第 13 章　职责链模式

职责链模式的定义是：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系，将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

职责链模式的名字非常形象，一系列可能会处理请求的对象被连接成一条链，请求在这些对象之间依次传递，直到遇到一个可以处理它的对象，我们把这些对象称为链中的节点，如图13-1所示。

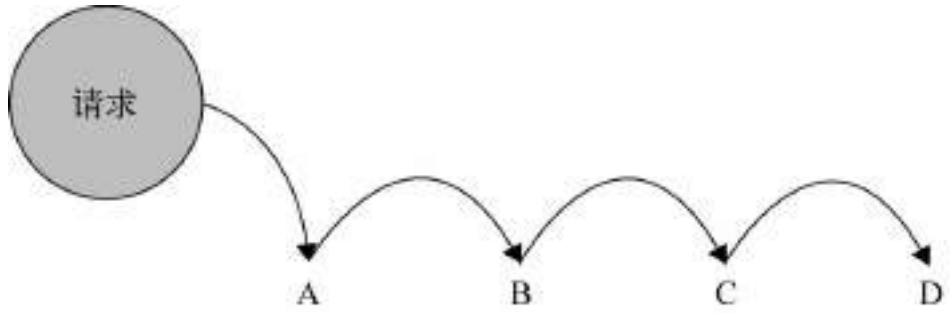

图　13-1

13.1　现实中的职责链模式

职责链模式的例子在现实中并不难找到，以下就是两个常见的跟职责链模式有关的场景。

- 如果早高峰能顺利挤上公交车的话，那么估计这一天都会过得很开心。因为公交车上人实在太多了，经常上车后却找不到售票员在哪，所以只好把两块钱硬币往前面递。除非你运气够好，站在你前面的第一个人就是售票员，否则，你的硬币通常要在N个人手上传递，才能最终到达售票员的手里。

- 中学时代的期末考试，如果你平时不太老实，考试时就会被安排在第一个位置。遇到不会答的题目，就把题目编号写在小纸条上往后传递，坐在后面的同学如果也不会答，他就会把这张小纸条继续递给他后面的人。

从这两个例子中，我们很容易找到职责链模式的最大优点：请求发送者只需要知道链中的第一个节点，从而弱化了发送者和一组接收者之间的强联系。如果不使用职责链模式，那么在公交车上，我就得先搞清楚谁是售票员，才能把硬币递给他。同样，在期末考试中，也许我就要先了解同学中有哪些可以解答这道题。

13.2 实际开发中的职责链模式

假设我们负责一个售卖手机的电商网站，经过分别交纳500元定金和200元定金的两轮预定后（订单已在此时生成），现在已经到了正式购买的阶段。

公司针对支付过定金的用户有一定的优惠政策。在正式购买后，已经支付过500元定金的用户会收到100元的商城优惠券，200元定金的用户可以收到50元的优惠券，而之前没有支付定金的用户只能进入普通购买模式，也就是没有优惠券，且在库存有限的情况下不一定保证能买到。

我们的订单页面是PHP吐出的模板，在页面加载之初，PHP会传递给页面几个字段。

- `orderType`： 表示订单类型（定金用户或者普通购买用户），`code`的值为1的时候是500元定金用户，为2的时候是200元定金用户，为3的时候是普通购买用户。
- `pay`： 表示用户是否已经支付定金，值为`true`或者`false`， 虽然用户已经下过500元定金的订单，但如果他一直没有支付定金，现在只能降级进入普通购买模式。
- `stock`： 表示当前用于普通购买的手机库存数量，已经支付过500元或者200元定金的用户不受此限制。

下面我们把这个流程写成代码:

```
var order = function( orderType, pay, stock ){
    if ( orderType === 1 ){         // 500元定金购买模式
        if ( pay === true ){     // 已支付定金
            console.log( '500元定金预购, 得到100优惠券' );
        }else{     // 未支付定金，降级到普通购买模式
            if ( stock > 0 ){     // 用于普通购买的手机还有库存
                console.log( '普通购买, 无优惠券' );
            }else{
                console.log( '手机库存不足' );
            }
        }
    }
```

```
    else if ( orderType === 2 ){      // 200元定金购买模式
        if ( pay === true ){
            console.log( '200元定金预购, 得到50优惠券' );
        }else{
            if ( stock > 0 ){
                console.log( '普通购买, 无优惠券' );
            }else{
                console.log( '手机库存不足' );
            }
        }
    }

    else if ( orderType === 3 ){
        if ( stock > 0 ){
            console.log( '普通购买, 无优惠券' );
        }else{
            console.log( '手机库存不足' );
        }
    }
};

order( 1 , true, 500);  // 输出： 500元定金预购, 得到100优惠券
```

虽然我们得到了意料中的运行结果，但这远远算不上一段值得夸奖的代码。order 函数不仅巨大到难以阅读，而且需要经常进行修改。虽然目前项目能正常运行，但接下来的维护工作无疑是个梦魇。恐怕只有最“新手”的程序员才会写出这样的代码。

13.3 用职责链模式重构代码

现在我们采用职责链模式重构这段代码，先把500元订单、200元订单以及普通购买分成3个函数。

接下来把orderType 、pay 、stock 这3个字段当作参数传递给500元订单函数，如果该函数不符合处理条件，则把这个请求传递给后面的200元订单函数，如果200元订单函数依然不能处理该请求，则继续传递请求给普通购买函数，代码如下：

```
// 500元订单

var order500 = function( orderType, pay, stock ){
    if ( orderType === 1 && pay === true ){
        console.log( '500元定金预购, 得到100优惠券' );
    }else{
        order200( orderType, pay, stock );    // 将请求传递给200元订单
    }
};

// 200元订单

var order200 = function( orderType, pay, stock ){
    if ( orderType === 2 && pay === true ){
        console.log( '200元定金预购, 得到50优惠券' );
    }else{
        orderNormal( orderType, pay, stock );    // 将请求传递给普通订单
    }
};

// 普通购买订单

var orderNormal = function( orderType, pay, stock ){
    if ( stock > 0 ){
        console.log( '普通购买, 无优惠券' );
    }else{
        console.log( '手机库存不足' );
    }
};

// 测试结果：

order500( 1 , true, 500);     // 输出：500元定金预购, 得到100优惠券
order500( 1, false, 500 );    // 输出：普通购买, 无优惠券
```

```
order500( 2, true, 500 );     // 输出：200元定金预购，得到50优惠券
order500( 3, false, 500 );    // 输出：普通购买，无优惠券
order500( 3, false, 0 );      // 输出：手机库存不足
```

可以看到，执行结果和前面那个巨大的order 函数完全一样，但是代码的结构已经清晰了很多，我们把一个大函数拆分了3个小函数，去掉了许多嵌套的条件分支语句。

目前已经有了不小的进步，但我们不会满足于此，虽然已经把大函数拆分成了互不影响的3个小函数，但可以看到，请求在链条传递中的顺序非常僵硬，传递请求的代码被耦合在了业务函数之中：

```
var order500 = function( orderType, pay, stock ){
    if ( orderType === 1 && pay === true ){
        console.log( '500元定金预购，得到100优惠券' );
    }else{
        order200( orderType, pay, stock );
        // order200和order500耦合在一起
    }
};
```

这依然是违反开放-封闭原则的，如果有天我们要增加300元预订或者去掉200元预订，意味着就必须改动这些业务函数内部。就像一根环环相扣打了死结的链条，如果要增加、拆除或者移动一个节点，就必须得先砸烂这根链条。

13.4 灵活可拆分的职责链节点

本节我们采用一种更灵活的方式，来改进上面的职责链模式，目标是让链中的各个节点可以灵活拆分和重组。

首先需要改写一下分别表示3种购买模式的节点函数，我们约定，如果某个节点不能处理请求，则返回一个特定的字符串 'nextSuccessor' 来表示该请求需要继续往后面传递：

```
var order500 = function( orderType, pay, stock ){
    if ( orderType === 1 && pay === true ){
        console.log( '500元定金预购，得到100优惠券' );
    }else{
        return 'nextSuccessor';    // 我不知道下一个节点是谁，反正把请求往后面
    }
};

var order200 = function( orderType, pay, stock ){
    if ( orderType === 2 && pay === true ){
        console.log( '200元定金预购，得到50优惠券' );
    }else{
        return 'nextSuccessor';    // 我不知道下一个节点是谁，反正把请求往后面
    }
};

var orderNormal = function( orderType, pay, stock ){
    if ( stock > 0 ){
        console.log( '普通购买，无优惠券' );
    }else{
        console.log( '手机库存不足' );
    }
};
```

接下来需要把函数包装进职责链节点，我们定义一个构造函数Chain ，在new Chain 的时候传递的参数即为需要被包装的函数， 同时它还拥有一个实例属性this.successor ，表示在链中的下一个节点。

此外Chain 的prototype 中还有两个函数，它们的作用如下所示：

```
// Chain.prototype.setNextSuccessor  指定在链中的下一个节点
// Chain.prototype.passRequest  传递请求给某个节点

var Chain = function( fn ){
    this.fn = fn;
    this.successor = null;
};

Chain.prototype.setNextSuccessor = function( successor ){
    return this.successor = successor;
};

Chain.prototype.passRequest = function(){
    var ret = this.fn.apply( this, arguments );

    if ( ret === 'nextSuccessor' ){
        return this.successor && this.successor.passRequest.apply( this.
    }

    return ret;
};
```

现在我们把3个订单函数分别包装成职责链的节点：

```
var chainOrder500 = new Chain( order500 );
var chainOrder200 = new Chain( order200 );
var chainOrderNormal = new Chain( orderNormal );
```

然后指定节点在职责链中的顺序：

```
chainOrder500.setNextSuccessor( chainOrder200 );
chainOrder200.setNextSuccessor( chainOrderNormal );
```

最后把请求传递给第一个节点：

```
chainOrder500.passRequest( 1, true, 500 );     // 输出：500元定金预购，得到1
chainOrder500.passRequest( 2, true, 500 );     // 输出：200元定金预购，得到5
chainOrder500.passRequest( 3, true, 500 );     // 输出：普通购买，无优惠券
chainOrder500.passRequest( 1, false, 0 );      // 输出：手机库存不足
```

通过改进，我们可以自由灵活地增加、移除和修改链中的节点顺序，假如某天网站运营人员又想出了支持300元定金购买，那我们就在该链中增加一个节点即可：

```
var order300 = function(){
    // 具体实现略
};

chainOrder300= new Chain( order300 );
chainOrder500.setNextSuccessor( chainOrder300);
chainOrder300.setNextSuccessor( chainOrder200);
```

对于程序员来说，我们总是喜欢去改动那些相对容易改动的地方，就像改动框架的配置文件远比改动框架的源代码简单得多。在这里完全不用理会原来的订单函数代码，我们要做的只是增加一个节点，然后重新设置链中相关节点的顺序。

13.5 异步的职责链

在上一节的职责链模式中，我们让每个节点函数同步返回一个特定的值"nextSuccessor"，来表示是否把请求传递给下一个节点。而在现实开发中，我们经常会遇到一些异步的问题，比如我们要在节点函数中发起一个ajax异步请求，异步请求返回的结果才能决定是否继续在职责链中passRequest。

这时候让节点函数同步返回"nextSuccessor"已经没有意义了，所以要给Chain类再增加一个原型方法Chain.prototype.next，表示手动传递请求给职责链中的下一个节点：

```
Chain.prototype.next= function(){
    return this.successor && this.successor.passRequest.apply( this.succ
};
```

来看一个异步职责链的例子：

```
var fn1 = new Chain(function(){
    console.log( 1 );
    return 'nextSuccessor';
});

var fn2 = new Chain(function(){
    console.log( 2 );
    var self = this;
    setTimeout(function(){
        self.next();
    }, 1000 );
});

var fn3 = new Chain(function(){
    console.log( 3 );
});

fn1.setNextSuccessor( fn2 ).setNextSuccessor( fn3 );
fn1.passRequest();
```

现在我们得到了一个特殊的链条，请求在链中的节点里传递，但节点有权利决定什么时候把请求交给下一个节点。可以想象，异步的职责链加上命令模式（把ajax请求封装成命令对象，详情请参考第9章），我们可以很方便地创建一个异步ajax队列库。

13.6　职责链模式的优缺点

前面已经说过，职责链模式的最大优点就是解耦了请求发送者和N个接收者之间的复杂关系，由于不知道链中的哪个节点可以处理你发出的请求，所以你只需把请求传递给第一个节点即可，如图13-2和图13-3所示。

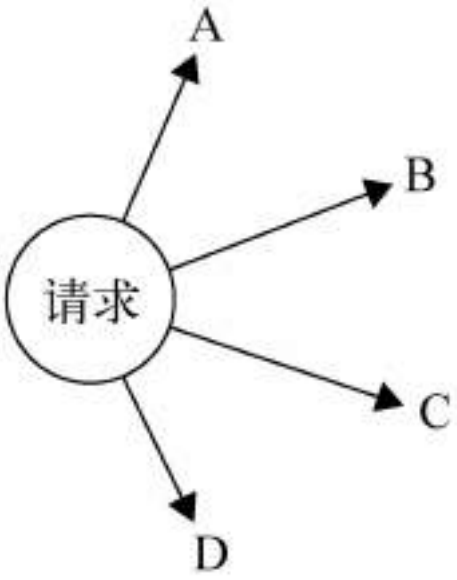

图　13-2

用职责链模式改进后：

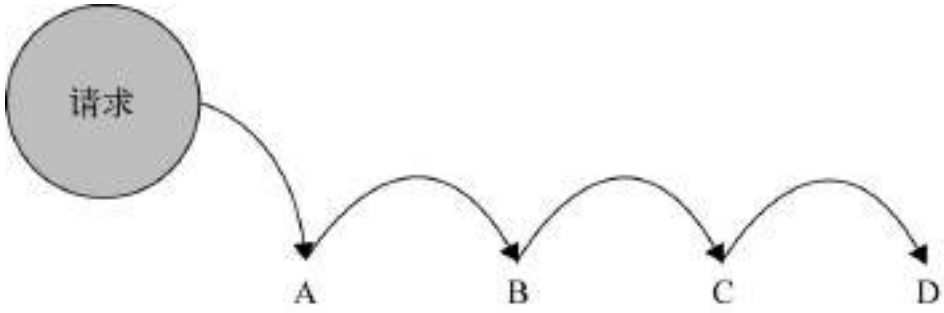

图　13-3

在手机商城的例子中，本来我们要被迫维护一个充斥着条件分支语句的巨大的函数，在例子里的购买过程中只打印了一条log 语句。其实在现实开发中，这里要做更多事情，比如根据订单种类弹出不同的浮层提示、渲染

不同的UI节点、组合不同的参数发送给不同的cgi等。 用了职责链模式之后，每种订单都有各自的处理函数而互不影响。

其次，使用了职责链模式之后，链中的节点对象可以灵活地拆分重组。增加或者删除一个节点，或者改变节点在链中的位置都是轻而易举的事情。这一点我们也已经看到，在上面的例子中，增加一种订单完全不需要改动其他订单函数中的代码。

职责链模式还有一个优点，那就是可以手动指定起始节点，请求并不是非得从链中的第一个节点开始传递。比如在公交车的例子中，如果我明确在我前面的第一个人不是售票员，那我当然可以越过他把公交卡递给他前面的人，这样可以减少请求在链中的传递次数，更快地找到合适的请求接受者。这在普通的条件分支语句下是做不到的，我们没有办法让请求越过某一个`if` 判断。

拿代码来证明这一点，假设某一天网站中支付过定金的订单已经全部结束购买流程，我们在接下来的时间里只需要处理普通购买订单，所以我们可以直接把请求交给普通购买订单节点：

```
orderNormal.passRequest( 1, false, 500 );    // 普通购买, 无优惠券
```

如果运用得当，职责链模式可以很好地帮助我们组织代码，但这种模式也并非没有弊端，首先我们不能保证某个请求一定会被链中的节点处理。比如在期末考试的例子中，小纸条上的题目也许没有任何一个同学知道如何解答，此时的请求就得不到答复，而是径直从链尾离开，或者抛出一个错误异常。在这种情况下，我们可以在链尾增加一个保底的接受者节点来处理这种即将离开链尾的请求。

另外，职责链模式使得程序中多了一些节点对象，可能在某一次的请求传递过程中，大部分节点并没有起到实质性的作用，它们的作用仅仅是让请求传递下去，从性能方面考虑，我们要避免过长的职责链带来的性能损耗。

13.7 用AOP实现职责链

在之前的职责链实现中，我们利用了一个Chain 类来把普通函数包装成职责链的节点。其实利用JavaScript的函数式特性，有一种更加方便的方法来创建职责链。

下面我们改写一下3.2.3节Function.prototype.after 函数，使得第一个函数返回'nextSuccessor' 时，将请求继续传递给下一个函数，无论是返回字符串'nextSuccessor' 或者false 都只是一个约定，当然在这里我们也可以让函数返回false 表示传递请求，选择'nextSuccessor' 字符串是因为它看起来更能表达我们的目的，代码如下：

```
Function.prototype.after = function( fn ){
    var self = this;
    return function(){
        var ret = self.apply( this, arguments );
        if ( ret === 'nextSuccessor' ){
            return fn.apply( this, arguments );
        }

        return ret;
    }
};

var order = order500yuan.after( order200yuan ).after( orderNormal );

order( 1, true, 500 );     // 输出：500元定金预购，得到100优惠券
order( 2, true, 500 );     // 输出：200元定金预购，得到50优惠券
order( 1, false, 500 );    // 输出：普通购买，无优惠券
```

用AOP来实现职责链既简单又巧妙，但这种把函数叠在一起的方式，同时也叠加了函数的作用域，如果链条太长的话，也会对性能有较大的影响。

13.8　用职责链模式获取文件上传对象

在第7章有一个用迭代器获取文件上传对象的例子：当时我们创建了一个迭代器来迭代获取合适的文件上传对象，其实用职责链模式可以更简单，我们完全不用创建这个多余的迭代器，完整代码如下：

```
var getActiveUploadObj = function(){
    try{
        return new ActiveXObject("TXFTNActiveX.FTNUpload");    // IE上传
    }catch(e){
        return 'nextSuccessor' ;
    }
};

var getFlashUploadObj = function(){
    if ( supportFlash() ){
        var str = '<object type="application/x-shockwave-flash"></object
        return $( str ).appendTo( $('body') );
    }
    return 'nextSuccessor' ;
};

var getFormUpladObj = function(){
    return $( '<form><input name="file" type="file"/></form>' ).appendTo
};

var getUploadObj = getActiveUploadObj.after( getFlashUploadObj ).after(

console.log(  getUploadObj()  );
```