说一说你对Mybatis Plugin的了解

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：Mybatis插件

题目描述

说一说对Mybatis Plugin的了解

面试题分析

第一要可以讲出Mybatis插件的原理，第二要讲出它所涉及的JDK动态代理和责任链设计模式

前言

Mybatis作为一个应用广泛的优秀的ORM框架，已经成了JavaWeb世界近乎标配的部分，这个框架具有强大的灵活性，在四大组件（Executor、StatementHandler、ParameterHandler、ResultSetHandler）处提供了简单易用的插件扩展机制。Mybatis对持久层的操作就是借助于四大核心对象。MyBatis支持用插件对四大核心对象进行拦截，对mybatis来说插件就是拦截器，用来增强核心对象的功能，增强功能本质上是借助于底层的动态代理实现的，换句话说，MyBatis中的四大对象都是代理对象。

四大核心对象简介

MyBatis 四大核心对象

ParameterHandler：处理SQL的参数对象

ResultSetHandler：处理SQL的返回结果集

StatementHandler：数据库的处理对象，用于执行SQL语句

Executor：MyBatis的执行器，用于执行增删改查操作

Mybatis插件原理

1. Mybatis的插件借助于JDK动态代理和责任链设计模式进行对拦截的处理
2. 使用动态代理对目标对象进行包装，达到拦截的目的
3. 作用于Mybatis的作用域对象之上

拦截

插件具体是如何拦截并附加额外的功能的呢?

以ParameterHandler 来说

```
 public ParameterHandler newParameterHandler(MappedStatement mappedStatement,
Object object, BoundSql sql, InterceptorChain interceptorChain){
 ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement,object,sql);
 parameterHandler = (ParameterHandler)
interceptorChain.pluginAll(parameterHandler);
 return parameterHandler;
 }
public Object pluginAll(Object target) {
 for (Interceptor interceptor : interceptors) {
 target = interceptor.plugin(target);
 }
 return target;
 }
```

interceptorChain 保存了所有的拦截器(interceptors)，是mybatis初始化的时候创建的。调用拦截器链中的拦截器依次的对目标进行拦截或增强。interceptor.plugin(target)中的target就可以理解为mybatis中的四大对象。返回的target是被重重代理后的对象。

插件接口

Mybatis插件接口-Interceptor

1.Intercept方法，插件的核心方法

2.plugin方法，生成target的代理对象

3.setProperties方法，传递插件所需参数

插件实例

插件开发需要以下步骤

1. 自定义插件需要实现上述接口
2. 增加@Intercepts注解（声明是哪个核心组件的插件，以及对哪些方法进行扩展）
3. 在xml文件中配置插件

```
/** 插件签名，告诉mybatis插件用来拦截那个对象的哪个方法 **/
@Intercepts({@Signature(type = ResultSetHandler.class,method
="handleResultSets",args = Statement.class)})
public class MyFirstInterceptor implements Interceptor {

 /** @Description 拦截目标对象的目标方法 **/
 @Override
 public Object intercept(Invocation invocation) throws Throwable {
 System.out.println("拦截的目标对象："+invocation.getTarget());
 Object object = invocation.proceed();
 return object;
 }
 /**
 * @Description 包装目标对象 为目标对象创建代理对象
 * @Param target为要拦截的对象
 * @Return 代理对象
 */
 @Override
 public Object plugin(Object target) {
```

```
 System.out.println("将要包装的目标对象："+target);
 return Plugin.wrap(target,this);
 }
 /** 获取配置文件的属性 **/
 @Override
 public void setProperties(Properties properties) {
 System.out.println("插件配置的初始化参数："+properties);
 }
}
```

在mybatis.xml中配置插件

```
<!-- 自定义插件 -->
 <plugins>
  <plugin interceptor="mybatis.interceptor.MyFirstInterceptor">
   <!--配置参数-->
   <property name="name" value="Bob"/>
  </plugin>
 </plugins>
```

调用查询方法，查询方法会返回ResultSet

```
public class MyBatisTest {
 public static SqlSessionFactory sqlSessionFactory = null;

 public static SqlSessionFactory getSqlSessionFactory() {
  if (sqlSessionFactory == null) {
   String resource = "mybatis-config.xml";
   try {
    Reader reader = Resources.getResourceAsReader(resource);
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
   } catch (IOException e) {
    e.printStackTrace();
   }
  }
  return sqlSessionFactory;
 }

 public void testGetById()
 {
  SqlSession sqlSession = this.getSqlSessionFactory().openSession();
  PersonMapper personMapper = sqlSession.getMapper(PersonMapper.class);
  Person person=personMapper.getById(2001);
  System.out.println(person.toString());
 }

public static void main(String[] args) {
  new MyBatisTest().testGetById();
 }
}
```

输出结果

```
插件配置的初始化参数：{name=Bob}
将要包装的目标对象：org.apache.ibatis.executor.CachingExecutor@754ba872
将要包装的目标对象：
org.apache.ibatis.scripting.defaults.DefaultParameterHandler@192b07fd
将要包装的目标对象：
org.apache.ibatis.executor.resultset.DefaultResultSetHandler@7e0b0338
将要包装的目标对象：
org.apache.ibatis.executor.statement.RoutingStatementHandler@1e127982
拦截的目标对象：
org.apache.ibatis.executor.resultset.DefaultResultSetHandler@7e0b0338
Person{id=2001, username='Tom', email='email@0', gender='F'}
```

JDK动态代理+责任链设计模式

Mybatis的插件其实就是个**拦截器功能**。它利用`JDK动态代理和责任链设计模式的综合运用`。采用责任链模式，通过动态代理组织多个拦截器,通过这些拦截器你可以做一些你想做的事。

1、JDK动态代理案例

```
public class MyProxy {
    /**
     * 一个接口
     */
    public interface HelloService{
        void sayHello();
    }
    /**
     * 目标类实现接口
     */
    static class HelloServiceImpl implements HelloService{

        @Override
        public void sayHello() {
            System.out.println("sayHello......");
        }
    }
    /**
     * 自定义代理类需要实现InvocationHandler接口
     */
    static  class HWInvocationHandler implements InvocationHandler {
        /**
         * 目标对象
         */
        private Object target;

        public HWInvocationHandler(Object target){
            this.target = target;
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println("------插入前置通知代码-------------");
            //执行相应的目标方法
            Object rs = method.invoke(target,args);
            System.out.println("------插入后置处理代码-------------");
```

```
            return rs;
        }

        public static Object wrap(Object target) {
            return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                    target.getClass().getInterfaces(),new
HWInvocationHandler(target));
        }
    }
    public static void main(String[] args)  {
        HelloService proxyService = (HelloService) HWInvocationHandler.wrap(new
HelloServiceImpl());
        proxyService.sayHello();
    }
}
```

运行结果

```
------插入前置通知代码-------------
sayHello......
------插入后置处理代码-------------
```

2、优化

上面代理的功能是实现了,但是有个很明显的缺陷，就是`HWInvocationHandler是动态代理类`，也可以理解成是个工具类，我们不可能会把业务代码写到写到到`invoke方法`里，

不符合面向对象的思想，可以抽象一下处理。可以设计一个`Interceptor接口`，需要做什么拦截处理实现接口就行了。

```
public interface Interceptor {
    /**
     * 具体拦截处理
     */
    void intercept();
}
```

intercept() 方法就可以处理各种前期准备了

```
public class LogInterceptor implements Interceptor {
    @Override
    public void intercept() {
        System.out.println("------插入前置通知代码-------------");
    }
}

public class TransactionInterceptor implements Interceptor {
    @Override
    public void intercept() {
        System.out.println("------插入后置处理代码-------------");
    }
}
```

代理对象也做一下修改

```
public class HWInvocationHandler implements InvocationHandler {

    private Object target;

    private List<Interceptor> interceptorList = new ArrayList<>();

    public TargetProxy(Object target,List<Interceptor> interceptorList) {
        this.target = target;
        this.interceptorList = interceptorList;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
       //处理多个拦截器
        for (Interceptor interceptor : interceptorList) {
            interceptor.intercept();
        }
        return method.invoke(target, args);
    }

    public static Object wrap(Object target,List<Interceptor> interceptorList) {
        HWInvocationHandler targetProxy = new HWInvocationHandler(target,
interceptorList);
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),

target.getClass().getInterfaces(),targetProxy);
    }
}
```

现在可以根据需要动态的添加拦截器了，在每次执行业务代码sayHello()之前都会拦截，看起来高级一点，来测试一下

```
public class Test {
    public static void main(String[] args) {
        List<Interceptor> interceptorList = new ArrayList<>();
        interceptorList.add(new LogInterceptor());
        interceptorList.add(new TransactionInterceptor());

        HelloService target = new HelloServiceImpl();
        Target targetProxy = (Target) TargetProxy.wrap(target,interceptorList);
        targetProxy.sayHello();
    }
}
```

运行结果

```
------插入前置通知代码-------------
------插入后置处理代码-------------
sayHello......
```

3、再优化

上面的动态代理确实可以把代理类中的业务逻辑抽离出来，但是我们注意到，只有前置代理，无法做到前后代理，所以还需要在优化下。所以需要做更一步的抽象，

把拦截对象信息进行封装，作为拦截器拦截方法的参数，把拦截目标对象真正的执行方法放到Interceptor中完成，这样就可以实现前后拦截，并且还能对拦截

对象的参数等做修改。设计一个`Invocation 对象`。

```
public class Invocation {

    /**
     * 目标对象
     */
    private Object target;
    /**
     * 执行的方法
     */
    private Method method;
    /**
     * 方法的参数
     */
    private Object[] args;

    //省略getset
    public Invocation(Object target, Method method, Object[] args) {
        this.target = target;
        this.method = method;
        this.args = args;
    }

    /**
     * 执行目标对象的方法
     */
    public Object process() throws Exception{
       return method.invoke(target,args);
    }
}
```

Interceptor拦截接口做修改

```
public interface Interceptor {
    /**
     * 具体拦截处理
     */
    Object intercept(Invocation invocation) throws Exception;
}
```

Interceptor实现类

```
public class TransactionInterceptor implements Interceptor {

    @Override
    public Object intercept(Invocation invocation) throws Exception{
        System.out.println("------插入前置通知代码-------------");
        Object result = invocation.process();
        System.out.println("------插入后置处理代码-------------");
        return result;
    }
}
```

Invocation 类就是被代理对象的封装，也就是要拦截的真正对象。HWInvocationHandler修改如下：

```
public class HWInvocationHandler implements InvocationHandler {

    private Object target;

    private Interceptor interceptor;

    public TargetProxy(Object target,Interceptor interceptor) {
        this.target = target;
        this.interceptor = interceptor;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        Invocation invocation = new Invocation(target,method,args);
        return interceptor.intercept(invocation);
    }

    public static Object wrap(Object target,Interceptor interceptor) {
        HWInvocationHandler targetProxy = new HWInvocationHandler(target,
interceptor);
        return
Proxy.newProxyInstance(target.getClass().getClassLoader(),target.getClass().getI
nterfaces(),targetProxy);
    }
}
```

测试类

```
public class Test {
    public static void main(String[] args) {
        HelloService target = new HelloServiceImpl();
        Interceptor transactionInterceptor = new TransactionInterceptor();
        HelloService targetProxy = (Target)
TargetProxy.wrap(target,transactionInterceptor);
        targetProxy.sayHello();
    }
}
```

运行结果

```
------插入前置通知代码-------------
sayHello......
------插入后置处理代码-------------
```

4、再再优化

上面这样就能实现前后拦截，并且拦截器能获取拦截对象信息。但是测试代码的这样调用看着很别扭，对应目标类来说，只需要了解对他插入了什么拦截就好。

再修改一下，在拦截器增加一个插入目标类的方法。

```
public interface Interceptor {
```

```
    /**
     * 具体拦截处理
     */
    Object intercept(Invocation invocation) throws Exception;

    /**
     *  插入目标类
     */
    Object plugin(Object target);

}

public class TransactionInterceptor implements Interceptor {

    @Override
    public Object intercept(Invocation invocation) throws Exception{
        System.out.println("------插入前置通知代码-------------");
        Object result = invocation.process();
        System.out.println("------插入后置处理代码-------------");
        return result;
    }

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

这样目标类仅仅需要在执行前，插入需要的拦截器就好了，测试代码：

```
public class Test {
    public static void main(String[] args) {
        HelloService target = new HelloServiceImpl();
        Interceptor transactionInterceptor = new TransactionInterceptor();
        //把事务拦截器插入到目标类中
        target = (HelloService) transactionInterceptor.plugin(target);
        target.sayHello();
    }
}
```

运行结果

```
------插入前置通知代码-------------
sayHello......
------插入后置处理代码-------------
```

5、多个拦截器如何处理

到这里就差不多完成了，那我们再来思考如果要添加多个拦截器呢，怎么搞？

```
public class Test {
    public static void main(String[] args) {
        HelloService target = new HelloServiceImpl();
        Interceptor transactionInterceptor = new TransactionInterceptor();
        target = (HelloService) transactionInterceptor.plugin(target);
        LogInterceptor logInterceptor = new LogInterceptor();
        target = (HelloService)logInterceptor.plugin(target);
        target.sayHello();
    }
}
```

运行结果

```
------插入前置通知代码-------------
------插入前置通知代码-------------
sayHello......
------插入后置处理代码-------------
------插入后置处理代码-------------
```

6、责任链设计模式

其实上面已经实现的没问题了，只是还差那么一点点，添加多个拦截器的时候不太美观，让我们再次利用面向对象思想封装一下。我们设计一个`InterceptorChain 拦截器链类`

```
public class InterceptorChain {

    private List<Interceptor> interceptorList = new ArrayList<>();

    /**
     * 插入所有拦截器
     */
    public Object pluginAll(Object target) {
        for (Interceptor interceptor : interceptorList) {
            target = interceptor.plugin(target);
        }
        return target;
    }

    public void addInterceptor(Interceptor interceptor) {
        interceptorList.add(interceptor);
    }
    /**
     * 返回一个不可修改集合，只能通过addInterceptor方法添加
     * 这样控制权就在自己手里
     */
    public List<Interceptor> getInterceptorList() {
        return Collections.unmodifiableList(interceptorList);
    }
}
```

其实就是通过pluginAll() 方法包一层把所有的拦截器插入到目标类去而已。测试代码：

```
public class Test {

    public static void main(String[] args) {
        HelloService target = new HelloServiceImpl();
        Interceptor transactionInterceptor = new TransactionInterceptor();
        LogInterceptor logInterceptor = new LogInterceptor();
        InterceptorChain interceptorChain = new InterceptorChain();
        interceptorChain.addInterceptor(transactionInterceptor);
        interceptorChain.addInterceptor(logInterceptor);
        target = (Target) interceptorChain.pluginAll(target);
        target.sayHello();
    }
}
```

这里展示的是`JDK动态代理+责任链设计模式`，那么Mybatis拦截器就是基于该组合进行开发。