第 19 章　最少知识原则

最少知识原则（LKP）说的是一个软件实体应当尽可能少地与其他实体发生相互作用。这里的软件实体是一个广义的概念，不仅包括对象，还包括系统、类、模块、函数、变量等。本节我们主要针对对象来说明这个原则，下面引用《面向对象设计原理与模式》一书中的例子来解释最少知识原则：

> 某军队中的将军需要挖掘一些散兵坑。下面是完成任务的一种方式：将军可以通知上校让他叫来少校，然后让少校找来上尉，并让上尉通知一个军士，最后军士唤来一个士兵，然后命令士兵挖掘一些散兵坑。

这种方式十分荒谬，不是吗？不过，我们还是先来看一下这个过程的等价代码：

```
gerneral.getColonel( c ).getMajor( m ).getCaptain( c ) .getSergeant( s )
```

让代码通过这么长的消息链才能完成一个任务，这就像让将军通过那么多繁琐的步骤才能命令别人挖掘散兵坑一样荒谬！而且，这条链中任何一个对象的改动都会影响整条链的结果。

最有可能的是，将军自己根本就不会考虑挖散兵坑这样的细节信息。但是如果将军真的考虑了这个问题的话，他一定会通知某个军官：“我不关心这个工作如何完成，但是你得命令人去挖散兵坑。”

19.1　减少对象之间的联系

单一职责原则指导我们把对象划分成较小的粒度，这可以提高对象的可复用性。但越来越多的对象之间可能会产生错综复杂的联系，如果修改了其中一个对象，很可能会影响到跟它相互引用的其他对象。对象和对象耦合在一起，有可能会降低它们的可复用性。在程序中，对象的“朋友”太多并不是一件好事，“城门失火，殃及池鱼”和“一人犯法，株连九族”的故事时有发生。

最少知识原则要求我们在设计程序时，应当尽量减少对象之间的交互。如果两个对象之间不必彼此直接通信，那么这两个对象就不要发生直接的相互联系。常见的做法是引入一个第三者对象，来承担这些对象之间的通信作用。如果一些对象需要向另一些对象发起请求，可以通过第三者对象来转发这些请求。

19.2　设计模式中的最少知识原则

最少知识原则在设计模式中体现得最多的地方是中介者模式和外观模式，下面我们分别进行介绍。

1. 中介者模式

在第14章我们曾讲过一个博彩公司的例子。

在世界杯期间购买足球彩票，如果没有博彩公司作为中介，上千万的人一起计算赔率和输赢绝对是不可能的事情。博彩公司作为中介，每个人都只和博彩公司发生关联，博彩公司会根据所有人的投注情况计算好赔率，彩民们赢了钱就从博彩公司拿，输了钱就赔给博彩公司。

中介者模式很好地体现了最少知识原则。通过增加一个中介者对象，让所有的相关对象都通过中介者对象来通信，而不是互相引用。所以，当一个对象发生改变时，只需要通知中介者对象即可。

2. 外观模式

我们在第二部分没有提到外观模式，是因为外观模式在JavaScript中的使用场景并不多。外观模式主要是为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使子系统更加容易使用，如图19-1所示。

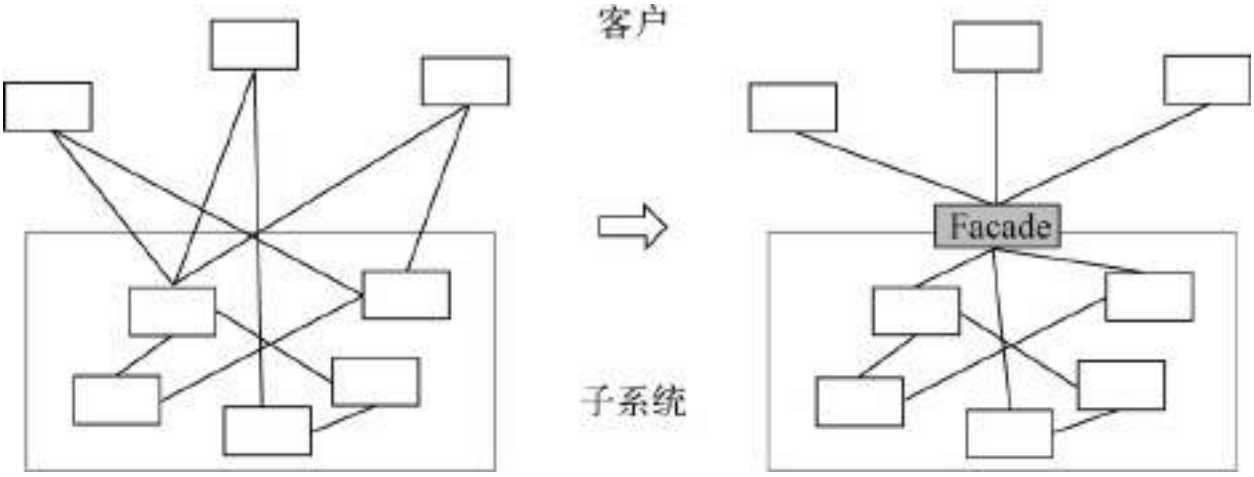

图　19-1

外观模式的作用是对客户屏蔽一组子系统的复杂性。外观模式对客户提供一个简单易用的高层接口，高层接口会把客户的请求转发给子系统来完成具体的功能实现。大多数客户都可以通过请求外观接口来达到访问子系统的目的。但在一段使用了外观模式的程序中，请求外观并不是强制的。如果外观不能满足客户的个性化需求，那么客户也可以选择越过外观来直接访问子系统。

拿全自动洗衣机的一键洗衣按钮举例，这个一键洗衣按钮就是一个外观。如果是老式洗衣机，客户要手动选择浸泡、洗衣、漂洗、脱水这4个步骤。如果这种洗衣机被淘汰了，新式洗衣机的漂洗方式发生了改变，那我们还得学习新的漂洗方式。而全自动洗衣机的好处很明显，不管洗衣机内部如何进化，客户要操作的，始终只是一个一键洗衣的按钮。这个按钮就是为一组子系统所创建的外观。但如果一键洗衣程序设定的默认漂洗时间是20分钟，而客户希望这个漂洗时间是30分钟，那么客户自然可以选择越过一键洗衣程序，自己手动来控制这些“子系统”运转。

外观模式容易跟普通的封装实现混淆。这两者都封装了一些事物，但外观模式的关键是定义一个高层接口去封装一组“子系统”。子系统在C++或者Java中指的是一组类的集合，这些类相互协作可以组成系统中一个相对独立的部分。在JavaScript中我们通常不会过多地考虑“类”，如果将外观模式

映射到JavaScript中，这个子系统至少应该指的是一组函数的集合。

最简单的外观模式应该是类似下面的代码:

```
var A = function(){
    a1();
    a2();
}

var B = function(){
    b1();
    b2();
}

var facade = function(){
    A();
    B();
}

facade();
```

许多JavaScript设计模式的图书或者文章喜欢把jQuery的`$.ajax` 函数当作外观模式的实现，这是不合适的。如果`$.ajax` 函数属于外观模式，那几乎所有的函数都可以被称为“外观模式”。问题是我们根本没有办法越过`$.ajax` “外观”去直接使用该函数中的某一段语句。

现在再来看看外观模式和最少知识原则之间的关系。外观模式的作用主要有两点。

- 为一组子系统提供一个简单便利的访问入口。
- 隔离客户与复杂子系统之间的联系，客户不用去了解子系统的细节。

从第二点来，外观模式是符合最少知识原则的。比如全自动洗衣机的一键洗衣按钮，隔开了客户和浸泡、洗衣、漂洗、脱水这些子系统的直接联系，客户不用去了解这些子系统的具体实现。

假设我们在编写这个老式洗衣机的程序，客户至少要和浸泡、洗衣、漂洗、脱水这4个子系统打交道。如果其中的一个子系统发生了改变，那么客

户的调用代码就得发生改变。而通过外观将客户和这些子系统隔开之后，如果修改子系统内部，只要外观不变，就不会影响客户的调用。同样，对外观的修改也不会影响到子系统，它们可以分别变化而互不影响。

19.3 封装在最少知识原则中的体现

封装在很大程度上表达的是数据的隐藏。一个模块或者对象可以将内部的数据或者实现细节隐藏起来，只暴露必要的接口API供外界访问。对象之间难免产生联系，当一个对象必须引用另外一个对象的时候，我们可以让对象只暴露必要的接口，让对象之间的联系限制在最小的范围之内。

同时，封装也用来限制变量的作用域。在JavaScript中对变量作用域的规定是：

- 变量在全局声明，或者在代码的任何位置隐式申明（不用`var`），则该变量在全局可见；
- 变量在函数内显式申明（使用`var`），则在函数内可见。

把变量的可见性限制在一个尽可能小的范围内，这个变量对其他不相关模块的影响就越小，变量被改写和发生冲突的机会也越小。这也是广义的最少知识原则的一种体现。

假设我们要编写一个具有缓存效果的计算乘积的函数`function mult(){}`，我们需要一个对象`var cache = {}`来保存已经计算过的结果。`cache`对象显然只对`mult`有用，把`cache`对象放在`mult`形成的闭包中，显然比把它放在全局作用域更加合适，代码如下：

```
var mult = (function(){
    var cache = {};
    return function(){
        var args = Array.prototype.join.call( arguments, ',' );
        if ( cache[ args ] ){
            return cache[ args ];
        }
        var a = 1;
        for ( var i = 0, l = arguments.length; i < l; i++ ){
            a = a * arguments[i];
        }
        return cache[ args ] = a;
    }

})();
```

```
mult( 1, 2, 3 );    // 输出：  6
```

其实，最少知识原则也叫迪米特法则（Law of Demeter，LoD），“迪米特”这个名字源自1987年美国东北大学一个名为“Demeter”的研究项目。

许多人更倾向于使用迪米特法则这个名字，也许是因为显得更酷一点。但本书参考***Head First Design Patterns*** 的建议，称之为最少知识原则。一是因为这个名字更能体现其含义，另一个原因是“法则”给人的感觉是必须强制遵守，而原则只是一种指导，没有哪条原则是在实际开发中必须遵守的。比如，虽然遵守最小知识原则减少了对象之间的依赖，但也有可能增加一些庞大到难以维护的第三者对象。跟单一职责原则一样，在实际开发中，是否选择让代码符合最少知识原则，要根据具体的环境来定。