# Mybatis需要关注的文件

* POJO类（DTO、VO、PO等）
* Mapper接口（就是dao接口）
* Mapper映射文件（**重点**）
* 全局配置文件

# 框架概述

## 什么是框架

可以说，一个框架是一个**可复用**的设计构件，它规定了应用的体系结构，阐明了整个设计、协作构件之间的依赖关系、责任分配和控制流程，表现为一组抽象类以及其实例之间协作的方法，它为构件复用提供了上下文(Context)关系。因此构件库的大规模重用也需要框架。

## 为什么使用框架

\* 因为**软件系统**发展到今天**已经很复杂了**，特别是**服务器端软件**，涉及到的知识，内容，问题太多。

\* 在某些方面使用别人成熟的**框架**，**就相当于让别人帮你完成一些基础工作**，你**只需要集中精力完成系统的业务逻辑设计**。

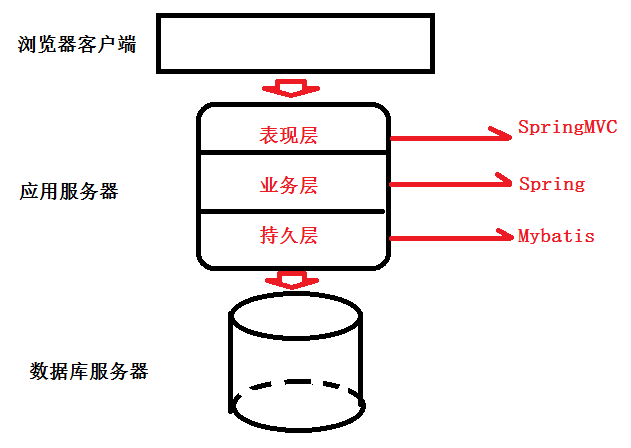
\* 而且**框架**一般是成熟，稳健的**，它可以处理系统很多细节问题**，比如，事务处理，安全性，数据流控制等问题。

\* 还有**框架**一般都经过很多人使用，所以结构很好，所以扩展性也很好，而且它**是**不断升级的，你**可以直接享受别人升级代码带来的好处**。

## 软件开发的三层结构

我们用三层结构主要是使项目结构更清楚，分工更明确，有利于后期的维护和升级．

三层结构包含：表现层，业务层，持久层



# Mybatis介绍

## 什么是mybatis

mybatis参考网址：<http://www.mybatis.org/mybatis-3/zh/index.html>

* **Mybatis是什么？**

\* MyBatis是一个优秀的**持久层框架**,它是一个**半自动化的ORM框架**

\* 它**对**使用**JDBC**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。

* **Mybatis的由来？**

\* MyBatis 本是apache的一个开源项目iBatis

\* 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。

\* 2013年11月迁移到Github。

* **Mybatis在哪写SQL语句？如何设置SQL参数和封装查询结果？**

\* 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。

## JDBC示例代码

|  |
| --- |
| **public** **class** JDBCTest {  **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  connection = DriverManager.*getConnection*(  "jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8",  "root",  "root"  );  // 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  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  **if** (resultSet != **null**) {  **try** {  resultSet.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **if** (preparedStatement != **null**) {  **try** {  preparedStatement.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **if** (connection != **null**) {  **try** {  connection.close();  } **catch** (SQLException e) {  // **TODO** Auto-generated catch block e.printStackTrace();  }  }  }  }  } |

## Mybatis如何解决JDBC代码中存在的问题

1、 创建数据库连接相关操作，存在**硬编码**

a) 解决方案：通过Mybatis**全局配置文件**，对数据库连接进行配置

2、 statement相关操作，存在**硬编码**

a) 解决方案：通过**Mapper映射文件**，对statement相关处理进行配置。

3、 频繁开启数据库连接，会降低数据库处理性能。

a) 解决方案：通过Mybatis全局配置文件，配置**连接池**。

## Mybatis架构原理

我们要找出的答案就是：**从XML配置文件到数据库的距离有多长？**



**说明：**

1. **mybatis配置文件**

a) SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

b) Mapper.xml，此文件作为mybatis的sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

1. **SqlSessionFactory**

通过mybatis环境等配置信息构造SqlSessionFactory，即会话工厂。

1. **sqlSession**

通过会话工厂创建sqlSession即会话，程序员通过sqlsession会话接口对数据库进行增删改查操作。

1. **Executor执行器**

mybatis底层自定义了Executor执行器接口来具体操作数据库，Executor接口有两个实现，一个是基本执行器（默认）、一个是缓存执行器，sqlsession底层是通过executor接口操作数据库的。

1. **Mapped Statement**

它也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个**select\insert\update\delete标签对应一个Mapped Statement对象**，select\insert\update\delete标签的id即是Mapped statement的id。

* 1. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
  2. Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

# Mybatis入门

## 需求

1、根据用户id查询一个用户信息

2、根据用户名称模糊查询用户信息列表

3、添加用户

4、更新用户

5、删除用户

## Mybatis开发框架搭建

* 创建maven工程：mybatis
* POM文件

|  |
| --- |
| <dependencies>  <!-- mybatis依赖 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.6</version>  </dependency>  <!-- mysql依赖 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.35</version>  </dependency>  <!-- 单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  </dependencies> |

* 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>  <properties resource=*"db.properties"*></properties>  <environments default=*"development"*>  <environment id=*"development"*>  <transactionManager type=*"JDBC"* />  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"${db.driver}"* />  <property name=*"url"* value=*"${db.url}"* />  <property name=*"username"* value=*"${db.username}"* />  <property name=*"password"* value=*"${db.password}"* />  </dataSource>  </environment>  </environments>  <mappers>  <mapper resource=*"UserMapper.xml"* />  </mappers>  </configuration> |

* UserMapper.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> |

* PO类

|  |
| --- |
| **public** **class** User {  **private** **int** id;  **private** String username;  **private** Date birthday;  **private** String sex;  **private** String address;  //getter\setter方法  } |

## 需求实现

### 根据id查询用户信息

* 映射文件

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"com.kkb.mybatis.po.User"*>

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

</select>

**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**：定义结果映射类型。

**#{}：**相当于JDBC中的？占位符

* 测试代码

**public** **class** MybatisFirst {

//会话工厂

**private** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** createSqlSessionFactory() **throws** IOException {

// 配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

// 使用SqlSessionFactoryBuilder从xml配置文件中创建SqlSessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

// 根据 id查询用户信息

@Test

**public** **void** testFindUserById() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 查询单个记录，根据用户id查询用户信息

User user = sqlSession.selectOne("test.findUserById", 10);

// 输出用户信息

System.out.println(user);

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

}

### 根据用户名模糊查询用户列表

* 映射文件

<!-- 自定义条件查询用户列表 -->

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

resultType=*"****com.kkb.mybatis.po.User****"*>

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

</select>

parameterType：定义输入到sql中的映射类型，${**value**}表示输入参数将${value}替换，做字符串的拼接。

注意：如果是取**简单数量类型**的参数，括号中的**参数名称必须为value**

**resultType**：定义结果映射类型。

* 测试代码

// 根据用户名称模糊查询用户信息

@Test

**public** **void** testFindUserByUsername() {

// 数据库会话实例

SqlSession sqlSession = **null**;

**try** {

// 创建数据库会话实例sqlSession

sqlSession = sqlSessionFactory.openSession();

// 查询单个记录，根据用户id查询用户信息

List<User> list = sqlSession.selectList("test.findUserByUsername", "小明");

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

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

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

sqlSession.close();

}

}

}

### #{}和${}区别

* 区别1

#{} ：相当于JDBC SQL语句中的占位符**?** (PreparedStatement)

${} : 相当于JDBC SQL语句中的连接符合 + (Statement)

* 区别2

#{} ： 进行输入映射的时候，会对参数进行类型解析（如果是String类型，那么SQL语句会自动加上’’）

${} :进行输入映射的时候，将参数原样输出到SQL语句中

* 区别3

#{} ： 如果进行简单类型（String、Date、8种基本类型的包装类）的输入映射时，#{}中参数名称可以任意

${} : 如果进行简单类型（String、Date、8种基本类型的包装类）的输入映射时，${}中参数名称必须是**value**

${} :存在**SQL注入**问题 ，使用OR 1=1 关键字将查询条件忽略

### 添加用户

* 映射文件

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"****com.kkb.mybatis.po.User****"*>

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

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

</insert>

* 测试代码

// 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();

}

}

}

* 主键返回

<insert id=*"insertUser"* parameterType=*"com.kkb.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标签实现主键返回。

**\* keyProperty**:指定返回的主键，存储在pojo中的哪个属性

**\* order**：selectKey标签中的sql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**:返回的主键对应的JAVA类型

**\* LAST\_INSERT\_ID()**:是mysql的函数，返回auto\_increment自增列新记录id值。

### OGNL

对象导航图语言

|---User（参数值对象）

|--username--张三

|--birthday

|--sex--男

|--**dept** -- Department

|--**name**

|--no

OGNL表达式去获取Department对象的name属性：dept.name

### 删除用户

* 映射文件

<!-- 删除用户 -->

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

delete from user where id=#{id}

</delete>

* 测试代码

// 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();

}

}

}

### 修改用户

* 映射文件

<!-- 更新用户 -->

<update id=*"updateUser"* parameterType=*"com.kkb.mybatis.po.User"*>

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

where id=#{id}

</update>

* 测试代码

// 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开发Dao层

## 原始dao开发方式

**说明：该种方式一般来说只会出现在那些从ibatis项目迁移过来的项目。**

* 生命周期（作用范围）

1. sqlsession：方法级别
2. sqlsessionFactory：全局范围（应用级别）
3. sqlsessionFactoryBuilder：方法级别

* dao接口和实现类

**public interface** UserDao {

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

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

}

**public class** UserDaoImpl **implements** UserDao {

//注入SqlSessionFactory

**public** UserDaoImpl(SqlSessionFactory sqlSessionFactory){

**this**. sqlSessionFactory = sqlSessionFactory;

}

**private** SqlSessionFactory sqlSessionFactory;

@Override

**public User** findUserById(**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", id);

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();

}

}

}

* 映射文件

<?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=*"com.kkb.mybatis.po.User"*>

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

</select>

<!-- 添加用户 -->

<insert id=*"insertUser"* parameterType=*"com.kkb.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>

* 测试代码

**private** SqlSessionFactory sqlSessionFactory;

@Before

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

SqlSessionFactoryBuilder sessionFactoryBuilder = **new** SqlSessionFactoryBuilder();

InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml");

sqlSessionFactory = sessionFactoryBuilder.build(inputStream);

}

@Test

**public** **void** testFindUserById() {

UserDao userDao = **new** UserDaoImpl(sqlSessionFactory);

User user = userDao.findUserById(22);

System.***out***.println(user);

}

}

## mapper代理开发方式（JDK的代理方式）

### 代理

动态代理分为两种方式：

基于JDK的动态代理--针对有**接口的类**进行动态代理

基于CGLIB的动态代理--通过**子类**继承**父类**的方式去进行代理。

### XML方式

* 使用

只需要开发Mapper接口（dao接口）和Mapper约束文件，不需要编写实现类。

* 开发规范

Mapper接口开发需要遵循以下规范：

1、 Mapper接口的类路径与Mapper.xml文件中的namespace相同。

2、 Mapper接口方法名称和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接口方法的返回值类型和mapper.xml中定义的每个sql的resultType的类型相同。

* mapper映射文件

<?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.kkb.mybatis.mapper.UserMapper"*>

<!-- 根据id获取用户信息 -->

<select id=*"findUserById"* parameterType=*"int"* resultType=*"com.kkb.mybatis.po.User"*>

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

</select>

</mapper>

* mapper接口

/\*\*

\* 用户管理mapper

\*/

**public interface** UserMapper {

//根据用户id查询用户信息

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

}

* 加载映射文件

<!-- 加载映射文件 -->

<mappers>

<mapper resource=*"mapper/UserMapper.xml"*/>

</mappers>

* 测试代码

**public class** UserMapperTest{

**private** SqlSessionFactory sqlSessionFactory;

@Before

p**ublic void** setUp() **throws** Exception {

//mybatis配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

//使用SqlSessionFactoryBuilder创建sessionFactory

sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

@Test

p**ublic void** testFindUserById() **throws** Exception {

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获取mapper接口的代理对象

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

//调用代理对象方法

User user = userMapper.findUserById(1);

System.out.println(user);

//关闭session

session.close();

}

}

### 注解方式（后续讲解）

# 全局配置文件

## 配置内容

SqlMapConfig.xml中配置的内容和顺序如下：

**properties**（属性）

**settings**（全局配置参数）

**typeAliases**（类型别名）

typeHandlers（类型处理器）--**Java类型**--JDBC类型--->数据库类型转换

objectFactory（对象工厂）

plugins（插件）--可以在Mybatis执行SQL语句的流程中，横叉一脚去实现一些功能增强，比如PageHelper分页插件，就是第三方实现的一个插件

environments（环境集合属性对象）

environment（环境子属性对象）

transactionManager（事务管理）

dataSource（数据源）

**mappers**（映射器）

## properties标签

SqlMapConfig.xml可以引用java属性文件中的配置信息。

1、在classpath下定义db.properties文件，

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/**ssm**?characterEncoding=utf-8

jdbc.username=**root**

jdbc.password=**root**

2、在SqlMapConfig.xml文件中，引用db.properties中的属性，具体如下：

<properties resource=*"**db.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>

properties标签除了可以使用resource属性，引用properties文件中的属性。还可以在properties标签内定义property子标签来定义属性和属性值，具体如下：

<properties>

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

</properties>

**注意： MyBatis 将按照下面的顺序来加载属性**：

* 读取properties 元素体内定义的属性。
* 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。

## typeAlias标签

**别名的作用**：就是为了简化映射文件中parameterType和ResultType中的POJO类型名称编写。

### 默认支持别名

|  |  |
| --- | --- |
| 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 |
| map | Map |

### 自定义别名

在SqlMapConfig.xml中进行如下配置：

<typeAliases>

<!-- **单个别名定义** -->

<typeAlias alias=*"user"* type=*"com.kkb.mybatis.po.User"*/>

<!-- **批量别名定义**，扫描整个包下的类，别名为类名（首字母大写或小写都可以） -->

<package name=*"com.kkb.mybatis.po"*/>

</typeAliases>

## mappers标签

### <mapper resource=""/>

使用相对于类路径的资源

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

### <mapper url="">

使用绝对路径加载资源

如：<mapper url="file://d:/sqlmap/User.xml" />

### <mapper class=""/>

使用mapper接口类路径，加载映射文件。

如：<mapper class="com.kkb.mybatis.mapper.UserMapper"/>

**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。**

### <package name=""/>

注册指定包下的所有mapper接口，来加载映射文件。

如：<package name="com.kkb.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。**

# 输入映射和输出映射

## parameterType(输入类型)

* parameterType属性可以映射的输入参数类型有：

**简单类型、POJO类型、Map类型、List类型（数组）**。

* Map类型和POJO类型的用法类似，本课程只讲POJO类型的相关配置。
* List类型在动态SQL部分进行讲解。

### 传递简单类型

参考入门案例中的根据用户ID查询用户信息的案例。

### 传递pojo对象

* 参考入门案例中的添加用户的案例。

### 传递pojo包装对象

* 包装对象：pojo类中包含pojo。
* **#{}:是通过反射获取数据的---StaticSqlSource**
* **${}：是通过OGNL表达式会随着对象的嵌套而相应的发生层级变化 --DynamicSqlSource**

#### 需求

* 在综合查询的场景中，需要根据不同的查询条件进行用户列表的查询。
* 这些查询条件中，可能会包括用户信息中的用户名称，还可能包括商品信息的商品名称。

#### 分析

1. 通过mybatis传递多个查询条件时，需要使用pojo或者map来传递。
2. 通过pojo传递参数的话，需要定义一个包装对象，该对象包括用户数据或者商品数据。

#### QueryVO

定义包装对象QueryVO

|  |
| --- |
| **public** **class** QueryVO {  **private** User user;  } |

#### SQL语句

SELECT \* FROM user where username like '%小明%'

#### Mapper文件

|  |
| --- |
| <!-- 使用包装类型查询用户  使用ognl从对象中取属性值，如果是包装对象可以使用.操作符来取内容部的属性  -->  <select id=*"findUserList"* parameterType=*"queryVo"* resultType=*"user"*>  SELECT \* FROM user where username like '%${**user.username**}%'  </select> |

#### Mapper接口

/\*\*

\* 用户管理mapper

\*/

**public interface** UserMapper {

//根据用户id查询用户信息

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

//查询用户列表

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

//添加用户信息

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

//综合查询用户列表

**public** List<User> findUserList(QueryVo queryVo)**throws** Exception;

}

#### 测试方法

在UserMapperTest测试类中，添加以下测试代码：

|  |
| --- |
| @Test  **public** **void** testFindUserList() **throws** Exception {  SqlSession sqlSession = sqlSessionFactory.openSession();  //获得mapper的代理对象  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  //创建QueryVo对象  QueryVo queryVo = **new** QueryVo();  //创建user对象  User user = **new** User();  user.setUsername("小明");  queryVo.setUser(user);  //根据queryvo查询用户  List<User> list = userMapper.findUserList(queryVo);  System.*out*.println(list);  sqlSession.close();  } |

## resultType(输出类型)

resultType属性可以映射的java类型有：

**简单类型、POJO类型、Map类型**。

不过Map类型和POJO类型的使用情况类型，所以只需讲解POJO类型即可。

### 使用要求

使用resultType进行输出映射时，要求sql语句中**查询的列名**和要映射的**pojo的属性名**一致。

### 映射简单类型

#### 需求

在综合查询用户列表时，要求分页查询，而分页查询需要查询符合条件的记录总数。

#### Mapper映射文件

<!-- 获取用户列表总数 -->

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

select count(1) from user where username like '%${user.username}%'

</select>

#### Mapper接口

//综合查询用户总数

**public** **int** findUserCount(QueryVo queryVo) **throws** Exception;

#### 测试代码

在UserMapperTest测试类中，添加以下测试代码：

|  |
| --- |
| @Test  **public** **void** testFindUserList() **throws** Exception {  SqlSession sqlSession = sessionFactory.openSession();  //获得mapper的代理对象  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  //创建QueryVo对象  QueryVo queryVo = **new** QueryVo();  //创建user对象  User user = **new** User();  user.setUsername("小明");  queryVo.setUser(user);  //根据queryvo查询用户  List<User> list = userMapper.findUserList(queryVo);  System.*out*.println(list);  int count = userMapper.findUserCount(queryVo);  System.*out*.println(count);  sqlSession.close();  } |

**注意：输出简单类型必须查询出来的结果集只有一列。**

### 映射pojo对象

**注意：不管是单个POJO还是POJO集合，在使用resultType完成映射时，用法一样。**

参考入门程序之根据用户ID查询用户信息和根据名称模糊查询用户列表的案例

## resultMap

### 使用要求

如果sql查询列名和pojo的属性名不一致，可以通过resultMap将列名和属性名作一个对应关系，最终将查询结果映射到指定的pojo对象中。

注意：resultType底层也是通过resultMap完成映射的。

### 需求

将以下sql的查询结果进行映射：

SELECT id id\_,username username\_,birthday birthday\_ FROM user

### Mapper接口

|  |
| --- |
| // resultMap入门  **public** List<User> findUserListResultMap() **throws** Exception; |

### Mapper映射文件

由于sql查询列名和Users类属性名不一致，所以不能使用resultType进行结构映射。

需要定义一个resultMap将sql查询列名和Users类的属性名对应起来，完成结果映射。

|  |
| --- |
| <!-- 定义resultMap：将查询的列名和映射的pojo的属性名做一个对应关系 -->  <!--  type：指定查询结果要映射的pojo的类型  id：指定resultMap的唯一标示  -->  <resultMap type=*"user"* id=*"userListResultMap"*>  <!--  id标签：映射查询结果的唯一列（主键列）  column：查询sql的列名  property：映射结果的属性名  -->  <id column=*"id\_"* property=*"id"*/>  <!-- result标签：映射查询结果的普通列 -->  <result column=*"username\_"* property=*"username"*/>  <result column=*"birthday\_"* property=*"birthday"*/>  </resultMap>  <!-- resultMap入门 -->  <select id=*"findUserListResultMap"* **resultMap=*"userListResultMap"***>  SELECT id id\_,username username\_,birthday birthday\_ FROM user  </select> |

<id/>：此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个<id />。

Property：表示User类的属性。

Column：表示sql查询出来的字段名。

Column和property放在一块儿表示将sql查询出来的字段映射到指定的pojo类属性上。

<result/>：普通结果，即pojo的属性。

# 关联查询

## 商品订单数据模型

用户表：user

记录了购买商品的用户信息

Id：唯一标识一个用户

订单表：orders

记录了用户创建的订单

创建用户:user\_id(外键)

订单号

创建时间

订单状态

一对一：一个订单只能由一个用户创建

一对多：一个用户可以创建多个订单

注意：因为一个订单信息只会是一个人下的订单，所以从查询订单信息出发，关联查询用户信息为一对一查询。如果从用户信息出发，查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

## 一对一查询

### 需求

查询所有订单信息，关联查询下单用户信息。

### SQL语句

SELECT

orders.\*,

user.username,

user.address

FROM

orders LEFT JOIN user

ON orders.user\_id = user.id

主信息：订单信息

从信息：用户信息

### 方法一：resultType

使用resultType进行结果映射。

#### 分析

1. 如果使用resultType进行结果映射，则查询的列名和映射pojo的属性名称要一致。
2. 关联查询的结果集，既包含订单信息，又包含用户信息。使用订单po或者用户po类都无法完成结果映射。
3. 需要定义扩展po类，此类继承订单信息po类，同时还要包括用户信息：

#### 定义扩展po类

扩展Po类中应该包括上边sql查询出来的所有字段，如下：

**public** **class** OrdersExt **extends** Orders {

**private** String username;// 用户名称

**private** String address;// 用户地址

get/set。。。。

OrdersExt类继承Orders类后，便会继承Orders类的所有字段，除此之外只需要定义用户信息字段即可。

#### Mapper映射文件

在UserMapper.xml中，添加以下代码：

<!-- 查询所有订单信息 -->

<select id=*"findOrdersList"* resultType=*"com.kkb.mybatis.po.OrdersExt"*>

SELECT

orders.\*,

user.username,

user.address

FROM

orders, user

WHERE orders.user\_id = user.id

</select>

#### Mapper接口

在UserMapper接口中，添加以下接口方法：

public List<*OrdersExt*> findOrdersList() throws Exception;

#### 测试代码

在UserMapperTest测试类中，添加测试代码：

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

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

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

//查询订单信息

List<*OrdersExt*> list = userMapper.findOrdersList();

System.*out*.println(list);

//关闭session

session.close();

}

#### 小结

1、一对一关联查询时，可以定义专门的扩展po类作为输出结果类型。

2、该扩展po类中定义了sql查询结果集所有的字段对应的属性。

3、此方法较为简单，企业中使用普遍。

### 方法二：resultMap

使用resultMap进行结果映射，定义专门的resultMap用于映射一对一查询结果。

#### 修改扩展po类

在OrdersExt类中加入User属性，user属性中用于存储关联查询的用户信息，因为订单关联查询用户是一对一关系，所以这里使用单个User对象存储关联查询的用户信息。

**public** **class** OrdersExt **extends** Orders {

**private** String username;// 用户名称

**private** String address;// 用户地址

**private** User user;// 用户对象

get/set。。。。

#### Mapper映射文件

在UserMapper.xml中，添加以下代码：

|  |
| --- |
| <!-- 查询订单关联用户信息使用resultmap -->  <resultMap type=*"OrdersExt"* id=*"ordersAndUserRstMap"*>  <id column=*"id"* property=*"id"*/>  <result column=*"user\_id"* property=*"userId"*/>  <result column=*"number"* property=*"number"*/>  <result column=*"createtime"* property=*"createtime"*/>  <result column=*"note"* property=*"note"*/>  <!-- 一对一关联映射 -->  <!--  property:Orders对象的user属性  javaType：user属性对应 的类型  -->  <**association** property=*"user"* javaType=*"com.kkb.mybatis.po.User"*>  <!-- column:user表的主键对应的列 property：user对象中id属性-->  <id column=*"user\_id"* property=*"id"*/>  <result column=*"username"* property=*"username"*/>  <result column=*"address"* property=*"address"*/>  </association>  </resultMap>  <select id=*"findOrdersAndUserRstMap"* resultMap=*"ordersAndUserRstMap"*>  SELECT  o.id,  o.user\_id,  o.number,  o.createtime,  o.note,  u.username,  u.address  FROM  orders o  JOIN `user` u ON u.id = o.user\_id  </select> |

**association**：表示进行一对一关联查询映射

**property**：表示关联查询的结果存储在com.kkb.mybatis.po.Orders的user属性中

**javaType**：表示关联查询的映射结果类型

#### Mapper接口

在UserMapper接口中，添加以下接口方法：

public List<OrdersExt> findOrdersAndUserRstMap() throws Exception;

#### 测试代码

在UserMapperTest测试类中，添加测试代码：

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

//获取session

SqlSession session = sqlSessionFactory.openSession();

//获限mapper接口实例

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

//查询订单信息

List<OrdersExt> list = userMapper.findOrdersAndUserRstMap();

System.*out*.println(list);

//关闭session

session.close();

}

#### 小结

使用resultMap进行结果映射时，具体是使用association完成关联查询的映射，将关联查询信息映射到pojo对象中。

## 一对多查询

### 需求

查询所有用户信息及用户关联的订单信息。

### Sql语句

SELECT

u.\*,

o.id oid,

o.number,

o.createtime,

o.note

FROM

`user` u

LEFT JOIN orders o ON u.id = o.user\_id

主信息：用户信息

从信息：订单信息

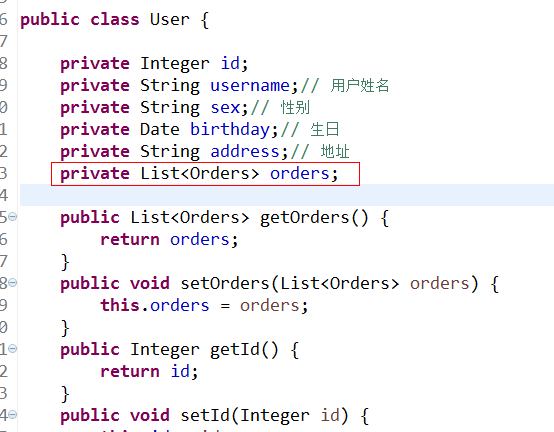
### 分析

在一对多关联查询时，只能使用resultMap进行结果映射。

1. 一对多关联查询时，sql查询结果有多条，而映射对象是一个。
2. resultType完成结果映射的方式的一条记录映射一个对象。
3. resultMap完成结果映射的方式是以[主信息]为主对象，[从信息]映射为集合或者对象，然后封装到主对象中。

### 修改po类

在User类中加入List<Orders> orders属性



### Mapper映射文件

在UserMapper.xml文件中，添加以下代码：

|  |
| --- |
| <resultMap type=*"user"* id=*"userAndOrderRstMap"*>  <!-- 用户信息映射 -->  <id property=*"id"* column=*"id"*/>  <result property=*"username"* column=*"username"*/>  <result property=*"birthday"* column=*"birthday"*/>  <result property=*"sex"* column=*"sex"*/>  <result property=*"address"* column=*"address"*/>  <!-- 一对多关联映射 -->  <collection property=*"orders"* ofType=*"orders"*>  <id property=*"id"* column=*"oid"*/>  <result property="userId" column="id"/>  <result property=*"number"* column=*"number"*/>  <result property=*"createtime"* column=*"createtime"*/>  <result property=*"note"* column=*"note"*/>  </collection>  </resultMap>  <select id=*"findUserAndOrderRstMap"* resultMap=*"userAndOrderRstMap"*>  SELECT  u.\*,  o.id oid,  o.number,  o.createtime,  o.note  FROM  `user` u  LEFT JOIN orders o ON u.id = o.user\_id  </select> |

Collection标签：定义了一对多关联的结果映射。

**property**=*"orders"：*关联查询的结果集存储在User对象的上哪个属性。

**ofType**=*"orders"：*指定关联查询的结果集中的对象类型即List中的对象类型。此处可以使用别名，也可以使用全限定名。

### Mapper接口

|  |
| --- |
| // resultMap入门  **public** List<User> findUserAndOrdersRstMap() **throws** Exception; |

### 测试代码

|  |
| --- |
| @Test  **public** **void** testFindUserAndOrdersRstMap() {  SqlSession session = sqlSessionFactory.openSession();  UserMapper userMapper = session.getMapper(UserMapper.**class**);  List<User> result = userMapper.findUserAndOrdersRstMap();  **for** (User user : result) {  System.***out***.println(user);  }  session.close();  } |

# 延迟加载

## 什么是延迟加载

\* MyBatis中的延迟加载，也称为**懒加载**，是指在进行关联查询时，按照设置延迟规则推迟对关联对象的select查询。延迟加载可以有效的减少数据库压力。

\* Mybatis的延迟加载，需要通过**resultMap标签中的association和collection**子标签才能演示成功。

\* Mybatis的延迟加载，也被称为是嵌套查询，对应的还有**嵌套结果**的概念，可以参考一对多关联的案例。

\* 注意：MyBatis的延迟加载只是对关联对象的查询有延迟设置，对于主加载对象都是直接执行查询语句的。

## 延迟加载的分类

\* MyBatis根据对关联对象查询的select语句的执行时机，分为三种类型：直接加载、侵入式加载与深度延迟加载

* **直接加载：** 执行完对主加载对象的select语句，马上执行对关联对象的select查询。
* **侵入式延迟**：执行对主加载对象的查询时，不会执行对关联对象的查询。但当要访问主加载对象的详情时，就会马上执行关联对象的select查询。即对关联对象的查询执行，侵入到了主加载对象的详情访问中。也可以这样理解：将关联对象的详情侵入到了主加载对象的详情中，即将关联对象的详情作为主加载对象的详情的一部分出现了。
* **深度延迟：**执行对主加载对象的查询时，不会执行对关联对象的查询。访问主加载对象的详情时也不会执行关联对象的select查询。只有当真正访问关联对象的详情时，才会执行对关联对象的select查询。

\* **延迟加载策略需要在Mybatis的全局配置文件中，通过<settings>标签进行设置。**

## 案例准备

查询订单信息及它的下单用户信息。

## 直接加载

通过对全局参数：lazyLoadingEnabled进行设置，默认就是false。

<settings>

<!-- 延迟加载总开关 -->

<setting name="**lazyLoadingEnabled**" value="**false**"/>

</settings>

## 侵入式延迟加载

<settings>

<!-- 延迟加载总开关 -->

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

<!-- 侵入式延迟加载开关 -->

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

</settings>

## 深度延迟加载

<settings>

<!-- 延迟加载总开关 -->

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

<!-- 侵入式延迟加载开关 -->

<setting name="aggressiveLazyLoading" value="**false**"/>

</settings>

## N+1问题

* 深度延迟加载的使用会提升性能。
* 如果延迟加载的表数据太多，此时会产生N+1问题，主信息加载一次算1次，而从信息是会根据主信息传递过来的条件，去查询从表多次。

# 动态SQL

动态SQL的思想：就是使用不同的动态SQL标签去完成字符串的拼接处理。

解决的问题是：

* 1. 在映射文件中，会编写很多有重叠部分的SQL语句，比如SELECT语句和WHERE语句等这些重叠语句，该如何处理
  2. 如果页面传递过来一个参数，但是SQL语句中的条件有多个，此时会发生问题。

## if标签

综合查询的案例中，查询条件是由页面传入，页面中的查询条件可能输入用户名称，也可能不输入用户名称。

|  |
| --- |
| <select id=*"findUserList"* parameterType=*"queryVo"* resultType=*"user"*>  SELECT \* FROM user where 1=1  <if test=*"user != null"*>  <if test=*"user.username != null and user.username != ''"*>  AND username like '%${user.username}%'  </if>  </if>  </select> |

**注意：要做『不等于空』字符串校验。**

## where标签

上边的sql中的1=1，虽然可以保证sql语句的完整性：但是存在性能问题。Mybatis提供where标签解决该问题。

代码修改如下：

|  |
| --- |
| <select id=*"findUserList"* parameterType=*"queryVo"* resultType=*"user"*>  SELECT \* FROM user  <!-- where标签会处理它后面的第一个and -->  <where>  <if test=*"user != null"*>  <if test=*"user.username != null and user.username != ''"*>  AND username like '%${user.username}%'  </if>  </if>  </where>  </select> |

## sql片段

在映射文件中可使用sql标签将重复的sql提取出来，然后使用include标签引用即可，最终达到sql重用的目的，具体实现如下：

* 原映射文件中的代码：

|  |
| --- |
| <select id=*"findUserList"* parameterType=*"queryVo"* resultType=*"user"*>  SELECT \* FROM user  <!-- where标签会处理它后面的第一个and -->  <where>  <if test=*"user != null"*>  <if test=*"user.username != null and user.username != ''"*>  AND username like '%${user.username}%'  </if>  </if>  </where>  </select> |

* 将where条件抽取出来：

|  |
| --- |
| <sql id=*"query\_user\_where"*>  <if test=*"user != null"*>  <if test=*"user.username != null and user.username != ''"*>  AND username like '%${user.username}%'  </if>  </if>  </sql> |

* 使用include引用：

|  |
| --- |
| <!-- 使用包装类型查询用户 使用ognl从对象中取属性值，如果是包装对象可以使用.操作符来取内容部的属性 -->  <select id=*"findUserList"* parameterType=*"queryVo"* resultType=*"user"*>  SELECT \* FROM user  <!-- where标签会处理它后面的第一个and -->  <where>  <include refid=*"query\_user\_where"*></include>  </where>  </select> |

**注意：**

**1、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”/>**

## foreach

### 需求

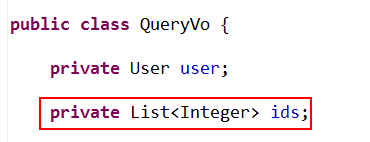
综合查询时，传入多个id查询用户信息，用下边两个sql实现：

SELECT \* FROM USERS WHERE username LIKE '%小明%' AND (id =1 OR id =10 OR id=16)

SELECT \* FROM USERS WHERE username LIKE '%小明%' AND id IN (1,10,16)

### POJO

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



### Mapper映射文件

|  |
| --- |
| <sql id=*"query\_user\_where"*>  <if test=*"user != null"*>  <if test=*"user.username != null and user.username != ''"*>  AND username like '%${user.username}%'  </if>  </if>  <if test=*"ids != null and ids.size() > 0"*>  <!-- collection：指定输入的集合参数的参数名称 -->  <!-- item：声明集合参数中的元素变量名 -->  <!-- open：集合遍历时，需要拼接到遍历sql语句的前面 -->  <!-- close：集合遍历时，需要拼接到遍历sql语句的后面 -->  <!-- separator：集合遍历时，需要拼接到遍历sql语句之间的分隔符号 -->  <foreach collection=*"ids"* item=*"id"* open=*" AND id IN ( "*  close=*" ) "* separator=*","*>  #{id}  </foreach>  </if>  </sql> |

### 测试代码

在UserMapperTest测试代码中，修改testFindUserList方法，如下：

|  |
| --- |
| @Test  **public** **void** testFindUserList() **throws** Exception {  SqlSession sqlSession = sqlSessionFactory.openSession();  // 获得mapper的代理对象  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  // 创建QueryVo对象  QueryVo queryVo = **new** QueryVo();  // 创建user对象  User user = **new** User();  user.setUsername("小明");  queryVo.setUser(user);  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);  // 根据queryvo查询用户  List<User> list = userMapper.findUserList(queryVo);  System.***out***.println(list);  sqlSession.close();  } |

### 注意事项

**如果parameterType不是POJO类型，而是List或者Array的话，那么foreach语句中，collection属性值需要固定写死为list或者array。**

### 作业

编写批量删除的select标签，parameterType指定为list

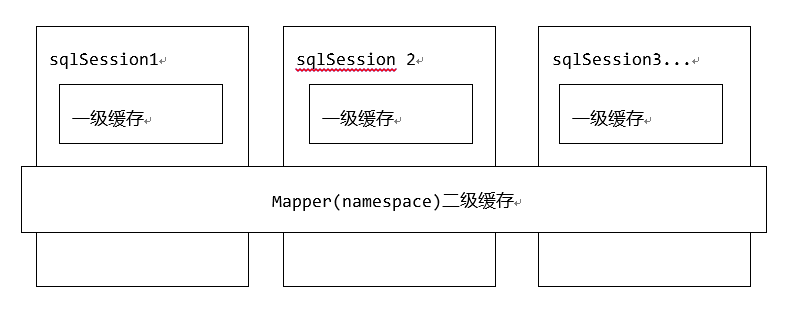
注意：foreach标签应该怎么写？

# Mybatis缓存

## 缓存介绍

\* Mybatis提供**查询缓存**，如果缓存中有数据就不用从数据库中获取，用于减轻数据压力，提高系统性能。

\* Mybatis的查询**缓存总共有两级**，我们称之为一级缓存和二级缓存，如图：

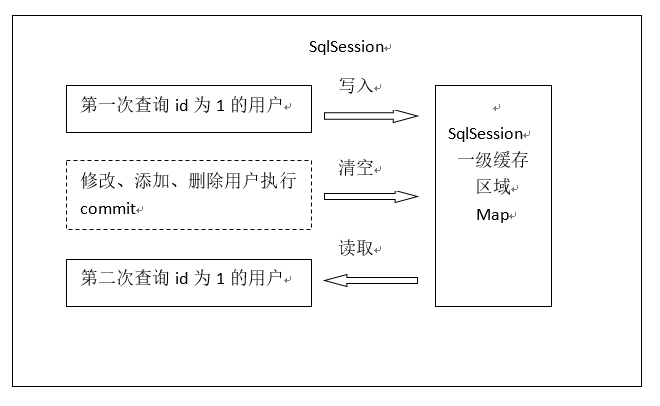


* 一级缓存是**SqlSession级别**的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。
* 二级缓存是Mapper（namespace）级别的缓存。多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

## 一级缓存

**Mybatis默认开启了一级缓存**

### 原理图



**说明：**

1. 第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息，将查询到的用户信息存储到一级缓存中。
2. 如果中间sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。
3. 第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

### 测试1

@Test

**public** **void** testOneLevelCache() {

SqlSession sqlSession = sqlSessionFactory.openSession();

UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper.findUserById(1);

System.*out*.println(user1);

// 第二次查询ID为1的用户

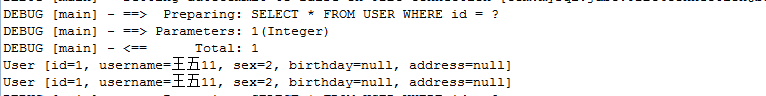
User user2 = mapper.findUserById(1);

System.*out*.println(user2);

sqlSession.close();

}

只输出一次SQL：



### 测试2

@Test

**public** **void** testOneLevelCache() {

SqlSession sqlSession = sqlSessionFactory.openSession();

UserMapper mapper = sqlSession.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper.findUserById(1);

System.*out*.println(user1);

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

user.setUsername("隔壁老詹1");

user.setAddress("洛杉矶湖人");

//执行增删改操作，清空缓存

mapper.insertUser(user);

// 第二次查询ID为1的用户

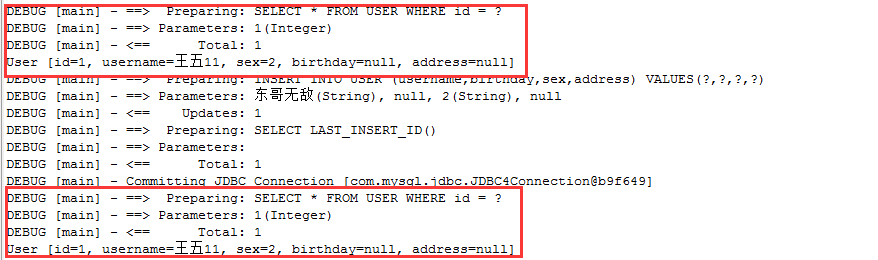
User user2 = mapper.findUserById(1);

System.*out*.println(user2);

sqlSession.close();

}

中间执行了commit操作，同样的查询SQL输出两次：



### 具体应用

正式开发，是将mybatis和spring进行整合开发，事务控制在service中。

一个service方法中包括 很多mapper方法调用：

service{

//开始执行时，开启事务，创建SqlSession对象

//第一次调用mapper的方法findUserById(1)

//第二次调用mapper的方法findUserById(1)，从一级缓存中取数据

//方法结束，sqlSession关闭

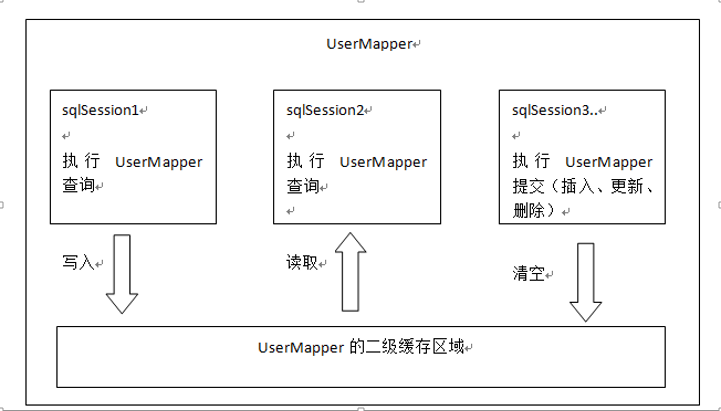
}

如果是执行两次service调用查询相同 的用户信息，是不走一级缓存的，因为mapper方法结束，sqlSession就关闭，一级缓存就清空。

## 二级缓存

### 原理

* 二级缓存是mapper（namespace）级别的。
* 下图是多个sqlSession请求UserMapper的二级缓存图解。



**说明：**

1. 第一次调用mapper下的SQL去查询用户信息。查询到的信息会存到该mapper对应的**二级缓存区域**内。
2. 第二次调用相同namespace下的mapper映射文件中相同的SQL去查询用户信息。会去对应的二级缓存内取结果。
3. 如果调用相同namespace下的mapper映射文件中的增删改SQL，并执行了commit操作。此时会清空该namespace下的二级缓存。

### 开启二级缓存

* **Mybatis默认是没有开启二级缓存**
* **开启步骤**

1. 在核心配置文件SqlMapConfig.xml中加入以下内容（开启二级缓存总开关）：

<!-- 开启二级缓存总开关 -->

<settings>

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

</settings>

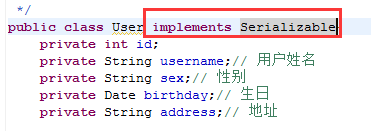
1. 在UserMapper映射文件中，加入以下内容，开启二级缓存：

<!-- 开启本mapper下的namespace的二级缓存，默认使用的是mybatis提供的PerpetualCache -->

**<cache></cache>**

### 实现序列化

* 由于二级缓存的数据不一定都是存储到内存中，它的存储介质多种多样，比如说存储到文件系统中，所以需要给缓存的对象执行序列化。
* 如果该类存在父类，那么父类也要实现序列化。



### 测试1

@Test

**public** **void** testTwoLevelCache() {

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

UserMapper mapper1 = sqlSession1.getMapper(UserMapper.**class**);

UserMapper mapper2 = sqlSession2.getMapper(UserMapper.**class**);

UserMapper mapper3 = sqlSession3.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper1.findUserById(1);

System.*out*.println(user1);

// 关闭SqlSession1

sqlSession1.close();

// 第二次查询ID为1的用户

User user2 = mapper2.findUserById(1);

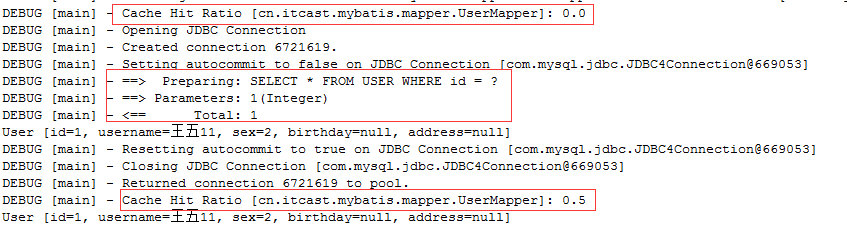
System.*out*.println(user2);

// 关闭SqlSession2

sqlSession2.close();

}

**SQL输出结果：**



Cache Hit Radio ： 缓存命中率

第一次缓存中没有记录，则命中率0.0；

第二次缓存中有记录，则命中率0.5（访问两次，有一次命中）

### 测试2

@Test

**public** **void** testTwoLevelCache() {

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession();

SqlSession sqlSession3 = sqlSessionFactory.openSession();

UserMapper mapper1 = sqlSession1.getMapper(UserMapper.**class**);

UserMapper mapper2 = sqlSession2.getMapper(UserMapper.**class**);

UserMapper mapper3 = sqlSession3.getMapper(UserMapper.**class**);

// 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

User user1 = mapper1.findUserById(1);

System.*out*.println(user1);

// 关闭SqlSession1

sqlSession1.close();

//修改查询出来的user1对象，作为插入语句的参数

user1.setUsername("隔壁老詹1");

user1.setAddress("洛杉矶湖人");

mapper3.insertUser(user1);

// 提交事务

sqlSession3.commit();

// 关闭SqlSession3

sqlSession3.close();

// 第二次查询ID为1的用户

User user2 = mapper2.findUserById(1);

System.*out*.println(user2);

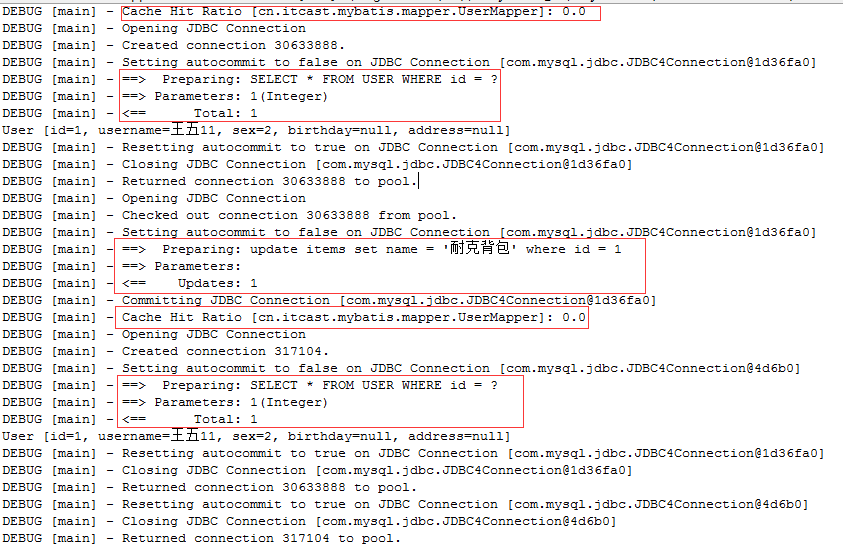
// 关闭SqlSession2

sqlSession2.close();

}

SQL输出结果：

根据SQL分析，确实是清空了二级缓存了。



### 禁用二级缓存

* 默认二级缓存的粒度是Mapper级别的，但是如果在同一个Mapper文件中某个查询不想使用二级缓存的话，就需要对缓存的控制粒度更细。
* 在select标签中设置**useCache=false**，可以禁用当前select语句的二级缓存，即每次查询都是去数据库中查询，**默认情况下是true**，即该statement使用二级缓存。

<select id=*"findUserById"* parameterType=*"int"*

resultType=*"com.kkb.mybatis.po.User"* **useCache=*"true"***>

SELECT \* FROM user WHERE id = #{id}

</select>

### 刷新二级缓存

* **通过flushCache属性，可以控制select、insert、update、delete标签是否属性二级缓存**
* **默认设置**

\* 默认情况下如果是select语句，那么flushCache是false。

\* 如果是insert、update、delete语句，那么flushCache是true。

* **默认配置解读**

\* 如果查询语句设置成true，那么每次查询都是去数据库查询，即意味着该查询的二级缓存失效。

\* 如果增删改语句设置成false，即使用二级缓存，那么如果在数据库中修改了数据，而缓存数据还是原来的，这个时候就会出现脏读。

flushCache设置如下：

<select id=*"findUserById"* parameterType=*"int"*

resultType=*"com.kkb.mybatis.po.User"* useCache=*"true"* **flushCache=*"true"***>

SELECT \* FROM user WHERE id = #{id}

</select>

### 整合Ehcache

#### Ehcache介绍

* 什么是Ehcache？

Ehcache是一种广泛使用的**开源Java分布式缓存**。主要面向通用缓存,Java EE和轻量级容器。它具有内存和磁盘存储，缓存加载器,缓存扩展,缓存异常处理程序,一个gzip缓存servlet过滤器,支持REST和SOAP api等特点。

* Ehcache的特点

1. 快速

2. 简单

3. 多种缓存策略

4. 缓存数据有两级：内存和磁盘，因此无需担心容量问题

5. 缓存数据会在虚拟机重启的过程中写入磁盘

6. 可以通过RMI、可插入API等方式进行分布式缓存

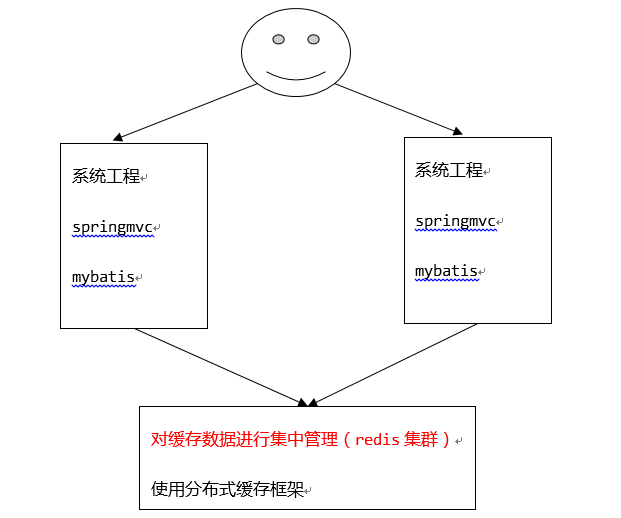
7. 具有缓存和缓存管理器的侦听接口

8. 支持多缓存管理器实例，以及一个实例的多个缓存区域

9. 提供Hibernate的缓存实现

#### 分布式缓存理解

* **Mybatis自身无法实现分布式缓存，需要和其它分布式缓存框架进行整合。**
* 分布式缓存的使用场景（集群部署方式）



* **为什么使用分布式缓存？**

\* 如果不使用分布式缓存，缓存的数据在各个服务单独存储，会存在缓存不一致等问题。所以要使用分布式缓存对缓存数据进行集中式管理。

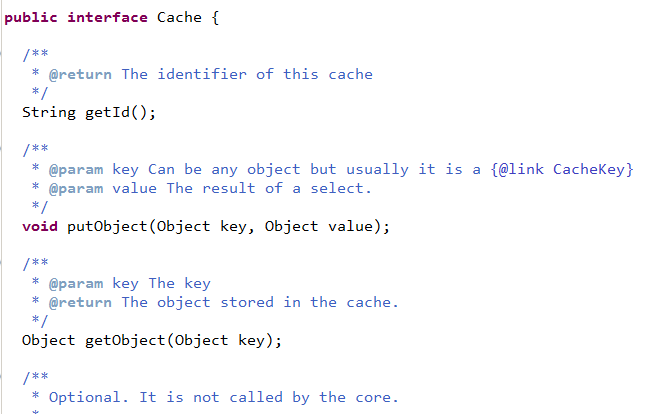
#### 整合思路（重点）

* **Mybatis提供了一个Cache接口，该接口有一个默认的实现类PerpetualCache。**
* **只要实现cache接口就可以实现mybatis的二级缓存。**

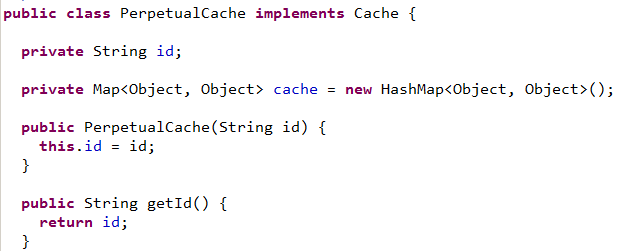
\* mybatis的定位是做持久层框架，对于缓存数据的管理不是mybatis的特长

\* 为了提高mybatis的性能，所以需要mybatis和第三方缓存数据库整合，比如ehcache、memcache、redis等

* **Mybatis提供接口如下：**



* **Mybatis的默认实现类：**



#### 整合ehcache的步骤

1. 引入ehcache的jar包；
2. 在mapper映射文件中，配置cache标签的type为ehcache对cache接口的实现类类型。
3. 加入ehcache的配置文件

##### 第一步：引入ehcache的jar包



##### 第二步：配置cache的type属性

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

##### 第三步：添加ehcache的配置文件

* **在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"*>

<!-- 缓存数据要存放的磁盘地址 -->

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

<!-- **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） -->

<defaultCache maxElementsInMemory=*"1000"*

maxElementsOnDisk=*"10000000"* eternal=*"false"* overflowToDisk=*"false"*

timeToIdleSeconds=*"120"* timeToLiveSeconds=*"120"*

diskExpiryThreadIntervalSeconds=*"120"* memoryStoreEvictionPolicy=*"LRU"*>

</defaultCache>

</ehcache>

### 应用场景

* 使用场景：

对于访问响应速度要求高，但是实时性不高的查询，可以采用二级缓存技术。

* 注意事项：

在使用二级缓存的时候，要设置一下**刷新间隔**（cache标签中有一个**flashInterval**属性）来定时刷新二级缓存，这个刷新间隔根据具体需求来设置，比如设置30分钟、60分钟等，**单位为毫秒**。

### 局限性

* **Mybatis二级缓存对细粒度的数据级别的缓存实现不好。**
* 场景：

对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次查询都是最新的商品信息，此时如果使用二级缓存，就无法实现当一个商品发生变化只刷新该商品的缓存信息而不刷新其他商品缓存信息，因为二级缓存是mapper级别的，当一个商品的信息发送更新，所有的商品信息缓存数据都会清空。

* 解决方法

此类问题，需要在业务层根据需要对数据有针对性的缓存。

比如可以对经常变化的 数据操作单独放到另一个namespace的mapper中。

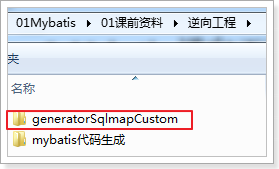
# Mybatis逆向工程

## 逆向工程介绍

使用官方网站的Mapper自动生成工具mybatis-generator-core-1.3.2来针对单表生成**po**类（Example）和Mapper接口和mapper映射文件

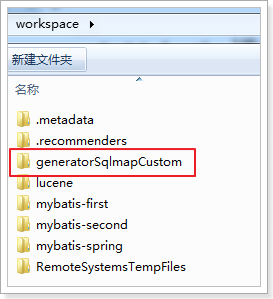
## 导入逆向工程

使用课前资料已有逆向工程，如下图：



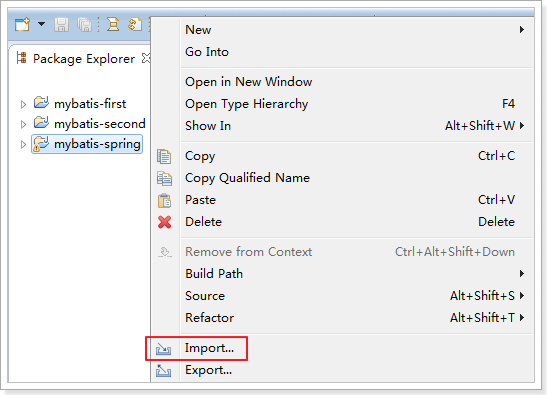
### 复制逆向工程到工作空间中

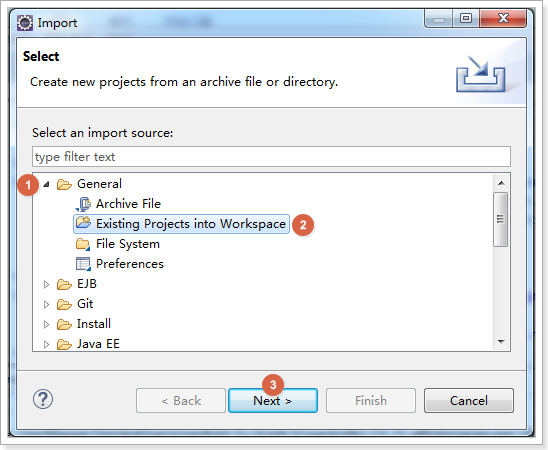
复制的效果如下图：

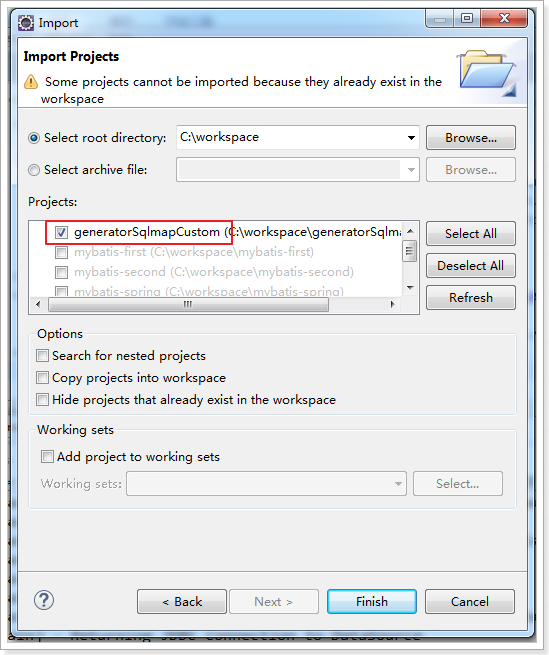


### 导入逆向工程到eclipse中

如下图方式进行导入：

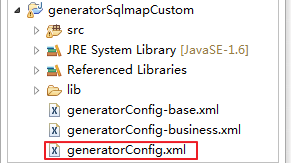






## 修改配置文件

在generatorConfig.xml中配置Mapper生成的详细信息，如下图：



注意修改以下几点:

1. **修改要生成的数据库表**
2. **pojo文件所在包路径**
3. **Mapper所在的包路径**

配置文件如下:

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

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<context id=*"testTables"* targetRuntime=*"MyBatis3"*>

<commentGenerator>

<!-- 是否去除自动生成的注释 true：是 ： false:否 -->

<property name=*"suppressAllComments"* value=*"true"* />

</commentGenerator>

<!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->

<jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*

connectionURL=*"jdbc:mysql://localhost:3306/mybatis" userId="root" password="root"*>

</jdbcConnection>

<!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver" connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"

userId="yycg" password="yycg"> </jdbcConnection> -->

<!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL

和 NUMERIC 类型解析为java.math.BigDecimal -->

<javaTypeResolver>

<property name=*"forceBigDecimals"* value=*"false"* />

</javaTypeResolver>

<!-- targetProject:生成PO类的位置 -->

<javaModelGenerator targetPackage=*"com.kkb.ms.po"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

<!-- 从数据库返回的值被清理前后的空格 -->

<property name=*"trimStrings"* value=*"true"* />

</javaModelGenerator>

<!-- targetProject:mapper映射文件生成的位置 -->

<sqlMapGenerator targetPackage=*"com.kkb.ms.mapper"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

</sqlMapGenerator>

<!-- targetPackage：mapper接口生成的位置 -->

<javaClientGenerator type=*"XMLMAPPER"*

targetPackage=*"com.kkb.ms.mapper"* targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

</javaClientGenerator>

<!-- 指定数据库表 -->

<table schema=*""* tableName=*"user"*></table>

<table schema=*""* tableName=*"order"*></table>

</context>

</generatorConfiguration>

## 注意事项

* 每次执行逆向工程代码之前，先**删除原来已经生成的mapper xml文件再进行生成。**

**\* mapper.xml**文件的内容不是被覆盖而是进行**内容追加**，会导致mybatis解析失败。

\* po类及mapper.java文件的内容是直接覆盖没有此问题。

# PageHelper分页插件

## PageHelper分页插件介绍

<https://github.com/pagehelper/Mybatis-PageHelper/blob/master/wikis/en/HowToUse.md>

\* 如果你也在用Mybatis，建议尝试该分页插件，这个一定是**最方便**使用的分页插件。

\* 目前几乎支持所有的关系型数据库

\* 最新版本是5.1.6。

## 使用方法

### 添加依赖

<dependency>

<groupId>com.github.pagehelper</groupId>

<artifactId>pagehelper</artifactId>

<version>5.1.6</version>

</dependency>

### 配置PageHelper

* Mybatis全局配置文件

<plugins>

<plugin interceptor="com.github.pagehelper.PageInterceptor">

<!-- config params as the following -->

<property name="helperDialect" value="mysql"/>

</plugin>

</plugins>

* Spring配置文件

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

<!-- other configuration -->

<property name="plugins">

<array>

<bean class="com.github.pagehelper.PageInterceptor">

<property name="properties">

<!-- config params as the following -->

<value>

helperDialect=mysql

</value>

</property>

</bean>

</array>

</property>

</bean>

### 项目中使用PageHelper

//获取第1页，10条内容，默认查询总数count

PageHelper.startPage(1, 10);

List<Country> list = countryMapper.selectAll();

//用PageInfo对结果进行包装

PageInfo page = new PageInfo(list);

//测试PageInfo全部属性

//PageInfo包含了非常全面的分页属性

assertEquals(1, page.getPageNum());

assertEquals(10, page.getPageSize());

assertEquals(1, page.getStartRow());

assertEquals(10, page.getEndRow());

assertEquals(183, page.getTotal());

assertEquals(19, page.getPages());

assertEquals(1, page.getFirstPage());

assertEquals(8, page.getLastPage());

assertEquals(true, page.isFirstPage());

assertEquals(false, page.isLastPage());

assertEquals(false, page.isHasPreviousPage());

assertEquals(true, page.isHasNextPage());

### 注意事项

1. 需要分页的查询语句，必须是处于PageHelper.startPage(1, 10);后面的第一条语句。
2. 如果查询语句是使用resultMap进行的嵌套结果映射，则无法使用PageHelper进行分页。

## plugin插件介绍

参考地址：<https://www.cnblogs.com/fangjian0423/p/mybatis-interceptor.html>

StatementHandler

ParameterHandler

ResultSetHandler

# 注解开发

## 介绍

\* MyBatis可以使用注解的方式开发DAO。

\* 这种注解开发的方式只需要程序员开发Mapper接口即可，不需要编写映射文件（XML）。

**\* 注解开发和XML配置的优缺点？**

## UserMapper 接口

|  |
| --- |
| **public** **interface** AnnotationUserMapper {  // 查询  @Select("SELECT \* FROM user WHERE id = #{id}")  **public** User findUserById(**int** id);  // 模糊查询用户列表  @Select("SELECT \* FROM user WHERE username LIKE '%${value}%'")  **public** List<User> findUserList(String username);  // 添加并实现主键返回  @Insert("INSERT INTO user (username,birthday,sex,address) VALUES (#{username},#{birthday},#{sex},#{address})")  @SelectKey(before=**false**,statement="SELECT LAST\_INSERT\_ID()",keyProperty="id",resultType=**int**.**class**)  **public** **void** insertUser(User user);  // 动态SQL  @SelectProvider(type=UserSqlBuilder.**class**,method="getDynamicSQL")  **public** List<User> dynamicSQL(UserQueryVO vo);    // 使用Results注解完成结果映射  @Results({  @Result(column="id",property="id"),  @Result(column="username",property="username"),  @Result(column="sex",property="sex"),  @Result(column="address",property="address")  })  @Select("SELECT \* FROM user WHERE id = #{id}")  **public** User findUserByIdWithResultMap(**int** id);    // 演示一对一  **public** List<OrdersExt> one2One();    // 演示一对多  **public** List<UserExt> one2Many();  // 演示延迟加载  @Results({  @Result(column="id",property="id"),  @Result(column="user\_id",property="user\_id"),  @Result(column="number",property="number"),  @Result(column="note",property="note"),  @Result(property="user",javaType=User.**class**,column="user\_id",one=@One(select="com.kkb.mybatis.anno.AnnotationUserMapper.findUserById",fetchType=FetchType.***LAZY***))  })  @Select("SELECT \* FROM orders")  **public** List<OrdersExt> lazyLoading();    **class** UserSqlBuilder {  **public** String getDynamicSQL(**final** UserQueryVO vo) {  **return** **new** SQL() {  {  SELECT("\*");  FROM("user");  User user = vo.getUser();  **if** (user != **null**) {  **if**(user.getUsername() != **null** && !user.equals("")) {  WHERE("username like '%"+user.getUsername()+"%'");  }  }  ORDER\_BY("id");  }  }.toString();  }  }  } |

## 加载UserMapper接口

在Mybatis的全局配置文件的<mappers>标签中加载mapper接口文件

## 常用注解说明

### 增删改查-静态SQL

@Insert:相当于<insert>标签，实现新增

@Update: 相当于<update>标签，实现更新

@Delete: 相当于<delete>标签，实现删除

@Select: 相当于<select>标签，实现查询

@SelectKey：相当于<selectKey>标签，实现主键返回

### 增删改查-动态SQL

@InsertProvider: 相当于<insert>标签，实现新增

@UpdateProvider: 相当于<update>标签，实现更新

@DeleteProvider: 相当于<delete>标签，实现删除

@SelectProvider: 相当于<select>标签，实现查询

### 多表关联

* @Results: 相当于<resultMap>标签，需要和@Result注解一起使用。
* @Result: 相当于<result>和<id>标签，实现结果集中某一列的数据映射

\* column 数据库的列名

\* property 需要装配的属性名

\* one 需要使用的@One 注解（@Result（one=@One（）））

\* many 需要使用的@Many 注解（@Result（many=@many（）））

* @One: 相当于<association>标签，实现一对一关系映射
* @Many:相当于<collection>标签，实现一对多关系映射
* @One和@Many注解的属性：

\* select 属性：代表将要执行的 sql 语句

\* fetchType 属性：代表加载方式，一般如果要延迟加载都设置为 LAZY 的值

* 使用格式：

1. @Results（{@Result（），@Result（）}）或@Results（@Result（））
2. @Result(column=" ",property="",one=@One(select=""))

### 辅助注解

@Options：相当于标签属性的设置

@Param：如果你的映射器的方法需要多个参数，这个注解可以被应用于映射器的方法参数来给每个参数一个名字。

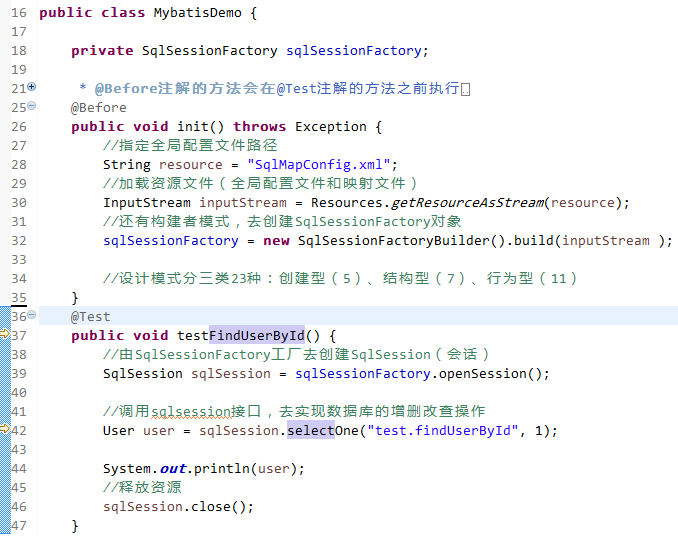
### 其他注解

@CacheNamespace：相当于<cache>标签，实现二级缓存。

属性：implemetation，eviction，flushInterval，size，readWrite，blocking和properties

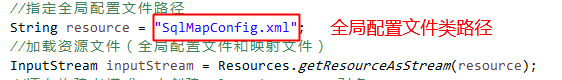
# 源码专题

## 测试代码



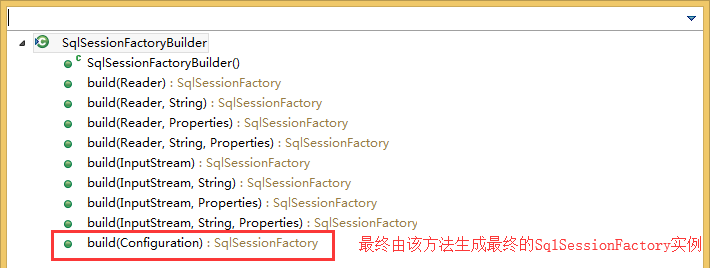
## MyBatis初始化过程解析

### 获取配置文件



### 创建SqlSessionFactoryBuilder对象

* 见名知意，该对象是用来创建SqlSessionFactory工厂对象的，我们去看一下源码：



* SqlSessionFactoryBuilder中只有一些重载的build函数，这些build函数的入参都是MyBatis配置文件的输入流，返回值都是SqlSessionFactory；
* 由此可见，SqlSessionFactoryBuilder的作用很纯粹，就是用来通过配置文件创建SqlSessionFactory对象的。

### 创建SqlSessionFactory

#### SqlSessionFactory构造过程

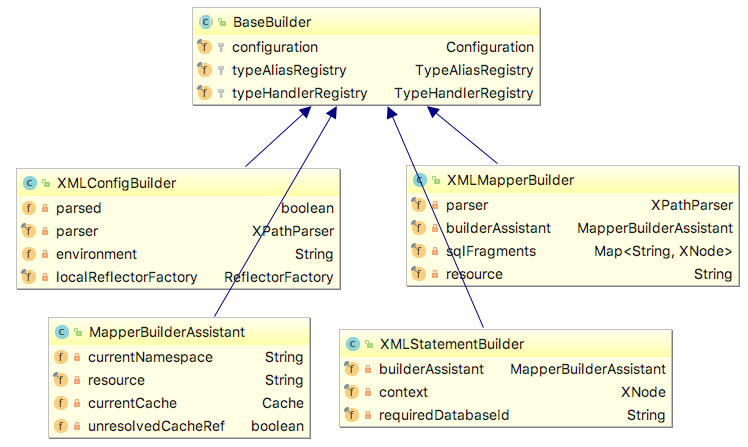
* 下面具体来看一下，build函数是如何创建SqlSessionFactory对象的。



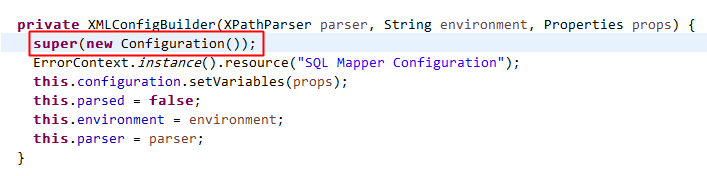
* 通过源码得知，最终会生成一个DefaultSqlSessionFactory实例。不过我们更关心的是它的参数，这个才是我们初始化最核心的对象：**Configuration，跟踪源码到第78行，我们发现它其实是由XMLConfigBuilder对象解析XML文件后得到的对象。**
* **接下来我们重点关注Configuration对象的创建过程**

#### 构造XMLConfigBuilder对象

build函数首先会构造一个XMLConfigBuilder对象，从名字上大致可以猜到，该对象是用来解析XML配置文件的。下面来看一下XMLConfigBuilder的体系结构：



* **XMLxxxBuilder**是用来解析XML配置文件的，不同类型XMLxxxBuilder用来解析MyBatis配置文件的不同部位。
  1. XMLConfigBuilder用来解析MyBatis的全局配置文件
  2. XMLMapperBuilder用来解析MyBatis中的映射文件
  3. XMLStatementBuilder用来解析映射文件中的statement语句。
  4. MapperBuilderAssistant用来辅助解析映射文件并生成**MappedStatement**对象
* 这些XMLxxxBuilder都有一个共同的父类——**BaseBuilder**。这个**父类维护了一个全局的Configuration对象**，**MyBatis的配置文件解析后就以Configuration对象的形式存储**。
* 当创建XMLConfigBuilder对象时，就会初始化Configuration对象，并且在初始化Configuration对象的时候，一些别名会被注册到Configuration的typeAliasRegistry容器中。



#### 解析全局配置文件

当有了XMLConfigBuilder对象之后，接下来就可以用它来解析配置文件了。



从上述代码中可以看到，XMLConfigBuilder会依次解析配置文件中的<properties>、<settings>、<environments>、<typeAliases>、<plugins>、<mappers>等属性。下面介绍下几个重要属性的解析过程。

##### 解析properties标签

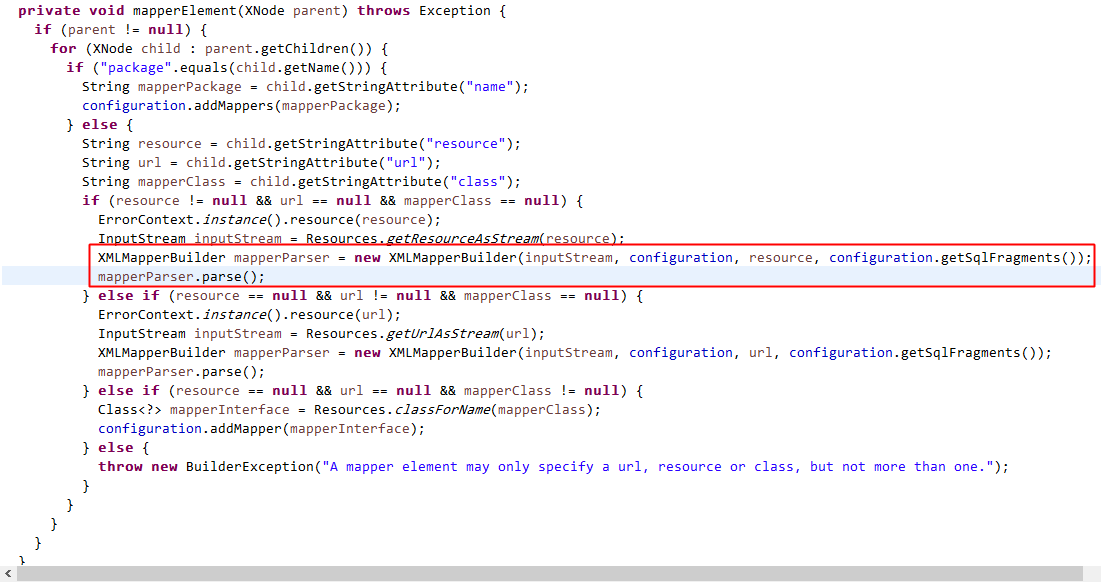
##### 解析settings标签

##### 解析typeAlias标签

##### 解析mappers标签（解析映射文件）

###### url或者resource加载方式

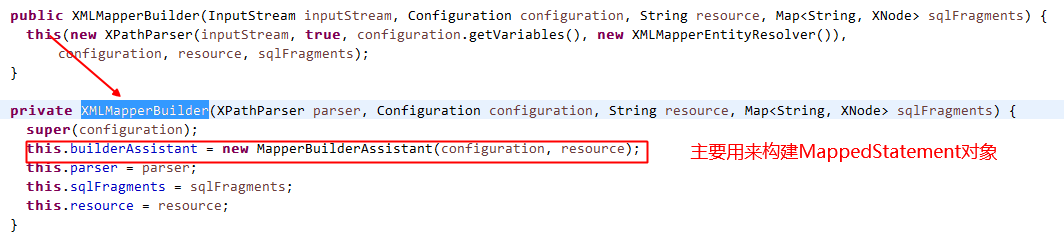
* <mappers>节点的定义方式有四种（参考章节5.4）：
* **<mappers>节点的解析过程如下**



**说明：**

1. MyBatis会遍历<mappers>下所有的子节点
2. 如果当前遍历到的节点是**<package>**，则MyBatis会将该包下的所有Mapper Class注册到configuration的mapperRegistry容器中。
3. 如果当前节点为**<mapper>且拥有class属性**，会将class指定的Mapper Class注册到configuration的mapperRegistry容器中。
4. 如果当前节点为**<mapper>且拥有resource或者url属性**，则直接找到对应的XML约束文件进行解析。

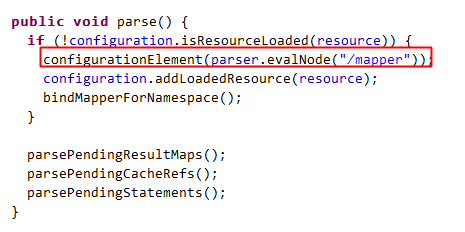
* 其中<mappers>节点解析的过程如上红色标记的内容，在解析之前，需要创建XMLMapperBuilder，创建过程如下：



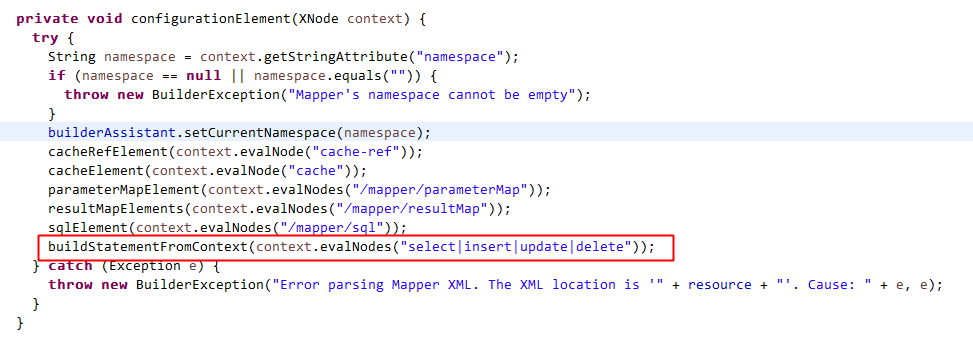
**说明：**

1. 首先会初始化父类BaseBuilder，并将configuration赋给BaseBuilder；
2. 然后创建MapperBuilderAssistant对象，该对象为XMLMapperBuilder的协助者，用来协助XMLMapperBuilder完成一些解析映射文件的动作。

* 当有了XMLMapperBuilder后，便可进入解析<mapper>的过程：



* **configurationElement函数**



* **buildStatementFromContext函数:**

该函数会将映射文件中的sql语句解析成MappedStatement对象，并存在configuration的mappedStatements集合中。



* 接下来我们进入XMLStatementBuilder类的parseStatementNode去看看：





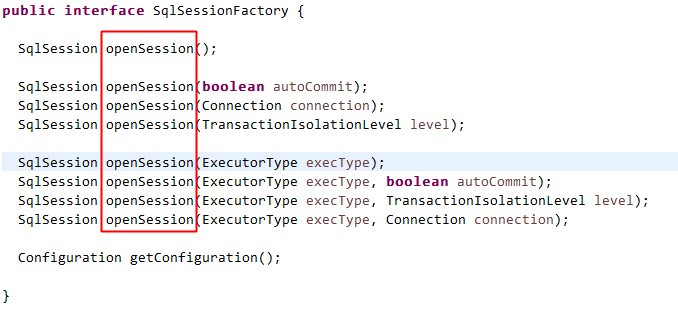
* 最终由MapperBuilderAssistant完成MappedStatement对象的封装，并且将**MappedStatement**对象放入Configuration对象的**mappedStatements**容器中。

###### package或者class加载方式

## SQL执行流程解析

### 获取SqlSession对象

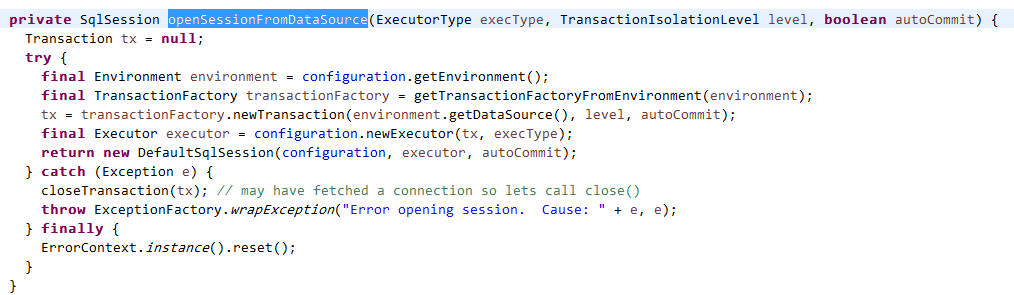
通过SqlSessionFactory的openSession方法获取SqlSession接口的实例**DefaultSqlSession对象**。



此处，SqlSessionFactory提供了很多重载的openSession方法，其中的参数说明如下：

* **autoCommit**：是否开启JDBC事务的自动提交，默认为false。
* Connection：提供一个连接
* TransactionIsolationLevel：事务隔离级别（四种隔离级别）
* ExecutorType：执行器类型（SIMPLE、REUSE、BATCH）

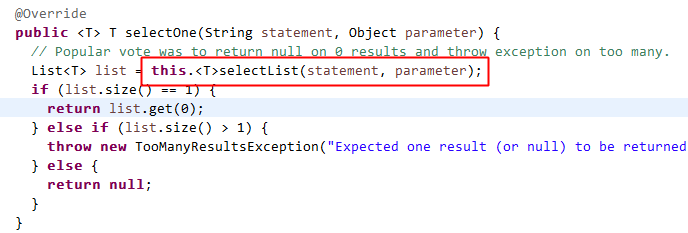
**DefaultSqlSessionFactory**：



### 调用SqlSession接口方法完成增删改查操作

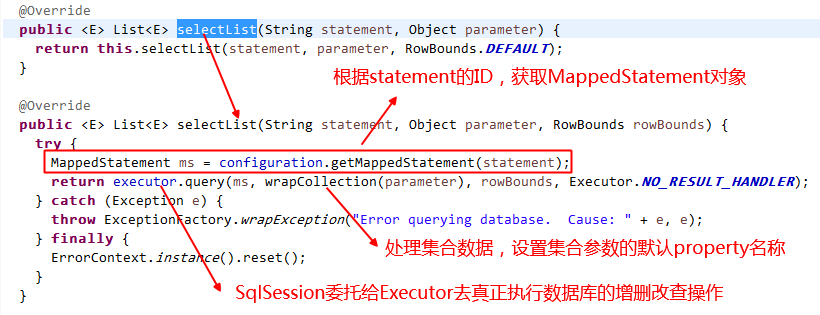
#### 调用DefaultSqlSession的方法

* 先进入的selectOne方法

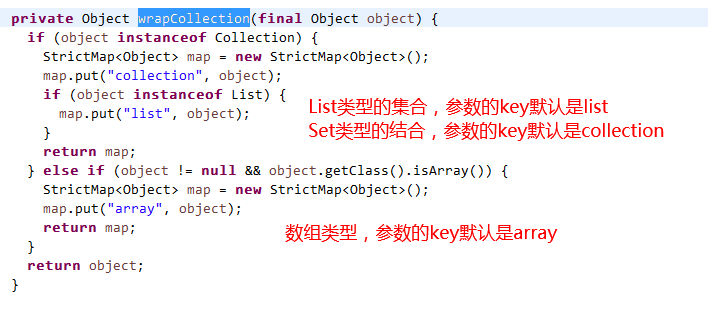


上面的selectOne方法底层调用了selectList方法

* 再进入selectList方法



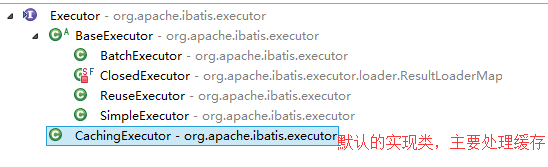
* 先看看wrapCollection方法



这个方法主要是指定了一些集合或者数组的**默认key**，如果输入参数是这三种类型的数据，只能通过这三个key去获取对应的数据。

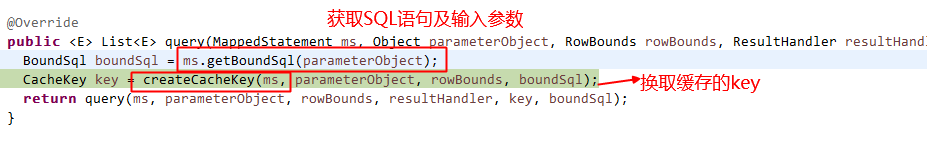
#### Executor执行器的操作流程

这个Executor（执行器），才是真正对JDBC操作的封装，它的结构如下：



##### CachingExecutor

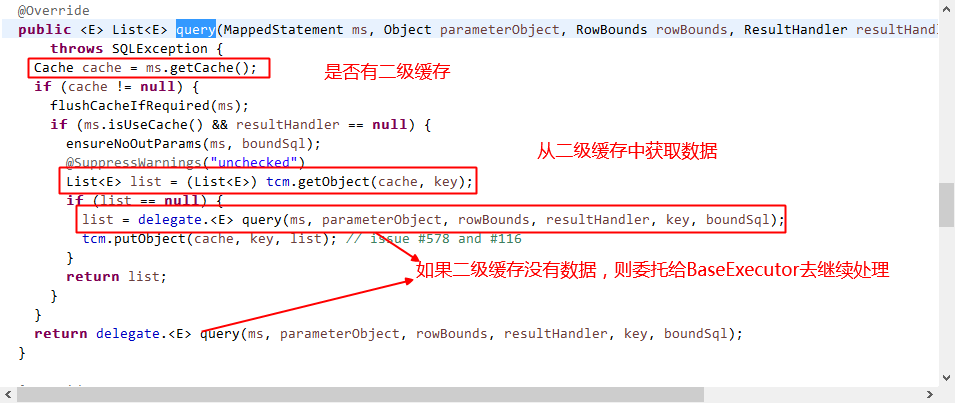
* SqlSession默认会调用CachingExecutor执行器的query方法



这个方法中有两个信息需要了解：一个是BoundSql对象的获取；一个是CacheKey的创建

不过这两个信息暂时先不去处理

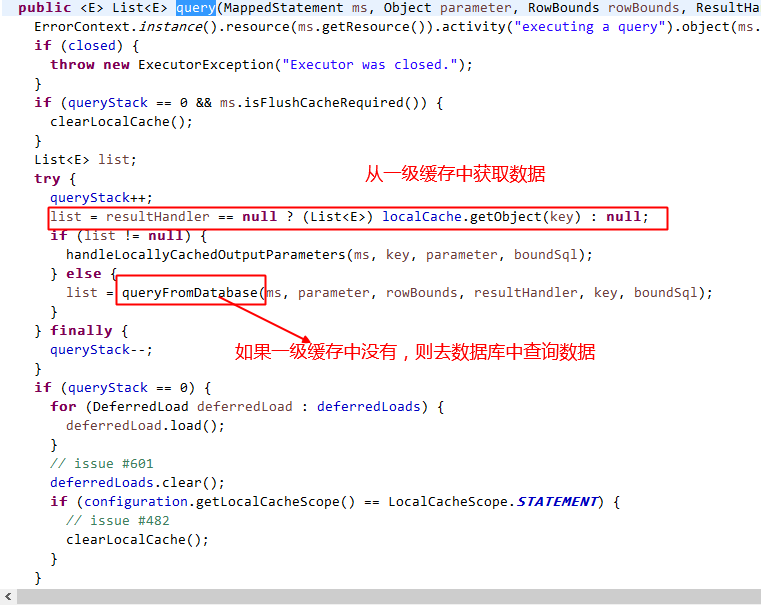
* 继续调用重载的query方法



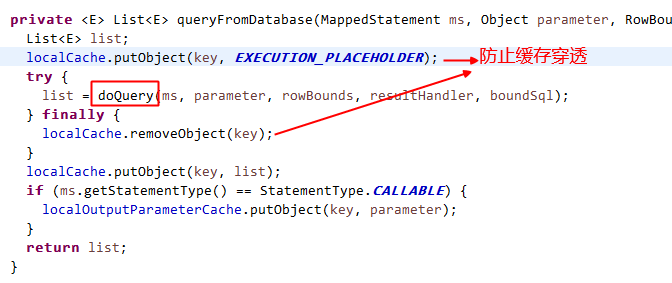
##### BaseExecutor

其实CachingExecutor只是去处理二级缓存数据，如果没有二级缓存，则委托给BaseExecutor去继续处理。

* 进入query方法



* 进入queryFromDatabase方法



这个地方，针对缓存穿透的问题进行了防范：先在缓存中存储一个占位符，这个缓存的数据主要是为了在查询的过程中，对外提供缓存数据，以此来保护数据库。

接下来调用的doQuery方法需要去BaseExecutor的子类SimpleExecutor中去看看了

##### SimpleExecutor



SimpleExecutor的doQuery方法，直接调用的就是JDBC的代码。

StatementHandler的体系结构如下：

