反射与代理

1.反射

写这篇文章的时候距离学反射已经过去1年了，在一年的学习过程中（学习框架啊，阅读框架源码啊等等）都多多少少有反射的影子，要不怎么说**反射是框架的灵魂**呢~~

1.1 什么是反射

反射（Reflection）是Java语言的特性之一，它允许**运行中**的Java程序获取自身的信息，并且可以操作类或对象的内部属性。

通过反射机制，可以在运行时访问Java对象的属性、方法等。

1.2 反射的应用场景

- **开发框架**：反射最重要的用途就是开发各种通用框架。拿我们熟悉的spring framework框架作为例子，我们知道原始的bean定义是不支持注解开发的，需要在xml文件中进行配置
- **jdk动态代理**：在AOP中需要拦截特定的方法通常会选择动态代理（基于接口的jdk动态代理），这时就需要使用反射技术来实现了。
- **注解**：注解本身仅仅是起到标记作用，它需要利用反射机制，根据注解标记去调用注解解释器，执行行为。

1.3 反射的具体使用

见有道云笔记。

1.3.1 java.lang.reflect 包

Java 中的 java.lang.reflect 包提供了反射功能。java.lang.reflect 包中的类都没有 public 构造方法。

java.lang.reflect 包的核心接口和类如下：

- Member 接口 - 反映关于单个成员(字段或方法)或构造函数的标识信息。
- **Field 类** - 提供一个类的域的信息以及访问类的域的接口。
- **Method 类** - 提供一个类的方法的信息以及访问类的方法的接口。
- **Constructor 类** - 提供一个类的构造函数的信息以及访问类的构造函数的接口。
- Array 类 - 该类提供动态地生成和访问 JAVA 数组的方法。
- Modifier 类 - 提供了 static 方法和常量，对类和成员访问修饰符进行解码。
- **Proxy 类 - 提供动态地生成代理类和类实例的静态方法。**(与动态代理有关，下面会详细讲解)

1.3.2 获取Class对象的3种方式

获取class对象的三种方式

① Class.forName("全类名");
② 类名.class
③ 对象.getClass();

源代码阶段：没有加载进内存，此时用方式1，是最常用的方法，一个字符串可以传入也可写在配置文件中。这个方法会出现类找不到的情况，因此使用这个方法获取Class对象时，必须捕获ClassNotFoundException异常。

加载阶段：方式2，一般常用作参数传递（锁对象），适用于在编译时已经知道具体的类，需要导入类的包，依赖太强，不导包就抛编译错误。

运行阶段：方式3，当已经有了这个类的对象时才可以使用

1.3.3 创建实例

通过反射来创建实例对象主要有两种方式：

1. **用 Class 对象的 newInstance 方法。**
2. **用 Constructor 对象的 newInstance 方法。**

示例：

```
public class NewInstanceDemo {
    public static void main(String[] args)
            throws IllegalAccessException, InstantiationException,
NoSuchMethodException, InvocationTargetException {
        Class<?> c1 = StringBuilder.class;
        StringBuilder sb = (StringBuilder) c1.newInstance();
        sb.append("aaa");
        System.out.println(sb.toString());
        //获取String所对应的Class对象
        Class<?> c2 = String.class;
        //获取String类带一个String参数的构造器
        Constructor constructor = c2.getConstructor(String.class);
        //根据构造器创建实例
        String str2 = (String) constructor.newInstance("bbb");
        System.out.println(str2);
    }
}
//Output:
//aaa
//bbb
```

1.3.4 Field

Class 对象提供以下方法获取对象的成员（Field）：

1. getFiled - 根据名称获取公有的（public）类成员。
2. getDeclaredField - 根据名称获取已声明的类成员。但不能得到其父类的类成员。
3. getFields - 获取所有公有的（public）类成员。
4. getDeclaredFields - 获取所有已声明的类成员。

示例如下：

```
public class ReflectFieldDemo {
    class FieldSpy<T> {
        public Boolean[][] b = {{false, false}, {true, true}};
        public String name = "Alice";
        public List<Integer> list;
        public T val;
    }
    public static void main(String[] args) throws NoSuchFieldException {
        Field f1 = FieldSpy.class.getField("b");
        System.out.format("Type: %s%n", f1.getType());
        Field f2 = FieldSpy.class.getField("name");
        System.out.format("Type: %s%n", f2.getType());
        Field f3 = FieldSpy.class.getField("list");
        System.out.format("Type: %s%n", f3.getType());
        Field f4 = FieldSpy.class.getField("val");
        System.out.format("Type: %s%n", f4.getType());
    }
}
//Output:
//Type: class [[Z
//Type: class java.lang.String
//Type: interface java.util.List
//Type: class java.lang.Object
```

1.3.5 Method

Class 对象提供以下方法获取对象的方法（Method）：

- getMethod - 返回类或接口的指定方法。其中第一个参数为方法名称，后面的参数为方法参数对应 Class 的对象。
- getDeclaredMethod - 返回类或接口的指定声明方法。其中第一个参数为方法名称，后面的参数为方法参数对应 Class 的对象。
- getMethods - **返回类或接口的所有 public 方法**，包括其父类的 public 方法。
- getDeclaredMethods - **返回类或接口声明的所有方法**，包括 public、protected、默认（包）访问和 private 方法，但不包括继承的方法。

获取一个 Method 对象后，可以用 invoke 方法来调用这个方法。

invoke 方法的签名为:

```
public Object invoke(Object obj, Object... args)
        throws IllegalAccessException, IllegalArgumentException,
           InvocationTargetException
```

示例：

```
public class ReflectMethodDemo {
    public static void main(String[] args)
            throws NoSuchMethodException, InvocationTargetException,
IllegalAccessException {
        // 返回所有方法
        Method[] methods1 = System.class.getDeclaredMethods();
        System.out.println("System getDeclaredMethods 清单（数量 = " +
methods1.length + "）: ");
        for (Method m : methods1) {
            System.out.println(m);
        }
        // 返回所有 public 方法
        Method[] methods2 = System.class.getMethods();
        System.out.println("System getMethods 清单（数量 = " + methods2.length +
"）: ");
        for (Method m : methods2) {
            System.out.println(m);
        }
        // 利用 Method 的 invoke 方法调用 System.currentTimeMillis()
        Method method = System.class.getMethod("currentTimeMillis");
        System.out.println(method);
        System.out.println(method.invoke(null));//静态方法
    }
}
```

> 这里我有个疑问：为什么method.invoke（obj,args）时需要把对象传过来?

对象的本质是用来存数据的，即一般是有状态的，而方法作为一种行为描述，是无状态的，是所有对象所共有的，并不属于某个对象私有，但是每一个对象都保存一份Method对象的话就太浪费内存，于是可以抽取出来放在方法区共用。

既然共用，那就又产生一个问题，看下面这个例子：

```
Person zhangsan=new Person(18);
Person lisi=new Person(20);
zhangsan.changeAge(30);
lisi.changeAge(40);
```

既然上面提出了方法Method是对象共用的，那JVM如何保证对象zhangsan调用changeAge(30)方法时修改的是自己本身呢？这其实是**JVM的一种隐性机制，每次对象调用方法时，都会隐性传递当前调用该方法的对象引用即this。**

1.3.6 Constructor

Class 对象提供以下方法获取对象的构造方法（Constructor）：

- getConstructor - 返回类的特定 public 构造方法。参数为方法参数对应 Class 的对象。
- getDeclaredConstructor - 返回类的特定构造方法。参数为方法参数对应 Class 的对象。
- getConstructors - 返回类的所有 public 构造方法。
- getDeclaredConstructors - 返回类的所有构造方法。

获取一个 Constructor 对象后，可以用 newInstance 方法来创建类实例。

示例：

```
public class ReflectMethodConstructorDemo {
    public static void main(String[] args)
            throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException, InstantiationException {
        Constructor<?>[] constructors1 = String.class.getDeclaredConstructors();
        System.out.println("String getDeclaredConstructors 清单（数量 = " +
constructors1.length + "）：");
        for (Constructor c : constructors1) {
            System.out.println(c);
        }
        Constructor<?>[] constructors2 = String.class.getConstructors();
        System.out.println("String getConstructors 清单（数量 = " +
constructors2.length + "）：");
        for (Constructor c : constructors2) {
            System.out.println(c);
        }
        System.out.println("====================");
        Constructor constructor = String.class.getConstructor(String.class);
        System.out.println(constructor);
        String str = (String) constructor.newInstance("bbb");
        System.out.println(str);
    }
}
```

1.4 java.lang.Class

java.lang.Class类是一个非常重要的概念，影响到你对反射的理解。这是java官方提供的用来**描述一个类的信息的类。**

对该类的通俗易懂的理解：

我们都听说过B超，在体检的时候，医生只需把一个探头在我们身上滑动就可以将我们体内的肝、胆、肾等器官反射到B超设备上显示。

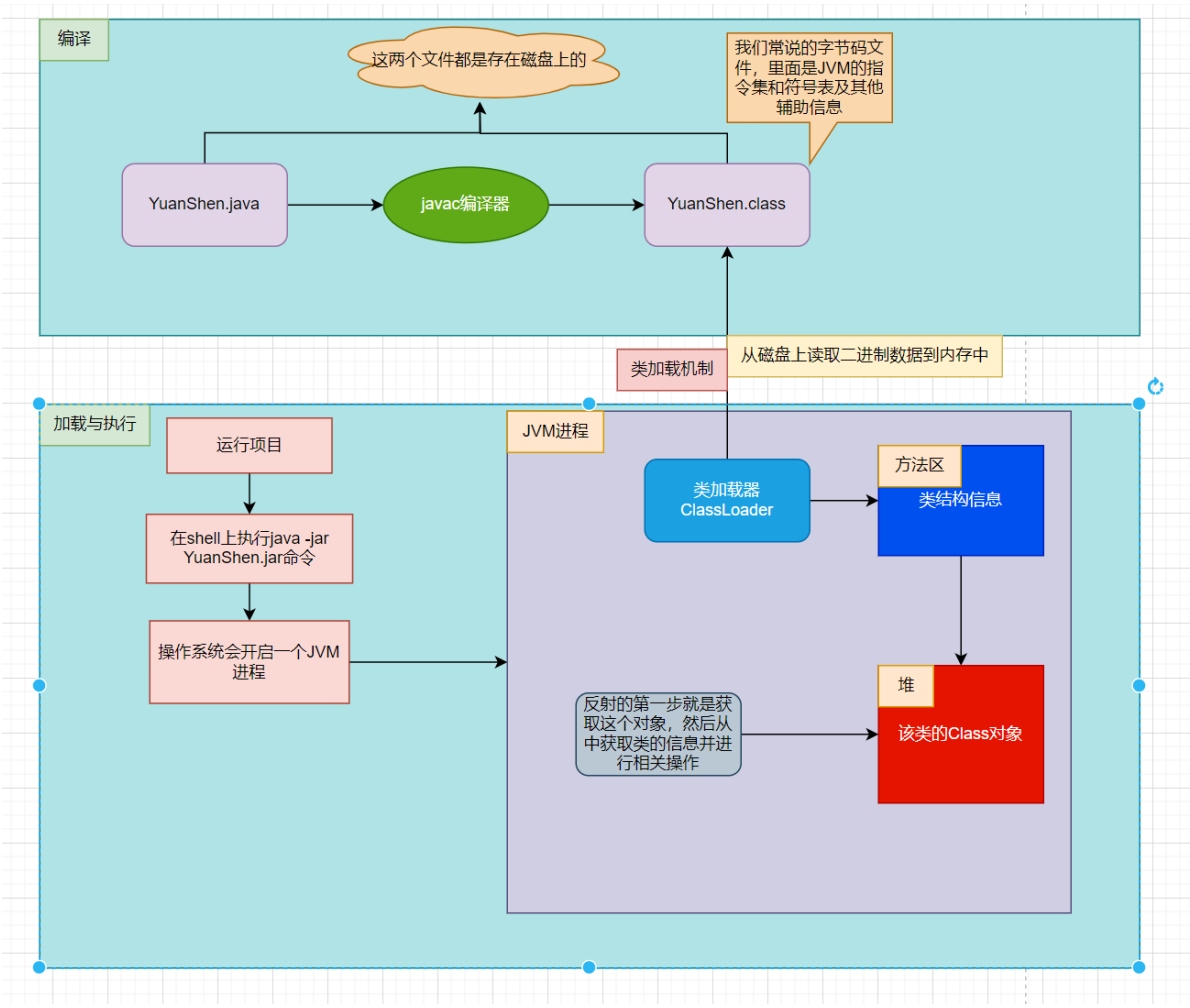
Class类对象就相当于B超的探头，将一个类的方法、变量、接口、类名、类修饰符等信息告诉运行的程序。

要想使用反射，首先需要获得该类的Class对象。Java中，某个类无论有多少个实例，这些实例都会对应于同一个Class对象。这个Class对象是由JVM生成的，通过它能获悉整个类的结构。

所以可以这么说，该**Class对象是所有反射API的入口点。**

反射的本质就是：在运行时，把Java类中的各种成分映射成一个个的Java对象（如Method对象，Field对象）

下面来看一下Class对象的生成过程，来加深对反射的理解：



无论是通过new还是反射创建实例，都离不开.class文件及Class对象。

2. 代理

2.1 代理模式

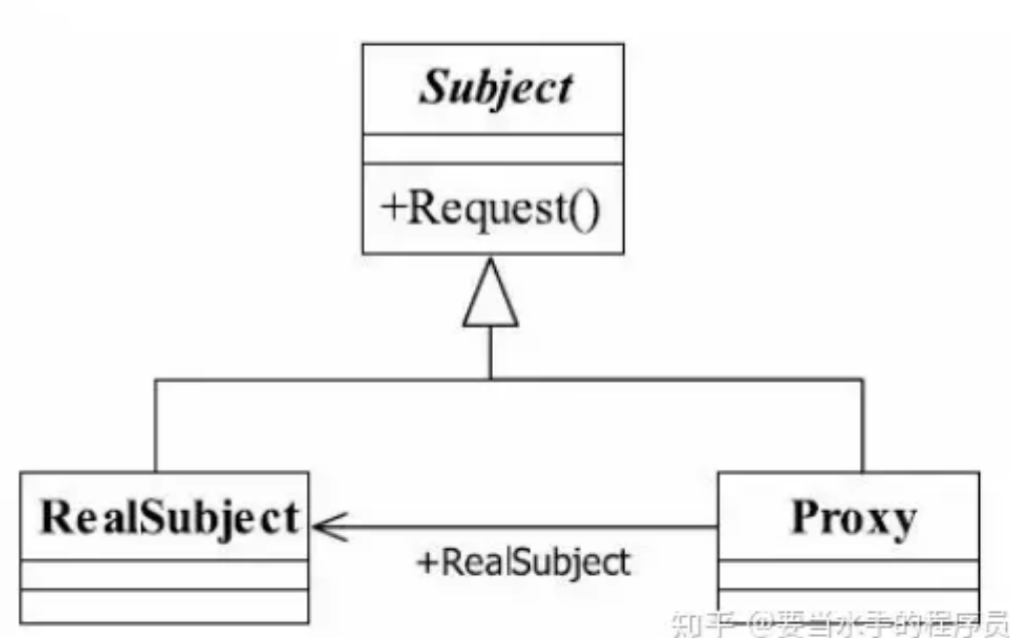
什么是代理模式?

这是设计模式之一，它的定义是：**为对象提供一种代理以控制对这个对象的访问。**

目前我能感受到的它的作用就是高扩展性：可以为被代理类提供额外的功能，并且在不改变原有代码的情况下对其进行控制和增强。

代理模式有三个核心概念：

- **抽象角色**：通过接口或抽象类声明真实角色要实现的业务方法。
- **真实角色**：实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。
- **代理角色**：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并且可以附加自己的操作。



2.2 Java中的代理

对应上面的3个概念：

- **代理接口**：定义了被代理类与代理类之间共同实现的方法，客户端通过该接口来访问被代理类的业务方法。
- **被代理类**：也称为真实主题（Real Subject），代理模式中的核心对象，代理类所代理的对象。
- **代理类**：实现了上面的代理接口，并持有一个被代理类的引用，在客户端请求时可以在调用被代理类对象的前后进行一些额外的逻辑处理。

在Java中，代理模式有两种实现方式：

静态代理和动态代理，其实二者的差别主要是看**代理类的字节码文件**生成时机：

静态代理在程序运行前就已经存在而动态代理是在程序运行过程中动态生成的。

2.2.1 静态代理

所谓静态指的是**需要程序员手动为每个被代理类编写代理类，代理类和被代理类实现同一个接口，代理类中调用被代理类的方法并可以进行增强。**

为什么要求代理类和被代理类实现同一个接口呢？所谓接口就是规范，**为了确保被代理类的所有功能都能被代理。**

优点是：可以在不修改目标对象的基础上对被代理类的**功能进行扩展。**

缺点是：代理类需要与被代理类实现一样的接口，所以一旦接口增加方法，二者要同时维护，增大了工作量。而且会导致项目变得臃肿。

此问题可以用动态代理来解决。

2.2.2动态代理

动态代理这篇文章讲得非常好

Java动态代理：JDK 和CGLIB、Javassist、ASM之间的差别 (详细) janino asm-CSDN博客

2.2.2.1基于接口的jdk动态代理

动态代理利用Java提供的反射机制，**在运行时动态生成代理类**，可以代理任意实现了接口的类，无需手动编写代理类。

Java原生的动态代理是必须要基于接口的，接口起到的作用至关重要。

InvocationHandler 接口

InvocationHandler 接口定义：

```
public interface InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable;
}
```

每一个动态代理类都必须要实现 InvocationHandler 这个接口，并且每个代理类的实例都关联到了一个 Handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由 InvocationHandler 这个接口的 invoke 方法来进行调用。

我们来看看 InvocationHandler 这个接口的唯一一个方法 invoke 方法：

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

参数说明：

- proxy - 代理的真实对象。
- method - 所要调用真实对象的某个方法的 Method 对象
- args - 所要调用真实对象某个方法时接受的参数

Proxy 类

Proxy 这个类的作用就是用来动态创建一个代理类及实例，它提供了许多的方法，但是我们用的最多的就是 newProxyInstance 这个方法：

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

这个方法的作用就是得到一个动态的代理对象。

参数说明：

- **loader** - 一个 ClassLoader 对象，定义了由哪个 ClassLoader 对象来对生成的代理对象进行加载。
- **interfaces** - 一个 Interface 对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了
- **h** - 一个 InvocationHandler 对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个 InvocationHandler 对象上

我们先通过一个简单的示例来感受一下jdk动态代理：

1.提供一个基础的接口(com.liu.ReChargeable)，作为被代理类（如com.liu.BYD）和代理类之间的统一入口；

```
public interface ReChargeable {
    void charge();
}
```

被代理类：

```
public class BYD implements ReChargeable{
    private String name;

    public BYD(String name) {
        this.name = name;
    }

    @Override
    public void charge() {
        System.out.println("BYD"+name+" 正在充电。。。");
    }
}
```

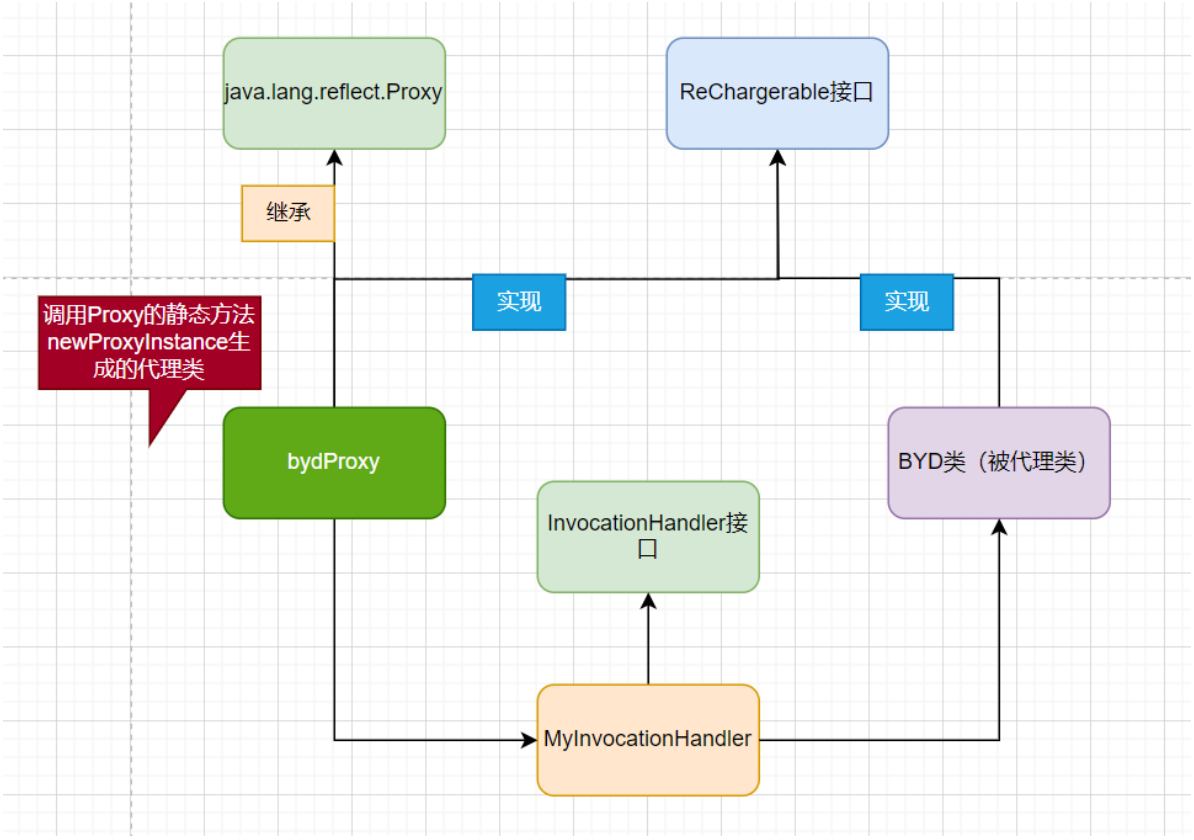
2.通过实现InvocationHandler接口来自定义自己的MyInvocationHandler，对代理对象方法的调用，会被分派到其 invoke 方法来真正实现动作；

```
public class MyInvocationHandler implements InvocationHandler {
    private Object realSubject;

    public MyInvocationHandler(Object realSubject) {
        this.realSubject = realSubject;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("before...");
        System.out.println("连接充电桩....");
        Object invoke = method.invoke(realSubject, args);
        System.out.println("after....");
        System.out.println("断开充电桩.....");
        return invoke;
    }
}
```

3.调用java.lang.reflect.Proxy类的newProxyInstance 方法，生成一个实现了相应基础接口的代理类实例；

```
public class Test {
    public static void main(String[] args) {
        BYD byd=new BYD("秦");
        MyInvocationHandler myInvocationHandler=new MyInvocationHandler(byd);
        /*
         * 通过Proxy的newProxyInstance方法来创建我们的代理对象，我们来看看其三个参数
         * 第一个参数 byd.getClass().getClassLoader() ，我们这里使用byd这个类的
ClassLoader对象来加载我们的代理对象
```

```
            * 第二个参数byd.getClass().getInterfaces()，这里为代理对象提供的接口是真实对象所
实现的接口，表示我要代理的是该真实对象，这样我就能调用这组接口中的方法了
            * 第三个参数handler，这里将这个代理对象关联到了上方的 InvocationHandler 这个对象
上
            */
           ReChargeable bydProxy = (ReChargeable)
Proxy.newProxyInstance(byd.getClass().getClassLoader(),
                   byd.getClass().getInterfaces(),
                   myInvocationHandler);
           bydProxy.charge();
       }
}
```



代理类Proxy和被代理类都需要实现相同的接口（这也意味着**被代理类自己的方法无法被增强**），然后还有一个核心接口InvocationHandler，它负责调用被代理类的方法（即真正对被代理类进行增强的是InvocationHandler，然后InvocationHandler还会再调用invoke方法调用被代理类的方法）

当调用代理对象中的方法时，代理对象并不会去调用被代理类的方法，而是将执行的权利交给InvocationHandler，执行与其关联的InvocationHandler中的invoke方法，通过Method参数来具体区分是什么方法，并调用method.invoke执行该方法

为什么JDK的动态代理只能基于接口实现，不能基于继承来实现?

通过上面的示例我们知道，**代理类默认继承了java.lang.Proxy类。而在Java中是不支持多继承的，代理类就不能再继承被代理类了，所以JDK只能通过接口去实现动态代理。这是jdk对动态代理实现的设计上的缺陷**

Proxy已经设计得非常优美，但是还是有一点点小小的遗憾之处，那就是它**始终无法摆脱仅支持interface代理的桎梏**，因为它的设计注定了这个遗憾。它们已经注定有一个共同的父类叫Proxy。Java的继承机制注定了这些动态代理类们无法实现对class的动态代理，原因是多继承在Java中本质上就行不通

2.2.2.2 Cglib子类代理

JDK的动态代理只能代理实现了接口的类，而没有实现接口的类就不能实现JDK的动态代理。此时cglib就发挥作用了，它是针对类来代理的，它的原理是**对指定的目标类生成一个子类并覆盖其中方法实现功能增强，但是因为采用的是继承，所以不能对final修饰的类进行代理。**

想使用cglib：

1.需要导入cglib的依赖（已经包括在spring-core.jar中）

2.引入功能包后就可以在内存中动态构建子类

3.代理的类不能为final

4.**目标对象的方法若为static或final**，那么就不会被拦截，即不会增强这些方法。

核心有：Enhancer类用来指定那个类需要增强，自己创建类实现MethodIntecepter接口对方法进行拦截并增强，Enhancer类中的setCallBack方法

```
import net.sf.cglib.proxy.Enhancer;
public class Test {
    public static void main(String[] args) {
        Programmer progammer = new Programmer();
        Hacker hacker = new Hacker();//实现了MethodIntecepter接口，对方法进行拦截并进行增强

        //cglib 中加强器，用来创建动态代理，是cglib的核心类
        Enhancer enhancer = new Enhancer();

        enhancer.setSuperclass(progammer.getClass());//设置要创建动态代理的类
        enhancer.setCallback(hacker);// 设置回调，这里相当于是对于代理类proxy上所有方法的调用，都会先调用CallBack，而Callback则需要实行intercept()方法进行拦截
        Programmer proxy =(Programmer)enhancer.create();//因为proxy继承了Programmer类所以可用多态
                proxy.code();
    }
}
```