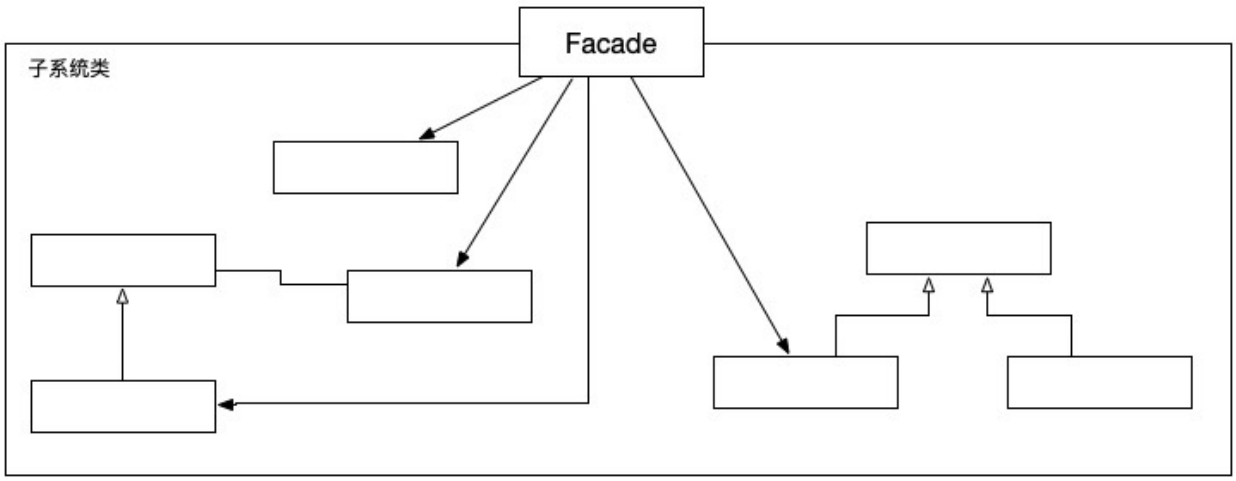
PHP设计模式之门面模式

门面模式，也叫外观模式。不管是门面还是外观，都是我们对外的媒介，就好像我们的脸面一样。所以，这个模式最大的特点就是要表现的“好看”。怎么说呢？一堆复杂的对象调用，自己都看蒙了，特别是对老系统进行升级维护的时候。用门面来把老系统的功能调用封装起来，在外面看来就和新系统一样，这就是门面模式的用途啦！

Gof类图及解释

GoF定义：为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

> GoF类图



> 代码实现

```php
class SubSystemOne
{
    public function MethodOne()
    {
        echo '子系统方法一', PHP_EOL;
    }
}
class SubSystemTwo
{
    public function MethodTwo()
    {
        echo '子系统方法二', PHP_EOL;
    }
}
```

```php
class SubSystemThree
{
    public function MethodThree()
    {
        echo '子系统方法三', PHP_EOL;
    }
}
class SubSystemFour
{
    public function MethodFour()
    {
        echo '子系统方法四', PHP_EOL;
    }
}
```

定义四个或者N多个子系统，这个没什么好说的吧，可以想象是很多子系统，而且他们之间并不一定和这四个子系统一样的相似，有可能是千差万别的。

```php
class Facade
{

    private $subStytemOne;
    private $subStytemTwo;
    private $subStytemThree;
    private $subStytemFour;
    public function __construct()
    {
        $this->subSystemOne = new SubSystemOne();
        $this->subSystemTwo = new SubSystemTwo();
        $this->subSystemThree = new SubSystemThree();
        $this->subSystemFour = new SubSystemFour();
    }

    public function MethodA()
    {
        $this->subSystemOne->MethodOne();
        $this->subSystemTwo->MethodTwo();
    }
```

```
    public function MethodB()
    {
        $this->subSystemOne->MethodOne();
        $this->subSystemTwo->MethodTwo();
        $this->subSystemThree->MethodThree();
        $this->subSystemFour->MethodFour();
    }
}
```

通过门面类将这些子系统包装起来，对外提供的只是门面新定义的方法。

```
$facade = new Facade();
$facade->MethodA();
$facade->MethodB();
```

客户端的调用就非常简单了，我们不用知道具体调用了哪些子系统，只需要知道门面的这些方法干什么了就行啦！

- 门面模式就是这么的简单，而且只要是真实的在项目中做过开发的朋友一定在不知不觉中就已经使用过这个模式了
- 当你需要为一个复杂子系统提供一个简单的接口时，门面模式就非常适用。同时，如果客户程序与抽象类的实现部分之间存在着很大的依赖性时，也可以引入门面模式来进行解耦，提高子系统的独立性和可维护性。另外就是你需要构建一个层次结构的子系统时，门面可以充当每层子系统的入口点
- Laravel中的门面系统相信使用过框架的人一定都用过，比如：Cache::put()。在Laravel中，门面的实现使用了一个魔术方法__callStatic()。然后让对象的方法可以实现直接使用静态方法来进行调用。是不是很神奇。有兴趣的朋友可以翻翻源码，就在/Illuminate/Support/Facades/Facade.php中。
- 划重点：三层结构或者MVC也是门面模式的体现哦。上面说了，门面模式适合分层子系统的维护。而三层结构、MVC、MVP、MVVM这些货，本质上都是为了分层，而分层的目的，就是为了降低系统的复杂性。

光卖我们的手机可不行，向X米一样做高科技的家电企业才是我们最终的目标。不过那么多的家电产品我们可生产不过来，于是，我们决定做一个商城（Facade）让一些质量非常好的商家加入我们的阵营，将他们的产品（SubSystem）放到商城中一起卖。当然，这些商品可是经过我们慎重挑选的，绝对都是优品中的优品哦！！

完整代码：https://github.com/zhangyue0503/designpatterns-php/blob/master/19.facade/source/facade.php

实例

这回我们将短信的发送以发送的维度进行包装，将不同的短信和推送运营商的接口包装起来，在发送的时候只需要通过发送类就可以控制用不同的第三方服务进行短信或推送的发送啦，想想都很方便呢！

> 短信发送类图

完整源码：https://github.com/zhangyue0503/designpatterns-php/blob/master/19.facade/source/facade-push.php

```php
<?php

class Send
{

    private $aliYunService;
    private $jiGuangService;

    private $message;
    private $push;

    public function __construct()
    {
        $this->aliYunService = new AliYunService();
        $this->jiGuangService = new JiGuangService();

        $this->message = new MessageInfo();
        $this->push = new PushInfo();
    }

    public function PushAndSendAliYun()
    {
        $this->message->Send($this->aliYunService);
        $this->push->Push($this->aliYunService);
    }

    public function PushAndSendJiGuang()
    {
        $this->message->Send($this->jiGuangService);
```

```
30          $this->push->Push($this->jiGuangService);
31      }
32  }
33
34  class MessageInfo
35  {
36      public function Send($service)
37      {
38          $service->Send();
39      }
40  }
41
42  class PushInfo
43  {
44      public function Push($service)
45      {
46          $service->Push();
47      }
48  }
49
50  class AliYunService
51  {
52      public function Send()
53      {
54          echo '发送阿里云短信! ', PHP_EOL;
55      }
56      public function Push()
57      {
58          echo '推送阿里云通知! ', PHP_EOL;
59      }
60  }
61
62  class JiGuangService
63  {
64      public function Send()
65      {
66          echo '发送极光短信! ', PHP_EOL;
67      }
68      public function Push()
69      {
```

```
70          echo '推送极光通知！', PHP_EOL;
71      }
72  }
73
74  $send = new Send();
75  $send->PushAndSendAliYun();
76  $send->PushAndSendJiGuang();
77
```

> 说明

- 依然还是熟悉的配方熟悉的味道。在这里，可以想象我们的第三方服务类都是较老的接口，或者已经是很复杂的接口了。这时，使用外观模式一来是可以与新系统配合，二来也能降低复杂度
- 但是要注意的，外观类本身可能成为复杂度的来源，不过幸好我们可以遵从单一职责的原则，一个外观类就做一件事就好啦