



MyBatis 拦截器原理探究

2017-04-10 ImportNew

(点击上方公众号，可快速关注)

来源：fanjian0423,
fangjian0423.github.io/2014/12/15/mybatis_interceptor/

如有好文章投稿，请点击 → 这里了解详情

MyBatis拦截器介绍

MyBatis提供了一种插件(plugin)的功能，虽然叫做插件，但其实这是拦截器功能。那么拦截器拦截MyBatis中的哪些内容呢？

我们进入官网看一看：

MyBatis 允许你在已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：

1. Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
2. ParameterHandler (getParameterObject, setParameters)
3. ResultSetHandler (handleResultSets, handleOutputParameters)
4. StatementHandler (prepare, parameterize, batch, update, query)

我们看到了可以拦截Executor接口的部分方法，比如update, query, commit, rollback等方法，还有其他接口的一些方法等。

总体概括为：

1. 拦截执行器的方法
2. 拦截参数的处理
3. 拦截结果集的处理
4. 拦截Sql语法构建的处理

拦截器的使用

拦截器介绍及配置

首先我们看下MyBatis拦截器的接口定义：

```
public interface Interceptor {  
  
    Object intercept(Invocation invocation) throws Throwable;  
  
    Object plugin(Object target);  
  
    void setProperties(Properties properties);  
  
}
```



比较简单，只有3个方法。MyBatis默认没有一个拦截器接口的实现类，开发者们可以实现符合自己需求的拦截器。

下面的MyBatis官网的一个拦截器实例：

```
@Intercepts({@Signature(  
    type= Executor.class,  
    method = "update",  
    args = {MappedStatement.class, Object.class}))  
public class ExamplePlugin implements Interceptor {  
    public Object intercept(Invocation invocation) throws Throwable {  
        return invocation.proceed();  
    }  
    public Object plugin(Object target) {  
        return Plugin.wrap(target, this);  
    }  
    public void setProperties(Properties properties) {  
    }  
}
```

全局xml配置：

```
<plugins>  
    <plugin interceptor="org.format.mybatis.cache.interceptor.ExamplePlugin"></plugin>  
</plugins>
```

这个拦截器拦截Executor接口的update方法（其实也就是SqlSession的新增，删除，修改操作），所有执行executor的update方法都会被该拦截器拦截到。

源码分析

下面我们分析一下这段代码背后的源码。



首先从源头->配置文件开始分析：

XMLConfigBuilder解析MyBatis全局配置文件的pluginElement私有方法：

```
private void pluginElement(XNode parent) throws Exception {
    if (parent != null) {
        for (XNode child : parent.getChildren()) {
            String interceptor = child.getStringAttribute("interceptor");
            Properties properties = child.getChildrenAsProperties();
            Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance();
            interceptorInstance.setProperties(properties);
            configuration.addInterceptor(interceptorInstance);
        }
    }
}
```

具体的解析代码其实比较简单，就不贴了，主要就是通过反射实例化plugin节点中的interceptor属性表示的类。然后调用全局配置类Configuration的addInterceptor方法。

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

这个interceptorChain是Configuration的内部属性，类型为InterceptorChain，也就是一个拦截器链，我们来看下它的定义：

```
public class InterceptorChain {

    private final List<Interceptor> interceptors = new ArrayList<Interceptor>();

    public Object pluginAll(Object target) {
        for (Interceptor interceptor : interceptors) {
            target = interceptor.plugin(target);
        }
        return target;
    }

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



```
}

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

}
```

现在我们理解了拦截器配置的解析以及拦截器的归属，现在我们回过头看下为何拦截器会拦截这些方法（Executor, ParameterHandler, ResultSetHandler, StatementHandler的部分方法）：

```
public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler = mappedStatement.getLang().createParameterHandler(mappedStatement, parameterObject, boundSql);
    parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
}
```

```
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);
    resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler);
    return resultSetHandler;
}
```

```
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);
    statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);
    return statementHandler;
}
```

```
public Executor newExecutor(Transaction transaction, ExecutorType executorType, boolean autoCommit) {
    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, autoCommit);
}
executor = (Executor) interceptorChain.pluginAll(executor);
return executor;
}
```

以上4个方法都是Configuration的方法。这些方法在MyBatis的一个操作(新增, 删除, 修改, 查询)中都会被执行到, 执行的先后顺序是Executor, ParameterHandler, ResultSetHandler, StatementHandler(其中ParameterHandler和ResultSetHandler的创建是在创建StatementHandler[3个可用的实现类CallableStatementHandler, PreparedStatementHandler, SimpleStatementHandler]的时候, 其构造函数调用的[这3个实现类的构造函数其实都调用了父类BaseStatementHandler的构造函数])。

这4个方法实例化了对应的对象之后, 都会调用interceptorChain的pluginAll方法, InterceptorChain的pluginAll刚才已经介绍过了, 就是遍历所有的拦截器, 然后调用各个拦截器的plugin方法。注意: 拦截器的plugin方法的返回值会直接被赋值给原先的对象

由于可以拦截StatementHandler, 这个接口主要处理sql语法的构建, 因此比如分页的功能, 可以用拦截器实现, 只需要在拦截器的plugin方法中处理StatementHandler接口实现类中的sql即可, 可使用反射实现。

MyBatis还提供了@Intercepts和@Signature关于拦截器的注解。官网的例子就是使用了这2个注解, 还包括了Plugin类的使用:

```
@Override
public Object plugin(Object target) {
    return Plugin.wrap(target, this);
}
```

下面我们就分析这3个“新组合”的源码, 首先先看Plugin类的wrap方法:

```
public static Object wrap(Object target, Interceptor interceptor) {
    Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
    Class<?> type = target.getClass();
```



```
Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
if (interfaces.length > 0) {
    return Proxy.newProxyInstance(
        type.getClassLoader(),
        interfaces,
        new Plugin(target, interceptor, signatureMap));
}
return target;
}
```

Plugin类实现了InvocationHandler接口，很明显，我们看到这里返回了一个JDK自身提供的动态代理类。我们解剖一下这个方法调用的其他方法：

getSignatureMap方法：

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

getSignatureMap方法解释：首先会拿到拦截器这个类的@Interceptors注解，然后拿到这个注解的属性@Signature注解集合，然后遍历这个集合，遍历的时候拿出@Signature注解的type属性(Class

类型), 然后根据这个type得到带有method属性和args属性的Method。由于@Interceptors注解的@Signature属性是一个属性, 所以最终会返回一个以type为key, value为Set<Method>的Map。



```
@Intercepts({@Signature(
    type= Executor.class,
    method = "update",
    args = {MappedStatement.class, Object.class}}))
```

比如这个@Interceptors注解会返回一个key为Executor, value为集合(这个集合只有一个元素, 也就是Method实例, 这个Method实例就是Executor接口的update方法, 且这个方法带有MappedStatement和Object类型的参数)。这个Method实例是根据@Signature的method和args属性得到的。如果args参数跟type类型的method方法对应不上, 那么将会抛出异常。

getAllInterfaces方法:

```
private static Class<?>[] getAllInterfaces(Class<?> type, Map<Class<?>, Set<Method>> signatureMap) {
    Set<Class<?>> interfaces = new HashSet<Class<?>>();
    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()]);
}
```

getAllInterfaces方法解释: 根据目标实例target(这个target就是之前所说的MyBatis拦截器可以拦截的类, Executor, ParameterHandler, ResultSetHandler, StatementHandler)和它的父类们, 返回signatureMap中含有target实现的接口数组。

所以Plugin这个类的作用就是根据@Interceptors注解, 得到这个注解的属性@Signature数组, 然后根据每个@Signature注解的type, method, args属性使用反射找到对应的Method。最终根据调用的target对象实现的接口决定是否返回一个代理对象替代原先的target对象。

比如MyBatis官网的例子, 当Configuration调用newExecutor方法的时候, 由于Executor接口的update(MappedStatement ms, Object parameter)方法被拦截器被截获。因此最终返回的是一个代理类Plugin, 而不是Executor。这样调用方法的时候, 如果是代理类, 那么会执行:

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
```



```
try {  
    Set<Method> methods = signatureMap.get(method.getDeclaringClass());  
    if (methods != null && methods.contains(method)) {  
        return interceptor.intercept(new Invocation(target, method, args));  
    }  
    return method.invoke(target, args);  
} catch (Exception e) {  
    throw ExceptionUtil.unwrapThrowable(e);  
}  
}
```

没错，如果找到对应的方法被代理之后，那么会执行Interceptor接口的interceptor方法。

这个Invocation类如下：

```
public class Invocation {  
  
    private Object target;  
    private Method method;  
    private Object[] args;  
  
    public Invocation(Object target, Method method, Object[] args) {  
        this.target = target;  
        this.method = method;  
        this.args = args;  
    }  
  
    public Object getTarget() {  
        return target;  
    }  
  
    public Method getMethod() {  
        return method;  
    }  
  
    public Object[] getArgs() {  
        return args;  
    }  
  
    public Object proceed() throws InvocationTargetException, IllegalAccessException {  
        return method.invoke(target, args);  
    }  
}
```




}

}

它的`proceed`方法也就是调用原先方法(不走代理)。

总结

MyBatis拦截器接口提供的3个方法中，`plugin`方法用于某些处理器(Handler)的构建过程。`interceptor`方法用于处理代理类的执行。`setProperties`方法用于拦截器属性的设置。

其实MyBatis官网提供的使用`@Interceptors`和`@Signature`注解以及`Plugin`类这样处理拦截器的方法，我们不一定要直接这样使用。我们也可以抛弃这3个类，直接在`plugin`方法内部根据`target`实例的类型做相应的操作。

总体来说MyBatis拦截器还是很简单的，拦截器本身不需要太多的知识点，但是学习拦截器需要对MyBatis中的各个接口很熟悉，因为拦截器涉及到了各个接口的知识点。

看完本文有收获？请转发分享给更多人

关注「ImportNew」，看技术干货



ImportNew

分享 Java 相关技术干货 · 资讯 · 高薪职位 · 教程



微信号: ImportNew



长按识别二维码关注

伯乐在线 旗下微信公众号

商务合作QQ: 2302462408

[阅读原文](#)