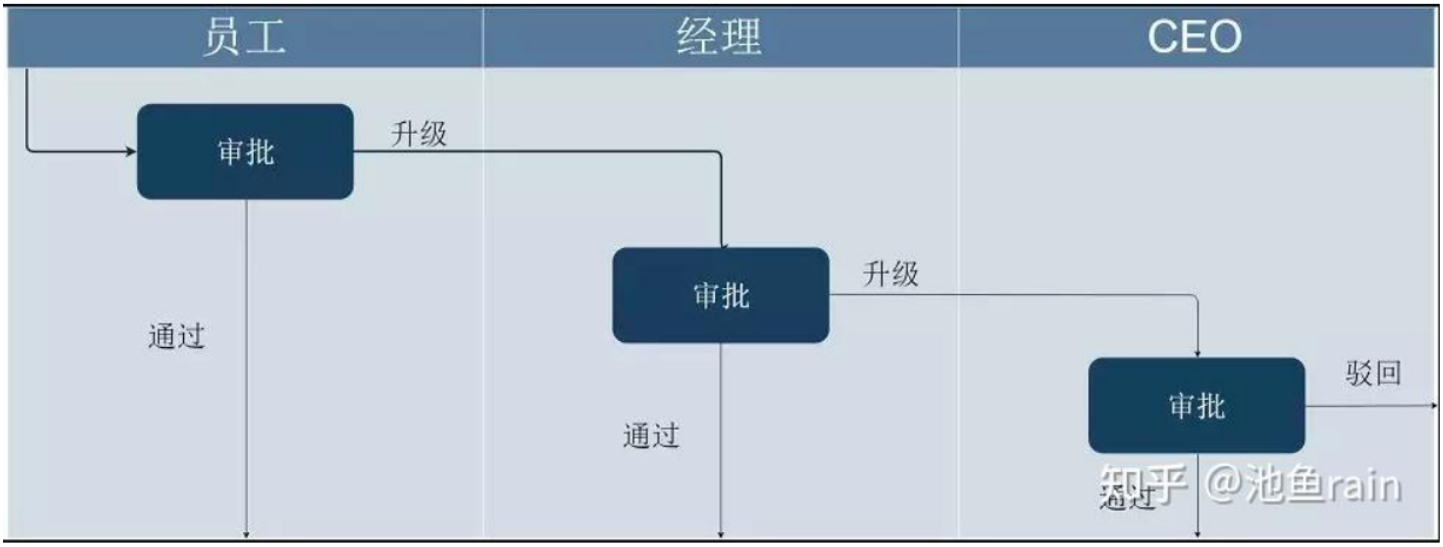
13 职责链模式

职责链（责任链）模式的定义：使很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，沿着链子传递请求直到有对象处理它



现实中的职责链模式

- 上面的审批流就是一个很好的例子
- 再例如学生作弊传纸条，遇到不会答的题目就往后传递纸条，直到有人会回答为止

优点就是请求发送者只需要知道链中的第一个节点，弱化了发送者和一组接收者之间的强联系。例如上面的审批流我们就必须知道我手上的审批单到底应该谁审批，作弊学生也必须知道到底谁会答案

实际开发中的职责链模式

假设这样一个业务：

我们负责一个购买网站，这里预购期间缴纳500定金预定之后可以在正式购买之后收到100元的商城优惠券，缴纳200元定金的用户可以收到50元的商城优惠券，而没有给定金的进入普通购买模式，没有优惠券。

php会传递下面几个字段用于渲染页面：

- orderType：code为1,2,3，分别表示500定金用户，200定金用户，普通用户
- pay：表示用户是否已经付过定金
- stock：表示普通购买的商品库存数量，支付定金的不受限制

我们尝试书写一下代码：

```
/**
 * order过程
 */
let order = (orderType, pay, stock) => {
    if(orderType === 1){
        if(pay){
            console.log('500元定金预购，得到100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');
            }
        }
    }

    // if(orderType === 2){...}
    // if(orderType === 3){...}
}
```

这样的代码很难阅读，而且会经常修改，难以维护

利用职责链来重构

我们尝试采用职责链模式重构，首先我们需要根据职责将500,200，普通分成3个函数，接着还需要一个传递的过程：

```
/**
 * 第一次重构
 */
let order500 = (orderType, pay, stock) => {
    if(orderType === 1 && pay){
        console.log('500元定金预购，得到100优惠券')
    } else {
        // 传递
        order200(orderType, pay, stock);
    }
}

let order200 = (orderType, pay, stock) => {
    if(orderType === 2 && pay){
```

```
        console.log('200元定金预购，得到50优惠券')
    } else {
        // 传递
        orderNormal(orderType, pay, stock);
    }
}

let orderNormal = (orderType, pay, stock) => {
    if(stock > 0){
        console.log('普通购买')
    } else {
        console.log('库存不足')
    }
}

order500(1,false,500);
// ...
```

可以看到这里我们将巨大的order函数根据职责拆分了3个小函数，去掉了冗余的嵌套if-else语句

但是目前的传递其实非常僵硬，并且传递的逻辑耦合到了业务逻辑里面，依然是违反开放-封闭原则的，如果有哪天去掉200元的逻辑的话，那么500元内部的业务代码就必须修改了

灵活可拆分的职责链节点

这里我们尝试使用一种更加灵活的方式处理，这里我们约定，某一个节点如果不能处理就返回“nextSuccessor”来表示：

```
/**
 * 灵活可拆分
 */
var order500 = (orderType, pay, stock) => {
    if(orderType === 1 && pay){
        console.log('500元定金预购，得到100优惠券')
    } else {
        return 'nextSuccessor'
    }
}

var order200 = (orderType, pay, stock) => {
    if(orderType === 2 && pay){
        console.log('200元定金预购，得到50优惠券')
    } else {
        return 'nextSuccessor'
    }
```

```
}

var orderNormal = (orderType, pay, stock) => {
    if(stock > 0){
        console.log('普通购买')
    } else {
        console.log('库存不足')
    }
}
```

接着我们一个Chain来包装上述函数为职责链的节点，同时他还应该可以描述职责链的传递顺序与传递动作：

```
class Chain {
    constructor(fn){
        this.fn = fn;
        this.successor = null;
    }

    // 指定链中下一个节点
    setNextSuccessor = function(successor){
        return this.successor = successor;
    }

    // 传递请求
    pastRequest = function(){
        const ret = this.fn.apply(this, arguments);
        if(ret = 'nextSuccessor'){
            return this.successor && this.successor.pastRequest.apply(this.succe
        }
        return ret;
    }
}

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

// 指定顺序
chainOrder500.setNextSuccessor(chainOrder200);
chainOrder200.setNextSuccessor(chainOrderNormal);

// 只需要关注第一个节点
chainOrder500.pastRequest(1,true, 500);
// ...
```

通过这个改进，我们可以灵活的增加和移除链子中节点以及顺序

对于程序员来说，我们总是喜欢去改动那些容易改动的地方，就像改动框架的配置文件远比改动框架源代码简单得多。在这里完全不用理会原来的订单函数代码，我们要做的只是调整节点，设置Chain顺序就行了

异步的职责链

现实开发中我们经常会遇到异步的问题，例如在一个节点函数中发出ajax请求，根据请求结果再判断是否继续passRequest。

此时之前的同步返回‘nextSuccessor’其实已经没有意义了，我们可以新增一个next方法来表示手动传递请求给职责链中的下一个节点：

```
/**
 * 异步职责链
 */
class Chain {
    constructor(fn){
        this.fn = fn;
        this.successor = null;
    }

    // ...

    // 新增一个方法
    next = function(){
        return this.successor && this.successor.pastRequest.apply(this.successor
    }
}

let fn1 = new Chain(() => {
    console.log(1);
    return 'nextSuccessor';
})

let asyncFn2 = new Chain(() => {
    console.log(2);
    let self = this;
    setTimeout(() => {
        self.next();
    }, 1000);
```

```
29 })
30
31 let fn3 = new Chain(() => {
32     console.log(3);
33 })
34
35 fn1.setNextSuccessor(asyncFn2).setNextSuccessor(fn3);
36 fn1.pastRequest();
```

优缺点

1. 优点

解耦了请求发送者和N个接收者的复杂关系，你只需要把请求传递给第一个请求而不需要知道谁能处理

不需要被迫维护一堆if-else巨大的函数

而且链条中的节点对象也可以灵活地拆分重组，增加和删除节点都很容易且不需要动节点内的逻辑

2. 缺点

我们不能保证请求一定能被链处理，最好在链尾加一个保底的接收者来处理兜底逻辑

多了一些节点对象且部分场景无用，性能方面来说避免请求链过长

利用AOP实现职责链

在JS中我们其实不需要把节点通过Chain类来包装，使用JS函数特性也可以更加方便：

```
1  /**
2   * 利用AOP来实现
3   */
4  Function.prototype.after = function(fn){
5      let self = this;
6      return function(){
7          var ret = self.apply(this, arguments);
8          if(ret === 'nextSuccessor'){
9              return fn.apply(this,arguments);
10         }
11         return ret;
12     }
13 }
14
```

```
const orderChain = order500.after(order200).after(orderNormal);
orderChain(1, true, 500);
```

用职责链重构之前的获取文件上传对象例子

之前第7章我们通过迭代器来迭代获取合适的文件上传对象，其实用职责链模式更简单，不需要创建多余的迭代器：

```

/**
 * 职责链重构获取文件上传对象
 */
const getActiveUploadObj = () => {
    try{
        return new ActiveXObject("TXFTNActiveX.FTNUpload");
    }catch(e){
        return 'nextSuccessor';
    }
}

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

const getFormUploadObj = () => {
    return $('<form><input ....>...').appendTo($('body'))
}

const getUploadObj = getActiveUploadObj.after(getFlashUploadObj).after(getFormUp

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

小结

在JS开发中，职责链比较容易被忽视。实际上只要运用得当，可以帮助我们管理代码，降低请求和处理者的耦合。

无论是作用域链，原型链，还是DOM节点的事件冒泡我们都可以找到职责链模式的影子。它还可以和组合模式一起用来连接部件和父部件，或是提高组合对象的效率