https://mp.weixin.qq.com/s/xvidIGCL4mIYfGni4pq9wQ

# Java代理

## 本文内容

1. **为什么需要用代理**
2. **jdk动态代理玩法详解**
3. **cglib代理常见的各种玩法详解**

代理spring中用到的挺多的，比如上篇文章中的lookup-method和replaced-method，以及后面我们要学的aop、spring中的事务、spring中解析@configuration注解等等，这些都是依靠代理来实现的，所以我们先把代理拿出来讲讲，有利于我们理解其他内容。

## 为什么要用代理

通用代理的2种实现：

* jdk动态代理
* cglib代理

## jdk动态代理详解

jdk中为实现代理提供了支持，主要用到2个类：

java.lang.reflect.Proxy  
java.lang.reflect.InvocationHandler

**jdk自带的代理使用上面有个限制，只能为接口创建代理类，如果需要给具体的类创建代理类，需要用后面要说的cglib**

### java.lang.reflect.Proxy

这是jdk动态代理中主要的一个类，里面有一些静态方法会经常用到，我们来熟悉一下：

#### getProxyClass方法

为指定的接口创建代理类，返回代理类的Class对象

public static Class<?> getProxyClass(ClassLoader loader,  
Class<?>... interfaces)

参数说明：

* loader：定义代理类的类加载器
* interfaces：指定需要实现的接口列表，创建的代理默认会按顺序实现interfaces指定的接口

#### newProxyInstance方法

创建代理类的实例对象

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h)

这个方法先为指定的接口创建代理类，然后会生成代理类的一个实例，最后一个参数比较特殊，是InvocationHandler类型的，这个是个接口如下：

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable;

上面方法会返回一个代理对象，当调用代理对象的任何方法的时候，会就被InvocationHandler接口的invoke方法处理，所以主要代码需要写在invoke方法中，稍后会有案例细说。

#### isProxy方法

判断指定的类是否是一个代理类

public static boolean isProxyClass(Class<?> cl)

#### getInvocationHandler方法

获取代理对象的InvocationHandler对象

public static InvocationHandler getInvocationHandler(Object proxy) throws IllegalArgumentException

**上面几个方法大家熟悉一下，下面我们来看创建代理具体的2种方式。**

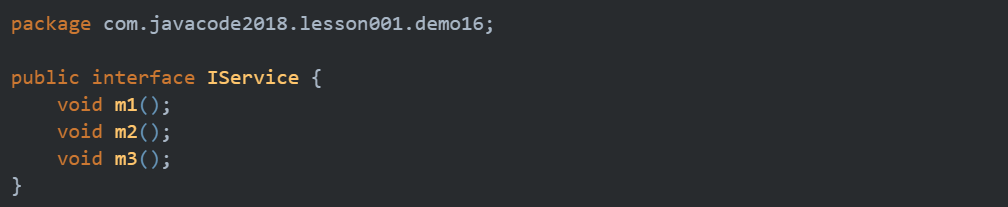
### 创建代理：方式一

#### 步骤

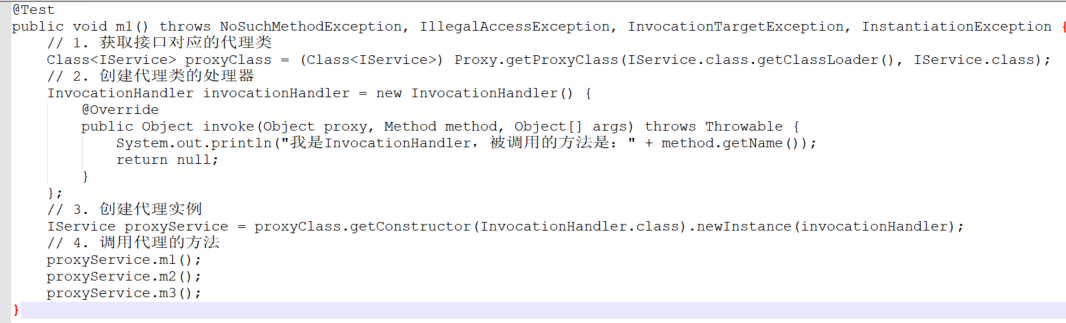
1.调用Proxy.getProxyClass方法获取代理类的Class对象  
2.使用InvocationHandler接口创建代理类的处理器  
3.通过代理类和InvocationHandler创建代理对象   
4.上面已经创建好代理对象了，接着我们就可以使用代理对象了

#### 案例

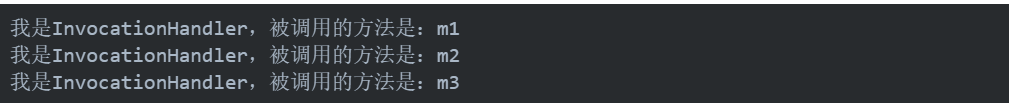
##### 先来个接口IService



##### 创建IService接口的代理对象



##### 运行输出



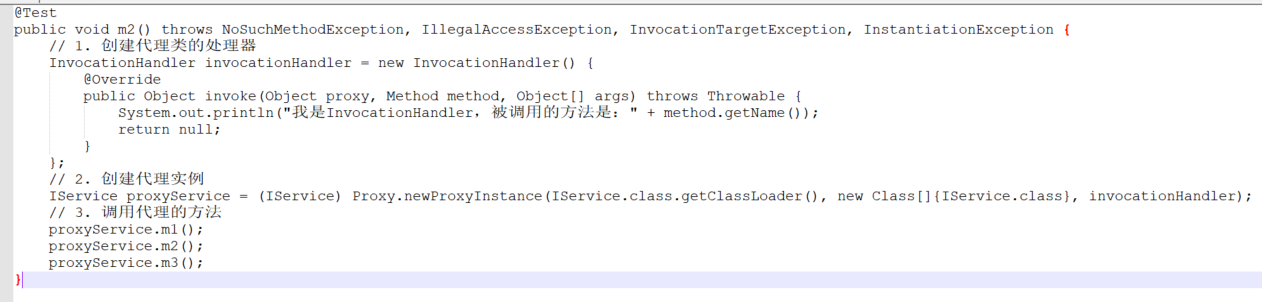
### 创建代理：方式二

创建代理对象有更简单的方式。

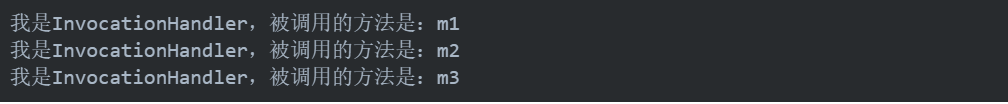
#### 步骤

1. 使用InvocationHandler接口创建代理类的处理器  
   2.使用Proxy类的静态方法newProxyInstance直接创建代理对象  
   3.使用代理对象

#### 案例



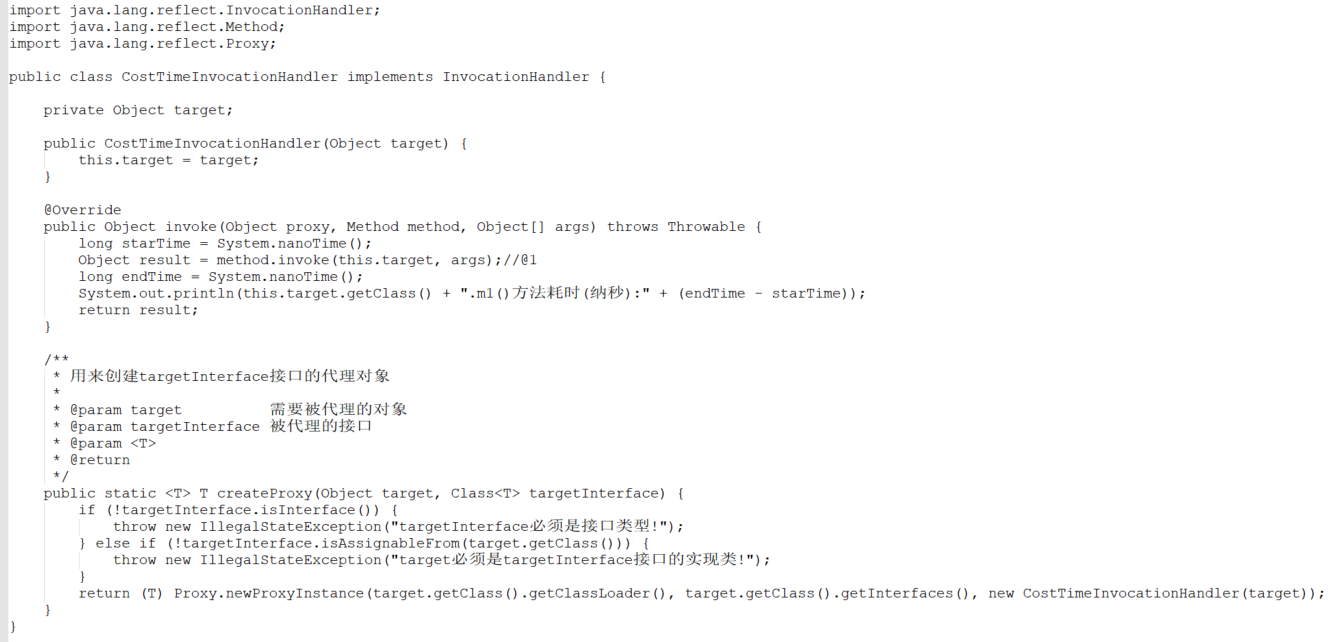
运行输出：



### 案例：任意接口中的方法耗时统计

**下面我们通过jdk动态代理实现一个通用的代理，解决统计所有接口方法耗时的问题。**

主要的代码在代理处理器InvocationHandler实现上面，如下：



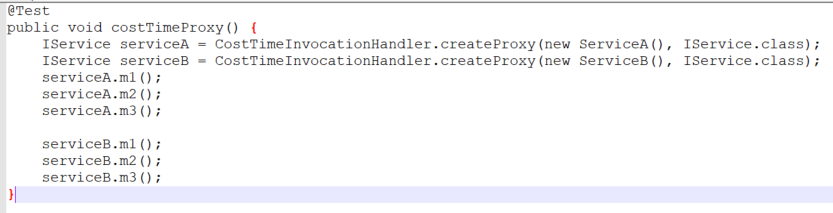
上面主要是createProxy方法用来创建代理对象，2个参数：

target：目标对象，需要实现targetInterface接口

targetInterface：需要创建代理的接口

invoke方法中通过method.invoke(this.target, args)调用目标方法，然后统计方法的耗时。

**测试用例**

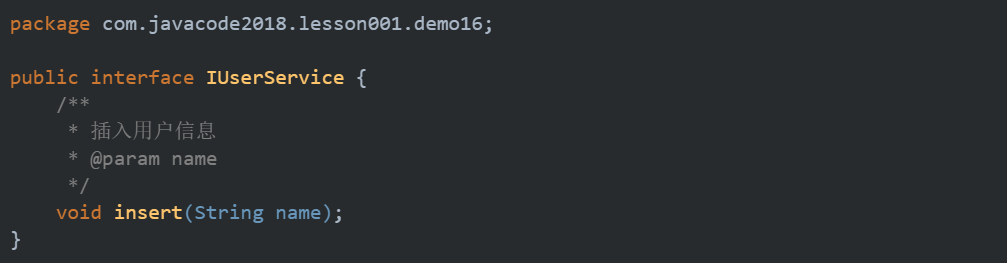


**运行输出**

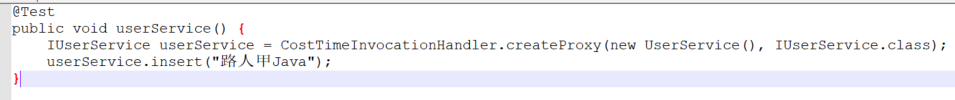


**我们再来的接口，也需要统计耗时的功能，此时我们无需去创建新的代理类即可实现同样的功能，如下：**

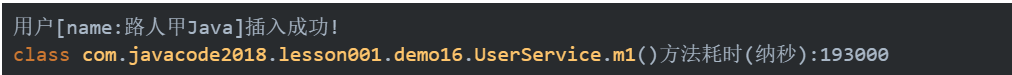
**IUserService接口**



**测试用例**



**运行输出：**



**上面当我们创建一个新的接口的时候，不需要再去新建一个代理类了，只需要使用**CostTimeInvocationHandler.createProxy**创建一个新的代理对象就可以了，方便了很多。**

### Proxy使用注意

* **jdk中的Proxy只能为接口生成代理类，如果你想给某个类创建代理类，那么Proxy是无能为力的，此时需要我们用到下面要说的cglib了。**
* **Proxy类中提供的几个常用的静态方法大家需要掌握**
* **通过Proxy创建代理对象，当调用代理对象任意方法时候，会被InvocationHandler接口中的invoke方法进行处理，这个接口内容是关键**

## cglib代理详解

### 什么是cglib

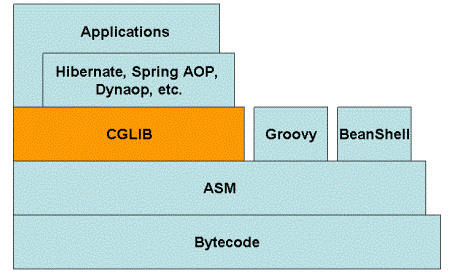
jdk动态代理只能为接口创建代理，使用上有局限性。实际的场景中我们的类不一定有接口，此时如果我们想为普通的类也实现代理功能，我们就需要用到cglib来实现了。

cglib是一个强大、高性能的字节码生成库，它用于在运行时扩展Java类和实现接口；本质上它是通过动态的生成一个子类去覆盖所要代理的类（非final修饰的类和方法）。Enhancer可能是CGLIB中最常用的一个类，和jdk中的Proxy不同的是，Enhancer既能够代理普通的class，也能够代理接口。Enhancer创建一个被代理对象的子类并且拦截所有的方法调用（包括从Object中继承的toString和hashCode方法）。Enhancer不能够拦截final方法，例如Object.getClass()方法，这是由于Java final方法语义决定的。基于同样的道理，Enhancer也不能对final类进行代理操作。

CGLIB作为一个开源项目，其代码托管在github，地址为：

<https://github.com/cglib/cglib>

### cglib组成结构

****

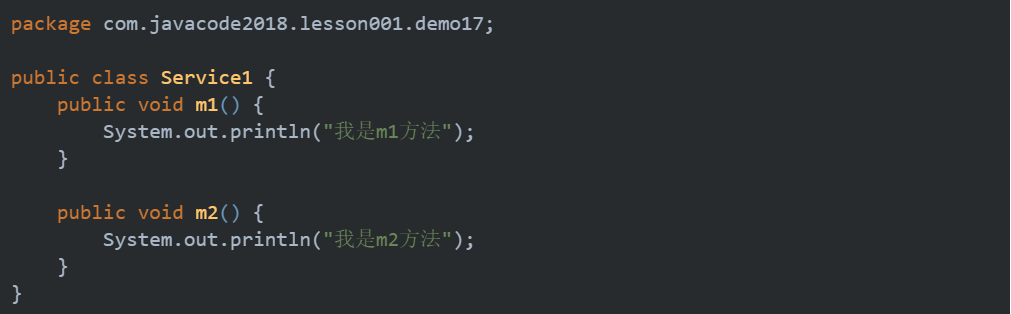
CGLIB底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。除了CGLIB库外，脚本语言（如Groovy和BeanShell）也使用ASM生成字节码。ASM使用类似SAX的解析器来实现高性能。我们不鼓励直接使用ASM，因为它需要对Java字节码的格式足够的了解。

spring已将第三方cglib jar包中所有的类集成到spring自己的jar包中，本系列内容都是和spring相关的，为了方便，我们直接使用spring内部已集成的来讲解。

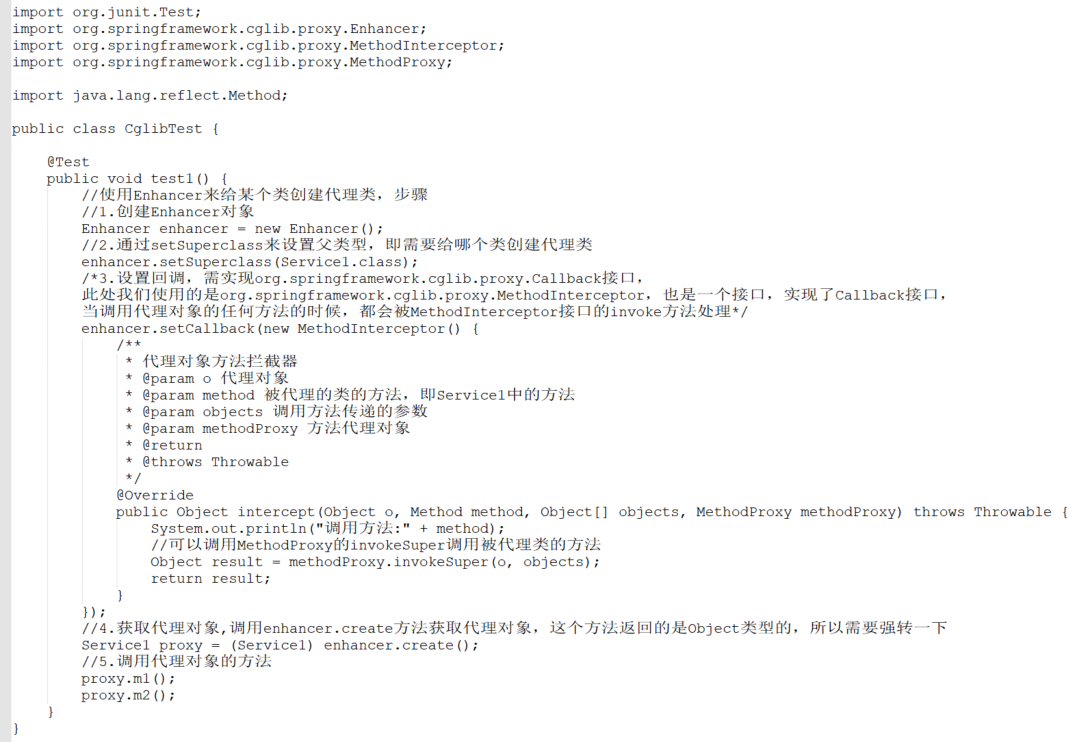
### 5个案例来演示cglib常见的用法

#### 案例1：拦截所有方法（MethodInterceptor）

创建一个具体的类，如下：



下面我们为这个类创建一个代理，代理中实现打印每个方法的调用日志。



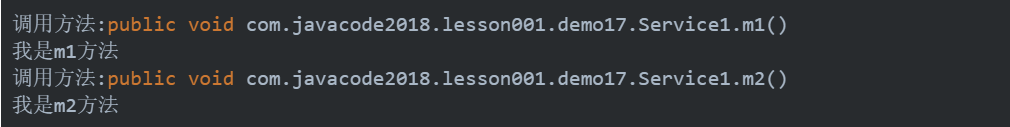
上面代码中的注释很详细，列出了给指定的类创建代理的具体步骤，整个过程中主要用到了Enhancer类和MethodInterceptor接口。

enhancer.setSuperclass用来设置代理类的父类，即需要给哪个类创建代理类，此处是Service1

enhancer.setCallback传递的是MethodInterceptor接口类型的参数，MethodInterceptor接口有个intercept方法，这个方法会拦截代理对象所有的方法调用。

还有一个重点是Object result = methodProxy.invokeSuper(o, objects);可以调用被代理类，也就是Service1类中的具体的方法，从方法名称的意思可以看出是调用父类，实际对某个类创建代理，cglib底层通过修改字节码的方式为Service1类创建了一个子类。

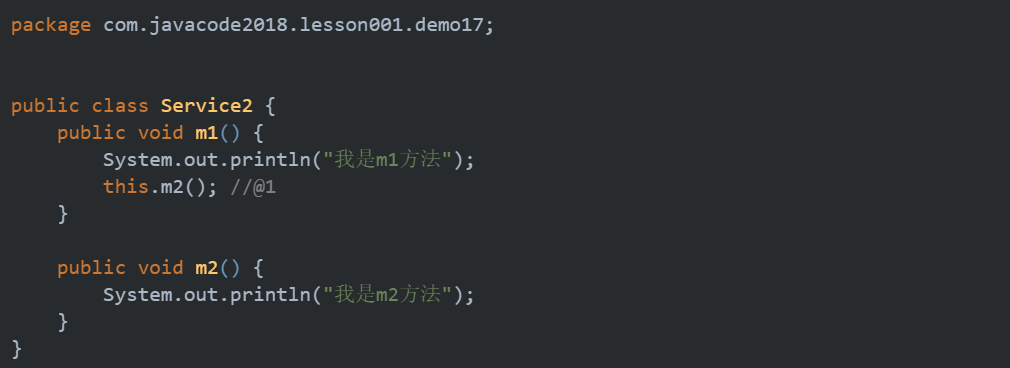
运行输出：



从输出中可以看出Service1中的2个方法都被MethodInterceptor中的invoke拦截处理了。

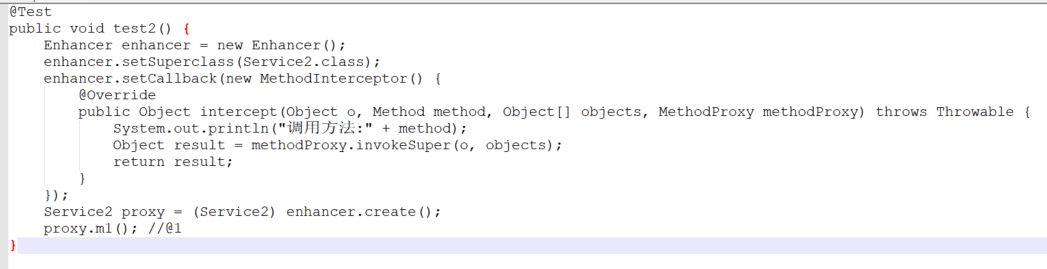
#### 案例2：拦截所有方法（MethodInterceptor）

在创建一个类，如下：

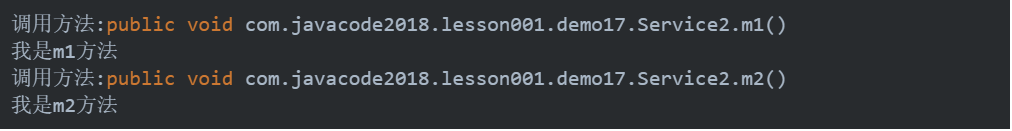


这个类和上面的Service1类似，有点不同是@1，在m1方法中调用了m2方法。

下面来采用案例1中同样的方式来给Service2创建代理，如下：

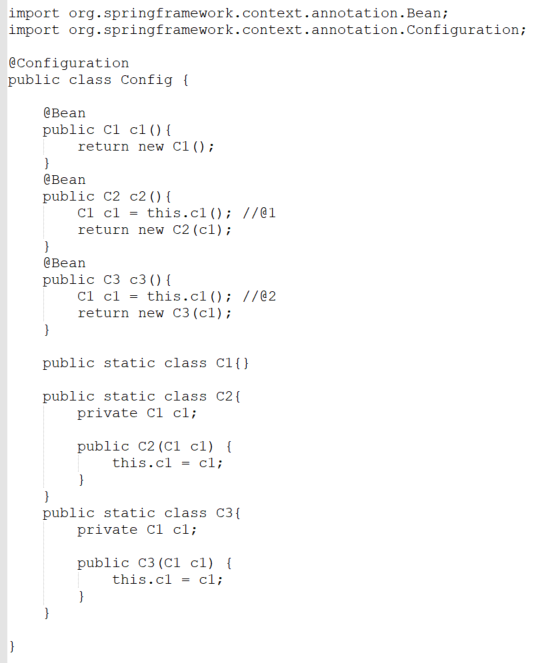


注意上面@1的代码，只调用了m1方法，看一下输出效果：



从输出中可以看出m1和m2方法都被拦截器处理了，而m2方法是在Service1的m1方法中调用的，也被拦截处理了。

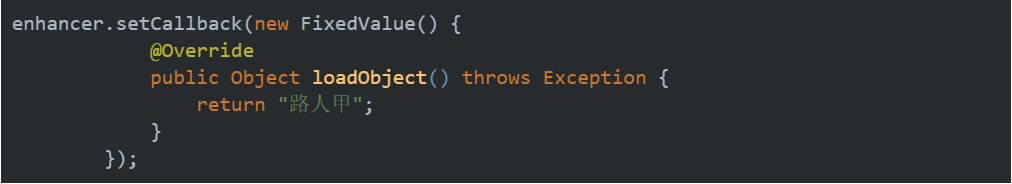
spring中的@configuration注解就是采用这种方式实现的，给大家上个@configuration案例眼熟一下：



上面代码中C2和C3依赖于C1，都是通过构造器注入C1，注意代码中的@1和@2都是调用c1方法获取容器中的C1，如何确保多次获取的C1都是一个的？这个地方就是使用cglib代理拦截@Bean注解的方法来实现的。

#### 案例3：拦截所有方法并返回固定值（FixedValue）

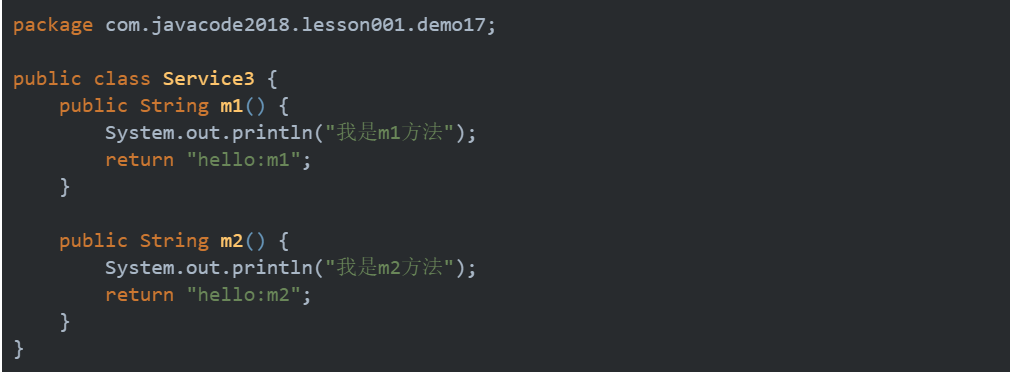
当调用某个类的任何方法的时候，都希望返回一个固定的值，此时可以使用FixedValue接口，如下：



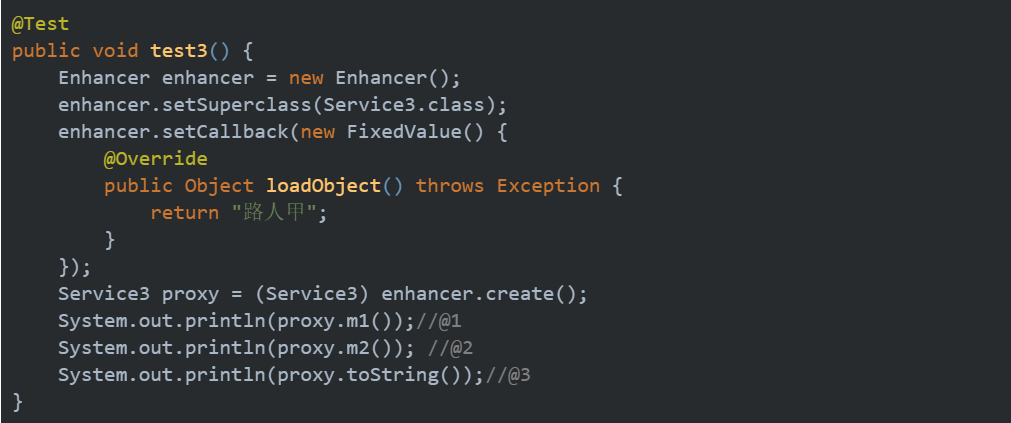
上面创建的代理对象，调用其任意方法返回的都是"路人甲"。

案例代码如下：

创建一个类Service3，如下：



对用的测试用例：



@1、@2、@3调用了代理对象的3个方法，运行输出：

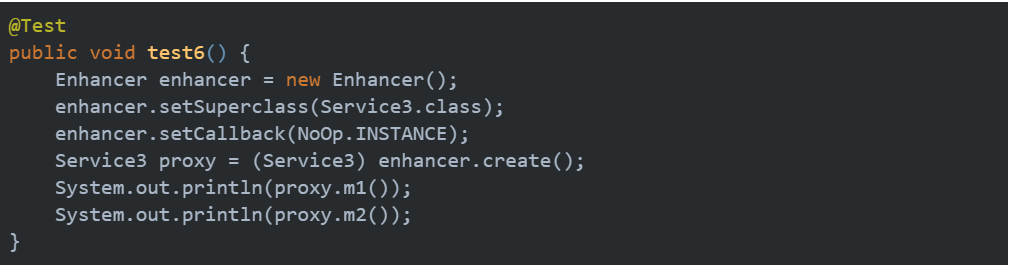
运行输出：



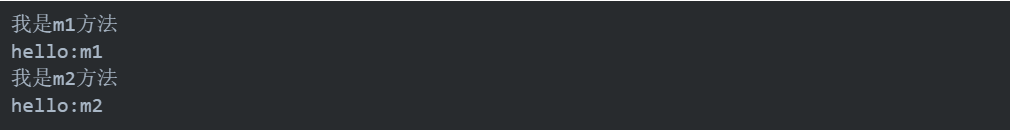
可以看出输出的都是一个固定的值。

#### 案例4：直接放行，不做任何操作（NoOp.INSTANCE）

Callback接口下面有个子接口org.springframework.cglib.proxy.NoOp，将这个作为Callback的时候，被调用的方法会直接放行，像没有任何代理一样，感受一下效果：



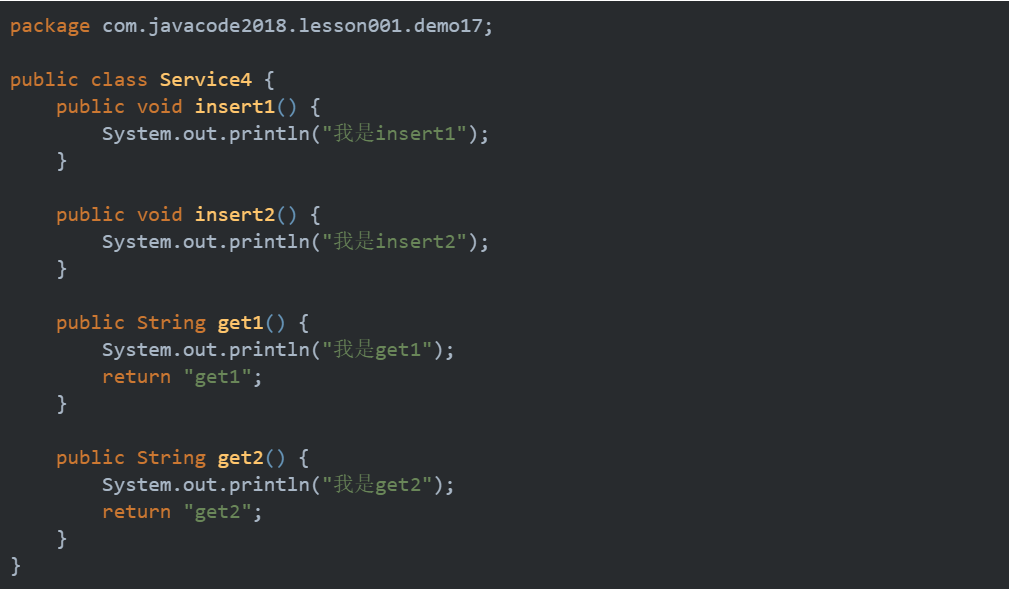
运行输出：



从输出中可以看出，被调用的方法没有被代理做任何处理，直接进到目标类Service3的方法中了。

#### 案例5：不同的方法使用不同的拦截器（CallbackFilter）

有个类如下：

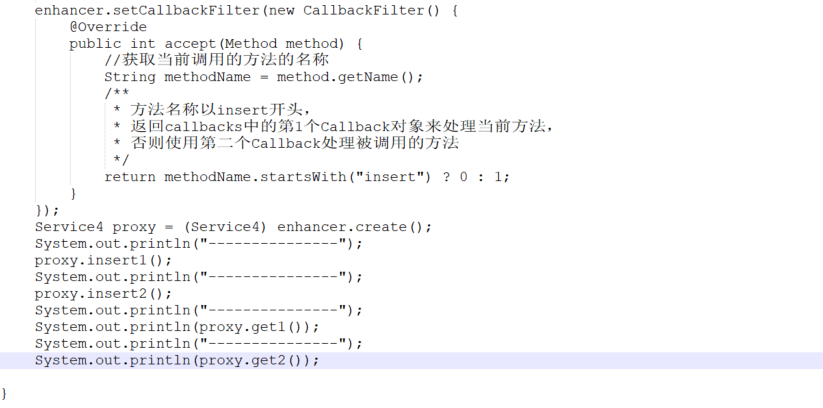


需求，给这个类创建一个代理需要实现下面的功能：

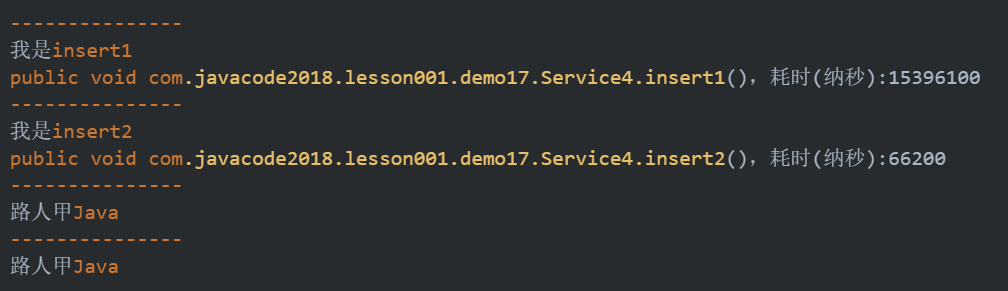
1. 以insert开头的方法需要统计方法耗时
2. 以get开头的的方法直接返回固定字符串`欢迎和【路人甲java】一起学spring！`

下来看代码，然后再解释：





运行输出：



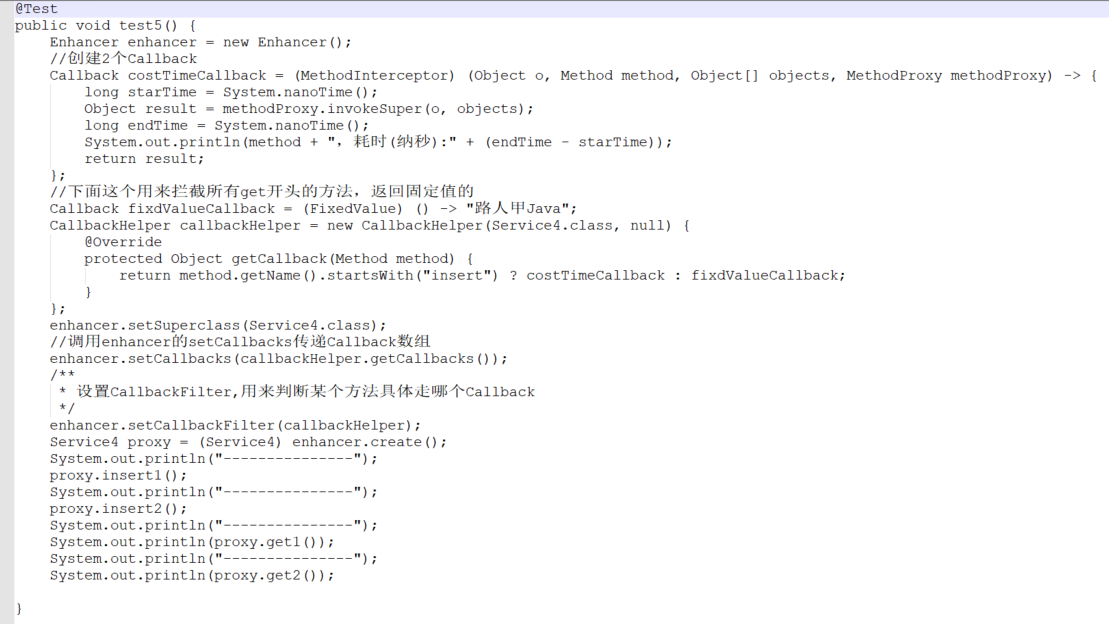
码说明：

由于需求中要对不同的方法做不同的处理，所以需要有2个Callback对象，当调用代理对象的方法的时候，具体会走哪个Callback呢，此时会通过CallbackFilter中的accept来判断，这个方法返回callbacks数组的索引。

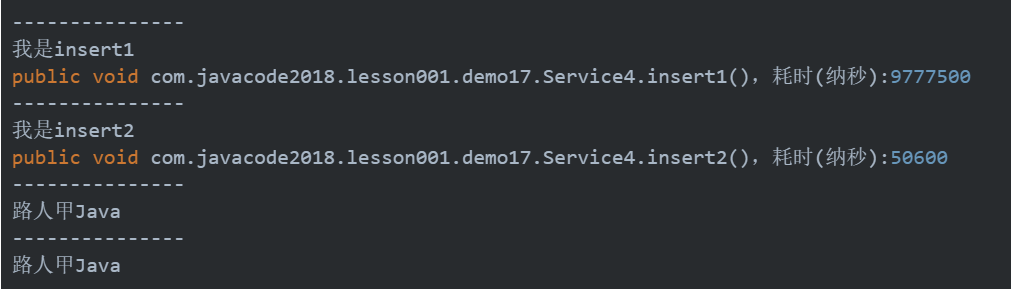
上面这个案例还有一种简单的实现，见案例6

#### 案例6：对案例5的优化（CallbackHelper）

cglib中有个CallbackHelper类，可以对案例5的代码进行有环，CallbackHelper类相当于对一些代码进行了封装，方便实现案例5的需求，实现如下：



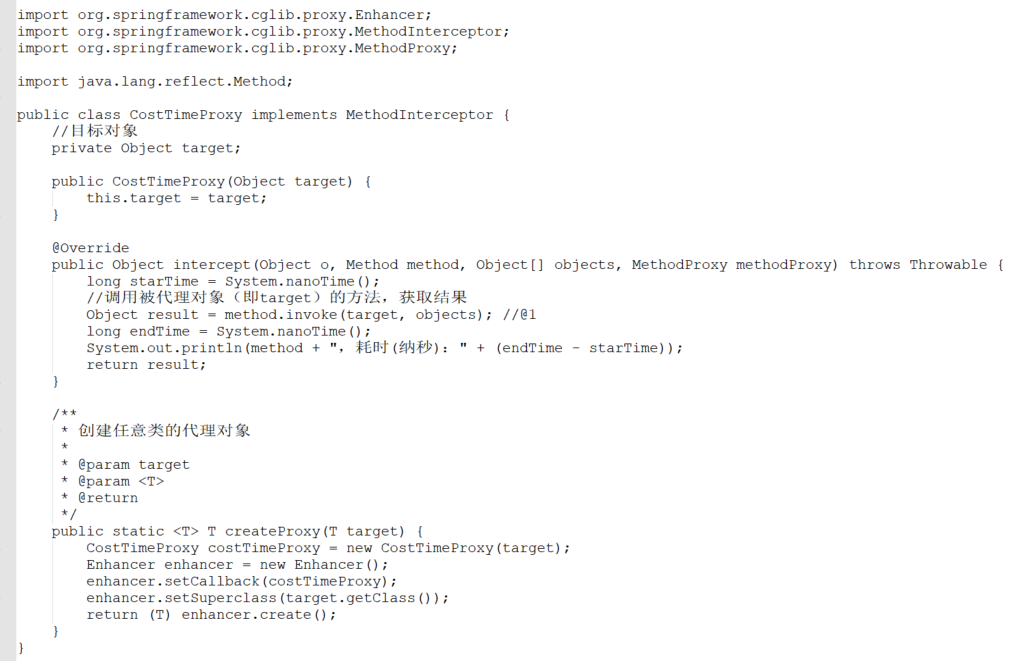
运行输出：



输出效果和案例4一模一样的，上面重点在于CallbackHelper，里面做了一些封装，有兴趣的可以去看一下源码，比较简单。

#### 案例7：实现通用的统计任意类方法耗时代理类

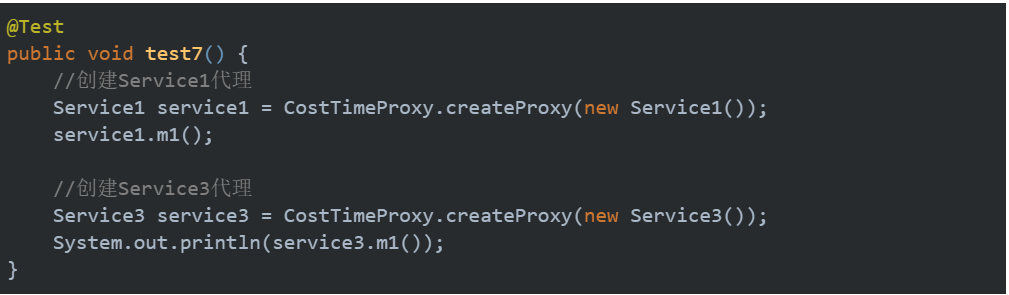
直接上代码，比较简单，如下：



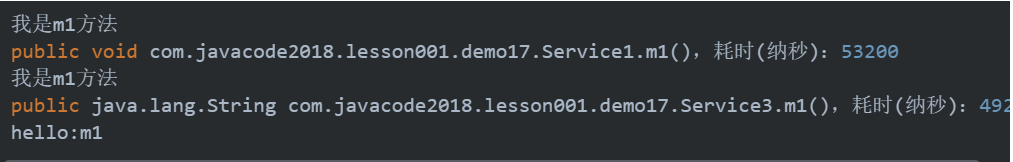
我们可以直接使用上面的静态方法createProxy来为目标对象target创建一个代理对象，被代理的对象自动实现方法调用耗时统计。

@1：调用被代理对象的方法获取真正的结果。

使用非常简单，来个测试用例，如下：



运行输出：



## CGLIB和Java动态代理的区别

**1、Java动态代理只能够对接口进行代理，不能对普通的类进行代理**（因为所有生成的代理类的父类为Proxy，Java类继承机制不允许多重继承）；CGLIB能够代理普通类；

2、 Java动态代理使用Java原生的反射API进行操作，在生成类上比较高效；CGLIB使用ASM框架直接对字节码进行操作，在类的执行过程中比较高效。