23 | 接口隔离原则：接口里的方法，你都用得到吗？

2020-07-20 郑晔

软件设计之美 进入课程 >

讲述：郑晔
时长 14:16 大小 13.08M

你好！我是郑晔。

在前面几讲中，我们讲的设计原则基本上都是关于如何设计一个类。SRP 告诉我们，一个类的变化来源应该是单一的；OCP 说，不要随意修改一个类；LSP 则教导我们应该设计好类的继承关系。

而在面向对象的设计中，接口设计也是一个非常重要的组成部分。我们一直都在强调面向接口编程，想实现 OCP 也好，或者是下一讲要讲的 DIP 也罢，都是要依赖于接口实现的。

也许你会说，接口不就是一个语法吗？把需要的方法都放到接口里面，接口不就出来了吗？顶多是 Java 用 interface，C++ 都声明成纯虚函数。这种对于接口的理解，显然还停

留在语法的层面上。这样设计出来的只能算作是有了一个接口，但想要设计出好的接口，还要有在设计维度上的思考。

那什么样的接口算是一个好接口呢？这就需要我们了解接口隔离原则。

接口隔离原则

接口隔离原则（Interface segregation principle，简称 ISP）是这样表述的：

> 不应强迫使用者依赖于它们不用的方法。
> No client should be forced to depend on methods it does not use.

这个表述看上去很容易理解，就是指在接口中，不要放置使用者用不到的方法。站在使用者的角度，这简直再合理不过了。每个人都会觉得，我怎么会依赖于我不用的方法呢？相信作为设计者，你也会同意这种观点。然而，真正在设计的时候，却不是人人都能记住这一点的。

首先，很多程序员分不清使用者和设计者两个是不同的角色。因为在很多人看来，接口的设计和使用常常是由同一个人完成。这就是角色区分意识的缺失，这种缺失导致我们不能把两种不同的角色区分开来，本质上来说，这也是分离关注点没有做好的一种体现。

实际上，很多程序员在开发过程中，其实是两种角色都没有的，他们根本没有思考过接口的问题，因为他们更关心的是一个个的具体类。只有到了必须的时候，接口才作为语法选项使用一次，这种做法干脆就是没在设计上进行思考。

然而，你不设计接口，并不代表没有接口。

在做软件设计的时候，我们经常考虑的是模型之间如何交互，接口只是一个方便描述的词汇，为了让我们把注意力从具体的实现细节中抽离出来。但是，**如果没有设计特定的接口，你的一个个具体类就变成它的接口**。同设计不好的接口一样，这样的“接口”往往也是存在问题的。

那接口设计不好会有什么问题呢？典型的问题就是接口过“胖”，什么叫接口过“胖”呢？我给你举个例子。

胖接口减肥

假设有一个银行的系统，对外提供存款、取款和转账的能力。它通过一个接口向外部系统暴露了它的这些能力，而不同能力的差异要通过请求的内容来区分。所以，我们在这里设计了一个表示业务请求的对象，像下面这样：

复制代码

```
class TransactionRequest {
  // 获取操作类型
  TransactionType getType() {
    ...
  }

  // 获取存款金额
  double getDepositAmount() {
    ...
  }

  // 获取取款金额
  double getWithdrawAmount() {
    ...
  }

  // 获取转账金额
  double getTransferAmount() {
    ...
  }
}
```

每种操作类型都对应着一个业务处理的模块，它们会根据自己的需要，去获取所需的信息，像下面这样：

复制代码

```
interface TransactionHandler {
  void handle(TransactionRequest request);
}

class DepositHandler implements TransactionHandler {
  void handle(final TransactionRequest request) {
    double amount = request.getDepositAmount();
    ...
  }
}

class WithdrawHandler implements TransactionHandler {
```

```
  void handle(final TransactionRequest request) {
    double amount = request.getWithdrawAmount();
    ...
  }
}

class TransferHandler implements TransactionHandler {
  void handle(final TransactionRequest request) {
    double amount = request.getTransferAmount();
    ...
  }
`
```

这样一来，我们只要在收到请求之后，做一个业务分发就好了：

复制代码

```
TransactionHandler handler = handlers.get(request.getType());
if (handler != null) {
  handler.handle(request);
}
```

一切看上去都很好，不少人在实际工作中也会写出类似的代码。然而，在这个实现里，有一个接口就太“胖”了，它就是 TransactionRequest。

TransactionRequest 这个类包含了相关的请求内容，虽然这是无可厚非的。但是在这里，我们容易直觉地把它作为参数传给 TransactionHandler。于是，它作为一个请求对象，摇身一变，变成了业务处理接口的一部分。

正如我在前面所说的，虽然你没有设计特定的接口，但具体类可以变成接口。不过，作为业务处理中的接口，TransactionRequest 就显得“胖”了：

- getDepositAmount 方法只在 DepositHandler 里使用；
- getWithdrawAmount 方法只在 WithdrawHandler 里使用；
- getTransferAmount 只在 TransferHandler 使用。

然而，传给它们的 TransactionRequest 却包含所有这些方法。

也许你会想，这有什么问题吗？问题就在于，一个“胖”接口常常是不稳定的。比如说，现在要增加一个生活缴费的功能，TransactionRequest 就要增加一个获取生活缴费金额的方法：

复制代码

```
class TransactionRequest {
  ...

  // 获取生活缴费金额
  double getLivingPaymentAmount() {
    ...
  }
}
```

相应地，还需要增加业务处理的方法：

复制代码

```
class LivingPaymentHandler implements TransactionHandler {
  void handle(final TransactionRequest request) {
    double amount = request.getLivingPaymentAmount();
    ...
  }
}
```

虽然这种做法看上去还挺符合 OCP 的，但实际上，由于 TransactionRequest 的修改，前面几个写好的业务处理类：DepositHandler、WithdrawHandler、TransferHandler 都会受到影响。为什么这么说呢？

如果我们用的是一些现代的程序设计语言，你的感觉可能不明显。假如这段代码是用 C/C++ 这些需要编译链接的语言写成的，TransactionRequest 的修改势必会导致其它几个业务处理类重新编译，因为它们都引用了 TransactionRequest。

实际上，**C/C++ 的程序在编译链接上常常需要花很多时间，除了语言本身的特点之外，因为设计没做好，造成本来不需要重新编译的文件也要重新编译的现象几乎是随处可见的**。

你可以理解为，如果一个接口修改了，依赖它的所有代码全部会受到影响，而这些代码往往也有依赖于它们实现的代码，这样一来，一个修改的影响就传播出去了。用这种角度去

评估，你就会发现，不稳定的“胖”接口影响面是非常之广的，所以，我们说“胖”接口不好。

怎样修改这段代码呢？既然这个接口是由于“胖”造成的，给它减肥就好了。根据 ISP，只给每个使用者提供它们关心的方法。所以，我们可以引入一些“瘦”接口：

复制代码

```
interface TransactionRequest {
}

interface DepositRequest extends TransactionRequest {
  double getDepositAmount();
}

interface WithdrawRequest extends TransactionRequest {
  double getWithdrawAmount();
}

interface TransferRequest extends TransactionRequest {
  double getTransferAmount();
}

class ActualTransactionRequest implements DepositRequest, WithdrawRequest, Tra
  ...
}
```

这里，我们把 TransactionRequest 变成了一个接口，目的是给后面的业务处理进行统一接口，而 ActualTransactionRequest 则对应着原来的实现类。我们引入了 DepositRequest、WithdrawRequest、TransferRequest 等几个“瘦”接口，它们就是分别供不同的业务处理方法使用的接口。

有了这个基础，我们也可以改造对应的业务处理方法了：

复制代码

```
interface TransactionHandler<T extends TransactionRequest> {
  void handle(T request);
}


class DepositHandler implements TransactionHandler<DepositRequest> {
  void handle(final DepositRequest request) {
    double amount = request.getDepositAmount();
```

```
    ...
  }
}


class WithdrawHandler implements TransactionHandler<WithdrawRequest> {
  void handle(final WithdrawRequest request) {
    double amount = request.getWithdrawAmount();
    ...
  }
}


class TransferHandler implements TransactionHandler<TransferRequest> {
  void handle(final TransferRequest request) {
    double amount = request.getTransferAmount();
    ...
  }

```

经过这个改造，每个业务处理方法就只关心自己相关的业务请求。那么，新增生活缴费该如何处理呢？你可能已经很清楚了，就是再增加一个新的接口：

复制代码

```
interface LivingPaymentRequest extends TransactionRequest {
  double getLivingPaymentAmount();
}

class ActualTransactionRequest implements DepositRequest, WithdrawRequest, Tra
}
```

然后，再增加一个新的业务处理方法：

复制代码

```
class LivingPaymentHandler implements TransactionHandler<LivingPaymentRequest>
  void handle(final LivingPaymentRequest request) {
    double amount = request.getLivingPaymentAmount();
    ...
  }
}
```

我们可以对比一下两个设计，只有 ActualTransactionRequest 做了修改，而因为这个类表示的是实际的请求对象，在现在的结构之下，它是无论如何都要修改的。而其他的部分

因为不存在依赖关系，所以，并不会受到这次需求增加的影响。相对于原来的做法，新设计改动的影响面变得更小了。

你的角色

我们来回顾一下这个设计的改进过程，其中的重点就在于，原本那个大的 TransactionRequest 被拆分成了若干个小接口，每个小接口就只为特定的使用者服务。这样做的好处就在于，每个使用者只要关注自己所使用的方法就行，这样的接口才可能是稳定的，“胖”接口不稳定的原因就是，它承担了太多的职责。

或许你从这个讨论里听出了一点 SRP 的味道，没错，你甚至可以把 ISP 理解成接口设计的 SRP。

这个改进还有一个有趣的地方，ActualTransactionRequest 实现了多个接口。在这个设计里面，每个接口代表着与不同使用者交互的角色，Martin Fowler 将这种接口称为角色接口（Role Interface）。

这就像每个人在实际生活中扮演着不同的角色一样。在家里，我们是父母的子女；在公司里，我们是公司的员工；购物时，我们是顾客；出行时，我们是乘客，但所有这些角色最终都是由我们一个人承担的。前面讲做接口设计时，我们虽然是一个个体，但常常要同时扮演设计者和使用者两个不同的角色。而在这段代码里，各种角色则汇聚到了 ActualTransactionRequest 这个类上。

在一个设计中，识别出不同的角色是至关重要的。你可能又发现了，我想强调的还是分离关注点。

我们在讲多态的时候说过，接口是把变和不变隔离开。现在有了对 ISP 的理解，我们知道了，接口应该是尽可能稳定的。接口的使用者对于接口是一种依赖关系，被依赖的一方越稳定越好，而只有规模越小，才越有可能稳定下来。

我们还可以从更广泛的角度理解 ISP，就是不依赖于任何不需要的东西。我曾经遇到过一个项目，项目里的核心计算中依赖了一个非常小众的数据库，选择它的理由只是它提供了一个特有的功能。

然而，由于项目组人员变迁，结果是，大家除了知道这个特有的功能，对其他能力知之甚少。这个系统只要运行一段时间，数据库占据的空间就会膨胀到硬盘的极限，而只要重新把数据库中的数据导出导入一次，空间瞬间就小了许多（如果你好奇产生这个现象的原因，其实就是这个数据库鼓励的是不变风格，而核心计算中有大量的修改，产生了大量的修改日志，导出导入之后，日志就减少了）。

于是，我们只能通过加上硬盘监控，定期去导数据，以维持系统的正常运行。最后，大家忍无可忍，想办法把这个数据库换掉了。

之所以会依赖于这个数据库，是因为在技术选型时，我们用到了一个特定的框架，而这个框架缺省就依赖于这个数据库。开发人员为了快速实现，就把框架和数据库一起引入到了项目中，引发了后面的这些问题。

从这个例子中，你可以看出，在高层次上依赖于不需要的东西，这和类依赖于不需要的东西，其实是异曲同工的，由此可见，ISP 同样是一个可以广泛使用的设计原则。

总结时刻

今天，我们讨论了接口隔离原则，它告诉我们不应强迫使用者依赖于它们不用的方法。之所以要把这个原则列出来，很重要的一个原因就是很多接口设计得太“胖”了，里面包含了太多的内容，所以，一个更好的设计是，把大接口分解成一个一个的小接口。

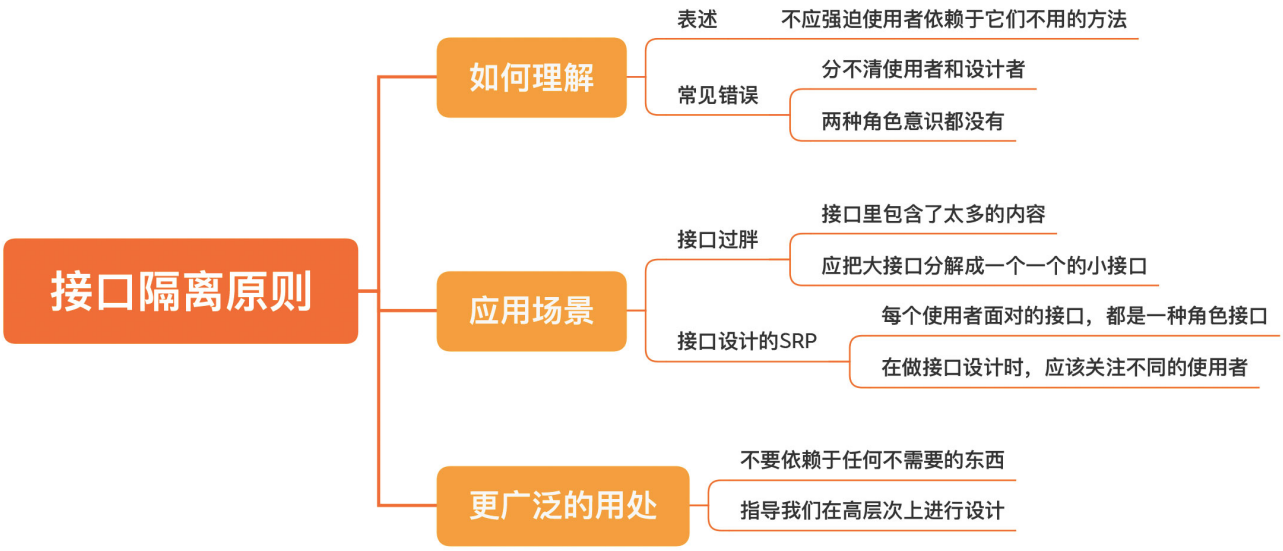
这里说的接口不仅仅是一种语法，实际上，每个类都有自己的接口，所有的公开方法都是接口。

我们在做接口设计时，需要关注不同的使用者。我们可以把 ISP 理解成接口设计的 SRP。每个使用者面对的接口，其实都是一种角色接口。识别出接口不同的角色是至关重要的，这也与分离关注点的能力是相关的。

ISP 还可以从更广泛的角度去理解，也就是说，不要依赖于任何不需要的东西，这个原则可以指导我们在高层次上进行设计。

在这一讲的案例里，除了接口太“胖”造成的问题，还有一个很重要的问题，它的依赖方向搞反了。我们下一讲就来讨论到底谁该依赖谁的设计原则：依赖倒置原则。

如果今天的内容你只能记住一件事，那请记住：**识别对象的不同角色，设计小接口。**



思考题

在今天的请求对象例子里面，为了支持生活付费，根据 ISP 原则，我改动了 ActualTransactionRequest，但其实这种做法一定程度上破坏了 OCP。你可以想一下，如何改进这个例子，能够让它更好地符合 OCP。 欢迎在留言区分享你的想法。

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

上一篇 22 | Liskov替换原则：用了继承，子类就设计对了吗？

下一篇 24 | 依赖倒置原则：高层代码和底层代码，到底谁该依赖谁？

精选留言 (10)

写留言

阳仔
2020-07-20

总结一下：软件设计时候需要从不同的用户角色来考虑，接口设计要尽量小。小接口其实也体现了单一职责原则，如果一个功能需要用到多个接口，那么可以通过组合（或者实现）各个小接口成一个大的接口。

作者回复: 分别用不同的接口就好了，不一定要组合。

2

shniu
2020-07-29

接口的定义不应该和具体的业务细节过度耦合，应该业务细节依赖更高层面的抽象

作者回复: 总结得不错。

shniu
2020-07-29

可以考虑把TransactionRequest 接口定义一个 getAmount 的行为，不同的业务场景，如充值/提现等去直接实现，这样简单直接，根据特定场景使用特定的对象，针对修改只需要扩展新的业务类，应该也符合ocp

Demon.Lee
2020-07-23

思考题：
把ActualTransactionRequest拆掉，DepositRequest, WithdrawRequest, TransferRequest 每个接口都有自己的实现类，这样就符合OCP了吧。
请老师指正。

展开

Being
2020-07-22

在LSP里面有提到公共接口是宝贵的资源，学习完ISP后，更觉得设计小接口的必要性了。

蓝士钦
2020-07-21

日常开发中常常为了写接口而写接口，一个service中的所有方法都提取到一个接口类中，感觉这样和不写接口没啥区别，因为这样实现类要实现接口类中的所有方法。
应该把不同行为的接口抽离出来，service组合这些接口实现类才比较灵活吧，controller不一定非得通过接口去调用service吧。毕竟controller的职责是处理入参校验，直接通过方法调用业务service没什么不妥吧

展开

桃子-夏勇杰
2020-07-20

郑老师，有金融类的，设计的比较好的，可以用来学习的开源项目么？如果是TDD方式开发的更佳。

2

Jxin
2020-07-20

接口隔离，感觉调用方在acl实现更合理些。毕竟每个调用方的关注点各不相同，服务提供方也没必要去感知各个调用方的关注点。

展开

1

OlafOO
2020-07-20

想到一个场景是之前在电商公司一个中台服务提供给外部业务部门的接口大部分都是胖接口

作者回复: 有什么不开心的，可以说出来开心一下。

骨汤鸡蛋面
2020-07-20

感觉开发经常忽视设计的一个重要原因就是：大部分开发日常工作就是用springmvc 写controller/service/dao，关于这个老师可以给一些建议嘛

展开

1