=*"array!=null"*>

<foreach collection=*"array"*index=*"index"*item=*"item"*open=*"and id in("*separator=*","*close=*")"*>

#{item}

</foreach>

</if>

</where>

</select>

如果数组中是简单类型则写为#{item}，不用再通过ognl获取对象属性值了。

* Mapper接口：

public List<User> selectUserByArray(Object[] userlist) throws Exception;

* 测试：

**Public void** testselectUserByArray()**throws** Exception{

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//构造查询条件List

Object[] userlist = **new** Object[2];

userlist[0]=”1”;

userlist[1]=”2”;

//传递user对象查询用户列表

List<User>list = userMapper.selectUserByArray(userlist);

//关闭session

session.close();

}

## Sql片段

Sql中可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的，如下：

<!-- 传递pojo综合查询用户信息 -->

<select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</where>

</select>

* 将where条件抽取出来：

<sql id=*"query\_user\_where"*>

<if test=*"id!=null and id!=''"*>

and id=#{id}

</if>

<if test=*"username!=null and username!=''"*>

and username like '%${username}%'

</if>

</sql>

* 使用include引用：

<select id=*"findUserList"* parameterType=*"user"* resultType=*"user"*>

select \* from user

<where>

<include refid=*"query\_user\_where"*/>

</where>

</select>

注意：如果引用其它mapper.xml的sql片段，则在引用时需要加上namespace，如下：

<include refid=*"*namespace.sql片段”/>

# 模糊查询和分页查询

## 模糊查询

mapper

|  |
| --- |
| //模式查询List<User> findByLike(User user); |

### 在查询字段中添加%

mapper.xml

|  |
| --- |
| <!--模糊查询--><**select id="findByLike" resultType="com.sx.domain.User"**>  select *\** from user where username like #{username} </**select**> |

测试

|  |
| --- |
| User userLike = **new** User(); userLike.setUsername(**"%二%"**); List<User> byLike = userDao.findByLike(userLike); |

### 使用#实现

|  |
| --- |
| *<!--模糊查询-->* <**select id="findByLike" resultType="com.sx.domain.User"**>  select *\** from user where username like "%"#{username}"%" </**select**> |

测试

|  |
| --- |
| User userLike = **new** User(); userLike.setUsername(**"二"**); List<User> byLike = userDao.findByLike(userLike); |

### 使用$符号实现(有注入风险)

mapper.xml

|  |
| --- |
| <**select id="findByLike" resultType="com.sx.domain.User"**>  select *\** from user where username like "%${username}$" </**select**> |

测试

|  |
| --- |
| User userLike = **new** User(); userLike.setUsername(**"二"**); List<User> byLike = userDao.findByLike(userLike); |

### 使用concat函数实现

mapper.xml

|  |
| --- |
| <**select id="findByLike" resultType="com.sx.domain.User"**>  select *\** from user where username like concat("%",#{username},"%") </**select**> |

测试

|  |
| --- |
| User userLike = **new** User(); userLike.setUsername(**"二"**); List<User> byLike = userDao.findByLike(userLike); |

### 使用bind标签

|  |
| --- |
| <**select id="findByLike" resultType="com.sx.domain.User"**>  <**bind name="keywords" value="'%'+username+'%'"**></**bind**>  select *\** from user where username like #{keywords} </**select**> |

测试

|  |
| --- |
| User userLike = **new** User(); userLike.setUsername(**"二"**); List<User> byLike = userDao.findByLike(userLike); |

## 分页查询

### 方式一

分页的sql

mysql分页 select \* from 表名 limit (currentPage-1) \* pageSize,pageSize

mapper接口

|  |
| --- |
| //分页查询  List<Employee> selectList(Integer startIndex,Integer pageSize);  //查总记录数  **int** queryCount(); |

mapper配置文件

|  |
| --- |
| <select id=*"selectList"* resultType=*"Employee"*>  select \* from t\_employee limit #{param1},#{param2}  </select>  <!-- 查询返回int或Integer时，resultType不要省略 -->  <select id=*"queryCount"* resultType=*"java.lang.Integer"*>  select count(eid) from t\_employee  </select> |

测试

|  |
| --- |
| **int** currentPage = 2; **int** pageSize = 2 ; List<User> byPage = userDao.findByPage((currentPage - 1) \* pageSize, pageSize); **for** (**int** i = 0; i < byPage.size(); i++) {  System.***out***.println(byPage.get(i)); } |

### 方式二(使用map作为参数的传入)

sql

mysql分页 select \* from 表名 limit (currentPage-1) \* pageSize,pageSize

mapper文件

|  |
| --- |
| <**select id="findByPage1" resultType="com.sx.domain.User"**>  select *\** from user limit #{startIndex},#{pageSize} </**select**> |

测试

|  |
| --- |
| **int** currentPage = 2; **int** pageSize = 2 ; Map<String,Integer> map = **new** HashMap<String, Integer>(); map.put(**"startIndex"**,(currentPage-1)\*pageSize); map.put(**"pageSize"**,pageSize); List<User> byPage = userDao.findByPage1(map); |

### 使用mybatis的api

sql

select *\** from user

mapper

|  |
| --- |
| <**select id="findByPage" resultType="com.sx.domain.User"**>  select *\** from user </**select**> |

测试

|  |
| --- |
| **int** currentPage = 2; **int** pageSize = 2 ; RowBounds rowBounds = **new** RowBounds((currentPage-1)\*pageSize,pageSize); List<User> byPage = session.selectList(**"com.sx.mapper.IUserDao.findByPage"**, **null**, rowBounds); **for** (**int** i = 0; i < byPage.size(); i++) {  System.***out***.println(byPage.get(i)); } |

### 查询总条数

sql

select count(\*) from user

mapper接口

|  |
| --- |
| Integer queryCount(); |

mapper

|  |
| --- |
| <**select id="queryCount" resultType="java.lang.Integer"**>  select *count*(*\**) from user </**select**> |

测试

|  |
| --- |
| Integer totalCount= userDao.queryCount(); System.***out***.println(totalCount); |

### 使用pageBean方式

pageBean

|  |
| --- |
| **public class** PageBean {  **private** Integer **currentPage**;  **private** Integer **pageSize**;  **private** Integer **totalCount**;  **private** Integer **totalPage**;  **private** Integer **startIndex**;  **public void** setCurrentPage(Integer currentPage) {  **this**.**currentPage** = currentPage;  **this**.**startIndex** = (**this**.**currentPage**-1)\***this**.**pageSize**; }  **public void** setTotalCount(Integer totalCount) {  **this**.**totalCount** = totalCount;  **this**.**totalPage** = (**int**)Math.*ceil*(**this**.**totalCount**\*1.0/**this**.**pageSize**); } |

mapper

|  |
| --- |
| PageBean pageBean = **new** PageBean(); pageBean.setPageSize(2); pageBean.setCurrentPage(2); *//查询总条数* Integer totalCount = userDao.queryCount(); pageBean.setTotalCount(totalCount);  //查询数据 List<User> byPage = userDao.findByPage3(pageBean); |

mapper接口

|  |
| --- |
| *//分页查询* List<User> findByPage3(PageBean pageBean); |

mapper.xml

|  |
| --- |
| *<!--模糊查询-->* <**select id="findByPage3" resultType="com.sx.domain.User"**>  select *\** from user limit #{startIndex},#{pageSize} </**select**> |

## 分页加模糊查询

### mapper.xml

|  |
| --- |
| *<!--模糊查询+分页查询-->* <**select id="findByPage5" resultType="com.sx.domain.User"**>  select *\** from user  where username like "%"#{kewwords}"%" or address like "%"#{kewwords}"%"  limit #{startIndex},#{pageSize} </**select**> *<!--模糊查询+分页查询--> <!--查询总条数-->* <**select id="queryCount5" resultType="java.lang.Integer"**>  select *count*(*\**) from user  where username like "%"#{keywords}"%" or address like "%"#{keywords}"%" </**select**> |

### mapper接口

|  |
| --- |
| *//分页查询+模糊查询* List<User> findByPage5(Map<String,Object> map); *//查询总条数* Integer queryCount5(String keywords); |

### 测试

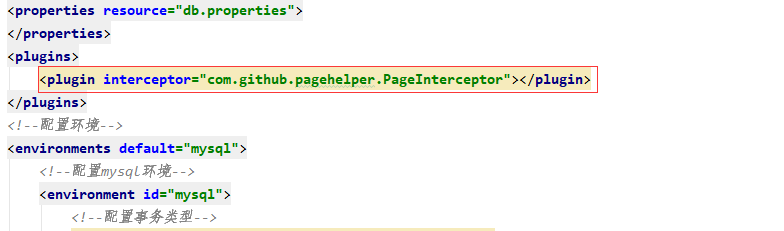
|  |
| --- |
| String keywords = **"二"**; PageBean pageBean = **new** PageBean(); pageBean.setPageSize(2); pageBean.setCurrentPage(1); *//模糊查询查询总条数* Integer totalCount = userDao.queryCount5(keywords); pageBean.setTotalCount(totalCount); *//查询数据* Map<String, Object> map = **new** HashMap<String, Object>(); map.put(**"startIndex"**,pageBean.getStartIndex()); map.put(**"pageSize"**,pageBean.getPageSize()); map.put(**"kewwords"**,keywords); List<User> byPage = userDao.findByPage5(map); **for** (**int** i = 0; i < byPage.size(); i++) {  System.***out***.println(byPage.get(i)); } |

## 阿里PageHelper

### 导入maven

|  |
| --- |
| <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper</**artifactId**>  <**version**>5.1.6</**version**> </**dependency**> <**dependency**>  <**groupId**>com.github.jsqlparser</**groupId**>  <**artifactId**>jsqlparser</**artifactId**>  <**version**>1.3</**version**> </**dependency**> |

### 修改核心配置文件sqlMapConfig.xml



### 测试

public static <E> Page<E> startPage(int pageNum, int pageSize, boolean count)

pageNum:当前页

pageSize:当前页大小

count:是否支持查询数量

|  |
| --- |
| PageBean pageBean = **new** PageBean(); pageBean.setPageSize(2); pageBean.setCurrentPage(1); *//查询数据* Page<User> page  = PageHelper.*startPage*(pageBean.getCurrentPage(),pageBean.getPageSize()); List<User> byPage = userDao.findByPage6(); *//获取查询的总条数* **long** total = page.getTotal(); **int** pages = page.getPages(); **for** (**int** i = 0; i < byPage.size(); i++) {  System.***out***.println(byPage.get(i)); } |

# mybatis事务

# 缓存

## mybatis缓存介绍

### mybatis的缓存

如下图，是mybatis一级缓存和二级缓存的区别图解：

sqlSession1

sqlSession 2

sqlSession3...

一级缓存

一级缓存

一级缓存

Mapper(namespace)二级缓存

Mybatis一级缓存的作用域是同一个SqlSession，在同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。Mybatis默认开启一级缓存。

Mybatis二级缓存是多个SqlSession共享的，其作用域是mapper的同一个namespace，不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis默认没有开启二级缓存需要在setting全局参数中配置开启二级缓存。

### 缓存的使用场景

1、适合

1、经常查询并且不经常改变的。

2、数据的正确与否对最终的结果影响不大。

2、不适合

1、经常改变的数据

2、数据的正确与否对最终的结果影响不大。

## 一级缓存

### 原理

下图是根据id查询用户的一级缓存图解：

第一次查询id为1的用户

SqlSession

一级缓存区域

Map

SqlSession

写入

修改、添加、删除用户执行commit

清空

第二次查询id为1的用户

读取

一级缓存区域是根据SqlSession为单位划分的。

每次查询会先从缓存区域找，如果找不到从数据库查询，查询到数据将数据写入缓存。

Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象

sqlSession执行insert、update、delete等操作commit提交后会清空缓存区域。

### 测试1

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//第一次查询

User user1 = userMapper.findUserById(1);

System.*out*.println(user1);

//第二次查询，由于是同一个session则不再向数据发出语句直接从缓存取出

User user2 = userMapper.findUserById(1);

System.*out*.println(user2);

//关闭session

session.close();

### 测试2

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

UserMapper userMapper = session.getMapper(UserMapper.**class**);

//第一次查询

User user1 = userMapper.findUserById(1);

System.*out*.println(user1);

//在同一个session执行更新

User user\_update = **new User()**;

user\_update.setId(1);

user\_update.setUsername("李奎");

userMapper.updateUser(user\_update);

session.commit();

//第二次查询，虽然是同一个session但是由于执行了更新操作session的缓存被清空，这里重新发出sql操作

User user2 = userMapper.findUserById(1);

System.*out*.println(user2);

## 二级缓存

### 原理

下图是多个sqlSession请求UserMapper的二级缓存图解。

UserMapper

sqlSession1

执行UserMapper查询

sqlSession2

执行UserMapper查询

sqlSession3..

执行UserMapper提交（插入、更新、删除）

UserMapper的二级缓存区域

写入

读取

清空

它指的是Mybatis中SqlSessionFac tory对象的缓存。由同一个SqlSessionFactory对象仓建的SqlSession共享其缓存。

二级缓存区域是根据mapper的namespace划分的，相同namespace的mapper查询数据放在同一个区域，如果使用mapper代理方法每个mapper的namespace都不同，此时可以理解为二级缓存区域是根据mapper划分。

每次查询会先从缓存区域找，如果找不到从数据库查询，查询到数据将数据写入缓存。

Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象

sqlSession执行insert、update、delete等操作commit提交后会清空缓存区域。

### 开启二级缓存：

使用步骤

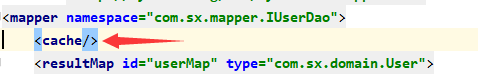
第一步:让Mybatis框架支持二级缓存(在SqlMapConfig. xml中配置)

<settings>

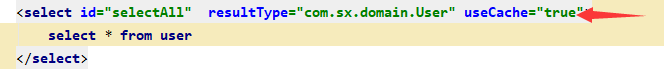
<setting name="cacheEnabled" value="true"/>

</settings>

第二步:让当前的映射文件支持二级缓存(在IUserDao.xml中配置)



第三步:让当前的操作支持二级缓存(在select标签中配置)



在核心配置文件SqlMapConfig.xml中加入

<setting name=*"cacheEnabled"* value=*"true"*/>

|  |  |  |  |
| --- | --- | --- | --- |
|  | 描述 | 允许值 | 默认值 |
| cacheEnabled | *对在此配置文件下的所有cache 进行全局性开/关设置。* | true false | true |

要在你的Mapper映射文件中添加一行：  <cache /> ，表示此mapper开启二级缓存。

### 实现序列化

二级缓存需要查询结果映射的pojo对象实现java.io.Serializable接口实现序列化和反序列化操作，注意如果存在父类、成员pojo都需要实现序列化接口。

public class Orders implements Serializable

public class User implements Serializable

....

### 测试

//获取session1

SqlSession session1 = sqlSessionFactory.openSession();

UserMapper userMapper = session1.getMapper(UserMapper.**class**);

//使用session1执行第一次查询

User user1 = userMapper.findUserById(1);

System.*out*.println(user1);

//关闭session1

session1.close();

//获取session2

SqlSession session2 = sqlSessionFactory.openSession();

UserMapper userMapper2 = session2.getMapper(UserMapper.**class**);

//使用session2执行第二次查询，由于开启了二级缓存这里从缓存中获取数据不再向数据库发出sql

User user2 = userMapper2.findUserById(1);

System.*out*.println(user2);

//关闭session2

session2.close();

### 禁用二级缓存

在statement中设置useCache=false可以禁用当前select语句的二级缓存，即每次查询都会发出sql去查询，默认情况是true，即该sql使用二级缓存。

<select id="findOrderListResultMap" resultMap="ordersUserMap" useCache="false">

### 刷新缓存

在mapper的同一个namespace中，如果有其它insert、update、delete操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

 设置statement配置中的flushCache="true" 属性，默认情况下为true即刷新缓存，如果改成false则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

如下：

<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User" flushCache="true">

### Mybatis Cache参数

flushInterval（刷新间隔）可以被设置为任意的正整数，而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。

size（引用数目）可以被设置为任意正整数，要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是1024。

readOnly（只读）属性可以被设置为true或false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false。

如下例子：

<cache  eviction="FIFO"  flushInterval="60000"  size="512"  readOnly="true"/>

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会导致冲突。可用的收回策略有, 默认的是 LRU:

1. LRU – 最近最少使用的:移除最长时间不被使用的对象。
2. FIFO – 先进先出:按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
4. WEAK – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

### mybatis整合ehcache

EhCache 是一个纯Java的进程内缓存框架，是一种广泛使用的开源Java分布式缓存，具有快速、精干等特点，是Hibernate中默认的CacheProvider。

## mybatis整合ehcache原理

mybatis提供二级缓存Cache接口，如下：



它的默认实现类：



通过实现Cache接口可以实现mybatis缓存数据通过其它缓存数据库整合，mybatis的特长是sql操作，缓存数据的管理不是mybatis的特长，为了提高缓存的性能将mybatis和第三方的缓存数据库整合，比如ehcache、memcache、redis等。

### 第一步：引入缓存的依赖包

JOZ58Q@U{X_}F`EN}BLVU1A

maven坐标：

<dependency>

<groupId>org.mybatis.caches</groupId>

<artifactId>mybatis-ehcache</artifactId>

<version>1.0.2</version>

</dependency>

### 第二步：引入缓存配置文件

classpath下添加：ehcache.xml

内容如下：

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

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

<diskStore path=*"F:\develop\ehcache"* />

<defaultCache

maxElementsInMemory=*"1000"*

maxElementsOnDisk=*"10000000"*

eternal=*"false"*

overflowToDisk=*"false"*

timeToIdleSeconds=*"120"*

timeToLiveSeconds=*"120"*

diskExpiryThreadIntervalSeconds=*"120"*

memoryStoreEvictionPolicy=*"LRU"*>

</defaultCache>

</ehcache>

属性说明：

diskStore：指定数据在磁盘中的存储位置。

defaultCache：当借助CacheManager.add("demoCache")创建Cache时，EhCache便会采用<defalutCache/>指定的的管理策略

以下属性是必须的：

maxElementsInMemory - 在内存中缓存的element的最大数目

maxElementsOnDisk - 在磁盘上缓存的element的最大数目，若是0表示无穷大

eternal - 设定缓存的elements是否永远不过期。如果为true，则缓存的数据始终有效，如果为false那么还要根据timeToIdleSeconds，timeToLiveSeconds判断

overflowToDisk - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上

以下属性是可选的：

timeToIdleSeconds - 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时，这些数据便会删除，默认值是0,也就是可闲置时间无穷大

timeToLiveSeconds - 缓存element的有效生命期，默认是0.,也就是element存活时间无穷大

diskSpoolBufferSizeMB 这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一个缓冲区.

diskPersistent - 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。

diskExpiryThreadIntervalSeconds - 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s，相应的线程会进行一次EhCache中数据的清理工作

memoryStoreEvictionPolicy - 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出）

### 第三步：开启ehcache缓存

EhcacheCache是ehcache对Cache接口的实现：



修改mapper.xml文件，在cache中指定EhcacheCache。

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

根据需求调整缓存参数：

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

<property name=*"timeToIdleSeconds"* value=*"3600"*/>

<property name=*"timeToLiveSeconds"* value=*"3600"*/>

<!-- 同ehcache参数maxElementsInMemory -->

<property name=*"maxEntriesLocalHeap"* value=*"1000"*/>

<!-- 同ehcache参数maxElementsOnDisk -->

<property name=*"maxEntriesLocalDisk"* value=*"10000000"*/>

<property name=*"memoryStoreEvictionPolicy"* value=*"LRU"*/>

</cache>

## 应用场景

对于访问多的查询请求且用户对查询结果实时性要求不高，此时可采用mybatis二级缓存技术降低数据库访问量，提高访问速度，业务场景比如：耗时较高的统计分析sql、电话账单查询sql等。

实现方法如下：通过设置刷新间隔时间，由mybatis每隔一段时间自动清空缓存，根据数据变化频率设置缓存刷新间隔flushInterval，比如设置为30分钟、60分钟、24小时等，根据需求而定。

## 局限性

mybatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用mybatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybaits的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

# mybatis注解开发

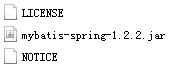
## 环境搭建

# 与spring整合

实现mybatis与spring进行整合，通过spring管理SqlSessionFactory、mapper接口。

## mybatis与spring整合jar

mybatis官方提供与mybatis与spring整合jar包：



还包括其它jar：

spring3.2.0

mybatis3.2.7

dbcp连接池

数据库驱动

参考：



## Mybatis配置文件

在classpath下创建*mybatis/SqlMapConfig.xml*

<?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>

<!—使用自动扫描器时，mapper.xml文件如果和mapper.java接口在一个目录则此处不用定义mappers -->

<mappers>

<package name="cn.itcast.mybatis.mapper" />

</mappers>

</configuration>

## Spring配置文件：

在classpath下创建applicationContext.xml，定义数据库链接池、SqlSessionFactory。

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd "*>

<!-- 加载配置文件 -->

<context:property-placeholder location=*"classpath:db.properties"*/>

<!-- 数据库连接池 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"${jdbc.driver}"*/>

<property name=*"url"* value=*"${jdbc.url}"*/>

<property name=*"username"* value=*"${jdbc.username}"*/>

<property name=*"password"* value=*"${jdbc.password}"*/>

<property name=*"maxActive"* value=*"10"*/>

<property name=*"maxIdle"* value=*"5"*/>

</bean>

<!-- mapper配置 -->

<!-- 让spring管理sqlsessionfactory 使用mybatis和spring整合包中的 -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<!-- 数据库连接池 -->

<property name=*"dataSource"* ref=*"dataSource"* />

<!-- 加载mybatis的全局配置文件 -->

<property name=*"configLocation"* value=*"classpath:**mybatis/SqlMapConfig.xml"* />

</bean>

</beans>

注意：在定义sqlSessionFactory时指定数据源dataSource和mybatis的配置文件。

## Mapper编写的三种方法

### Dao接口实现类继承SqlSessionDaoSupport

使用此种方法即原始dao开发方法，需要编写dao接口，dao接口实现类、映射文件。

1. 在sqlMapConfig.xml中配置映射文件的位置

<mappers>

<mapper resource=*"mapper.xml文件的地址"* />

<mapper resource=*"mapper.xml文件的地址"* />

</mappers>

1. 定义dao接口
2. dao接口实现类集成SqlSessionDaoSupport

dao接口实现类方法中可以this.getSqlSession()进行数据增删改查。

1. spring 配置

<bean id=*" "*class=*"mapper接口的实现"*>

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*></property>

</bean>

### 使用org.mybatis.spring.mapper.MapperFactoryBean

此方法即mapper接口开发方法，只需定义mapper接口，不用编写mapper接口实现类。每个mapper接口都需要在spring配置文件中定义。

1. 在sqlMapConfig.xml中配置mapper.xml的位置

如果mapper.xml和mappre接口的名称相同且在同一个目录，这里可以不用配置

<mappers>

<mapper resource=*"mapper.xml文件的地址"* />

<mapper resource=*"mapper.xml文件的地址"* />

</mappers>

1. 定义mapper接口
2. Spring中定义

<bean id=*""* class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>

<property name=*"mapperInterface"* value=*"mapper接口地址"*/>

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

</bean>

### 使用mapper扫描器

此方法即mapper接口开发方法，只需定义mapper接口，不用编写mapper接口实现类。只需要在spring配置文件中定义一个mapper扫描器，自动扫描包中的mapper接口生成代代理对象。

1. mapper.xml文件编写，
2. 定义mapper接口

**注意mapper.xml的文件名和mapper的接口名称保持一致，且放在同一个目录**

1. 配置mapper扫描器

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<property name=*"basePackage"* value=*"mapper接口包地址"*></property>

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"*/>

</bean>

*basePackage：扫描包路径，中间可以用逗号或分号分隔定义多个包*

1. 使用扫描器后从spring容器中获取mapper的实现对象

**如果将mapper.xml和mapper接口的名称保持一致且放在一个目录 则不用在sqlMapConfig.xml中进行配置**

# Mybatis逆向工程

使用官方网站的mapper自动生成工具mybatis-generator-core-1.3.2来生成po类和mapper映射文件。

## 第一步：mapper生成配置文件：

在generatorConfig.xml中配置mapper生成的详细信息，注意改下几点：

1. 添加要生成的数据库表
2. po文件所在包路径
3. mapper文件所在包路径

配置文件如下：

详见generatorSqlmapCustom工程

## 第二步：使用java类生成mapper文件：

**Public void** generator() **throws** Exception{

List<String> warnings = **new** ArrayList<String>();

**boolean** overwrite = **true**;

File configFile = **new** File("generatorConfig.xml");

ConfigurationParser cp = **new** ConfigurationParser(warnings);

Configuration config = cp.parseConfiguration(configFile);

DefaultShellCallback callback = new DefaultShellCallback(overwrite);

MyBatisGenerator myBatisGenerator = **new** MyBatisGenerator(config,

callback, warnings);

myBatisGenerator.generate(**null**);

}

**Public static void** main(String[] args) **throws** Exception {

**try** {

GeneratorSqlmap generatorSqlmap = **new** GeneratorSqlmap();

generatorSqlmap.generator();

} **catch** (Exception e) {

e.printStackTrace();

}

}

## 第三步：拷贝生成的mapper文件到工程中指定的目录中

### Mapper.xml

Mapper.xml的文件拷贝至mapper目录内

### Mapper.java

Mapper.java的文件拷贝至mapper 目录内

注意：mapper xml文件和mapper.java文件在一个目录内且文件名相同。

### 第四步Mapper接口测试

学会使用mapper自动生成的增、删、改、查方法。

//删除符合条件的记录

**int** deleteByExample(UserExample example);

//根据主键删除

**int** deleteByPrimaryKey(String id);

//插入对象所有字段

**int** insert(User record);

//插入对象不为空的字段

**int** insertSelective(User record);

//自定义查询条件查询结果集

List<User> selectByExample(UserExample example);

//根据主键查询

UserselectByPrimaryKey(String id);

//根据主键将对象中不为空的值更新至数据库

**int** updateByPrimaryKeySelective(User record);

//根据主键将对象中所有字段的值更新至数据库

**int** updateByPrimaryKey(User record);

## 逆向工程注意事项

### Mapper文件内容不覆盖而是追加

XXXMapper.xml文件已经存在时，如果进行重新生成则mapper.xml文件内容不被覆盖而是进行内容追加，结果导致mybatis解析失败。

解决方法：删除原来已经生成的mapper xml文件再进行生成。

Mybatis自动生成的po及mapper.java文件不是内容而是直接覆盖没有此问题。

### Table schema问题

下边是关于针对oracle数据库表生成代码的schema问题：

Schma即数据库模式，oracle中一个用户对应一个schema，可以理解为用户就是schema。

当Oralce数据库存在多个schema可以访问相同的表名时，使用mybatis生成该表的mapper.xml将会出现mapper.xml内容重复的问题，结果导致mybatis解析错误。

解决方法：在table中填写schema，如下：

<table schema="XXXX" tableName=" " >

XXXX即为一个schema的名称，生成后将mapper.xml的schema前缀批量去掉，如果不去掉当oracle用户变更了sql语句将查询失败。

快捷操作方式：mapper.xml文件中批量替换：“from XXXX.”为空

Oracle查询对象的schema可从dba\_objects中查询，如下：

select \* from dba\_objects