

## 48 | 代理模式：代理在RPC、缓存、监控等场景中的应用

2020-02-21 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 10:41 大小 8.57M



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：

 复制代码

```
1 public class UserController {
2     //...省略其他属性和方法...
3     private MetricsCollector metricsCollector; // 依赖注入
4
5     public UserVo login(String telephone, String password) {
6         long startTimestamp = System.currentTimeMillis();
7
8         // ... 省略login逻辑...
9
10        long endTimeStamp = System.currentTimeMillis();
11        long responseTime = endTimeStamp - startTimestamp;
12        RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
13        metricsCollector.recordRequest(requestInfo);
14
15        //...返回UserVo数据...
16    }
17
18    public UserVo register(String telephone, String password) {
19        long startTimestamp = System.currentTimeMillis();
20
21        // ... 省略register逻辑...
22
23        long endTimeStamp = System.currentTimeMillis();
24        long responseTime = endTimeStamp - startTimestamp;
25        RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
26        metricsCollector.recordRequest(requestInfo);
27
28        //...返回UserVo数据...
```

```
29     }
30 }
```

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 `IUserController`。 `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：

 复制代码

```
1  public interface IUserController {
2      UserVo login(String telephone, String password);
3      UserVo register(String telephone, String password);
4  }
5
6  public class UserController implements IUserController {
7      //...省略其他属性和方法...
8
9      @Override
10     public UserVo login(String telephone, String password) {
11         //...省略login逻辑...
12         //...返回UserVo数据...
13     }
14
15     @Override
16     public UserVo register(String telephone, String password) {
17         //...省略register逻辑...
18         //...返回UserVo数据...
19     }
20 }
21
22 public class UserControllerProxy implements IUserController {
23     private MetricsCollector metricsCollector;
24     private UserController userController;
25
26     public UserControllerProxy(UserController userController) {
27         this.userController = userController;
28         this.metricsCollector = new MetricsCollector();
29     }
30 }
```

```

31  @Override
32  public UserVo login(String telephone, String password) {
33      long startTimestamp = System.currentTimeMillis();
34
35      // 委托
36      UserVo userVo = userController.login(telephone, password);
37
38      long endTimeStamp = System.currentTimeMillis();
39      long responseTime = endTimeStamp - startTimestamp;
40      RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
41      metricsCollector.recordRequest(requestInfo);
42
43      return userVo;
44  }
45
46  @Override
47  public UserVo register(String telephone, String password) {
48      long startTimestamp = System.currentTimeMillis();
49
50      UserVo userVo = userController.register(telephone, password);
51
52      long endTimeStamp = System.currentTimeMillis();
53      long responseTime = endTimeStamp - startTimestamp;
54      RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
55      metricsCollector.recordRequest(requestInfo);
56
57      return userVo;
58  }
59 }
60
61 //UserControllerProxy使用举例
62 //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
63 //将UserController类对象替换为UserControllerProxy类对象，不需要改动太多代码
64 IUserController 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：

```
1 public class UserControllerProxy extends UserController {
2     private MetricsCollector metricsCollector;
3
4     public UserControllerProxy() {
5         this.metricsCollector = new MetricsCollector();
6     }
7
8     public UserVo login(String telephone, String password) {
9         long startTimestamp = System.currentTimeMillis();
10
11         UserVo userVo = super.login(telephone, password);
12
13         long endTimeStamp = System.currentTimeMillis();
14         long responseTime = endTimeStamp - startTimestamp;
15         RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);
16         metricsCollector.recordRequest(requestInfo);
17
18         return userVo;
19     }
20
21     public UserVo register(String telephone, String password) {
22         long startTimestamp = System.currentTimeMillis();
23
24         UserVo userVo = super.register(telephone, password);
25
26         long endTimeStamp = System.currentTimeMillis();
27         long responseTime = endTimeStamp - startTimestamp;
28         RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);
29         metricsCollector.recordRequest(requestInfo);
30
31         return userVo;
32     }
33 }
34 //UserControllerProxy使用举例
35 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不止一个，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。

 复制代码

```
1 public class MetricsCollectorProxy {
2     private MetricsCollector metricsCollector;
3
4     public MetricsCollectorProxy() {
5         this.metricsCollector = new MetricsCollector();
6     }
7
8     public Object createProxy(Object proxiedObject) {
9         Class<?>[] interfaces = proxiedObject.getClass().getInterfaces();
10        DynamicProxyHandler handler = new DynamicProxyHandler(proxiedObject);
11        return Proxy.newProxyInstance(proxiedObject.getClass().getClassLoader(), i
12    }
13
14    private class DynamicProxyHandler implements InvocationHandler {
15        private Object proxiedObject;
16
17        public DynamicProxyHandler(Object proxiedObject) {
18            this.proxiedObject = proxiedObject;
19        }
20
21        @Override
22        public Object invoke(Object proxy, Method method, Object[] args) throws Th
23            long startTimestamp = System.currentTimeMillis();
24            Object result = method.invoke(proxiedObject, args);
25            long endTimeStamp = System.currentTimeMillis();
26            long responseTime = endTimeStamp - startTimestamp;
27            String apiName = proxiedObject.getClass().getName() + ":" + method.getNa
28            RequestInfo requestInfo = new RequestInfo(apiName, responseTime, startTi
29            metricsCollector.recordRequest(requestInfo);
30            return result;
31        }
32    }
33 }
34
35 //MetricsCollectorProxy使用举例
```

```
36 MetricsCollectorProxy proxy = new MetricsCollectorProxy();
37 IUserController userController = (IUserController) proxy.createProxy(new UserC
```

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。那如何来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。

### 课程学习计划

# 关注极客时间服务号 每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 47 | 原型模式：如何最快速地clone一个HashMap散列表？

下一篇 49 | 桥接模式：如何实现支持不同类型和渠道的消息推送系统？

## 精选留言 (34)

写留言



大土豆

2020-02-21

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。

展开 ∨

💬 5

👍 7



**Eden Ma**

2020-02-21

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.

💬 1

👍 4



**Jeff.Smile**

2020-02-21

动态代理有两种:jdk动态代理和cglib动态代理。

展开 ∨

💬 2

👍 3



**小兵**

2020-02-23

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。

展开 ∨

💬

👍 2



**webmin**

2020-02-21

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：...

展开 ∨

💬

👍 2



**小晏子**

2020-02-21

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。

展开 ∨



👍 2



**LJK**

2020-02-21

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):...
```

展开 ∨



👍 2



**distdev**

2020-02-24

请问 如果对于业务方法 有多个非业务功能 比如metrics, logging还有其他的 应该实现在一个代理class里？ 还是一个filter chain里？

展开 ∨



👍 1



2020-02-21

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:...

展开 ∨



👍 1



**Summer 空城**

2020-02-21

老师好，有个地方不太明白，请指点下。

Spring框架实现AOP的时候是在BeanFactory中生成bean的时候触发动态代理替换成代理类的么？

如果我们自己想对某个Controller做代理的时候要怎么处理呢？一般是用@Controller注解

某个Controller的，而且这个Controller不会实现接口。...

展开 ▾

💬 5

👍 1



**辣么大**

2020-02-25

感谢争哥，今天终于学会了“动态代理”

还是要动手试试，代码在这 <https://bit.ly/37UqLNf>

学有余力的小伙伴，附上一些资料吧：

<https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html...>

展开 ▾

💬

👍



**cricket1981**

2020-02-25

@Aspect

@Component

```
public class CacheAspect {
```

```
    @Pointcut("execution(* com.example.demo.controller..*(..))")...
```

展开 ▾

💬

👍



**uranusleon**

2020-02-25

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。 -- 谁可以帮忙解释一下为什么替换成本较大？如果在代理类中调用计数器框架，后面如果更换计数器框架，则代理类中也需要修改，替换成本和不使用代理类也没有看出区别。

展开 ▾

💬

👍



**守拙**

2020-02-24

课堂讨论：

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。

...

展开 ▾



**不似旧日**

2020-02-24

笔记:

- 什么是代理模式：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。

...

展开 ▾



**xinquanv1**

2020-02-24

组合方式更好。组合更加灵活，特别是需要用到一些其他类提供的额外功能的时候，组合可以更好的实现复用，而继承则做不到，而且java只能单继承！

设计原则也有云:多用组合少用继承。●\_●。



**平风造雨**

2020-02-24

1. java和.net通过反射实现动态代理，其它语言比如python可以用自省来实现，本质都是利用类的metadata进行编程。

2. 组合的方式创建代理类显得更轻量 and 灵活，继承的层次多了，有时候会不太理解逻辑，方法不停的给override。

展开 ▾



**李小四**

2020-02-23

设计模式\_48:

我怎么感觉在代理这种用法中，组合的方式完胜呢。

展开 ▾



**相逢是缘**

2020-02-23

打卡

一、使用场景：

1) 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；

2) 在 RPC、缓存等应用场景； ...

展开 ▾



贺宇

2020-02-23

python的装饰器是不是代理模式

展开 ▾

