要说动态代理，必须先聊聊静态代理

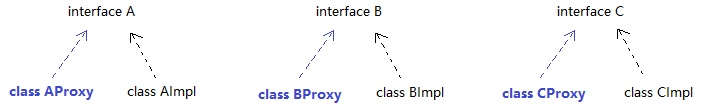
**静态代理**

**假设现在项目经理有一个需求：在项目现有所有类的方法前后打印日志**

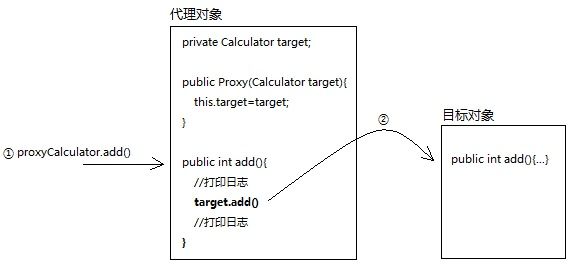
**你如何在不修改已有代码的前提，完成这个需求？**

**我首先想到的是静态代理。具体做法是**

**1.为现有的每一个类都编写一对应代理类，并且让它实现和目标类相同的接口（假设都有**

****

**2.在创建代理对象时，通过构造器塞入一个目标对象，然后在代理对象的方法内部调用目标对象同名方法，并在调用前后打印日志。也就是说，代理对象=增强代码+目标对象（原对象有了代理对象后，就不用原对象了**

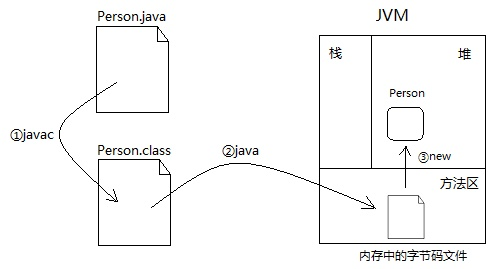
****

**静态代理的缺陷**

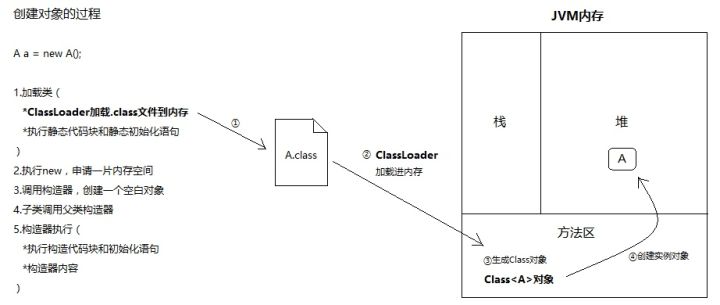
**程序员要手动为每一个目标类编写对应的代理类。如果当前系统已经有成百上千个类，工作量太大了。所以，现在我们的努力方向是：如何少写或者不写代理类，却能完成代理功能？**

**复习对象的创**

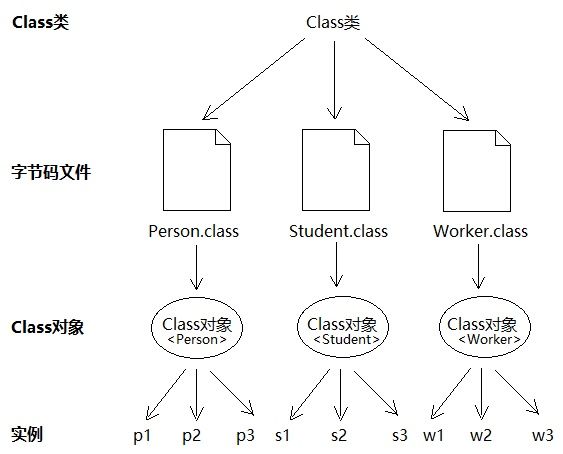
**很多初学Java的朋友眼中创建对象的过程**

****

**实际上可以换个角度，也说得**

****

**所谓的Class对象，是Class类的实例，而Class类是描述所有类的，比如Person类，Student**

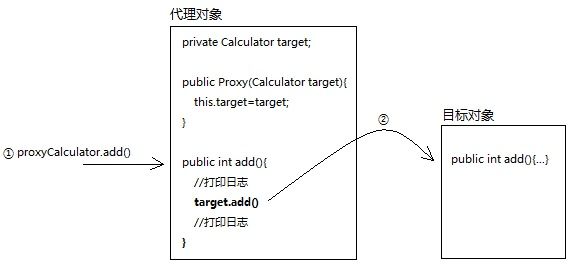
****

**可以看出，要创建一个实例，最关键的就�得到对应的Class对象�只不过对于初学者来说，new这个关键字配合构造方法，实在太好用了，底层隐藏了太多细节，一�?Personp=newPerson();直接把对象返回给你了。我自己刚开始学Java时，也没意识到Class对象的存在**

**分析到这里，貌似有了思路**

**能否不写代理类，而直接得到代理Class对象，然后根据它创建代理实例（反射）**

**Class对象包含了一个类的所有信息，比如构造器、方法、字段等。如果我们不写代理类，这些信息从哪获取呢？苦思冥想，突然灵光一现：代理类和目标类理应实现同一组接口之所以实现相同接口，是为了尽可能保证代理对象的内部结构和目标对象一致，这样我们对代理对象的操作最终都可以转移到目标对象身上，代理对象只需专注于增强代码的编写还是上面这幅图：**

****

**所以，可以这样说：接口拥有代理对象和目标对象共同的类信息。所以，我们可以从接口那得到理应由代理类提供的信息。但是别忘了，接口是无法创建对象的，怎么办？**

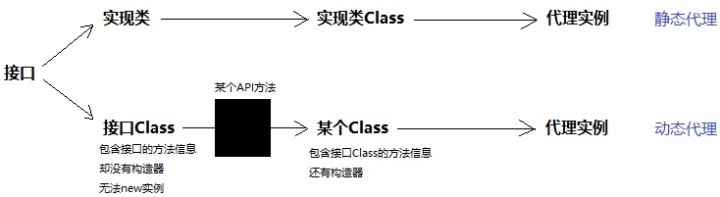
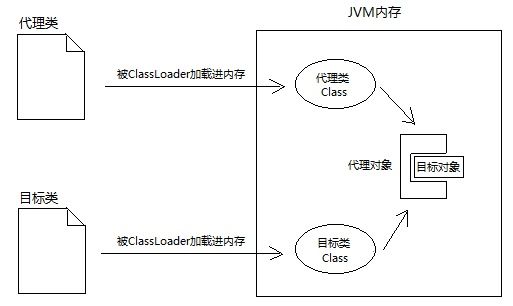
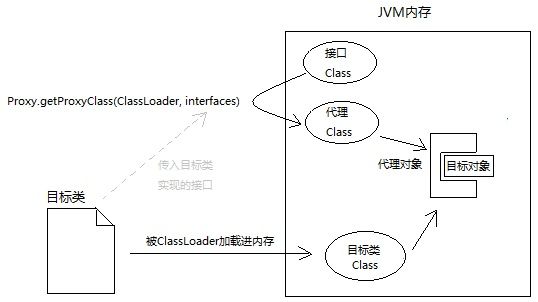
**动态代**

**JDK提供了java.lang.reflect.InvocationHandler接口java.lang.reflect.Proxy类，这两个类相互配合，入口是Proxy，所以我们先聊它**

**Proxy有个静态方法：getProxyClass(ClassLoader,interfaces)，只要你给它传入类加载器和一组接口，它就给你返回代理Class对象**

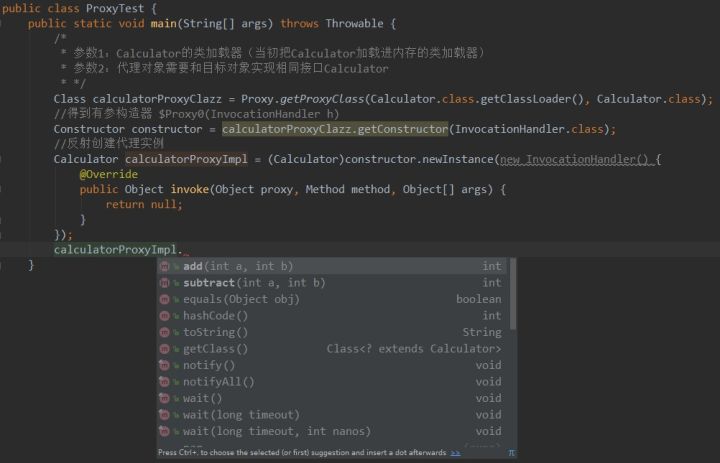
**用通俗的话说，getProxyClass()这个方法，会从你传入的接口Class中，“拷贝”类结构信息到一个新的Class对象中，但新的Class对象带有构造器，是可以创建对象的。打个比方，一个大内太监（接口Class），空有一身武艺（类信息），但是无法传给后人。现在江湖上有个妙手神医（Proxy类），发明了克隆大法（getProxyClass），不仅能克隆太监的一身武艺，还保留了小DD（构造器（这到底是道德の沦丧，还是人性的扭曲，欢迎走进动态代理）**

**所以，一旦我们明确接口，完全可以通过接口的Class对象，创建一个代理Class，通过代理Class即可创建代理对象�**

**大体思路静态代动态代**

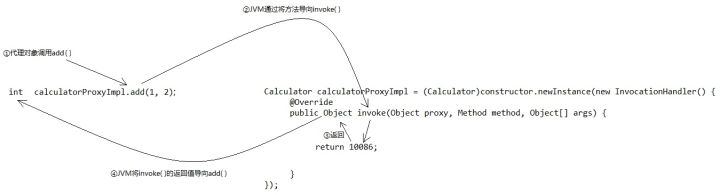
**所以，按我理解，Proxy.getProxyClass()这个方法的本质就是：以Class造Class**

**有了Class对象，就很好办了，具体看代码**

****

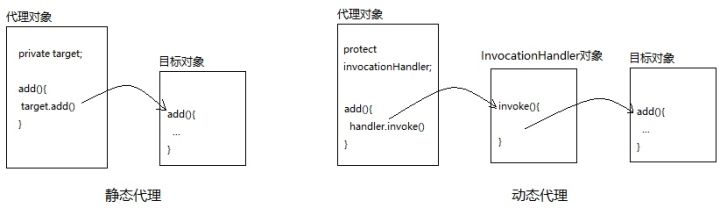
**完美**

**根据代理Class的构造器创建对象时，需要传入InvocationHandler。每次调用代理对象的方法，最终都会调用InvocationHandler的invoke()方法**

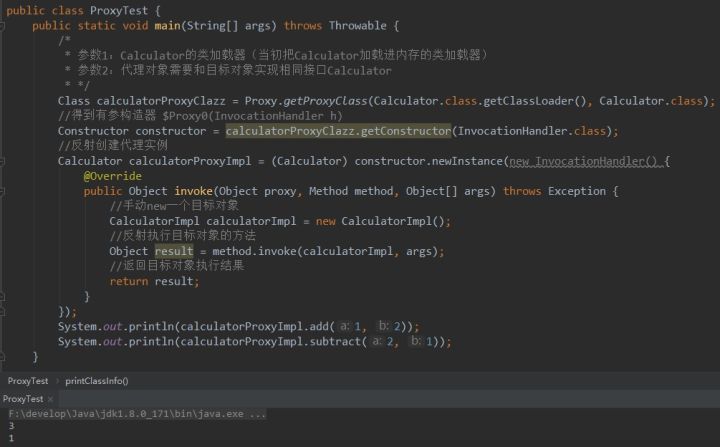
****

**怎么做到的呢�**

**上面不是说了吗，根据代理Class的构造器创建对象时，需要传入InvocationHandler通过构造器传入一个引用，那么必然有个成员变量去接收没错，代理对象的内部确实有个成员变量invocationHandler，而且代理对象的每个方法内部都会调用handler.invoke()！InvocationHandler对象成了代理对象和目标对象的桥梁，不像静态代理这么直接**

****

**大家仔细看上图右侧的动态代理，我在invocationHandler的invoke()方法中并没有写目标对象。因为一开始invocationHandler的invoke()里确实没有目标对象，需要我们手动new**

****

**但这种写法不够优雅，属于硬编码。我这次代理A对象，下次想代理B对象还要进来改invoke()方法，太差劲了。改进一下，让调用者把目标对象作为参数传进来：**

**publicclassProxyTest{**

**publicstaticvoidmain(String[]args)throwsThrowable{**

**CalculatorImpltarget=newCalculatorImpl();**

***//传入目标对象***

***//目的�?.根据它实现的接口生成代理对象2.代理对象调用目标对象方法***

**CalculatorcalculatorProxy=(Calculator)getProxy(target);**

**calculatorProxy.add(1,2);**

**calculatorProxy.subtract(2,1);**

**}**

**privatestaticObjectgetProxy(finalObjecttarget)throwsException{**

***//参数1：随便找个类加载器给它，参数2：目标对象实现的接口，让代理对象实现相同接口***

**ClassproxyClazz=Proxy.getProxyClass(target.getClass().getClassLoader(),target.getClass().getInterfaces());**

**Constructorconstructor=proxyClazz.getConstructor(InvocationHandler.class);**

**Objectproxy=constructor.newInstance(newInvocationHandler(){**

**@Override**

**publicObjectinvoke(Objectproxy,Methodmethod,Object[]args)throwsThrowable{**

**System.out.println(method.getName()+"方法开始执�?..");**

**Objectresult=method.invoke(target,args);**

**System.out.println(result);**

**System.out.println(method.getName()+"方法执行结束...");**

**returnresult;**

**}**

**});**

**returnproxy;**

**}**

**}**

**这样就非常灵活，非常优雅了。无论现在系统有多少类，只要你把实例传进来，getProxy()都能给你返回对应的代理对象。就这样，我们完美地跳过了代理类，直接创建了代理对象**

**不过实际编程中，一般不用getProxyClass()，而是使用Proxy类的另一个静态方法：Proxy.newProxyInstance()，直接返回代理实例，连中间得到代理Class对象的过程都帮你隐藏**

**publicclassProxyTest{**

**publicstaticvoidmain(String[]args)throwsThrowable{**

**CalculatorImpltarget=newCalculatorImpl();**

**CalculatorcalculatorProxy=(Calculator)getProxy(target);**

**calculatorProxy.add(1,2);**

**calculatorProxy.subtract(2,1);**

**}**

**privatestaticObjectgetProxy(finalObjecttarget)throwsException{**

**Objectproxy=Proxy.newProxyInstance(**

**target.getClass().getClassLoader(),*/\*类加载器\*/***

**target.getClass().getInterfaces(),*/\*让代理对象和目标对象实现相同接口\*/***

**newInvocationHandler(){*/\*代理对象的方法最终都会被JVM导向它的invoke方法\*/***

**publicObjectinvoke(Objectproxy,Methodmethod,Object[]args)throwsThrowable{**

**System.out.println(method.getName()+"方法开始执�?..");**

**Objectresult=method.invoke(target,args);**

**System.out.println(result);**

**System.out.println(method.getName()+"方法执行结束...");**

**returnresult;**

**}**

**}**

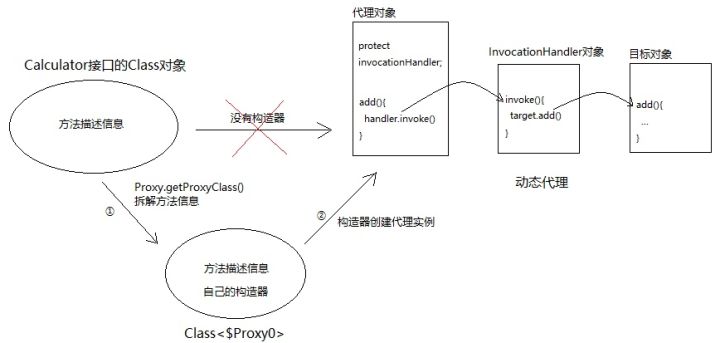
**);**

**returnproxy;**

**}**

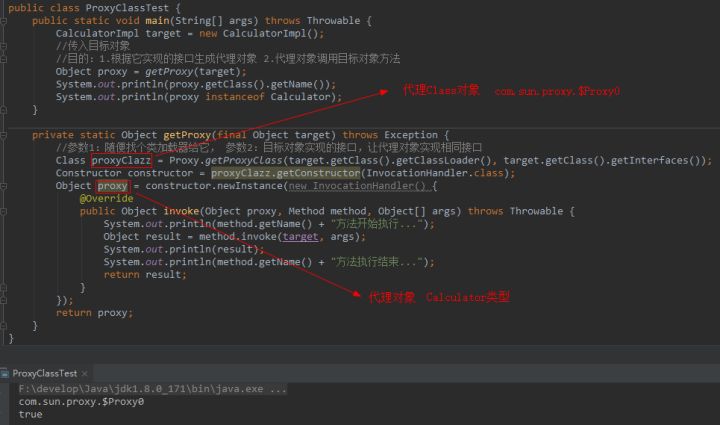
**}**

**现在，我想题主应该能看懂动态代理了**

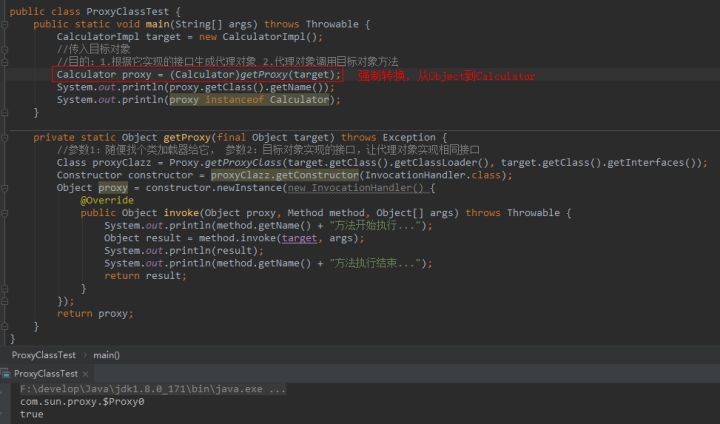
****

**最后讨论一下代理对象是什么类型**

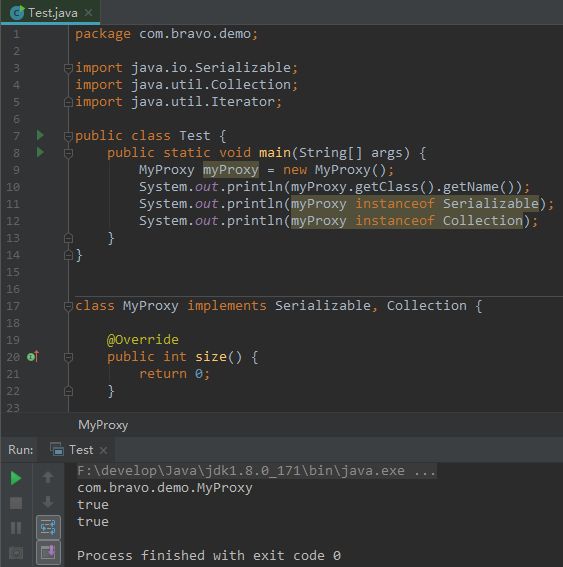
**首先，请区分两个概念：代理Class对象和代理对象**

****

**单从名字看，代理Class和Calculator的接口确实相去甚远，但是我们却能将代理对象赋值给接口类型**

****

**千万别觉得名字奇怪，就怀疑它不能用接口接收，只要实现该接口就是该类型**

**代理对象的本质就是：和目标对象实现相同接口的实例。代理Class可以叫任何名字，whatever，只要它实现某个接口，就能成为该接口类型**

**我写了一个MyProxy类，那么它的Class名字必然叫MyProxy但这和能否赋值给接口没有任何关系由于它实现了Serializable和Collection，所以myProxy（代理实例）同时是这两个接口的类型**

**小结**

**我想了个很骚的比喻，希望能解释清楚：**

**接口Class对象是大内太监，里面的方法和字段比做他的一身武艺，但是他没有小DD（构造器），所以不能new实例。一身武艺后继无人**

**那怎么办呢**

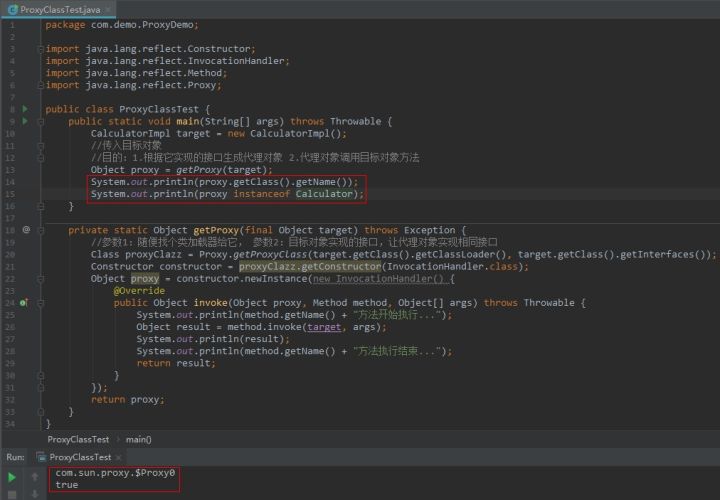
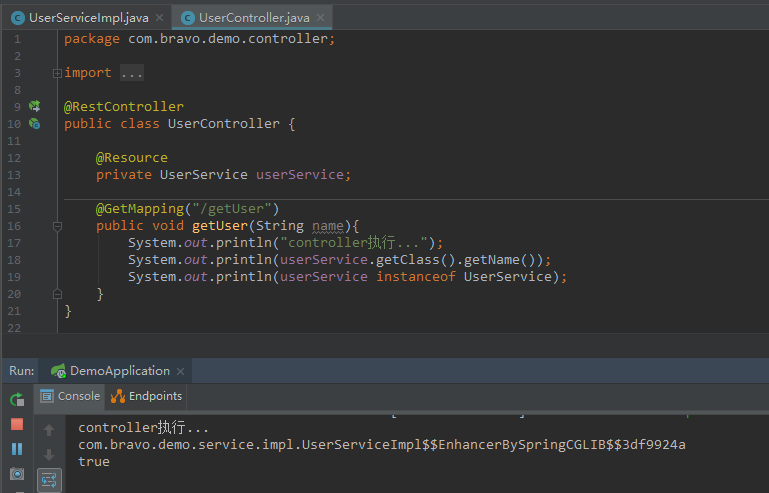
**正常途径（implements）：**

**写一个类，实现该接口。这个就相当于大街上拉了一个人，认他做干爹。一身武艺传给他，只是比他干爹多了小DD，可以new实例**

**非正常途径（动态代理）�**

**通过妙手圣医Proxy的克隆大法（Proxy.getProxyClass()），克隆一个Class，但是有小DD。所以这个克隆人Class可以创建实例，也就是代理对象**

**代理Class其实就是附有构造器的接口Class，一样的类结构信息，却能创建实例**

**JDK动态代理生成的实例CGLib动态代理生成的实例**

**如果说继承的父类是亲爹（只有一个），那么实现的接口是干爹（可以有多个）�**

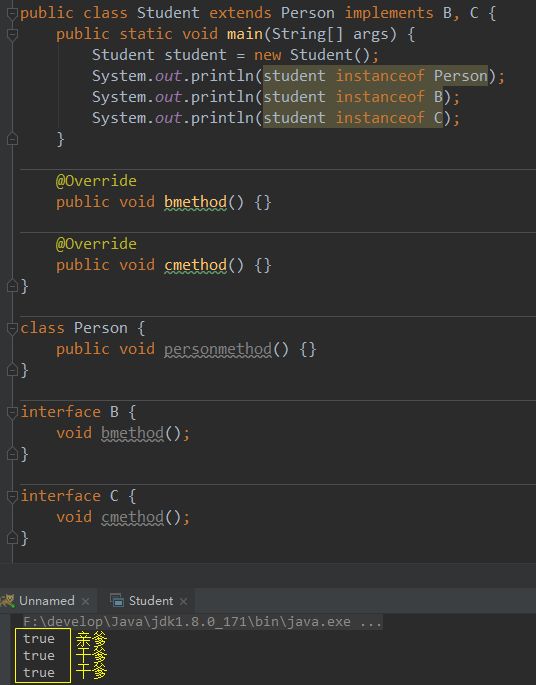
**实现接口是一个类认干爹的过程。接口无法创建对象，但实现该接口的类可以�**

**比如**

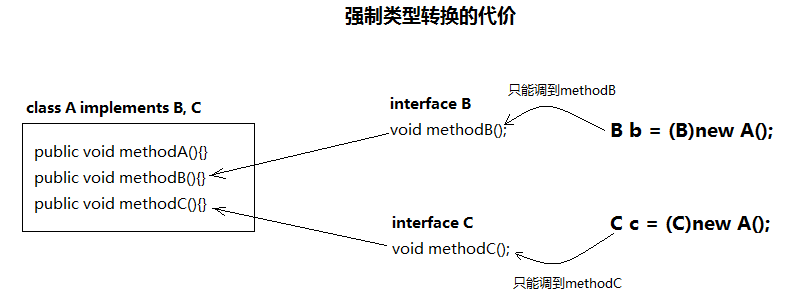
**Class Student extends Person implements A,B**

**这个类new一个实例出来，你问它：你爸爸是谁啊？它会告诉你：我只有一个爸爸Person�**

**但是student instance of A interface，或者student instance of B interface，它会告诉你两个都是它干爹（true），都可以用来接收它**

****

**然而，凡是有利必有弊�**

****

**也就是说，动态代理生成的代理对象，最终都可以用接口接收，和目标对象一起形成了多态，可以随意切换展示不同的功能。但是切换的同时，只能使用该接口定义的方法**

**关于类加载器**

**初学者可能对诸如“字节码文件”、Class对象比较陌生。所以这里花一点点篇幅介绍一下类加载器的部分原理。如果我们要定义类加载器，需要继承ClassLoader类，并覆盖findClass()方法?**

**@Override**

**Public Class<?> findClass(Stringname) throws ClassNotFoundException{**

**try{**

**/\*自己另外写一个getClassData()**

**通过IO流从指定位置读取xxx.class文件得到字节数组\*/**

**byte[]datas=getClassData(name);**

**if(datas==null){**

**thrownewClassNotFoundException("类没有找到："+name);**

**}**

**//调用类加载器本身的defineClass()方法，由字节码得到Class对象**

**returnthis.defineClass(name,datas,0,datas.length);**

**}catch(IOExceptione){**

**e.printStackTrace();**

**thrownewClassNotFoundException("类找不到�?+name);**

**}**

**}**

**所以，这就是类加载之所以能把xxx.class文件加载进内存，并创建对应Class对象的深层原因。具体文章可以参考基友写的另一篇：**[**请叫我程序猿大人：好怕怕的类加载器**](https://zhuanlan.zhihu.com/p/54693308)