

作者简介

大家好，我是不才陈某。关于我就不过多介绍了，曾经的蚂蚁金服P8老兵，现运营着微信公众号：码猿技术专栏，之所以起这个名字就是想把一门技术从入门到进阶的教程分享出来，我的文章全部都是成体系的，非常适合初学者从入门到深入的学习，专栏如下：

- 《Mybatis 进阶》：
 - 地址：https://mp.weixin.qq.com/mp/appmsgalbum?__biz=MzU3MDAzNDg1MA==&action=getalbum&album_id=1500819225232343046#wechat_redirect
 - PDF：关注微信公众号：码猿技术专栏，回复关键词【Mybatis 进阶】获取。
- 《Spring Boot 进阶》
 - 地址：https://mp.weixin.qq.com/mp/appmsgalbum?__biz=MzU3MDAzNDg1MA==&action=getalbum&album_id=1532834475389288449#wechat_redirect
 - PDF：关注微信公众号：码猿技术专栏，回复关键词【Spring Boot 进阶】获取。
- 《Spring Cloud 进阶》：
 - 地址：https://mp.weixin.qq.com/mp/appmsgalbum?__biz=MzU3MDAzNDg1MA==&action=getalbum&album_id=2042874937312346114#wechat_redirect
 - PDF：关注微信公众号：码猿技术专栏，回复关键词【Spring Cloud 进阶】获取。

另外《Spring Cloud 进阶》这个专栏并没有结束，只是刚介绍了**Spring Cloud Alibaba**体系的几种组件，陈某只是提前整理成PDF供读者学习，后续还会继续更新实战方面的内容，首发公众号，可以扫描下方二维码关注。



另外陈某最近新开了一个公众号，用于每天分享经典面试题，有需要的可以关注一下，二维码如下：



Java后端面试官

微信扫描二维码，关注我的公众号

Mybatis入门之基本CRUD

前言

- 作为一个资深后端码农天天都要和数据库打交道，最早使用的是 **Hibernate**，一个封装性极强的持久性框架。自从接触到 **Mybatis** 就被它的灵活性所折服了，可以自己写 **SQL**，虽然轻量级，但是麻雀虽小，五脏俱全。这篇文章就来讲讲什么是 **Mybatis**，如何简单的使用 **Mybatis**。

什么是 **Mybatis**

- **MyBatis** 是一款优秀的持久层框架，它支持自定义 **SQL**、存储过程以及高级映射。**MyBatis** 免除了几乎所有的 **JDBC** 代码以及设置参数和获取结果集的工作。**MyBatis** 可以通过简单的 **XML** 或注解来配置和映射原始类型、接口和 **Java POJO**（Plain Old Java Objects，普通老式 **Java** 对象）为数据库中的记录。

环境搭建

- 本篇文章使用的环境是 **SpringBoot+Mybatis+Mysql**

Maven 依赖

- **MySQL** 驱动依赖和 **Druid** 连接池的依赖

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.40</version>
    <scope>runtime</scope>
</dependency>

<!--druid连接池-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.9</version>
</dependency>
```

- **Mybatis** 启动包依赖，此处导入的是 **SpringBoot** 和 **Mybatis** 整合启动器的依赖，点击去可以看到，这个启动包依赖了 **mybatis** 和 **mybatis-spring**（**Mybatis** 和 **Spring** 整合的 **Jar** 包），因此使用 **SpringBoot** 之后只需要导入这个启动器的依赖即可。

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

- 以上两个依赖添加成功后，Maven 环境就已经配置完了。

数据库连接池配置（Druid）

- 这个不是本文的重点，而且网上很多教程，我就简单的配置一下，在 SpringBoot 的 `application.properties` 中配置即可。

```
##单一数据源
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/vivachekcloud_
pzhdermyy?useUnicode=true&characterEncoding=UTF-
8&zeroDateTimeBehavior=convertToNull&useSSL=false
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
#初始化连接大小
spring.datasource.druid.initial-size=0
#连接池最大使用连接数量
spring.datasource.druid.max-active=20
#连接池最小空闲
spring.datasource.druid.min-idle=0
#获取连接最大等待时间
spring.datasource.druid.max-wait=6000
spring.datasource.druid.validation-query=SELECT 1
#spring.datasource.druid.validation-query-timeout=6000
spring.datasource.druid.test-on-borrow=false
spring.datasource.druid.test-on-return=false
spring.datasource.druid.test-while-idle=true
#配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
spring.datasource.druid.time-between-connection-validation-runs-millis=60000
#置一个连接在池中最小生存的时间，单位是毫秒
spring.datasource.druid.min-evictable-idle-time-millis=25200000
#spring.datasource.druid.max-evictable-idle-time-millis=
#打开removeAbandoned功能,多少时间内必须关闭连接
spring.datasource.druid.removeAbandoned=true
#1800秒，也就是30分钟
```

```
spring.datasource.druid.remove-abandoned-timeout=1800
#<!-- 1800秒，也就是30分钟 -->
spring.datasource.druid.log-abandoned=true
spring.datasource.druid.filters=mergeStat
#spring.datasource.druid.verifyServerCertificate
#spring.datasource.filters=stat,wall,log4j
# 通过connectProperties属性来打开mergeSql功能：慢SQL记录
spring.datasource.connectionProperties=druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
```

基础概念

- **dao**层：用于存放和数据库交互的文件，Mybatis 的 **interface** 都放在此层
- **service**层：用于存放业务逻辑的文件。

配置 **xml** 文件存放的位置

- Mybatis 中 **xml** 的文件默认是要和 **interface** 放在一个包下的，并且文件的名称要一样。
- 在和 **SpringBoot** 整合后有两种配置方式，下面详细介绍。

application.properties 中设置

- 既然是和 **SpringBoot** 整合，那么万变不离 **xxxAutoConfiguration** 这个配置类了，Mybatis 的配置类就是 **MybatisAutoConfiguration**，如下：

```
@org.springframework.context.annotation.Configuration
@ConditionalOnClass({ SqlSessionFactory.class,
SqlSessionFactoryBean.class })
@ConditionalOnSingleCandidate(DataSource.class)
@EnableConfigurationProperties(MybatisProperties.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
public class MybatisAutoConfiguration implements InitializingBean
{}
```

- 可以看到 **@EnableConfigurationProperties(MybatisProperties.class)** 这行代码，就是将 **properties** 中的属性映射到 **MybatisProperties** 这个成员属性中，因此设置的方式就要看其中的属性。

```
public class MybatisProperties {
    //前缀
    public static final String MYBATIS_PREFIX = "mybatis";

    /**
     * Mybatis配置文件的位置
     */
    private String configLocation;

    /**
     * Mybatis的Mapper的xml文件的位置
     */
    private String[] mapperLocations;
}
```

- 因此设置的方式很简单，如下：

```
## xml文件放置在/src/main/resource/mapper/文件夹下
mybatis.mapper-locations=classpath*:mapper/**/*.xml
```

配置类中设置

- 不是本章重点，后面在讲 Mybatis 和 SpringBoot 整合的文章会涉及到该内容。

配置扫描 Mybatis 的 interface

- 在和 SpringBoot 整合后，扫描 Mybatis 的接口，生成代理对象是一件很简单的事，只需要一个注解即可。

@Mapper

- 该注解标注在 Mybatis 的 `interface` 类上，SpringBoot 启动之后会扫描后会自动生成代理对象。实例如下：

```
@Mapper
public interface UserInfoMapper {

    int insert(UserInfo record);

    int insertSelective(UserInfo record);
}
```

- 缺点：每个 `interface` 都要标注一个，很鸡肋，一个项目中的 `interface` 少说也有上百个吧。

@MapperScan

- `@Mapper` 注解的升级版，标注在配置类上，用于一键扫描 Mybatis 的 `interface`。
- 使用也是很简单的，直接指定接口所在的包即可，如下：

```
@MapperScan({"com.xxx.dao"})
public class ApiApplication {}
```

- `@MapperScan` 和 `@Mapper` 这两个注解千万不要重复使用。
- 优点：一键扫描，不用每个 `interface` 配置。

基本的 crud

- 既然和数据库交互，避免不了 `crud` 操作，就安心做一个妥妥的 `crud boy` 吧。
- 针对 Mybatis 其实有两套方法映射，一个是 XML 文件的方式，一个是注解的方式。但是今天只讲 XML 文件的方式，原因很简单，注解的方式企业不用，谁用谁倒霉，哈哈。

查询

- 查询语句是 MyBatis 中最常用的元素之一——光能把数据存到数据库中价值并不大，还要能重新取出来才有用，多数应用也都是查询比修改要频繁。MyBatis 的基本原则之一是：在每个插入、更新或删除操作之间，通常会执行多个查询操作。因此，MyBatis 在查询和结果映射做了相当多的改进。一个简单查询的 `select` 元素是非常简单的。

```
<select id="selectPersonById" parameterType="int"
resultType="com.myjszl.domain.Person">
    SELECT name,age,id FROM PERSON WHERE ID = #{id}
</select>
```

- 对应的 `interface` 的方法如下：

```
Person selectPersonById(int id);
```

- `<select>` 这个标签有很多属性，比较常用的属性如下：
 - **id**（必填）：在命名空间中唯一的标识符，可以被用来引用这条语句。和 `interface` 中的 `方法名` 要一致。
 - **parameterType**（可选）：将会传入这条语句的参数的类全限定名或别名。这个属性是可选的，因为 **MyBatis** 可以通过类型处理器（**TypeHandler**）推断出具体传入语句的参数，默认值为未设置（unset）。
 - **resultType**：期望从这条语句中返回结果的类全限定名或别名。注意，如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身的类型。**resultType** 和 **resultMap** 之间只能同时使用一个。
 - **resultMap**：对外部 **resultMap** 的命名引用。结果映射是 **MyBatis** 最强大的特性，如果你对其理解透彻，许多复杂的映射问题都能迎刃而解。**resultType** 和 **resultMap** 之间只能同时使用一个。

变更

- 数据变更语句 `insert`，`update` 和 `delete` 的实现非常接近。
- 下面是 `insert`，`update` 和 `delete` 语句的示例：

```
<insert id="insertAuthor">
    insert into Author (id,username,password,email,bio)
    values (#{id},#{username},#{password},#{email},#{bio})
</insert>

<update id="updateAuthor">
    update Author set
        username = #{username},
        password = #{password},
        email = #{email},
        bio = #{bio}
    where id = #{id}
</update>

<delete id="deleteAuthor">
    delete from Author where id = #{id}
</delete>
```


#{}和\${}的区别

- 上面的例子中我们可以看到使用的都是#{}, 关于#{ }和\${ }的区别也是在很多初级工程师的面试最常被问到的, 现在只需要记住区别就是#{ }使用了JDBC的预编译, 可以防止SQL注入, 提高了安全性, \${ }并没有预编译, 安全性不够。在后面Mybatis的源码讲解中将会涉及到为什么一个用了预编译, 一个没用。

自增ID的返回

- 关于Mysql的文章中有提到, 设计一个表最好要有一个自增ID, 无论这个ID你是否用到, 具体原因不在解释, 可以翻看之前的文章。
- 有了自增ID, 插入之后并不能自动返回, 但是我们又需要这个ID值, 那么如何返回呢?
- <insert>标签提供了两个属性用来解决这个问题, 如下:
 - useGeneratedKeys: 设置为true, 表示使用自增主键返回
 - keyProperty: 指定返回的自增主键映射到parameterType的哪个属性中。
- 假设插入Person, 并且person表中的自增主键id需要返回, XML文件如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xxx.dao.PersonMapper">
  <insert id='addPerson' parameterType='com.xxx.domain.Person'
    useGeneratedKeys="true"
    keyProperty="id" >
    insert into person(name,age)
    values(#{name},#{age});
  </insert>
</mapper>
```

SQL 代码片段

- 这个元素可以用来定义可重用的SQL代码片段, 以便在其它语句中使用。参数可以静态地(在加载的时候)确定下来, 并且可以在不同的include元素中定义不同的参数值。比如:

```
<sql id="userColumns">
  ${alias}.id,${alias}.username,${alias}.password </sql>
```

- 这个 SQL 片段可以在其它语句中使用，例如：

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias"
value="t1"/></include>,
    <include refid="userColumns"><property name="alias"
value="t2"/></include>
  from some_table t1
  cross join some_table t2
</select>
```

开启驼峰映射

- DBA 在设计数据库的时候，往往使用的是下划线(_)的方式，比如 `user_id`。但是 Java 是不规范的，我们通常将它转换为 `userId`，这就是驼峰命名方法。
- 但是在使用 Mybatis 查询的时候，比如：

```
<select id='selectById' resultType='com.xxx.doamin.User'>
  select user_id from user_info
</select>
```

- 上面的 `user_id` 和 `User` 中的 `userId` 根本不对应，也就映射不进去，此时查询的结果就是 `userId` 是 `null`，当然我们可以使用别名的方式，SQL 可以改写为 `select user_id as userId from user_info`
- 另外一种方式是不用别名，直接开启 Mybatis 的驼峰映射规则，会自动映射，开启的方式很简单，就是在 `application.properties` 文件配置一下，如下：

```
mybatis.configuration.map-underscore-to-camel-case=true
```

总结

- 本文主要讲了 Mybatis 与 SpringBoot 的整合过程，基本的 crud，各种标签的属性等内容，属于一个入门级别的教程，后续的内容会逐渐深入。
- 另外，MySQL 进阶的教程已经写了五篇文章了，每一篇都是经典，已经出了一个专辑，感兴趣的可以收藏一下 [MySQL 进阶](#)。
- 感谢你的阅读，作者会定时的更新原创文章，如果觉得写的不错的话，可以关注一下本公众号。



Mybatis入门之结果映射

前言

- 上一篇文章介绍了Mybatis基础的CRUD操作、常用的标签、属性等内容，如果对部分不熟悉的朋友可以看[Mybatis入门之基本操作](#)。
- 本篇文章继续讲解Mybatis的结果映射的内容，想要在企业开发中灵活的使用Mybatis，这部分的内容是必须要精通的。

什么是结果映射？

- 简单的来说就是一条SQL查询语句返回的字段如何与Java实体类中的属性相对应。
- 如下一条SQL语句，查询患者的用户id，科室id，主治医生id：

```
<select id='selectPatientInfos'
resultType='com.xxx.domain.PatientInfo'>
    select user_id,dept_id,doc_id from patient_info;
</select>
```

- Java实体类PatientInfo如下：

```
@Data
public class PatientInfo{
    private String userId;
    private String deptId;
    private String docId;
}
```

- 程序员写这条SQL的目的就是想查询出来的`user_id,dept_id,doc_id`分别赋值给实体类中的`userId,deptId,docId`。这就是简单的结果映射。

如何映射？

- Mybatis中的结果映射有很多种方式，下面会逐一介绍。

别名映射

- 这个简单，保持查询的SQL返回的字段和Java实体类一样即可，比如上面例子的SQL可以写成：

```
<select id='selectPatientInfos'
resultType='com.xxx.domain.PatientInfo'>
    select user_id as userId,
    dept_id as deptId,
    doc_id as docId
    from patient_info;
</select>
```

- 这样就能和实体类中的属性映射成功了。

驼峰映射

- Mybatis提供了驼峰命名映射的方式，比如数据库中的`user_id`这个字段，能够自动映射到`userId`属性。那么此时的查询的SQL变成如下即可：

```
<select id='selectPatientInfos'
resultType='com.xxx.domain.PatientInfo'>
    select user_id,dept_id,doc_id from patient_info;
</select>
```

- 如何开启呢？与SpringBoot整合后开启其实很简单，有两种方式，一个是配置文件中开启，一个是配置类开启。

配置文件开启驼峰映射

- 只需要在`application.properties`文件中添加如下一行代码即可：

```
mybatis.configuration.map-underscore-to-camel-case=true
```

配置类中开启驼峰映射【简单了解，后续源码章节着重介绍】

- 这种方式需要你对源码有一定的了解，上一篇入门教程中有提到，Mybatis与Springboot整合后适配了一个starter，那么肯定会有自动配置类，Mybatis的自动配置类是`MybatisAutoConfiguration`，其中有这么一段代码，如下：

```
@Bean
@ConditionalOnMissingBean
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
    SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
    factory.setDataSource(dataSource);
    factory.setVfs(SpringBootVFS.class);
    if (StringUtils.hasText(this.properties.getConfigLocation())) {
        factory.setConfigLocation(this.resourceLoader.getResource(this.properties.getConfigLocation()));
    }
    applyConfiguration(factory);
    if (this.properties.getConfigurationProperties() != null) {
        factory.setConfigurationProperties(this.properties.getConfigurationProperties());
    }
    if (!ObjectUtils.isEmpty(this.interceptors)) {
        factory.setPlugins(this.interceptors);
    }
}
```

- `@ConditionalOnMissingBean`这个注解的意思就是当IOC容器中没有`SqlSessionFactory`这个Bean对象这个配置才会生效；`applyConfiguration(factory)`这行代码就是创建一个`org.apache.ibatis.session.Configuration`赋值给`SqlSessionFactoryBean`。源码分析到这，应该很清楚了，无非就是自己在容器中创建一个`SqlSessionFactory`，然后设置属性即可，如下代码：

```
@Bean("sqlSessionFactory")
public SqlSessionFactory sqlSessionFactory(DataSource
dataSource) throws Exception {
    SqlSessionFactoryBean sqlSessionFactoryBean = new
SqlSessionFactoryBean();
    //设置数据源
    sqlSessionFactoryBean.setDataSource(dataSource);
    //设置xml文件的位置
```

```

        sqlSessionFactoryBean.setMapperLocations(new
PathMatchingResourcePatternResolver().getResources(MAPPER_LOCATORIN)
);

        //创建Configuration
        org.apache.ibatis.session.Configuration configuration = new
org.apache.ibatis.session.Configuration();
        // 开启驼峰命名映射
        configuration.setMapUnderscoreToCamelCase(true);
        configuration.setDefaultFetchSize(100);
        configuration.setDefaultStatementTimeout(30);
        sqlSessionFactoryBean.setConfiguration(configuration);
        //将typeHandler注册到mybatis
        sqlSessionFactoryBean.setTypeHandlers(typeHandlers());
        return sqlSessionFactoryBean.getObject();
    }

```

- 注意：如果对 `SqlSessionFactory` 没有特殊定制，不介意重写，因为这会自动覆盖自动配置类中的配置。

resultMap映射

- 什么是 `resultMap`？简单的说就是一个类似Map的结构，将数据库中的字段和JavaBean中的属性字段对应起来，这样就能做到一一映射了。
- 上述的例子使用 `resultMap` 又会怎么写呢？如下：

```

<!--创建一个resultMap映射-->
<resultMap id="patResultMap" type="com.xxx.domain.PatientInfo">
    <id property="userId" column="user_id" />
    <result property="docId" column="doc_id"/>
    <result property="deptId" column="dept_id"/>
</resultMap>

<!--使用resultMap映射结果到com.xxx.domain.PatientInfo这个Bean中-->
<select id='selectPatientInfos' resultMap='patResultMap'>
    select user_id,dept_id,doc_id from patient_info;
</select>

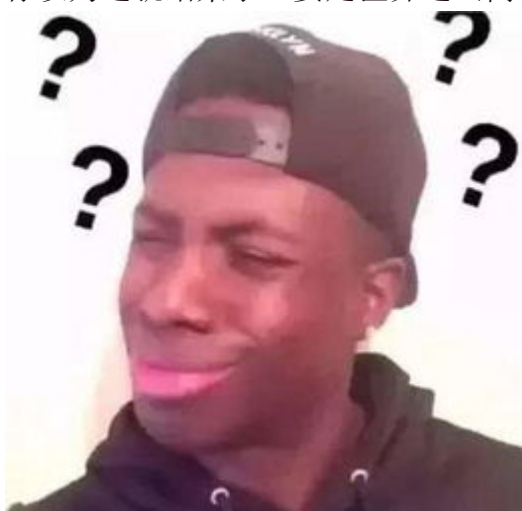
```

- 其实很简单，就是创建一个 `<resultMap>`，然后 `<select>` 标签指定这个 `resultMap` 即可。

- `<resultMap>` 的属性如下：
 - `id`: 唯一标识这个`resultMap`，同一个`Mapper.xml`中不能重复
 - `type`: 指定`JavaBean`的类型，可以是全类名，也可以是别名
- 子标签`<result>`的属性如下：
 - `column`: SQL返回的字段名称
 - `property`: `JavaBean`中属性的名称
 - `javaType`: 一个 `Java` 类的全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。如果你映射到一个 `JavaBean`，`MyBatis` 通常可以推断类型。然而，如果你映射到的是 `HashMap`，那么你应该明确地指定 `javaType` 来保证行为与期望的相一致。
 - `jdbcType`: `JDBC` 类型，所支持的 `JDBC` 类型参见这个表格之后的“支持的 `JDBC` 类型”。只需要在可能执行插入、更新和删除的且允许空值的列上指定 `JDBC` 类型。这是 `JDBC` 的要求而非 `MyBatis` 的要求。如果你直接面向 `JDBC` 编程，你需要对可以为空值的列指定这个类型。
 - `typeHandler`: 这个属性值是一个类型处理器实现类的全限定名，或者是类型别名。
 - `resultMap`: 结果映射的 ID，可以将此关联的嵌套结果集映射到一个合适的对象树中。它可以作为使用额外 `select` 语句的替代方案。

总结

- 以上列举了三种映射的方式，分别是别名映射，驼峰映射、`resultMap`映射。
- 你以为这就结束了？要是世界这么简单多好，做梦吧，哈哈!!!



高级结果映射

- `MyBatis` 创建时的一个思想是：数据库不可能永远是你所想或所需的那个样子。我们希望每个数据库都具备良好的第三范式或 `BCNF` 范式，可惜它们并不都是那样。如果能有一种数据库映射模式，完美适配所有的应用程序，那就太好了，但可惜也没有。而 `ResultMap` 就是 `MyBatis` 对这个问题的答案。

- 我们知道在数据库的关系中一对一，多对一，一对多，多对多的关系，那么这种关系如何在Mybatis中体现并映射成功呢？

关联(association)

- 关联（association）元素处理有一个类型的关系。比如，在我们的示例中，一个员工属于一个部门。关联结果映射和其它类型的映射工作方式差不多。你需要指定目标属性名以及属性的 `javaType`（很多时候 MyBatis 可以自己推断出来），在必要的情况下你还可以设置 `JDBC` 类型，如果你想覆盖获取结果值的过程，还可以设置类型处理器。
- 关联的不同之处是，你需要告诉 MyBatis 如何加载关联。MyBatis 有两种不同的方式加载关联：
 - **嵌套 select 查询**：通过执行另外一个 SQL 映射语句来加载期望的复杂类型。
 - **嵌套结果映射**：使用嵌套的结果映射来处理连接结果的重复子集。
- 首先，先让我们来看看这个元素的属性。你将会发现，和普通的结果映射相比，它只在 `select` 和 `resultMap` 属性上有所不同。
 - **property**：映射到列结果的字段或属性。如果用来匹配的 `JavaBean` 存在给定名字的属性，那么它将会被使用。
 - **javaType**：一个 Java 类的完全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）
jdbcType：JDBC 类型，只需要在可能执行插入、更新和删除的且允许空值的列上指定 JDBC 类型
 - **typeHandler**：使用这个属性，你可以覆盖默认的类型处理器。这个属性值是一个类型处理器实现类的完全限定名，或者是类型别名。
column：数据库中的列名，或者是列的别名。一般情况下，这和传递给 `resultSet.getString(columnName)` 方法的参数一样。注意：在使用复合主键的时候，你可以使用 `column="{prop1=col1,prop2=col2}"` 这样的语法来指定多个传递给嵌套 Select 查询语句的列名。这会使得 `prop1` 和 `prop2` 作为参数对象，被设置为对应嵌套 Select 语句的参数。
 - **select**：用于加载复杂类型属性的映射语句的 ID，它会从 `column` 属性指定的列中检索数据，作为参数传递给目标 `select` 语句。具体请参考下面的例子。注意：在使用复合主键的时候，你可以使用 `column="{prop1=col1,prop2=col2}"` 这样的语法来指定多个传递给嵌套 Select 查询语句的列名。这会使得 `prop1` 和 `prop2` 作为参数对象，被设置为对应嵌套 Select 语句的参数。
 - **fetchType**：可选的。有效值为 `lazy` 和 `eager`。指定属性后，将在映射中忽略全局配置参数 `lazyLoadingEnabled`，使用属性的值。

例子

- 一对一的关系比如：一个员工属于一个部门，那么数据库表就会在员工表中加一个部门的id作为逻辑外键。
- 创建员工JavaBean

```
@Data
public class User {
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private Integer deptId;
    //部门
    private Department department;
}
```

- 部门JavaBean

```
@Data
public class Department {
    private Integer id;
    private String name;
}
```

- 那么我们要查询所有的用户信息和其所在的部门信息，此时的sql语句为：`select * from user u left join department d on u.department_id=d.id;`。但是我们在mybatis中如果使用这条语句查询，那么返回的结果类型是什么呢？如果是User类型的，那么查询结果返回的还有Department类型的数据，那么肯定会对应不上的。此时<resultMap>来了，它来了!!!

关联的嵌套 **Select** 查询【可以忽略】

- 查询员工和所在的部门在Mybatis如何写呢？代码如下：

```
<resultMap id="userResult" type="com.xxx.domain.User">
    <id column="id" property="id"/>
    <result column="password" property="password"/>
    <result column="age" property="age"/>
    <result column="username" property="username"/>
```

```

<result column="dept_id" property="deptId"/>
<!--关联查询，select嵌套查询-->
<association property="department" column="dept_id"
javaType="com.xxx.domain.Department" select="selectDept"/>
</resultMap>

<!--查询员工-->
<select id="selectUser" resultMap="userResult">
    SELECT * FROM user WHERE id = #{id}
</select>

<!--查询部门-->
<select id="selectDept" resultType="com.xxx.domain.Department ">
    SELECT * FROM department WHERE ID = #{id}
</select>

```

- 就是这么简单，两个select语句，一个用来加载员工，一个用来加载部门。
- 这种方式虽然很简单，但在大型数据集或大型数据表上表现不佳。这个问题被称为N+1 查询问题。概括地讲，N+1 查询问题是这样子的：
 - 你执行了一个单独的 SQL 语句来获取结果的一个列表（就是+1）。
 - 对列表返回的每条记录，你执行一个 select 查询语句来为每条记录加载详细信息（就是N）。
- 这个问题会导致成百上千的 SQL 语句被执行。有时候，我们不希望产生这样的后果。

关联的嵌套结果映射【重点】

- `<association>` 标签中还可以直接嵌套结果映射，此时的Mybatis的查询如下：

```

<!-- 定义resultMap -->
<resultMap id="UserDepartment" type="com.xxx.domain.User" >
    <id column="user_id" property="id"/>
    <result column="password" property="password"/>
    <result column="age" property="age"/>
    <result column="username" property="username"/>
    <result column="dept_id" property="deptId"/>

    <!--
        property: 指定User中对应的部门属性名称
        javaType: 指定类型，可以是全类名或者别名
    -->

```

```

-->
<association property="department"
javaType="com.xx.domain.Department">
    <!--指定Department中的属性映射，这里也可以使用单独拎出来，然后使用
association中的resultMap属性指定-->
        <id column="id" property="id"/>
        <result column="dept_name" property="name"/>
    </association>
</resultMap>

<!--
    resultMap: 指定上面resultMap的id的值
-->
<select id="findUserAndDepartment" resultMap="UserDepartment">
    select
    u.id as user_id,
    u.dept_id,
    u.name,
    u.password,
    u.age,
    d.id,
    d.name as dept_name
    from user u left join department d on u.department_id=d.id
</select>

```

总结

- 至此有一个类型的关联已经完成了，学会一个 `<association>` 使用即能完成。
- 注意： 关联的嵌套 Select 查询不建议使用，**N+1**是个重大问题，虽说Mybatis提供了延迟加载的功能，但是仍然不建议使用，企业开发中也是不常用的。

集合collection

- 集合，顾名思义，就是处理有很多个类型的关联。
- 其中的属性和 `association` 中的属性类似，不再重复了。
- 比如这样一个例子： 查询一个部门中的全部员工，查询SQL如何写呢？如下：

```

select * from department d left join user u on
u.department_id=d.id;

```

- 此时的 `User` 实体类如下：

```
@Data
public class User {
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private Integer deptId;
}
```

- 此时的 `Department` 实体类如下：

```
@Data
public class Department {
    private Integer id;
    private String name;
    private List<User> users;
}
```

- 和 `association` 类似，同样有两种方式，我们可以使用嵌套 `Select` 查询，或基于连接的嵌套结果映射集合。

集合的嵌套 `Select` 查询【可以忽略】

- 不太重要，查询如下：

```
<resultMap id="deptResult" type="com.xxx.domain.Department">
    <!--指定Department中的属性映射，这里也可以使用单独拎出来，然后使用
    association中的resultMap属性指定-->
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <!--
    ofType: 指定实际的JavaBean的全类型或者别名
    select: 指定嵌套的select查询
    javaType: 集合的类型，可以不写，Mybatis可以推测出来
-->
    <collection property="users" javaType="java.util.ArrayList"
    column="id" ofType="com.xxx.doamin.User" select="selectByDeptId"/>
</resultMap>
```

```

<select id="selectDept" resultMap="deptResult">
    SELECT * FROM department WHERE ID = #{id}
</select>

<select id="selectByDeptId" resultType="com.xxx.domain.User">
    SELECT * FROM user WHERE dept_id = #{id}
</select>

```

- 注意：这里出现了一个不同于 `association` 的属性 `ofType`，这个属性非常重要，它用来将 `JavaBean`（或字段）属性的类型和集合存储的类型区分开来。

集合的嵌套结果映射【重点】

- 现在你可能已经猜到了集合的嵌套结果映射是怎样工作的——除了新增的 `ofType` 属性，它和关联的完全相同。
- 此时的Mybatis查询如下：

```

<!--部门的resultMap-->
<resultMap id="deptResult" type="com.xxx.domain.Department">
    <!--指定Department中的属性映射，这里也可以使用单独拎出来，然后使用
    association中的resultMap属性指定-->
        <id column="dept_id" property="id"/>
        <result column="dept_name" property="name"/>
    <!--
    ofType: 指定实际的JavaBean的全类型或者别名
    resultMap: 指定员工的resultMap
-->
    <collection property="users" ofType="com.xxx.doamin.User"
    resultMap='userResult'/>
</resultMap>

<!--员工的resultMap-->
<resultMap id="userResult" type="com.xxx.domain.User">
    <id column="user_id" property="id"/>
    <result column="password" property="password"/>
    <result column="age" property="age"/>
    <result column="username" property="username"/>
</resultMap>

<select id="selectDeptById" resultType="com.xxx.domain.Department">

```

```
select
d.id as dept_id,
d.name as dept_name,
u.id as user_id,
u.password,
u.name
from department d left join user u on u.department_id=d.id
where d.id=#{id}
</select>
```

总结

- 至此Mybatis第二弹之结果映射已经写完了，如果觉得作者写的不错，给个在看关注一波，后续还有更多精彩内容推出。



Mybatis动态SQL，你真的会了吗？

前言

- 通过前两篇的文章我们了解了Mybatis基本的CRUD操作、一些基本标签的属性以及如何映射结果，感兴趣的可以看我的前两篇文章，分别是[Mybatis入门之基础操](#)

作和Mybatis结果映射，你射准了吗？，如果有什么疑问的地方可以在文章下方留言，作者统一回复。

- 这篇文章就来聊一聊Mybatis的动态SQL，在实际的开发中Mybatis的这项功能是非常重要的，至于什么是动态SQL？如何实现动态SQL？下面文章将会详细介绍。

什么是动态SQL？

- 动态 SQL 是 MyBatis 的强大特性之一。顾名思义，就是会动的SQL，即是能够灵活的根据某种条件拼接出完整的SQL语句。这种类似于MySQL中的 `case when then else then end....` 这种语法，能够根据某种条件动态的拼接出需要的SQL。
- 至于Mybatis如何实现动态SQL呢，Mybatis提供了非常多的标签，能够让我们在XML文件中灵活的运用这些标签达到拼接SQL的目的。

常用的标签

- Mybatis为了能够让开发者灵活的写SQL也是费了一番功夫，定义了很多的标签和语法，下面将会一一介绍。

if

- 虽然英文不太好，但是在这么简单的不会不知道是如果的意思吧，Java语法中也有，只有判断条件为 `true` 才会执行其中的SQL语句。
- 举个栗子：HIS系统中医护人员需要根据特定条件筛选患者，比如住院号，床位，性别等。当然这些条件并不是必填的，具体的功能截图如下：

<input type="text" value="所有科室"/>	<input type="text" value="所有医生"/>	<input type="text" value="住院号"/>	<input type="text" value="姓名"/>	<input type="text" value="床号"/>	<input type="text" value="入院时间"/>	<input type="button" value="查询"/>	<input type="button" value="重置"/>
-----------------------------------	-----------------------------------	----------------------------------	---------------------------------	---------------------------------	-----------------------------------	-----------------------------------	-----------------------------------

- 以上截图中的条件筛选并不是必填的，因此我们不能在SQL中固定，要根据前端是否传值来判断是否需要加上这个条件。那么此时查询语句如何写呢？如下：

```

<select id='selectPats' resultType='com.xxx.domain.PatientInfo'>
    select * from patient_info
    where status=1
    <!--前端传来的住院号不为null，表示需要根据住院号筛选，此时where语句就需要加上
    这个条件-->
    <if test="iptNum!=null">
        and ipt_num=#{iptNum}
    </if>

    <!--床位号筛选-->
    <if test="bedNum!=null">
        and bed_num=#{bedNum}
    </if>
</select>

```

- `<if>` 标签中的属性 `test` 用来指定判断条件，那么问题来了，上面的例子中的 `test` 中判断条件都是一个条件，如果此时变成两个或者多个条件呢？和SQL的语法类似，`and` 连接即可，如下：

```

<if test="bedNum!=null and bedNum!='' ">
    and bed_num=#{bedNum}
</if>

```

choose、when、otherwise

- 有时候，我们不想使用所有的条件，而只是想从多个条件中选择一个使用。针对这种情况，MyBatis 提供了 `choose` 元素，它有点像 Java 中的 `switch` 语句。
- 还是上面的例子改变一下：此时只能满足一个筛选条件，如果前端传来住院号就只按照住院号查找，如果传来床位号就只按照床位号筛选，如果什么都没传，就筛选所有在院的。此时的查询如下：

```

<select id="selectPats"
    resultType="com.xxx.domain.PatientInfo">
    select * from patient_info where 1=1
    <choose>
        <!--住院号不为null时，根据住院号查找-->
        <when test="iptNum != null">
            AND ipt_num=#{iptNum}
        </when>
        <!--床位号不是NULL-->
        <when test="bedNum != null">

```



```

        AND bed_num = #{bedNum}
    </when>
    <otherwise>
        AND status=1
    </otherwise>
</choose>
</select>

```

- MyBatis 提供了 `choose` 元素，按顺序判断 `when` 中的条件是否成立，如果有一个成立，则 `choose` 结束。当 `choose` 中所有 `when` 的条件都不满足时，则执行 `otherwise` 中的 sql。类似于 Java 的 `switch` 语句，`choose` 为 `switch`，`when` 为 `case`，`otherwise` 则为 `default`。

where

- 举个栗子：对于 `choose` 标签的例子中的查询，如果去掉 `where` 后的 `1=1` 此时的 SQL 语句会变成什么样子，有三种可能的 SQL，如下：

```

select * from patient_info where AND ipt_num=#{iptNum};

select * from patient_info where AND bed_num = #{bedNum};

select * from patient_info where AND status=1;

```

- 发生了什么，以上三条 SQL 语句对吗？很显然是不对的，显然 `where` 后面多了个 `AND`。如何解决呢？此时就要用到 `where` 这个标签了。
- `where` 元素只会在子元素返回任何内容的前提下才插入 `WHERE` 子句。而且，若子句的开头为 `AND` 或 `OR`，`where` 元素也会将它们去除。
- 此时的查询改造如下：

```

<select id="selectPats"
    resultType="com.xxx.domain.PatientInfo">
    select * from patient_info
    <where>
        <choose>
            <!--住院号不为null时，根据住院号查找-->
            <when test="iptNum != null">
                AND ipt_num=#{iptNum}
            </when>
            <!--床位号不是NULL-->

```

```
<when test="bedNum != null">
    AND bed_num = #{bedNum}
</when>
<otherwise>
    AND status=1
</otherwise>
</choose>
</where>
</select>
```

foreach

- **foreach** 是用来对集合的遍历，这个和Java中的功能很类似。通常处理SQL中的 **in** 语句。
- **foreach** 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（**item**）和索引（**index**）变量。它也允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符。这个元素也不会错误地添加多余的分隔符
- 你可以将任何可迭代对象（如 **List**、**Set** 等）、**Map** 对象或者数组对象作为集合参数传递给 **foreach**。当使用可迭代对象或者数组时，**index** 是当前迭代的序号，**item** 的值是本次迭代获取到的元素。当使用 **Map** 对象（或者 **Map.Entry** 对象的集合）时，**index** 是键，**item** 是值。
- 例子如下：

```
<select id="selectPats" resultType="com.xxx.domain.PatientInfo">
    SELECT *
    FROM patient_info
    WHERE ID in
    <foreach item="item" index="index" collection="list"
        open="(" separator="," close=")">
        #{item}
    </foreach>
</select>
```

- 改标签中的各个属性的含义如下：

属性	含义
item	表示在迭代过程中每一个元素的别名
index	表示在迭代过程中每次迭代到的位置（下标）
open	前缀

属性	含义
close	后缀
separator	分隔符，表示迭代时每个元素之间以什么分隔

set

- 讲这个标签之前，先看下面这个例子：

```
<update id="updateStudent" parameterType="Object">
    UPDATE STUDENT
    SET NAME = #{name},
    MAJOR = #{major},
    HOBBY = #{hobby}
    WHERE ID = #{id};
</update>

<update id="updateStudent" parameterType="Object">
    UPDATE STUDENT SET
    <if test="name!=null and name!='' ">
        NAME = #{name},
    </if>
    <if test="hobby!=null and hobby!='' ">
        MAJOR = #{major},
    </if>
    <if test="hobby!=null and hobby!='' ">
        HOBBY = #{hobby}
    </if>
    WHERE ID = #{id};
</update>
```

- 上面的例子中没有使用 **if** 标签时，如果有一个参数为 **null**，都会导致错误。当在 **update** 语句中使用 **if** 标签时，如果最后的 **if** 没有执行，则或导致逗号多余错误。使用 **set** 标签可以将动态的配置 **set** 关键字，和剔除追加到条件末尾的任何不相关的逗号。
- 使用 **set+if** 标签修改后，如果某项为 **null** 则不进行更新，而是保持数据库原值。此时的查询如下：

```
<update id="updateStudent" parameterType="Object">
    UPDATE STUDENT
    <set>
```

```

        <if test="name!=null and name!='' ">
            NAME = #{name},
        </if>
        <if test="hobby!=null and hobby!='' ">
            MAJOR = #{major},
        </if>
        <if test="hobby!=null and hobby!='' ">
            HOBBY = #{hobby}
        </if>
    </set>
    WHERE ID = #{id};
</update>

```

sql

- 在实际开发中会遇到许多相同的SQL，比如根据某个条件筛选，这个筛选很多地方都能用到，我们可以将其抽取出来成为一个公用的部分，这样修改也方便，一旦出现了错误，只需要改这一处便能处处生效了，此时就用到了 `<sql>` 这个标签了。
- 当多种类型的查询语句的查询字段或者查询条件相同时，可以将其定义为常量，方便调用。为求 `<select>` 结构清晰也可将 `sql` 语句分解。如下：

```

<!-- 查询字段 -->
<sql id="Base_Column_List">
    ID,MAJOR,BIRTHDAY,AGE,NAME,HOBBY
</sql>

<!-- 查询条件 -->
<sql id="Example_where_clause">
    where 1=1
    <trim suffixOverrides=",">
        <if test="id != null and id !=''">
            and id = #{id}
        </if>
        <if test="major != null and major != ''">
            and MAJOR = #{major}
        </if>
        <if test="birthday != null ">
            and BIRTHDAY = #{birthday}
        </if>
        <if test="age != null ">
            and AGE = #{age}
        </if>
    </trim>
</sql>

```

```

        <if test="name != null and name != ''">
            and NAME = #{name}
        </if>
        <if test="hobby != null and hobby != ''">
            and HOBBY = #{hobby}
        </if>
    </trim>
</sql>

```

include

- 这个标签和 `<sql>` 是天仙配，是共生的，`include` 用于引用 `sql` 标签定义的常量。比如引用上面 `sql` 标签定义的常量，如下：

```

<select id="selectAll" resultMap="BaseResultMap">
    SELECT
    <include refid="Base_Column_List" />
    FROM student
    <include refid="Example_where_clause" />
</select>

```

- `refid` 这个属性就是指定 `<sql>` 标签中的 `id` 值（唯一标识）。

总结

- 至此，Mybatis 动态 SQL 中常用的标签就已经介绍完了，这部分的内容在实际工作中是必须会用到的，除非你们公司不用 Mybatis。

拓展一下

- 前面介绍了动态 SQL 的一些标签以及属性，相信看完之后应该能够灵活的应用了，但是在实际开发中还是有一些奇技淫巧的，陈某今天简单的讲几个。

Mybatis 中如何避免魔数

- 开过阿里巴巴开发手册的大概都知道代码中是不允许出现魔数的，何为魔数？简单的说就是一个数字，一个只有你知道，别人不知道这个代表什么意思的数字。通常我们在 Java 代码中都会定义一个常量类专门定义这些数字。

- 比如获取医生和护士的权限，但是医生和护士的权限都不相同，在这条SQL中肯定需要根据登录的类型 `type` 来区分，比如 `type=1` 是医生，`type=2` 是护士，估计一般人会这样写：

```
<if test="type!=null and type==1">
    -- ....获取医生的权限
</if>

<if test="type!=null and type==2">
    -- ....获取护士的权限
</if>
```

- 这样写也没什么错，但是一旦这个 `type` 代表的含义变了，那你是不是涉及到的SQL都要改一遍。
- 开发中通常定义一个常量类，如下：

```
package com.xxx.core.Constants;
public class CommonConstants{
    //医生
    public final static int DOC_TYPE=1;

    //护士
    public final static int NUR_TYPE=2;
}
```

- 那么此时的SQL应该如何写呢？如下：

```
<if test="type!=null and
type==@com.xxx.core.Constants.CommonConstants@DOC_TYPE">
    -- ....获取医生的权限
</if>

<if test="type!=null and
type==@com.xxx.core.Constants.CommonConstants@NUR_TYPE">
    -- ....获取护士的权限
</if>
```

- 就是这么简单，就是 `@+全类名+@+常量`。

- 除了调用常量类中的常量，还可以类中的方法，很少用到，不再介绍了，感兴趣的可以问下度娘。

如何引用其他XML中的SQL片段

- 实际开发中你可能遇到一个问题，比如这个 `resultMap` 或者这个 `<sql>` 片段已经在另外一个 `xxxMapper.xml` 中已经定义过了，此时当前的xml还需要用到，难不成我复制一份？小白什么也不问上来就复制了，好吧，后期修改来了，每个地方都需要修改了。难受不？
- 其实Mybatis中也是支持引用其他Mapper文件中的SQL片段的。其实很简单，比如你在 `com.xxx.dao.xxMapper` 这个Mapper的XML中定义了一个SQL片段如下：

```
<sql id="Base_Column_List">
    ID,MAJOR,BIRTHDAY,AGE,NAME,HOBBY
</sql>
```

- 此时我在 `com.xxx.dao.PatinetMapper` 中的XML文件中需要引用，如下：

```
<include refid="com.xxx.dao.xxMapper.Base_Column_List"></include>
```

- 如此简单，类似于Java中的全类名。
- `<select>` 标签中的 `resultMap` 同样可以这么引用，和上面引用的方式一样，不再赘述了。

总结

- 好了，Myabtis的动态SQL的内容已经介绍完了，你会了吗？每日来看看，下面会有更多精彩内容！！！！

- 如果 觉得写的不错的，点点在看，关注一波不迷路，每天都会有精彩内容分享。



Mybatis几种传参方式，你了解吗？

前言

- 前几天恰好面试一个应届生，问了一个很简单的问题：你了解过Mybatis中有几种传参方式吗？
- 没想到其他问题回答的很好，唯独这个问题一知半解，勉强回答了其中两种方式。
- 于是这篇文章就来说一说Mybatis传参的几种常见方式，给正在面试或者准备面试的朋友巩固一下。

单个参数

- 单个参数的传参比较简单，可以是任意形式的，比如`#{a}`、`#{b}`或者`#{param1}`，但是为了开发规范，尽量使用和入参时一样。
- Mapper如下：

```
UserInfo selectById(String userId);
```

- XML如下：


```
<select id="selectByUserId"
resultType="cn.cb.demo.domain.UserInfo">
    select * from user_info where user_id=#{userId} and
status=1
</select>
```

多个参数

- 多个参数的情况下有很多种传参的方式，下面一一介绍。

使用索引【不推荐】

- 多个参数可以使用类似于索引的方式传值，比如 `#{param1}` 对应第一个参数，`#{param2}` 对应第二个参数.....
- Mapper方法如下：

```
UserInfo selectByUserIdAndStatus(String userId,Integer status);
```

- XML如下：

```
<select id="selectByUserIdAndStatus"
resultType="cn.cb.demo.domain.UserInfo">
    select * from user_info where user_id=#{param1} and
status=#{param2}
</select>
```

- 注意：由于开发规范，此种方式不推荐开发中使用。

使用@Param

- `@Param` 这个注解用于指定key，一旦指定了key，在SQL中即可对应的key入参。
- Mapper方法如下：

```
UserInfo selectByUserIdAndStatus(@Param("userId") String
userId,@Param("status") Integer status);
```

- XML如下：

```
<select id="selectByUserIdAndStatus"
resultType="cn.cb.demo.domain.UserInfo">
    select * from user_info where user_id=#{userId} and
status=#{status}
</select>
```

使用Map

- Mybatis底层就是将入参转换成Map，入参传Map当然也行，此时#{key}中的key就对应Map中的key。
- Mapper中的方法如下：

```
UserInfo selectByUserIdAndStatusMap(Map<String,Object> map);
```

- XML如下：

```
<select id="selectByUserIdAndStatusMap"
resultType="cn.cb.demo.domain.UserInfo">
    select * from user_info where user_id=#{userId} and
status=#{status}
</select>
```

- 测试如下：

```
@Test
void contextLoads() {
    Map<String,Object> map=new HashMap<>();
    map.put("userId","1222");
    map.put("status",1);
    UserInfo userInfo =
userMapper.selectByUserIdAndStatusMap(map);
    System.out.println(userInfo);
}
```

POJO【推荐】

- 多个参数可以使用实体类封装，此时对应的key就是属性名称，注意一定要有get方法。
- Mapper方法如下：

```
UserInfo selectByEntity(UserInfoReq userInfoReq);
```

- XML如下:

```
<select id="selectByEntity"
resultType="cn.cb.demo.domain.UserInfo">
    select * from user_info where user_id=#{userId} and
status=#{status}
</select>
```

- 实体类如下:

```
@Data
public class UserInfoReq {
    private String userId;
    private Integer status;
}
```

List传参

- List传参也是比较常见的，通常是SQL中的in。
- Mapper方法如下:

```
List<UserInfo> selectList( List<String> userIds);
```

- XML如下:

```
<select id="selectList" resultMap="userResultMap">
    select * from user_info where status=1
    and user_id in
        <foreach collection="list" item="item" open="("
separator="," close=")" >
            #{item}
        </foreach>
</select>
```

数组传参

- 这种方式类似List传参，依旧使用 `foreach` 语法。
- Mapper方法如下：

```
List<UserInfo> selectList( String[] userIds);
```

- XML如下：

```
<select id="selectList" resultMap="userResultMap">
    select * from user_info where status=1
    and user_id in
        <foreach collection="array" item="item" open="("
separator="," close=")" >
            #{item}
        </foreach>
</select>
```

总结

- 以上几种传参的方式在面试或者工作中都会用到，不了解的朋友可以收藏下。
- Mybatis专题文章写到这里也算是到了尾声，后期准备再写写Mybatis的面经，如果觉得作者写的不错，欢迎关注分享。



Myabtis中Mapper接口的方法为什么不能重载？

前言

- 在初入门 **Mybatis** 的时候可能都犯过一个错误，那就是在写 **Mapper** 接口的时候都重载过其中的方法，但是运行起来总是报错，那时候真的挺郁闷的，但是自己也查不出来原因，只能默默的改了方法名，哈哈，多么卑微的操作。
- 今天就写一篇文章从源码角度为大家解惑为什么 **Mybatis** 中的方法不能重载？

环境配置

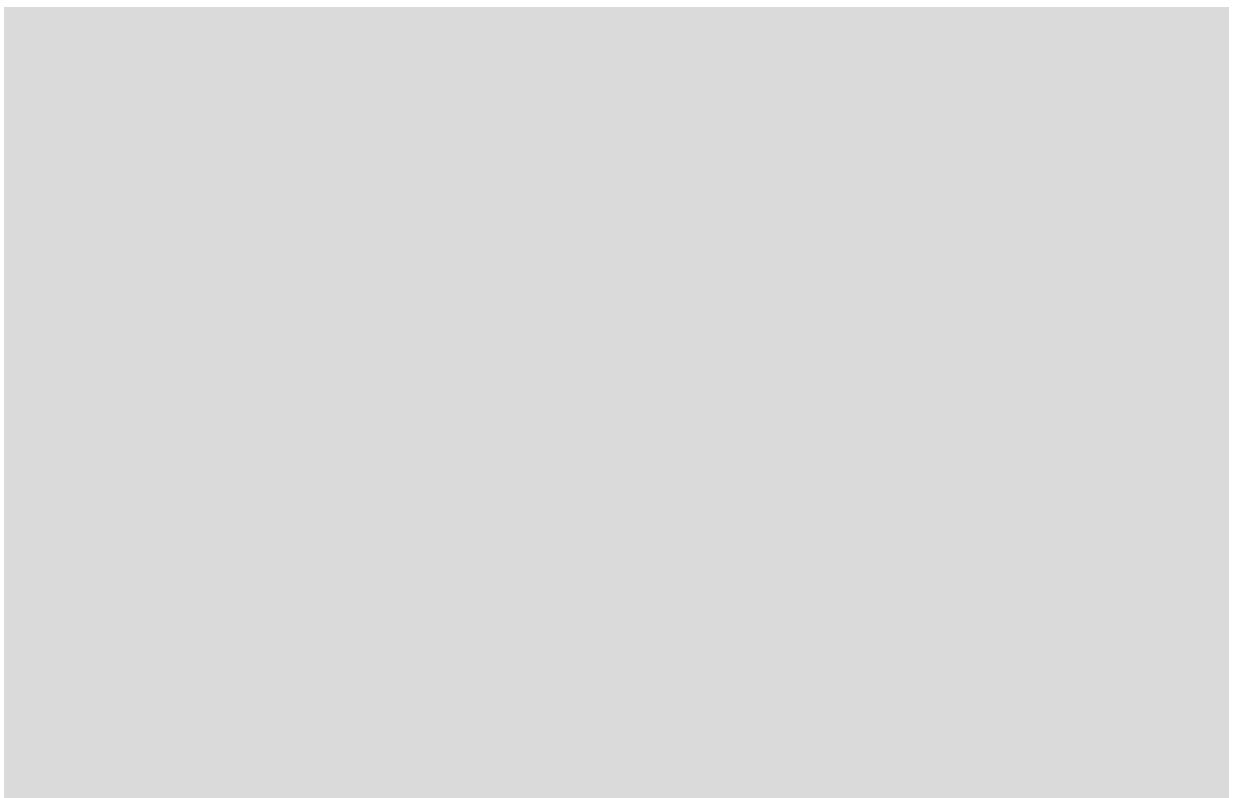
- 本篇文章讲的一切内容都是基于 **Mybatis3.5** 和 **SpringBoot-2.3.3.RELEASE**。

错误示范

- 举个栗子：假设现在有两个需求，一个是根据用户的id筛选用户，一个是根据用户的性别筛选，此时在 **Mapper** 中重载的方法如下：

```
public interface UserMapper {  
    List<UserInfo> selectList(@Param("userIds") List<String>  
        userIds);  
  
    List<UserInfo> selectList(Integer gender);  
}
```

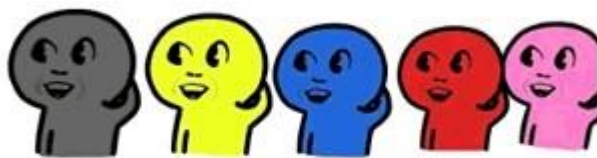
- 这个并没有什么错误，但是启动项目，报出如下的错误：



Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'sqlSessionFactory' defined in class path resource [org/mybatis/spring/boot/autoconfigure/MybatisAutoConfiguration.class]: Bean instantiation via factory method failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instantiate [org.apache.ibatis.session.SqlSessionFactory]: Factory method 'sqlSessionFactory' threw exception; nested exception is org.springframework.core.NestedIOException: Failed to parse mapping resource: 'file [H:\work_project\demo\target\classes\mapper\UserInfoMapper.xml]'; nested exception is org.apache.ibatis.builder.BuilderException: Error parsing Mapper XML. The XML location is 'file [H:\work_project\demo\target\classes\mapper\UserInfoMapper.xml]'. Cause: java.lang.IllegalArgumentException: Mapped Statements collection already contains value for cn.cb.demo.dao.UserMapper.selectList. please check file [H:\work_project\demo\target\classes\mapper\UserInfoMapper.xml] and file [H:\work_project\demo\target\classes\mapper\UserInfoMapper.xml] at org.springframework.beans.factory.support.ConstructorResolver.instantiate(ConstructorResolver.java:655) at org.springframework.beans.factory.support.ConstructorResolver.instantiateUsingFactoryMethod(ConstructorResolver.java:635) at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.instantiateUsingFactoryMethod(AbstractAutowireCapableBeanFactory.java:1336) at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBeanInstance(AbstractAutowireCapableBeanFactory.java:1176) at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:556) at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:516) at org.springframework.beans.factory.support.AbstractBeanFactory.lambda\$doGetBean\$0(AbstractBeanFactory.java:324)

```
at
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:226)
at
org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:322)
at
org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:202)
at
org.springframework.beans.factory.config.DependencyDescriptor.resolveCandidate(DependencyDescriptor.java:276)
at
org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:1307)
at
org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListableBeanFactory.java:1227)
at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireByType(AbstractAutowireCapableBeanFactory.java:1509)
... 81 more
```

- 这么一大串什么意思？懵逼了~



五脸懵逼队

- 大致的意思：`cn.cb.demo.dao.UserMapper.selectList`这个id已经存在了，导致创建`sqlSessionFactory`失败。

为什么不能重载？

- 通过上面的异常提示可以知道创建 `sqlSessionFactory` 失败了，这个想必已经不再陌生吧，顾名思义，就是创建 `SqlSession` 的工厂。
- Springboot与Mybatis会有一个启动器的自动配置类 `MybatisAutoConfiguration`，其中有一段代码就是创建 `sqlSessionFactory`，如下图：

```
@Bean
@ConditionalOnMissingBean
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
    SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
    factory.setDataSource(dataSource);
    factory.setVfs(SpringBootVFS.class);
    if (StringUtils.hasText(this.properties.getConfigLocation())) {
        factory.setConfigLocation(this.resourceLoader.getResource(this.properties.getConfigLocation()));
    }
    applyConfiguration(factory);
    if (this.properties.getConfigurationProperties() != null) {
        factory.setConfigurationProperties(this.properties.getConfigurationProperties());
    }
    if (!ObjectUtils.isEmpty(this.interceptors)) {
        factory.setPlugins(this.interceptors);
    }
    if (this.databaseIdProvider != null) {
        factory.setDatabaseIdProvider(this.databaseIdProvider);
    }
}
```

- 既然是创建失败，那么肯定是这里出现异常了，这里的大致思路就是：

解析XML文件和Mapper接口，将Mapper中的方法与XML文件中<select>、<insert>等标签一一对应，那么Mapper中的方法如何与XML中<select>这些标签对应了，当然是唯一的id对应了，具体如何这个id的值是什么，如何对应？下面一一讲解。

- 如上图的 `SqlSessionFactory` 的创建过程中，前面的部分代码都是设置一些配置，并没有涉及到解析XML的内容，因此答案肯定是在最后一行 `return factory.getObject();`，于是此处打上断点，一点点看。于是一直到了 `org.mybatis.spring.SqlSessionFactoryBean#buildSqlSessionFactory` 这个方法中，其中一段代码如下：

```
try {
    XMLMapperBuilder xmlMapperBuilder = new XMLMapperBuilder(mapperLocation.getInputStream(),
        targetConfiguration, mapperLocation.toString(), targetConfiguration.getSqlFragments());
    xmlMapperBuilder.parse();
} catch (Exception e) {
    throw new NestedIOException("Failed to parse mapping resource: '" + mapperLocation + "'");
} finally {
```

- 这里的 `xmlMapperBuilder.parse();` 就是解析XML文件与Mapper接口，继续向下看。
- 略过不重要的代码，在 `org.apache.ibatis.builder.xml.XMLMapperBuilder#configurationElement`

这个方法中有一行重要的代码，如下图：

```
builderAssistant.setCurrentNamespace(namespace); builderAssistant: MapperBuilderAssistant
cacheRefElement(context.evalNode("cache-ref"));
cacheElement(context.evalNode("cache"));
parameterMapElement(context.evalNodes( expression: "/mapper/parameterMap"));
resultMapElements(context.evalNodes( expression: "/mapper/resultMap"));
sqlElement(context.evalNodes( expression: "/mapper/sql"));
buildStatementFromContext(context.evalNodes( expression: "select|insert|update|delete"));
} catch (Exception e) {
throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource +
}
```

- 此处就是根据XML文件中的 `select|insert|update|delete` 这些标签开始构建 `MappedStatement` 了。继续跟进去看。
- 略过不重要的代码，此时看到 `org.apache.ibatis.builder.MapperBuilderAssistant#addMappedStatement` 这个方法返回值就是 `MappedStatement`，不用多说，肯定是这个方法了，仔细一看，很清楚的看到了构建 `id` 的代码，如下图：

```
id = applyCurrentNamespace(id, isReference: false); id: "selectList"
boolean isSelect = sqlCommandType == SqlCommandType.SELECT = true; 解析出id

MappedStatement.Builder statementBuilder = new MappedStatement.Builder(configuration, id, s
    .resource(resource)
    .fetchSize(fetchSize)
    .timeout(timeout)
    .statementType(statementType)
    .keyGenerator(keyGenerator)
    .keyProperty(keyProperty)
    .keyColumn(keyColumn)
    .databaseId(databaseId)
    .lang(lang)
    .resultOrdered(resultOrdered = false)
    .resultSets(resultSets)
    .resultMaps(getStatementResultMaps(resultMap, resultType, id))
    .resultSetType(resultSetType)
    .flushCacheRequired(valueOrDefault(flushCache = false, !isSelect = false))
    .useCache(valueOrDefault(useCache = true, isSelect = true))
```

- 从上图可以知道，创建 `id` 的代码就是 `id = applyCurrentNamespace(id, false);`，具体实现如下图：

```
public String applyCurrentNamespace(String base, boolean isReference) { base: "selectList" i
    if (base == null || isReference) { base: "selectList"
        return null;
    }
    if (isReference == false) {
        // is it qualified with any namespace yet?
        if (base.contains(".")) {
            return base;
        }
    } else {
        // is it qualified with this namespace yet?
        if (base.startsWith(currentNamespace + ".") == false) {
            return base;
        }
    }
    if (base.contains(".") == false) {
        throw new BuilderException("Dots are not allowed in element names, please remove it from
    }
    return currentNamespace + "." + base;
}
```

上图的代码已经很清楚了，`MappedStatement`中的`id=Mapper`的全类名+'.'+方法名。如果重载话，肯定会存在`id`相同的`MappedStatement`。

- 到了这其实并不能说明方法不能重载啊，重复就重复呗，并没有冲突啊。这里需要看一个结构，如下：

```
protected final Map<String, MappedStatement> mappedStatements = new
StrictMap<MappedStatement>("Mapped statements collection")
    .conflictMessageProducer((savedValue, targetValue) ->
        ". please check " + savedValue.getResource() + " and " +
targetValue.getResource());
```

- 构建好的`MappedStatement`都会存入`mappedStatements`中，如下代码：

```
public void addMappedStatement(MappedStatement ms) {
    //key 是id
    mappedStatements.put(ms.getId(), ms);
}
```

- `StrictMap`的`put(k,v)`方法如下图：



```
@SuppressWarnings("unchecked")
public V put(String key, V value) {
    if (containsKey(key)) {
        throw new IllegalArgumentException(name + " already contains value for " + key
            + (conflictMessageProducer == null ? "" : conflictMessageProducer.apply(super.get(key), value)));
    }
    if (key.contains(".")) {
        final String shortKey = getShortName(key);
        if (super.get(shortKey) == null) {
            super.put(shortKey, value);
        } else {
            super.put(shortKey, (V) new Ambiguity(shortKey));
        }
    }
}
```

到了这里应该理解了吧，这下抛出的异常和上面的异常信息对应起来了。这个`StrictMap`不允许有重复的`key`，而存入的`key`就是`id`。因此`Mapper`中的方法不能重载。

如何找到XML中对应的SQL？

- 在使用Mybatis的时候只是简单的调用`Mapper`中的方法就可以执行SQL，如下代码：

```
List<UserInfo> userInfos =
mapper.selectList(Arrays.asList("192", "198"));
```

一行简单的调用到底如何找到对应的SQL呢？其实就是根据id从Map<String, MappedStatement> mappedStatements中查找对应的MappedStatement。

- 在org.apache.ibatis.session.defaults.DefaultSqlSession#selectList方法有这一行代码如下：

```
@Override
public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
    try {
        MappedStatement ms = configuration.getMappedStatement(statement);
        return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
    } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error querying database. Cause: " + e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}
```

- MappedStatement ms = configuration.getMappedStatement(statement);这行代码就是根据id从mappedStatements获取对应的MappedStatement，源码如下：

```
public MappedStatement getMappedStatement(String id) {
    return this.getMappedStatement(id, true);
}
```

总结

- 文章写到这，想必已经很清楚Mapper中的方法为什么不能重载了，归根到底就是因为这个这个id=Mapper的全类名+'.'+方法名。
- 如果觉得作者写的不错，有所收获的话，点点关注，分享一波，关注微信公众号码猿技术专栏 第一手文章推送！！



Mybatis中的TypeHandler你真的会用吗？

前言

- 相信大家用Mybatis这个框架至少一年以上了吧，有没有思考过这样一个问题：数据库有自己的数据类型，Java有自己的数据类型，那么Mybatis是如何把数据库中的类型和Java的数据类型对应的呢？
- 本篇文章就来讲讲Mybatis中的黑匣子TypeHandler(类型处理器)，说它是黑匣子一点都不为过，总是在默默的奉献着，但是不为人知。

环境配置

- 本篇文章讲的一切内容都是基于Mybatis3.5和SpringBoot-2.3.3.RELEASE。

什么是TypeHandler？

- 顾名思义，类型处理器，将入参和结果转换为所需要的类型，Mybatis中对于内置了许多类型处理器，实际开发中已经足够使用了，如下图：

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	java.lang.Boolean, boolean	数据库兼容的 BOOLEAN
ByteTypeHandler	java.lang.Byte, byte	数据库兼容的 NUMERIC 或 BYTE
ShortTypeHandler	java.lang.Short, short	数据库兼容的 NUMERIC 或 SMALLINT
IntegerTypeHandler	java.lang.Integer, int	数据库兼容的 NUMERIC 或 INTEGER
LongTypeHandler	java.lang.Long, long	数据库兼容的 NUMERIC 或 BIGINT
FloatTypeHandler	java.lang.Float, float	数据库兼容的 NUMERIC 或 FLOAT
DoubleTypeHandler	java.lang.Double, double	数据库兼容的 NUMERIC 或 DOUBLE
BigDecimalTypeHandler	java.math.BigDecimal	数据库兼容的 NUMERIC 或 DECIMAL
StringTypeHandler	java.lang.String	CHAR, VARCHAR
ClobReaderTypeHandler	java.io.Reader	-
ClobTypeHandler	java.lang.String	CLOB, LONGVARCHAR
NStringTypeHandler	java.lang.String	NVARCHAR, NCHAR
NClobTypeHandler	java.lang.String	NCLOB
BlobInputStreamTypeHandler	java.io.InputStream	-
ByteArrayTypeHandler	byte[]	数据库兼容的字节流类型
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	java.util.Date	TIMESTAMP
DateOnlyTypeHandler	java.util.Date	DATE
TimeOnlyTypeHandler	java.util.Date	TIME
SqlTimestampTypeHandler	java.sql.Timestamp	TIMESTAMP
SqlDateTypeHandler	java.sql.Date	DATE
SqlTimeTypeHandler	java.sql.Time	TIME
ObjectTypeHandler	Any	OTHER 或未指定类型
EnumTypeHandler	Enumeration Type	VARCHAR 或任何兼容的字符串类型，用来存储枚举的名称（而不是索引序数值）
EnumOrdinalTypeHandler	Enumeration Type	任何兼容的 NUMERIC 或 DOUBLE 类型，用来存储枚举的序数值（而不是名称）。
SqlxmlTypeHandler	java.lang.String	SQLXML
InstantTypeHandler	java.time.Instant	TIMESTAMP
LocalDateTimeTypeHandler	java.time.LocalDateTime	TIMESTAMP
LocalDateTypeHandler	java.time.LocalDate	DATE
LocalTimeTypeHandler	java.time.LocalTime	TIME
OffsetDateTimeTypeHandler	java.time.OffsetDateTime	TIMESTAMP
OffsetTimeTypeHandler	java.time.OffsetTime	TIME
ZonedDateTimeTypeHandler	java.time.ZonedDateTime	TIMESTAMP
YearTypeHandler	java.time.Year	INTEGER
MonthTypeHandler	java.time.Month	INTEGER
YearMonthTypeHandler	java.time.YearMonth	VARCHAR 或 LONGVARCHAR
JapaneseDateTypeHandler	java.time.chrono.JapaneseDate	DATE

- 类型处理器这个接口其实很简单，总共四个方法，一个方法将入参的Java类型的数据转换为JDBC类型，三个方法将返回结果转换为Java类型。源码如下：

```
public interface TypeHandler<T> {
    //设置参数，java类型转换为jdbc类型
    void setParameter(PreparedStatement ps, int i, T parameter,
        JdbcType jdbcType) throws SQLException;
    //将查询的结果转换为java类型
    T getResult(ResultSet rs, String columnName) throws SQLException;

    T getResult(ResultSet rs, int columnIndex) throws SQLException;

    T getResult(CallableStatement cs, int columnIndex) throws
        SQLException;
}
```

如何自定义并使用TypeHandler?

- 实际应用开发中的难免会有一些需求要自定义一个TypeHandler，比如这样一个需求：前端传来的年龄是男,女，但是数据库定义的字段却是int类型（1男2女）。此时可以自定义一个年龄的类型处理器，进行转换。

如何自定义?

- 自定义的方式有两种，一种是实现TypeHandler这个接口，另一个就是继承BaseTypeHandler这个便捷的抽象类。
- 下面直接继承BaseTypeHandler这个抽象类，定义一个年龄的类型处理器，如下：

```
@MappedJdbcTypes(JdbcType.INTEGER)
@MappedTypes(String.class)
public class GenderTypeHandler extends BaseTypeHandler {

    //设置参数，这里将Java的String类型转换为JDBC的Integer类型
    @Override
    public void setNonNullParameter(PreparedStatement ps, int i,
    Object parameter, JdbcType jdbcType) throws SQLException {
        ps.setInt(i, StringUtils.equals(parameter.toString(),"男")?
    1:2);
    }

    //以下三个参数都是将查询的结果转换
    @Override
    public Object getNullableResult(ResultSet rs, String
    columnName) throws SQLException {
        return rs.getInt(columnName)==1?"男":"女";
    }

    @Override
    public Object getNullableResult(ResultSet rs, int columnIndex)
    throws SQLException {
        return rs.getInt(columnIndex)==1?"男":"女";
    }

    @Override
    public Object getNullableResult(CallableStatement cs, int
    columnIndex) throws SQLException {
        return cs.getInt(columnIndex)==1?"男":"女";
    }
}
```

```
}  
}
```

- 这里涉及到两个注解，如下：
 - `@MappedTypes`：指定与其关联的 `Java` 类型列表。如果在 `javaType` 属性中也同时指定，则注解上的配置将被忽略。
 - `@MappedJdbcTypes`：指定与其关联的 `JDBC` 类型列表。如果在 `jdbcType` 属性中也同时指定，则注解上的配置将被忽略。

如何将其添加到Mybatis中？

- Mybatis在与SpringBoot整合之后一切都变得很简单了，其实这里有两种配置方式，下面将会一一介绍。
- 第一种：只需要在配置文件 `application.properties` 中添加一行配置即可，如下：

```
## 设置自定义的TypeHandler所在的包，启动的时候会自动扫描配置到Mybatis中  
mybatis.type-handlers-package=cn.cb.demo.typehandler
```

- 第二种：其实任何框架与Springboot整合之后，只要配置文件中能够配置的，在配置类中都可以配置（除非有特殊定制，否则不要轻易覆盖自动配置）。如下：

```
@Bean("sqlSessionFactory")  
public SqlSessionFactory sqlSessionFactory(DataSource  
dataSource) throws Exception {  
    SqlSessionFactoryBean sqlSessionFactoryBean = new  
    SqlSessionFactoryBean();  
    sqlSessionFactoryBean.setDataSource(dataSource);  
    sqlSessionFactoryBean.setMapperLocations(new  
    PathMatchingResourcePatternResolver().getResources(MAPPER_LOCATORIN)  
    );  
    org.apache.ibatis.session.Configuration configuration = new  
    org.apache.ibatis.session.Configuration();  
    // 自动将数据库中的下划线转换为驼峰格式  
    configuration.setMapUnderscoreToCamelCase(true);  
    configuration.setDefaultFetchSize(100);  
    configuration.setDefaultStatementTimeout(30);  
    sqlSessionFactoryBean.setConfiguration(configuration);  
    //将typeHandler注册到mybatis
```

```

        GenderTypeHandler genderTypeHandler = new
GenderTypeHandler();
        TypeHandler[] typeHandlers=new TypeHandler[]
{genderTypeHandler};
        sqlSessionFactoryBean.setTypeHandlers(typeHandlers);
        return sqlSessionFactoryBean.getObject();
    }

```

- 第二种方式的思想其实就是重写自动配置类 `MybatisAutoConfiguration` 中的方法。注意：除非自己有特殊定制，否则不要轻易重写自动配置类中的方法。

XML文件中如何指定TypeHandler?

- 上面的两个步骤分别是自定义和注入到Mybatis中，那么如何在XML文件中使用呢？
- 使用其实很简单，分为两种，一种是更新，一种查询，下面将会一一介绍。
- 更新：删除自不必说了，这里讲的是 `update` 和 `insert` 两种，只需要在 `#{}` 中指定的属性 `typeHandler` 为自定义的全类名即可，代码如下：

```

<insert id="insertUser">
    insert into
user_info(user_id,his_id,name,gender,password,create_time)
    values(#{userId,jdbcType=VARCHAR},#
{hisId,jdbcType=VARCHAR},#{name,jdbcType=VARCHAR},
    #
{gender,jdbcType=INTEGER,typeHandler=cn.cb.demo.typehandler.GenderT
ypeHandler},#{password,jdbcType=VARCHAR},now())
</insert>

```

- 查询：查询的时候类型处理会将JDBC类型的转化为Java类型，因此也是需要指定 `typeHandler`，需要在 `resultMap` 中指定 `typeHandler` 这个属性，值为全类名，如下：

```

<resultMap id="userResultMap" type="cn.cb.demo.domain.UserInfo">
    <id column="id" property="id"/>
    <result column="user_id" property="userId"/>
    <result column="his_id" property="hisId"/>
    <!-- 指定typeHandler属性为全类名-->
    <result column="gender" property="gender"
typeHandler="cn.cb.demo.typehandler.GenderTypeHandler"/>

```



```

        <result column="name" property="name"/>
        <result column="password" property="password"/>
    </resultMap>

    <select id="selectList" resultMap="userResultMap">
        select * from user_info where status=1
        and user_id in
        <foreach collection="userIds" item="item" open="("
separator="," close=")" >
            #{item}
        </foreach>
    </select>

```

源码中如何执行TypeHandler?

- 既然会使用TypeHandler了，那么肯定要知道其中的执行原理了，在Mybatis中类型处理器是如何在JDBC类型和Java类型进行转换的，下面的将从源码角度详细介绍。

入参如何转换?

- 这个肯定是发生在设置参数的过程中，详细的代码在PreparedStatementHandler中的parameterize()方法中，这个方法就是设置参数的方法。源码如下：

```

@Override
public void parameterize(Statement statement) throws SQLException
{
    //实际调用的是DefaultParameterHandler
    parameterHandler.setParameters((PreparedStatement) statement);
}

```

- 实际执行的是DefaultParameterHandler中的setParameters方法，如下：

```

public void setParameters(PreparedStatement ps) {
    ErrorContext.instance().activity("setting
parameters").object(mappedStatement.getParameterMap().getId());
    //获取参数映射
    List<ParameterMapping> parameterMappings =
    boundsSql.getParameterMappings();
    //遍历参数映射，一一设置
    if (parameterMappings != null) {

```

```

        for (int i = 0; i < parameterMappings.size(); i++) {
            ParameterMapping parameterMapping =
parameterMappings.get(i);
            if (parameterMapping.getMode() != ParameterMode.OUT) {
                Object value;
                String propertyName = parameterMapping.getProperty();
                if (boundSql.hasAdditionalParameter(propertyName)) { //
issue #448 ask first for additional params
                    value = boundSql.getAdditionalParameter(propertyName);
                } else if (parameterObject == null) {
                    value = null;
                } else if
(typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
                    value = parameterObject;
                } else {
                    MetaObject metaObject =
configuration.newMetaObject(parameterObject);
                    value = metaObject.getValue(propertyName);
                }
                //获取类型处理器，如果不存在，使用默认的
                TypeHandler typeHandler =
parameterMapping.getTypeHandler();
                //JdbcType
                JdbcType jdbcType = parameterMapping.getJdbcType();
                if (value == null && jdbcType == null) {
                    jdbcType = configuration.getJdbcTypeForNull();
                }
                try {
                    //调用类型处理器中的方法设置参数，将Java类型转换为JDBC类型
                    typeHandler.setParameter(ps, i + 1, value, jdbcType);
                } catch (TypeException e) {
                    throw new TypeException("Could not set parameters for
mapping: " + parameterMapping + ". Cause: " + e, e);
                } catch (SQLException e) {
                    throw new TypeException("Could not set parameters for
mapping: " + parameterMapping + ". Cause: " + e, e);
                }
            }
        }
    }
}

```

- 从上面的源码中可以知道这行代码 `typeHandler.setParameter(ps, i + 1, value, jdbcType)`; 就是调用类型处理器中的设置参数的方法，将 `Java` 类型转换为 `JDBC` 类型。

结果如何转换？

- 这一过程肯定是发生在执行查询语句的过程中，之前也是介绍过 `Mybatis` 的六大剑客，其中的 `ResultSetHandler` 这个组件就是对查询的结果进行处理的，那么肯定是发生在这一组件中的某个方法。
- 在 `PreparedStatementHandler` 执行查询结束之后，调用的是 `ResultSetHandler` 中的 `handleResultSets()` 方法，对结果进行处理，如下：

```
public <E> List<E> query(Statement statement, ResultHandler
resultHandler) throws SQLException {
    PreparedStatement ps = (PreparedStatement) statement;
    //执行SQL
    ps.execute();
    //处理结果
    return resultSetHandler.handleResultSets(ps);
}
```

- 最终的在 `DefaultResultSetHandler` 中的 `getPropertyMappingValue()` 方法中调用了 `TypeHandler` 中的 `getResult()` 方法，如下：

```
private Object getPropertyMappingValue(ResultSet rs, MetaObject
metaResultObject, ResultMapping propertyMapping, ResultLoaderMap
lazyLoader, String columnPrefix)
    throws SQLException {
    if (propertyMapping.getNestedQueryId() != null) {
        return getNestedQueryMappingValue(rs, metaResultObject,
propertyMapping, lazyLoader, columnPrefix);
    } else if (propertyMapping.getResultSet() != null) {
        addPendingChildRelation(rs, metaResultObject,
propertyMapping); // TODO is that OK?
        return DEFERRED;
    } else {
        final TypeHandler<?> typeHandler =
propertyMapping.getTypeHandler();
        final String column =
prependPrefix(propertyMapping.getColumn(), columnPrefix);
        //执行typeHandler中的方法获取结果并且转换为对应的Java类型
    }
```

```
        return typeHandler.getResult(rs, column);  
    }  
}
```

总结

- 上述只是简要的介绍了类型处理器如何在Mybatis中执行的，可能其中有些概念东西如果不清楚的，可以看一下作者前面的文章，如下：
 - [Mybatis源码解析之六剑客](#)
 - [Mybatis源码如何阅读，教你一招!!!](#)
 - [Mybatis如何执行Select语句，你知道吗？](#)

总结

- 本文详细的介绍了TypeHandler在Mybatis中的应用、自定义使用以及从源码角度分析了类型处理器的执行流程，如果觉得作者写的不错，有所收获的话，不妨点点关注，分享一波。



Mybatis的插件原理以及如何实现？

前言

- Mybatis的分页插件相信大家都使用过，那么可知道其中的实现原理？分页插件就是利用的Mybatis中的插件机制实现的，在`Executor`的`query`执行前后进行分页处理。
- 此篇文章就来介绍以下Mybatis的插件机制以及在底层是如何实现的。

环境配置

- 本篇文章讲的一切内容都是基于`mybatis3.5`和`SpringBoot-2.3.3.RELEASE`。

什么是插件？

- 插件是Mybatis中的最重要的功能之一，能够对特定组件的特定方法进行增强。
- MyBatis 允许你在映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：
 - **Executor:** `update`, `query`, `flushStatements`, `commit`, `rollback`, `getTransaction`, `close`, `isClosed`
 - **ParameterHandler:** `getParameterObject`, `setParameters`
 - **ResultSetHandler:** `handleResultSets`, `handleOutputParameters`
 - **StatementHandler:** `prepare`, `parameterize`, `batch`, `update`, `query`

如何自定义插件？

- 插件的实现其实很简单，只需要实现Mybatis提供的`Interceptor`这个接口即可，源码如下：

```
public interface Interceptor {  
    //拦截的方法  
    Object intercept(Invocation invocation) throws Throwable;  
    //返回拦截器的代理对象  
    Object plugin(Object target);  
    //设置一些属性  
    void setProperties(Properties properties);  
}
```

举个栗子

- 有这样一个需求：需要在Mybatis执行的时候篡改 `selectById` 的参数值。
- 分析：修改SQL的入参，应该在哪个组件的哪个方法上拦截篡改呢？研究过源码的估计都很清楚的知道， `ParameterHandler` 中的 `setParameters()` 方法就是对参数进行处理的。因此肯定是拦截这个方法是最合适。
- 自定义的插件如下：

```
/**
 * @Intercepts 注解标记这是一个拦截器,其中可以指定多个@Signature
 * @Signature 指定该拦截器拦截的是四大对象中的哪个方法
 *      type: 拦截器的四大对象的类型
 *      method: 拦截器的方法, 方法名
 *      args: 入参的类型, 可以是多个, 根据方法的参数指定, 以此来区分方法的重载
 */
@Intercepts(
    {
        @Signature(type = ParameterHandler.class, method = "setParameters", args = {PreparedStatement.class})
    }
)
public class ParameterInterceptor implements Interceptor {
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("拦截器执行: "+invocation.getTarget());
        //目标对象
        Object target = invocation.getTarget();
        //获取目标对象中所有属性的值, 因为ParameterHandler使用的是DefaultParameterHandler, 因此里面的所有的属性都封装在其中
        MetaObject metaObject = SystemMetaObject.forObject(target);
        //使用xxx.xxx.xx的方式可以层层获取属性值, 这里获取的是mappedStatement中的id值
        String value = (String) metaObject.getValue("mappedStatement.id");
        //如果是指定的查询方法
        if ("cn.cb.demo.dao.UserMapper.selectById".equals(value)){
            //设置参数的值是admin_1, 即是设置id=admin_1, 因为这里只有一个参数, 可以这么设置, 如果有多个需要需要循环
            metaObject.setValue("parameterObject", "admin_1");
        }
        //执行目标方法
```

```

        return invocation.proceed();
    }

    @Override
    public Object plugin(Object target) {
        //如果没有特殊定制，直接使用Plugin这个工具类返回一个代理对象即可
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {
    }
}

```

- `intercept` 方法：最终会拦截的方法，最重要的一个方法。
- `plugin` 方法：返回一个代理对象，如果没有特殊要求，直接使用Mybatis的工具类`Plugin`返回即可。
- `setProperties`：设置一些属性，不重要。

用到哪些注解？

- 自定义插件需要用到两个注解，分别是`@Intercepts`和`@Signature`。
- `@Intercepts`：标注在实现类上，表示这个类是一个插件的实现类。
- `@Signature`：作为`@Intercepts`的属性，表示需要增强Mybatis的某些组件中的某些方法（可以指定多个）。常用的属性如下：
 - `Class<?> type()`：指定哪个组件（`Executor`、`ParameterHandler`、`ResultSetHandler`、`StatementHandler`）
 - `String method()`：指定增强组件中的哪个方法，直接写方法名称。
 - `Class<?>[] args()`：方法中的参数，必须一一对应，可以写多个；这个属性非常重用，区分重载方法。

如何注入Mybatis？

- 上面已经将插件定义好了，那么如何注入到Mybatis中使其生效呢？
- 前提：由于本篇文章的环境是`SpringBoot+Mybatis`，因此讲一讲如何在SpringBoot中将插件注入到Mybatis中。
- 在Mybatis的自动配置类`MybatisAutoConfiguration`中，注入`SqlSessionFactory`的时候，有如下一段代码：

```

@Bean
@ConditionalOnMissingBean
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
    SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
    factory.setDataSource(dataSource);
    factory.setVfs(SpringBootVFS.class);
    if (StringUtils.hasText(this.properties.getConfigLocation())) {
        factory.setConfigLocation(this.resourceLoader.getResource(this.properties.getConfigLocation()));
    }
    applyConfiguration(factory);
    if (this.properties.getConfigurationProperties() != null) {
        factory.setConfigurationProperties(this.properties.getConfigurationProperties());
    }
    if (!ObjectUtils.isEmpty(this.interceptors)) {
        factory.setPlugins(this.interceptors);
    }
    if (this.databaseIdProvider != null) {
        factory.setDatabaseIdProvider(this.databaseIdProvider);
    }
}

```

将容器中的插件设置到factory中

- 上图中的 `this.interceptors` 是什么，从何而来，其实就是从容器中的获取的 `Interceptor[]`，如下一段代码：

```

public MybatisAutoConfiguration(MybatisProperties properties,
    ObjectProvider<Interceptor[]> interceptorsProvider,
    ResourceLoader resourceLoader,
    ObjectProvider<DatabaseIdProvider> databaseIdProvider,
    ObjectProvider<List<ConfigurationCustomizer>> configurationCustomizersProvider) {
    this.properties = properties;
    this.interceptors = interceptorsProvider.getIfAvailable();
    this.resourceLoader = resourceLoader;
    this.databaseIdProvider = databaseIdProvider.getIfAvailable();
    this.configurationCustomizers = configurationCustomizersProvider.getIfAvailable();
}

```

Spring中古老的一种方式：在构造方法中使用，获取ioc容器中的注入的Bean

- 从上图我们知道，这插件最终还是从IOC容器中获取的 `Interceptor[]` 这个 `Bean`，因此我们只需要在配置类中注入这个 `Bean` 即可，如下代码：

```

/**
 * @Configuration: 这个注解标注该类是一个配置类
 */
@Configuration
public class MybatisConfig{

    /**
     * @Bean : 该注解用于向容器中注入一个Bean
     * 注入Interceptor[]这个Bean
     * @return
     */
    @Bean
    public Interceptor[] interceptors(){
        //创建ParameterInterceptor这个插件
    }
}

```



```

        ParameterInterceptor parameterInterceptor = new
ParameterInterceptor();
        //放入数组返回
        return new Interceptor[]{parameterInterceptor};
    }
}

```

测试

- 此时自定义的插件已经注入了Mybatis中了，现在测试看看能不能成功执行呢？测试代码如下：

```

@Test
void contextLoads() {
    //传入的是1222
    UserInfo userInfo = userMapper.selectByUserId("1222");
    System.out.println(userInfo);
}

```

- 测试代码传入的是1222，由于插件改变了入参，因此查询出来的应该是admin_1这个人。

插件原理分析

- 插件的原理其实很简单，就是在创建组件的时候生成代理对象(Plugin)，执行组件方法的时候拦截即可。下面就来详细介绍一下插件在Mybatis底层是如何工作的？
- Mybatis的四大组件都是在Mybatis的配置类Configuration中创建的，具体的方法如下：

```

//创建Executor
public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {

```

```

        executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
        executor = new ReuseExecutor(this, transaction);
    } else {
        executor = new SimpleExecutor(this, transaction);
    }
    if (cacheEnabled) {
        executor = new CachingExecutor(executor);
    }
    //调用pluginAll方法, 生成代理对象
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}

//创建ParameterHandler
public ParameterHandler newParameterHandler(MappedStatement
mappedStatement, Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
    //调用pluginAll方法, 生成代理对象
    parameterHandler = (ParameterHandler)
interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
}

//创建ResultSetHandler
public ResultSetHandler newResultSetHandler(Executor executor,
MappedStatement mappedStatement, RowBounds rowBounds,
ParameterHandler parameterHandler,
    ResultHandler resultHandler, BoundSql boundSql) {
    ResultSetHandler resultSetHandler = new
DefaultResultSetHandler(executor, mappedStatement,
parameterHandler, resultHandler, boundSql, rowBounds);
    //调用pluginAll方法, 生成代理对象
    resultSetHandler = (ResultSetHandler)
interceptorChain.pluginAll(resultSetHandler);
    return resultSetHandler;
}

//创建StatementHandler

```

```

    public StatementHandler newStatementHandler(Executor executor,
        MappedStatement mappedStatement, Object parameterObject, RowBounds
        rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
        StatementHandler statementHandler = new
        RoutingStatementHandler(executor, mappedStatement, parameterObject,
        rowBounds, resultHandler, boundSql);
        //调用pluginAll方法，生成代理对象
        statementHandler = (StatementHandler)
        interceptorChain.pluginAll(statementHandler);
        return statementHandler;
    }

```

- 从上面的源码可以知道，创建四大组件的方法中都会执行 `pluginAll()` 这个方法来生成一个代理对象。具体如何生成的，下面详解。

如何生成代理对象？

- 创建四大组件过程中都执行了 `pluginAll()` 这个方法，此方法源码如下：

```

public Object pluginAll(Object target) {
    //循环遍历插件
    for (Interceptor interceptor : interceptors) {
        //调用插件的plugin()方法
        target = interceptor.plugin(target);
    }
    //返回
    return target;
}

```

- `pluginAll()` 方法很简单，直接循环调用插件的 `plugin()` 方法，但是我们调用的是 `Plugin.wrap(target, this)` 这行代码，因此要看一下 `wrap()` 这个方法的源码，如下：

```

public static Object wrap(Object target, Interceptor interceptor) {
    //获取注解的@signature的定义
    Map<Class<?>, Set<Method>> signatureMap =
    getSignatureMap(interceptor);
    //目标类
    Class<?> type = target.getClass();
    //获取需要拦截的接口

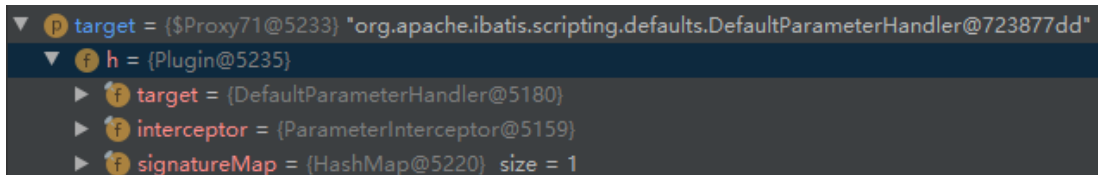
```

```

class<?>[] interfaces = getAllInterfaces(type, signatureMap);
if (interfaces.length > 0) {
    //生成代理对象
    return Proxy.newProxyInstance(
        type.getClassLoader(),
        interfaces,
        new Plugin(target, interceptor, signatureMap));
}
return target;
}

```

- `Plugin.wrap()` 这个方法的逻辑很简单，判断这个插件是否是拦截对应的组件，如果拦截了，生成代理对象（`Plugin`）返回，没有拦截直接返回，上面例子中生成的代理对象如下图：



```

▼ p target = {$Proxy71@5233} "org.apache.ibatis.scripting.defaults.DefaultParameterHandler@723877dd"
  ▼ f h = {Plugin@5235}
    ► f target = {DefaultParameterHandler@5180}
    ► f interceptor = {ParameterInterceptor@5159}
    ► f signatureMap = {HashMap@5220} size = 1

```

如何执行？

- 上面讲了Mybatis启动的时候如何根据插件生成代理对象的(`Plugin`)。现在就来看看这个代理对象是如何执行的？
- 既然是动态代理，肯定会执行的`invoke()`这个方法，`Plugin`类中的`invoke()`源码如下：

```

@Override
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    try {
        //获取@signature标注的方法
        Set<Method> methods =
signatureMap.get(method.getDeclaringClass());
        //如果这个方法被拦截了
        if (methods != null && methods.contains(method)) {
            //直接执行插件的intercept()这个方法
            return interceptor.intercept(new Invocation(target, method,
args));
        }
    }
}

```

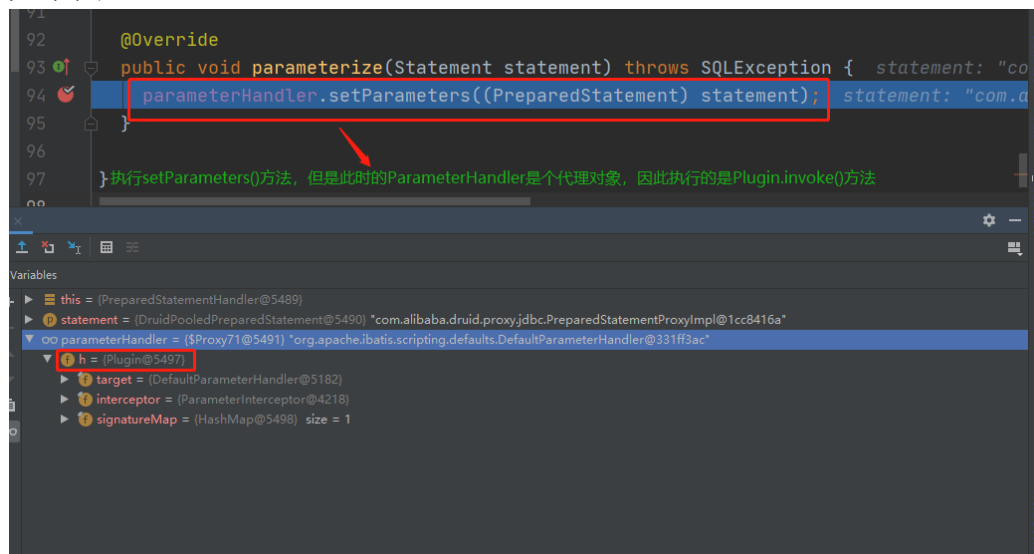
```

        //没有被拦截，执行原方法
        return method.invoke(target, args);
    } catch (Exception e) {
        throw ExceptionUtil.unwrapThrowable(e);
    }
}

```

- 逻辑很简单，这个方法被拦截了就执行插件的 `intercept()` 方法，没有被拦截，则执行原方法。
- 还是以上面自定义的插件来看看执行的流程：

- `setParameters()` 这个方法在 `PreparedStatementHandler` 中被调用，如下图：



- 执行 `invoke()` 方法，发现 `setParameters()` 这个方法被拦截了，因此直接执行的是 `intercept()` 方法。

总结

- Mybatis中插件的原理其实很简单，分为以下几步：
 - a. 在项目启动的时候判断组件是否有被拦截，如果没有直接返回原对象。
 - b. 如果有被拦截，返回动态代理的对象（`Plugin`）。
 - c. 执行到的组件的中的方法时，如果不是代理对象，直接执行原方法
 - d. 如果是代理对象，执行 `Plugin` 的 `invoke()` 方法。

分页插件的原理分析

- 此处安利一款经常用的分页插件 `pagehelper`，Maven 依赖如下：

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.1.6</version>
</dependency>
```

- 分页插件很显然也是根据 Mybatis 的插件来定制的，来看看插件 `PageInterceptor` 的源码如下：

```
@Intercepts(
    {
        @Signature(type = Executor.class, method = "query",
            args = {MappedStatement.class, Object.class, RowBounds.class,
                ResultHandler.class}),
        @Signature(type = Executor.class, method = "query",
            args = {MappedStatement.class, Object.class, RowBounds.class,
                ResultHandler.class, CacheKey.class, BoundSql.class}),
    }
)
public class PageInterceptor implements Interceptor {}
```

- 既然是分页功能，肯定是在 `query()` 的时候拦截，因此肯定是在 `Executor` 这个组件中。
- 分页插件的原理其实很简单，不再一一分析源码了，根据的自己定义的分页数据重新赋值 `RowBounds` 来达到分页的目的，当然其中涉及到数据库方言等等内容，不是本章重点，有兴趣可以看一下 [GitHub 上的文档](#)。

总结

- 对于业务开发的程序员来说，插件的这个功能很少用到，但是不用就不应该了解吗？做人要有追求，哈哈。
- 欢迎关注作者的微信公众号 `码猿技术专栏`，作者为你们精心准备了 `springCloud` 最新精彩视频教程、精选 500 本电子书、架构师免费视频教程等等免费资源，让我们一起进阶，一起成长。

Mybatis源码阅读之六剑客

前言

- Mybatis的专题文章写到这里已经是第四篇了，前三篇讲了Mybatis的基本使用，相信只要认真看了的朋友，在实际开发中正常使用应该不是问题。没有看过的朋友，作者建议去看一看，三篇文章分别是[Mybatis入门之基本操作](#)、[Mybatis结果映射，你射准了吗？](#)、[Mybatis动态SQL，你真的会了吗？](#)。
- 当然，任何一个技术都不能浅尝辄止，今天作者就带大家深入底层源码看一看Mybatis的基础架构。此篇文章只是源码的入门篇，讲一些Mybatis中重要的组件，作者称之为六剑客。

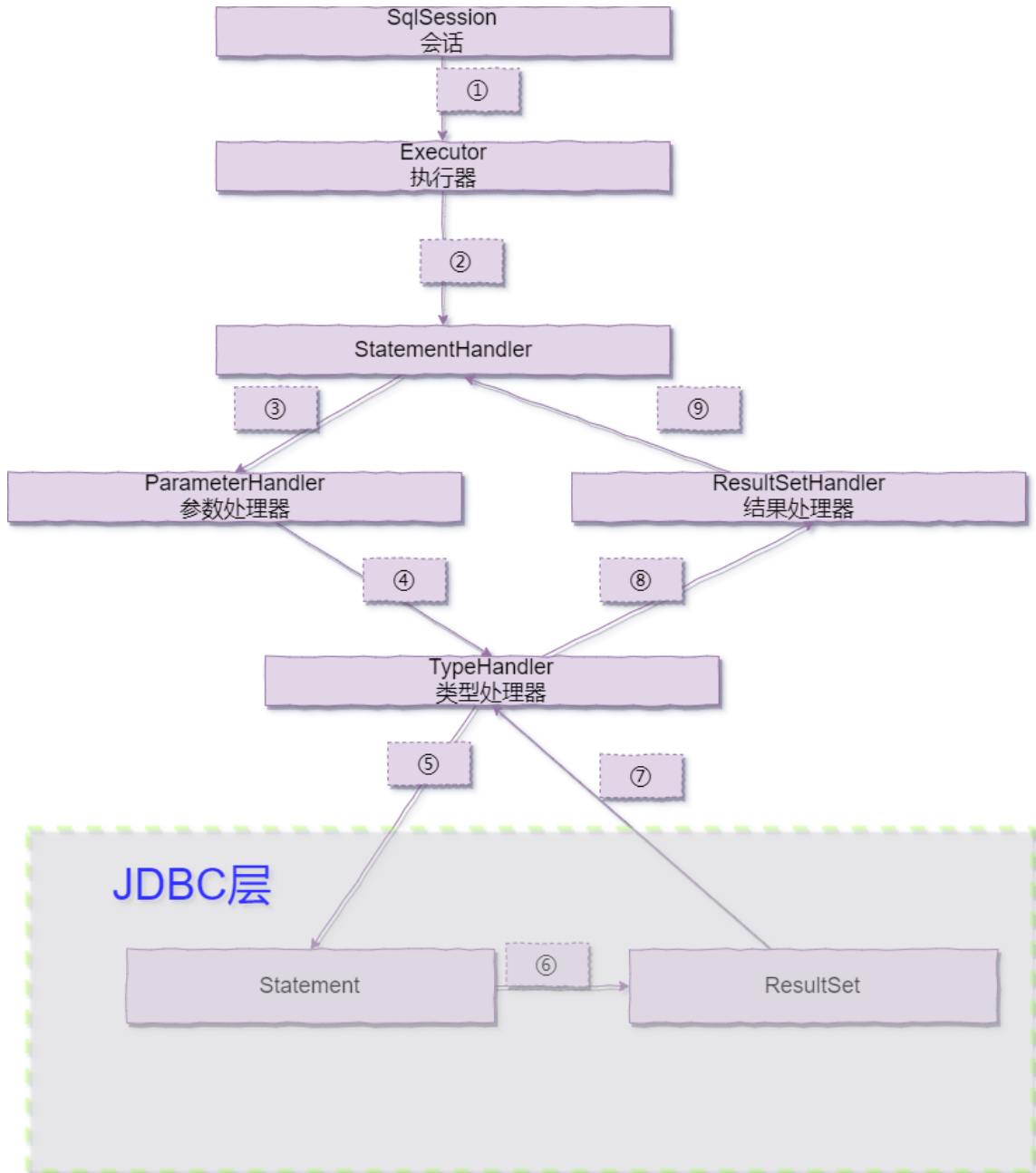
环境版本

- 本篇文章讲的一切内容都是基于Mybatis3.5和SpringBoot-2.3.3.RELEASE。

Mybatis的六剑客

- 其实Mybatis的底层源码和Spring比起来还是非常容易读懂的，作者将其中六个重要的接口抽离出来称之为Mybatis的六剑客，分别是SqlSession、Executor、StatementHandler、ParameterHandler、ResultSetHandler、TypeHandler。
- 六剑客在Mybatis中分别承担着什么角色？下面将会逐一介绍。

- 介绍六剑客之前，先来一张六剑客执行的流程图，如下：



SqlSession

- SqlSession**是Mybatis中的核心API，主要用来执行命令，获取映射，管理事务。它包含了所有执行语句、提交或回滚事务以及获取映射器实例的方法。

有何方法

- 其中定义了将近20个方法，其中涉及的到语句执行，事务提交回滚等方法。下面对于这些方法进行分类总结。

语句执行方法

- 这些方法被用来执行定义在 SQL 映射 XML 文件中的 SELECT、INSERT、UPDATE 和 DELETE 语句。你可以通过名字快速了解它们的作用，每一方法都接受语句的 ID 以及参数对象，参数可以是原始类型（支持自动装箱或包装类）、JavaBean、POJO 或 Map。

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
<T> Cursor<T> selectCursor(String statement, Object parameter)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String
mapKey)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

- 其中的最容易误解的就是 `selectOne` 和 `selectList`，从方法名称就很容易知道区别，一个是查询单个，一个是查询多个。如果你对自己的 SQL 无法确定返回一个还是多个结果的时候，建议使用 `selectList`。
- `insert`，`update`，`delete` 方法返回值是受影响的行数。
- `select` 还有几个重用的方法，用于限制返回行数，在 MySQL 中对应的就是 `limit`，如下：

```
<E> List<E> selectList (String statement, Object parameter,
RowBounds rowBounds)
<T> Cursor<T> selectCursor(String statement, Object parameter,
RowBounds rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String
mapKey, RowBounds rowbounds)
void select (String statement, Object parameter, ResultHandler<T>
handler)
void select (String statement, Object parameter, RowBounds
rowBounds, ResultHandler<T> handler)
```

- 其中的 `RowBounds` 参数中保存了限制的行数，起始行数。

立即批量更新方法

- 当你将 `ExecutorType` 设置为 `ExecutorType.BATCH` 时，可以使用这个方法清除（执行）缓存在 `JDBC` 驱动类中的批量更新语句。

```
List<BatchResult> flushStatements()
```

事务控制方法

- 有四个方法用来控制事务作用域。当然，如果你已经设置了自动提交或你使用了外部事务管理器，这些方法就没什么作用了。然而，如果你正在使用由 `Connection` 实例控制的 `JDBC` 事务管理器，那么这四个方法就会派上用场：

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

- 默认情况下 `MyBatis` 不会自动提交事务，除非它检测到调用了插入、更新或删除方法改变了数据库。如果你没有使用这些方法提交修改，那么你可以在 `commit` 和 `rollback` 方法参数中传入 `true` 值，来保证事务被正常提交（注意，在自动提交模式或者使用了外部事务管理器的情况下，设置 `force` 值对 `session` 无效）。大部分情况下你无需调用 `rollback()`，因为 `MyBatis` 会在你没有调用 `commit` 时替你完成回滚操作。不过，当你要在一个可能多次提交或回滚的 `session` 中详细控制事务，回滚操作就派上用场了。

本地缓存方法

- `Mybatis` 使用到了两种缓存：本地缓存（`local cache`）和二级缓存（`second level cache`）。
- 默认情况下，本地缓存数据的生命周期等同于整个 `session` 的周期。由于缓存会被用来解决循环引用问题和加快重复嵌套查询的速度，所以无法将其完全禁用。但是你可以通过设置 `localCacheScope=STATEMENT` 来只在语句执行时使用缓存。
- 可以调用以下方法清除本地缓存。

```
void clearCache()
```

获取映射器

- 在SqlSession中你也可以获取自己的映射器，直接使用下面的方法，如下：

```
<T> T getMapper(Class<T> type)
```

- 比如你需要获取一个UserMapper，如下：

```
UserMapper mapper = sqlSessionTemplate.getMapper(UserMapper.class);
```

有何实现类

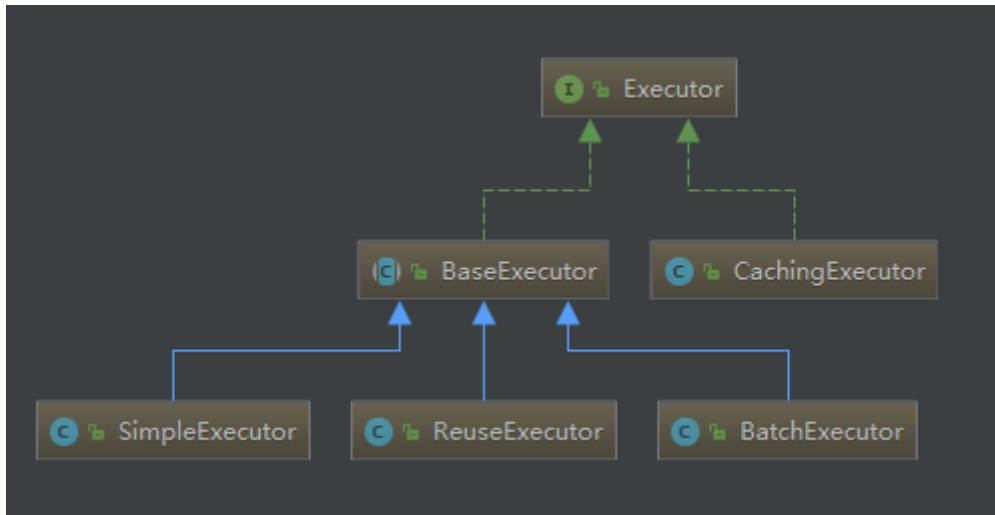
- 在Mybatis中有三个实现类，分别是DefaultSqlSession，SqlSessionManager、SqlSessionTemplate，其中重要的就是DefaultSqlSession，这个后面讲到Mybatis执行源码的时候会一一分析。
- 在与SpringBoot整合时，Mybatis的启动器配置类会默认注入一个SqlSessionTemplate，源码如下：

```
@Bean
@ConditionalOnMissingBean
public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory
sqlSessionFactory)
    //根据执行器的类型创建不同的执行器，默认CachingExecutor
    ExecutorType executorType = this.properties.getExecutorType();
    if (executorType != null) {
        return new SqlSessionTemplate(sqlSessionFactory,
executorType);
    } else {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

Executor

- Mybatis的执行器，是Mybatis的调度核心，负责SQL语句的生成和缓存的维护，SqlSession中的crud方法实际上都是调用执行器中的对应方法执行。

- 继承结构如下图：



实现类

- 下面我们来看看都有哪些实现类，分别有什么作用。

BaseExecutor

- 这是一个抽象类，采用模板方法的模式，有意思的是这个老弟模仿Spring的方式，真正的执行的方法都是 `doxxx()`。
- 其中有一个方法值得注意，查询的时候走的 **一级缓存**，因此这里注意下，既然这是个模板类，那么Mybatis执行select的时候默认都会走一级缓存。代码如下：

```
private <E> List<E> queryFromDatabase(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler resultHandler,
CacheKey key, BoundSql boundSql) throws SQLException {
    List<E> list;
    //此处的localCache即是一级缓存，是一个Map的结构
    localCache.putObject(key, EXECUTION_PLACEHOLDER);
    try {
        //执行真正的查询
        list = doQuery(ms, parameter, rowBounds, resultHandler,
boundSql);
    } finally {
        localCache.removeObject(key);
    }
    localCache.putObject(key, list);
    if (ms.getStatementType() == StatementType.CALLABLE) {
        localOutputParameterCache.putObject(key, parameter);
    }
}
```

```
    return list;
}
```

CachingExecutor

- 这个比较有名了，二级缓存的维护类，与SpringBoot整合默认创建的就是这个家伙。下面来看一下如何走的二级缓存，源码如下：

```
@Override
    public <E> List<E> query(MappedStatement ms, Object
parameterObject, RowBounds rowBounds, ResultHandler resultHandler,
CacheKey key, BoundSql boundSql)
        throws SQLException {
        //查看当前sql是否使用了二级缓存
        Cache cache = ms.getCache();
        //使用缓存了，直接从缓存中取
        if (cache != null) {
            flushCacheIfRequired(ms);
            if (ms.isUseCache() && resultHandler == null) {
                ensureNoOutParams(ms, boundSql);
                @SuppressWarnings("unchecked")
                //从缓存中取数据
                List<E> list = (List<E>) tcm.getObject(cache, key);
                if (list == null) {
                    //没取到数据，则执行SQL从数据库查询
                    list = delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
                    //查到了，放入缓存中
                    tcm.putObject(cache, key, list); // issue #578 and #116
                }
                //直接返回
                return list;
            }
        }
        //没使用二级缓存，直接执行SQL从数据库查询
        return delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
    }
```

- 这玩意就是走个二级缓存，其他没什么。

SimpleExecutor

- 这个类像个直男，最简单的一个执行器，就是根据对应的SQL执行，不会做一些额外的操作。

BatchExecutor

- 通过批量操作来优化性能。通常需要注意的是批量更新操作，由于内部有缓存的实现，使用完成后记得调用 `flushStatements` 来清除缓存。

ReuseExecutor

- 可重用的执行器，重用的对象是Statement，也就是说该执行器会缓存同一个sql的Statement，省去Statement的重新创建，优化性能。
- 内部的实现是通过一个HashMap来维护Statement对象的。由于当前Map只在该session中有效，所以使用完成后记得调用 `flushStatements` 来清除Map。

SpringBoot中如何创建

- 在SpringBoot到底创建的是哪个执行器呢？其实只要阅读一下源码可以很清楚的知道，答案就在 `org.apache.ibatis.session.Configuration` 类中，其中创建执行器的源码如下：

```
public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    //没有指定执行器的类型，创建默认的，即是SimpleExecutor
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    //类型是BATCH，创建BatchExecutor
    if (ExecutorType.BATCH == executorType) {
        executor = new BatchExecutor(this, transaction);
        //类型为REUSE，创建ReuseExecutor
    } else if (ExecutorType.REUSE == executorType) {
        executor = new ReuseExecutor(this, transaction);
    } else {
        //除了上面两种，创建的都是SimpleExecutor
        executor = new SimpleExecutor(this, transaction);
    }
}
```

```

    }
    //如果全局配置了二级缓存，则创建CachingExecutor，SpringBoot中这个参数默
    认是true，可以自己设置为false
    if (cacheEnabled) {
        //创建CachingExecutor
        executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}

```

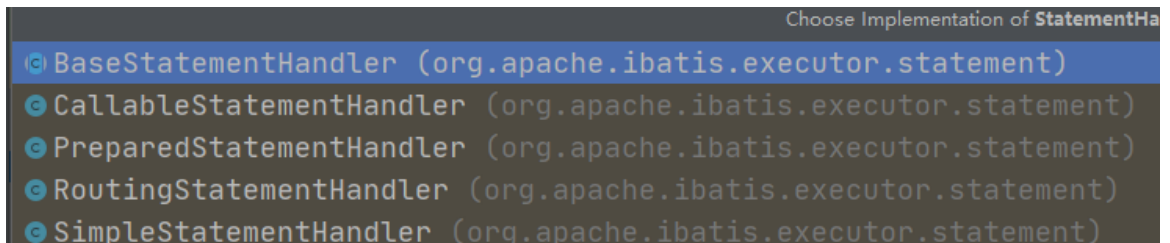
- 显而易见，SpringBoot中默认创建的是CachingExecutor，因为默认的cacheEnabled的值为true。

StatementHandler

- 熟悉JDBC的朋友应该都能猜到这个接口是干嘛的，很显然，这个是对SQL语句进行处理和参数赋值的。

实现类

- 该接口也是有很多的实现类，如下图：



SimpleStatementHandler

- 这个很简单了，就是对应我们JDBC中常用的Statement接口，用于简单SQL的处理

PreparedStatementHandler

- 这个对应JDBC中的PreparedStatement，预编译SQL的接口。

CallableStatementHandler

- 这个对应JDBC中CallableStatement，用于执行存储过程相关的接口。

RoutingStatementHandler

- 这个接口是以上三个接口的路由，没有实际操作，只是负责上面三个StatementHandler的创建及调用。

ParameterHandler

- `ParameterHandler`在Mybatis中负责将sql中的占位符替换为真正的参数，它是一个接口，有且只有一个实现类`DefaultParameterHandler`。
- `setParameters`是处理参数最核心的方法。这里不再详细的讲，后面会讲到。

TypeHandler

- 这位大神应该都听说过，也都自定义过吧，简单的说就是在预编译设置参数和取出结果的时候将Java类型和JDBC的类型进行相应的转换。当然，Mybatis内置了很多默认的类型处理器，基本够用，除非有特殊的定制，我们才会去自定义，比如需要将Java对象以JSON字符串的形式存入数据库，此时就可以自定义一个类型处理器。
- 很简单的东西，此处就不再详细的讲了，后面会单独出一篇如何自定义类型处理器的文章。

ResultSetHandler

- 结果处理器，负责将JDBC返回的ResultSet结果集对象转换成List类型的集合或者`Cursor`。
- 具体实现类就是`DefaultResultSetHandler`，其实现的步骤就是将Statement执行后的结果集，按照Mapper文件中配置的ResultType或ResultMap来封装成对应的对象，最后将封装的对象返回。
- 源码及其复杂，尤其是其中对嵌套查询的解析，这里只做个了解，后续会专门写一篇文章介绍。

总结

- 至此，Mybatis源码第一篇就已经讲完了，本篇文章对Mybatis中的重要组件做了初步的了解，为后面更深入的源码阅读做了铺垫，如果觉得作者写的不错，在看分享一波，谢谢支持。



Mybatis源码如何阅读，教你一招

前言

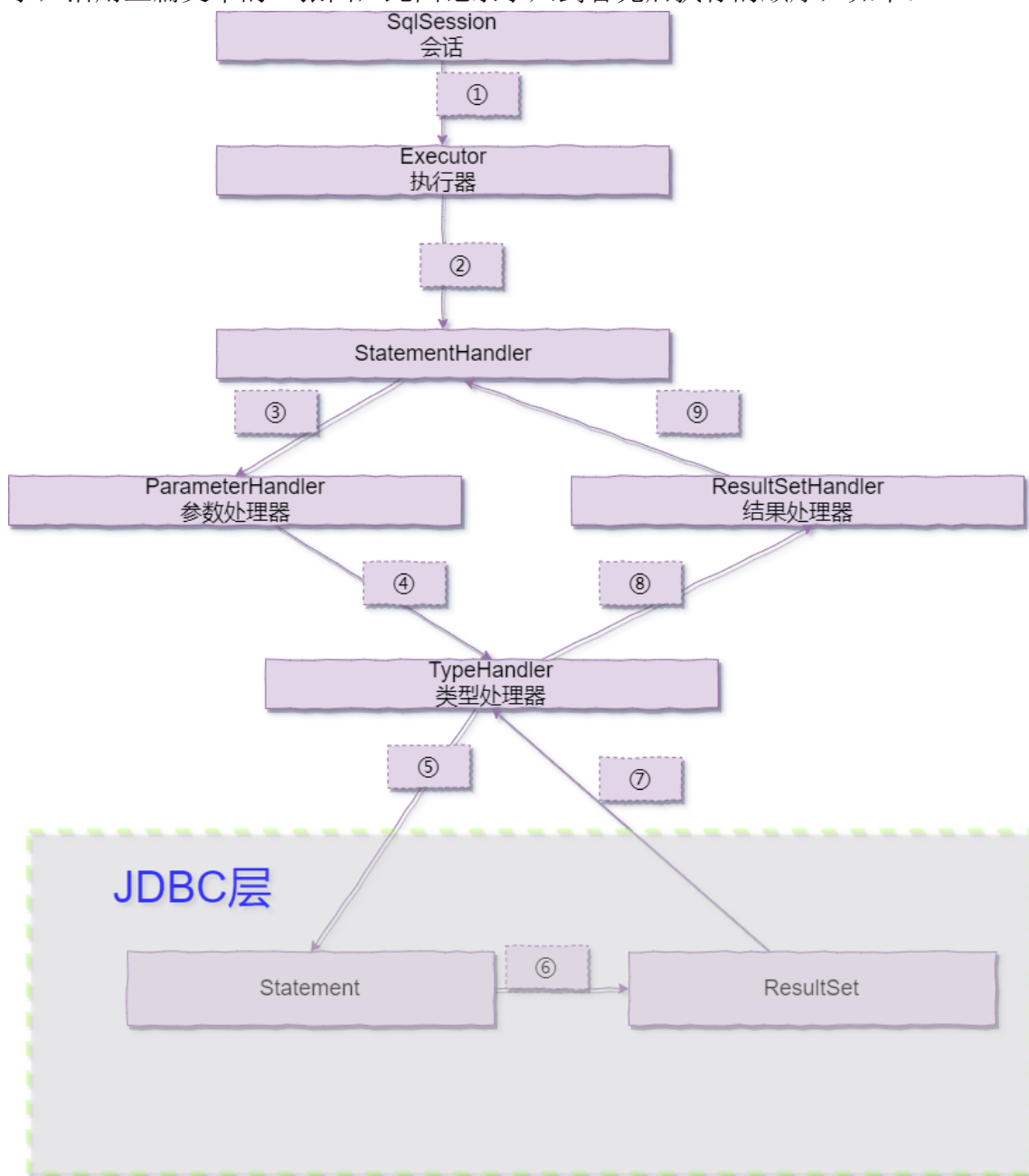
- 前一篇文章简单的介绍了Mybatis的六个重要组件，这六剑客占据了Mybatis的半壁江山，和六剑客搞了基友，那么Mybatis就是囊中之物了。对六剑客感兴趣的朋友，可以看看这篇文章：[Mybatis源码解析篇之六剑客](#)
- 有些初入门的朋友可能很害怕阅读源码，不知道如何阅读源码，与其我一篇文章按照自己的思路写完Mybatis的源码，但是你们又能理解多少呢？不如教会你们思路，让你们能够自己知道如何阅读源码。

环境配置

- 本篇文章讲的一切内容都是基于Mybatis3.5和SpringBoot-2.3.3.RELEASE。

从哪入手？

- 还是要说一说六剑客的故事，既然是Mybatis的重要组件，当然要从六剑客下手了，沿用上篇文章的一张图，此图记录了六剑客先后执行的顺序，如下：



- 阅读源码最重要的一点不能忘了，就是开启 **DEBUG** 模式，重要方法打上断点，重要语句打上断点，先把握整体，再研究细节，基本就不难了。
- 下面就以Myabtis的查询语句 `selectList()` 来具体分析下如何阅读。

总体把握六剑客

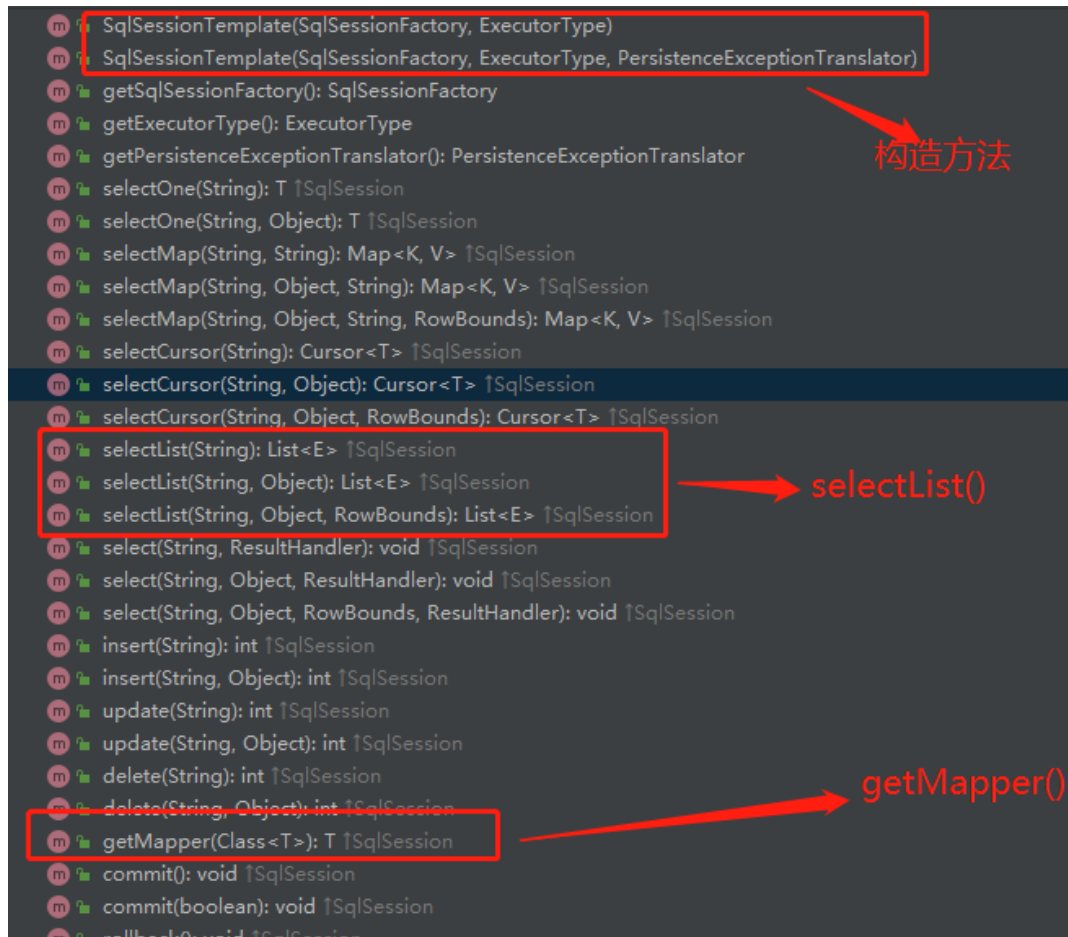
- 从六剑客开整，既然是重要组件，源码执行流程肯定都是围绕着六剑客，下面来对六剑客一一分析，如何打断点。
- 下面只是简单的教你如何打断点，对于六剑客是什么不再介绍，请看上篇文章。

SqlSession

- 既然是接口，肯定不能在接口方法上打断点，上文介绍有两个实现类，分别是 `DefaultSqlSession`、`SqlSessionTemplate`。那么SpringBoot在初始化的时候到底注入的是哪一个呢？这个就要看Mybatis的启动器的自动配置类了，其中有一段这样的代码，如下：

```
//如果容器中没有SqlSessionTemplate这个Bean，则注入
@Bean
@ConditionalOnMissingBean
public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory
sqlSessionFactory) {
    ExecutorType executorType = this.properties.getExecutorType();
    if (executorType != null) {
        return new SqlSessionTemplate(sqlSessionFactory,
executorType);
    } else {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}
```

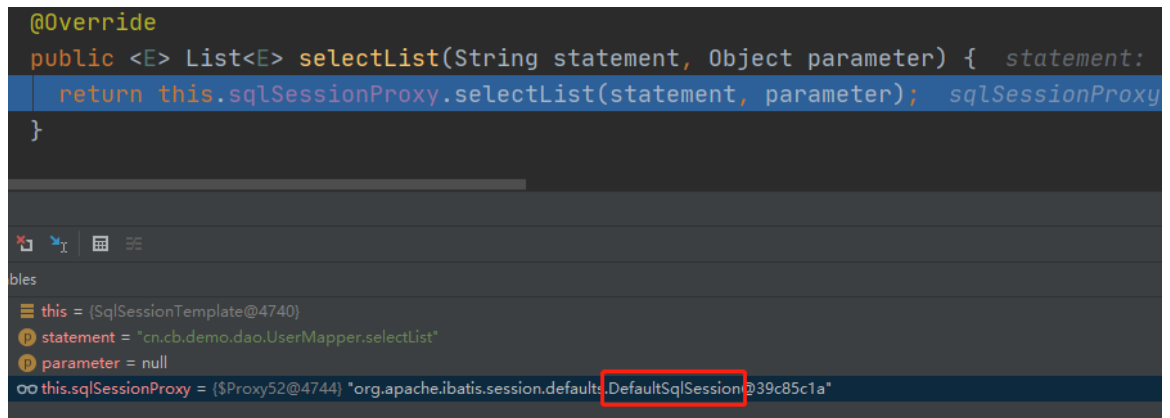
- 从上面的代码可以知道，SpringBoot启动时注入了`SqlSessionTemplate`，此时就肯定从`SqlSessionTemplate`入手了。它的一些方法如下图：



- 从上图的标记可以知道，首当其冲的就是构造方法了;既然是分析 `selectList()` 的查询流程，当然全部的 `selectList()` 方法要打上断点了;上篇文章也讲了 `Mapper` 的接口最终是走的动态代理生成的实例，因此此处的 `getMapper()` 也打上断点。
- 对于初入门的来说，上面三处打上断点已经足够了，但是如果你仔细看一眼 `selectList()` 方法，如下：

```
@Override
public <E> List<E> selectList(String statement) {
    //此处的sqlSessionProxy是什么，也是SqlSession类型的，此处断点运行到这里
    可以知道，就是DefaultSqlSession实例
    return this.sqlSessionProxy.selectList(statement);
}
```

- `sqlSessionProxy` 是什么，没关系，这个不能靠猜，那么此时断点走一波，走到 `selectList()` 方法内部，如下图：



```
@Override
public <E> List<E> selectList(String statement, Object parameter) { statement:
    return this.sqlSessionProxy.selectList(statement, parameter); sqlSessionProxy
}
```

bles

- this = {SqlSessionTemplate@4740}
- statement = "cn.cb.demo.dao.UserMapper.selectList"
- parameter = null
- this.sqlSessionProxy = (\$Proxy52@4744) "org.apache.ibatis.session.defaults.DefaultSqlSession@39c85c1a"

- 从上图可以很清楚的看到了，其实就是 `DefaultSqlSession`。哦，明白了，原来 `SqlSessionTemplate` 把过甩给了 `DefaultSqlSession` 了，太狡诈了。
- `DefaultSqlSession` 如何打断点就不用说了吧，自己搞搞吧。

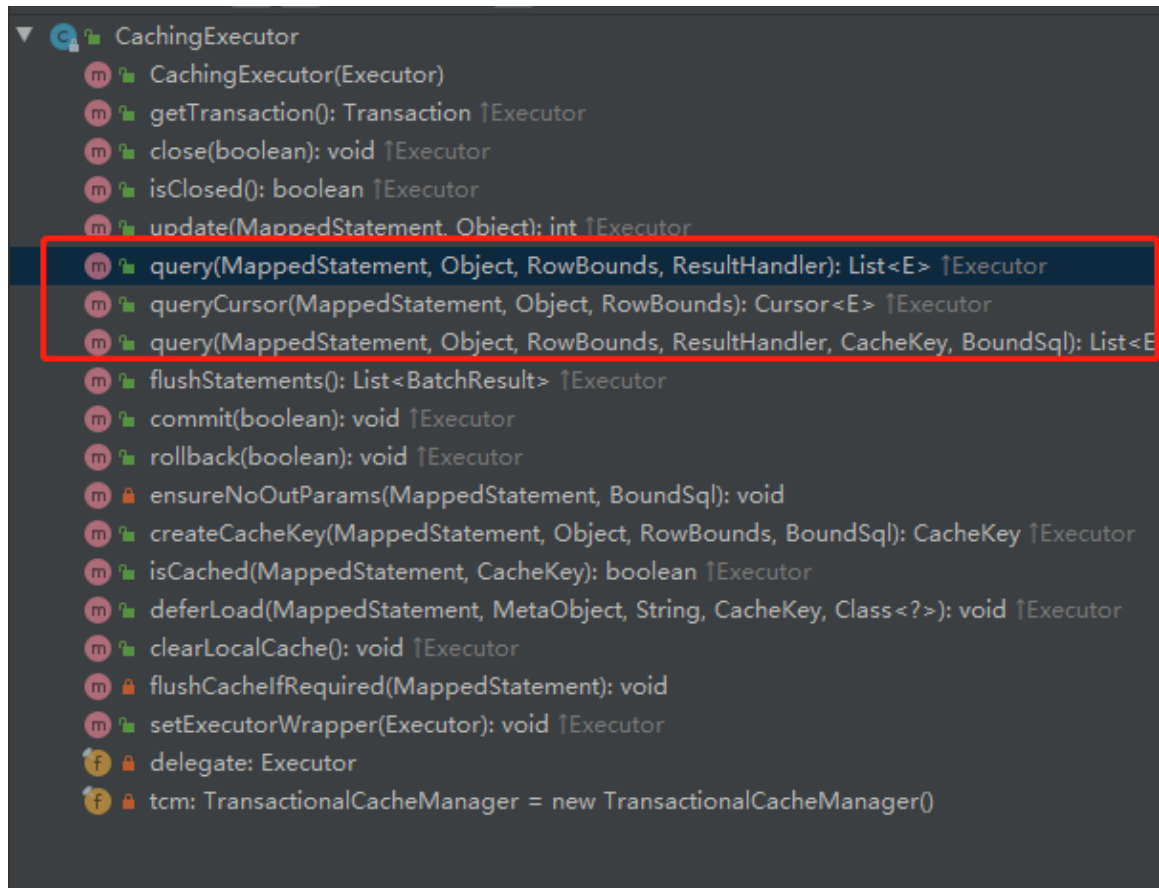
Executor

- 上面文章讲过执行器是什么作用，也讲过 `Mybatis` 内部是根据什么创建执行器的。此处不再赘述了。
- `SpringBoot` 整合各种框架有个特点，万变不离自动配置类，框架的一些初始化动作基本全是在自动配置类中完成，于是我们在配置类找一找在哪里注入了 `Executor`

的Bean，于是找到了如下的一段代码：

```
public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor; // 根据ExecutorType的类型创建不同的执行器
    if (ExecutorType.BATCH == executorType) {
        executor = new BatchExecutor(configuration: this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
        executor = new ReuseExecutor(configuration: this, transaction);
    } else {
        executor = new SimpleExecutor(configuration: this, transaction);
    }
    if (cacheEnabled) { // 如果cacheEnabled=true，默认配置就是true
        executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}
```

- 从上面的代码可以知道默认创建了 **CachingExecutor**，二级缓存的执行器，别管那么多，看看它重写了 **Executor** 的哪些接口，与 **selectList()** 相关的方法打上断点，如下图：



```
▼ CachingExecutor
  ◯ CachingExecutor(Executor)
  ◯ getTransaction(): Transaction ↑Executor
  ◯ close(boolean): void ↑Executor
  ◯ isClosed(): boolean ↑Executor
  ◯ update(MappedStatement, Object): int ↑Executor
  ◯ query(MappedStatement, Object, RowBounds, ResultHandler): List<E> ↑Executor
  ◯ queryCursor(MappedStatement, Object, RowBounds): Cursor<E> ↑Executor
  ◯ query(MappedStatement, Object, RowBounds, ResultHandler, CacheKey, BoundSql): List<E>
  ◯ flushStatements(): List<BatchResult> ↑Executor
  ◯ commit(boolean): void ↑Executor
  ◯ rollback(boolean): void ↑Executor
  ◯ ensureNoOutParams(MappedStatement, BoundSql): void
  ◯ createCacheKey(MappedStatement, Object, RowBounds, BoundSql): CacheKey ↑Executor
  ◯ isCached(MappedStatement, CacheKey): boolean ↑Executor
  ◯ deferLoad(MappedStatement, MetaObject, String, CacheKey, Class<?>): void ↑Executor
  ◯ clearLocalCache(): void ↑Executor
  ◯ flushCacheIfRequired(MappedStatement): void
  ◯ setExecutorWrapper(Executor): void ↑Executor
  ◯ delegate: Executor
  ◯ tcm: TransactionalCacheManager = new TransactionalCacheManager()
```

- 从上图也知道哪些方法和 **selectList()** 相关了，显然的 **query** 是查询的意思，别管那么多，先打上断点。
- 此时再仔细瞅一眼 **query()** 的方法怎么执行的，哦？发现了什么，如下：

@Override

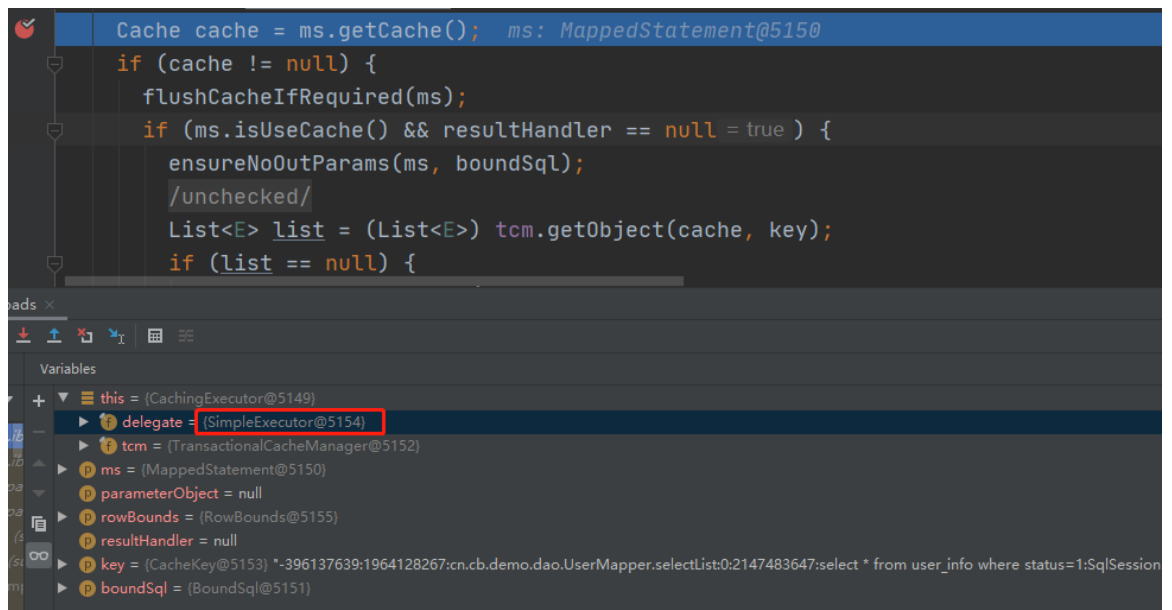
```
public <E> List<E> query(MappedStatement ms, Object
parameterObject, RowBounds rowBounds, ResultHandler resultHandler,
CacheKey key, BoundSql boundSql)
```

```

throws SQLException {
    //先尝试从缓存中获取
    Cache cache = ms.getCache();
    if (cache != null) {
        flushCacheIfRequired(ms);
        if (ms.isUseCache() && resultHandler == null) {
            ensureNoOutParams(ms, boundSql);
            @SuppressWarnings("unchecked")
            List<E> list = (List<E>) tcm.getObject(cache, key);
            if (list == null) {
                list = delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
                tcm.putObject(cache, key, list); // issue #578 and #116
            }
            return list;
        }
    }
    //没有缓存，直接调用delegate的query方法
    return delegate.query(ms, parameterObject, rowBounds,
resultHandler, key, boundSql);
}

```

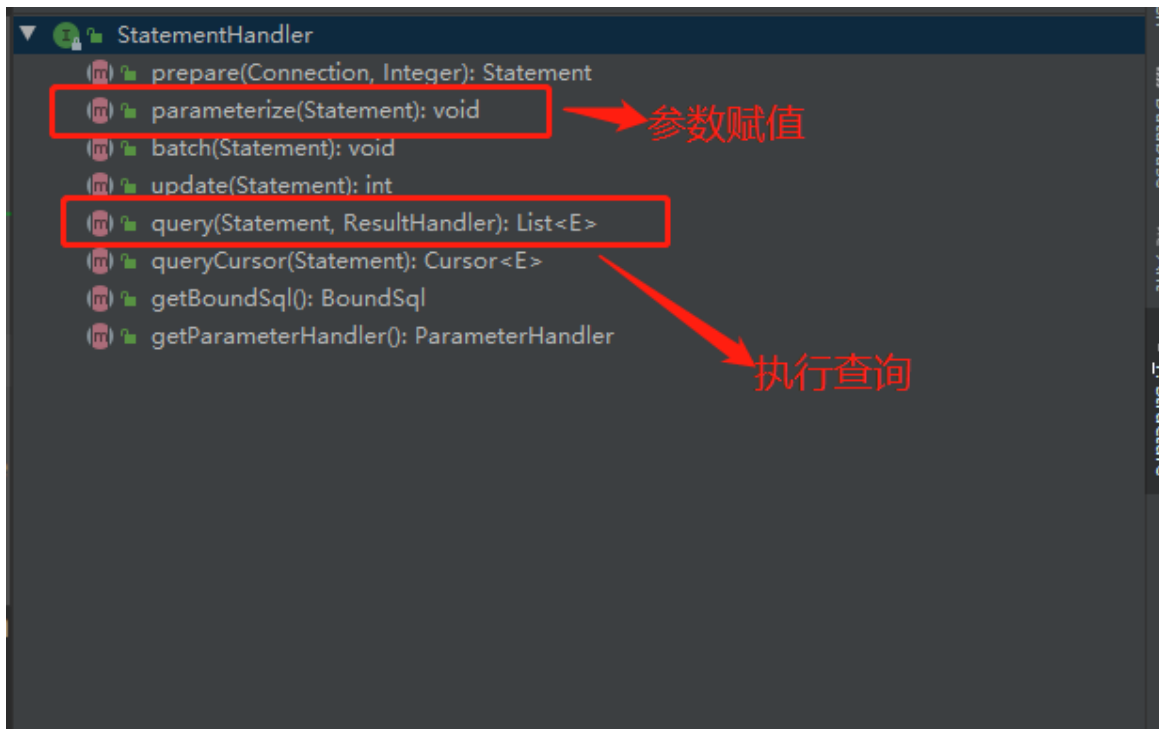
- 从上面的代码知道，有缓存了，直接返回了，没有缓存，调用了 `delegate` 中的 `query` 方法，那么这个 `delegate` 是哪个类的对象呢？参照 `sqlSession` 的分析的方法，调试走起，可以知道是 `SimpleExecutor` 的实例，如下图：



- 后面的 `SimpleExecutor` 如何打断点就不再说了，自己尝试找找。

StatementHandler

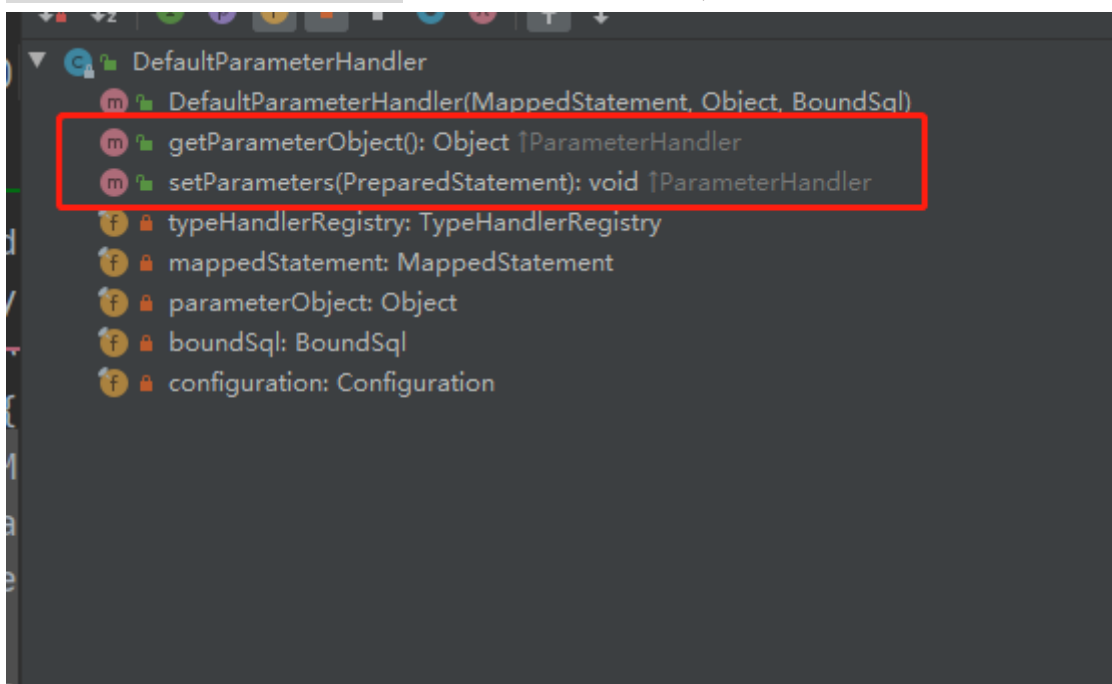
- 很熟悉的一个接口，在学JDBC的时候就接触过类似的，执行语句和设置参数的作用。
- 这个接口很简单，大佬写的代码，看到方法名就知道这个方法是干什么的，如下图：



- 最重要的实现类是什么？当然是 `PreparedStatementHandler`，因此在对应的方法上打上断点即可。

ParameterHandler

- 这个接口很简单，也别选择了，总共两个方法，一个设置，一个获取，在实现类 `DefaultParameterHandler` 中对应的方法上打上断点即可。



TypeHandler

- 类型处理器，也是一个简单的接口，总共两个方法，一个设置参数的转换，一个对结果的转换，啥也别说了，自己找到对应参数类型的处理器，在其中的方法打上断点。

ResultSetHandler

- 结果处理器，负责对结果的处理，总共三个方法，一个实现类 `DefaultResultSetHandler`，全部安排断点。

总结

- 授人以鱼不授人以渔，与其都分析了给你看，不如教会你阅读源码的方式，先自己去研究，不仅仅是阅读Mybatis的源码是这样，阅读任何框架的源码都是如此，比如Spring的源码，只要找到其中重要的组件，比如前置处理器，后置处理器，事件触发器等等，一切都迎刃而解。
- 如果你觉得作者写的不错，有所收获，不妨关注分享一波，后续更多精彩内容更新。



Mybatis如何执行Select语句，你真的知道吗？

前言

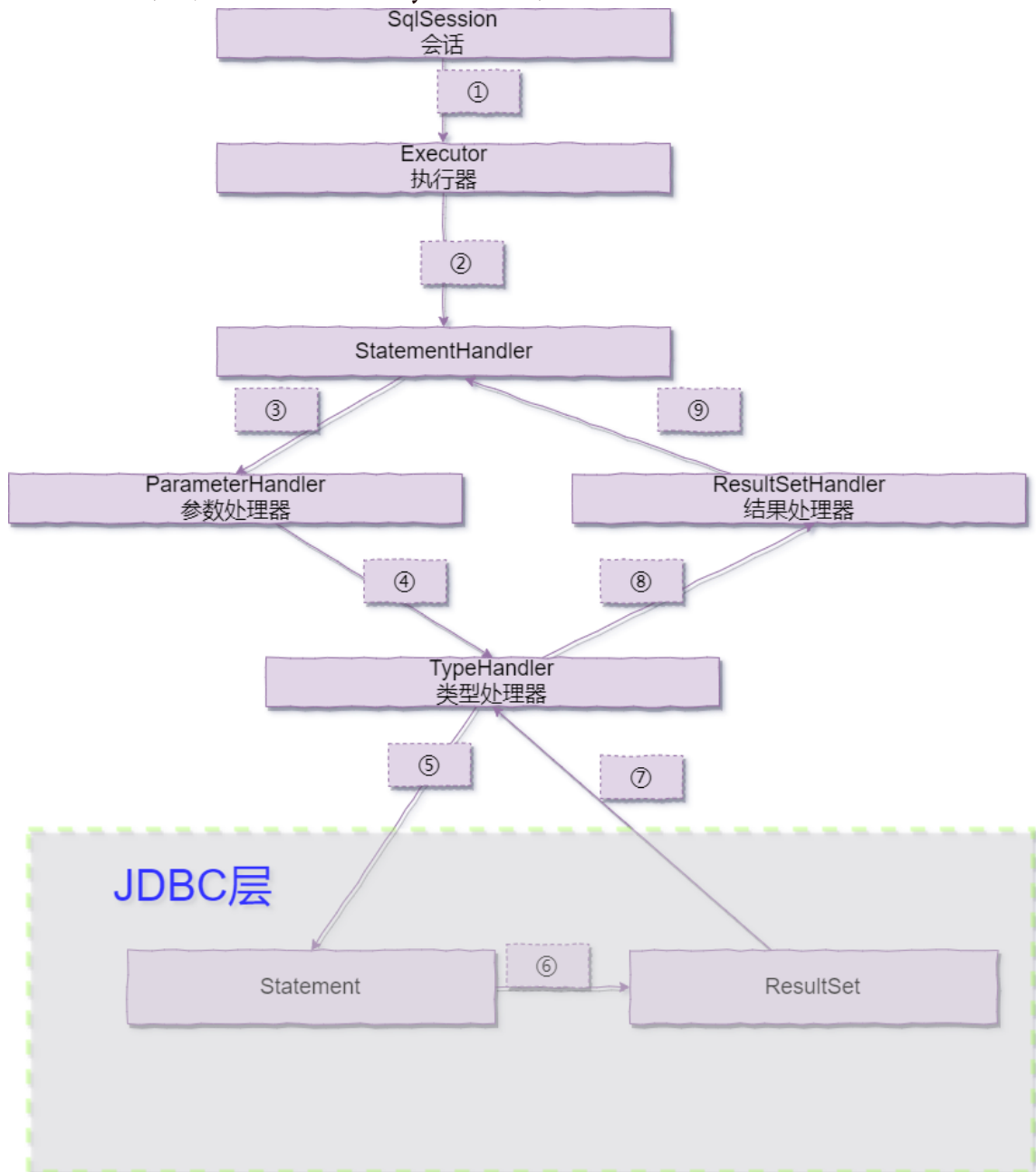
- 本篇文章是Mybatis源码分析的第三篇，前两篇分别介绍了Mybatis的重要组件和围绕着Mybatis中的重要组件教大家如何阅读源码的一些方法，有了前面两篇文章的基础，来看这篇文章的才不会觉得吃力，如果没有看过的朋友，陈某建议去看看，两篇文章分别是[Mybatis源码解析之六剑客](#)和[Mybatis源码如何阅读，教你一招!!!](#)。
- 今天接上一篇，围绕Mybatis中的`selectList()`来看一看Mybatis底层到底做了什么，有什么高级的地方。

环境准备

- 本篇文章讲的一切内容都是基于`Mybatis3.5`和`SpringBoot-2.3.3.RELEASE`。
- 由于此篇文章是基于前两篇文章的基础之上，因此重复的内容不再详细赘述了。

撸起袖子就是干

- 二话不说，先来一张流程图，Mybatis六剑客，如下：



- 上图中的这六剑客在前面两篇文章中已经介绍的非常清楚了，此处略过。为什么源码解析的每一篇文章中都要放一张这个流程图呢？因为Mybatis底层就是围绕着这六剑客展开的，我们需要从全局掌握Mybatis的源码究竟如何执行的。

测试环境搭建

- 举个栗子：根据用户id查询用户信息，Mapper定义如下：

```
List<UserInfo> selectList(@Param("userIds") List<String> userIds);
```

- 对应XML配置如下：

```

<mapper namespace="cn.cb.demo.dao.UserMapper">
    <!--开启二级缓存-->
    <cache/>
    <select id="selectList"
resultType="cn.cb.demo.domain.UserInfo">
        select * from user_info where status=1
        and user_id in
        <foreach collection="userIds" item="item" open="("
separator="," close=")" >
            #{item}
        </foreach>
    </select>
</mapper>

```

- 单元测试如下：

```

@Test
void contextLoads() {
    List<UserInfo> userInfos =
userMapper.selectList(Arrays.asList("192","198"));
    System.out.println(userInfos);
}

```

DEBUG走起

- 具体在哪里打上断点，上篇文章已经讲过了，不再赘述了。
- 由于SpringBoot与Mybatis整合之后，自动注入的是SqlSessionTemplate，因此代码执行到
org.mybatis.spring.SqlSessionTemplate#selectList(java.lang.String,
java.lang.Object)，如图1：

```
    * {@inheritDoc}
    */
    @Override
    public <E> List<E> selectList(String statement, Object parameter) { statement: "cn.cb.d...
        return this.sqlSessionProxy.selectList(statement, parameter); sqlSessionProxy: "org...
    }

    /**
        实际调用的是DefaultSqlSession
    */

this = (SqlSessionTemplate@4743)
sqlSessionFactory = (DefaultSqlSessionFactory@4745)
executorType = (ExecutorType@4746) "SIMPLE"
sqlSessionProxy = ($Proxy52@4747) "org.apache.ibatis.session.defaults.DefaultSqlSession@3f96f020"
exceptionTranslator = (MyBatisExceptionTranslator@4748)
statement = "cn.cb.demo.dao.UserMapper.selectList"
parameter = (MapperMethod$ParamMap@5148) size = 2
this.sqlSessionProxy = ($Proxy52@4747) "org.apache.ibatis.session.defaults.DefaultSqlSession@550e9be6"
```

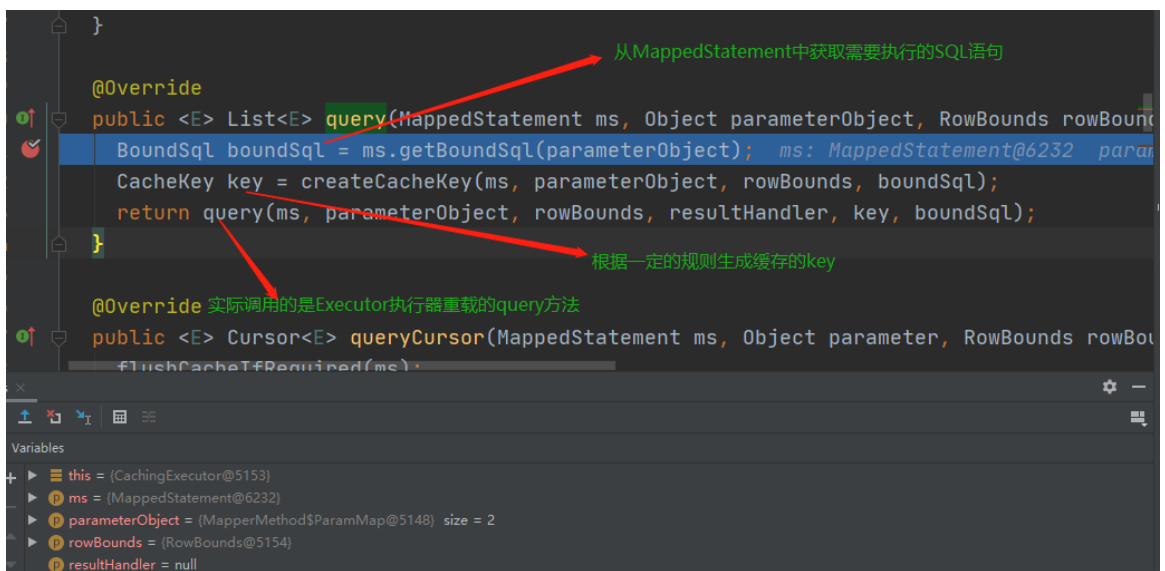
- 从源码可以看到，实际调用的还是DefaultSqlSession中的selectList方法。如下图2：

```
    @Override
    public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds)
    {
        try {
            MappedStatement ms = configuration.getMappedStatement(statement); configuration:
            return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
        } catch (Exception e) {
            throw ExceptionFactory.wrapException("Error querying database. Cause: " + e, e);
        } finally {
            MappedStatement中存储了这条SQL语句的详细信息，包括resultType,resultMap等内容
            ErrorContext.instance().reset();
        }
    }

    DefaultSqlSession最终调用的还是Executor（执行器）中的query方法

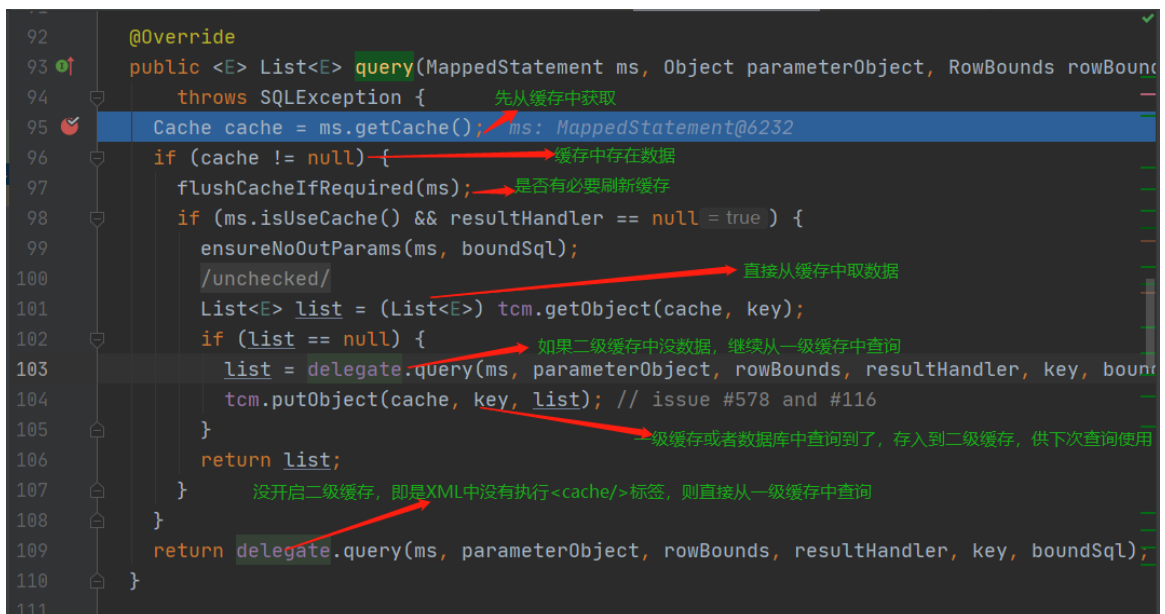
Variables
this = (DefaultSqlSession@5151)
statement = "cn.cb.demo.dao.UserMapper.selectList"
parameter = (MapperMethod$ParamMap@5148) size = 2
rowBounds = (RowBounds@5154)
configuration = (Configuration@5152)
executor = (CachingExecutor@5153) SpringBoot全局中默认开启二级缓存，因此使用的是CachingExecutor
```

- 具体的逻辑如下：
 - 根据Mapper方法的全类名从Mybatis的配置中获取到这条SQL的详细信息，比如paramterType,resultMap等等。
 - 既然开启了二级缓存，肯定先要判断这条SQL是否缓存过，因此实际调用的是CachingExecutor这个缓存执行器。
- DefaultSqlSession只是简单的获取SQL的详细配置，最终还是把任务交给了Executor（当然这里走的是二级缓存，因此交给了缓存执行器）。下面DEBUG走到CachingExecutor#query(MappedStatement, java.lang.Object, RowBounds, ResultHandler)，源码如下图3：



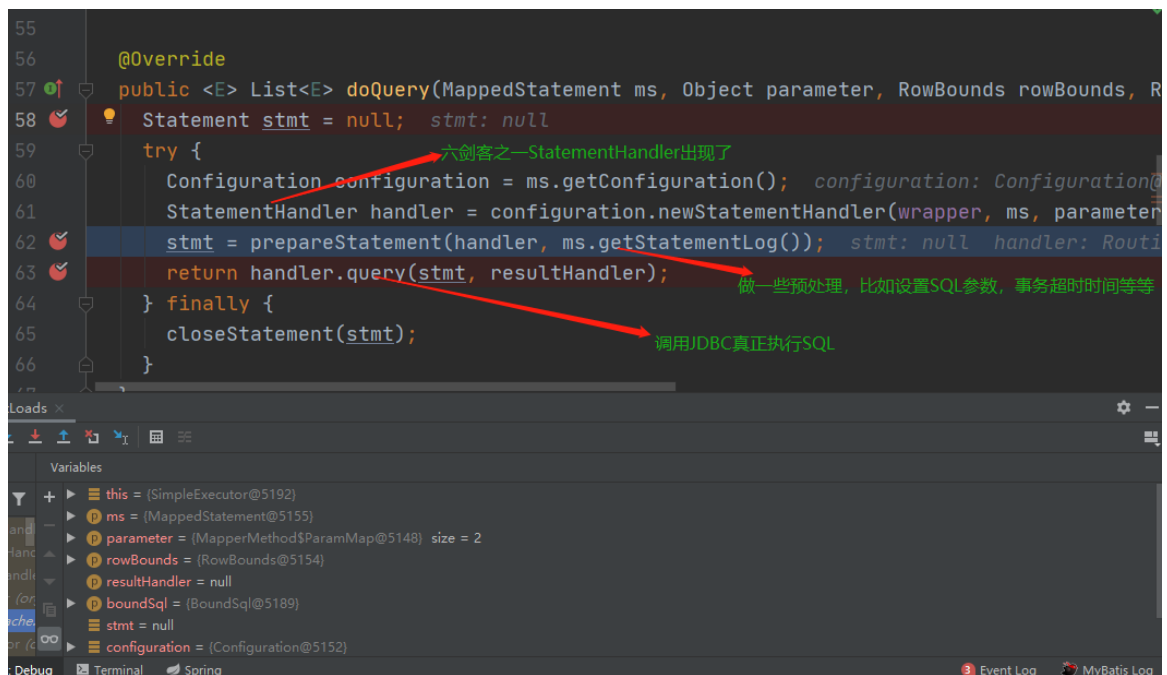
- 上图中的query方法实际做了两件事，实际执行的查询还是其中重载的方法

List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql), 如下图4:

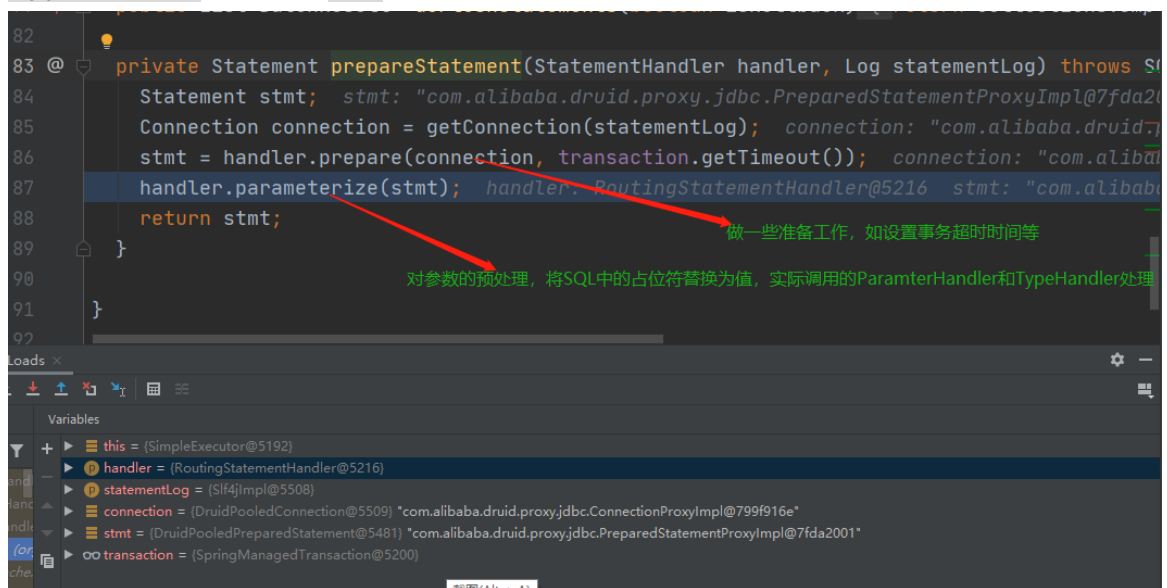


- 根据上图源码的分析，其实CachingExecutor执行的逻辑并不是很难，反倒很容易理解，具体的逻辑如下：
 - 如果开启了二级缓存，先根据cacheKey从二级缓存中查询，如果查询到了直接返回
 - 如果未开启二级缓存，再执行BaseExecutor中的query方法从一级缓存中查询。
 - 如果二级缓存中未查询到数据，再执行BaseExecutor中的query方法从一级缓存中查询。
 - 将查询到的结果存入到二级缓存中。
- BaseExecutor中的query方法无非就是从一级缓存中取数据，没查到再从数据库中取数据，一级缓存实际就是一个Map结构，这里不再细说，真正执行SQL从数据库中取数据的是SimpleExecutor中的public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,

ResultHandler resultHandler, BoundSql boundSql) 方法，源码如下 图5：

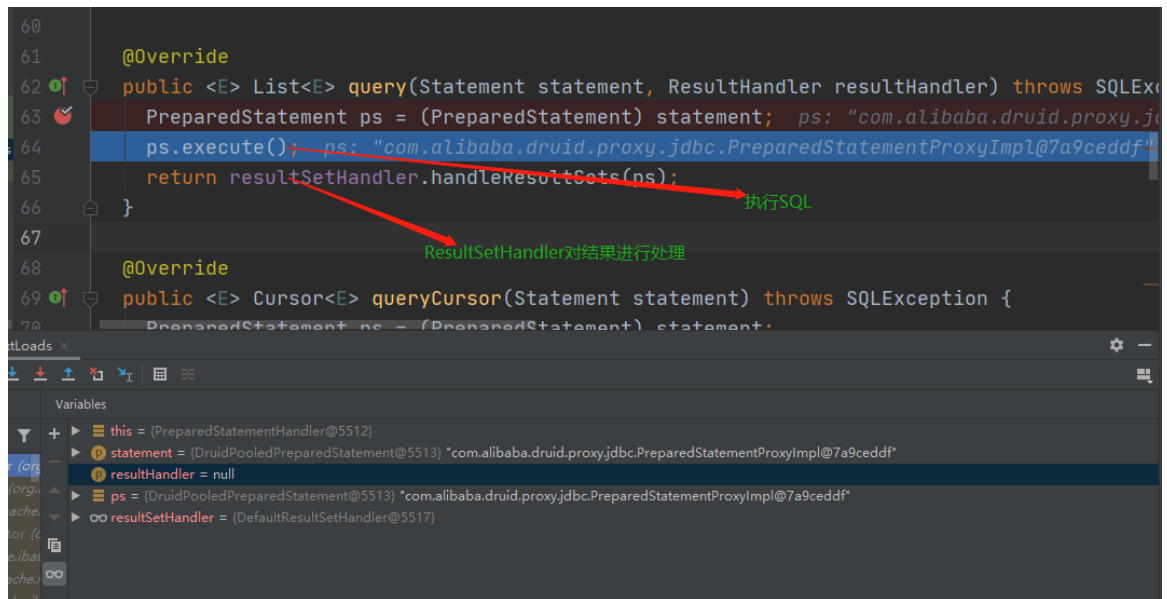


- 从上面的源码也是可以知道，在真正执行SQL之前，是要调用 `prepareStatement(handler, ms.getStatementLog())` 方法做一些参数的预处理的，其中涉及到了六大剑客的另外两位，分别是 `ParameterHandler` 和 `TypeHandler`，源码如图6：



- 从上图可以知道设置SQL参数的真正方法是 `handler.parameterize(stmt)`，真正执行的是 `DefaultParameterHandler` 中的 `setParameters` 方法，由于篇幅较长，简单的说一下思路：
 - 获取所有参数的映射
 - 循环遍历，获取参数的值，使用对应的 `TypeHandler` 将其转换成相应类型的参数。
 - 真正的设置参数的方法是 `TypeHandler` 中 `setParameter` 方法

- 继续图6的逻辑，参数已经设置完了，此时就该执行SQL了，真正执行SQL的是 `PreparedStatementHandler` 中的 `<E> List<E> query(Statement statement, ResultHandler resultHandler)` 方法，源码如下图7：



```

60
61     @Override
62     public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {
63         PreparedStatement ps = (PreparedStatement) statement;
64         ps.execute();
65         return resultSetHandler.handleResultSets(rs);
66     }
67
68     @Override
69     public <E> Cursor<E> queryCursor(Statement statement) throws SQLException {
70         PreparedStatement ps = (PreparedStatement) statement;

```

- 上图的逻辑其实很简单，一个是JDBC执行SQL语句，一个是调用六剑客之一的 `ResultSetHandler` 对结果进行处理。
- 真正对结果进行处理的是 `DefaultResultSetHandler` 中的 `handleResultSets` 方法，源码比较复杂，这里就不再展示了，具体的逻辑如下：
 - 获取结果映射(`resultMap`)，如果没有指定，使用内置的结果映射
 - 遍历结果集，对SQL返回的每个结果通过结果集和 `TypeHandler` 进行结果映射。
 - 返回结果
- `ResultSetHandler`对结果处理结束之后就会返回。至此一条 `selectList()` 如何执行的大概心里已经有了把握，其他的更新，删除都是大同小异。

总结

- Mybatis的源码算是几种常用框架中比较简单的，都是围绕六大组件进行的，只要搞懂了每个组件是什么角色，有什么作用，一切都会很简单。
- 一条select语句简单执行的逻辑总结如下（前提：默认配置）：
 - SqlSession:** `#SqlSessionTemplate.selectList()` 实际调用 `#DefaultSqlSession.selectList()`
 - Executor:** `#DefaultSqlSession.query()` 实际调用的是 `#CachingExecutor().query()`，如果二级缓存中存在直接返回，不存在调用 `#BaseExecutor.query()` 查询一级缓存，如果一级缓存中存在直接返回。不存在调用 `#SimpleExecutor.doQuery()` 方法查询数据库。

c. **StatementHandler**: `#SimpleExecutor.doQuery()` 生成

`StatementHandler`实例，执行

`#PreparedStatementHandler.parameterize()` 方法设置参数，实际调用的是 `#ParameterHandler.setParameters()` 方法，该方法内部调用 `TypeHandler.setParameter()` 方法进行类型转换;参数设置成功后，调用 `#PreparedStatementHandler.parameterize().query()` 方法执行 SQL，返回结果

d. **ResultSetHandler**:

`#DefaultResultSetHandler.handleResultSets()` 对返回的结果进行处理，内部调用 `#TypeHandler.getResult()` 对结果进行类型转换。全部映射完成，返回结果。

- 以上就是六剑客在Select的执行流程，如果有错误之处欢迎指正，如果觉得陈某写得不错，有所收获，关注分享一波。



Mybatis Log plugin破解

前言

- 今天重新装了IDEA2020，顺带重装了一些插件，毕竟这些插件都是习惯一直在用，其中一款就是Mybatis Log plugin，按照往常的思路，在IDEA插件市场搜索安装，艹，眼睛一瞟，竟然收费了，对于我这种支持盗版的人来说太难了，于是自己开始捣鼓各种尝试破解，下文分享自己的破解方式。

什么是Mybatis Log plugin

- 举个栗子，通常在找bug的时候都会查看执行了什么SQL，想把这条SQL拼接出来执行调试，可能有些小白还在傻傻的把各个参数复制出来，补到?占位符中，哈哈。

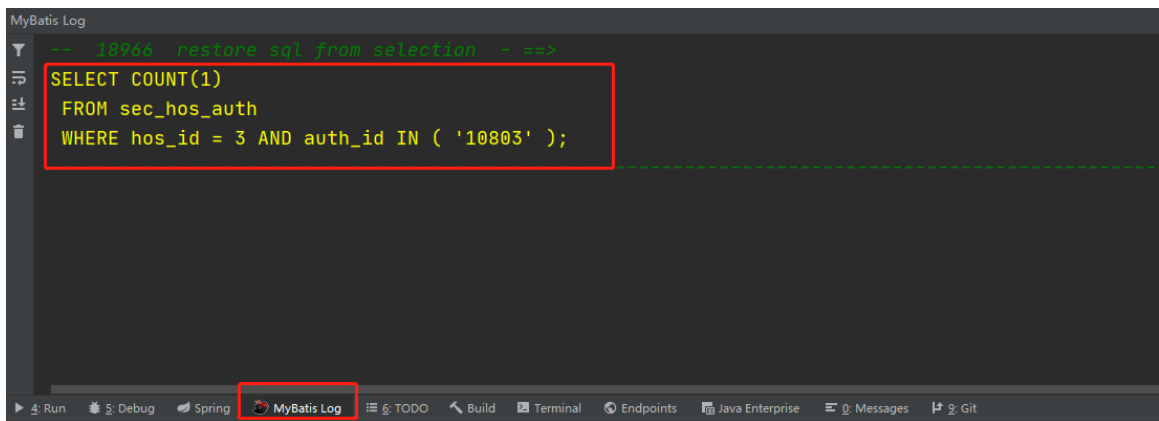


- 简单的说就是能根据log4j的打印的sql日志一键生成执行的sql语句。

- 类似如下一个日志信息：

```
==> Preparing: SELECT COUNT(1) FROM sec_hos_auth WHERE hos_id = ? AND auth_id IN ( ? )  
==> Parameters: 3(Integer), 10803(String)  
<== Total: 1
```

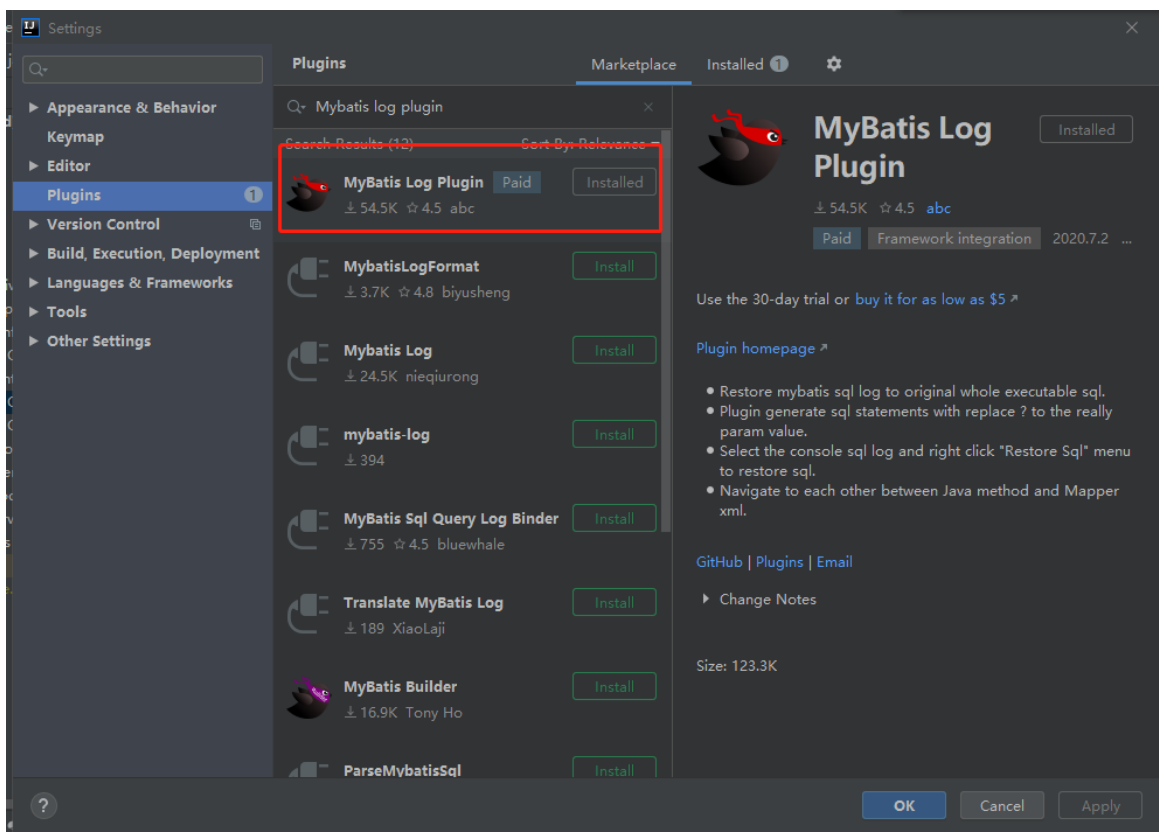
- 如果使用Log plugin这个插件，将会很容易的把参数添加到sql语句中得到一条完整的sql，效果如下：



- 一旦开启了Mybatis Log plugin这个插件，在程序运行过程中只要有SQL语句都会自动生成在Mybatis Log这个界面，当然也可以自己关掉。

如何安装

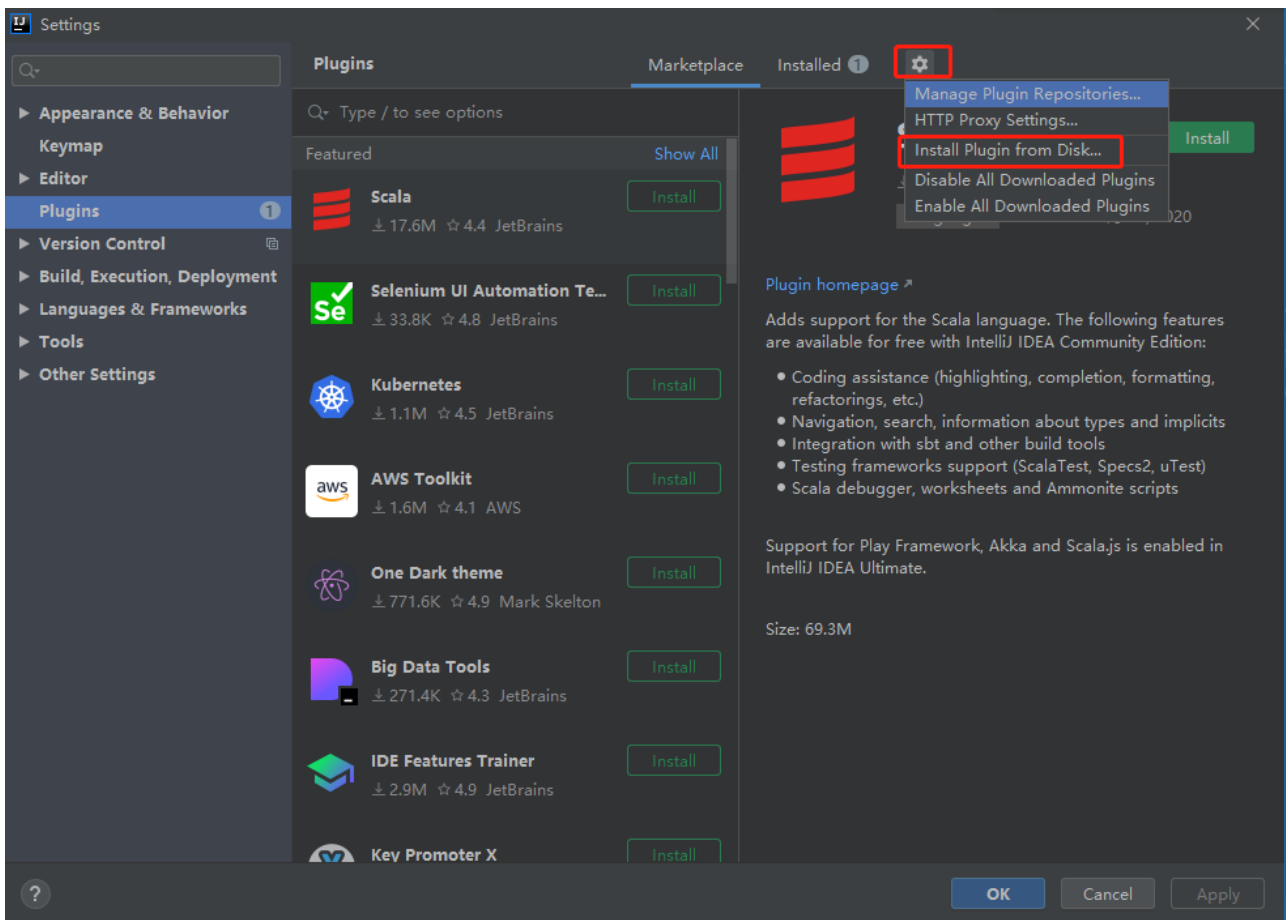
- Setting->plugin->Marketplace搜索框输入Mybatis Log plugin，如下：



- 很遗憾的是，IDEA2020中已经开始收费了，艹，对于一向支持盗版的我来说，很不爽~

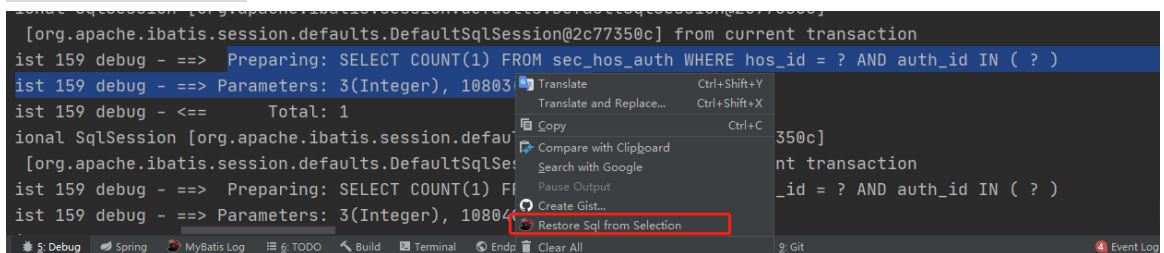
如何破解

- 下载jar包 `plugin.intelliij.assistant.mybatislog-2020.1-1.0.3.jar`，文末附有下载方式。
- `setting->plugin->设置-> install plugin from Disk...`



如何使用

- 日志中从 `Preparing` 到 `Parameters` 这两行的参数选中，右键选择 `restore sql from Selection`



- 此时将会在 `Mybatis Log` 界面出现完整的SQL语句。

总结

- 对于复杂的SQL语句来说，Mybatis Log plugin这款插件简直是太爱了，能够自动拼接参数生成执行的SQL语句。
- 老规矩，关注公众号【码猿技术专栏】，公众号回复 `mybatis log` 获取破解包。

