<https://www.cnblogs.com/wenxiangchen/p/12689713.html>

JDK基于接口与CGLib基于继承

**1）、JDK和CGLib的区别**

* JDK动态代理只能对实现了接口的类生成代理，而不能针对类
* CGLib是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法（继承）

**2）、Spring在选择用JDK还是CGLib的依据**

* 当Bean实现接口时，Spring就会用JDK的动态代理
* 当Bean没有实现接口时，Spring使用CGLib来实现
* 可以强制使用CGLib（在Spring配置中加入<aop:aspectj-autoproxy proxy-target-class=“true”/>）

**3）、JDK和CGLib的性能对比**

* 使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，在JDK1.6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，因为CGLib原理是动态生成被代理类的子类。
* 在JDK1.6、JDK1.7、JDK1.8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLib代理效率，只有当进行大量调用的时候，JDK1.6和JDK1.7比CGLib代理效率低一点，但是到JDK1.8的时候，JDK代理效率高于CGLib代理

**public class** MyMethodInterceptor **implements** MethodInterceptor {  
 @Override  
 **public** Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) **throws** Throwable {  
 System.***out***.println(**"======插入前置通知======"**);  
 Object object = methodProxy.invokeSuper(o, objects);  
 System.***out***.println(**"======插入后者通知======"**);  
 **return** object;  
 }  
}

*/\*\*  
 \* 接着创建一个代理类JDKDynamicProxy实现java.lang.reflect.InvocationHandler接口，重写invoke方法  
 \*/***public class** JDKDynamicProxy **implements** InvocationHandler {  
 **private** Object **target**;  
 **public** JDKDynamicProxy(Object target){  
 **this**.**target**=target;  
 }  
 **public** <T> T getProxy(){  
 **return** (T) Proxy.*newProxyInstance*(**target**.getClass().getClassLoader(), **target**.getClass().getInterfaces(), **this**);  
 }  
 @Override  
 **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  
 System.***out***.println(**"Do something before"**);  
 Object result = method.invoke(**target**, args);  
 System.***out***.println(**"Do something after"**);  
 **return** result;  
 }  
}