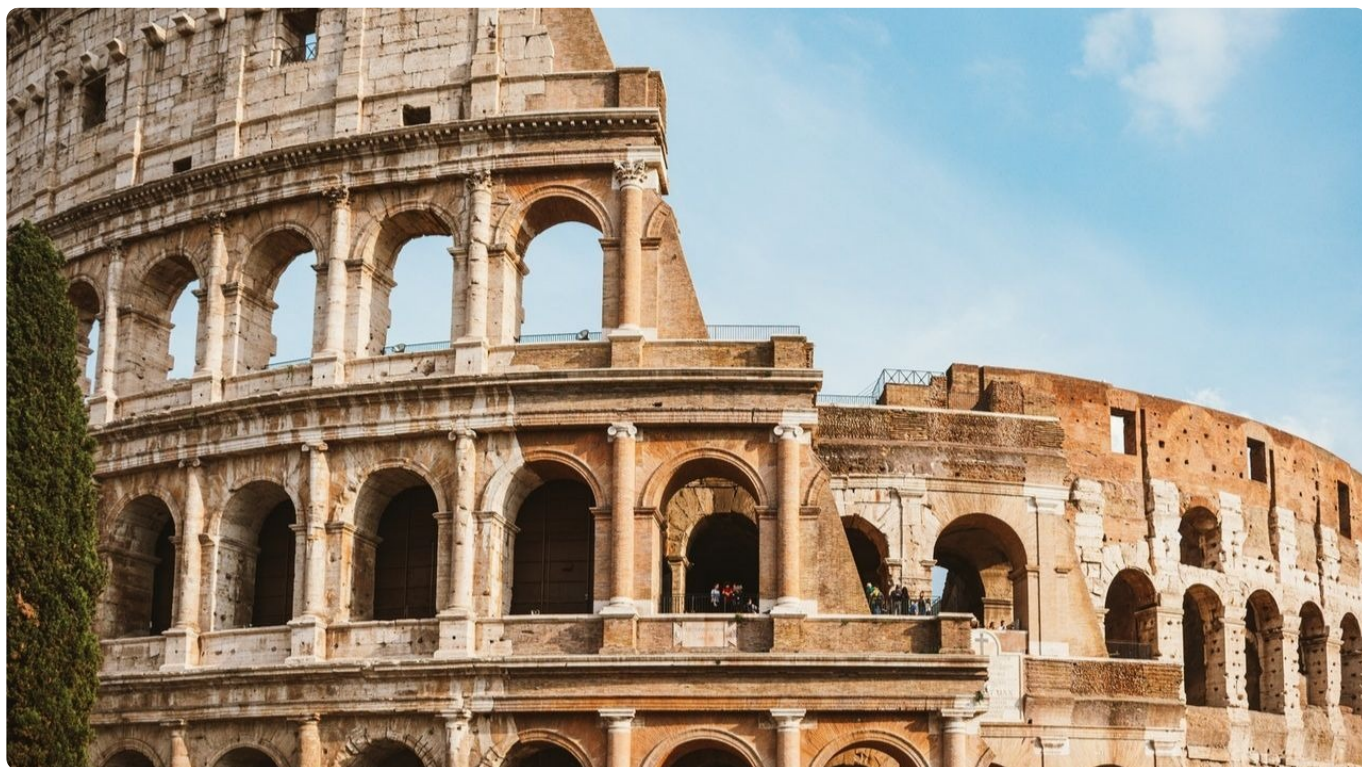


88 | 开源实战五（中）：如何利用职责链与代理模式实现MyBatis Plugin?

2020-05-25 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 10:57 大小 10.04M



上节课，我们对 MyBatis 框架做了简单的背景介绍，并且通过对比各种 ORM 框架，学习了代码的易用性、性能、灵活性之间的关系。一般来讲，框架提供的高级功能越多，那性能损耗就会越大；框架用起来越简单，提供越简化的使用方式，那灵活性也就越低。

接下来的两节课，我们再学习一下 MyBatis 用到一些经典设计模式。其中，今天，我们主要讲解 MyBatis Plugin。尽管名字叫 Plugin（插件），但它实际上跟之前讲到的 Servlet Filter（过滤器）、Spring Interceptor（拦截器）类似，设计的初衷都是为了框架的，☆性，用到的主要设计模式都是职责链模式。

不过，相对于 Servlet Filter 和 Spring Interceptor，MyBatis Plugin 中职责链模式的代码实现稍微有点复杂。它是借助动态代理模式来实现的职责链。今天我就带你看下，如何利用这两个模式实现 MyBatis Plugin。


话不多说，让我们正式开始今天的学习吧！

MyBatis Plugin 功能介绍

实际上，MyBatis Plugin 跟 Servlet Filter、Spring Interceptor 的功能是类似的，都是在不需要修改原有流程代码的情况下，拦截某些方法调用，在拦截的方法调用的前后，执行一些额外的代码逻辑。它们的唯一区别在于拦截的位置是不同的。Servlet Filter 主要拦截 Servlet 请求，Spring Interceptor 主要拦截 Spring 管理的 Bean 方法（比如 Controller 类的方法等），而 MyBatis Plugin 主要拦截的是 MyBatis 在执行 SQL 的过程中涉及的一些方法。

MyBatis Plugin 使用起来比较简单，我们通过一个例子来快速看下。

假设我们需要统计应用中每个 SQL 的执行耗时，如果使用 MyBatis Plugin 来实现的话，我们只需要定义一个 SqlCostTimeInterceptor 类，让它实现 MyBatis 的 Interceptor 接口，并且，在 MyBatis 的全局配置文件中，简单声明一下这个插件就可以了。具体的代码和配置如下所示：

 复制代码

```
1 @Intercepts({
2     @Signature(type = StatementHandler.class, method = "query", args = {St
3     @Signature(type = StatementHandler.class, method = "update", args = {S
4     @Signature(type = StatementHandler.class, method = "batch", args = {St
5 public class SqlCostTimeInterceptor implements Interceptor {
6     private static Logger logger = LoggerFactory.getLogger(SqlCostTimeIntercepto
7
8     @Override
9     public Object intercept(Invocation invocation) throws Throwable {
10         Object target = invocation.getTarget();
11         long startTime = System.currentTimeMillis();
12         StatementHandler statementHandler = (StatementHandler) target;
13         try {
14             return invocation.proceed();
15         } finally {
16             long costTime = System.currentTimeMillis() - startTime;
17             BoundSql boundSql = statementHandler.getBoundSql();
18             String sql = boundSql.getSql();
```

```
19     logger.info("执行 SQL: [ {} ] 执行耗时[ {} ms]", sql, costTime);
20 }
21 }
22
23 @Override
24 public Object plugin(Object target) {
25     return Plugin.wrap(target, this);
26 }
27
28 @Override
29 public void setProperties(Properties properties) {
30     System.out.println("插件配置的信息: "+properties);
31 }
32 }
33
34 <!-- MyBatis全局配置文件: mybatis-config.xml -->
35 <plugins>
36     <plugin interceptor="com.xzg.cd.a88.SqlCostTimeInterceptor">
37         <property name="someProperty" value="100"/>
38     </plugin>
39 </plugins>
```

因为待会我会详细地介绍 MyBatis Plugin 的底层实现原理，所以，这里暂时不对上面的代码做详细地解释。现在，我们只重点看下 @Intercepts 注解这一部分。

我们知道，不管是拦截器、过滤器还是插件，都需要明确地标明拦截的目标方法。

@Intercepts 注解实际上就是起了这个作用。其中，@Intercepts 注解又可以嵌套 @Signature 注解。一个 @Signature 注解标明一个要拦截的目标方法。如果要拦截多个方法，我们可以像例子中那样，编写多条 @Signature 注解。

@Signature 注解包含三个元素：type、method、args。其中，type 指明要拦截的类、method 指明方法名、args 指明方法的参数列表。通过指定这三个元素，我们就能完全确定一个要拦截的方法。

默认情况下，MyBatis Plugin 允许拦截的方法有下面这样几个：

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



为什么默认允许拦截的是这样几个类的方法呢？

MyBatis 底层是通过 Executor 类来执行 SQL 的。Executor 类会创建 StatementHandler、ParameterHandler、ResultSetHandler 三个对象，并且，首先使用 ParameterHandler 设置 SQL 中的占位符参数，然后使用 StatementHandler 执行 SQL 语句，最后使用 ResultSetHandler 封装执行结果。所以，我们只需要拦截 Executor、ParameterHandler、ResultSetHandler、StatementHandler 这几个类的方法，基本上就能满足我们对整个 SQL 执行流程的拦截了。

实际上，除了统计 SQL 的执行耗时，利用 MyBatis Plugin，我们还可以做很多事情，比如分库分表、自动分页、数据脱敏、加密解密等等。如果感兴趣的话，你可以自己实现一下。

MyBatis Plugin 的设计与实现


刚刚我们简单介绍了 MyBatis Plugin 是如何使用的。现在，我们再剖析一下源码，看看如此简洁的使用方式，底层是如何实现的，隐藏了哪些复杂的设计。

相对于 Servlet Filter、Spring Interceptor 中职责链模式的代码实现，MyBatis Plugin 的代码实现还是蛮有技巧的，因为它是借助动态代理来实现职责链的。

在 [第 62 节](#)和 [第 63 节](#)中，我们讲到，职责链模式的实现一般包含处理器（Handler）和处理器链（HandlerChain）两部分。这两个部分对应到 Servlet Filter 的源码就是 Filter 和 FilterChain，对应到 Spring Interceptor 的源码就是 HandlerInterceptor 和 HandlerExecutionChain，对应到 MyBatis Plugin 的源码就是 Interceptor 和

InterceptorChain。除此之外，MyBatis Plugin 还包含另外一个非常重要的类：Plugin。它用来生成被拦截对象的动态代理。

集成了 MyBatis 的应用在启动的时候，MyBatis 框架会读取全局配置文件（前面例子中的 mybatis-config.xml 文件），解析出 Interceptor（也就是例子中的 SqlCostTimeInterceptor），并且将它注入到 Configuration 类的 InterceptorChain 对象中。这部分逻辑对应到源码如下所示：

 复制代码

```
1 public class XMLConfigBuilder extends BaseBuilder {
2     //解析配置
3     private void parseConfiguration(XNode root) {
4         try {
5             //省略部分代码...
6             pluginElement(root.evalNode("plugins")); //解析插件
7         } catch (Exception e) {
8             throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e);
9         }
10    }
11
12    //解析插件
13    private void pluginElement(XNode parent) throws Exception {
14        if (parent != null) {
15            for (XNode child : parent.getChildren()) {
16                String interceptor = child.getStringAttribute("interceptor");
17                Properties properties = child.getChildrenAsProperties();
18                //创建Interceptor类对象
19                Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor);
20                //调用Interceptor上的setProperties()方法设置properties
21                interceptorInstance.setProperties(properties);
22                //下面这行代码会调用InterceptorChain.addInterceptor()方法
23                configuration.addInterceptor(interceptorInstance);
24            }
25        }
26    }
27 }
28
29 // Configuration类的addInterceptor()方法的代码如下所示
30 public void addInterceptor(Interceptor interceptor) {
31     interceptorChain.addInterceptor(interceptor);
32 }
```

我们再来看 Interceptor 和 InterceptorChain 这两个类的代码，如下所示。Interceptor 的 setProperties() 方法就是一个单纯的 setter 方法，主要是为了方便通过配置文件配置

Interceptor 的一些属性值，没有其他作用。Interceptor 类中 intercept() 和 plugin() 函数，以及 InterceptorChain 类中的 pluginAll() 函数，是最核心的三个函数，我们待会再详细解释。


 复制代码

```
1 public class Invocation {
2     private final Object target;
3     private final Method method;
4     private final Object[] args;
5     // 省略构造函数和getter方法...
6     public Object proceed() throws InvocationTargetException, IllegalAccessException {
7         return method.invoke(target, args);
8     }
9 }
10 public interface Interceptor {
11     Object intercept(Invocation invocation) throws Throwable;
12     Object plugin(Object target);
13     void setProperties(Properties properties);
14 }
15
16 public class InterceptorChain {
17     private final List<Interceptor> interceptors = new ArrayList<Interceptor>();
18
19     public Object pluginAll(Object target) {
20         for (Interceptor interceptor : interceptors) {
21             target = interceptor.plugin(target);
22         }
23         return target;
24     }
25
26     public void addInterceptor(Interceptor interceptor) {
27         interceptors.add(interceptor);
28     }
29
30     public List<Interceptor> getInterceptors() {
31         return Collections.unmodifiableList(interceptors);
32     }
33 }
```

解析完配置文件之后，所有的 Interceptor 都加载到了 InterceptorChain 中。接下来，我们再来看下，这些拦截器是在什么时候被触发执行的？又是如何被触发执行的呢？

前面我们提到，在执行 SQL 的过程中，MyBatis 会创建 Executor、StatementHandler、ParameterHandler、ResultSetHandler 这几个类的对象，对应的创建代码在


Configuration 类中，如下所示：

 复制代码

```
1 public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
2     executorType = executorType == null ? defaultExecutorType : executorType;
3     executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
4     Executor executor;
5     if (ExecutorType.BATCH == executorType) {
6         executor = new BatchExecutor(this, transaction);
7     } else if (ExecutorType.REUSE == executorType) {
8         executor = new ReuseExecutor(this, transaction);
9     } else {
10        executor = new SimpleExecutor(this, transaction);
11    }
12    if (cacheEnabled) {
13        executor = new CachingExecutor(executor);
14    }
15    executor = (Executor) interceptorChain.pluginAll(executor);
16    return executor;
17 }
18
19 public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object
20     ParameterHandler parameterHandler = mappedStatement.getLang().createParameter
21     parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHan
22     return parameterHandler;
23 }
24
25 public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement
26     ResultHandler resultHandler, BoundSql boundSql) {
27     ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, map
28     resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHan
29     return resultSetHandler;
30 }
31
32 public StatementHandler newStatementHandler(Executor executor, MappedStatement
33     StatementHandler statementHandler = new RoutingStatementHandler(executor, map
34     statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHan
35     return statementHandler;
36 }
```

从上面的代码中，我们可以发现，这几个类对象的创建过程都调用了 `InteceptorChain` 的 `pluginAll()` 方法。这个方法的代码前面已经给出了。你可以回过头去再看一眼。它的代码实现很简单，嵌套调用 `InterceptorChain` 上每个 `Interceptor` 的 `plugin()` 方法。`plugin()` 是一个接口方法（不包含实现代码），需要由用户给出具体的实现代码。在之前的例子中，

SQLTimeCostInterceptor 的 plugin() 方法通过直接调用 Plugin 的 wrap() 方法来实现。
wrap() 方法的代码实现如下所示:

 复制代码

```
1 // 借助Java InvocationHandler实现的动态代理模式
2 public class Plugin implements InvocationHandler {
3     private final Object target;
4     private final Interceptor interceptor;
5     private final Map<Class<?>, Set<Method>> signatureMap;
6
7     private Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Metl
8         this.target = target;
9         this.interceptor = interceptor;
10        this.signatureMap = signatureMap;
11    }
12
13    // wrap()静态方法, 用来生成target的动态代理,
14    // 动态代理对象=target对象+interceptor对象。
15    public static Object wrap(Object target, Interceptor interceptor) {
16        Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
17        Class<?> type = target.getClass();
18        Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
19        if (interfaces.length > 0) {
20            return Proxy.newProxyInstance(
21                type.getClassLoader(),
22                interfaces,
23                new Plugin(target, interceptor, signatureMap));
24        }
25        return target;
26    }
27
28    // 调用target上的f()方法, 会触发执行下面这个方法。
29    // 这个方法包含: 执行interceptor的intecept()方法 + 执行target上f()方法。
30    @Override
31    public Object invoke(Object proxy, Method method, Object[] args) throws Throi
32        try {
33            Set<Method> methods = signatureMap.get(method.getDeclaringClass());
34            if (methods != null && methods.contains(method)) {
35                return interceptor.intercept(new Invocation(target, method, args));
36            }
37            return method.invoke(target, args);
38        } catch (Exception e) {
39            throw ExceptionUtil.unwrapThrowable(e);
40        }
41    }
42
43    private static Map<Class<?>, Set<Method>> getSignatureMap(Interceptor interce
44        Intercepts interceptsAnnotation = interceptor.getClass().getAnnotation(Int
45        // issue #251
46        if (interceptsAnnotation == null) {
```



```

47     throw new PluginException("No @Intercepts annotation was found in interce
48 }
49 Signature[] sigs = interceptsAnnotation.value();
50 Map<Class<?>, Set<Method>> signatureMap = new HashMap<Class<?>, Set<Method>
51 for (Signature sig : sigs) {
52     Set<Method> methods = signatureMap.get(sig.type());
53     if (methods == null) {
54         methods = new HashSet<Method>();
55         signatureMap.put(sig.type(), methods);
56     }
57     try {
58         Method method = sig.type().getMethod(sig.method(), sig.args());
59         methods.add(method);
60     } catch (NoSuchMethodException e) {
61         throw new PluginException("Could not find method on " + sig.type() + "
62     }
63 }
64 return signatureMap;
65 }
66
67 private static Class<?>[] getAllInterfaces(Class<?> type, Map<Class<?>, Set<I
68     Set<Class<?>> interfaces = new HashSet<Class<?>>();
69     while (type != null) {
70         for (Class<?> c : type.getInterfaces()) {
71             if (signatureMap.containsKey(c)) {
72                 interfaces.add(c);
73             }
74         }
75         type = type.getSuperclass();
76     }
77     return interfaces.toArray(new Class<?>[interfaces.size()]);
78 }
79 }

```

实际上，Plugin 是借助 Java InvocationHandler 实现的动态代理类。用来代理给 target 对象添加 Interceptor 功能。其中，要代理的 target 对象就是 Executor、StatementHandler、ParameterHandler、ResultSetHandler 这四个类的对象。wrap() 静态方法是一个工具函数，用来生成 target 对象的动态代理对象。

当然，只有 interceptor 与 target 互相匹配的时候，wrap() 方法才会返回代理对象，否则就返回 target 对象本身。怎么才算是匹配呢？那就是 interceptor 通过 @Signature 注解要拦截的类包含 target 对象，具体可以参看 wrap() 函数的代码实现（上面一段代码中的第 16~19 行）。

MyBatis 中的职责链模式的实现方式比较特殊。它对同一个目标对象嵌套多次代理（也就是 InteceptorChain 中的 pluginAll() 函数要执行的任务）。每个代理对象（Plugin 对象）代理一个拦截器（Interceptor 对象）功能。为了方便你查看，我将 pluginAll() 函数的代码又拷贝到了下面。

 复制代码

```
1 public Object pluginAll(Object target) {
2     // 嵌套代理
3     for (Interceptor interceptor : interceptors) {
4         target = interceptor.plugin(target);
5         // 上面这行代码等于下面这行代码，target(代理对象)=target(目标对象)+interceptor(拦截
6         // target = Plugin.wrap(target, interceptor);
7     }
8     return target;
9 }
10
11 // MyBatis像下面这样创建target(Executor、StatementHandler、ParameterHandler、Result
12 Object target = interceptorChain.pluginAll(target);
```

当执行 Executor、StatementHandler、ParameterHandler、ResultSetHandler 这四个类上的某个方法的时候，MyBatis 会嵌套执行每层代理对象（Plugin 对象）上的 invoke() 方法。而 invoke() 方法会先执行代理对象中的 interceptor 的 intecept() 函数，然后再执行被代理对象上的方法。就这样，一层一层地把代理对象上的 intercept() 函数执行完之后，MyBatis 才最终执行那 4 个原始类对象上的方法。

重点回顾

好了，今天内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

今天，我们带你剖析了如何利用职责链模式和动态代理模式来实现 MyBatis Plugin。至此，我们就已经学习了三种职责链常用的应用场景：过滤器（Servlet Filter）、拦截器（Spring Interceptor）、插件（MyBatis Plugin）。

职责链模式的实现一般包含处理器和处理器链两部分。这两个部分对应到 Servlet Filter 的源码就是 Filter 和 FilterChain，对应到 Spring Interceptor 的源码就是 HandlerInterceptor 和 HandlerExecutionChain，对应到 MyBatis Plugin 的源码就是 Interceptor 和 InterceptorChain。除此之外，MyBatis Plugin 还包含另外一个非常重要的类：Plugin 类。它用来生成被拦截对象的动态代理。

在这三种应用场景中，职责链模式的实现思路都不大一样。其中，Servlet Filter 采用递归来实现拦截方法前后添加逻辑。Spring Interceptor 的实现比较简单，把拦截方法前后要添加的逻辑放到两个方法中实现。MyBatis Plugin 采用嵌套动态代理的方法来实现，实现思路很有技巧。

课堂讨论

Servlet Filter、Spring Interceptor 可以用来拦截用户自己定义的类的方法，而 MyBatis Plugin 默认可以拦截的只有 Executor、StatementHandler、ParameterHandler、ResultSetHandler 这四个类的方法，而且这四个类是 MyBatis 实现的类，并非用户自己定义的类。那 MyBatis Plugin 为什么不像 Servlet Filter、Spring Interceptor 那样，提供拦截用户自定义类的方法的功能呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

课程预告

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片，立即查看 >>>

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 87 | 开源实战五（上）：MyBatis如何权衡易用性、性能和灵活性？

下一篇 89 | 开源实战五（下）：总结MyBatis框架中用到的10种设计模式

精选留言 (9)

写留言



test

2020-05-25

思考题：因为用mybatis就是为了使用数据库。

展开 ∨



5



小晏子

2020-05-25

我感觉这要从mybatis的使用场景考虑，mybatis主要用于简化数据库操作，所以对于SQL语句的解析才是其本质，而不需要额外支持其他的东西，所以不需要拦截用户自定义类的方法



4



Monday

2020-05-26

看第一篇以为听懂了，再第二篇，发现根本没懂。如果换成是我要实现sql耗时的操作，走两步就行

- 1、写一个切面拦截StatementHandler的某些方法，在执行sql前后加开始结束时间就行。
- 2、上一点中拦截哪些方法，还是需要一个类似Plugin中的getSignatureMap方法的解析，没感觉到Plugin类其他的价值。。

展开 ∨



2



Monday

2020-05-26

精彩，看了源码，Mybatis分布工具PageHelper也通过Plugin方式实现的。

```
@Intercepts({@Signature(  
    type = Executor.class,  
    method = "query",  
    args = {MappedStatement.class, Object.class, RowBounds.class, ResultHandler...
```

展开 ∨



2



your problem?

2020-05-25

思考题：YAGNI，单一职责原则，MyBatis就是负责简化以及通用数据库的处理，没有必

要支持过多无关的东西

展开 ∨



2



Heaven

2020-05-26

职责单一,我用Mybatis就是为了更快更好的处理数据库之间的关系,所以专注于这四类是必然的,之前咱自己也看过Mybatis源码,但是并没有看出来是利用代理和职责链实现的整体执行过程



1



Jeff.Smile

2020-05-30

Springaop中的前置通知, 后置通知, 异常通知也是基于动态代理的职责链模式。



Bern

2020-05-28

是不是因为mybatis的版本问题, 有些方法和入参是不一样的



Lambor

2020-05-27

MyBatis 每次SQL执行都会创建 Executor 等对象, 再通过 pluginAll 方法创建一个代理的职责链, 然后递归调用每个代理对象, 最后调用 Executor 对象的方法。个人认为这个代理职责链主要就是控制 Executor 的方法在最后一步执行, 这种职责链+代理的实现方式虽然巧妙, 但感觉得不偿失, 每次SQL调用都会创建一个新的嵌套代理调用链, 这本身就是有性能消耗的, 而且是作为底层框架, 这点性能还是要考虑的。感觉采用 ApplicationFilte...

展开 ∨

