



前面几节，我们学习了设计模式中的创建型模式。创建型模式主要解决对象的创建问题，封装复杂的创建过程，解耦对象的创建代码和使用代码。

其中，单例模式用来创建全局唯一的对象。工厂模式用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。建造者模式是用来创建复杂对象，可以通过设置不同的可选参数，“定制化”地创建不同的对象。原型模式针对创建成本比较大的对象，利用对已有对象进行复制的方式进行创建，以达到节省创建时间的目的。

从今天起，我们开始学习另外一种类型的设计模式：结构型模式。结构型模式主要总结了一些类或对象组合在一起的经典结构，这些经典的结构可以解决特定应用场景的问题。结构型模式包括：代理模式、桥接模式、装饰器模式、适配器模式、门面模式、组合模式、享元模式。今天我们要讲其中的代理模式。它也是在实际开发中经常被用到的一种设计模式。

话不多说，让我们正式开始今天的学习吧！

代理模式的原理解析

代理模式（Proxy Design Pattern）的原理和代码实现都不难掌握。它在不改变原始类（或叫被代理类）代码的情况下，通过引入代理类来给原始类附加功能。我们通过一个简单的例子来解释一下这段话。

这个例子来自我们在第25、26、39、40节中讲的性能计数器。当时我们开发了一个MetricsCollector类，用来收集接口请求的原始数据，比如访问时间、处理时长等。在业务系统中，我们采用如下方式来使用这个MetricsCollector类：

```

public class UserController {
    //...省略其他属性和方法...
    private MetricsCollector metricsCollector; // 依赖注入

    public UserVo login(String telephone, String password) {
        long startTimestamp = System.currentTimeMillis();

        // ... 省略login逻辑...

        long endTimeStamp = System.currentTimeMillis();
        long responseTime = endTimeStamp - startTimestamp;
        RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
        metricsCollector.recordRequest(requestInfo);

        //...返回UserVo数据...
    }

    public UserVo register(String telephone, String password) {
        long startTimestamp = System.currentTimeMillis();

        // ... 省略register逻辑...

        long endTimeStamp = System.currentTimeMillis();
        long responseTime = endTimeStamp - startTimestamp;
        RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
        metricsCollector.recordRequest(requestInfo);

        //...返回UserVo数据...
    }
}

```

很明显，上面的写法有两个问题。第一，性能计数器框架代码侵入到业务代码中，跟业务代码高度耦合。如果未来需要替换这个框架，那替换的成本会比较大。第二，收集接口请求的代码跟业务代码无关，本就不应该放到一个类中。业务类最好职责更加单一，只聚焦业务处理。

为了将框架代码和业务代码解耦，代理模式就派上用场了。代理类UserControllerProxy和原始类UserController实现相同的接口UserController。UserController类只负责业务功能。代理类UserControllerProxy负责在业务代码执行前后附加其他逻辑代码，并通过委托的方式调用原始类来执行业务代码。具体的代码实现如下所示：

```

public interface IUserController {
    UserVo login(String telephone, String password);
    UserVo register(String telephone, String password);
}

```

```

public class UserController implements IUserController {
    //...省略其他属性和方法...

    @Override
    public UserVo login(String telephone, String password) {
        //...省略login逻辑...
        //...返回UserVo数据...
    }

    @Override
    public UserVo register(String telephone, String password) {
        //...省略register逻辑...
        //...返回UserVo数据...
    }
}

public class UserControllerProxy implements IUserController {
    private MetricsCollector metricsCollector;
    private UserController userController;

    public UserControllerProxy(UserController userController) {
        this.userController = userController;
        this.metricsCollector = new MetricsCollector();
    }

    @Override
    public UserVo login(String telephone, String password) {
        long startTimestamp = System.currentTimeMillis();

        // 委托
        UserVo userVo = userController.login(telephone, password);

        long endTimeStamp = System.currentTimeMillis();
        long responseTime = endTimeStamp - startTimestamp;
        RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
        metricsCollector.recordRequest(requestInfo);

        return userVo;
    }

    @Override
    public UserVo register(String telephone, String password) {
        long startTimestamp = System.currentTimeMillis();

```

```
UserVo userVo = userController.register(telephone, password);

long endTimeStamp = System.currentTimeMillis();
long responseTime = endTimeStamp - startTimestamp;
RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
metricsCollector.recordRequest(requestInfo);

return userVo;
}
}

//UserControllerProxy使用举例
//因为原始类和代理类实现相同的接口，是基于接口而非实现编程
//将UserController类对象替换为UserControllerProxy类对象，不需要改动太多代码
UserController userController = new UserControllerProxy(new UserController());
```

参照基于接口而非实现编程的设计思想，将原始类对象替换为代理类对象的时候，为了让代码改动尽量少，在刚刚的代理模式的代码实现中，代理类和原始类需要实现相同的接口。但是，如果原始类并没有定义接口，并且原始类代码并不是我们开发维护的（比如它来自一个第三方的类库），我们也没办法直接修改原始类，给它重新定义一个接口。在这种情况下，我们该如何实现代理模式呢？

对于这种外部类的扩展，我们一般都是采用继承的方式。这里也不例外。我们让代理类继承原始类，然后扩展附加功能。原理很简单，不需要过多解释，你直接看代码就能明白。具体代码如下所示：

```

public class UserControllerProxy extends UserController {
    private MetricsCollector metricsCollector;

    public UserControllerProxy() {
        this.metricsCollector = new MetricsCollector();
    }

    public UserVo login(String telephone, String password) {
        long startTimestamp = System.currentTimeMillis();

        UserVo userVo = super.login(telephone, password);

        long endTimeStamp = System.currentTimeMillis();
        long responseTime = endTimeStamp - startTimestamp;
        RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
        metricsCollector.recordRequest(requestInfo);

        return userVo;
    }

    public UserVo register(String telephone, String password) {
        long startTimestamp = System.currentTimeMillis();

        UserVo userVo = super.register(telephone, password);

        long endTimeStamp = System.currentTimeMillis();
        long responseTime = endTimeStamp - startTimestamp;
        RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
        metricsCollector.recordRequest(requestInfo);

        return userVo;
    }
}

//UserControllerProxy使用举例
UserController userController = new UserControllerProxy();

```

动态代理的原理解析

不过，刚刚的代码实现还是有点问题。一方面，我们需要在代理类中，将原始类中的所有的方法，都重新实现一遍，并且为每个方法都附加相似的代码逻辑。另一方面，如果要添加的附加功能的类有不只一个，我们需要针对每个类都创建一个代理类。

如果有50个要添加附加功能的原始类，那我们就要创建50个对应的代理类。这会导致项目中类的个数成倍增加，增加了代码维护成本。并且，每个代理类中的代码都有点像模板式的“重复”代码，也增加了不必要的开发成本。那这个问题怎么解决呢？

我们可以使用动态代理来解决这个问题。所谓**动态代理**（Dynamic Proxy），就是我们不事先为每个原始类编写代理类，而是在运行的时候，动态地创建原始类对应的代理类，然后在系统中用代理类替换掉原始类。那如何实现动态代理呢？

如果你熟悉的是Java语言，实现动态代理就是件很简单的事情。因为Java语言本身就已经提供了动态代理的语法（实际上，动态代理底层依赖的就是Java的反射语法）。我们来看一下，如何用Java的动态代理来实现刚刚的功能。具体的代码如下所示。其中，MetricsCollectorProxy作为一个动态代理类，动态地给每个需要收集接口请求信息的类创建代理类。

```
public class MetricsCollectorProxy {
    private MetricsCollector metricsCollector;

    public MetricsCollectorProxy() {
        this.metricsCollector = new MetricsCollector();
    }

    public Object createProxy(Object proxiedObject) {
        Class<?>[] interfaces = proxiedObject.getClass().getInterfaces();
        DynamicProxyHandler handler = new DynamicProxyHandler(proxiedObject);
        return Proxy.newProxyInstance(proxiedObject.getClass().getClassLoader(), interfaces, handler);
    }

    private class DynamicProxyHandler implements InvocationHandler {
        private Object proxiedObject;

        public DynamicProxyHandler(Object proxiedObject) {
            this.proxiedObject = proxiedObject;
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            long startTimestamp = System.currentTimeMillis();
            Object result = method.invoke(proxiedObject, args);
            long endTimeStamp = System.currentTimeMillis();
            long responseTime = endTimeStamp - startTimestamp;
            String apiName = proxiedObject.getClass().getName() + ":" + method.getName();
            RequestInfo requestInfo = new RequestInfo(apiName, responseTime, startTimestamp);
            metricsCollector.recordRequest(requestInfo);
            return result;
        }
    }
}

//MetricsCollectorProxy使用举例
MetricsCollectorProxy proxy = new MetricsCollectorProxy();
UserController userController = (UserController) proxy.createProxy(new UserController());
```

实际上，Spring AOP底层的实现原理就是基于动态代理。用户配置好需要给哪些类创建代理，并定义好在执行原始类的业务代码前后执行哪些附加功能。Spring为这些类创建动态代理对象，并在JVM中替代原始类对象。原本在代码中执行的原始类的方法，被换作执行代理类的方法，也就实现了给原始类添加附加功能的目的。

代理模式的应用场景

代理模式的应用场景非常多，我这里列举一些比较常见的用法，希望你能举一反三地应用在你的项目开发中。

1.业务系统的非功能性需求开发

代理模式最常用的一个应用场景就是，在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。我们将这些附加功能与业务功能解耦，放到代理类中统一处理，让程序员只需要关注业务方面的开发。实际上，前面举的搜集接口请求信息的例子，就是这个应用场景的一个典型例子。

如果你熟悉Java语言和Spring开发框架，这部分工作都是可以在Spring AOP切面中完成的。前面我们也提到，Spring AOP底层的实现原理就是基于动态代理。

2.代理模式在RPC、缓存中的应用

实际上，RPC框架也可以看作一种代理模式，GoF的《设计模式》一书中把它称作远程代理。通过远程代理，将网络通信、数据编解码等细节隐藏起来。客户端在使用RPC服务的时候，就像使用本地函数一样，无需了解跟服务器交互的细节。除此之外，RPC服务的开发者也只需要开发业务逻辑，就像开发本地使用的函数一样，不需要关注跟客户端的交互细节。

关于远程代理的代码示例，我自己实现了一个简单的RPC框架Demo，放到了GitHub中，你可以点击这里的[链接](#)查看。

我们再来看代理模式在缓存中的应用。假设我们要开发一个接口请求的缓存功能，对于某些接口请求，如果入参相同，在设定的过期时间内，直接返回缓存结果，而不用重新进行逻辑处理。比如，针对获取用户个人信息的需求，我们可以开发两个接口，一个支持缓存，一个支持实时查询。对于需要实时数据的需求，我们让其调用实时查询接口，对于不需要实时数据的需求，我们让其调用支持缓存的接口。那如何实现接口请求的缓存功能呢？

最简单的实现方法就是刚刚我们讲到的，给每个需要支持缓存的查询需求都开发两个不同的接口，一个支持缓存，一个支持实时查询。但是，这样做显然增加了开发成本，而且会让代码看起来非常臃肿（接口个数成倍增加），也不方便缓存接口的集中管理（增加、删除缓存接口）、集中配置（比如配置每个接口缓存过期时间）。

针对这些问题，代理模式就能派上用场了，确切地说，应该是动态代理。如果是基于Spring框架来开发的话，那就可以在AOP切面中完成接口缓存的功能。在应用启动的时候，我们从配置文件中加载需要支持缓存的接口，以及相应的缓存策略（比如过期时间）等。当请求到来的时候，我们在AOP切面中拦截请求，如果请求中带有支持缓存的字段（比如http://...?..&cached=true），我们便从缓存（内存缓存或者Redis缓存等）中获取数据直接返回。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

1.代理模式的原理与实现

在不改变原始类（或叫被代理类）的情况下，通过引入代理类来给原始类附加功能。一般情况下，我们让代理类和原始类实现同样的接口。但是，如果原始类并没有定义接口，并且原始类代码并不是我们开发维护的。在这种情况下，我们可以通过让代理类继承原始类的方法来实现代理模式。

2.动态代理的原理与实现

静态代理需要针对每个类都创建一个代理类，并且每个代理类中的代码都有点像模板式的”重复“代码，增加了维护成本和开发成本。对于静态代理存在的问题，我们可以通过动态代理来解决。我们不事先为每个原始类编写代理类，而是在运行的时候动态地创建原始类对应的代理类，然后在系统中用代理类替换掉原始类。

3.代理模式的应用场景

代理模式常用在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。我们将这些附加功能与业务功能解耦，放到代理类统一处理，让程序员只需要关注业务方面的开发。除此之外，代理模式还可以用在RPC、缓存等应用场景中。

课堂讨论

1. 除了Java语言之外，在你熟悉的其他语言中，如何实现动态代理呢？
2. 我们今天讲了两种代理模式的实现方法，一种是基于组合，一种基于继承，请对比一下两者的优缺点。

欢迎留言和我分享你的思考，如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言



大土豆

争哥的专栏，真的是太影响我了，每个设计模式都贴近实战，无比通透，今年我做了一个很重要的决定，我要把23种设计模式，都用在项目中。

2020-02-21 09:48



Eden Ma

1、OC中通过runtime和分类来实现动态代理。

2、组合优势可以直接使用原始类实例,继承要通过代理类实例来操作,可能会导致有人用原始类有人用代理类.而继承可以不改变原始类代码来使用.

2020-02-21 12:04



小晏子

C#中可以通过emit技术实现动态代理。

基于继承的代理适合代理第三方类，jdk中的动态代理只能代理基于接口实现的类，无法代理不是基于接口实现的类。所以在spring中有提供基于jdk实现的动态代理和基于cglib实现的动态代理。

2020-02-21 10:45



LJK

是时候展示我动态语言Python的彪悍了，通过__getattr__和闭包的配合实现，其中有个注意点就是在获取target时不能使用self.target，不然会递归调用self.__getattr__导致堆栈溢出：

```
class RealClass(object):
```

```
def realFunc(self, s):
```

```
print(f"Real func is coming {s}")
```

```
class DynamicProxy(object):
```

```
def __init__(self, target):
```

```
self.target = target
```

```
def __getattr__(self, name):
```

```
target = object.__getattr__(self, "target")
```

```
attr = object.__getattr__(target, name)
```

```
def newAttr(*args, **kwargs):
```

```
print("Before Calling Func")
```

```
res = attr(*args, **kwargs)
```

```
print("After Calling Func")
```


return res
return newAttr

2020-02-21 02:59



Jeff.Smile

动态代理有两种:jdk动态代理和cglib动态代理。

2020-02-21 00:33



webmin

1. .net支持反射和动态代理，所以实现方式和java类似；golang目前看到的都是习惯使用代码生成的方式来达成，根据已有代码生成一份加壳代码，调用方使用加壳代码的方法，例好：easyJson给类加上序列化和反序列化功能；gomock生成mock代理。

2. 组合与继承的优缺点：

没有绝对的优缺点，要看场景比如：

当被代理的类所有功能都需要被代理时，使用继承方式就可以编译器检查（被代理类修改时编译期就可以检查出问题）；

当被代理的类只是部分功能需要被代理时，使用组合方式就可按需代理，但是如果原来不需要的，后来也需要了就比较尴尬了。

继承可能会让代理类被迫实现一些对代理类来说无意义代码，继承方式对代理类的侵入比较大，而组合的侵入影响比继承可控。

2020-02-21 11:14



贺宇

python的装饰器是不是代理模式

2020-02-23 10:13



小兵

组合模式的优点在于更加灵活，对于接口的所有子类都可以代理，缺点在于不需要扩展的方法也需要进行代理。

继承模式的优点在于只需要针对需要扩展的方法进行代理，缺点在于只能针对单一父类进行代理。

2020-02-23 09:42



javaadu

最熟悉的还只有Java，问题1看别同学留言学习了。

问题2我的思考如下：

基于组合

1. 优点：扩展性更好，多层增强下来也比较清楚

2. 缺点：对于三方库无法应用

基于继承

1. 优点：应用面更广

2. 缺点：代码多；多层增强的话，继承体系变得越来越复杂

2020-02-23 06:17



Frank

代理模式可以实现业务需求与非业务需求之间的解耦。这样一来使得业务列与非业务类职责更加单一，可维护性提高。代理有静态代理与动态代理。

静态代理有两种实现方式：基于接口+委派 和 基于类+继承覆写目标类方法。静态代理可适用于被代理类不多，不复杂，可控的情况下。其劣势在于：需要为每一个目标类创建代理类，一旦类增多，维护成本增加。一旦要扩展接口中的功能，代理类与被代理类都需要作相应的修改（违反开闭原则）；在做代理之前，被代理类所有的东西都需要已知，人工干预太多；

动态代理能弥补静态代理的问题，在代理之前，所有的东西可以是未知的。不事先为每个被代理类编写代理类，而是在运行时，动态地创建原始类对应的代理类，然后在系统中用代理类替换掉原始类。其劣势在于：必须要有接口的支持。如果需要绕开接口这一点，则使用cglib动态代理来实现。

回到代理模式的本质是实现业务与非业务之间的解耦。其可以应用在业务系统的非功能性需求开发，如日志，监控，限流，事务等。同时也应用于RPC,接口缓存等场景。

2020-02-22 22:45



Demon.Lee



如果这篇都看不懂，就真是我们自己的问题了。

2020-02-22 20:43



峰

我在想有木有可能需要在方法中间添加相关的逻辑。

2020-02-22 16:15



Yang

Java中的动态代理原理就是运行的时候通过asm在内存中生成一份字节码，而这个字节码就是代理类的字节码，通过System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");设置可以保存这份字节码，反编译后看下其源码就知道Java中的动态代理是什么原理了。

2020-02-22 12:56



天之炼狱

争哥，数据转换用什么样的设计模式比较好。比如数据库表转换到XML，或者是XML转换到数据库表，但XML的结构可能因地区要求还不一样。

2020-02-22 11:59



Wh1

终于等到代理模式

2020-02-21 23:30



岁月

课堂讨论题

1. 略

2. 组合模式有一个缺点，就是需要对原始类的全部方法都实现一遍。继承则没有这个问题了。不过继承确实无法对一些声明为final的方法进行代理了。

对我有一个疑问，就是代理模式只能代理接口方法吗？像有对象属性这些，貌似只能写一个getter方法来代理啊？如果是这样的话，那组合模式遇到那种有属性的类，岂不是要写一大堆getter方法了？而且其他设计模式好像都是关注接口里面的方法，并不会关注对象的属。大家能否指点一下，谢谢。（这个问题源于我在写自己的类库的时候，因为对象的类型声明为接口，所以使用对象的时候无法访问它的属性，给编码造成了一些麻烦）

最后顺便提一下，iOS中的代理模式跟这篇文章讲的思想有很大区别，iOS的代理类主要是用来做数据源，用户自己实现代理类中规定的接口，然后提供给使用类的对象用来获取数据。

2020-02-21 16:26



java中,动态代理的实现基于字节码生成技术(代码里就是newProxyInstance片段),可以在jvm运行时动态生成和加载字节码,类似的技术还有asm,cglib,javassist,平时编译java用的javac命令就是字节码生成技术的"老祖宗"

java中用到字节码生成技术的还有JSP编译器.AOP框架,反射等等

深入理解java虚拟机第三版里对动态代理的描述:

动态代理中所说的"动态",是针对使用Java代码实际编写了代理类的"静态"代理而言的,它的优势不在于省去了编写代理类那一点编码工作量,而是实现了可以在原始类和接口还未知的时候,就确定了代理类的行为,当代理类与原始类脱离直接联系后,就可以很灵活的重用于不同的应用场景之中

2020-02-21 13:56



SXPeople

组合代理比继承代理模式，在纵向横向的扩展都要灵活

继续跟着大牛 的脚步向前进

2020-02-21 11:16



Vicent

一：Objective-C中Delegate委托我一般叫做代理，oc中可以声明代理方法和代理对象，如果原有类想要增强，就要声明代理协议和持有代理对象，调用的时候去调用代理对象的代理方法，但我觉的这种编程方式应该不属于动态代理的范畴。因为即使是通过协议约定了方法（类似接口）但实现代码块是维护在一个单独类中，并且需要手动的实例化并传递给实用类。

二：

组合模式：

优点：觉得这种更灵活，可以根据接口定义不同的代理方法进行区分，职责分明，封装性好

缺点：代码量会增加；调用链会稍微增长；

继承模式：

优点是可以增强一些无法侵入修改的代码；

缺点是感觉耦合度会提高，如果要修改代理方法，必须要修改继承类，不如使用接口切换方便。有些类 final了，神仙难救啊。

2020-02-21 10:25



progyoung

以前准备面试的时候，总也搞不懂aop背后的原理，今天算是整明白了，谢谢小争哥。

2020-02-21 09:29