

上一节我们讲了扩展性的方法层面，这一节主要讲的是扩展属性是模块层面，一般程序我们都可以划分为模块+组织模块沟通，那如何做模块呢？当我们初期没有经验，没有模块的经历，我们如何设计呢？当需求出来=》完成需求需要哪些步骤=》完成这些步骤需要什么模块=》完成模块=》组织模块沟通。所谓设计架构=》把需求分成低耦合模块，并组织沟通。

提高整体项目扩展性的核心(模块层面)

1. 低耦合
2. 良好的组织沟通方式

提高可扩展性的设计模式

1. 观察者模式

观察者模式有很多别名，例如监听者模式等等，它的主要目的：减少对象(模块)间的耦合，来提高扩展性。比如有两个模块，一个a模块，一个b模块 a=》b，b模块直接调用a模块，中间的耦合度就非常高，面对面调用耦合就非常高，不容易分开，解耦。现在a=》观察者-》b中间添加一个观察者，这样耦合度就降低了。观察者的应用场景：当两个模块直接沟通会增加他们的耦合性时，所以不方便直接沟通。这里说的不方便沟通？类似a模块异步请求什么时候有请求结果，如何跟b模块同步进行沟通呢？这时候就不方便直接沟通，这时候就可以通过观察者模式，通过在b模块在观察者注册一个观察，a模块的异步完成之后，通知给观察，观察者再发送消息给b模块。这里和发布订阅有点类似。总结一下，观察者模式主要是给两个完全没有想过要沟通，突然想要沟通，这里通过就可以使用观察者模式。这样改动成本的就大大降低了

2. 职责链模式

刚好和观察者模式有点相对，它更多的会用于组织一系列的同步模块。它的主要目的：为了避免请求发送者与多个请求处理耦合在一起，形成一个链条。比如我们要开一个水果加工厂=》清洗=》切水果=》加工水果，这样把我的组织模块设计成一个链条，在这个链条下面依次的执行他们的任务。应用场景：把操作分割成一系列的模块，每个模块处理好自己的事情。中间如果需要扩展一些新的环节，直接添加上就好了，无需影响到其他的模块。

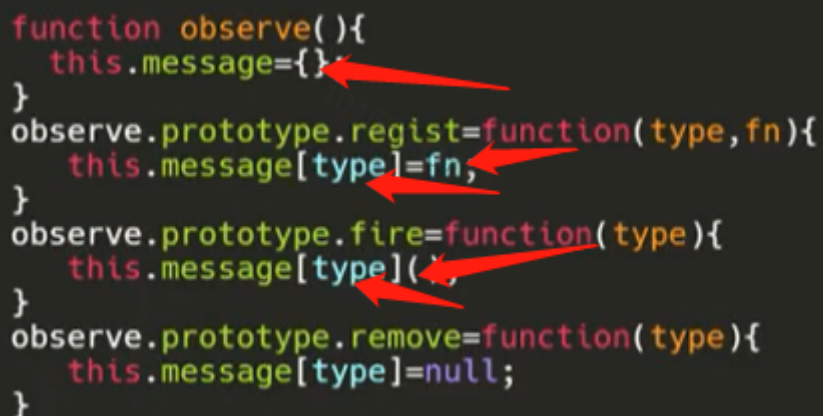
3. 访问者模式(很少用到)

我们之前都是 操作 =》数据 现在是中间多加了一层访问者， 数据=》访问者《=操作 数据和操作都提交给访问者，由访问者去处理。主要是目的:解耦数据结构与数据的操

作。应用场景：数据结构不希望与操作有关联。

基本结构

1. 观察者基本结构

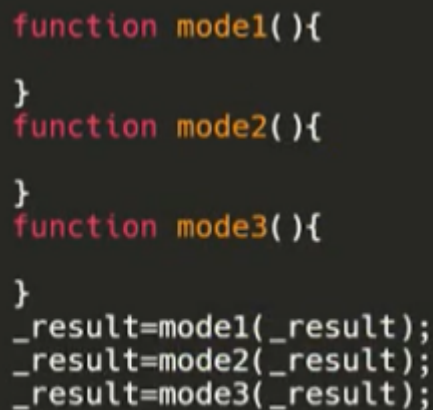


```
function observe(){
  this.message={}
}
observe.prototype.regist=function(type,fn){
  this.message[type]=fn,
}
observe.prototype.fire=function(type){
  this.message[type](*)
}
observe.prototype.remove=function(type){
  this.message[type]=null;
}
```

The image shows a JavaScript code snippet for an Observer pattern. Red arrows point to the following elements: the `this.message` property in the constructor, the `regist` method, the `fire` method, and the `this.message[type]` property access within the `fire` method.

定义一个中转观察者，两个模块直接不直接沟通，而是通过观察者，一般适用于不方便直接沟通，或者异步操作。观察者是全局的，是独立的，而不是为哪个模块单独定义的。它有两个最基本的 一个regist 另外一个fire。

2. 职责链模式



```
function model1(){
}
function mode2(){
}
function mode3(){
}
_result=model1(_result);
_result=mode2(_result);
_result=mode3(_result);
```

The image shows a JavaScript code snippet for the Chain of Responsibility pattern. It defines three functions: `model1`, `mode2`, and `mode3`. Below these, a sequence of calls is shown: `_result=model1(_result);`, `_result=mode2(_result);`, and `_result=mode3(_result);`.

把要做的事情组织为一条有序的链条，通过这条链条来传递消息来完成功能，适用于不涉及到复杂异步操作。

3. 访问者模式

```
var data=[];  
var handler=function(){  
}  
handler.prototype.get=function(){  
}  
var vistor=function(handler,data){  
  handler.get(data);  
}
```

通过定义一个访问者，代替直接访问对象，来减少两个对象之间的耦合。其中data是数据，handler是操作，vistor则是访问者，数据和操作的给访问者，由访问者自己处理。

应用示例

观察者模式示例

需求:现在假设A工程写了首页模块，然后B工程写了写了一个评论模块，现在要把评论展示在首页。

```

function observer() {
  this.message = {

  }
};
observer.prototype.regist = function (type, fn) {
  this.message[type] = fn;
}
observer.prototype.fire = function (type) {
  this.message[type].apply(this, arguments);
}

//a
function index() {
  observer.fire('indexComment');
}
//b
function comment() {
  observer.regist("indexComment", () => {
    //给首页评论
  })
}

```

通过观察者模式，组织两个毫无相关的模块。在评论模块注册，在首页模块，触发。这样就以小的成本组织了两个毫无相关的模块

职责链模式

需求:axios拦截器的设置，大家可以看成一个用职责链的思想去处理请求。axios源码

```

1 // 职责链模式-axios源码
2 function Axios (instanceConfig) {
3   this.default=instanceConfig;
4   this.interceptors={
5     request:new interceptorsManner(),
6     response:new interceptorsManner(),
7   }
8 }
9 Axios.prototype.request=function(){
10   var chain=[dispatchRequest,undefined];

```

```

11  var promise=Promise.resolve(config);
12  this.interceptors.request.handlers.forEach(function(interceptor){
13  chain.unshift(interceptor.fulfilled.interceptor.injected)
14  })
15  this.interceptor.response.handlers.forEach(function(interceptor){
16  chain.push(interceptor.fulfilled,intersecor.rejected);
17  })
18  while(chain.length){ //这里就是链条依次执行
19  promise=promise.then(chain.shift,chain.shift());
20  }
21  return promise;
22  }
23
24  function interceptorsManner(){
25  this.handlers=[]//存放use加入的方法
26  }
27  interceptorsManner.prototype.use=function use(fulfilled,rejected){
28  this.handlers.push({
29  fulfilled:fulfilled,
30  rejected:rejected
31  })
32  }

```

需求二：有一个表单们,需要先给前台校验,然后再给后台校验

```

1  //职责链模式-表单颜值
2  //表单事件绑定-> >表单前端验证-》表单后端颜值
3
4  input.onblur=function(){
5  var _value=input.value;
6  var _arr=[font,back];
7  async function test(){
8  var _result=_value;
9  while(_arr.length>0){
10  _result=await _arr.shift()(_result);
11  }
12  return _result
13  }
14  test().then((res)=>{
15  console.log(res);
16  })
17  }

```

```
18
19  function font(_result){
20  return _result+=1;
21  }
22  function back(_result){
23  return new Promise((resolve,reject)=>{
24  resolve(_result+=3);
25  })
26  }
```

访问者模式(这个很少用)

```
1 //访问者模式-不同角色访问财务
2  function report(){
3  this.income="";
4  this.cost="";
5  this.profit="";
6  }
7
8  function boss(){
9
10 }
11 boss.prototype.get=function(num){
12
13 }
14 function account(){
15
16 }
17 account.prototype.get=function(num1,num2){
18
19 }
20 function visitor(data,man){
21 var handle={
22 boss:function(data){
23 man.get(data.profit);
24 },
25 account:function(data){
26 account.get(data.income,data.cost);
27 }
28 }
```

```
29 handle[man.constructor.name](data);
30 }
31 vistor(new report(),new boss());
32
33
34 function table(){
35
36 }
37 table.prototype.show=function(){
38
39 }
40 table.prototype.delete=function(id){
41 vistor(this,tableData,'delete',id)
42 }
43 table.prototype.add=function(){
44
45 }
46
47
48
49
50 //访问者模式-表格
51 var tableData=[
52 {
53 id:1,
54 name:'xxx',
55 price:'xxx'
56 }
57 ]
58
59 function vistor(table,data,handle){
60 var handleOb={
61 delete:function(id){
62
63 },
64 add:function(){
65
66 }
67 };
68 var arg=Array.prototype.splice(arguments);
69 arg.splice(0,3);
```

```
70  handleOb[handle].apply(this, arg);  
71  }  
72
```