1、Mybatis插件

Mybatis提供了插件机制供使用方扩展，例如基于插件机制可以实现分页、分表、监控等功能，并且做到业务无感知。
Mybatis插件的本质是拦截器，**可以针对Mybatis的四大核心组件的方法进行拦截：Executor、StatementHandler、ParameterHandler、ResultSetHandler**

- Executor：SQL执行器，例如增删改查等方法；
- StatementHandler：SQL语法编译器，如预编译等；
- ParameterHandler：参数处理器；
- ResultSetHandler：查询结果集处理器。

2、Mybatis插件原理

对于Mybatis可以使用插件机制扩展的四大核心组件**Executor、StatementHandler、ParameterHandler、ResultSetHandler**，每次执行增删改查时，通过调用Configuration类的对应方法创建出对象后，立即对原始实例对象进行了包装增强，包装增强实际上是使用JDK动态代理为该原始对象创建了代理对象，并返回。
以Executor为例，源码分析如下：

```
Configuration.java

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);
  }
  // 根据sqlMapConfig.xml中配置的<plugins></plugins>指定的所有拦截器，
  // 创建executor的代理对象
  executor = (Executor) interceptorChain.pluginAll(executor);
  return executor;
}
```

```
InterceptorChain .java
// 拦截器链
public class InterceptorChain {

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

  // 对target创建代理对象，再对代理对象创建代理对象，层层代理,最后返回被重重代理后的对象
```

```
  // 可以在interceptor.plugin(target)中调用Plugin.wrap()创建代理对象
  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);
  }

}
```

```
// 继承了JDK动态代理拦截处理类
public class Plugin implements InvocationHandler {

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

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

  // 创建代理对象
  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;
  }

  // 拦截方法
```

```
29    @Override
30    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
31      try {
32        // 判断当前被调方法是否是Inteceptor上通过注解@Intercepts指定的方法
33        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
34        if (methods != null && methods.contains(method)) {
35          // 在Inteceptor中对被代理对象方法调用进行前后增强
36          return interceptor.intercept(new Invocation(target, method, args));
37        }
38        return method.invoke(target, args);
39      } catch (Exception e) {
40        throw ExceptionUtil.unwrapThrowable(e);
41      }
42    }
43
44    // 解析Interceptor上的@Intercepts注解，获取要拦截的方法集合
45    private static Map<Class<?>, Set<Method>> getSignatureMap(Interceptor interceptor) {
46      Intercepts interceptsAnnotation = interceptor.getClass().getAnnotation(Intercepts.cla
47      if (interceptsAnnotation == null) {
48        throw new PluginException("No @Intercepts annotation was found in interceptor " + i
49      }
50      Signature[] sigs = interceptsAnnotation.value();
51      Map<Class<?>, Set<Method>> signatureMap = new HashMap<Class<?>, Set<Method>>();
52      for (Signature sig : sigs) {
53        Set<Method> methods = signatureMap.get(sig.type());
54        if (methods == null) {
55          methods = new HashSet<Method>();
56          signatureMap.put(sig.type(), methods);
57        }
58        try {
59          Method method = sig.type().getMethod(sig.method(), sig.args());
60          methods.add(method);
61        } catch (NoSuchMethodException e) {
62          throw new PluginException("Could not find method on " + sig.type() + " named " +
63        }
64      }
65      return signatureMap;
66    }
67
68  }
69
```

总结：Mybatis在初始化时解析sqlMapConfig.xml文件中配置的插件，并保存插件实例到InteceptorChain中。初始化完成后，在执行SQL时，会先通过DefaultSqlSessionFactory创建SqlSession，在创建SqlSession的过程中会调用

Configuration中的相应方法创建Executor，创建后通过interceptorChain.pluginAll(executor)创建原始对象的JDK代理对象，这样，当调用目标对象的任何方法时，都会被代理处理器拦截，在拦截方法中判断是否是拦截器配置的想要拦截的目标方法，如果是，则执行Inteceptor的inteceptor方法逻辑进行增强，否则，直接调用目标方法。

3、自定义Mybatis插件

通过以上Mybatis插件的原理分析可知，要自定义插件，需要两个步骤：

1. 新建一个类，实现接口Interceptor，实现方法，并在类上用@Intercepts注解定义要拦截的方法集合

```
public interface Interceptor {

  // 拦截增强方法,每次执行都会调用
  Object intercept(Invocation invocation) throws Throwable;

  // 为目标对象创建代理对象(通过Plugin.wrap()方法)
  Object plugin(Object target);

  // 插件初始化时调用，只会调用一次
   // 将sqlMapConfig.xml中<plugin></plugin>中<property>标签配置的属性传递给参数properties
   void setProperties(Properties properties);

}
```

```
/**
 * @author zhangyongchao
 * @date 2020/4/24 18:12
 * @description
 */
@Intercepts({
        @Signature(
                type = StatementHandler.class,
                method = "prepare",
                 args = {Connection.class, Integer.class})
})
public class MyPlugin implements Interceptor {
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("执行前增强逻辑");
        Object proceed = invocation.proceed();
        System.out.println("执行后增强逻辑");
        return proceed;
    }
```

```
20
21      @Override
22      public Object plugin(Object target) {
23          return target instanceof  StatementHandler ? Plugin.wrap(target, this) : target;
24      }
25
26      @Override
27      public void setProperties(Properties properties) {
28          System.out.println("sqlMapConfig.xml中配置的属性: " + properties);
29      }
30  }
```

2. 在sqlMapConfig.xml中配置插件

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
3          "http://mybatis.org/dtd/mybatis-3-config.dtd">
4
5  <configuration>
6
7      <!--给实体类的全限定类名给别名-->
8      <typeAliases>
9          <!--给单独的实体起别名-->
10         <!--  <typeAlias type="com.zyc.pojo.User" alias="user"></typeAlias>-->
11         <!--批量起别名：该包下所有的类的本身的类名：别名还不区分大小写-->
12         <package name="com.zyc.pojo"/>
13     </typeAliases>
14
15     <!--引入自定义的插件-->
16     <plugins>
17         <plugin interceptor="com.zyc.plugin.MyPlugin">
18             <property name="name" value="mysql"/>
19         </plugin>
20     </plugins>
21
22     <!--引入映射配置文件-->
23     <mappers>
24         <!-- <mapper class="com.zyc.mapper.IUserMapper"></mapper>-->
25         <package name="com.zyc.mapper"/>
26     </mappers>
27
28 </configuration>
```