复习：

基础观察者模式：

// 基础观察者

var Observer = (function() {

// 定义存储对象

var ob = {

}

// 定义接口

return {

// on方法用于向观察者对象中添加事件

on: function(type, fn) {

ob[type] = fn;

},

// trigger方法用于触发观察者对象中的事件

trigger: function(type) {

ob[type]();

}

}

})()

// DOM2级观察者

var Observer = (function() {

// 定义存储对象

var ob = {

}

// 定义接口

return {

// on方法用于向观察者对象中添加事件

on: function(type, fn) {

// 判断是否有数据存在

if (ob[type]) {

ob[type].push(fn);

} else {

ob[type] = [fn];

}

},

// trigger方法用于触发观察者对象中的事件

trigger: function(type) {

for (var i = 0; i < ob[type].length; i++) {

ob[type][i]();

}

}

}

})()

// DOM2级观察者

var Observer = (function() {

// 定义存储对象

var ob = {

}

// 定义接口

return {

// on方法用于向观察者对象中添加事件

on: function(type, fn) {

// 判断是否有数据存在

if (ob[type]) {

ob[type].push(fn);

} else {

ob[type] = [fn];

}

},

// trigger方法用于触发观察者对象中的事件

trigger: function(type) {

for (var i = 0; i < ob[type].length; i++) {

ob[type][i]();

}

},

// off方法用于移除某个事件

off: function(type, fn) {

// 判断是要删除一组还是要删除某一个事件函数

if (fn) {

// 说明是要删除某个函数

for (var i = 0; i < ob[type].length; i++) {

if (ob[type][i] == fn) {

ob[type].splice(i, 1);

}

}

} else if (type) {

ob[type] = [];

} else {

// 情况观察者对象中的所有数据

ob = {};

}

}

}

})()

// DOM2级观察者

var Observer = (function() {

// 定义存储对象

var ob = {

}

// 定义接口

return {

// on方法用于向观察者对象中添加事件

on: function(type, fn) {

// 判断是否有数据存在

if (ob[type]) {

ob[type].push(fn);

} else {

ob[type] = [fn];

}

},

// trigger方法用于触发观察者对象中的事件

trigger: function(type) {

for (var i = 0; i < ob[type].length; i++) {

ob[type][i]();

}

},

// off方法用于移除某个事件

off: function(type, fn) {

// 判断是要删除一组还是要删除某一个事件函数

if (fn) {

// 说明是要删除某个函数

for (var i = 0; i < ob[type].length; i++) {

if (ob[type][i] == fn) {

ob[type].splice(i, 1);

}

}

} else if (type) {

ob[type] = [];

} else {

// 情况观察者对象中的所有数据

ob = {};

}

},

// once方法 通过该方法只能触发一次事件

once: function(type, fn) {

// 定义的新的函数

function aaa() {

// 执行fn

fn();

// 执行完之后移除该函数

Observer.off(type, aaa);

}

Observer.on(type, aaa);

}

}

})()

委托模式： 本来将子元素要做的事件交给父元素

能够解决三件事件：

1 减少事件数量

2 预言未来元素

3 防止内存泄漏

# 节流模式

在某些高频率事件中， 代码执行一次和执行多次的结果是一致的，所以没有必要执行多次，此时要进行节流，我们要将节流的共同点提取出来，最终要完成一个节流函数

例如：在onscroll事件中，如果直接把$(dom).animate({opacity: 1}, 1000), 代码将会被执行很多次

|  |
| --- |
| 1. /\*\* 2. \* Throttle方法可以实现两种功能 3. \* 1: 开启节流器 Throttle(fn, {}) 4. \* @fn: 要节流的函数 5. \* @options 配置对象 6. \* @context 执行时候的上下文(this) 7. \* @args 原函数所需要的参数 8. \* @time 决定延迟的时间 9. \* 10. \* 2: 关闭节流器 Throttle(false, fn) 11. \* @fasle 是一个标识， 当传递false的时候 说明是要关闭节流器了 12. \* @fn 要节流的函数 13. \* 14. \* 15. \*\*\*\*\*\*\*\*\*/ 16. function Throttle(fn, options) { 17. // 判断fn传递的是函数还是布尔值 18. if (typeof fn === "function") { 19. // 说明要开启节流器了 20. // console.log("开启"); 21. // 配置信息 22. options = options || { 23. context: document, 24. args: [], 25. time: 1000 26. } 27. // 清除之前的延时器 28. clearTimeout(fn.\_\_throttle); 29. // 由于每一个函数要对应一个节流器，所以给fn添加一个自定义属性 30. fn.\_\_throttle = setTimeout(function() { 31. fn.apply(options.context, options.args); 32. }, options.time); 33. } else if (typeof fn === "boolean") { 34. // 说明要关闭节流器 35. // console.log("关闭"); 36. // 此时fn 表示false, options表示要节流的函数 37. clearTimeout(options.\_\_throttle); 38. } 39. } |

# 迭代器模式

迭代器可以简单理解为循环

迭代器模式定义：迭代器模式是向外暴露一个接口，通过该接口可以按照顺序的访问里面的内容， 而不去暴露内部的结构

## 2.1 jquery中的迭代器

$.each(target, fn)

target: 要遍历的目标，可以是对象，可以是数组

fn: 要执行的函数

有两个参数

第一个参数：索引值

第二个参数：成员值

作用域是遍地到的该对象

举例：

|  |
| --- |
| 1. // 定义数组 2. var arr = [1, 2, 3, 4, 5, 6]; 3. // 使用$.each 4. $.each(arr, function(index, value) { 5. // console.log(this, arguments); 6. }) |

结果：

|  |
| --- |
|  |

再比如：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. num: 100, 4. color: "red" 5. } 6. // 使用$.each遍历对象 7. $.each(obj, function() { 8. console.log(this, arguments); 9. }) |

结果：

|  |
| --- |
|  |

## 2.2 underscore中的迭代器

underscore是js的一个实用库，当引入该文件之后向全局暴露的是\_

\_.each(target, fn)

target: 要遍历的目标

fn: 要执行的函数

有三个参数

第一个参数： 成员值

第二个参数： 索引值

第三个参数： 原数组

作用域window

举例：

|  |
| --- |
| 1. // 定义数组 2. var arr = ["华为", "小米", "苹果", "三星"]; 3. // 使用\_.each方法 4. \_.each(arr, function() { 5. console.log(this, arguments); 6. }) |

结果：

|  |
| --- |
|  |

## 2.3 ES5中的迭代器

使用方式：

[].forEach(fn)

fn: 要执行的函数

函数中有三个参数

第一个参数：成员值

第二个参数：索引值

第三个参数：原数组

作用域window

举例：

|  |
| --- |
| 1. // 定义数组 2. var arr = ["华为", "小米", "苹果", "三星"]; 3. // 使用ES5中的迭代器方法 4. arr.forEach(function() { 5. console.log(this, arguments); 6. }) |

结果：

|  |
| --- |
|  |

## 2.4 自己的迭代器

|  |
| --- |
| 1. /\*\* 2. \* each方法 可以实现对数组和对象的遍历 3. \* @target 要遍历的目标 4. \* @fn 要执行的函数 5. \*\*/ 6. function each(target, fn) { 7. // 判断target传递的是数组还是对象 8. if (Array.isArray(target)) { 9. // 说明传递的是数组 10. // 遍历数组 11. for (var i = 0; i < target.length; i++) { 12. // 执行函数并传递三个参数 13. // 三个参数： 成员值: target[i]、 索引值： i、 原数组: target 14. // fn(target[i], i, target); 15. // 改变执行时候的作用域 16. fn.call(target[i], target[i], i, target); 17. } 18. } else if (Object.prototype.toString.call(target) === "[object Object]") { 19. // 请问有几种方式确定是一个对象？ 20. // 1: 判断实例化对象 instanceof Object 这种方式虽然可行， 但是不保险 21. // 2: 判断构造函数 constructor Object 22. // 3: 判断对象的类型 Object.prototype.toString === "[object Object]" 23. for (var i in target) { 24. // 执行fn传递参数 25. // 两个参数: 属性名: i、 属性值: target[i] 26. // fn(i, target[i]); 27. // 改变执行时候的作用域 28. fn.call(i, i, target[i]); 29. } 30. } 31. } |

# 适配器模式

适配器模式的定义： 将一个程序的接口转接到另一个程序的接口，称为适配器

比如：苹果手机的充电器是两项插头，但是家中没有两项插座，于是去买来一个两项插座（适配器），结果成功的将手机充上电。

|  |
| --- |
| 1. // 将ickt.js中的功能全部修改为jquery实现 2. var ickt = { 3. // 获取元素 4. $: function(id) { 5. return $("#" + id)[0]; 6. }, 7. // html方法 8. html: function(dom, value) { 9. $(dom).html(value); 10. }, 12. // css方法 13. css: function(dom, key, value) { 14. $(dom).css(key, value); 15. }, 16. // 点击事件 17. click: function(dom, fn) { 18. $(dom).click(fn); 19. } 20. } |

# 组合模式

定义: 组合模式又称为“部分”-“整体”模式，它是把一个对象看成是“部分”-“整体”的表示结构，部分和整体具备一致性

|  |
| --- |
| 1. // 定义基类 2. function Base() { 3. this.dom = null; 4. this.children = []; 5. } 6. // 方法要写在原型中 7. // add方法 8. Base.prototype.add = function(value) { 9. this.children.push(value); 10. return this; 11. } 12. // render 13. Base.prototype.render = function() { 14. // 备份this 15. var me = this; 16. this.children.forEach(function(value) { 17. me.dom.appendChild(value.dom); 18. }) 19. return this; 20. } 21. // 定义最外成的容器 22. function Container() { 23. // 构造函数式继承 24. Base.call(this); 25. this.dom = document.createElement("ul"); 26. } 27. // 继承基类原型中的方法 28. Container.prototype = Object.create(Base.prototype); 29. // 定义li 30. function Item() { 31. // 构造函数式继承 32. Container.call(this); 33. this.dom = document.createElement("li"); 34. } 35. Item.prototype = Object.create(Container.prototype); 36. // 定义最底层的元素 37. function Leaf(str) { 38. this.dom = document.createTextNode(str); 39. } 40. // 最外层的元素 41. var container = new Container(); 42. // 定义五个li元素 43. var item1 = new Item(); 44. var item2 = new Item(); 45. var item3 = new Item(); 46. var item4 = new Item(); 47. var item5 = new Item(); 48. // 最顶层元素 49. var leaf1 = new Leaf("习近平走进青年一代"); 50. var leaf2 = new Leaf("人民日报头版头条：中国经济发展韧性十足"); 51. var leaf3 = new Leaf("筑牢平安中国的铜墙铁壁 理上网来"); 52. var leaf4 = new Leaf("第二届数字中国建设峰会在福州市开幕 专题"); 53. var leaf5 = new Leaf("媒体：面对美方波动 最好的回答是淡定"); 54. item1.add(leaf1).render(); 55. item2.add(leaf2).render(); 56. item3.add(leaf3).render(); 57. item4.add(leaf4).render(); 58. item5.add(leaf5).render(); 59. container.add(item1).add(item2).add(item3).add(item4).add(item5).render(); 60. // 上树 61. document.body.appendChild(container.dom); |

结果：

|  |
| --- |
|  |

# 命令模式

命令模式的定义: 将命令的请求与实现解耦并封装

它与策略模式很像

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |