42 离开Spring AOP，我们如何实现AOP功能？

更新时间：2020-08-26 10:57:17

> 人生的旅途，前途很远，也很暗。然而不要怕，不怕的人的面前才有路。—— 鲁 迅

背景

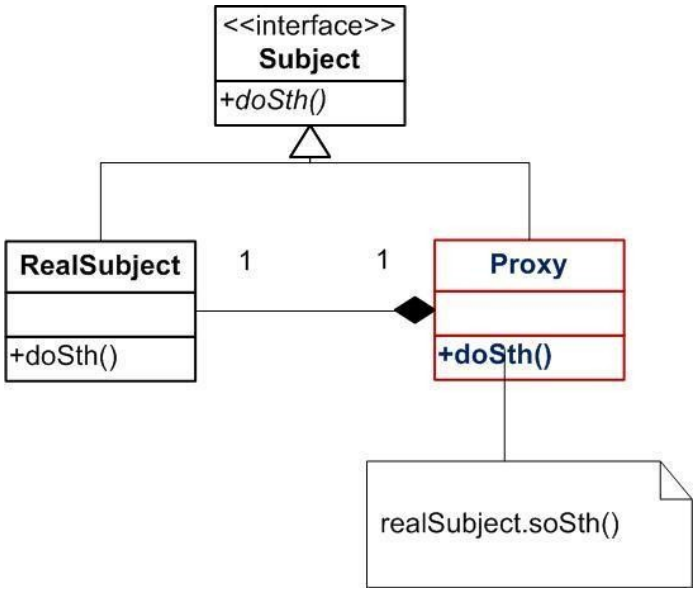
Spring AOP 本身依赖 Spring 其它项目，不能单独作为一个项目引入，如果我们的开发框架不是 Spring 或者衍生的框架，那么 Spring AOP 使用起来就比较麻烦了。此时我们该怎么办呢？我们知道 Spring AOP 基于 JDK 动态代理实现，如果在我们使用的框架内实现了 JDK 动态代理，不就可以利用 AOP 的功能了嘛。

这个之前我们先复习一下设计模式中的代理模式。

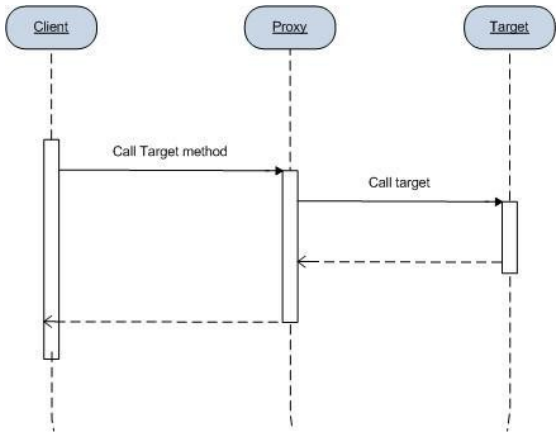
代理模式

代理模式是非常常用的一种设计模式，在我们的应用中经常被使用。一般场景是，我们有一个现成的类，它的功能已经比较完善了，但是还是存在某些欠缺，这个时候我们需要去扩展一些新的功能，但又不想去重造轮子，这个时候可以使用代理类来替代原来的目标类，通过组合的模式，增加一种为目标类增加一些额外的功能。

代理模式的类结构图一般如下：



它的调用顺序如下图所示：



当 Client 调用 目标方法时，其实是调用代理类中与之对应的方法，该方法则会去调用目标类的方法。

Spring AOP 代理内部实现原理

Spring AOP 默认的 AOP 代理使用标准 JDK 动态代理。它允许代理任何接口（或接口集）。

Spring AOP 还可以使用 Cglib 代理。使用 CGLIG 的三种情况：

- ProxyConfig 中的 optimize 标识被置为 true；
- ProxyConfig 中的 proxyTargetClass 标识被置为 true；
- 目标类没有可用的代理接口即目标类没有实现接口。

源码不会欺骗我们，我们来看看吧：

```
@Override
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
 StackUtils.getStack();
 if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
  Class<?> targetClass = config.getTargetClass();
  if (targetClass == null) {
   throw new AopConfigException("TargetSource cannot determine target class: " +
     "Either an interface or a target is required for proxy creation.");
  }
  if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
   return new JdkDynamicAopProxy(config);
  }
  return new ObjenesisCglibAopProxy(config);
 }
 else {
  return new JdkDynamicAopProxy(config);
 }
}
```

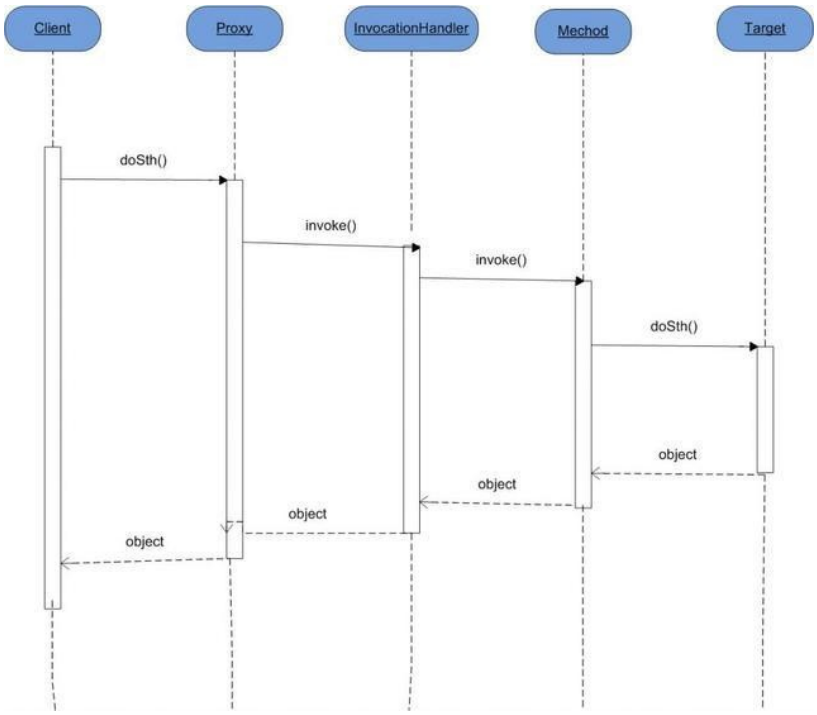
利用 JDK 动态代理实现 AOP

由于对接口而不是类进行编程是一种良好的实践，所以业务类通常实现一个或多个业务接口。所以想实现 AOP 可以利用 JDK 的动态代理。JDK 的动态代理有两个主类：

`java.lang.reflect.Proxy`：创建代理实例；

`java.lang.reflect.InvocationHandler`：增强的业务逻辑写在里面。

其时序图如下所示：



示例：

- 接口定义

```
package com.davidwang.test;
public interface HelloWorld {
 public void sayHello();
}
```

- 实现类

```
package com.davidwang.test;
public class HelloWorldImp implements HelloWorld {
    public void sayHello() {
     System.out.println("hello world!");
    }
}
```

- 代理类

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
public class HelloWorldHandler implements InvocationHandler{
    public Object target;
    HelloWorldHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {        System.out.println(method.getName() + " invoked!");
  Object result = method.invoke(target, args);
  System.out.println(method.getName() + " return!");
        return result;
        }
}
```

- 测试类

```
package com.davidwang.test;
import java.lang.reflect.Proxy;
public class TestDynamicProxy {
    public static void main(String[] args) {
        HelloWorld hello= (HelloWorld)Proxy.newProxyInstance(TestDynamicProxy.class.getClassLoader(),  new Class[] {HelloWorld.class}, new HelloWo
rldHandler(new HelloWorldImp()));
        hello.sayHello();
    }
}
```

总结

我们进一步打开 JdkDynamicAopProxy 的源码，发现它实现 InvocationHandler 接口。

```
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {
    .....
}
```

有兴趣的同学可以参考 invoke 实现，和上面 JDK 的动态代理时序一致。

参考资料：

【1】https://blog.csdn.net/abing37/article/details/5449401

}

41 注解 @Aspect 是如何工作的？

43 离开了 Spring AOP，我们如何切面编程？