48-代理模式：代理在RPC、缓存、监控等场景中的应用

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：

```
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
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：

```
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。

```
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();
IUserController userController = (IUserController) 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.业务代码和框架代码解耦
2. 组合、依赖注入，委托
3. 应用场景

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。

精选留言：

- 大土豆 2020-02-21 09:48:20
  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。 [2赞]

- LJK 2020-02-21 02:59:58
  是时候展示我动态语言Python的彪悍了，通过__getattribute__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.__getattribute__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 __getattribute__(self, name):
target = object.__getattribute__(self, "target")
attr = object.__getattribute__(target, name)

def newAttr(*args, **kwargs):
print("Before Calling Func")
res = attr(*args, **kwargs)
print("After Calling Func")
return res
return newAttr [2赞]
```

- progyoung 2020-02-21 09:29:30
  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。

- 吴帆 2020-02-21 09:20:33
  看完争哥对代理模式的讲解，不仅明白了其原理和实现，更理解到了它的使用场景。感觉学习模式最精华的部分不在原理和实现，而在使用场景，争哥在这方面的内容做得很好，点赞。

- hanazawakana 2020-02-21 09:18:23
  组合比继承更灵活，但可能需要重复的代码实现，这就需要用到委托技术解决。继承更简单，如果相应类的行为不太会改变，可以用继承。

- Summer 空城 2020-02-21 09:15:23
  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。
  谢谢老师！

- FIGNT 2020-02-21 08:47:52
  只会java。基于接口和继承都是使用了多态的特性，在运行时用代理类替换。没有优劣之分，应用场景不同。就像争哥提到的，有些没有实现任何接口，而且不是我们开发的类。这时候就用基于继承的动态代理。

- Jeff.Smile 2020-02-21 00:33:56
  动态代理有两种:jdk动态代理和cglib动态代理。