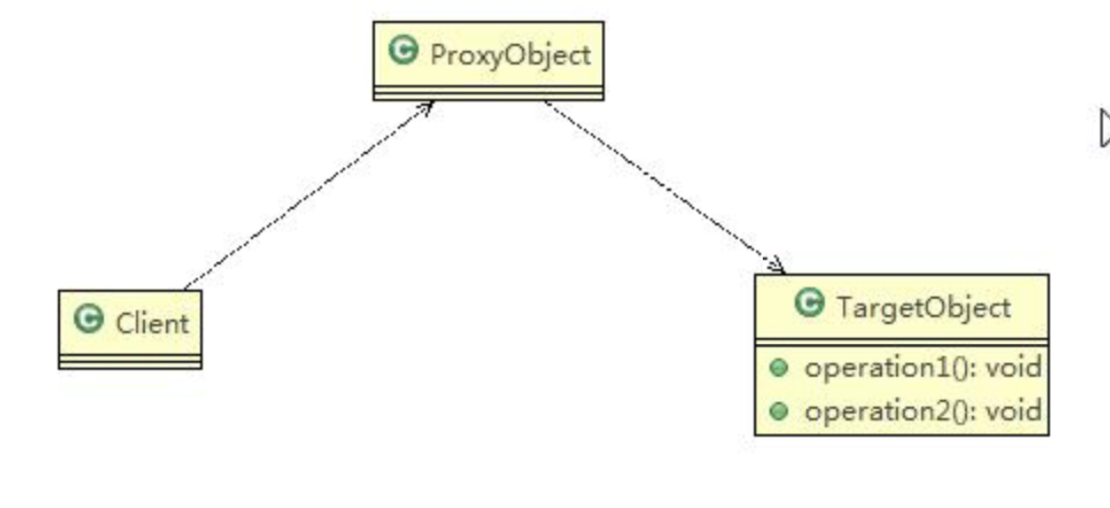
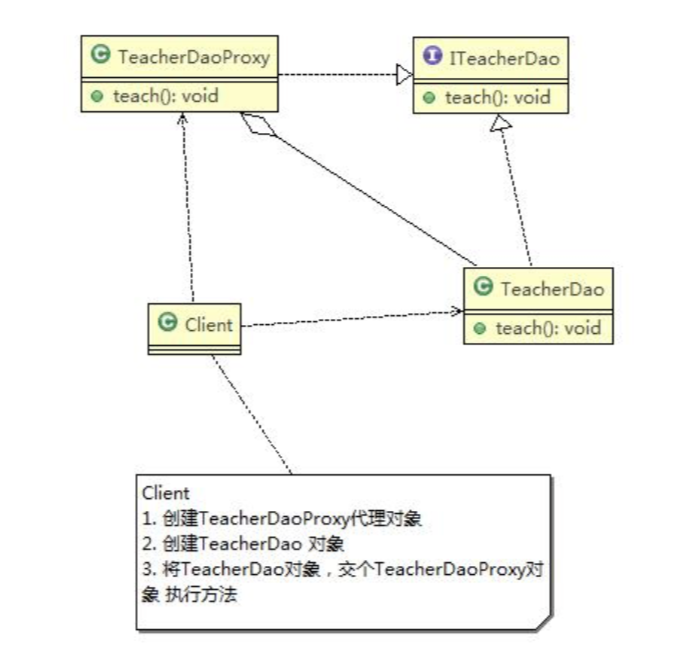
1. 代理模式(Proxy Pattern)：为其他对象提供一种代理以控制对这个对象的访问。为真实对象提供一个代理, 从而控制对真实对象的访问，是一种对象结构型模式。这样做的好处 是: 可以在目标对象实现的基础上，增强额外的功能操作，即扩展目标对象的功能
2. 被代理的对象可以是远程对象、创建开销大的对象或需要安全控制的对象
3. 代理模式有不同的形式，主要有三种 **静态代理**、**动态代理(**JDK代理、接口代理)和**Cglib代理** (可以在内存动态的创建对象，而不需要实现接口，他是属于动态代理的范畴)
4. 类图



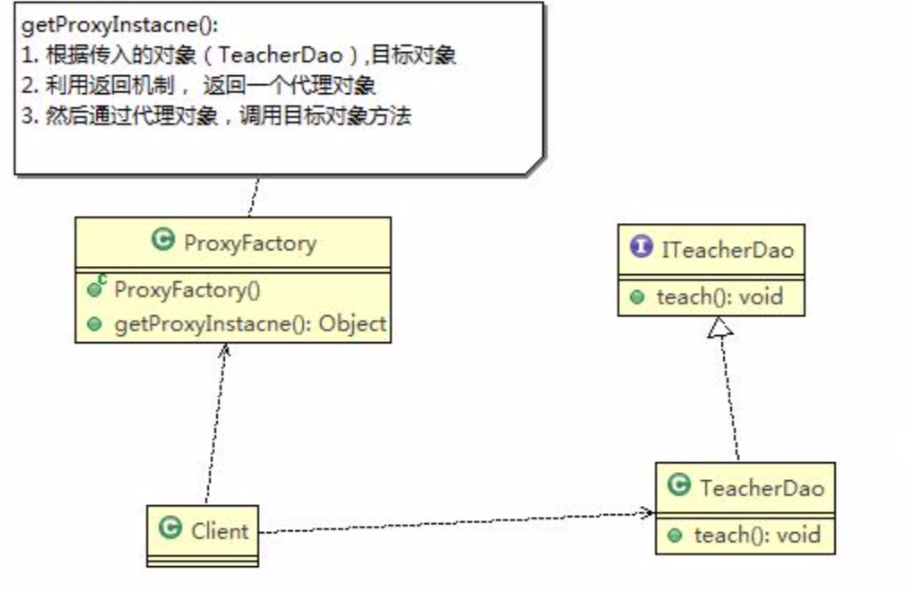
1. 代理模式的三种形式
   1. 静态代理，静态代理在使用时,需要定义接口或者父类,被代理对象(即目标对象)与代理对象一起实现相同的接口或者是继承相同父类
      1. 实例.
         1. 定义一个接口: ITeacherDao
         2. 目标对象TeacherDAO实现接口ITeacherDAO
         3. 使用静态代理方式,就需要在代理对象TeacherDAOProxy中也实现ITeacherDAO
         4. 调用的时候通过调用代理对象的方法来调用目标对象.
         5. 特别提醒:代理对象与目标对象要实现相同的接口,然后通过调用相同的方法来调用目标对象的方法
      2. 类图



* + 1. 实例代码

|  |
| --- |
| 1.接口  public interface ITeacherDao {  void teach(); // 授课的方法  }  2.目标对象  public class TeacherDao implements ITeacherDao {  @Override public void teach() {  System.out.println(" 老师授课中 。。。。。");  }  }  3.代理对象，静态代理  public class TeacherDaoProxy implements ITeacherDao{  // 目标对象，通过接口来聚合  private ITeacherDao target;  // 构造器 public TeacherDaoProxy(ITeacherDao target) {  this.target = target;  }  @Override public void teach() {  System.out.println("开始代理 完成某些操作。。。。。 ");  //方法  target.teach(); System.out.println("提交。。。。。");//方法  }  }  4.使用Client  public class Client {  public static void main(String[] args) { // 创建目标对象(被代理对象) TeacherDao teacherDao = new TeacherDao();  // 创建代理对象, 同时将被代理对象传递给代理对象 TeacherDaoProxy teacherDaoProxy = new TeacherDaoProxy(teacherDao);  // 通过代理对象，调用到被代理对象的方法  // 即:执行的是代理对象的方法，代理对象再去调用目标对象的方法  teacherDaoProxy.teach();  }  } |

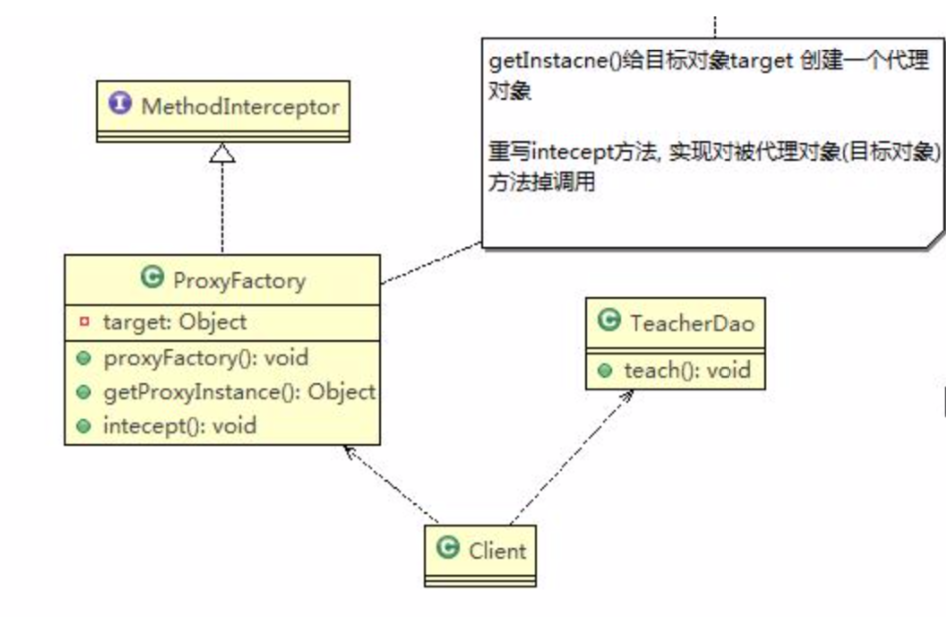
* + 1. 静态代理优缺点
       1. 优点: 在不修改目标对象的功能前提下, 能通过代理对象对目标功能扩展
       2. 缺点:因为代理对象需要与目标对象实现一样的接口,所以会有很多代理；一旦接口增加方法,目标对象与代理对象都要维护
  1. 动态代理。代理对象,不需要实现接口，但是目标对象要实现接口，否则不能用动态代理。动态代理也叫做: JDK代理、接口代理，代理对象的生成，是利用JDK的API, java.lang.reflect.Proxy#newProxyInstance动态的在内存中构建代理对象, 完整的写法 public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
     1. 实例类图



* + 1. 实例代码

|  |
| --- |
| 1.接口  public interface ITeacherDao {  // 授课的方法  void teach();  // 带参数的方法  void sayHello(String name);  }  2.目标对象  public class TeacherDao implements ITeacherDao {  @Override public void teach() {  System.out.println(" 老师授课中 。。。。。");  }  @Override public void sayHello(String name) {  System.out.println("hello " + name);  }  }  3.代理工厂，JDK代理  public class ProxyFactory {  // 维护一个目标对象，Object  private Object target;  // 构造器，对 target 进行初始化  public ProxyFactory(Object target) {  this.target = target;  }  // 给目标对象 生成一个代理对象  public Object **getProxyInstance**() {  // 说明  // public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)  // 1. ClassLoader loader : 指定当前目标对象使用的类加载器, 获取加载器的方法固定  // 2. Class<?>[] interfaces: 目标对象实现的接口类型，使用泛型方法确认类型  // 3. InvocationHandler h : 事情处理，执行目标对象的方法时，会触发事情处理器方法, 会把当前执行  的目标对象方法作为参数传入  return Proxy.newProxyInstance(target.getClass().getClassLoader(),  target.getClass().getInterfaces(), new **InvocationHandler**() {  @Override public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.out.println("JDK 代理开始~~");  //反射机制调用目标对象的方法  Object returnVal = method.invoke(target, args);  System.out.println("JDK 代理提交");  return returnVal;  }  });  }  }  4.使用Client  public class Client {  public static void main(String[] args) { // 创建目标对象(被代理对象) TeacherDao teacherDao = new TeacherDao();  // 给目标对象，创建代理对象, 可以转成 ITeacherDao  ITeacherDao proxyInstance = (ITeacherDao)new ProxyFactory(target).getProxyInstance();  // proxyInstance=class com.sun.proxy.$Proxy0 内存中动态生成了代理对象  System.out.println("proxyInstance=" + proxyInstance.getClass());  // 通过代理对象，调用目标对象的方法  proxyInstance.teach();  proxyInstance.sayHello(" tom ");  }  } |

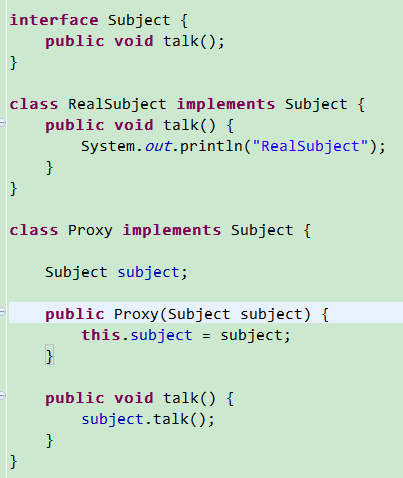
* 1. **Cglib代理**，静态代理和JDK代理模式都要求目标对象是实现一个接口,但是有时候目标对象只是一个单独的对象,并没有实现任何的接口,这个时候可使用目标对象子类来实现代理，这就是 Cglib 代理。
     1. Cglib 代理也叫作**子类代理,**它是在内存中构建一个子类对象从而实现对目标对象功能扩展, 有些书也将 Cglib 代 理归属到动态代理
     2. Cglib 是一个强大的高性能的代码生成包,它可以在运行期扩展 java 类与实现 java 接口.它广泛的被许多 AOP 的 框架使用,例如 Spring AOP，实现方法拦截
     3. 在AOP编程中如何选择代理模式
        1. 目标对象**需要实现接口**，用 JDK 代理
        2. 目标对象**不需要实现接口**，用 Cglib 代理
     4. Cglib 包的底层是通过使用字节码处理框架 ASM 来转换字节码并生成新的类
     5. 在内存中动态构建子类，注意代理的类不能为final，否则报错 java.lang.IllegalArgumentException:
     6. 目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.
     7. 实例

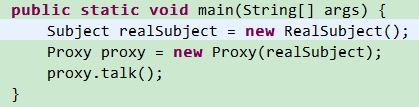


* + 1. 实例代码

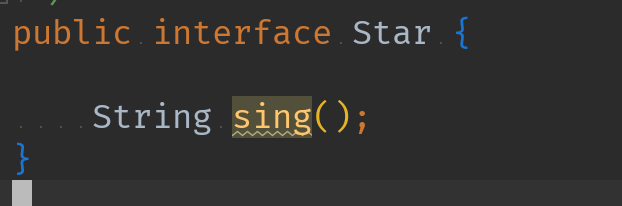
|  |
| --- |
| 1.目标对象  public class TeacherDao {  public String teach() { System.out.println(" 老师授课中 ， 我是 cglib 代理，不需要实现接口 ");  return "hello";  }  }  2.代理工厂，CGLIB代理  public class ProxyFactory implements MethodInterceptor {  // 维护一个目标对象 private Object target;  // 构造器，传入一个被代理的对象  public ProxyFactory(Object target) {  this.target = target;  }  // 返回一个代理对象: 是 target 对象的代理对象  public Object getProxyInstance() {  // 1. 创建一个工具类 Enhancer enhancer = new Enhancer();  / /2. 设置父类  enhancer.setSuperclass(target.getClass());  // 3. 设置回调函数  enhancer.setCallback(this); // 4. 创建子类对象，即代理对象 return enhancer.create();  }  // 重写 intercept 方法，会调用目标对象的方法  @Override public Object intercept(Object arg0, Method method, Object[] args, MethodProxy arg3) throws Throwable {  System.out.println("Cglib 代理模式 ~~ 开始");  Object returnVal = method.invoke(target, args);  System.out.println("Cglib 代理模式 ~~ 提交");  return returnVal;  }  }  3.Client使用  public class Client {  public static void main(String[] args) { // 创建目标对象 TeacherDao target = new TeacherDao();  //获取到代理对象，并且将目标对象传递给代理对象 TeacherDao proxyInstance = (TeacherDao)new ProxyFactory(target).getProxyInstance();  // 执行代理对象的方法，触发 intecept 方法，从而实现 对目标对象的调用  String res = proxyInstance.teach();  System.out.println("res=" + res);  }  } |

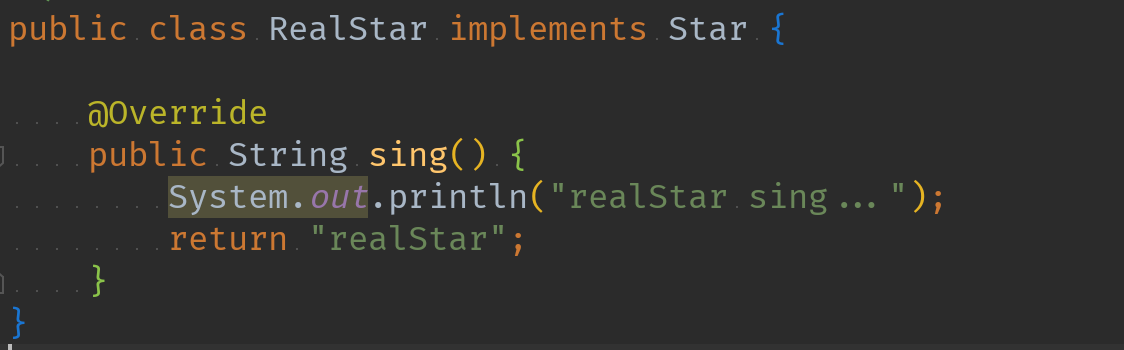
1. 代理模式应用场合(场景)
   1. 远程代理，也就是为一个对象在不同的地址空间提供局部代表，这样可以隐藏一个对象存在于不同地址空间的事实。
   2. 虚拟代理，是根据需要创建开销很大的对象，通过它来存放实例化需要很长时间的真实对象。
   3. 安全(保护)代理，用来控制真实对象访问时的权限。
   4. 智能引用，是指当调用真实对象时，代理处理另外一些事
2. 分类
   1. 静态代理(静态定义代理类)
   2. 动态代理(动态生成代理类), 可以更加灵活和统一处理多个方法(每个方法调用都需要经过代理)
      1. JDK自带动态代理
      2. Javaassist字节码操作库实现
      3. CGLIB
      4. ASM(底层使用指令, 维护性差)
3. 简单例子—静态代理



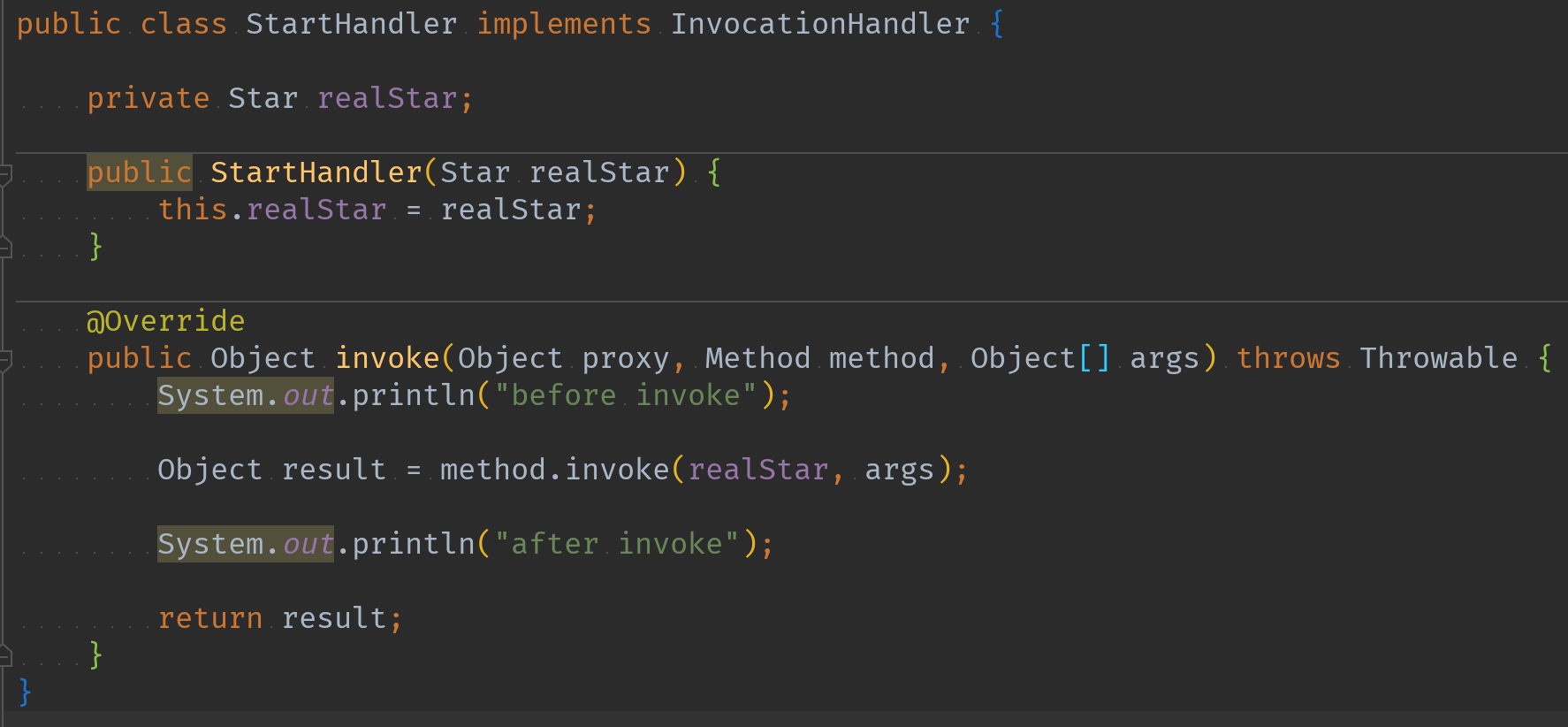
测试

1. 例子—动态代理(JDK自带)

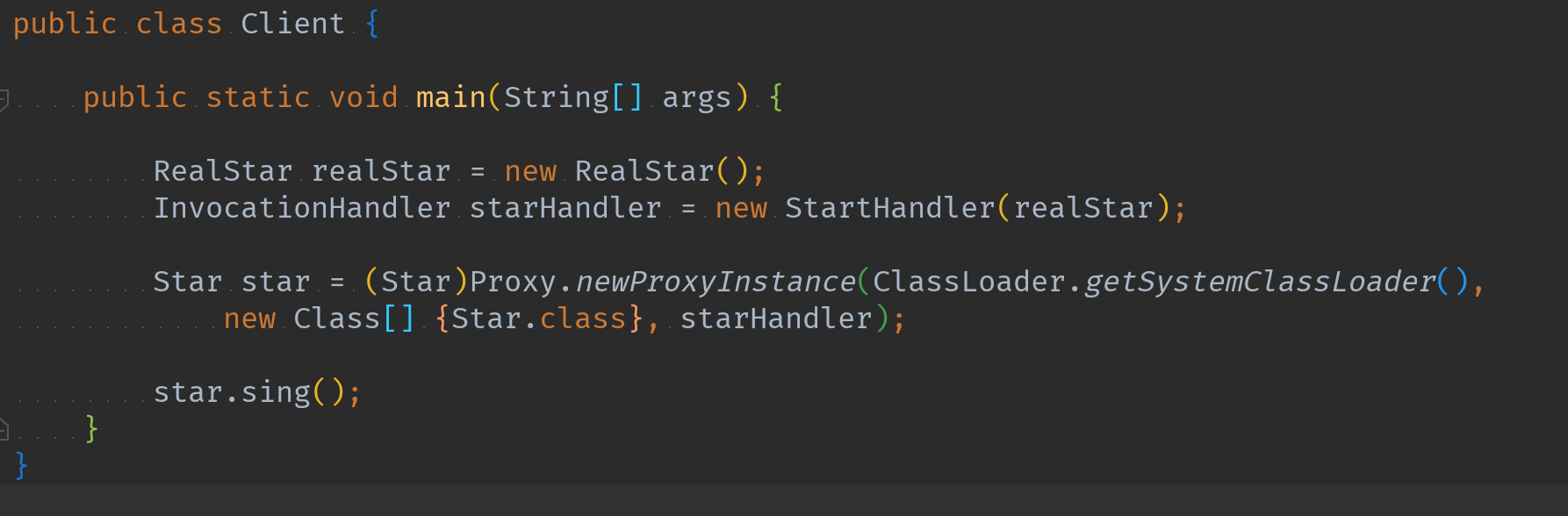




* 1. 处理器接口java.lang.reflect.InvocationHandler, 通过invoke方法实现对真实角色的代理访问



* 1. 动态生成代理类和对象java.lang.reflect.Proxy



1. 开发框架应用场景
   1. Struts2 拦截器实现
   2. DB连接池关闭处理
   3. MyBatis实现拦截器插件
   4. AspectJ的实现
   5. Spring中AOP的实现
      1. 日志拦截
      2. 声明式事务处理
   6. RMI远程方法调用