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

王争 2020-03-11

00:00 11:41
讲述：冯永吉 大小：10.71M

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：

```java
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);
    //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：

```java
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：

```java
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。