Published in 程式愛好者

YC Following
Aug 24, 2020 · 6 min read

使人疯狂的SOLID 原则：单一职责原则(Single Responsibility Principle)

> *今天我们要说的是第一个原则：单一职责原则(SRP)。*

嗯.. 这真的是一个是一个害死人的名称，很多人会以为他意味每个模块都应该只做一件事。但真正的定义却并非如此：

> *按照惯例，先上定义：“**A class should have only one reason to change.**”*
> *翻译成中文是：**「一个模块应有且只有一个理由会使其改变。」***

但是在 Clear Architecture 一书中，作者说到软件系统改变是**为了满足用户与利益相关者**，所以我们应该把这理由套用到定义上，即变成：
「一个模块应只对唯一一个用户或利益相关者负责。」

可是如果现在有多个用户或利益相关者希望以相同的方式改变的话，我们不就会轻易地违反定义了吗?

所以我们需要为一群希望以相同的方式改变的用户或利益相关者一个定义，在这边作者称之为角色。

最终定义为：**「一个模块应只对唯一的一个角色负责。」**

对，疑问来了，角色是什么？这样做的好处是什么？违反这原则会带来什么坏处?

为什么原来的定义跟推导后得出的定义会差这么多？？？？？？？？

所以，要瞭解单一职责原则我们需要一些例子、一点更具体的想法。

让我们来看看第一个例子：
假如今天我们在做一个电商平台，而里面分别有计算消费税、信用卡手续费的 function，且同时应用在卖家后台与电商管理后台。

```
class feeCalculator {
    public function calcTax();
    public function calcCreditCardFee();
}
```

> *如果今天信用卡公司跟电商平台有了个协议，*
> *信用卡公司：「每笔单我给贵公司减少 5% 的抽成！」*
> *电商公司高兴地想：「那我就可以维持对消费者来 10% 的抽成，自己赚那5%，爽翻天啰！」*
> *然后马上请工程帅去改程式。*

故事的结果当然是工程帅去改同时面对卖家后台与电商后台的 calcCreditCardFee()，*然后公司就是少赚了*

*几千万，最后工程帅被炒了。*Happy Ending!

这边就可以回到「**一个模块应只对唯一的一个角色负责。**」的定义，今天因为工程师不知道原来这模块同时对两个角色负责了 — — 卖家后台与电商管理后台，**把不同角色所依赖的程序码放在一起**，他的改动同时造成两个角色的行为改变，最后导致错误发生。

而SRP 就是希望我们极力避免这种事情的发生，正是要限制改变带来的影响力。

那上面的错误我们又该如何做一个更好的处理呢?
最简单的想法就是为 calcCreditCardFee() 分别封装起来，以应对不同的角色

```
class SalesFeeCalculator {      // for 卖家
    public function calcTax();
    public function calcCreditCardFee();
}

class AdminFeeCalculator {      // for 后台
    public function calcTax();
    public function calcCreditCardFee();
}
```

OK，这样我们 10 个角色用到很类似的计算 function，我们就要 10 个 class，然后 class 里面的 function 都只有点一差异，这样好像又有点怪怪的?

那如果是透过控件来管理呢?

```
class FeeCalculator {

private role = null;

    public __construct(Role role) {
        this.role = role
    }
```

```
    public function calcTax();

public function calcCreditCardFee() {
        if (this.role == Sales) {
            return ....;
        }else if (this.role == Admin){
            return ....;
        }else{
            return ....;
        }
    }
}
```

这样的类别虽然可以保持简洁，我们又要面对在一个类别中会同时面向两个角色的问题。

往更小的分类方向去看，我们会发现一直会改变的是 calcCreditCardFee() 这个方法。如果我们把 calcCreditCardFee() 抽出来独立使用呢?

```
interface FeeCalculator {
    public function calc();
}

class CreditCardFeeHandler implements FeeCalculator {
    public function calc(percentage) {
        return ....;
    }
}

class TaxFeeHandler implements FeeCalculator {
    public function calc(percentage) {
        return ....;
    }
}
```

我们把 calcCreditCardFee() 变为 CreditCardFeeHandler 类别，再在里面创建一个 calc() 方法。这样我们不管是那位用户要使用计算信用卡手续费，只要透过 CreditCardFeeHandler 就可以实例化它的计算方法出来。

另外，可以看到我们使用了 interface 并分别实作到 CreditCardFeeHandler 与 TaxFeeHandler 上。这样我们的计算类别**都可以有效解耦**，而且在要注入的情况时，又可以获得**多态**的好处。

(不瞭解 interface？看看面向对象中的接口与抽象类别是什么？)

最后一个疑问，这样整个结构不就变成一个 function 一个 class 吗?
是有这样的可能的！但是我们要瞭解到，你要实作一个良好运作的模块，除了对外的 function 时，我们其实还有很多数据与方法是私有的。只要这些内容都指向同一个用户，很多 class 的 SRP 是可以被接受的。

总结一下，角色就是一群会使用该模块的用户，可能是真的人，也可能是别的模块。所以我们在考虑模块映射的角色时，可以从模块会被谁使用出发。

而这样做的好处就是可以「分开不同角色所依赖的程序码」，从而减少不同模块因过度耦合而在改变时所造成的错误，同时亦可以更容易的进行测试。

老实说 SRP 可能 SOLID 原则中最不好理解的，如果大家对我所理解的 SRP 有不一样的意见，欢迎下面留言一起讨论！

如果你觉得我的文章帮助到你，希望你也可以为文章拍手，分别 Follow 我的个人页与程式爱好者出版，按赞我们的粉丝页喔，支持我们推出更多更好的内容创作！

Rp Solid Single Responsibility 单一职责 Object Oriented