

[静态代理&动态代理（jdk&cglib）]

[静态代理&动态代理学习文档]



2018-10-25

[公司名称]

[公司地址]

目录

[1 Shiro 2](#_Toc528659816)

[1.1是什么 2](#_Toc528659817)

[1.2表的设计 2](#_Toc528659818)

[1.3关键代码 6](#_Toc528659819)

[1.3.1 Shiro配置: 6](#_Toc528659820)

[1.3.2 Shiro登录&认证&授权: 8](#_Toc528659821)

[1.3.3 前台菜单隐藏 14](#_Toc528659822)

[2 SpringSecurity 15](#_Toc528659823)

[2.1为什么要使用Maven？ 15](#_Toc528659824)

[2.2后台工程搭建分析 15](#_Toc528659825)

[2.2.1常见的Maven打包方式 15](#_Toc528659826)

[2.2.2工程依赖关系解析 15](#_Toc528659827)

[2.3搭建taotao-parent工程 16](#_Toc528659828)

[2.3.1工程pom.xml文件 16](#_Toc528659829)

[2.4搭建taotao-common工程 23](#_Toc528659830)

# 1 代理模式

## 1.1简介

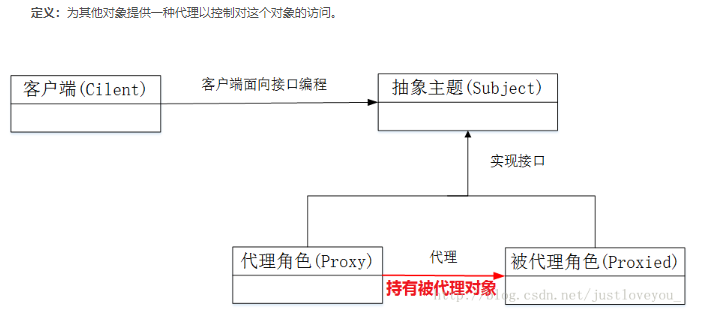
**代理模式是一种常用的设计模式，在AOP、RPC等诸多框架中均有它的身影。根据代理类的创建时机和创建方式的不同，可以将其分为静态代理和动态代理两种形式：在程序运行前就已经存在的编译好的代理类是为静态代理，在程序运行期间根据需要动态创建代理类及其实例来完成具体的功能是为动态代理。代理模式的目的就是为真实业务对象提供一个代理对象以控制对真实业务对象的访问，代理对象的作用有：**

**代理对象存在的价值主要用于拦截对真实业务对象的访问；**

**代理对象具有和目标对象(真实业务对象)实现共同的接口或继承于同一个类；**

**代理对象是对目标对象的增强，以便对消息进行预处理和后处理。**

## 1.2定义与结构



# 2 静态代理

抽象主题角色:HelloService接口

public interface HelloService {

String hello(String name);

String hi(String msg);

}

委托类角色: HelloServiceImpl类

public class HelloServiceImpl implements HelloService{

@Override

public String hello(String name) {

return "Hello " + name;

}

@Override

public String hi(String msg) {

return "Hi, " + msg;

}

}

代理类角色: HelloServiceProxy类

public class HelloServiceProxy implements HelloService {

private HelloService helloService;

public HelloServiceProxy(HelloService helloService) {

this.helloService = helloService;

}

@Override

public String hello(String name) {

System.out.println("预处理...");

String result = helloService.hello(name);

System.out.println(result);

System.out.println("后处理...");

return result;

}

@Override

public String hi(String msg) {

System.out.println("预处理...");

String result = helloService.hi(msg);

System.out.println(result);

System.out.println("后处理...");

return result;

}

}

客户端：

public class Main {

public static void main(String[] args){

HelloService helloService = new HelloServiceImpl();

HelloServiceProxy helloServiceProxy = new HelloServiceProxy(helloService);

helloServiceProxy.hello("Panda");

helloServiceProxy.hi("Panda");

}

}

/\*\* Output

预处理...

Hello Panda

后处理...

预处理...

Hi, Panda

后处理...

\*\*/

------

# 2 jdk动态代理

## 2.1 动态代理引入

对代理模式而言，一般来说，具体主题类与其代理类是一一对应的，这也是静态代理的特点。但是，也存在这样的情况：有N个主题类，但是代理类中的“预处理、后处理”都是相同的，仅仅是调用主题不同。那么，若采用静态代理，那么必然需要手动创建N个代理类，这显然让人相当不爽。动态代理则可以简单地为各个主题类分别生成代理类，共享“预处理，后处理”功能，这样可以大大减小程序规模，这也是动态代理的一大亮点。

在动态代理中，代理类是在运行时期生成的。因此，相比静态代理，动态代理可以很方便地对委托类的相关方法进行统一增强处理，如添加方法调用次数、添加日志功能等等。动态代理主要分为JDK动态代理和cglib动态代理两大类，本文主要对JDK动态代理进行探讨。

## 2.2 jdk动态代理机制的相关接口

要想使用JDK动态代理，首先需要了解其相关的类或接口：

java.lang.reflect.Proxy：该类用于动态生成代理类，只需传入目标接口、目标接口的类加载器以及InvocationHandler便可为目标接口生成代理类及代理对象。

// 方法 1: 该方法用于获取指定代理对象所关联的InvocationHandler

static InvocationHandler getInvocationHandler(Object proxy)

// 方法 2：该方法用于获取关联于指定类装载器和一组接口的动态代理类的类对象

static Class getProxyClass(ClassLoader loader, Class[] interfaces)

// 方法 3：该方法用于判断指定类是否是一个动态代理类

static boolean isProxyClass(Class cl)

// 方法 4：该方法用于为指定类装载器、一组接口及调用处理器生成动态代理类实例

static Object newProxyInstance(ClassLoader loader, Class[] interfaces,

InvocationHandler h)

ava.lang.reflect.InvocationHandler：该接口包含一个invoke方法，通过该方法实现对委托类的代理的访问，是代理类完整逻辑的集中体现，包括要切入的增强逻辑和进行反射执行的真实业务逻辑。

// 该方法代理类完整逻辑的集中体现。第一个参数既是代理类实例，第二个参数是被调用的方法对象，

// 第三个方法是调用参数。通常通过反射完成对具体角色业务逻辑的调用，并对其进行增强。

Object invoke(Object proxy, Method method, Object[] args)

java.lang.ClassLoader：类加载器类，负责将类的字节码装载到Java虚拟机中并为其定义类对象，然后该类才能被使用。Proxy静态方法生成动态代理类同样需要通过类加载器来进行加载才能使用，它与普通类的唯一区别就是其字节码是由JVM在运行时动态生成的而非预存在于任何一个.class 文件中。

## 2.3 jdk动态代理开始

创建被代理的接口和类

// 抽象主题角色

public interface HelloService {

String hello(String name);

String hi(String msg);

}

// 具体(真实)主题角色

public class HelloServiceImpl implements HelloService{

@Override

public String hello(String name) {

return "Hello " + name;

}

@Override

public String hi(String msg) {

return "Hi, " + msg;

}

}

实现InvocationHandler接口

public class MyInvocationHandler implements InvocationHandler{

// 真实业务对象

private Object target;

public MyInvocationHandler(Object target){

this.target = target;

}

@Override

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

// 增强逻辑

System.out.println("PROXY : " + proxy.getClass().getName());

// 反射调用，目标方法

Object result = method.invoke(target, args);

// 增强逻辑

System.out.println(method.getName() + " : " + result);

return result;

}

}

创建代理类并生成相应的代理对象

// 生成代理类的class对象

Class<?> clazz = Proxy.getProxyClass(helloService.getClass().getClassLoader(), helloService

.getClass().getInterfaces());

// 创建InvocationHandler

InvocationHandler myInvocationHandler = new MyInvocationHandler(helloService);

// 获取代理类的构造器对象

Constructor constructor = clazz.getConstructor(new Class[] {InvocationHandler.class});

// 反射创建代理对象

HelloService proxy = (HelloService)constructor.newInstance(myInvocationHandler);

也可以一步到位，

HelloService proxy = (HelloService)Proxy.newProxyInstance(HelloService.class.getClassLoader(),

helloService.getClass().getInterfaces(), new MyInvocationHandler(helloService));

使用代理

proxy.hello("rico");

proxy.hi("panda");

/\*\* Output

PROXY : com.sun.proxy.$Proxy0

hello : Hello rico

PROXY : com.sun.proxy.$Proxy0

hi : Hi, panda

\*\*/