爱创课堂前端培训

# 设计模式第三天

第3天课堂笔记（本课程共3天）

班级：北京前端训练营19期

讲师：彭帅伟

日期：2019年1月9日

彭帅伟老师

QQ: 284337908

爱创课堂官网 ：[www.icketang.com](http://www.icketang.com)

观察者模式：

var Observer = (function() {

var ob = {};

return {

on: function(type, fn) {

ob[type] = fn;

},

trigger: function(type, val) {

ob[type](val);

}

}

})()

DOM2级观察者模式：

var Observer = (function() {

var ob = {};

return {

on: function(type, fn) {

// 判断是否存在内容， 直接Push

if (ob[type]) {

ob[type].push(fn);

} else {

//不是数组， 变成数组

ob[type] = [fn];

}

},

trigger: function(type, val) {

// 循环执行

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

ob[type][i](val);

}

}

}

})()

off方法：

var Observer = (function() {

var ob = {};

return {

on: function(type, fn) {

// 判断是否存在内容， 直接Push

if (ob[type]) {

ob[type].push(fn);

} else {

//不是数组， 变成数组

ob[type] = [fn];

}

},

trigger: function(type, val) {

// 循环执行

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

ob[type][i](val);

}

}

}，

// off方法用于取消事件

of f: function(type, fn) {

// 判断是要删除一组还是要删除某个函数

if (fn) {

// 说明要删除某个函数

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

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

// 移除当前项

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

}

}

} else if (type) {

// 说明要删除一组

ob[type] = [];

} else {

ob = {};

}

}

})()

once方法：通过once方法绑定的事件，只会执行一次

var Observer = (function() {

var ob = {};

return {

on: function(type, fn) {

// 判断是否存在内容， 直接Push

if (ob[type]) {

ob[type].push(fn);

} else {

//不是数组， 变成数组

ob[type] = [fn];

}

},

trigger: function(type, val) {

// 循环执行

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

ob[type][i](val);

}

}

}，

// off方法用于取消事件

of f: function(type, fn) {

// 判断是要删除一组还是要删除某个函数

if (fn) {

// 说明要删除某个函数

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

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

// 移除当前项

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

}

}

} else if (type) {

// 说明要删除一组

ob[type] = [];

} else {

ob = {};

}

}，

once: function(type, fn) {

function aaa() {

//执行fn

fn();

// 执行完毕移除该函数

Observer.off(type, aaa);

}

Observer.on(type, aaa);

}

})()

策略模式：

将一组一组的算法，封装起来使之可以相互替换

委托模式：

本来子元素要做的事件，交给父元素去做

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， 这个时候就需要进行节流，我们把节流的共同点提取出来，最终要完成一个节流函数

比如：在onscroll事件，$(dom).animate({opacity: 1}, 1000, funtion() {})

|  |
| --- |
| 1. /\*\* 2. \* Throttle 可以实现节流函数 3. \* 1 开启节流器 Throttle(fn, {}); 4. \* @fn 要被节流的函数 5. \* @options 是一个配置对象 6. \* context: 函数执行的时候的上下文(this) 7. \* args: [], 是一个数组， 数组中的每一项都是原函数所需要的参数 8. \* time: 决定了要延迟的时间 9. \* 10. \*2 关闭节流器 Throttle(false, fn); 11. \*@false 是一个标识， 当传递false的时候，是确定要关闭节流器了 12. \*@fn 要节流的函数 13. \* 14. \*/ 15. function Throttle(fn, options) { 16. // 判断第一个参数是函数还是布尔值 17. if (typeof fn === "function") { 18. // 说明要开启节流器了 19. // 配置 20. var options = options || { 21. context: null, 22. args: [], 23. time: 1000 24. } 25. // 清除延迟器 26. clearTimeout(fn.\_\_throttle); 27. // 因为每一个函数都要对应一个节流器， 所以我们可以给fn添加一个自定义属性 28. fn.\_\_throttle = setTimeout(function() { 29. fn.apply(options.context, options.args); 30. }, options.time); 31. } else { 32. // 说明要关闭节流器了 33. // fn 对应的是false, options对应的是函数 34. clearTimeout(options.\_\_throttle); 35. } 36. } |

# 迭代器模式

迭代器可以简单的理解为循环

迭代器模式：向外提供一个接口， 通过该接口可以按照顺序的遍历里面的内容， 并且不暴露内部结构

## 2.1 jquery中迭代器

$.each(target, fn)

target: 要遍历的目标， 可以是数组，还可以是对象

fn: 要执行的函数

有两个参数

第一个参数是：索引值

第二个参数是：成员值

作用域就是遍历到的该对象

举例：

|  |
| --- |
| 1. // 定义一个数组 2. var arr = [1, 2, 3, 4, 5, 6]; 3. $.each(arr, function(index, value) { 4. console.log(this, arguments); 5. }) |

结果：

|  |
| --- |
|  |

## 2.2 underscore中的迭代器

underscore是js的实用库

当引入underscore的时候， 向全局暴露一个变量\_

使用方式：

\_.each(targer, fn)

targer: 要遍历的目标

fn: 要执行的函数

有三个参数

第一个参数是成员值

第二个参数是索引值

第三个参数是原数组

作用域是window

举例：

|  |
| --- |
| 1. // 定义一个数组 2. var arr = ["三星", "华为", "苹果", "香蕉"]; 3. \_.each(arr, function() { 4. console.log(this, arguments); 5. }) |

结果：

|  |
| --- |
|  |

## 2.3 ES5中的迭代器

使用方式：

[].forEach(fn)

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. // 定义一个数组 2. var arr = ["三星", "华为", "苹果", "香蕉"]; 3. // 使用forEach 4. arr.forEach(function() { 5. console.log(this, arguments); 6. }) |

结果：

|  |
| --- |
|  |

## 2.4 自己的迭代器

|  |
| --- |
| 1. /\*\* 2. \* each 可以实现对数组以及对对象的遍历 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 (target instanceof Array) { 9. // 说明是传递的数组 10. for (var i = 0; i < target.length; i++) { 11. // 执行函数并传递三个参数 12. // 成员值: target[i], 索引值： i, 原数组： target 13. // fn(target[i], i, target); 14. fn.call(target[i], target[i], i, target); 15. } 16. } else if (Object.prototype.toString.call(target) === "[object Object]") { 17. // 请问有几种方式可以确定一个对象？ 18. // 1 instanceof Object 这种方式虽然可行， 但是不保险 19. // 2 constructor Object 20. // 3 Object.protoytpe.toString.call(obj) === "[object Object]" 21. // 说明传递的是对象 22. for (var i in target) { 23. // fn执行并且传递两个参数 24. // 属性名: i, 属性值： target[i] 25. // fn(i, target[i]); 26. fn.call(target[i], i, target[i]); 27. } 28. } 29. } |

# 适配器模式

将一个程序的接口转接到另外一个程序上的接口，称为适配器模式

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，于是买来一个两项插座（适配器）， 成功的将苹果手机给充上电

|  |
| --- |
| 1. <div id="box">你好，世界</div> 2. <script type="text/javascript" src="js/jquery-1.7.2.js"></script> 3. <script type="text/javascript" src="js/ickt.js"></script> 4. <script type="text/javascript"> 5. // 将代码库中的代码， 修改为juqery实现 6. var ickt = { 7. // 选择元素 8. $: function(id) { 9. return $("#" + id)[0]; 10. }, 11. // html方法 12. html: function(dom, str) { 13. return $(dom).html(str); 14. }, 15. css: function(dom, key, value) { 16. return $(dom).css(key, value); 17. }, 18. // 点击方法 19. click: function(dom, fn) { 20. return $(dom).click(fn); 21. } 22. } 23. </script> 24. <script type="text/javascript"> 25. // 获取元素 26. var $box = ickt.$("box"); 27. // console.log($box); 28. ickt.click($box, function() { 29. // 改变样式 30. // ickt.css(this, "color", "white"); 31. ickt.css(this, { 32. "backgroundColor": "blue", 33. "fontSize": "30px" 34. }) 35. // 改变内部文本 36. ickt.html(this, "今天天好热"); 37. }) |
|  |

# 组合模式

定义： 组合模式是分为“部分”-“整体”模式， 是将一个对象分成“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. // 定义方法 7. Base.prototype.add = function(child) { 8. this.children.push(child); 9. return this; 10. } 11. // 渲染方法 12. Base.prototype.render = function() { 13. // 缓存this 14. var me = this; 15. this.children.forEach(function(value) { 16. me.dom.appendChild(value.dom); 17. }) 18. return this; 19. } 20. // 定义最外层的容器 21. function Container() { 22. Base.call(this); 23. this.dom = document.createElement("ul"); 24. } 25. // 继承基类的方法 26. Container.prototype = Object.create(Base.prototype); 27. // 创建li容器 28. function Item() { 29. Container.call(this); 30. this.dom = document.createElement("li"); 31. } 32. Item.prototype = Object.create(Container.prototype); 33. // 创建最低层元素 34. function Leaf(text) { 35. this.dom = document.createTextNode(text); 36. } |

# 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

它与策略模式很相像

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |