



下载APP



## 21 | 开放封闭原则：不改代码怎么写新功能？

2020-07-15 郑晔

软件设计之美

[进入课程 >](#)**讲述：郑晔**

时长 12:04 大小 27.65M



你好！我是郑晔。

上一讲，我们讲了一个最基础的设计原则：单一职责原则，从这个原则中，你知道了一个模块只应该包含来自同一个变化来源的内容。这一讲，我们来看下一个设计原则：开放封闭原则。



作为一名程序员，来了一个需求就要改一次代码，这种方式我们已经见怪不怪了，甚至已经变成了一种下意识的反应。修改也很容易，只要我们按照之前的惯例如法炮制就好了。

这是一种不费脑子的做法，却伴随着长期的伤害。每人每次都只改了一点点，但是，经过长期积累，再来一个新的需求，改动量就要很大了。而在这个过程中，每个人都很无辜，因为每个人都只是遵照惯例在修改。但结果是，所有人都受到了伤害，代码越来越难以维护。

既然“修改”会带来这么多问题，那我们可以不修改吗？开放封闭原则就提供了这样一个新方向。

## 不修改代码

开放封闭原则是这样表述的：


软件实体（类、模块、函数）应该对扩展开放，对修改封闭。

这个说法是 Bertrand Meyer 在其著作《面向对象软件构造》（Object-Oriented Software Construction）中提出来的，它给软件设计提出了一个极高的要求：不修改代码。

或许你想问，不修改代码，那我怎么实现新的需求呢？答案就是**靠扩展**。用更通俗的话来解释，就是新需求应该用新代码实现。

开放封闭原则向我们描述的是一个结果，就是我们可以不修改代码而仅凭扩展就完成新功能。但是，这个结果的前提是要在软件内部留好扩展点，而这正是需要我们去设计的地方。因为**每一个扩展点都是一个需要设计的模型**。

举个例子，假如我们正在开发一个酒店预订系统，针对不同的用户，我们需要计算出不同的房价。比如，普通用户是全价，金卡是 8 折，银卡是 9 折，代码写出来可能是这样的：

 复制代码

```
1 class HotelService {
2     public double getRoomPrice(final User user, final Room room) {
3         double price = room.getPrice();
4         if (user.getLevel() == Level.GOLD) {
5             return price * 0.8;
6         }
7
8         if (user.getLevel() == Level.SILVER) {
9             return price * 0.9;
```

```
10     }
11
12     return price;
13 }
14 }
```

这时，新的需求来了，要增加白金卡会员，给出 75 折的优惠，如法炮制的写法应该是这样的：

[复制代码](#)

```
1 class HotelService {
2     public double getRoomPrice(final User user, final Room room) {
3         double price = room.getPrice();
4         if (user.getLevel() == UserLevel.GOLD) {
5             return price * 0.8;
6         }
7
8         if (user.getLevel() == UserLevel.SILVER) {
9             return price * 0.9;
10        }
11
12        if (user.getLevel() == UserLevel.PLATINUM) {
13            return price * 0.75;
14        }
15
16        return price;
17    }
18 }
```

显然，这种做法就是修改代码的做法，每增加一个新的类型就要修改一次代码。但是，一个有各种级别用户的酒店系统肯定不只是房价有区别，提供的服务也可能有区别。可想而知，每增加一个用户级别，我们要改的代码就漫山遍野。

那应该怎么办呢？我们应该考虑如何把它设计成一个可以扩展的模型。在这个例子里面，既然每次要增加的是用户级别，而且各种服务的差异都体现在用户级别上，我们就需要一个用户级别的模型。在前面的代码里，用户级别只是一个简单的枚举，我们可以给它丰富一下：

[复制代码](#)

```
1 interface UserLevel {
2     double getRoomPrice(Room room);
3 }
```

```
4  }
5
6  class GoldUserLevel implements UserLevel {
7      public double getRoomPrice(final Room room) {
8          return room.getPrice() * 0.8;
9      }
10 }
11
12 class SilverUserLevel implements UserLevel {
13     public double getRoomPrice(final Room room) {
14         return room.getPrice() * 0.9;
15     }
```

我们原来的代码就可以变成这样：

[复制代码](#)

```
1  class HotelService {
2      public double getRoomPrice(final User user, final Room room) {
3          return user.getRoomPrice(room);
4      }
5  }
6
7  class User {
8      private UserLevel level;
9      ...
10
11     public double getRoomPrice(final Room room) {
12         return level.getRoomPrice(room);
13     }
14 }
```

这样一来，再增加白金用户，我们只要写一个新的类就好了：

[复制代码](#)

```
1  class PlatinumUserLevel implements UserLevel {
2      public double getRoomPrice(final Room room) {
3          return room.getPrice() * 0.75;
4      }
```

之所以我们可以这么做，是因为我们在代码里留好了扩展点：UserLevel。在这里，我们把原来的只支持枚举值的 UserLevel 升级成了一个有行为的 UserLevel。

经过这番改造，HotelService 的 getRoomPrice 这个方法就稳定了下来，我们就不需要根据用户级别不断地调整这个方法了。至此，我们就拥有了一个稳定的构造块，可以在后期的工作中把它当做一个稳定的模块来使用。

当然，在这个例子里，这个方法是比较简单的。而在实际的项目中，业务方法都会比较复杂。

## 构建扩展点

好，现在我们已经对开放封闭原则有了一个基本的认识。其实，我们都知道修改是不好的，道理我们都懂，就是在**代码层面**，有人就糊涂了。我做个类比你就知道了，比如说，如果我问你，你正在开发的系统有问题吗？相信大部人的答案都是有。

那我又问你，那你会经常性主动调整它吗？大部人都不会。为什么呢？因为它在线上运行得好好的，万一我调整它，调整坏了怎么办。是啊！你看，道理就是这么个道理，放在系统层面人人都懂，而在代码层面，却总是习惯性被忽视。

所以，我们写软件就应该提供一个又一个稳定的小模块，然后，将它们组合起来。一个经常变动的模块必然是不稳定的，用它去构造更大的模块，就是将隐患深埋其中。


你可能会说，嗯，我懂了，可我还是做不好啊！为什么我们懂了道理后，依旧过不好“这一关”呢？因为**阻碍程序员们构造出稳定模块的障碍，其实是构建模型的能力**。你可以回顾一下前面那段代码，看看让这段代码产生变化的 UserLevel 是如何升级成一个有行为的 UserLevel 的。

在讲封装的时候，我说过，封装的要点是行为，数据只是实现细节，而很多人习惯性的写法是面向数据的，这也是导致很多人在设计上缺乏扩展性思考的一个重要原因。

**构建模型的难点，首先在于分离关注点，这个我们之前说过很多次了，不再赘述，其次在于找到共性。**

在多态那一讲，我们说过，要构建起抽象就要找到事物的共同点，有了这个理解，我们看前面的例子应该还算容易理解。而在一个业务处理的过程中，发现共性这件事对很多人来说就已经开始有难度了。

我们再来看个例子，下面是一个常见的报表服务，首先我们取出当天的订单，然后生成订单的统计报表，还要把统计结果发送给相关的人等：

 复制代码

```
1 class ReportService {
2     public void process() {
3         // 获取当天的订单
4         List<Order> orders = fetchDailyOrders();
5         // 生成统计信息
6         OrderStatistics statistics = generateOrderStatistics(orders);
7         // 生成统计报表
8         generateStatisticsReport(statistics);
9         // 发送统计邮件
10        sendStatisticsByMail(statistics);
11    }
12 }
```

很多人在日常工作中写出的代码都是与此类似的，但这个流程肯定是比较僵化的。出现一个新需求就需要调整这段代码。我们这就有一个新需求，把统计信息发给另外一个内部系统，这个内部系统可以把统计信息展示出来，供外部合作伙伴查阅。该怎么做呢？

我们先分析一下，发送给另一个系统的内容是统计信息，在原有的代码里，前面两步分别是获取源数据和生成统计信息，后面两步分别是，生成报表和将统计信息通过邮件发送出去。

也就是说，后两步和即将添加的步骤有一个共同点，都使用了统计信息，这样我们就找到了它们的共性，所以，我们就可以用一个共同的模型去涵盖它们，比如，  
**OrderStatisticsConsumer**：

 复制代码

```
1 interface OrderStatisticsConsumer {
2     void consume(OrderStatistics statistics);
3 }
4
5 class StatisticsReporter implements OrderStatisticsConsumer {
6     public void consume(OrderStatistics statistics) {
7         generateStatisticsReport(statistics);
8     }
9 }
10
11 class StatisticsByMailer implements OrderStatisticsConsumer {
```



```
12     public void consume(OrderStatistics statistics) {
13         sendStatisticsByMail(statistics);
14     }
15 }
16
17 class ReportService {
18     private List<OrderStatisticsConsumer> consumers;
19
20     void process() {
21         // 获取当天的订单
22         List<Order> orders = fetchDailyOrders();
23         // 生成统计信息
24         OrderStatistics statistics = generateOrderStatistics(orders);
25
26         for (OrderStatisticsConsumer consumer: consumers) {
27             consumer.consume(statistics);
28         }
29     }
30 }
```

如此一来，我们的新需求也只要添加一个新的类就可以实现了：

[复制代码](#)

```
1 class StatisticsSender implements OrderStatisticsConsumer {
2     public void consume(final OrderStatistics statistics) {
3         sendStatisticsToOtherSystem(statistics);
4     }
5 }
```

你能看出来，在这个例子里，我们第一步做的事情还是分解，就是把一个一个的步骤分开，然后找出步骤之间相似的地方，由此构建出一个新的模型。

真实项目里的代码可能比这个代码要复杂，但其实，并不一定是业务逻辑复杂，而是代码本身写得复杂了。所以，我们要先根据上一讲的单一职责原则，将不同需求来源引起的变动拆分到不同的方法里，形成一个又一个的小单元，再来做我们这里的分析。

通过这个例子你也可以看出，在真实的项目中，想要达到开放封闭原则的要求并不是一蹴而就的。这里我们只是因为有了需求的变动，才提取出一个 `OrderStatisticsConsumer`。

未来可能还会有其他的变动，比如，生成报表的逻辑。到那时，也许我们还会提取出一个新的 `OrderStatisticsGenerator` 的接口。但总的来说，我们每做一次这种模型构建，最核

心的类就会朝着稳定的方向迈进一步。

所以，好的设计都会提供足够的扩展点给新功能去扩展。在《Unix 编程艺术》一书中，Unix 编程就提倡“提供机制，而不是策略”，这就是开放封闭原则的一种体现。

同样的，我们知道很多系统是有插件机制的，比如，很多人使用的 VIM 和 Emacs，离我们比较近的还有 Eclipse 和 Visual Studio Code，它们都体现着开放封闭原则。去了解它们的接口，我们就可以看到这个软件给我们提供的各种能力，这也是一种很好的学习方式。

开放封闭原则还可以帮助我们改进自己的系统，我们可以通过查看自己的源码控制系统，找出那些最经常变动的文件，它们通常都是没有满足开放封闭原则的，而这可以成为我们改进系统的起点。

## 总结时刻

今天，我们讲了开放封闭原则，软件实体应该对扩展开放，对修改封闭。简单地说，就是不要修改代码，新的功能要用新的代码实现。

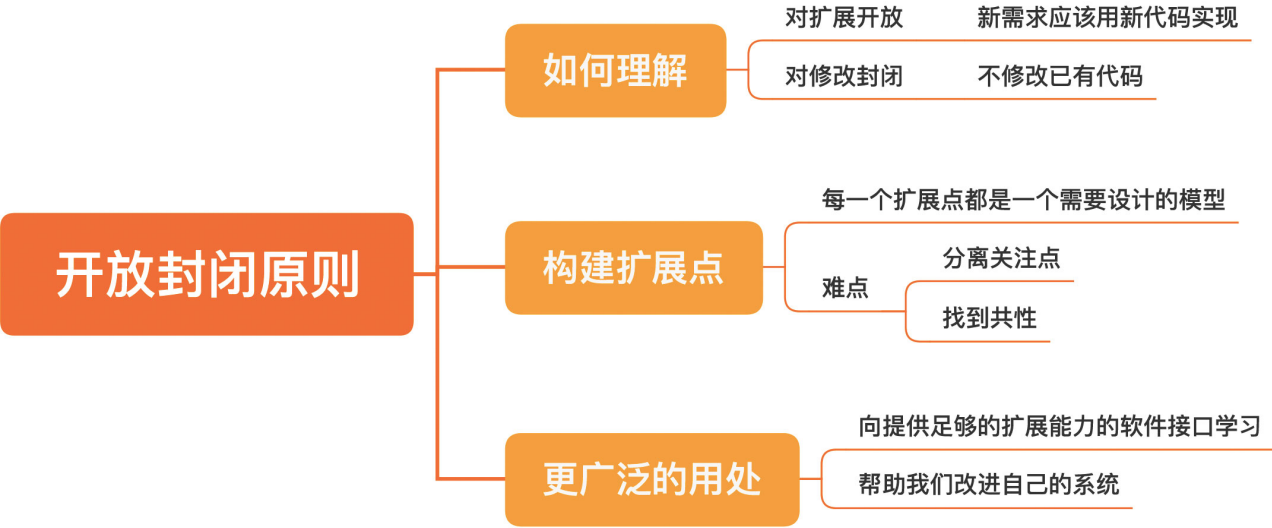
其实，道理大家都懂，但对很多人来说，做到是有难度的，尤其是在代码里留下扩展点，往往是需要有一定设计能力的。而构建模型的难点，首先就在于分离关注点，其次是找到共性。今天我们也讲了在一个真实项目中，怎样逐步地去构建扩展点，让系统稳定下来。

很多优秀的软件在设计上都给我们提供了足够的扩展能力，向这些软件的接口学习，我们可以学到更多的东西。

如果说单一职责原则主要看的还是封装，开放封闭原则就必须有多态参与其中了。显然，要想提供扩展点，就需要面向接口编程。但是，是不是有了接口，就是好的设计了呢？下一讲，我们来看设计一个接口还需要满足什么样的原则。

如果今天的内容你只能记住一件事，那请记住：**设计扩展点，迈向开放封闭原则。**





思考题

最后，我想请你找一个提供了扩展点的开源项目，分析一下它是如何设计这个扩展点的。欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

提建议

更多课程推荐

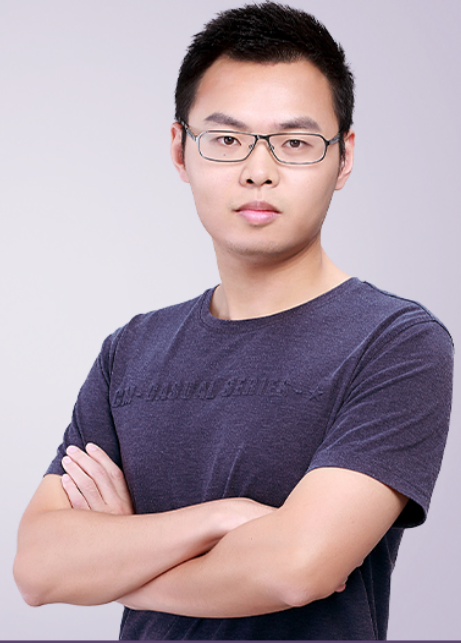
# 设计模式之美

前 Google 工程师手把手教你写高质量代码

王争

前 Google 工程师

《数据结构与算法之美》专栏作者



涨价倒计时 🕒

限时秒杀 **¥149**，7月31日涨价至 **¥299**

© 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 20 | 单一职责原则：你的模块到底为谁负责？

下一篇 22 | Liskov替换原则：用了继承，子类就设计对了吗？

## 精选留言 (6)

写留言



业余爱好者

2020-07-15

第一个案例感觉就是把user类改成了充血模型，这样确实合理一些，因为价格生成策略因用户不同而不同，同时又加入userlevel类，这样就更职责单一了。

第二个案例从方法命名上就可以看出职责不单一了，连原作者都不知道这个方法干了什...  
展开

作者回复: 去看看《Unix 编程艺术》，非常值得读的一本好书。

1

4

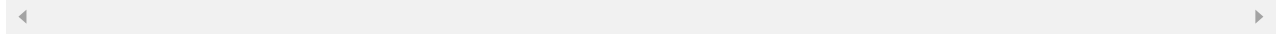
**liliumss**

2020-07-19

DDD的思路我觉得比较适合做，难就难到领域建模

展开 ▾

作者回复: DDD只能帮助你把骨架建起来，其中的细节，还是需要遵循着设计原则进行调整。



👍 3

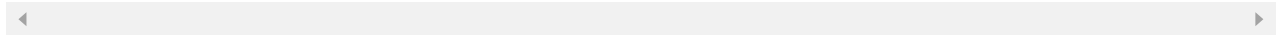
**Being**

2020-07-16

可以简单说下我们公司GIS平台的框架，也是插件的扩展机制。比如对于不同文件的格式解析和保存，抽象出DataSource模型和Saver模型，作为一类数据源注册进插件模块来扩展，而框架则提供类似驱动的能力，由用户组合需要的数据源放入驱动，然后通过驱动，来获得按流程处理后的文件导出。

展开 ▾

作者回复: 很好的分享！



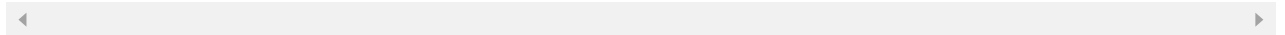
👍 2

**阳仔**

2020-07-15

- 1、识别修改点，构建模型，将原来静态的逻辑转为动态的逻辑
- 2、构建模型的难点在于分离关注点，其次就是找到共性

作者回复: 非常好的总结！



👍 2

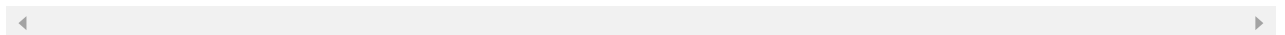
**桃子-夏勇杰**

2020-07-28

软件系统是变与不变的交融艺术，变化带来发展，不变的是本质，是共性。没有不变的变化只是绚丽的海市蜃楼，透过变化抓住不变，才是抓住了核心与要义。

展开 ▾

作者回复: 写出了一种诗意。





Harry

2020-07-23

第一个例子不太好理解，  
第二个就相对容易多了  
展开 ∨

