第十三课 职责链模式

学习目录

- **使用场景**
- **使用方式**
- **优缺点**

一．使用场景

在生活中，我们的日常工作可能或多或少都会用到 oa 之类的办公系统，我们在里面提交一个工作计划的请求，这个请求可能会在 oa 系统里逐级的进行审批处理，中间大多数参与请求处理其他部门同事都可以来处理这个工作计划的请求，有的只有查看和传递给下一级的权力，直到这个工作计划请求到老板那里，老板会对我们的工作计划进行审核得到审核结果。在比如，大家在外出购物在商场买衣服，一件羽绒袄市场销售价 1000 元，实际你在准备购买的时候呢，店员为了成交会给你优惠活动，店员给你 800 元，但是你觉得还不划算，想继续便宜，所以你又找到店长要她继续给优惠，店长接收到你的请求，利用她的权力给了你员工价 600 元，因此你愉快的用 600 元买到了一件羽绒服。

由此我们可以看出职责链模式其实是把一个请求按照一个链条的形式一级一级的传递下去，在这个链条上的每一个节点都可以处理这个请求，如果这个请求不属于某个节点处理范围的时候，这个节点就会把这个请求继续向下一个链条节点传递，直到找到可以处理这个请求的节点。说的更简单一点，职责链模式是把请求与处理函数进行解耦，请求不知道是否会有节点能够处理它，它的任务就是按照链条向下传递，链条节点的任务是按照自己的处理函数来处理请求或者传递请求。

职责链模式的核心思想是把请求的发送对象和请求的接收对象进行解耦，请求会按照链条一样的形式传递给每个接收对象进行处理或者传递，直到找到能够处理请求的接收对象完成请求处理。

二．使用方式

我们以数字大小的函数来看下职责链模式的基本结构。

普通数字大小

```
function fn1(num){
    if(num <= 100){
        console.log('数字小于等于 100');
    }else if(num > 100 && num <= 300){
        console.log('数字大于 100 小于等于 300');
    }else if(num > 300 && num <= 600){
        console.log('数字大于 300 小于等于 600');
    }else if(typeof num != 'number'){
        console.log('数字错误');
    }
}
fn1(100);
```

在这个例子中，传入一个数字参数，这个数字参数需要根据不同的条件分支判断来打印对应的内容，条件语句会经过每一层直到找到符合要求的条件并打印对应内容，这个结构的代码可能随着业务逻辑的增加会越来越复杂。

因此我们可以考虑把条件分支独立成几个单独的函数来编写，这样就可以避免大量的条件语句带来的代码冗余。

```
function f100(num){
    if(num <= 100){
        console.log('数字小于等于 100');
    }else{
        console.log('f100 传递数字');
        return 'next';
    }
}

function f300(num){
    if(num > 100 && num <= 300){
        console.log('数字大于 100 小于等于 300');
    }else{
        console.log('f300 传递数字');
        return 'next';
    }
}

function f600(num){
    if(num > 300 && num <= 600){
```

```
            console.log('数字大于 300 小于等于 600');
        }else{
            console.log('f600 传递数字');
            return 'next';
        }
    }

    function fError(num){
        if(typeof num != 'number'){
            console.log('数字错误');
        }else{
            console.log('数字大于 600');
        }
    }
```

在我们之前说过的装饰器模式中，我们可以利用 aop 动态的给函数增加额外功能，这里刚好可以利用这个特点，来实现职责链模式所需要的请求与处理节点函数的解耦。

装饰器职责链模式

```
Function.prototype.afterFn = function(fn){
    var _fn = this;
    return function(){
        var result = _fn.apply(this,arguments);
```

```
            if(result == 'next'){
                return fn.apply(this,arguments);
            }
            return result;
        }
    }
    var fn = f100.afterFn(f300).afterFn(f600).afterFn(fError);
    fn('fdgdfgd');
```

以上是我们利用装饰器模式来达成的职责链模式，通过返回函数可以实现链式调用职责链处理请求的效果。还有一种通过构建构造函数来实现的职责链模式。

构造函数职责链模式

```
    function LineFn(fn){
        this.fn = fn;
        this.next = null;
    }
    LineFn.prototype.bindFn = function(next){
        return this.next = next;
    }
    LineFn.prototype.setFn = function(){
        var result = this.fn.apply(this,arguments);
        if(result == 'next'){
```

```
            this.nextFn();
        }
        return result;
    }
    LineFn.prototype.nextFn = function(){
        return this.next && this.next.setFn.apply(this.next,arguments);
    }
    var f1 = new LineFn(f100);
    var f2 = new LineFn(f300);
    var f3 = new LineFn(f600);
    var fe = new LineFn(fError);
    f1.bindFn(f2);
    f2.bindFn(f3);
    f3.bindFn(fe);

    f1.setFn('dsfsdgfdgfd');
```

由此可见，不管是装饰器模式的职责链模式还是构造函数的职责链模式，本质上都实现了请求与处理函数的解耦，最重要的是调用者可以决定职责链从哪里开始进行处理和传递，因此这样做可以更加灵活的设置请求开始的位置，在业务逻辑复杂的代码中，可以有效的增强代码性能，同时我们可以随意根据需要条件新的链条，不必担心之前的链条是否会收到影响，只要保证新条件的链条函数中符合整体链条的规则条件即可。

同时职责链模式中的每个链条节点还可以处理异步的请求，等到异步请求完成之后在根据异步请求的结果来处理请求或者传递请求。

三．优缺点

1.职责链模式它把请求与处理函数进行解耦，这一点符合开放封闭原则，请求在链条中进行处理和传递，每个链条函数都可以处理请求或者传递请求。

2.职责链模式可以由调用者决定从哪个链条开始处理请求，并不一定要从第一个链条开始处理，职责链模式还可以灵活的增加或者删除链条节点，同时也可以通过手动调用方法处理异步请求，在实际业务中特别好用。

3.职责链模式适合处理较为复杂的条件语句等场景，如果在简单的业务逻辑中随便使用可能会增加代码的复杂性，因此要根据业务逻辑需求来使用。

4.职责链模式中因为是按链条传递请求，有可能大多数链条函数只是负责传递请求，并没有其他操作，但是确实又需要调用这些链条函数，因此如果链条函数过多也会带来一定的性能损耗问题，因此我们得设置好链条函数的数量问题。

谢谢观看！

我是星星课堂老师：周小周