

大多数框架，都支持插件，用户可通过编写插件来自行扩展功能，Mybatis也不例外。

我们从插件配置、插件编写、插件运行原理、插件注册与执行拦截的时机、初始化插件、分页插件的原理等六个方面展开阐述。

1. 插件配置

Mybatis的插件配置在configuration内部，初始化时，会读取这些插件，保存于Configuration对象的InterceptorChain中。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <plugins>
        <plugin interceptor="com.mybatis3.interceptor.MyBatisInterceptor">
            <property name="value" value="100" />
        </plugin>
    </plugins>
</configuration>

public class Configuration {
    protected final InterceptorChain interceptorChain = new
    InterceptorChain();
}
```

org.apache.ibatis.plugin.InterceptorChain.java源码。

```
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);
    }
}
```

```
}
```

上面的for循环代表了只要是插件，都会以责任链的方式逐一执行（别指望它能跳过某个节点），所谓插件，其实就类似于拦截器。

2. 如何编写一个插件

插件必须实现org.apache.ibatis.plugin.Interceptor接口。

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

intercept()方法：执行拦截内容的地方，比如想收点保护费。由plugin()方法触发，interceptor.plugin(target)足以证明。

plugin()方法：决定是否触发intercept()方法。

setProperties()方法：给自定义的拦截器传递xml配置的属性参数。

下面自定义一个拦截器：

```
@Intercepts({  
    @Signature(type = Executor.class, method = "query", args = {  
MappedStatement.class, Object.class,  
        RowBounds.class, ResultHandler.class } ),  
    @Signature(type = Executor.class, method = "close", args = { boolean.class  
}) })  
public class MyBatisInterceptor implements Interceptor {  
  
    private Integer value;  
  
    @Override  
    public Object intercept(Invocation invocation) throws Throwable {  
        return invocation.proceed();  
    }  
  
    @Override  
    public Object plugin(Object target) {  
        System.out.println(value);  
    }  
}
```

```

        // Plugin类是插件的核心类，用于给target创建一个JDK的动态代理对象，触发
        intercept()方法
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {
        value = Integer.valueOf((String) properties.get("value"));
    }
}

```

面对上面的代码，我们需要解决两个疑问：

1. 为什么要写Annotation注解？注解都是什么含义？

答：Mybatis规定插件必须编写Annotation注解，是必须，而不是可选。

@Intercepts注解：装载一个@Signature列表，一个@Signature其实就是一个需要拦截的方法封装。那么，一个拦截器要拦截多个方法，自然就是一个@Signature列表。

type = Executor.class, method = "query", args = { MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class }

解释：要拦截Executor接口内的query()方法，参数类型为args列表。

2. Plugin.wrap(target, this)是干什么的？

答：使用JDK的动态代理，给target对象创建一个delegate代理对象，以此来实现方法拦截和增强功能，它会回调intercept()方法。

org.apache.ibatis.plugin.Plugin.java源码：

```

public class Plugin implements InvocationHandler {

    private Object target;
    private Interceptor interceptor;
    private Map<Class<?>, Set<Method>> signatureMap;

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

    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) {
            // 创建JDK动态代理对象

```

```

        return Proxy.newProxyInstance(
            type.getClassLoader(),
            interfaces,
            new Plugin(target, interceptor, signatureMap));
    }
    return target;
}

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
        // 判断是否需要拦截的方法(很重要)
        if (methods != null && methods.contains(method)) {
            // 回调intercept()方法
            return interceptor.intercept(new Invocation(target, method, args));
        }
        return method.invoke(target, args);
    } catch (Exception e) {
        throw ExceptionUtil.unwrapThrowable(e);
    }
}
//...
}

```

Map<Class<?>, Set> signatureMap: 缓存需拦截对象的反射结果，避免多次反射，即target的反射结果。

所以，我们不要动不动就说反射性能很差，那是因为你没有像Mybatis一样去缓存一个对象的反射结果。

判断是否需要拦截的方法，这句注释很重要，一旦忽略了，都不知道Mybatis是怎么判断是否执行拦截内容的，要记住。

3. Mybatis可以拦截哪些接口对象？

```

public class Configuration {
    //...
    public ParameterHandler newParameterHandler(MappedStatement mappedStatement,
        Object parameterObject, BoundSql boundSql) {
        ParameterHandler parameterHandler =
            mappedStatement.getLang().createParameterHandler(mappedStatement,
                parameterObject, boundSql);
        parameterHandler = (ParameterHandler)
            interceptorChain.pluginAll(parameterHandler); // 1
    }
}

```

```

        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); // 2
        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); // 3
        return statementHandler;
    }

    public Executor newExecutor(Transaction transaction) {
        return newExecutor(transaction, defaultExecutorType);
    }

    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);
        }
        executor = (Executor) interceptorChain.pluginAll(executor); // 4
        return executor;
    }
    //...
}

```

Mybatis只能拦截ParameterHandler、ResultSetHandler、StatementHandler、Executor共4个接口对象内的方法。

重新审视interceptorChain.pluginAll()方法：该方法在创建上述4个接口对象时调用，其含义为给这些接口对象注册拦截器功能，注意是注册，而不是执行拦截。

拦截器执行时机：plugin()方法注册拦截器后，那么，在执行上述4个接口对象内的具体方法时，就会自动触发拦截器的执行，也就是插件的执行。

所以，一定要分清，何时注册，何时执行。切不可认为pluginAll()或plugin()就是执行，它只是注册。

4. Invocation

```
public class Invocation {  
    private Object target;  
    private Method method;  
    private Object[] args;  
}
```

intercept(Invocation invocation)方法的参数Invocation，我相信你一定可以看得懂，不解释。

5. 初始化插件源码解析

org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XNode)方法部分源码。

```
pluginElement(root.evalNode("plugins"));  
  
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();  
            // 这里展示了setProperties()方法的调用时机  
            interceptorInstance.setProperties(properties);  
            configuration.addInterceptor(interceptorInstance);  
        }  
    }  
}
```

对于Mybatis，它并不区分是何种拦截器接口，所有的插件都是Interceptor，Mybatis完全依靠Annotation去标识对谁进行拦截，所以，具备接口一致性。

6. 分页插件原理

由于Mybatis采用的是逻辑分页，而非物理分页，那么，市场上就出现了可以实现物理分页的Mybatis的分页插件。

要实现物理分页，就需要对String sql进行拦截并增强，Mybatis通过BoundSql对象存储String sql，而BoundSql则由StatementHandler对象获取。

```
public interface StatementHandler {
    <E> List<E> query(Statement statement, ResultHandler resultHandler) throws
    SQLException{
        String sql = getBoundSql();
        分页语句: sql+"limit 语句"
        查询总数语句: "SELECT COUNT(1) " +sql.substring(from语句之后)
    };

    BoundSql getBoundSql();
}

public class BoundSql {
    public String getSql() {
        return sql;
    }
}
```

因此，就需要编写一个针对StatementHandler的query方法拦截器，然后获取到sql，对sql进行重写增强。

任它天高海阔，任它变化无穷，我们只要懂得原理，再多插件，我们都可以对其投送王之蔑视。