56-观察者模式（上）：详解各种应用场景下观察者模式的不同实现方式

我们常把23种经典的设计模式分为三类：创建型、结构型、行为型。前面我们已经学习了创建型和结构型，从今天起，我们开始学习行为型设计模式。我们知道，创建型设计模式主要解决“对象的创建”问题，结构型设计模式主要解决“类或对象的组合或组装”问题，那行为型设计模式主要解决的就是“类或对象之间的交互”问题。

行为型设计模式比较多，有11个，几乎占了23种经典设计模式的一半。它们分别是：观察者模式、模板模式、策略模式、职责链模式、状态模式、迭代器模式、访问者模式、备忘录模式、命令模式、解释器模式、中介模式。

今天，我们学习第一个行为型设计模式，也是在实际的开发中用的比较多的一种模式：观察者模式。根据应用场景的不同，观察者模式会对应不同的代码实现方式：有同步阻塞的实现方式，也有异步非阻塞的实现方式；有进程内的实现方式，也有跨进程的实现方式。今天我会重点讲解原理、实现、应用场景。下一节课，我会带你一块实现一个基于观察者模式的异步非阻塞的EventBus，加深你对这个模式的理解。

话不多说，让我们正式开始今天的学习吧！

原理及应用场景剖析

观察者模式（Observer Design Pattern）也被称为**发布订阅模式**（Publish-Subscribe Design Pattern）。在GoF的《设计模式》一书中，它的定义是这样的：

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

翻译成中文就是：在对象之间定义一个一对多的依赖，当一个对象状态改变的时候，所有依赖的对象都会自动收到通知。

一般情况下，被依赖的对象叫作**被观察者**（Observable），依赖的对象叫作**观察者**（Observer）。不过，在实际的项目开发中，这两种对象的称呼是比较灵活的，有各种不同的叫法，比如：Subject-Observer、Publisher-Subscriber、Producer-Consumer、EventEmitter-EventListener、Dispatcher-Listener。不管怎么称呼，只要应用场景符合刚刚给出的定义，都可以看作观察者模式。

实际上，观察者模式是一个比较抽象的模式，根据不同的应用场景和需求，有完全不同的实现方式，待会我们会详细地讲到。现在，我们先来看其中最经典的一种实现方式。这也是在讲到这种模式的时候，很多书籍或资料给出的最常见的实现方式。具体的代码如下所示：

```
public interface Subject {
  void registerObserver(Observer observer);
  void removeObserver(Observer observer);
  void notifyObservers(Message message);
}

public interface Observer {
  void update(Message message);
}

public class ConcreteSubject implements Subject {
```

```
  private List<Observer> observers = new ArrayList<Observer>();

  @Override
  public void registerObserver(Observer observer) {
    observers.add(observer);
  }

  @Override
  public void removeObserver(Observer observer) {
    observers.remove(observer);
  }

  @Override
  public void notifyObservers(Message message) {
    for (Observer observer : observers) {
      observer.update(message);
    }
  }

}

public class ConcreteObserverOne implements Observer {
  @Override
  public void update(Message message) {
    //TODO: 获取消息通知，执行自己的逻辑...
    System.out.println("ConcreteObserverOne is notified.");
  }
}

public class ConcreteObserverTwo implements Observer {
  @Override
  public void update(Message message) {
    //TODO: 获取消息通知，执行自己的逻辑...
    System.out.println("ConcreteObserverTwo is notified.");
  }
}

public class Demo {
  public static void main(String[] args) {
    ConcreteSubject subject = new ConcreteSubject();
    subject.registerObserver(new ConcreteObserverOne());
    subject.registerObserver(new ConcreteObserverTwo());
    subject.notifyObservers(new Message());
  }
}
```

实际上，上面的代码算是观察者模式的“模板代码”，只能反映大体的设计思路。在真实的软件开发中，并不需要照搬上面的模板代码。观察者模式的实现方法各式各样，函数、类的命名等会根据业务场景的不同有很大的差别，比如register函数还可以叫作attach，remove函数还可以叫作detach等等。不过，万变不离其宗，设计思路都是差不多的。

原理和代码实现都非常简单，也比较好理解，不需要我过多的解释。我们还是通过一个具体的例子来重点讲一下，什么情况下需要用到这种设计模式？或者说，这种设计模式能解决什么问题呢？

假设我们在开发一个P2P投资理财系统，用户注册成功之后，我们会给用户发放投资体验金。代码实现大致是下面这个样子的：

```
public class UserController {
  private UserService userService; // 依赖注入
  private PromotionService promotionService; // 依赖注入

  public Long register(String telephone, String password) {
    //省略输入参数的校验代码
    //省略userService.register()异常的try-catch代码
    long userId = userService.register(telephone, password);
    promotionService.issueNewUserExperienceCash(userId);
    return userId;
  }
}
```

虽然注册接口做了两件事情，注册和发放体验金，违反单一职责原则，但是，如果没有扩展和修改的需求，现在的代码实现是可以接受的。如果非得用观察者模式，就需要引入更多的类和更加复杂的代码结构，反倒是一种过度设计。

相反，如果需求频繁变动，比如，用户注册成功之后，不再发放体验金，而是改为发放优惠券，并且还要给用户发送一封“欢迎注册成功”的站内信。这种情况下，我们就需要频繁地修改register()函数中的代码，违反开闭原则。而且，如果注册成功之后需要执行的后续操作越来越多，那register()函数的逻辑会变得越来越复杂，也就影响到代码的可读性和可维护性。

这个时候，观察者模式就能派上用场了。利用观察者模式，我对上面的代码进行了重构。重构之后的代码如下所示：

```
public interface RegObserver {
  void handleRegSuccess(long userId);
}

public class RegPromotionObserver implements RegObserver {
  private PromotionService promotionService; // 依赖注入

  @Override
  public void handleRegSuccess(long userId) {
    promotionService.issueNewUserExperienceCash(userId);
  }
}

public class RegNotificationObserver implements RegObserver {
  private NotificationService notificationService;

  @Override
  public void handleRegSuccess(long userId) {
    notificationService.sendInboxMessage(userId, "Welcome...");
  }
}

public class UserController {
  private UserService userService; // 依赖注入
  private List<RegObserver> regObservers = new ArrayList<>();

  // 一次性设置好，之后也不可能动态的修改
  public void setRegObservers(List<RegObserver> observers) {
    regObservers.addAll(observers);
  }
```

```
  public Long register(String telephone, String password) {
    //省略输入参数的校验代码
    //省略userService.register()异常的try-catch代码
    long userId = userService.register(telephone, password);

    for (RegObserver observer : regObservers) {
      observer.handleRegSuccess(userId);
    }

    return userId;
  }
}
```

当我们需要添加新的观察者的时候，比如，用户注册成功之后，推送用户注册信息给大数据征信系统，基于观察者模式的代码实现，UserController类的register()函数完全不需要修改，只需要再添加一个实现了RegObserver接口的类，并且通过setRegObservers()函数将它注册到UserController类中即可。

不过，你可能会说，当我们把发送体验金替换为发送优惠券的时候，需要修改RegPromotionObserver类中handleRegSuccess()函数的代码，这还是违反开闭原则呀？你说得没错，不过，相对于register()函数来说，handleRegSuccess()函数的逻辑要简单很多，修改更不容易出错，引入bug的风险更低。

前面我们已经学习了很多设计模式，不知道你有没有发现，实际上，**设计模式要干的事情就是解耦。创建型模式是将创建和使用代码解耦，结构型模式是将不同功能代码解耦，行为型模式是将不同的行为代码解耦，具体到观察者模式，它是将观察者和被观察者代码解耦。**借助设计模式，我们利用更好的代码结构，将一大坨代码拆分成职责更单一的小类，让其满足开闭原则、高内聚松耦合等特性，以此来控制和应对代码的复杂性，提高代码的可扩展性。

基于不同应用场景的不同实现方式

观察者模式的应用场景非常广泛，小到代码层面的解耦，大到架构层面的系统解耦，再或者一些产品的设计思路，都有这种模式的影子，比如，邮件订阅、RSS Feeds，本质上都是观察者模式。

不同的应用场景和需求下，这个模式也有截然不同的实现方式，开篇的时候我们也提到，有同步阻塞的实现方式，也有异步非阻塞的实现方式；有进程内的实现方式，也有跨进程的实现方式。

之前讲到的实现方式，从刚刚的分类方式上来看，它是一种同步阻塞的实现方式。观察者和被观察者代码在同一个线程内执行，被观察者一直阻塞，直到所有的观察者代码都执行完成之后，才执行后续的代码。对照上面讲到的用户注册的例子，register()函数依次调用执行每个观察者的handleRegSuccess()函数，等到都执行完成之后，才会返回结果给客户端。

如果注册接口是一个调用比较频繁的接口，对性能非常敏感，希望接口的响应时间尽可能短，那我们可以将同步阻塞的实现方式改为异步非阻塞的实现方式，以此来减少响应时间。具体来讲，当userService.register()函数执行完成之后，我们启动一个新的线程来执行观察者的handleRegSuccess()函数，这样userController.register()函数就不需要等到所有的handleRegSuccess()函数都执行完成之后才返回结果给客户端。userController.register()函数从执行3个SQL语句才返回，减少到只需要执行1个SQL语句就返回，响应时间粗略来讲减少为原来的1/3。

那如何实现一个异步非阻塞的观察者模式呢？简单一点的做法是，在每个handleRegSuccess()函数中，创

建一个新的线程执行代码。不过，我们还有更加优雅的实现方式，那就是基于EventBus来实现。今天，我们就不展开讲解了。在下一讲中，我会用一节课的时间，借鉴Google Guava EventBus框架的设计思想，手把手带你开发一个支持异步非阻塞的EventBus框架。它可以复用在任何需要异步非阻塞观察者模式的应用场景中。

刚刚讲到的两个场景，不管是同步阻塞实现方式还是异步非阻塞实现方式，都是进程内的实现方式。如果用户注册成功之后，我们需要发送用户信息给大数据征信系统，而大数据征信系统是一个独立的系统，跟它之间的交互是跨不同进程的，那如何实现一个跨进程的观察者模式呢？

如果大数据征信系统提供了发送用户注册信息的RPC接口，我们仍然可以沿用之前的实现思路，在handleRegSuccess()函数中调用RPC接口来发送数据。但是，我们还有更加优雅、更加常用的一种实现方式，那就是基于消息队列（Message Queue，比如ActiveMQ）来实现。

当然，这种实现方式也有弊端，那就是需要引入一个新的系统（消息队列），增加了维护成本。不过，它的好处也非常明显。在原来的实现方式中，观察者需要注册到被观察者中，被观察者需要依次遍历观察者来发送消息。而基于消息队列的实现方式，被观察者和观察者解耦更加彻底，两部分的耦合更小。被观察者完全不感知观察者，同理，观察者也完全不感知被观察者。被观察者只管发送消息到消息队列，观察者只管从消息队列中读取消息来执行相应的逻辑。

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。

设计模式要干的事情就是解耦，创建型模式是将创建和使用代码解耦，结构型模式是将不同功能代码解耦，行为型模式是将不同的行为代码解耦，具体到观察者模式，它将观察者和被观察者代码解耦。借助设计模式，我们利用更好的代码结构，将一大坨代码拆分成职责更单一的小类，让其满足开闭原则、高内聚低耦合等特性，以此来控制和应对代码的复杂性，提高代码的可扩展性。

观察者模式的应用场景非常广泛，小到代码层面的解耦，大到架构层面的系统解耦，再或者一些产品的设计思路，都有这种模式的影子，比如，邮件订阅、RSS Feeds，本质上都是观察者模式。不同的应用场景和需求下，这个模式也有截然不同的实现方式，有同步阻塞的实现方式，也有异步非阻塞的实现方式；有进程内的实现方式，也有跨进程的实现方式。

课堂讨论

1. 请对比一下“生产者-消费者”模型和观察者模式的区别和联系。
2. 除了今天提到的观察者模式的几个应用场景，比如邮件订阅，你还能想到有哪些其他的应用场景吗？

欢迎留言和我分享你的想法。如果有收获，欢迎你把这篇文章分享给你的朋友。

精选留言：

- Sinclairs 2020-03-11 04:26:48
  发布-订阅模型，是一对多的关系，可以以同步的方式实现，也可以以异步的方式实现。
  生产-消费模型，是多对多的关系，一般以异步的方式实现
  两者都可以达到解耦的作用 [9赞]

- 小兵 2020-03-11 07:13:16

区别在于生产消费模型以异步形式实现，消费者之间存在竞争关系。发布订阅以同步或异步的方式实现，订阅者之间没有竞争关系。联系在于两者在流程上都有先后关系。 [5赞]

- Yeoman 2020-03-11 09:48:34
一路学来，看着下面的评论越来越少，终于跟上进度，继续加油。小争哥的栏目做的真的很好，干货满满，与看书感觉完全不同，关键是对读者的思想启发深远。 [4赞]

- 侯金彪 2020-03-11 20:00:23
生产者消费者一条消息只能被一个消费者消费，实现上需要依赖队列，生产者消费者各自有独立的工作线程
观察者模式可以一个事件被多个观察者处理，观察者之间相互独立。实现本质上是接口回调。 [1赞]

- 忆水寒 2020-03-11 09:52:12
我负责的软件是一个网关软件，主要是协议转换并且与外部不同厂商的系统进行数据交换。
我目前采用分进程的方式，各个接口进程启动的时候会连接主进程，并在主进程进行注册。
主进程在有内容更新的时候会采用观察者模式群发给需要的接口进程。由接口进程去完成协议转换并发给外部厂商。 [1赞]

- Frank 2020-03-11 09:09:11
打卡 今日学习观察者模式，收获如下：观察者模式在开发中比较常见，小到代码层面的解耦，大到架构层面的系统结构设计，或者是一些产品的设计思路。之前在学习ActiveMQ的发布订阅模型的时候，就对观察者模式有过了解，但是只知道它使用到了观察者模式这种设计思路。通过今天的学习进一步理解了为什么需要总结出这么多的设计模式，回归到本质上还是为了写出“高质量”的代码，即满足单一职责，开闭原则、高内聚松耦合等特性，依次来控制和应对代码的复杂性，提高代码的可扩展性，可读性等。事件是不是也可以理解为观察者模式的一种实现？在JavaBeans的的架构中，事件是其核心特征之一, 在实际开发中事件也比较常见，比如AWT ，Netty编程中的IO事件，Spring和SpringBoot中的事件等。 [1赞]

- zx 2020-03-11 22:52:17
在UserController 中 没看到private List regObservers = new ArrayList<>(); 是如何添加被依赖的对象，下面setRegObservers的值，也不会有新的对象加入啊

- 小刀 2020-03-11 21:28:48
一对多 多对多

- jaryoung 2020-03-11 17:32:41
问题2，ddd的事件驱动?

- test 2020-03-11 12:50:19
观察者模式，一对多的关系；生产-消费者模式，多对多的关系，通常有一个中间件做消息管道。

- ， 2020-03-11 11:00:25
课后题:
1.个人认为生产者消费者模式可以属于观察者模式的一种更具体的描述,他们的相同之处是将生产者与消费者解耦,不同之处则是生产者消费者模式通常在不同的进程之间使用,而观察者模式则没有这个要求
2.项目中用到物联网设备,天线扫描仓库的库位,将扫描到的标签推送到第三方的应用里,在对第三方应用做扩展时,就是实现了一个接口,然后将它添加到一个list中,物联网设备就可以推到我的实现类里

- 大头 2020-03-11 10:40:43

非常期待下节课的学习，已经想好应用场景了。我们系统的员工会调用部门，调动后很多关联数据需要更新，觉得比较适合观察者模式，部门变化后通知不同的接口，做异步数据处理

- 守拙 2020-03-11 10:09:37
  1. 生产者-消费者 模式 是多对多依赖关系, 观察者模式是一对多依赖关系.
  2. 微信公众号符合观察者模式的定义: publisher发布消息, 所有observer都会收到消息.

- gogo 2020-03-11 09:48:33
  个人认为，广义上，"生产者-消费者"模型属于观察者模式

- Jackey 2020-03-11 09:48:15
  个人认为观察者模式的应用范围更广吧，两者的目的都是解耦，生产-消费模型主要用于系统间解耦。而观察者模式既可以用于线程内代码解耦（同步阻塞），也可以用于线程间解耦（异步非阻塞），还可以用于进程间解耦（RPC或生产-消费模型）

- 业余爱好者 2020-03-11 09:24:18
  好代码的设计有很多标准，可读性，扩展性等。设计模式主要解决了软件需求频繁变更面临的扩展性的问题。解决方法是面向接口编程。将变化点与稳定点分离，抽象出接口。

  设计模式不光适用于面向对象的代码设计。架构设计同样受益。其中很重要的能力就是抽象。

  把相似的东西放在一起比较是种很好的学习方法。

- 墨雨 2020-03-11 09:02:46
  类似于微信公众号吧

- 小晏子 2020-03-11 08:42:20
  生产者消费者模型和观察者模型的相同之处是一方数据状态变化，另一方获取通知并做相关工作，不同之处是生产者消费者模型是个异步模型，生产者不知道有多少消费者消费消息，而观察者模型是个同步模型，而且被观察者知道有有多少观察者观察它的状态变化。应用场景除了文中提到的，还有微博用户关注等。

- Jeff.Smile 2020-03-11 07:20:37
  观察者模式跟之前看到的回调模式有点像，许多worker订阅boss，worker干完活把结果回调通知给boss。worker就像是observer,boss就像是subject.不一样的是观察者模式是subject主动通知observers,而回调模式是worker干完活通知boss.
  还有个疑问:文中说的异步非阻塞模式去调用observers时，如果还想知道每个observer执行后的返回值怎么办？比如后边的逻辑依赖于这些返回值。请老师解答。