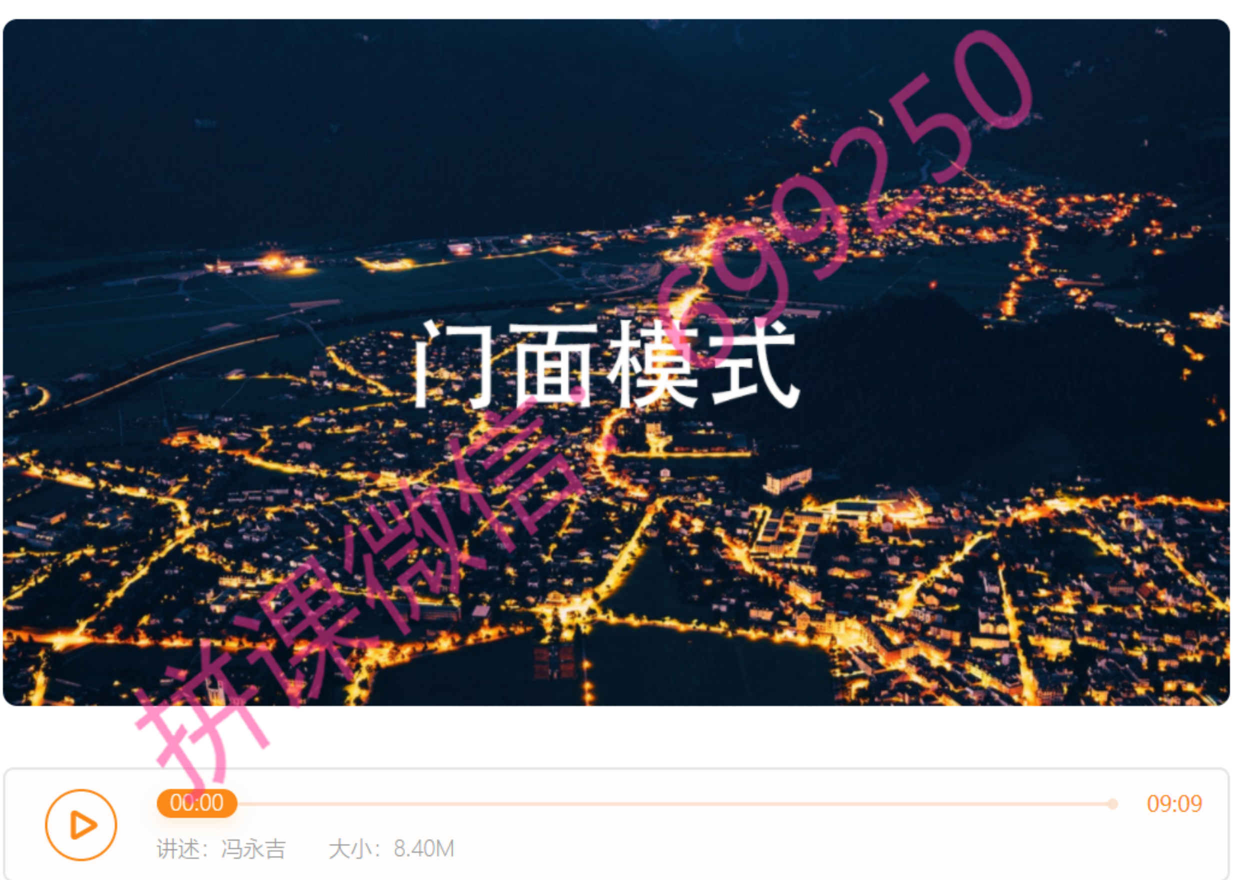
52 | 门面模式：如何设计合理的接口粒度以兼顾接口的易用性和通用性？

王争 2020-03-02



00:00 09:09
讲述：冯永吉 大小：8.40M

前面我们已经学习了代理模式、桥接模式、装饰器模式、适配器模式，这 4 种结构型设计模式。今天，我们再来学习一种新的结构型模式：门面模式。门面模式原理和实现都特别简单，应用场景也比较明确，主要在接口设计方面使用。

如果你平时的工作涉及接口开发，不知道你有没有遇到关于接口粒度的问题呢？

为了保证接口的可复用性（或者叫通用性），我们需要将接口尽量设计得细粒度一点，职责单一一点。但是，如果接口的粒度过小，在接口的使用者开发一个业务功能时，就会导致需要调用 n 多细粒度的接口才能完成。调用者肯定会抱怨接口不好用。

相反，如果接口粒度设计得太大，一个接口返回 n 多数据，要做 n 多事情，就会导致接口不够通用、可复用性不好。接口不可复用，那针对不同的调用者的业务需求，我们就需要开发不同的接口来满足，这就会导致系统的接口无限膨胀。

那如何来解决接口的可复用性（通用性）和易用性之间的矛盾呢？通过今天对于门面模式的学习，我想你心中会有答案。话不多说，让我们正式开始今天的学习吧！

门面模式的原理与实现

门面模式，也叫外观模式，英文全称是 Facade Design Pattern。在 GoF 的《设计模式》一书中，门面模式是这样定义的：

> Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

翻译成中文就是：门面模式为子系统提供一组统一的接口，定义一组高层接口让子系统更易用。

这个定义很简洁，我再进一步解释一下。

假设有一个系统 A，提供了 a、b、c、d 四个接口。系统 B 完成某个业务功能，需要调用 A 系统的 a、b、d 接口。利用门面模式，我们提供一个包裹 a、b、d 接口调用的门面接口 x，给系统 B 直接使用。

不知道你会不会有这样的疑问，让系统 B 直接调用 a、b、d 感觉没有太大问题呀，为什么还要提供一个包裹 a、b、d 的接口 x 呢？关于这个问题，我通过一个具体的例子来解释一下。

假设我们刚刚提到的系统 A 是一个后端服务器，系统 B 是 App 客户端。App 客户端通过后端服务器提供的接口来获取数据。我们知道，App 和服务器之间是通过移动网络通信的，网络通信耗时比较多，为了提高 App 的响应速度，我们要尽量减少 App 与服务器之间的网络通信次数。

假设，完成某个业务功能（比如显示某个页面信息）需要“依次”调用 a、b、d 三个接口，因自身业务的特点，不支持并发调用这三个接口。

如果我们现在发现 App 客户端的响应速度比较慢，排查之后发现，是因为过多的接口调用过多的网络通信。针对这种情况，我们就可以利用门面模式，让后端服务器提供一个包裹 a、b、d 三个接口调用的接口 x。App 客户端调用一次接口 x，来获取到所有想要的数据，将网络通信的次数从 3 次减少到 1 次，也就提高了 App 的响应速度。

这里举的例子只是应用门面模式的其中一个意图，也就是解决性能问题。实际上，不同的应用场景下，使用门面模式的意图也不同。接下来，我们就来看一下门面模式的各种应用场景。

门面模式的应用场景举例

在 GoF 给出的定义中提到，“门面模式让子系统更加易用”，实际上，它除了解决易用性问题之外，还能解决其他很多方面的问题。关于这一点，我总结罗列了 3 个常用的应用场景，你可以参考一下，举一反三地借鉴到自己的项目中。

除此之外，我还要强调一下，门面模式定义中的“子系统（subsystem）”也可以有多种理解方式。它既可以是一个完整的系统，也可以是更细粒度的类或者模块。关于这一点，在下面的讲解中也会有体现。

1. 解决易用性问题

门面模式可以用来封装系统的底层实现，隐藏系统的复杂性，提供一组更加简单易用、更高层的接口。比如，Linux 系统调用函数就可以看作一种“门面”。它是 Linux 操作系统暴露给开发者的一组“特殊”的编程接口，它封装了底层更基础的 Linux 内核调用。再比如，Linux 的 Shell 命令，实际上也可以看作一种门面模式的应用。它继续封装系统调用，提供更加友好、简单的命令，让我们可以直接通过执行命令来跟操作系统交互。

我们前面也多次讲过，设计原则、思想、模式很多都是相通的，是同一个道理不同角度的表述。实际上，从隐藏实现复杂性，提供更易用接口这个意图来看，门面模式有点类似之前讲到的迪米特法则（最少知识原则）和接口隔离原则：两个有交互的系统，只暴露有限的必要的接口。除此之外，门面模式还有点类似之前提到封装、抽象的设计思想，提供更抽象的接口，封装底层实现细节。

2. 解决性能问题

关于利用门面模式解决性能问题这一点，刚刚我们已经讲过了。我们通过将多个接口调用替换为一个门面接口调用，减少网络通信成本，提高 App 客户端的响应速度。所以，关于这点，我就不再举例说明了。我们来讨论一下这样一个问题：从代码实现的角度来看，该如何组织门面接口和非门面接口？

如果门面接口不多，我们完全可以将它跟非门面接口放到一块，也不需要特殊标记，当作普通接口来用即可。如果门面接口很多，我们可以在已有的接口之上，再重新抽象出一层，专门放置门面接口，从类、包的命名上跟原来的接口层做区分。如果门面接口特别多，并且很多都是跨多个子系统的，我们可以将门面接口放到一个新的子系统中。

3. 解决分布式事务问题

关于利用门面模式来解决分布式事务问题，我们通过一个例子来解释一下。

在一个金融系统中，有两个业务领域模型，用户和钱包。这两个业务领域模型都对外暴露了一系列接口，比如用户的增删改查接口、钱包的增删改查接口。假设有这样一个业务场景：在用户注册的时候，我们不仅会创建用户（在数据库 User 表中），还会给用户创建一个钱包（在数据库的 Wallet 表中）。

对于这样一个简单的业务需求，我们可以通过依次调用用户的创建接口和钱包的创建接口来完成。但是，用户注册需要支持事务，也就是说，创建用户和钱包的两个操作，要么都成功，要么都失败，不能一个成功、一个失败。

要支持两个接口调用在一个事务中执行，是比较难实现的，这涉及分布式事务问题。虽然我们可以通过引入分布式事务框架或者事后补偿的机制来解决，但代码实现都比较复杂。而最简单的解决方案是，利用数据库事务或者 Spring 框架提供的事务（如果是 Java 语言的话），在一个事务中，执行创建用户和创建钱包这两个 SQL 操作。这就要求两个 SQL 操作要在一个接口中完成，所以，我们可以借鉴门面模式的思想，再设计一个包裹这两个操作的新接口，让新接口在一个事务中执行两个 SQL 操作。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

我们知道，类、模块、系统之间的“通信”，一般都是通过接口调用来完成的。接口设计的好坏，直接影响到类、模块、系统是否好用。所以，我们要多花点心思在接口设计上。我经常说，**完成接口设计，就相当于完成了一半的开发任务。只要接口设计得好，那代码就差不到哪里去。**

接口粒度设计得太大，太小都不好。太大会导致接口不可复用，太小会导致接口不易用。在实际的开发中，接口的可复用性和易用性需要“微妙”的权衡。针对这个问题，我的一个基本的处理原则是，**尽量保持接口的可复用性，但针对特殊情况，允许提供冗余的门面接口，来提供更易用的接口**。

门面模式除了解决接口易用性问题之外，我们今天还讲到了其他 2 个应用场景，用它来解决性能问题和分布式事务问题。

课堂讨论

1. 适配器模式和门面模式的共同点是，将不好用的接口适配成好用的接口。你可以试着总结一下它们的区别吗？
2. 在你过往的项目开发中，有没有遇到过不合理的接口需求？又或者，有没有遇到过非常难用的接口？可以留言“吐槽”一下。

欢迎留言和我分享，如果有收获，也欢迎你把这篇文章分享给你的朋友。