https://mp.weixin.qq.com/s/rAAKRJjPJU2\_FVR37dzzcg

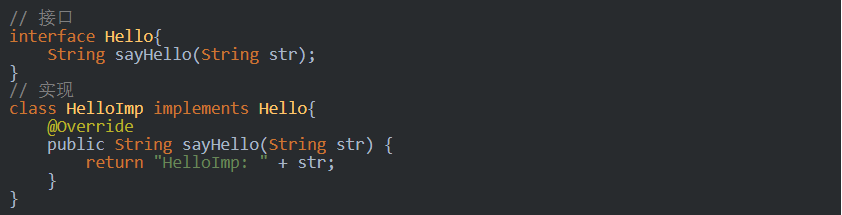
# Java中的原生动态代理和CGLIB动态代理的原理

动态代理在Java中有着广泛的应用，比如Spring AOP，Hibernate数据查询、测试框架的后端mock、RPC，Java注解对象获取等。静态代理的代理关系在编译时就确定了，而动态代理的代理关系是在编译期确定的。静态代理实现简单，适合于代理类较少且确定的情况，而动态代理则给我们提供了更大的灵活性。

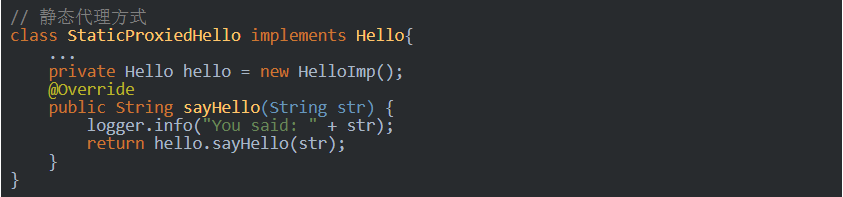
**今天我们来探讨Java中两种常见的动态代理方式：JDK原生动态代理和CGLIB动态代理。**

## JDK原生动态代理

先从直观的示例说起，假设我们有一个接口Hello和一个简单实现HelloImp：



这是Java种再常见不过的场景，使用接口制定协议，然后用不同的实现来实现具体行为。假设你已经拿到上述类库，如果我们想通过[日志记录](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247491875&idx=1&sn=f924372666bfb0ef372f8e14a7623913&chksm=ebd5de0fdca25719cf285eca6413b067f9bbf7143c024a40c1941a88101cf68c52fbc8c28865&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/s/_blank)对sayHello()的调用，使用静态代理可以这样做：



上例中静态代理类StaticProxiedHello作为HelloImp的代理，实现了相同的Hello接口。

用Java动态代理可以这样做：

首先实现一个InvocationHandler，方法调用会被转发到该类的invoke()方法。

然后在需要使用Hello的时候，通过JDK动态代理获取Hello的代理对象。



运行上述代码输出结果：



上述代码的关键是Proxy.newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)方法，该方法会根据指定的参数动态创建代理对象。三个参数的意义如下：

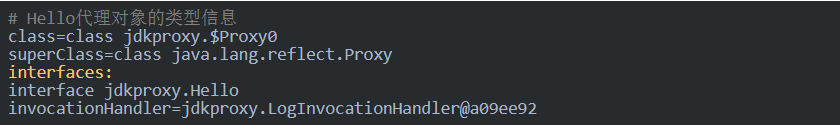
* loader，指定代理对象的类加载器；
* interfaces，代理对象需要实现的接口，可以同时指定多个接口；
* handler，方法调用的实际处理者，代理对象的方法调用都会转发到这里（\*注意1）。

newProxyInstance()会返回一个实现了指定接口的代理对象，对该对象的所有方法调用都会转发给InvocationHandler.invoke()方法。理解上述代码需要对Java反射机制有一定了解。动态代理神奇的地方就是：

* 代理对象是在程序运行时产生的，而不是编译期；
* 对代理对象的所有接口方法调用都会转发到InvocationHandler.invoke()方法，在invoke()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；之后我们通过某种方式执行真正的方法体，示例中通过反射调用了Hello对象的相应方法，还可以通过RPC调用远程方法。

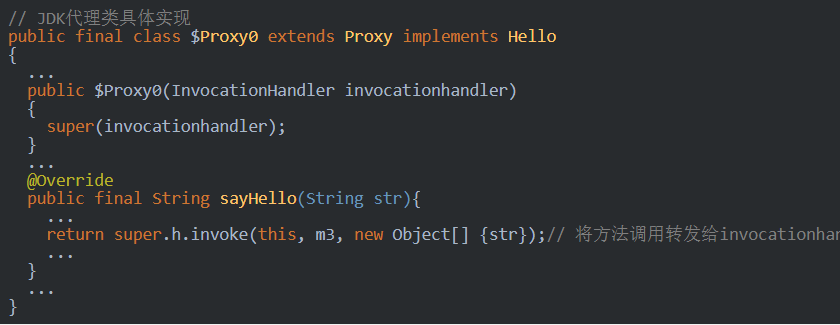
注意1：对于从Object中继承的方法，JDK Proxy会把hashCode()、equals()、toString()这三个非接口方法转发给InvocationHandler，其余的Object方法则不会转发。详见JDK Proxy官方文档。

如果对JDK代理后的对象类型进行深挖，可以看到如下信息：



代理对象的类型是jdkproxy.$Proxy0，这是个动态生成的类型，类名是形如$ProxyN的形式；父类是java.lang.reflect.Proxy，所有的JDK动态代理都会继承这个类；同时实现了Hello接口，也就是我们接口列表中指定的那些接口。

如果你还对jdkproxy.$Proxy0具体实现感兴趣，它大致长这个样子：

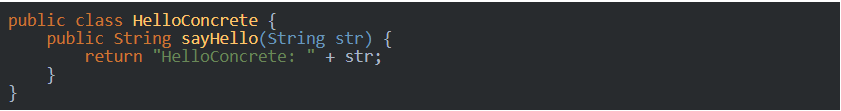


Java动态代理为我们提供了非常灵活的代理机制，但Java动态代理是基于接口的，如果对象没有实现接口我们该如何代理呢？CGLIB登场。

## CGLIB动态代理

CGLIB(Code Generation Library)是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过继承方式实现代理。

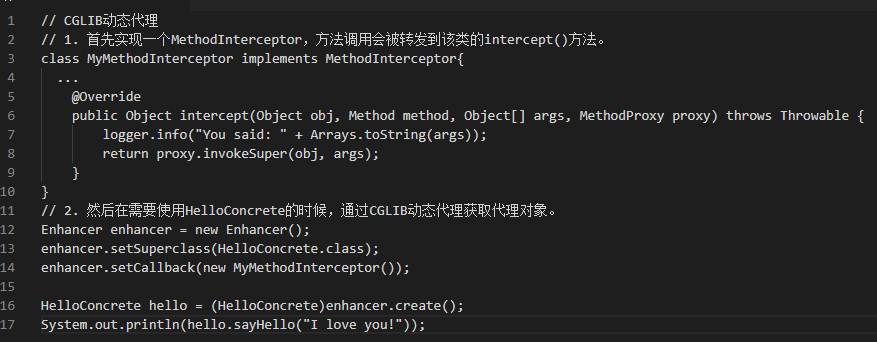
来看示例，假设我们有一个没有实现任何接口的类HelloConcrete：



因为没有实现接口该类无法使用JDK代理，通过CGLIB代理实现如下：

首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()方法。

然后在需要使用HelloConcrete的时候，通过CGLIB动态代理获取代理对象。



运行上述代码输出结果：

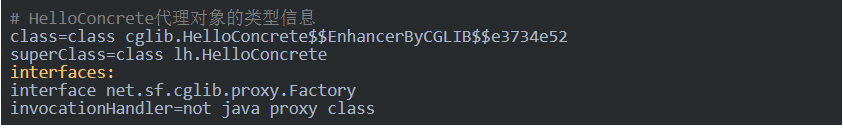


上述代码中，我们通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法，在intercept()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；

通过调用MethodProxy.invokeSuper()方法，我们将调用转发给原始对象，具体到本例，就是HelloConcrete的具体方法。CGLIG中MethodInterceptor的作用跟JDK代理中的InvocationHandler很类似，都是方法调用的中转站。

注意：对于从Object中继承的方法，CGLIB代理也会进行代理，如hashCode()、equals()、toString()等，但是getClass()、wait()等方法不会，因为它是final方法，CGLIB无法代理。

如果对CGLIB代理之后的对象类型进行深挖，可以看到如下信息：



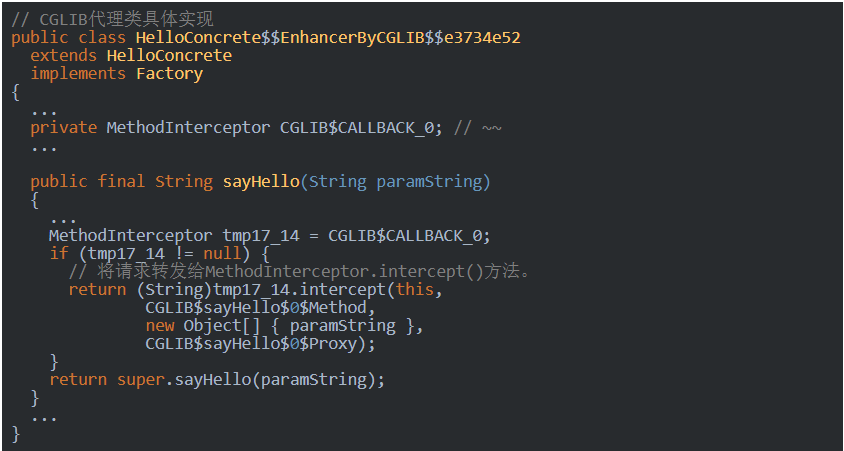
我们看到使用CGLIB代理之后的对象类型是cglib.HelloConcrete$$EnhancerByCGLIB$$e3734e52，这是CGLIB动态生成的类型；父类是HelloConcrete，印证了CGLIB是通过继承实现代理；同时实现了net.sf.cglib.proxy.Factory接口，这个接口是CGLIB自己加入的，包含一些工具方法。

注意，既然是继承就不得不考虑final的问题。我们知道final类型不能有子类，所以CGLIB不能代理final类型，遇到这种情况会抛出类似如下异常：

java.lang.IllegalArgumentException: Cannot subclass final class cglib.HelloConcrete

同样的，final方法是不能重载的，所以也不能通过CGLIB代理，遇到这种情况不会抛异常，而是会跳过final方法只代理其他方法。

如果你还对代理类cglib.HelloConcrete$$EnhancerByCGLIB$$e3734e52具体实现感兴趣，它大致长这个样子：



上述代码我们看到，当调用代理对象的sayHello()方法时，首先会尝试转发给MethodInterceptor.intercept()方法，如果没有MethodInterceptor就执行父类的sayHello()。这些逻辑没什么复杂之处，但是他们是在运行时动态产生的，无需我们手动编写。

## 结语

本文介绍了Java两种常见动态代理机制的用法和原理，JDK原生动态代理是Java原生支持的，不需要任何外部依赖，但是它只能基于接口进行代理；CGLIB通过继承的方式进行代理，无论目标对象有没有实现接口都可以代理，但是无法处理final的情况。

动态代理是Spring AOP(Aspect Orient Programming, 面向切面编程)的实现方式，了解动态代理原理，对理解Spring AOP大有帮助。