Java动态代理（Spring AOP原理）

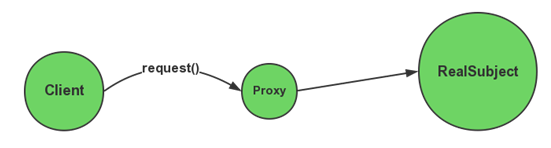
# 介绍

Spring重要的两个基础概念：控制反转（IOC）和面向切面编程（AOP）。前面的文章我们已经介绍了依赖注入，下面我们就来了解什么是面向切面编程。

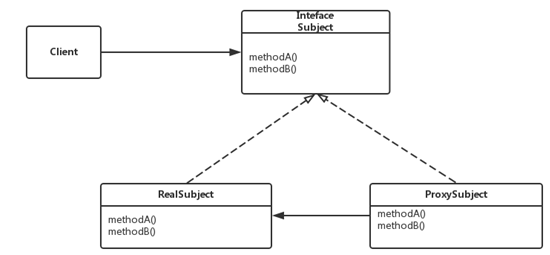
本文不打算介绍面向切面编程，而是介绍AOP背后的原理——代理（Proxy）。

代理模式：另一个对象提供一个替身或占位符以控制对这个对象的访问。

比如，你（RealSubject）是一个演员，你有一个经纪人（代理人、Agent），如果有电影（客户端、Client）需要找你合作。他们不是直接打电话找你，而是通过你的经纪人来和你联系，这时候经纪人就充当着代理的角色。其实，编程中的代理也就是这个经纪人的概念，我们为某些对象提供一个代理类。代理类负责为委托类（被代理类、演员）提供预处理消息，过滤消息，以及消息被委托类执行后的后续处理。用如下的图形来表示：



相对应的类图如下：



为了更清楚的阐述代理这个概念，我们先从没有代理的世界讲起。

# 不使用代理

首先，我们有如下接口：

|  |
| --- |
| // Subject接口，有一个表演的方法  public interface Performance {  // 表演接口  void perform(String msg);  }  // RealSubject，Performance接口的实现  public class Singer implements Performance {  @Override  public void perform(String msg) {  System.out.println("实际对象：" + msg);  }  } |

我们看到一个歌手（RealSubject）是Performance接口（Subject）的具体实现，打印了一串消息。如果我们需要在歌手执行表演方法之前给手机静音，然后在表演方法执行之后观众鼓掌，我们应该怎么实现。我们采用如下的实现：

|  |
| --- |
| // RealSubject  // Performance接口的实现  public class Singer implements Performance {  @Override  public void perform(String msg) {  // 实际方法调用之前  System.out.println("实际方法调用之前：手机静音。");  System.out.println("实际对象：" + msg);  // 实际方法调用之后  System.out.println("实际方法调用之后：观众鼓掌！");  }  } |

这样，我们就实现了我们的需求。但是这样的问题是：歌手只需要关心自己的表演部分，怎么还需要关心表演开始后结束之后的事情呢。这样也增加了代码的耦合度，而且如果有很多个Performance实现，那么我们岂不是需要每个类都去修改。这样岂不是要累死程序员。有什么办法能够在不修改Singer实现代码的情况下，实现添加功能的需求呢？

# 静态代理

使用静态代理就需要为Performance实现类增加代理类，根据上面代理的类图我们知道，代理类也需要实现Performance接口。所以，代理类如下：

|  |
| --- |
| // SingerProxy.class  public class SingerProxy implements Performance {  // 对真实代理对象的引用  private Performance per;  public SingerProxy(Performance performance) {  this.per = performance;  }  @Override  public void perform(String msg) {  System.out.println("实际方法调用之前：手机静音。");  this.per.perform(msg);  System.out.println("实际方法调用之后：鼓掌！！！");  }  }  // 客户端使用代理类  public class StaticProxyTest {  public static void main(String[] args) {  Performance performance = new SingerProxy(new Singer());  performance.perform("天降一个女朋友！！");  }  }  // 代码运行结果  实际方法调用之前：手机静音。  实际对象：天降一个女朋友！！  实际方法调用之后：鼓掌！！！ |

如此处理以后，我们看到在不需要修改RealSubject的情况下，我们就实现了在方法调用前和调用后增加通用逻辑的功能。如果，以后有了多个的Performance实现，我们也不需要为每个实现类增加多余的代码。那么，静态代理有哪些缺点呢：

代理类的一个接口只服务于一种类型的对象，如果接口的方法过多，那么必须为每种方法都进行代理。如果程序规模稍大，那么静态代理就不能胜任了。

如果接口增加了一个方法，那么不只是实现类需要修改以增加方法，相应的代理类也需要修改以便增加对新方法的代理。增加了代码维护的难度。

那么，对于以上的问题，我们使用Java动态代理（Dynamic Proxy）来解决。

# 动态代理

首先，我们需要理解什么叫动态代理：代理类的源码是在运行期间由JVM动态生成的，也就是说在编译期根本就不存在代理类的字节码（源码）文件，这也是和静态代理很大的不同之处。要使用Java动态代理，我们首先要了解以下接口和API：

|  |
| --- |
| java.lang.reflect.Proxy：这是 Java 动态代理机制的主类，它提供了一组静态方法来为一组接口动态地生成代理类及其对象。主要我们要使用的方法是如下方法：  // 该方法用于为指定类装载器、一组接口及调用处理器生成动态代理类实例  static Object newProxyInstance(ClassLoader loader, Class[] interfaces,  InvocationHandler h)  ClassLoader loader：类加载器，如果为null则是默认加载器，该加载器必须能够访问被代理的interfaces接口  Class[] interfaces：被代理的接口数组对象，生成的代理对象会继承每一个接口  InvocationHandler h：通过代理类实例调用某个接口的方法，都会被转发到InvocationHandler对象，通过该对象实现代理访问 |
| java.lang.reflect.InvocationHandler：这是调用处理器接口，它自定义了一个 invoke 方法，用于集中处理在动态代理类对象上的方法调用，通常在该方法中实现对委托类的代理访问。  // 该方法负责集中处理动态代理类上的所有方法调用。  // 第一个参数既是代理类实例，  // 第二个参数是被调用的方法对象  // 第三个方法是调用参数。  // 调用处理器根据这三个参数进行预处理或分派到委托类实例上发射执行  Object invoke(Object proxy, Method method, Object[] args)  Object proxy：代理类实例  Method method：被调用的接口的方法对象  Object[] args：调用方法的参数 |

现在，我们使用这些API实现对Performance接口的动态代理访问：

实现InvocationHandler接口，创建自己的调用处理器

使用Proxy的newProxyInstance静态方法创建代理类

|  |
| --- |
| // 创建自己的调用处理器  **public** **class** DynamicProxy **implements** InvocationHandler {  // 对真实对象的引用  **private** Object target;  // 注入真实对象  **public** DynamicProxy(Object obj) {  **this**.target = obj;  }  @Override  **public** Object invoke(Object arg0, Method method, Object[] args) **throws** Throwable {  // 方法调用前添加的逻辑  System.***out***.println("实际方法调用之前：手机静音。");  // 真实方法调用  method.invoke(target, args);  // 真实方法调用后添加的逻辑  System.***out***.println("实际方法调用之后：鼓掌！！");  **return** **null**;  }    @SuppressWarnings("unchecked")  **public** <T> T getProxy() {  **return** (T) Proxy.*newProxyInstance*(  target.getClass().getClassLoader(),  target.getClass().getInterfaces(),  **this**  );  }  } |
| // 客户端使用动态代理  **public** **class** DynamicProxyTest {  **public** **static** **void** main(String[] args) {    DynamicProxy dynamicProxy = **new** DynamicProxy(**new** Singer());  Performance proxy = dynamicProxy.getProxy();  proxy.perform("天降一个女朋友！");  }  }  // 运行结果  实际方法调用之前：手机静音。  实际对象：天降一个女朋友！！  实际方法调用之后：鼓掌！！  注意：调用代理对象方法，会调用其invoke()进行实现。 |

以上，我们使用动态代理实现了我们的需求，有什么优点呢。首先，我们把所有的控制对RealSubject的逻辑都放在了一个InvocationHandler实现中，所以不需要为每个方法编写代理，增加了可维护性。其次，如果接口新增了方法，我们也不需要修改代理的代码去实现对其的代理。但是，动态代理的不足在于它只能对接口实现代理，不能对类实现代理。如果需要对类实现代理，可以使用CGLib实现。

# GLib动态代理原理及实现

JDK实现动态代理需要实现类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

|  |
| --- |
| **public** **class** CglibProxy **implements** MethodInterceptor {  **private** Enhancer enhancer = **new** Enhancer();  **public** Object getProxy(Class clazz) {  //设置需要创建子类的类  enhancer.setSuperclass(clazz);  enhancer.setCallback(**this**);  //通过字节码技术动态创建子类实例  **return** enhancer.create();  }  //实现MethodInterceptor接口方法  **public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {  System.***out***.println("前置代理");  //通过代理类调用父类中的方法  Object result = proxy.invokeSuper(obj, args);  System.***out***.println("后置代理");  **return** result;  }  } |
| **public** **class** CglibProxyTest {  **public** **static** **void** main(String[] args) {  CglibProxy proxy = **new** CglibProxy();  //通过生成子类的方式创建代理类  Singer singerProxy = (Singer)proxy.getProxy(Singer.**class**);  singerProxy.perform("天降一个女朋友");  }  } |

# 总结

我们看到使用动态代理可以在不修改源码的情况下为某个某些接口实现对其的访问。而Spring AOP的作用就是把程序中公共业务分离出来单独编写在一处。所以，动态代理是理想的实现原理。下一遍文章将讨论什么是Spring AOP，以及使用。