

一、MyBatis插件

插件是一种常见的扩展方式，大多数开源框架也都支持用户通过添加自定义插件的方式来扩展或者改变原有的功能，MyBatis中也提供的有插件，虽然叫插件，但是实际上是通过拦截器(Interceptor)实现的，在MyBatis的插件模块中涉及到责任链模式和JDK动态代理，这两种设计模式的技术知识也是大家要提前掌握的。

1. 自定义插件

首先我们来看下一个自定义的插件我们要如何实现。

1.1 创建Interceptor实现类

我们创建的拦截器必须要实现Interceptor接口，Interceptor接口的定义为

```
public interface Interceptor {  
  
    // 执行拦截逻辑的方法  
    Object intercept(Invocation invocation) throws Throwable;  
  
    // 决定是否触发 intercept()方法  
    default Object plugin(Object target) {  
        return Plugin.wrap(target, this);  
    }  
  
    // 根据配置 初始化 Intercept 对象  
    default void setProperties(Properties properties) {  
        // NOP  
    }  
  
}
```

在MyBatis中Interceptor允许拦截的内容是

- Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler (getParameterObject, setParameters)
- ResultSetHandler (handleResultSets, handleOutputParameters)
- StatementHandler (prepare, parameterize, batch, update, query)

我们创建一个拦截Executor中的query和close的方法

```
package com.boboedu.interceptor;  
  
import org.apache.ibatis.executor.Executor;  
import org.apache.ibatis.mapping.MappedStatement;  
import org.apache.ibatis.plugin.*;  
import org.apache.ibatis.session.ResultHandler;  
import org.apache.ibatis.session.RowBounds;
```

```

import java.util.Properties;

/**
 * 自定义的拦截器
 * @Signature 注解就可以表示一个方法签名， 唯一确定一个方法
 */
@Intercepts({
    @Signature(
        type = Executor.class // 需要拦截的类型
        ,method = "query"      // 需要拦截的方法
        // args 中指定 被拦截方法的 参数列表
        ,args={MappedStatement.class,Object.class, RowBounds.class,
ResultHandler.class}
    ),
    @Signature(
        type = Executor.class
        ,method = "close"
        ,args = {boolean.class}
    )
})
public class FirstInterceptor implements Interceptor {

    private int testProp;

    /**
     * 执行拦截逻辑的方法
     * @param invocation
     * @return
     * @throws Throwable
     */
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("FirtInterceptor 拦截之前 ....");
        Object obj = invocation.proceed();
        System.out.println("FirtInterceptor 拦截之后 ....");
        return obj;
    }

    /**
     * 决定是否触发 intercept方法
     * @param target
     * @return
     */
    @Override
    public Object plugin(Object target) {
        return Plugin.wrap(target,this);
    }

    @Override
    public void setProperties(Properties properties) {
        System.out.println("---->" +properties.get("testProp"));
    }

    public int getTestProp() {
        return testProp;
    }

    public void setTestProp(int testProp) {

```

```
        this.testProp = testProp;
    }
}
```

1.2 配置拦截器

创建好自定义的拦截器后，我们需要在全局配置文件中添加自定义插件的注册

```
<plugins>
    <plugin interceptor="com.bobo.interceptor.FirstInterceptor">
        <property name="testProp" value="1000"/>
    </plugin>
</plugins>
```

1.3 运行程序

然后我们执行对应的查询操作。

```
@Test
public void test1() throws Exception{
    // 1.获取配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2.加载解析配置文件并获取SqlSessionFactory对象
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3.根据SqlSessionFactory对象获取SqlSession对象
    SqlSession sqlSession = factory.openSession();
    // 4.通过SqlSession中提供的 API方法来操作数据库
    List<User> list =
sqlSession.selectList("com.bobo.mapper.UserMapper.selectUserList");
    for (User user : list) {
        System.out.println(user);
    }
    // 5.关闭会话
    sqlSession.close();
}
```

拦截的query方法和close方法的源码位置在如下：

```
DefaultSqlSession selectList
XMLMapperBuilder.java Test1.java SqlSession.java DefaultSqlSession.java Executor.java CachingExecutor.java FirstInterceptor.java mybatis-config.xml Intercep
// 让参数少的调用参数多的方法, 只实现一次
return this.selectList(statement, parameter, RowBounds.DEFAULT);
}

@Override
public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
    try {
        MappedStatement ms = configuration.getMappedStatement(statement);
        // 如果 cacheEnabled = true (默认), Executor会被 CachingExecutor装饰
        return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
    } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error flushing statements.", e);
    } finally {
        ErrorContext.instance().sql(statement);
    }
}

@Override
public void select(String statement, Object parameter, ResultHandler handler) {
}
```

```
Tools VCS Window Help MybatisDemo - G:\tools\Mybatis\mybatis-3\src\main\java\org\apache\ibatis\session\defaults\DefaultSqlSession.java
DefaultSqlSession close
259 } finally {
260     ErrorContext.instance().reset();
261 }
262 }
263
264 @Override
265 public void close() {
266     try {
267         executor.close(isCommitOrRollbackRequired(false));
268         closeCursors();
269         dirty = false;
270     } finally {
271         ErrorContext.instance().reset();
272     }
273 }
274
275 private void closeCursors() {
276     if (cursorList != null && !cursorList.isEmpty()) {
277         for (Cursor<?> cursor : cursorList) {
}
```

2. 插件实现原理

自定义插件的步骤还是比较简单的，接下来我们分析下插件的实现原理是怎么回事。

2.1 初始化操作

首先我们来看下在全局配置文件加载解析的时候做了什么操作。

```
XMLConfigBuilder - parseConfiguration
11  SqlSessionFactoryBuilder.java  XMLConfigBuilder.java  SqlSession.java  DefaultSqlSession.java  Executor.java  CachingExecutor.java  FirstInterceptor.java  mybatis-config.xml  InterceptorChain.java
12  Properties settings = settingsAsProperties(root.evalNode("settings"));
13  // 读取文件
14  loadCustomVfs(settings);
15  // 日志设置
16  loadCustomLogImpl(settings);
17  // 类型别名
18  typeAliasesElement(root.evalNode("typeAliases"));
19  // 插件
20  pluginElement(root.evalNode("plugins"));
21  // 用于创建对象
22  objectFactoryElement(root.evalNode("objectFactory"));
23  // 用于对对象进行加工
24  objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
25  // 反射工具箱
26  reflectorFactoryElement(root.evalNode("reflectorFactory"));
27  // settings 子标签赋值，默认值就是在这里提供的 >>
28  settingsElement(settings);
29  // read it after objectFactory and objectWrapperFactory issue #631
30  // 创建了数据源 >>
```

进入方法内部可以看到具体的解析操作

```
private void pluginElement(XNode parent) throws Exception {
    if (parent != null) {
        for (XNode child : parent.getChildren()) {
            // 获取<plugin> 节点的 interceptor 属性的值
            String interceptor = child.getStringAttribute("interceptor");
            // 获取<plugin> 下的所有的properties子节点
            Properties properties = child.getChildrenAsProperties();
            // 获取 Interceptor 对象
            Interceptor interceptorInstance = (Interceptor)
            resolveClass(interceptor).getDeclaredConstructor().newInstance();
            // 设置 interceptor 的属性
            interceptorInstance.setProperties(properties);
            // Configuration中记录 Interceptor
            configuration.addInterceptor(interceptorInstance);
        }
    }
}
```

该方法用来解析全局配置文件中的plugins标签，然后对应的创建Interceptor对象，并且封装对应的属性信息。最后调用了Configuration对象中的方法。

configuration.addInterceptor(interceptorInstance)

```
public void addInterceptor(Interceptor interceptor) {
    interceptorChain.addInterceptor(interceptor);
}
```

通过这个代码我们发现我们自定义的拦截器最终是保存在了InterceptorChain这个对象中。而InterceptorChain的定义为

```
public class InterceptorChain {

    // 保存所有的 Interceptor 也就我所有的插件是保存在 Interceptors 这个List集合中的
    private final List<Interceptor> interceptors = new ArrayList<>();

    //
```

```

public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) { // 获取拦截器链中的所有拦截器
        target = interceptor.plugin(target); // 创建对应的拦截器的代理对象
    }
    return target;
}

public void addInterceptor(Interceptor interceptor) {
    interceptors.add(interceptor);
}

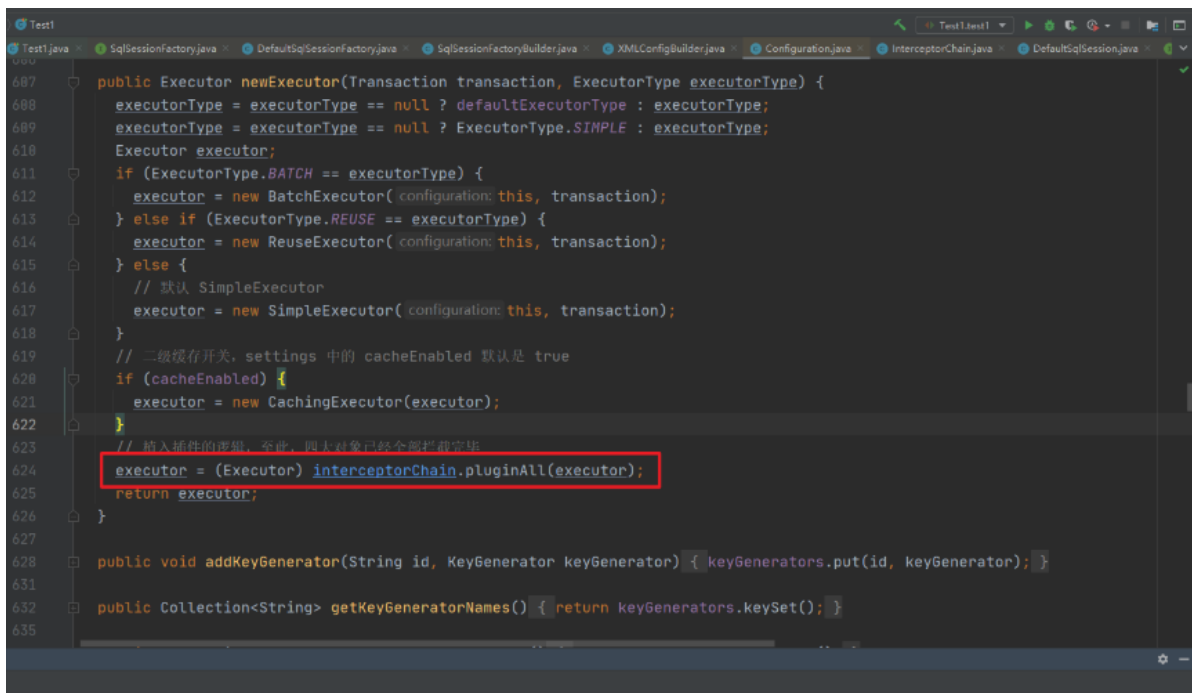
public List<Interceptor> getInterceptors() {
    return Collections.unmodifiableList(interceptors);
}
}

```

2.2 如何创建代理对象

在解析的时候创建了对应的Interceptor对象，并保存在了InterceptorChain中，那么这个拦截器是如何和对应的目标对象进行关联的呢？首先拦截器可以拦截的对象是Executor,ParameterHandler,ResultSetHandler,StatementHandler.那么我们来看下这四个对象在创建的时候又什么要注意的

2.2.1 Executor



我们可以看到Executor在装饰完二级缓存后会通过pluginAll来创建Executor的代理对象。

```

public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) { // 获取拦截器链中的所有拦截器
        target = interceptor.plugin(target); // 创建对应的拦截器的代理对象
    }
    return target;
}

```

进入plugin方法中，我们会进入到

```
// 决定是否触发 intercept()方法
default Object plugin(Object target) {
    return Plugin.wrap(target, this);
}
```

然后进入到MyBatis给我们提供的Plugin工具类的实现 wrap方法中。

```
/**
 * 创建目标对象的代理对象
 * 目标对象 Executor ParameterHandler ResultSetHandler StatementHandler
 * @param target 目标对象
 * @param interceptor 拦截器
 * @return
 */
public static Object wrap(Object target, Interceptor interceptor) {
    // 获取用户自定义 Interceptor中@Signature注解的信息
    // getSignatureMap 负责处理@Signature 注解
    Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
    // 获取目标类型
    Class<?> type = target.getClass();
    // 获取目标类型 实现的所有的接口
    Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
    // 如果目标类型有实现的接口 就创建代理对象
    if (interfaces.length > 0) {
        return Proxy.newProxyInstance(
            type.getClassLoader(),
            interfaces,
            new Plugin(target, interceptor, signatureMap));
    }
    // 否则原封不动的返回目标对象
    return target;
}
```

Plugin中的各个方法的作用

```
public class Plugin implements InvocationHandler {

    private final Object target; // 目标对象
    private final Interceptor interceptor; // 拦截器
    private final Map<Class<?>, Set<Method>> signatureMap; // 记录 @Signature 注解的信息

    private Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Method>> signatureMap) {
        this.target = target;
        this.interceptor = interceptor;
        this.signatureMap = signatureMap;
    }

    /**
     * 创建目标对象的代理对象
     * 目标对象 Executor ParameterHandler ResultSetHandler StatementHandler
     * @param target 目标对象
     * @param interceptor 拦截器
     * @return
     */
```

```

public static Object wrap(Object target, Interceptor interceptor) {
    // 获取用户自定义 Interceptor中@Signature注解的信息
    // getSignatureMap 负责处理@Signature 注解
    Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
    // 获取目标类型
    Class<?> type = target.getClass();
    // 获取目标类型 实现的所有的接口
    Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
    // 如果目标类型有实现的接口 就创建代理对象
    if (interfaces.length > 0) {
        return Proxy.newProxyInstance(
            type.getClassLoader(),
            interfaces,
            new Plugin(target, interceptor, signatureMap));
    }
    // 否则原封不动的返回目标对象
    return target;
}

/**
 * 代理对象方法被调用时执行的代码
 * @param proxy
 * @param method
 * @param args
 * @return
 * @throws Throwable
 */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        // 获取当前方法所在类或接口中, 可被当前Interceptor拦截的方法
        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
        if (methods != null && methods.contains(method)) {
            // 当前调用的方法需要被拦截 执行拦截操作
            return interceptor.intercept(new Invocation(target, method, args));
        }
        // 不需要拦截 则调用 目标对象中的方法
        return method.invoke(target, args);
    } catch (Exception e) {
        throw ExceptionUtil.unwrapThrowable(e);
    }
}

private static Map<Class<?>, Set<Method>> getSignatureMap(Interceptor
interceptor) {
    Intercepts interceptsAnnotation =
interceptor.getClass().getAnnotation(Intercepts.class);
    // issue #251
    if (interceptsAnnotation == null) {
        throw new PluginException("No @Intercepts annotation was found in
interceptor " + interceptor.getClass().getName());
    }
    Signature[] sigs = interceptsAnnotation.value();
    Map<Class<?>, Set<Method>> signatureMap = new HashMap<>();
    for (Signature sig : sigs) {
        Set<Method> methods = signatureMap.computeIfAbsent(sig.type(), k -> new
HashSet<>());

```



```

        try {
            Method method = sig.type().getMethod(sig.method(), sig.args());
            methods.add(method);
        } catch (NoSuchMethodException e) {
            throw new PluginException("Could not find method on " + sig.type() + "
named " + sig.method() + ". Cause: " + e, e);
        }
    }
    return signatureMap;
}

private static Class<?>[] getAllInterfaces(Class<?> type, Map<Class<?>,
Set<Method>> signatureMap) {
    Set<Class<?>> interfaces = new HashSet<>();
    while (type != null) {
        for (Class<?> c : type.getInterfaces()) {
            if (signatureMap.containsKey(c)) {
                interfaces.add(c);
            }
        }
        type = type.getSuperclass();
    }
    return interfaces.toArray(new Class<?>[interfaces.size()]);
}
}

```

2.2.2 StatementHandler

在获取StatementHandler的方法中

```

@Override
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds
rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        // 注意, 已经来到SQL处理的关键对象 StatementHandler >>
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
        // 获取一个 Statement对象
        stmt = prepareStatement(handler, ms.getStatementLog());
        // 执行查询
        return handler.query(stmt, resultHandler);
    } finally {
        // 用完就关闭
        closeStatement(stmt);
    }
}
}

```

在进入newStatementHandler方法

```

    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);
        // 植入插件逻辑（返回代理对象）
        statementHandler = (StatementHandler)
interceptorChain.pluginAll(statementHandler);
        return statementHandler;
    }

```

可以看到statementHandler的代理对象

2.2.3 ParameterHandler

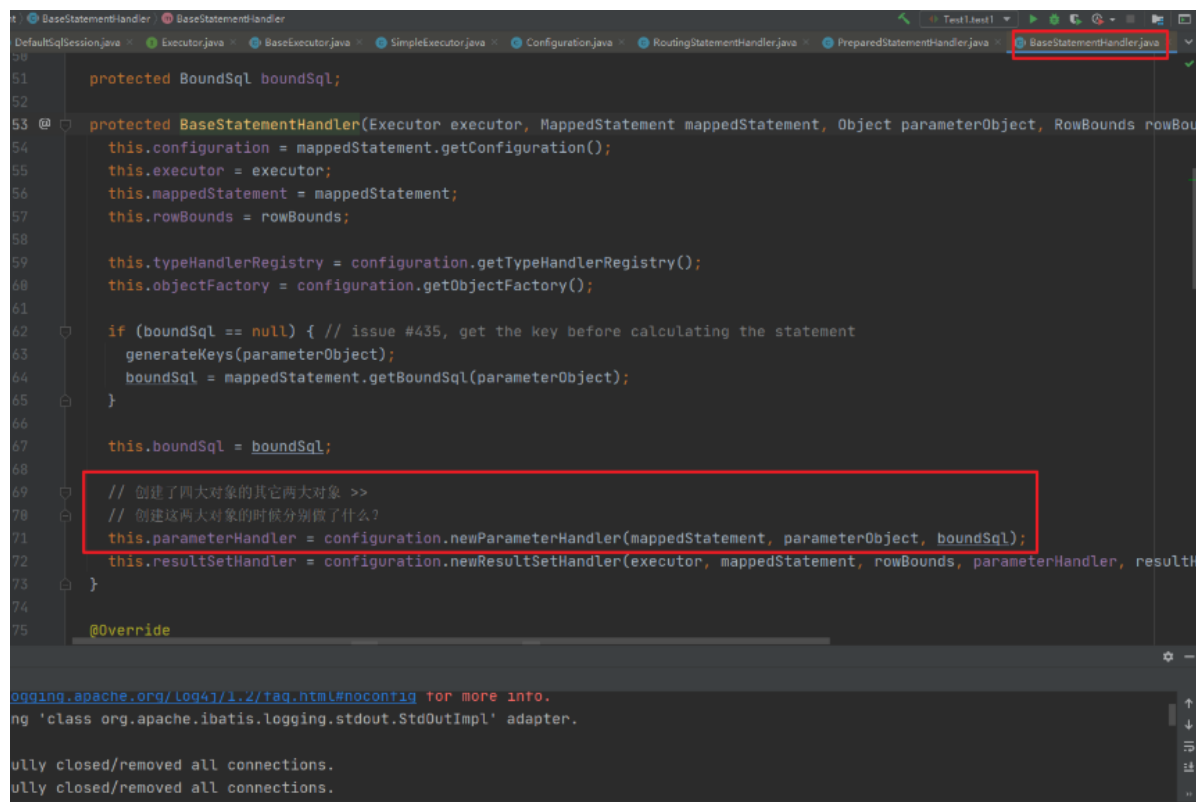
在上面步骤的RoutingStatementHandler方法中，我们来看看

```

    public RoutingStatementHandler(Executor executor, MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql)
{
        // StatementType 是怎么来的？ 增删改查标签中的 statementType="PREPARED", 默认值
PREPARED
        switch (ms.getStatementType()) {
            case STATEMENT:
                delegate = new SimpleStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
                break;
            case PREPARED:
                // 创建 StatementHandler 的时候做了什么？ >>
                delegate = new PreparedStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
                break;
            case CALLABLE:
                delegate = new CallableStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundSql);
                break;
            default:
                throw new ExecutorException("Unknown statement type: " +
ms.getStatementType());
        }
    }
}

```

然后我们随便选择一个分支进入，比如PreparedStatementHandler



在newParameterHandler的步骤我们可以发现代理对象的创建

```
public ParameterHandler newParameterHandler(MappedStatement mappedStatement,
Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
    // 植入插件逻辑（返回代理对象）
    parameterHandler = (ParameterHandler)
interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
}
```

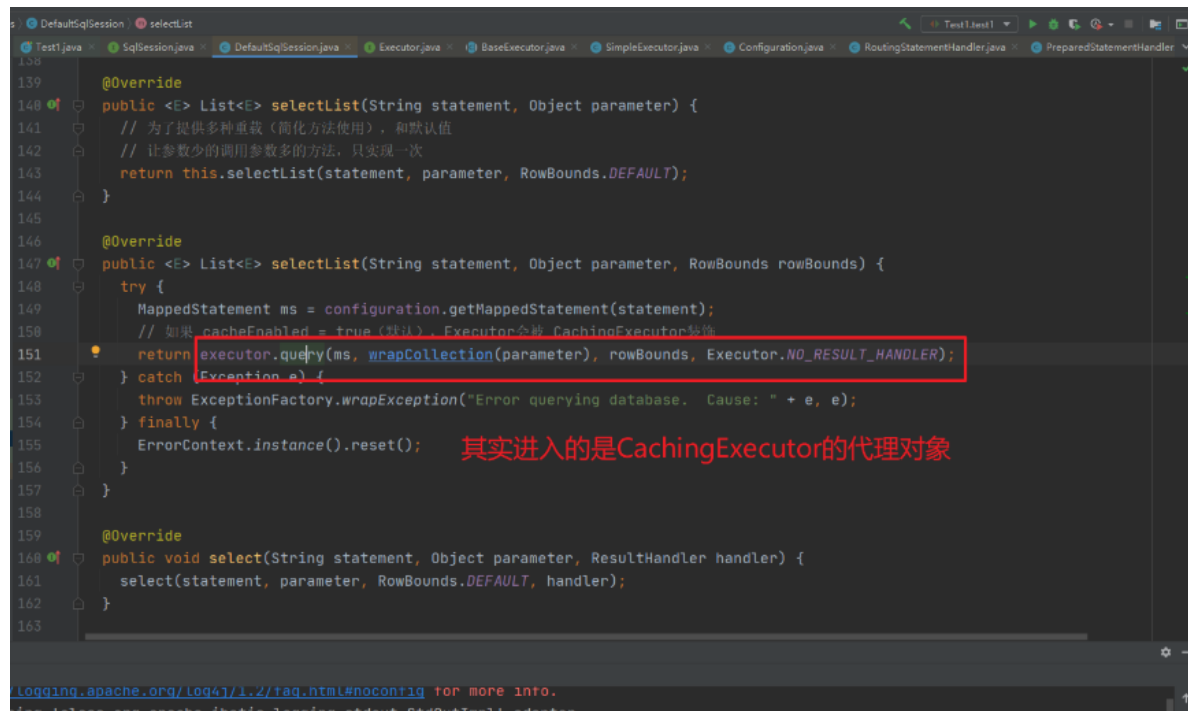
2.2.4 ResultSetHandler

在上面的newResultSetHandler()方法中，也可以看到ResultSetHandler的代理对象

```
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);
    // 植入插件逻辑（返回代理对象）
    resultSetHandler = (ResultSetHandler)
interceptorChain.pluginAll(resultSetHandler);
    return resultSetHandler;
}
```

2.3 执行流程

以Executor的query方法为例，当查询请求到来的时候，Executor的代理对象是如何处理拦截请求的呢？我们来看下。当请求到了executor.query方法的时候



然后会执行Plugin的invoke方法

```
/**
 * 代理对象方法被调用时执行的代码
 * @param proxy
 * @param method
 * @param args
 * @return
 * @throws Throwable
 */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        // 获取当前方法所在类或接口中，可被当前Interceptor拦截的方法
        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
        if (methods != null && methods.contains(method)) {
            // 当前调用的方法需要被拦截 执行拦截操作
            return interceptor.intercept(new Invocation(target, method, args));
        }
        // 不需要拦截 则调用 目标对象中的方法
        return method.invoke(target, args);
    } catch (Exception e) {
        throw ExceptionUtil.unwrapThrowable(e);
    }
}
```

然后进入interceptor.intercept 会进入我们自定义的 FirstInterceptor对象中

```

/**
 * 执行拦截逻辑的方法
 * @param invocation
 * @return
 * @throws Throwable
 */
@Override
public Object intercept(Invocation invocation) throws Throwable {
    System.out.println("FirtInterceptor 拦截之前 ....");
    Object obj = invocation.proceed(); // 执行目标方法
    System.out.println("FirtInterceptor 拦截之后 ....");
    return obj;
}

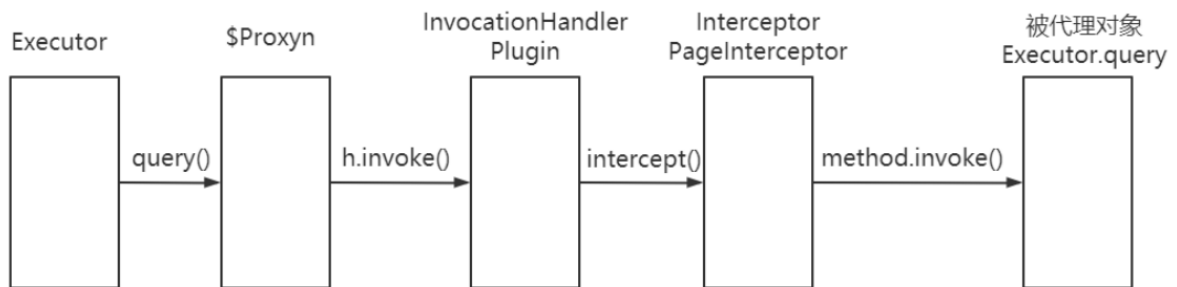
```

这个就是自定义的拦截器执行的完整流程，

2.4 多拦截器

如果我们有多个自定义的拦截器，那么他的执行流程是怎么样的呢？比如我们创建了两个 **Interceptor** 都是用来拦截 **Executor** 的 **query** 方法，一个是用来执行逻辑A 一个是用来执行逻辑B的。

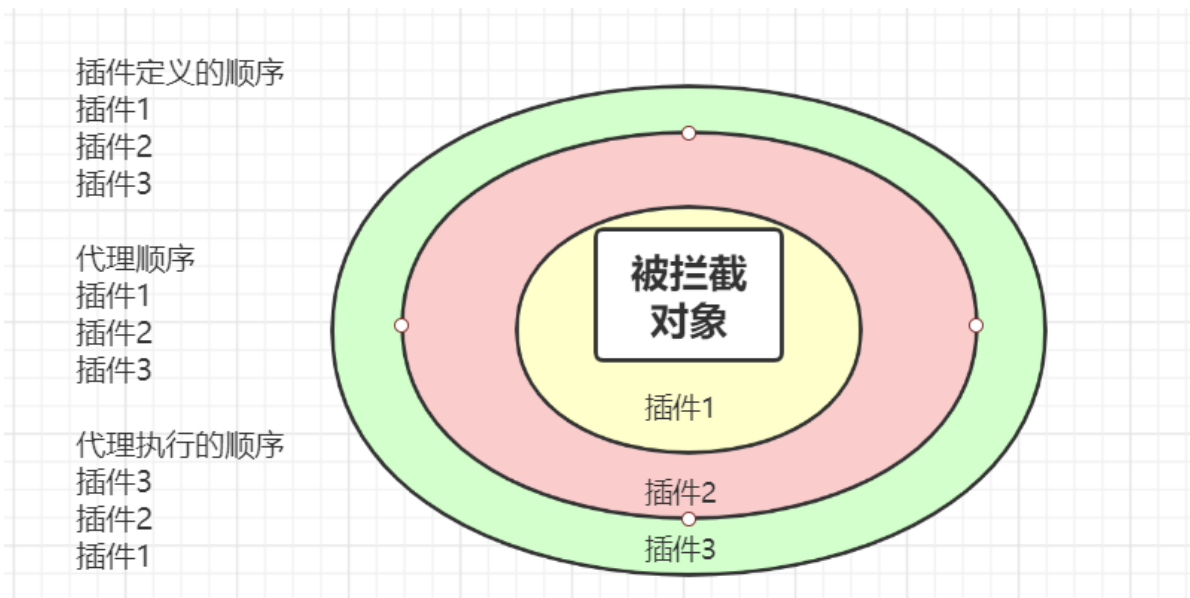
单个拦截器的执行流程



如果说对象被代理了多次，这里会继续调用下一个插件的逻辑，再走一次 **Plugin** 的 **invoke()** 方法。这里我们需要关注一下有多个插件的时候的运行顺序。

配置的顺序和执行的顺序是相反的。**InterceptorChain** 的 **List** 是按照插件从上往下的顺序解析、添加的。

而创建代理的时候也是按照 **list** 的顺序代理。执行的时候当然是从最后代理的对象开始。



这个我们可以通过实际的案例来得到验证，最后来总结下**Interceptor**的相关对象的作用

对象	作用
Interceptor	自定义插件需要实现接口，实现4个方法
InterceptChain	配置的插件解析后会保存在Configuration的InterceptChain中
Plugin	触发管理类，还可以用来创建代理对象
Invocation	对被代理类进行包装，可以调用proceed()调用到被拦截的方法

3. PageHelper分析

Mybatis的插件使用中分页插件**PageHelper**应该是我们使用到的比较多的插件应用。我们先来回顾下**PageHelper**的应用

3.1 PageHelper的应用

添加依赖

```
<dependency>  
  <groupId>com.github.pagehelper</groupId>  
  <artifactId>pagehelper</artifactId>  
  <version>4.1.6</version>  
</dependency>
```

在全局配置文件中注册

```
<!-- com.github.pagehelper为PageHelper类所在包名 -->  
<plugin interceptor="com.github.pagehelper.PageHelper">  
  <property name="dialect" value="mysql" />  
  <!-- 该参数默认为false -->  
  <!-- 设置为true时，会将RowBounds第一个参数offset当成pageNum页码使用 -->  
  <!-- 和startPage中的pageNum效果一样 -->
```

```

<property name="offsetAsPageNum" value="true" />
<!-- 该参数默认为false -->
<!-- 设置为true时,使用RowBounds分页会进行count查询 -->
<property name="rowBoundsWithCount" value="true" />
<!-- 设置为true时,如果pageSize=0或者RowBounds.limit = 0就会查询出全部的结果 -->
<!-- (相当于没有执行分页查询,但是返回结果仍然是Page类型) -->
<property name="pageSizeZero" value="true" />
<!-- 3.3.0版本可用 - 分页参数合理化,默认false禁用 -->
<!-- 启用合理化时,如果pageNum<1会查询第一页,如果pageNum>pages会查询最后一页 -->
<!-- 禁用合理化时,如果pageNum<1或pageNum>pages会返回空数据 -->
<property name="reasonable" value="false" />
<!-- 3.5.0版本可用 - 为了支持startPage(Object params)方法 -->
<!-- 增加了一个`params`参数来配置参数映射,用于从Map或ServletRequest中取值 -->
<!-- 可以配置pageNum,pageSize,count,pageSizeZero,reasonable,不配置映射的用默认值 -->
-->
<!-- 不理解该含义的前提下,不要随便复制该配置 -->
<property name="params" value="pageNum=start;pageSize=limit;" />
<!-- always总是返回PageInfo类型,check检查返回类型是否为PageInfo,none返回Page -->
<property name="returnPageInfo" value="check" />
</plugin>

```

然后就是分页查询操作

```

35 @Test
36 public void test1() throws Exception{
37     // 1. 获取配置文件
38     InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
39     // 2. 加载解析配置文件并获取SqlSessionFactory对象
40     SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
41     // 3. 根据SqlSessionFactory对象获取SqlSession对象
42     SqlSession sqlSession = factory.openSession();
43     // 分页设置
44     PageHelper.startPage(1, 5);
45     // 4. 通过SqlSession中提供的 API 方法来操作数据库
46     List<User> list = sqlSession.selectList( statement: "com.gupaoedu.mapper.UserMapper.selectUserList");
47     for (User user : list) {
48         System.out.println(user);
49     }
50     // 5. 关闭会话
51     sqlSession.close();

```

查看日志我们能够发现查询出来的只有5条记录

```

==> Preparing select * from t_user limit ?,?
==> Parameters: 0(Integer), 5(Integer)
<== Columns: id, user_name, real_name, password, age, d_id
<== Row: 1, zhangsan, 张三, 123456, 18, null
<== Row: 2, lisi, 李四, 11111, 19, null
<== Row: 3, wangwu, 王五, 111, 22, 1001
<== Row: 4, wangwu, 王五, 111, 22, 1001
<== Row: 5, wangwu, 王五, 111, 22, 1001
<== Total: 5
FirstInterceptor 拦截之后 ...
User(id=1, userName=zhangsan666, realName=张三, password=123456, age=18, dId=null)
User(id=2, userName=lisi, realName=李四, password=11111, age=19, dId=null)

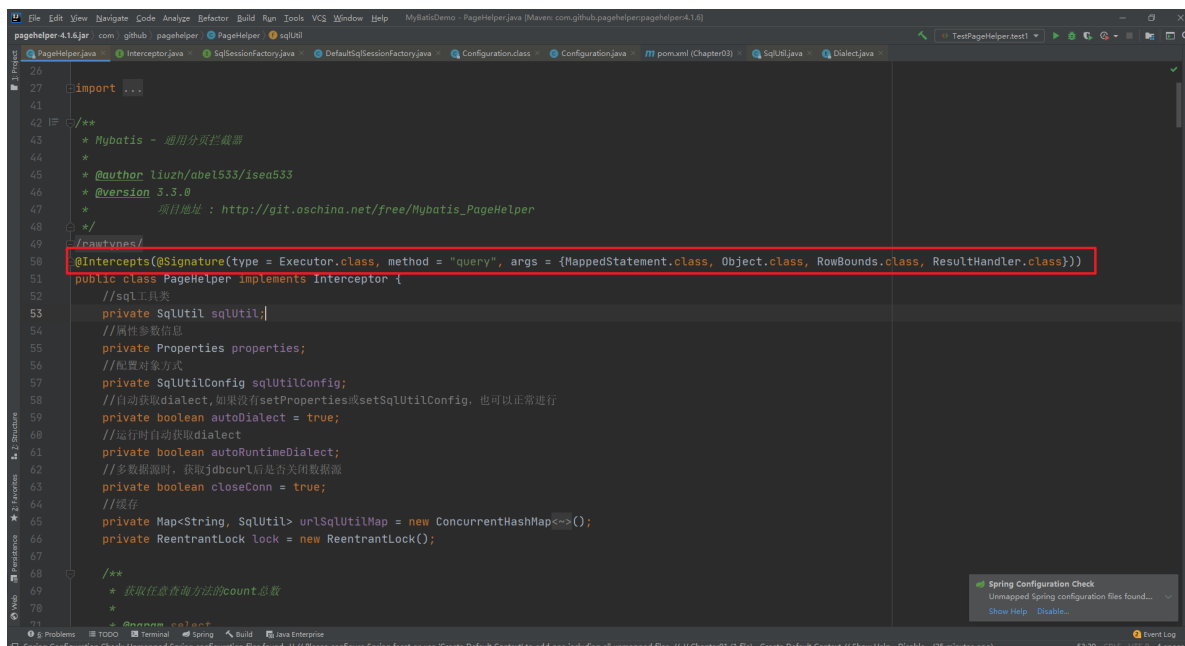
```

通过MyBatis的分页插件的使用,我们发现我们仅仅是在 执行操作之前设置了一句

`PageHelper.startPage(1,5);` 并没有做其他操作,也就是没有改变任何其他业务代码。这就是它的优点,那么我再来看下他的实现原理

3.2 实现原理剖析

在PageHelper中，肯定有提供Interceptor的实现类，通过源码我们可以发现是PageHelper，而且我们也可以看到在该方法头部添加的注解，声明了该拦截器拦截的是Executor的query方法




```

    * Mybatis拦截器方法
    *
    * @param invocation 拦截器入参
    * @return 返回执行结果
    * @throws Throwable 抛出异常
    */
    public Object intercept(Invocation invocation) throws Throwable {
        if (autoRuntimeDialect) {
            SqlUtil sqlUtil = getSqlUtil(invocation);
            return sqlUtil.processPage(invocation);
        } else {
            if (autoDialect) {
                initSqlUtil(invocation);
            }
            return sqlUtil.processPage(invocation);
        }
    }
}

```

intercept方法

```

/**
 * Mybatis拦截器方法
 *
 * @param invocation 拦截器入参
 * @return 返回执行结果
 * @throws Throwable 抛出异常
 */
    public Object intercept(Invocation invocation) throws Throwable {
        if (autoRuntimeDialect) { // 多数据源
            SqlUtil sqlUtil = getSqlUtil(invocation);
            return sqlUtil.processPage(invocation);
        } else { // 单数据源
            if (autoDialect) {
                initSqlUtil(invocation);
            }
            return sqlUtil.processPage(invocation);
        }
    }
}

```

在interceptor方法中首先会获取一个SqlUtils对象

SqlUtil：数据库类型专用sql工具类，一个数据库url对应一个SqlUtil实例，SqlUtil内有一个Parser对象，如果是mysql，它是MysqlParser，如果是oracle，它是OracleParser，这个Parser对象是SqlUtil不同实例的主要存在价值。执行count查询、设置Parser对象、执行分页查询、保存Page分页对象等功能，均由SqlUtil来完成。

initSqlUtil 方法

```

/**
 * 初始化sqlUtil
 *
 * @param invocation
 */
    public synchronized void initSqlUtil(Invocation invocation) {
        if (this.sqlUtil == null) {
            this.sqlUtil = getSqlUtil(invocation);
            if (!autoRuntimeDialect) {

```

```

        properties = null;
        sqlUtilConfig = null;
    }
    autoDialect = false;
}
}

```

getSqlUtil方法

```

/**
 * 根据datasource创建对应的sqlUtil
 *
 * @param invocation
 */
public SqlUtil getSqlUtil(Invocation invocation) {
    MappedStatement ms = (MappedStatement) invocation.getArgs()[0];
    //改为对datasource做缓存
    DataSource dataSource =
ms.getConfiguration().getEnvironment().getDataSource();
    String url = getUrl(dataSource);
    if (urlSqlUtilMap.containsKey(url)) {
        return urlSqlUtilMap.get(url);
    }
    try {
        lock.lock();
        if (urlSqlUtilMap.containsKey(url)) {
            return urlSqlUtilMap.get(url);
        }
        if (StringUtil.isEmpty(url)) {
            throw new RuntimeException("无法自动获取jdbcurl，请在分页插件中配置
dialect参数!");
        }
        String dialect = Dialect.fromJdbcUrl(url);
        if (dialect == null) {
            throw new RuntimeException("无法自动获取数据库类型，请通过dialect参数指
定!");
        }
        // 创建SqlUtil
        sqlUtil sqlUtil = new SqlUtil(dialect);
        if (this.properties != null) {
            sqlUtil.setProperties(properties);
        } else if (this.sqlUtilConfig != null) {
            sqlUtil.setSqlUtilConfig(this.sqlUtilConfig);
        }
        urlSqlUtilMap.put(url, sqlUtil);
        return sqlUtil;
    } finally {
        lock.unlock();
    }
}

```

查看SqlUtil的构造方法

```

/**
 * 构造方法
 *

```

```

    * @param strDialect
    */
    public SqlUtil(String strDialect) {
        if (strDialect == null || "".equals(strDialect)) {
            throw new IllegalArgumentException("Mybatis分页插件无法获取dialect参
数!");
        }
        Exception exception = null;
        try {
            Dialect dialect = Dialect.of(strDialect);
            // 根据方言创建对应的解析器
            parser = AbstractParser.newParser(dialect);
        } catch (Exception e) {
            exception = e;
            //异常的时候尝试反射，允许自己写实现类传递进来
            try {
                Class<?> parserClass = Class.forName(strDialect);
                if (Parser.class.isAssignableFrom(parserClass)) {
                    parser = (Parser) parserClass.newInstance();
                }
            } catch (ClassNotFoundException ex) {
                exception = ex;
            } catch (InstantiationException ex) {
                exception = ex;
            } catch (IllegalAccessException ex) {
                exception = ex;
            }
        }
        if (parser == null) {
            throw new RuntimeException(exception);
        }
    }
}

```

创建的解析器

```

public static Parser newParser(Dialect dialect) {
    Parser parser = null;
    switch (dialect) {
        case mysql:
        case mariadb:
        case sqlite:
            parser = new MysqlParser();
            break;
        case oracle:
            parser = new OracleParser();
            break;
        case hsqldb:
            parser = new HsqldbParser();
            break;
        case sqlserver:
            parser = new SqlServerParser();
            break;
        case sqlserver2012:
            parser = new SqlServer2012Dialect();
            break;
        case db2:
            parser = new Db2Parser();

```

```

        break;
    case postgresql:
        parser = new PostgreSQLParser();
        break;
    case informix:
        parser = new InformixParser();
        break;
    case h2:
        parser = new H2Parser();
        break;
    default:
        throw new RuntimeException("分页插件" + dialect + "方言错误!");
    }
    return parser;
}

```

我们可以看到不同的数据库方言，创建了对应的解析器。

然后我们再回到前面的interceptor方法中继续，查看sqlUtil.processPage(invocation);方法

```

private Object _processPage(Invocation invocation) throws Throwable {
    final Object[] args = invocation.getArgs();
    Page page = null;
    //支持方法参数时，会先尝试获取Page
    if (supportMethodsArguments) {
        page = getPage(args); // 有通过ThreadLocal来获取分页数据信息
    }
    //分页信息
    RowBounds rowBounds = (RowBounds) args[2];
    //支持方法参数时，如果page == null就说明没有分页条件，不需要分页查询
    if ((supportMethodsArguments && page == null)
        //当不支持分页参数时，判断LocalPage和RowBounds判断是否需要分页
        || (!supportMethodsArguments && SqlUtil.getLocalPage() == null
        && rowBounds == RowBounds.DEFAULT)) {
        return invocation.proceed();
    } else {
        //不支持分页参数时，page==null，这里需要获取
        if (!supportMethodsArguments && page == null) {
            page = getPage(args);
        }
        return doProcessPage(invocation, page, args);
    }
}

```

doProcessPage进入该方法

```

private Page doProcessPage(Invocation invocation, Page page, Object[] args)
throws Throwable {
    //保存RowBounds状态
    RowBounds rowBounds = (RowBounds) args[2];
    //获取原始的ms
    MappedStatement ms = (MappedStatement) args[0];
    //判断并处理为PageSqlSource
    if (!isPageSqlSource(ms)) {
        processMappedStatement(ms);
    }
    //设置当前的parser，后面每次使用前都会set，ThreadLocal的值不会产生不良影响
}

```

```

((PageSqlSource)ms.getSqlSource()).setParser(parser);
try {
    //忽略RowBounds-否则会进行Mybatis自带的内存分页
    args[2] = RowBounds.DEFAULT;
    //如果只进行排序 或 pageSizeZero的判断
    if (isQueryOnly(page)) {
        return doQueryOnly(page, invocation);
    }

    //简单的通过total的值来判断是否进行count查询
    if (page.isCount()) {
        page.setCountSignal(Boolean.TRUE);
        //替换MS
        args[0] = msCountMap.get(ms.getId());
        //查询总数
        Object result = invocation.proceed();
        //还原ms
        args[0] = ms;
        //设置总数
        page.setTotal((Integer) ((List) result).get(0));
        if (page.getTotal() == 0) {
            return page;
        }
    } else {
        page.setTotal(-1);
    }
    //pageSize>0的时候执行分页查询, pageSize<=0的时候不执行相当于可能只返回了一个
count
    if (page.getPageSize() > 0 &&
        ((rowBounds == RowBounds.DEFAULT && page.getPageNum() > 0)
         || rowBounds != RowBounds.DEFAULT)) {
        //将参数中的MappedStatement替换为新的qs
        page.setCountSignal(null);
        BoundSql boundSql = ms.getBoundSql(args[1]);
        // 在 invocation中绑定 分页的数据
        args[1] = parser.setPageParameter(ms, args[1], boundSql, page);
        page.setCountSignal(Boolean.FALSE);
        //执行分页查询
        Object result = invocation.proceed();
        //得到处理结果
        page.addAll((List) result);
    }
} finally {
    ((PageSqlSource)ms.getSqlSource()).removeParser();
}

//返回结果
return page;
}

```

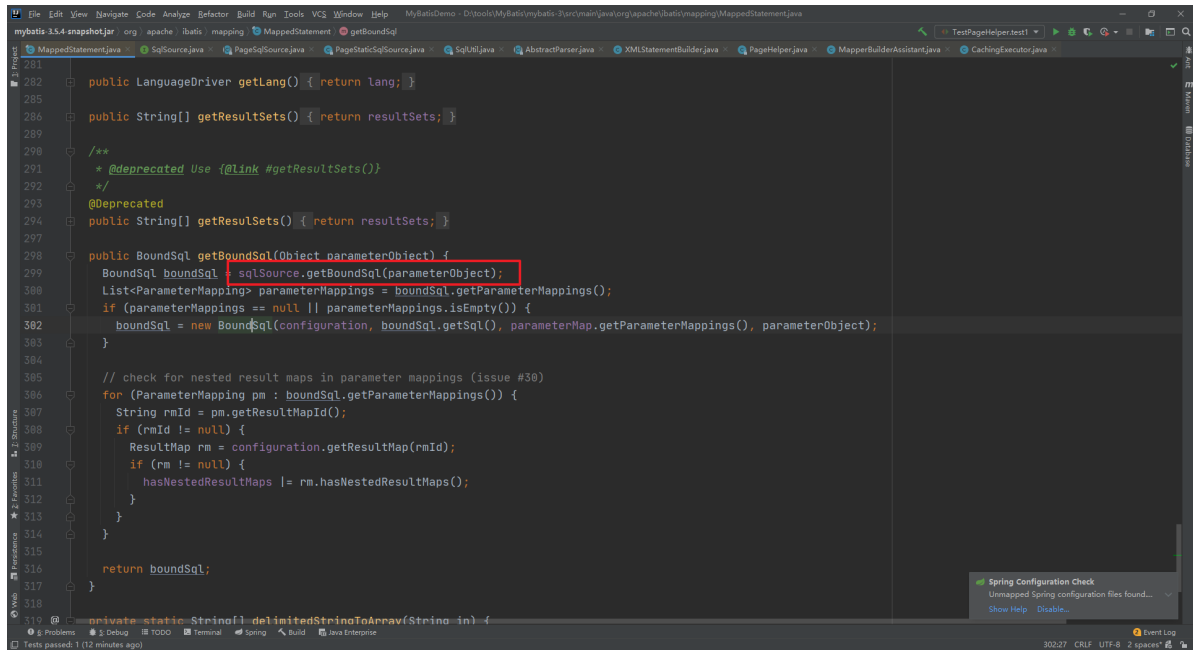
invocation.proceed();方法会进入 CachingExecutor中的query方法。

```

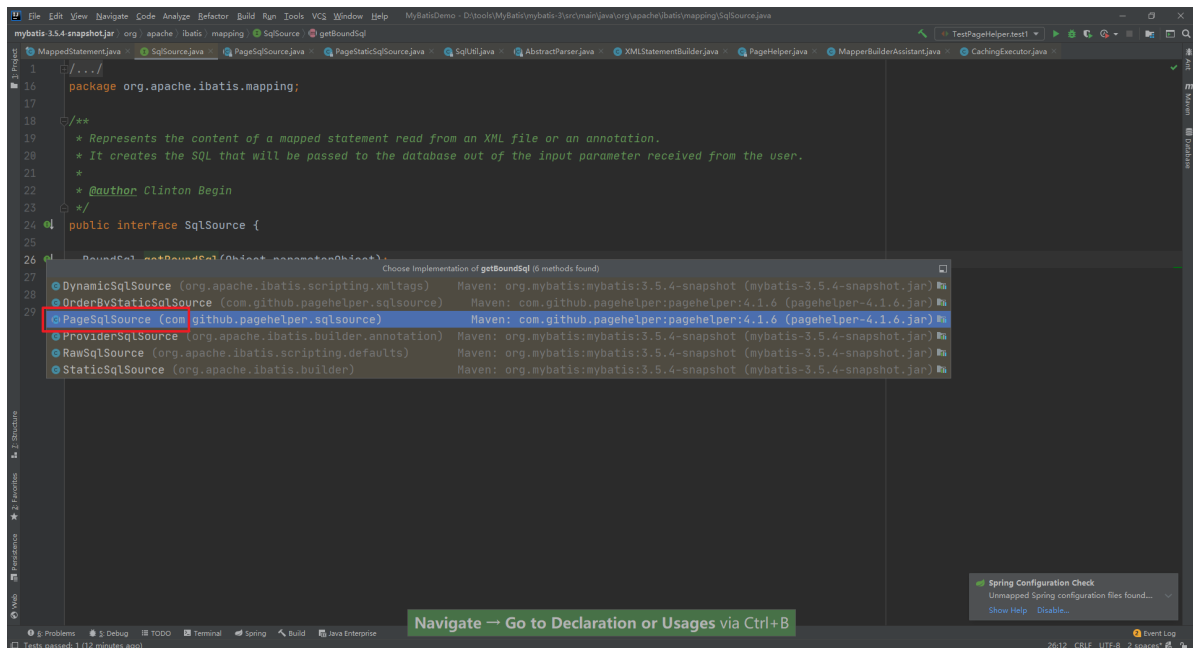
@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds
rowBounds, ResultHandler resultHandler) throws SQLException {
    // 获取SQL
    BoundSql boundSql = ms.getBoundSql(parameterObject);
    // 创建CacheKey: 什么样的SQL是同一条SQL? >>
    CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql);
    return query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}

```

ms.getBoundSql方法中会完成分页SQL的绑定



进入 PageSqlSource 的getBoundSql方法中



```

/**
 * 获取BoundSql
 *
 * @param parameterObject
 * @return
 */

```

```

@Override
public BoundSql getBoundSql(Object parameterObject) {
    Boolean count = getCount();
    if (count == null) {
        return getDefaultBoundSql(parameterObject);
    } else if (count) {
        return getCountBoundSql(parameterObject);
    } else {
        return getPageBoundSql(parameterObject);
    }
}

```

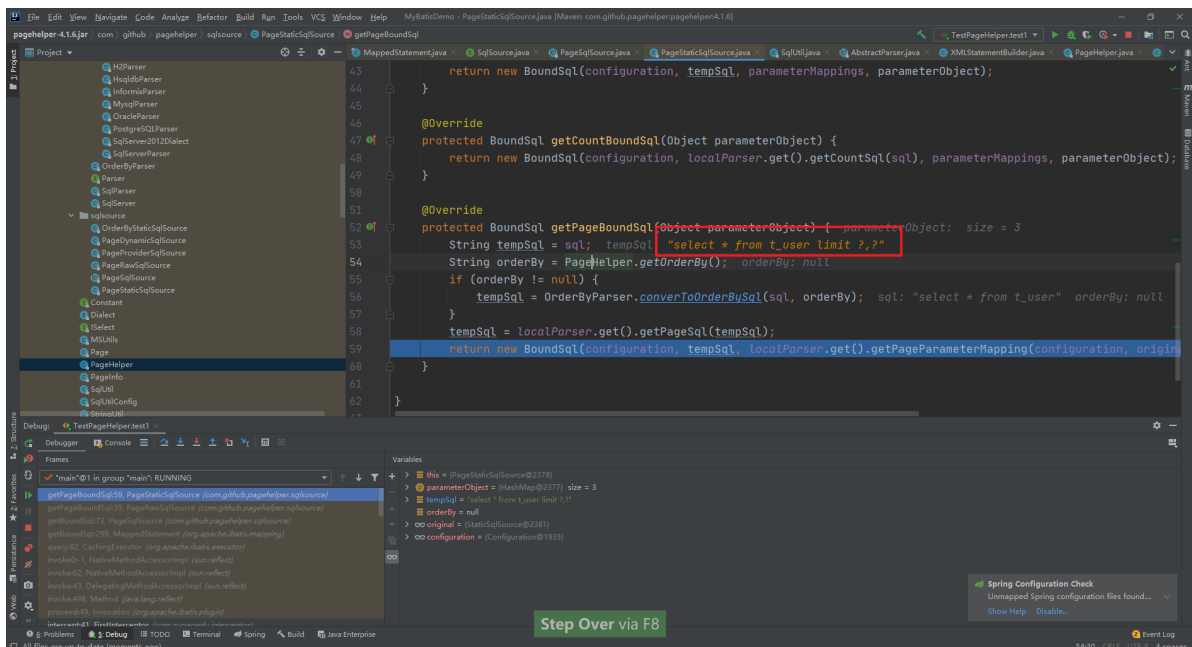
然后进入getPageBoundSql获取分页的SQL语句，在这个方法中大家也可以发现查询总的记录数的SQL生成

```

@Override
protected BoundSql getPageBoundSql(Object parameterObject) {
    String tempSql = sql;
    String orderBy = PageHelper.getOrderBy();
    if (orderBy != null) {
        tempSql = OrderByParser.convertToOrderBySql(sql, orderBy);
    }
    tempSql = localParser.get().getPageSql(tempSql);
    return new BoundSql(configuration, tempSql,
        localParser.get().getPageParameterMapping(configuration,
            original.getBoundSql(parameterObject)), parameterObject);
}

```

最终在这个方法中生成了对应数据库的分页语句



4.应用场景分析

作用	描述	实现方式
水平分表	一张费用表按月度拆分为12张表。fee_202001-202012。当查询条件出现月度（ tran_month ）时，把select语句中的逻辑表名修改为对应的月份表。	对query update方法进行拦截在接口上添加注解，通过反射获取接口注解，根据注解上配置的参数进行分表，修改原SQL，例如id取模，按月分表
数据脱敏	手机号和身份证在数据库完整存储。但是返回给用户，屏蔽手机号的中间四位。屏蔽身份证号中的出生日期。	query——对结果集脱敏
菜单权限控制	不同的用户登录，查询菜单权限表时获得不同的结果，在前端展示不同的菜单	对query方法进行拦截在方法上添加注解，根据权限配置，以及用户登录信息，在SQL上加上权限过滤条件
黑白名单	有些SQL语句在生产环境中是不允许执行的，比如like %%	对Executor的update和query方法进行拦截，将拦截的SQL语句和黑白名单进行比较，控制SQL语句的执行
全局唯一ID	在高并发的环境下传统的生成ID的方式不太适用，这时我们就需要考虑其他方式了	创建插件拦截Executor的insert方法，通过UUID或者雪花算法来生成ID，并修改SQL中的插入信息