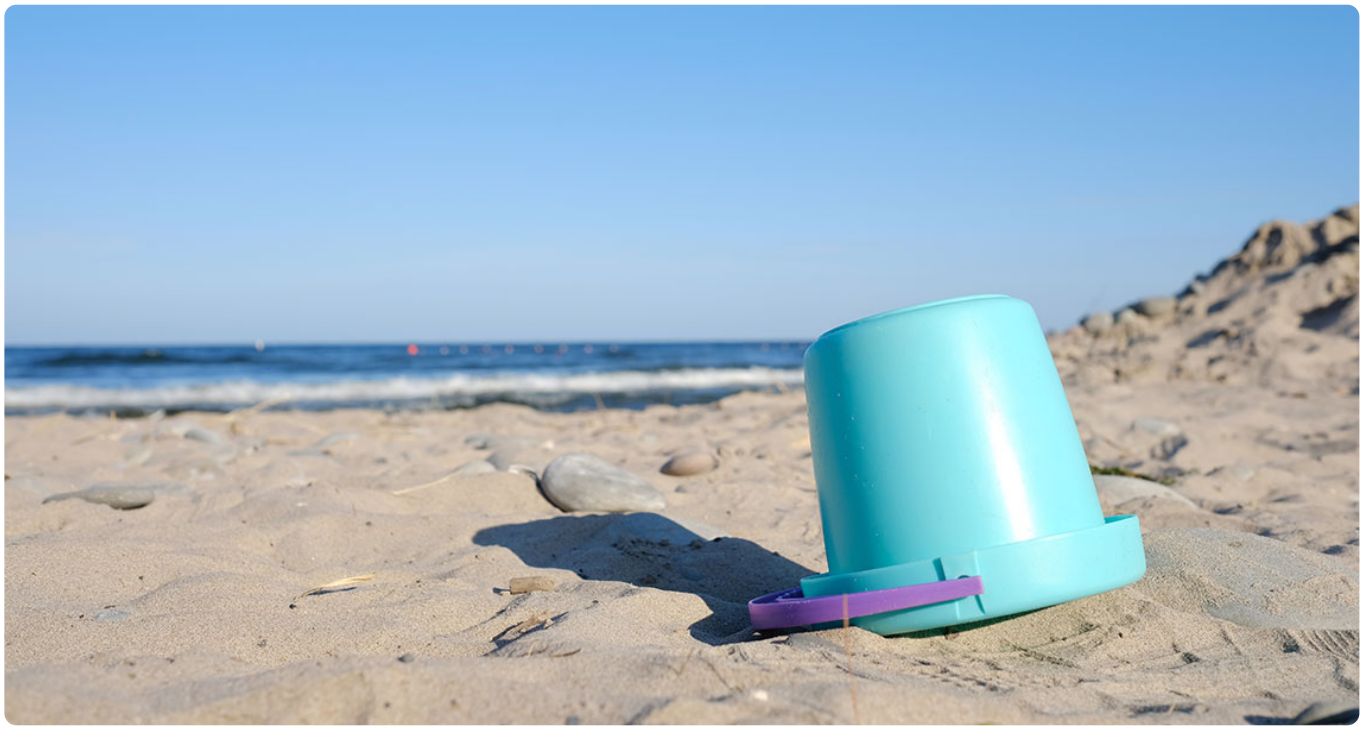
2 | 原则： ，到 ？

2020-07-22 郑晔

软件设计之美



讲述：郑晔

时长 13:38 大小 12.4 M

你好！我是郑晔。

上一讲，我们讲了 ISP 原则，知道了在设计接口的时候，我们应该设计小接口，不应该让使用者依赖于用不到的方法。但在结 的时候，我留下了一个 ，说在那个例子里面还有一个根本性的问题：依赖方向搞反了。

依赖这个词，程序员们都好理解，意思就是，我这段代码用到了谁，我就依赖了谁。依赖容易有，但能不能把依赖 对，就需要动点 子了。如果依赖关系没有处理好，就会导致一个小改动影响一大 ，而把依赖方向搞反，就是最典型的错 。

那什么叫依赖方向搞反呢？这一讲我们就来讨论关于依赖的设计原则：依赖倒置原则。

观供话观

，Deenen inesion inile D

> **igh-leel moules shoul no een on lo-leel moules. Boh shoul een on asaions.**

> **saions shoul no een on eails. Deails onee imlemenaions shoul een on asaions.**

汤

桃 杰 桃 不 都

都

不

夏

```
n   '  . .   - $/ $   '     / 0-    Å
o     +- $1  /     /   +n  . /   +n”
p     +- $1  /     /   +o  . /   +o”
q     ‘ ‘ ‘
r
s     1* $   - 0) ÁÂ  Å
t        ÇÇ
u        . /   +n‘   3    0/   ÁÂ”
v        ÇÇ
nm       . /   +o‘   3    0/   ÁÂ”
nn       ‘ ‘ ‘
no     Æ
np  Æ
```

桃 非 供话

Ciialeaue e e e

e

afa

afaoue

夏

```
class Handler {
  private  af aProducer producer;

  void execute() {
    ...
     essage message = ...;
    producer.send(new  af aRecord<>( topic , message);
    ...
  }
}
```

afa afaoue 具

afa 如 编

郑

具

如

觉 （ 趣省也 曲（ 非 下 供话

> ll olems in omue siene an e sole anohe leel of inieion
> Dai heele

D

层模块在这个过程中所承担的角色。

既然这个模块扮演的就是 息发 者的角色，那我们就可以引入一个 息发 者（MessageSender）的模型：

```
n  $)/ -!      . . "    )  - Å
o    1*$  . ) Á  . . "  ( . . " Â”
p  Æ
q
r   ' . .    ) ' - Å
s    +-$1 /     . . "    )  - . )  -”
t
u    1*$   3  0/ ÁÂ Å
v      ‘ ‘ ‘
nm       . . "  ( . . "  …‘ ‘ ‘ ”
nn      . )  -‘ . ) Á( . . " Â”
no      ‘ ‘ ‘
np    Æ
nq  Æ
```

有了 息发 者这个模型，那我们又该如何把 afka 和这个模型结合起来呢？那就要实现一个 afka 的 息发 者：

```
n   ' . .    !&   . . "    )  - $(+' ( )/ .    . . "    )  - Å
o    +-$1 /     !&  -* 0  -  +-* 0  -”
p
q    +0 '$  1*$  . ) Á!$) '    . . "  ( . . " Â Å
r      /#$.‘ +-* 0  -‘ . ) Á) 2   !&    *-  ‡ˆÁ›/*+$ ›’ ( . . "  ÂÂ”
s    Æ
t  Æ
```

这样一来，高层模块就不像原来一样 依赖 层模块，而是将依赖关系“倒置”过来，让 层模块去依赖由高层定义好的接口。这样做的好处就在于，将高层模块与 层实现解耦开来。

如果未来我们要替换掉 afka，只要重写一个 MessageSender 就好了，其他部分并不需要改变。这样一来，我们就可以让高层模块 持相对稳定，不会随着 层代码的改变而改变。

供话

供话

A 复制代码

在实际的项目中，这些编码规则有时候也并不是 对的。如果一个类特别稳定，我们也是可以直接用的，比如 符 类。但是，请注意，这种 非常少。因为大多数人写的代码稳定度并没有那么高。所以，上面几 编码规则可以成为 大部分 的规则，出现例外时，我们就需要特别关注一下。

到这里，你已经理解了在 DIP 的指导下，具体类还是能少用就少用。但还有一个问题，最终，具体类我们还是要用的，毕竟代码要运行起来不能只依赖于接口。那具体类应该在用呢？

我们讨论的这些设计原则，核心的关注点都是一个个的业务模型。此外，还有一些代码做的工作是 责把这些模型组 起来，这些 责组 的代码就需要用到一个一个的具体类。

是不是说到这里，感觉话题很 呢？是的，我们在 讲讨论过 DI 容 的来 去 ，在 Java 里，做这些组 工作的就是 DI 容 。

因为这些组 工作几 是 化的，而 非常 。如果你常用的语言中，没有提供 DI 容 ，最好还是把 责组 的代码和业务模型放到不同的代码里。

DI 容 在最 的讨论中有 外一个说法叫 IoC 容 ，这个 IoC 是 Inversion of Control 的 写，你会看到 IoC 和 DIP 中的 I 都是 inversion， 者表现的意 实际上是一致的。

理解了 DIP，再来使用 DI 容 ，你会觉得一切 理成 ，因为依赖之所以可以注入，是因为我们的设计 了 DIP。而只知道 DI 容 不了解 DIP，时常会出现让你觉得很为 的模型组 ，根本的原因就是设计没有做好。

关于 DIP，还有一个 象的说法，称为好 规则：“Dont call us, well call ou”。放在设计里面，这个 译应该是“别调用我，我会调你的”。显然，这是一个框架才会有的说法，有了一个稳定的抽象，各种具体的实现都应该是由框架去调用。

是的，如果你想去编写一个框架，理解 DIP 是非常重要的。 不 地说，不理解 DIP 的程序员，就只能写功能，不能构建出模型，也就很 再上一个台 。在前面讨论程序库时，我建议每个程序员都去 编写程序库，这其实就是让你去 构建模型的能力。

有了对 DIP 的讨论，我们再回过 看上一讲留下的 问，为什么说一开 TransactionRequest 是把依赖方向搞反了？因为最 的 TransactionRequest 是一个具体类，而 TransactionHandler 是业务类。

我们后来改进的版本里引入一个模型，把 TransactionRequest 变成了接口，ActualTransactionRequest 实现这个接口，TransactionHandler 只依赖于接口，而原来的具体类从这个接口继承而来，相对来说，比原来的版本好一些。

对 一个 言，了 不同 的 是一件很重要的 。你可以去 一些工具去生成项目的依赖关系 ，然后，你就可以用 DIP 作为一个评 ，去 量一下你的项目在依赖关系上表现得到底怎么样了。很有可能，你就 到了项目改造的一些着力点。

理解了 DIP，再来看一些关于依赖的讨论，我们也可以看到不同的角度。比如， 依赖，有人会说从技术上要如何解 它，但实际上， 依赖就是设计没有做好的结果，把依赖关系 错了，才可能会出现 依赖，先把设计做对，把该有的接口提取出来，依赖就不会 了。

至此，SOLID 的 个原则，我们已经讲了一 。有了前面对于分离关注点和面向对象基础知识的 ，相信你理解这些原则的 度也会相应的 了一些。

你会看到，理解这些原则，关 的 一 还是 **离 点**，把不同的内容区分开来。然后，用这些原则再把它们组合起来。而 你理解了这些原则，再回 去看，也能加 对面向对象特点的 识，现在你应该更能 体会到多态在面向对象 里发 的作用了。

总结时刻

今天我们讲了依赖倒置原则，它的表述是：

> 高层模块不应依赖于 层模块， 者应依赖于抽象。
>
> 抽象不应依赖于细节，细节应依赖于抽象。

理解这个原则的关 在于理解“倒置”，它是相对于传统自上而下的解 问题然后组合的方式而言的。高层模块不依赖于 层模块，可以通过引入一个抽象，或者模型，将 者解耦开来。高层模块依赖于这个模型，而 层模块实现这个模型。

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