71-命令模式：如何利用命令模式实现一个手游后端架构？

设计模式模块已经接近尾声了，现在我们只剩下3个模式还没有学习，它们分别是：命令模式、解释器模式、中介模式。这3个模式使用频率低、理解难度大，只在非常特定的应用场景下才会用到，所以，不是我们学习的重点，你只需要稍微了解，见了能认识就可以了。

今天呢，我们来学习其中的命令模式。在学习这个模式的过程中，你可能会遇到的最大的疑惑是，感觉命令模式没啥用，是一种过度设计，有更加简单的设计思路可以替代。所以，我今天讲解的重点是这个模式的设计意图，带你搞清楚到底什么情况下才真正需要使用它。

话不多说，让我们正式开始今天的学习吧！

命令模式的原理解读

命令模式的英文翻译是Command Design Pattern。在GoF的《设计模式》一书中，它是这么定义的：

> The command pattern encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

翻译成中文就是下面这样。为了帮助你理解，我对这个翻译稍微做了补充和解释，也一起放在了下面的括号中。

命令模式将请求（命令）封装为一个对象，这样可以使用不同的请求参数化其他对象（将不同请求依赖注入到其他对象），并且能够支持请求（命令）的排队执行、记录日志、撤销等（附加控制）功能。

对于GoF给出的定义，我这里再进一步解读一下。

落实到编码实现，命令模式用的最核心的实现手段，是将函数封装成对象。我们知道，C语言支持函数指针，我们可以把函数当作变量传递来传递去。但是，在大部分编程语言中，函数没法儿作为参数传递给其他函数，也没法儿赋值给变量。借助命令模式，我们可以将函数封装成对象。具体来说就是，设计一个包含这个函数的类，实例化一个对象传来传去，这样就可以实现把函数像对象一样使用。从实现的角度来说，它类似我们之前讲过的回调。

当我们把函数封装成对象之后，对象就可以存储下来，方便控制执行。所以，命令模式的主要作用和应用场景，是用来控制命令的执行，比如，异步、延迟、排队执行命令、撤销重做命令、存储命令、给命令记录日志等等，这才是命令模式能发挥独一无二作用的地方。

命令模式的实战讲解

上面的讲解比较偏理论，比较不好理解，我这里再结合一个具体的例子来解释一下。

假设我们正在开发一个类似《天天酷跑》或者《QQ卡丁车》这样的手游。这种游戏本身的复杂度集中在客户端。后端基本上只负责数据（比如积分、生命值、装备）的更新和查询，所以，后端逻辑相对于客户端来说，要简单很多。

考虑到你可能对游戏开发不熟悉，我这里稍微交代一些背景知识。

为了提高性能，我们会把游戏中玩家的信息保存在内存中。在游戏进行的过程中，只更新内存中的数据，游戏结束之后，再将内存中的数据存档，也就是持久化到数据库中。为了降低实现的难度，一般来说，同一个游戏场景里的玩家，会被分配到同一台服务上。这样，一个玩家拉取同一个游戏场景中的其他玩家的信息，就不需要跨服务器去查找了，实现起来就简单了很多。

一般来说，游戏客户端和服务器之间的数据交互是比较频繁的，所以，为了节省网络连接建立的开销，客户端和服务器之间一般采用长连接的方式来通信。通信的格式有多种，比如Protocol Buffer、JSON、XML，甚至可以自定义格式。不管是什么格式，客户端发送给服务器的请求，一般都包括两部分内容：指令和数据。其中，指令我们也可以叫作事件，数据是执行这个指令所需的数据。

服务器在接收到客户端的请求之后，会解析出指令和数据，并且根据指令的不同，执行不同的处理逻辑。对于这样的一个业务场景，一般有两种架构实现思路。

常用的一种实现思路是利用多线程。一个线程接收请求，接收到请求之后，启动一个新的线程来处理请求。具体点讲，一般是通过一个主线程来接收客户端发来的请求。每当接收到一个请求之后，就从一个专门用来处理请求的线程池中，捞出一个空闲线程来处理。

另一种实现思路是在一个线程内轮询接收请求和处理请求。这种处理方式不太常见。尽管它无法利用多线程多核处理的优势，但是对于IO密集型的业务来说，它避免了多线程不停切换对性能的损耗，并且克服了多线程编程Bug比较难调试的缺点，也算是手游后端服务器开发中比较常见的架构模式了。

我们接下来就重点讲一下第二种实现方式。

整个手游后端服务器轮询获取客户端发来的请求，获取到请求之后，借助命令模式，把请求包含的数据和处理逻辑封装为命令对象，并存储在内存队列中。然后，再从队列中取出一定数量的命令来执行。执行完成之后，再重新开始新的一轮轮询。具体的示例代码如下所示，你可以结合着一块看下。

```
public interface Command {
  void execute();
}

public class GotDiamondCommand implements Command {
  // 省略成员变量

  public GotDiamondCommand(/*数据*/) {
    //...
  }

  @Override
  public void execute() {
    // 执行相应的逻辑
  }
}
//GotStartCommand/HitObstacleCommand/ArchiveCommand类省略

public class GameApplication {
  private static final int MAX_HANDLED_REQ_COUNT_PER_LOOP = 100;
  private Queue<Command> queue = new LinkedList<>();

  public void mainloop() {
    while (true) {
      List<Request> requests = new ArrayList<>();
```

```
      //省略从epoll或者select中获取数据，并封装成Request的逻辑，
      //注意设置超时时间，如果很长时间没有接收到请求，就继续下面的逻辑处理。

      for (Request request : requests) {
        Event event = request.getEvent();
        Command command = null;
        if (event.equals(Event.GOT_DIAMOND)) {
          command = new GotDiamondCommand(/*数据*/);
        } else if (event.equals(Event.GOT_STAR)) {
          command = new GotStartCommand(/*数据*/);
        } else if (event.equals(Event.HIT_OBSTACLE)) {
          command = new HitObstacleCommand(/*数据*/);
        } else if (event.equals(Event.ARCHIVE)) {
          command = new ArchiveCommand(/*数据*/);
        } // ...一堆else if...

        queue.add(command);
      }

      int handledCount = 0;
      while (handledCount < MAX_HANDLED_REQ_COUNT_PER_LOOP) {
        if (queue.isEmpty()) {
          break;
        }
        Command command = queue.poll();
        command.execute();
      }
    }
  }
}
```

命令模式 VS 策略模式

看了刚才的讲解，你可能会觉得，命令模式跟策略模式、工厂模式非常相似啊，那它们的区别在哪里呢？不仅如此，在留言区中我还看到有不止一个同学反映，感觉学过的很多模式都很相似。不知道你有没有类似的感觉呢？

实际上，这个问题我之前简单提到过，可能没有作为重点来说，有些同学印象不是很深刻，这里我就再跟你讲一讲。

实际上，每个设计模式都应该由两部分组成：第一部分是应用场景，即这个模式可以解决哪类问题；第二部分是解决方案，即这个模式的设计思路和具体的代码实现。不过，代码实现并不是模式必须包含的。如果你单纯地只关注解决方案这一部分，甚至只关注代码实现，就会产生大部分模式看起来都很相似的错觉。

实际上，设计模式之间的主要区别还是在于设计意图，也就是应用场景。单纯地看设计思路或者代码实现，有些模式确实很相似，比如策略模式和工厂模式。

之前讲策略模式的时候，我们有讲到，策略模式包含策略的定义、创建和使用三部分，从代码结构上来，它非常像工厂模式。它们的区别在于，策略模式侧重“策略”或“算法”这个特定的应用场景，用来解决根据运行时状态从一组策略中选择不同策略的问题，而工厂模式侧重封装对象的创建过程，这里的对象没有任何业务场景的限定，可以是策略，但也可以是其他东西。从设计意图上来，这两个模式完全是两回事儿。

有了刚刚的铺垫，接下来，我们再来看命令模式跟策略模式的区别。你可能会觉得，命令的执行逻辑也可以

看作策略，那它是不是就是策略模式了呢？实际上，这两者有一点细微的区别。

在策略模式中，不同的策略具有相同的目的、不同的实现、互相之间可以替换。比如，BubbleSort、SelectionSort都是为了实现排序的，只不过一个是用冒泡排序算法来实现的，另一个是用选择排序算法来实现的。而在命令模式中，不同的命令具有不同的目的，对应不同的处理逻辑，并且互相之间不可替换。

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，你需要重点掌握的内容。

命令模式在平时工作中并不常用，你稍微了解一下就可以。今天，我重点讲解了它的设计意图，也就是能解决什么问题。

落实到编码实现，命令模式用到最核心的实现手段，就是将函数封装成对象。我们知道，在大部分编程语言中，函数是没法作为参数传递给其他函数的，也没法赋值给变量。借助命令模式，我们将函数封装成对象，这样就可以实现把函数像对象一样使用。

命令模式的主要作用和应用场景，是用来控制命令的执行，比如，异步、延迟、排队执行命令、撤销重做命令、存储命令、给命令记录日志等等，这才是命令模式能发挥独一无二作用的地方。

课堂讨论

从我们已经学过的这些设计模式中，找两个代码实现或者设计思路很相似的模式，说一说它们的不同点。

欢迎留言和我分享你的想法。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：

- qinsi 2020-04-15 06:17:19
  GoF的设计模式脱胎于开发smalltalk的经验，而smalltalk的一大特点是自带图形界面，所以很多设计模式都是脱胎于解决图形界面的开发问题。命令模式在早期图形界面的编辑器中用于解决编辑历史的问题，比如在实现操作的同时还要实现一个反操作用来undo，而重放操作就可以redo。由于现代计算机的存储成本比设计模式刚出现那会低太多了，于是通过快照方式来实现编辑历史也变得可行了 [10赞]

- eason2017 2020-04-15 07:51:03
  Hystix熔断框架就用到了命令模式。 [4赞]

- 大头 2020-04-15 05:55:10
  观察者模式和用接口实现的职责链模式，从设计意图来看，观察者重在状态变化时通知所有的观察者，观察者之间是并列关系。职责链模式侧重顺序处理，处理类之间是串行关系 [3赞]

- 饭 2020-04-15 18:13:03
  老师，我看不少讲策略模式的文章，喜欢用电商促销打折作为例子，但是我看每种打折方案是不能相互替换的，反倒适合命令模式或工厂 [2赞]

- 小晏子 2020-04-15 09:22:41
  再比较下命令模式和策略模式，策略模式的意图是封装算法，它认为“行为”是一个完整的、不可拆分的业务，即其意图是让这些行为独立，并且可以相互替换，让行为的变化独立于拥有行为的客户；而命令模式则是对动作的解耦，把一个动作的执行分为执行对象，执行行为，让两者相互独立而不相互影响。二者

的关注点不同：策略模式提供多种行为由调用者自己选用，算法的自由选用是其关注点。命令模式关注解耦，将请求的内容封装成命令由接受者执行。二者使用场景不同，策略模式适用于有多种行为可以相互替换的场景；而命令模式适用于解耦两个紧耦合关系的对象或多命令对撤销的场景。 [2赞]

- shniu 2020-04-15 08:17:18
  感觉redis就是使用了命令模式来处理指令的 [2赞]

- 黄林晴 2020-04-15 08:33:22
  打卡 结束后 开启重刷
  第一遍 了解定义
  第二遍代码实现
  第三遍 实际应用 [1赞]

- Ken张云忠 2020-04-15 08:26:14
  代理模式与装饰器模式代码差不多，代理模式是业务非功能性的增强，装饰器模式是对业务功能性的增强。 [1赞]

- 唔多志 2020-04-17 09:51:20
  敲黑板：设计模式之间的主要区别还是在于设计意图，也就是应用场景。是的，在代码层面，无非就是继承、组合、多态；但每一种模式最初的目的是不一样的，想要解决的问题也是不一样的。

- Hubbert伟涛 2020-04-15 23:22:10
  代理模式跟模板模式。虽然一个是结构型，一个是行为型，但是感觉他们有点相同之处。
  代理模式是在不改变原有类的基础上，对原有类进行功能的扩展，可以使附加功能与非业务逻辑解耦，更加关注业务逻辑。
  模板模式也是有两大作用，那就是复用和扩展。跟代理模式一样有个扩展功能。但却不是代码的扩展性，是指框架的扩展性。就例如测试框架，让框架用户可以在这些扩展点上扩展功能。
  它们的相同之处是可扩展，不相同之处是一个方便代码的扩展，一个方便框架的扩展。

- 大橙子 2020-04-15 11:02:54
  类与类之间通信：使用命令对象（包含数据与行为）。解耦：请求转换成命命令，命令调度，命令执行

- ， 2020-04-15 10:42:51
  单例模式和享元模式,缓存,静态工厂方法:
  他们的共同点是:对象的复用
  不同点:
  应用享元模式是为了对象"共享使用"，节省内存
  而应用单例/多例模式是为了限制对象的个数
  应用缓存是为了提高访问效率
  应用对象池(数据库连接池,线程池)是为了对象的"重复使用"和管理,主要是为了节省时间
  有一种方法叫 静态工厂方法,例如 Boolean.valueof(),不会在每次调用时返回一个新对象,而是复用已有的,这一点有点像享元模式

  适配器,代理,装饰器,桥接,:
  他们的共同点是:对方法的增强
  不同点:
  适配器模式的作用是"适配",通常用于适配不同的组件,新旧系统
  桥接模式将接口部分和实现部分分离,使两者可以分别扩展
  装饰者模式是对原始类功能进行增强，并且可以支持多次,多种增强

代理模式实现了代理类和原始类的解耦,使代理类可以用于增强不同的功能

策略模式与简单工厂模式,命令模式:
共同点:都有对if/else进行下沉
不同点:
策略模式根据运行时状态返回一个"策略"/"算法",这些"策略"具有相同目的,比如BubbleSort、SelectionSort 都是为了实现排序
命令模式中不同的命令具有不同的目的，对应不同的处理逻辑，并且互相之间不可替换
而简单工厂更侧重返回一个创建的对象

桥接模式与服务提供者框架,静态工厂的辨析(来自effective java):
服务提供者框架是这样一个系统:
多个服务提供者实现一个服务,系统为服务提供者的客户端提供多个实现,并把它们从多个实现中解耦出来
它分为四部分:
服务接口:系统抽象出一个接口,交给服务提供者实现(JDBC中的connection即为服务接口)
提供者注册API:交给服务提供者注册自己的API (Class.forName(),将自己的Driver类加载到JVM中,JDBC会查找该类并注册他的api)
服务访问API:是一个静态方法,供客户端获取服务实例(DriverManager.gtConnection()便是此处的静态工厂,它拿到的connection实例,其实是mysql包里的connection实现)
服务提供者接口:是服务接口实例的工厂对象(在此为mysql包内的Driver类)

在此处,桥接模式和静态工厂方法的使用,共同构成了服务提供者框架

- Heaven 2020-04-15 10:17:03
装饰器模式和适配器模式在组合的实现时候就很相似,但是装饰器模式是为了增强原有类的功能而适配器模式虽然也是修改原有类,但是是为了补救原有的缺陷的

- 守拙 2020-04-15 09:55:21
课堂讨论:
观察者模式(Observer Pattern)与职责链模式(Chain Of Responsibility Pattern)的实现方式是类似的:

在Observer Pattern中, Observable注册多个Observer, Observable发送事件时遍历所有Observer执行操作.

在COR Pattern中, Chain添加多个Responsibility Object, 事件触发时Chain遍历所有Responsibility Ojbect执行操作.

虽然UML类图和代码实现上两者十分相似, 不同之处在于Observer Pattern专注Observable与Observer的解耦, COR Pattern关注当事件触发时, 事件沿着链条传递, 链条上的对象顺序处理事件, 或拦截事件.

- Jeff.Smile 2020-04-15 09:02:00
“整个手游后端服务器轮询获取客户端发来的请求，获取到请求之后，借助命令模式，把请求包含的数据和处理逻辑封装为命令对象，并存储在内存队列中。然后，再从队列中取出一定数量的命令来执行。”
————————————————————
感觉不需要放入队列呢，直接执行命令对象呢？取出一定数量，指的是线程池里批处理吗？

- 马以 2020-04-15 00:49:26
设计模式接近尾声，坐等顶级互联网编程经验