**专题1javaScript进阶攻略**

**一、课程导学**

**高级前端进阶导学**

**1.1.1前端高级课程体系梳理**

前端发展方向：

自动化 可视化（echart、d3、three）

从内部来说我们如何应对未来：

1.编辑思想 2.学会学习源码 3.关注 4.提升角度（从软件工程来考虑-->人月神话）

从外部来说我们需要哪引起技能：

基础语法：javascript typeScript

上层框架：vue react

工具链： webpack vue-cli git 测试

全栈： node 服务器 数据库

新兴技术: fluter

技术方案：最佳实践 ssr 部署

**1.1.2高效源码学习技巧**

源码阅读步骤：

单文件型：理架构-找核心类-按照顺序依次找下去-理解方法层次，寻找关键内容

模块化型：看packjson-找到入口文件的export-依次查找对应文件export

源码阅读通常的共性：

方法一般分为三部分-参数处理部分，核心操作部分，最终输出部分

一般都会有一个init方法

通常都会有分模块，然后合并的操作

通常都会有一系列工具函数

所以我们应该注意什么内容：

值得借鉴的写法

体现思想与设计模式

让你实现与源码上实现的对比

架构的和模块的划分

复习：

享元模式：减少对象数量，提取外部数据和内部数据 -- 抽取共用的代码

装饰器模式：不改变原来的函数，新增功能

 // 需求：在a方法要输出3  改动源码是大忌

function a(){  console.log(2)  }

// 方法一：直接写

function b(){ a()  console.log(3) }

// 方法二：重写

var \_a = a()

a = function(){  \_a()  console.log(3) }

// 重写array push()

var Arrpro = Array.prototype

var arrob = object.create(Array.prototype)

arrob.push = function(){  Arrpro.push()  render() }

**1.1.3单元测试**

**单元测试技术：**

测试框架（执行方法）：Qunit jasmine mocha intern

断言库（检查结果）：chai should expect assert

mock库（数据、外部依赖）：sinon

test runner(提供执行环境，管理执行流程):karma buster.js

覆盖率工具:lastanbul

**前端单元测试：mocha+chai+sinon**

**mocha：**

1.安装

npm install [--global/--save-dev] mocha

2.基本语法(https://mochajs.org/)

常用语法

describe:对应测试的模块/函数

it:对应test case

only/skip：只运行/忽略一个或者一组用例

异步代码

callback

promise

async/await

hooke

before/after

beforeEach/afterEach:每个用列都会执行一遍

**chai：**

1.安装

npm install [--global/--save-dev] chai

2.基本语法

BDD( <http://chaijs.com/api/bdd> ):

1.expect

chai.expect -- > expect( approve({amount:3}) ).to.eql(“通过”)

2.should

chai.should() -->approve( { amount:3 } ).should.eql(“通过”)

TDD( http://chaijs.com/api/assert/ )

1.assert

chai.assert -- > assert.equal( approve({amount:3}).“通过”)

**sinon**

1.安装

npm install [--global/--save-dev] sinon

2.基本语法

Spies(spy)：提供了函数调用的信息，但不会改变其行为

Stubs：类似Spies，但是是完全替换目标函数

Mocks：组合spies与stubs，可替换一个完整的对象

1. **Karma**

1.安装

npm install -g karma

npm install -g karma-cli

2.执行

karma init 生成配置项

Karma start 启动karma

1. **设计模式jQuery源码分析**

**重点： 插件的设计思想 callbacks 单一职责 data数据缓存+事件系统**

**核心功能**

**2.1.1jQuery整体架构-核心功能函数揭秘**

**Sizzle**

**2.2.1选择器**

**一、常用jQuery选择器接口**

1.传入对象

$(this) $(document) 把传入的对象包装成jQuery对象

2.传入函数

$(function(){}) 这个是在页面DOM文档加载完成后加载执行的，

等效于在DOM加载完毕后执行了$(document).ready()方法

3.传入字符串（查询）

$('.box')查询DOM节点包装成jQuery对象

4.传入HTML字符串（创建）

$('<div>')创建DOM节点包装成jQuery对象

5. 空

$()创建jQuery对象

二、问题分析

1.如何把创建的DOM节点包装成jQuery对象

context.createElement创建DOM节点存储在数组中,调用merge方法把数组中

存储的DOM节点的成员添加到jQuery实例对象上

2.jQuery实例对象length属性的作用

存储DOM节点的数组对象平滑地添加到jQuery实例对象上

3.merge方法的应用场景有哪些

合并数组

把数组成员合并在length属性的对象上

4.$(document).ready()与$(function(){})的关系

$(document).ready()是对document.DOMContentLoaded事件封装

$(function(){})每次调用$()传入的参数会收集在readyList数组中，

当document.Contentedness事件触发依次执行readyList中收集的处理函数

三、课后作业（阅读源码，了解它们如何进行处理）

1.根据 [/^<(\w+)\s\*\/?>(?:<\/\1>|)$/](https://www.cnblogs.com/yina-526/p/11138199.html) ，分析其正则执行过程中如何进行过滤？包含哪些执行步骤？

var rejectExp = /^<(\w+)\s\*\/?>(?:<\/\1>|)$/;  
var parse = rejectExp.exec('<a>')  
console.log(parse[1]);//a  
分析如何过滤:  
 (1)正则表达式都包含在//中  
 (2)^< 表示字符串是以'<'开头的  
 (3)(\w+)其中\w为匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9\_]'。  
 (4)\s\* 匹配任何空白字符，包括空格、制表符、换页符等等。  
 (5)\/?>表示匹配"/"零次或一次,后面加">"  
 (6)(?:<\/\1>|)表示后面接'<\/\1>'或者不接任何内容  
 (7)<\/\1> 其中‘\/’表示匹配‘/’符号，‘\1’指前面的‘\w+’  
 (8)$结束  
包含的执行步骤：  
 (1)标签以 '<'开头  
 (2)在开始标签内进行匹配，包括  
 任意单词字符+  
 0个或多个空白字符+  
 0个或1个'\'+  
 '>'  
 +或不+'</'+上一个标签内的字符

1. 当selector传来一个函数时，怎么进行处理？

**回调对象设计**

**2.3.1 Callbacks入门&原理分析**

$.callbacks用于管理函数队列

通过add添加处理函数到队列当中，通过fire去执行这些处理函数

// 通过调用$.callbakcs,获取到一个callbacks实列

        var cb = $.callbacks( )

        // add向内部队列添加函数

        cb.add( function(){   console.log('add one') })

        // fire依次执行队列里的函数

        cb.fire() // add one

**$.callbacks通过字符串参数的形式，支持4种的特定功能:**

1.once

函数队列只执行一次

2.unique

往内部队列添加的函数保持唯一，不能重复添加

3.stopOnFalse

内部队列里的函数是依次执行的，当某个函数的返回值是false时，停止继续剩下的函数

4.memory

当函数队列fire一次过后，内部会记录当前fire的参数。当下次调用add的时候，会把记录的参数传递给新添加的函数并立即执行这个新添加的函数

**异步回调解决方案**

**2.4.1 延时对象-Deferred概念&源码剖析**

Deferred是Promise/A+规范的具体实现

**Deferred API(https://www.jquery123.com/category/deferred-object/):**

1.jQuery.Deferred()

一个构造函数，返回一个链式实用对象方法注册多个回调，回调队列，调用回调队列，并转达任何同步或异步函数的成功或失败状态

2.deferred.done()

当Deferred（延迟）对象解决时，调用添加处理程序。

3.deferred.fail()

当Deferred（延迟）对象拒绝时，调用添加的处理程序。

4.deferred.progress()

当Deferred（延迟）对象生成进度通知时，调用添加处理程序。

5.jQuery.when()

提供一种方法来执行一个或多个对象的回调函数， Deferred(延迟)对象通常表示异步事件

6..promise()

返回一个 Promise 对象，用来观察当某种类型的所有行动绑定到集合，排队与否还是已经完成。

**事件绑定**  
**2.5.1 事件绑定-bind/delegate/on**  
**2.5.2 事件绑定-体系结构/委托设计**

jQuery事件处理处理机制解决问题：

1.不同浏览器下的事件兼容

2.一个事件类型添加多个事件函数，一次添加多个事件类型的处理函数

3.简洁的定义方式

4.允许自定义事件



**2.5.3 事件绑定-自定义设计&模拟事件**

**自定义事件**

在实际开发中，或许原生的javaScript事件或者jQuery事件不能满足开发的要求，这时候就需要我们自己编写相应的事件

添加自定义事件：

通过jQuery的bind( )或者on( )方法添加自定义事件如下代码，自定义一个createEvent事件

$('#Id').on('createEvent',function(){

            console.log('createEvent')

        })

触发自定义事件：

通过jQuery的trigger( )方法触发自定义事件createEvent

$('#Id').trigger('createEvent')

**模拟事件**

trigger( )方法触发事件后，会执行浏览器默认行为

$('input').trigger('focus')

以上代码不仅会触发input元素绑定的focus事件，还会使得input元素本身得到焦点(这是浏览器的默认行为)

如果只想触发绑定的focus事件，而不想执行浏览器的默认行为，可以使用另一种类似的方法triggerHandler( )方法

**DOM操作 & 样式操作**

**2.6.1 DOM操作方法-html,text,css**

html( ) 用于读取和修改元素的HTML标签

text( ) 用来读取或修改元素的纯文本内容

css( ) 设置或返回被选元素的一个或多个样式属性

text( ),html( ),csss( )都是通过jQuery.access提供底层支持

jQuery.access( )是一个多功能方法，作为set和get值来使用

**2.6.2 - 2.6.3 DOM操作核心 & CSS / addClass原理分析**

addClass( )

对元素的样式操作，底层的实现修改元素的className值

实现的功能:

增加一条样式规则 addClass(“box”)

增加多条样式规则 addClass(“box item”)

**高级进阶-动画队列**

**2.7.1 动画队列 / 动画设计 / 动画实现**

**1.动画队列 （Queue队列）**

Quenu队列，如同data数据缓存与Deferred异步模型一样,都是jQuery库内部实现的基础设施

Quenu队列是animate动画依赖的基础设施，整个jQuery中队列仅供给动画使用

Quenu数据存储重度依赖data函数

动画会调用队列，队列会调data数据接口还保存队列里面的动画数据

**2.数据缓存接口.data( )**

****

**3.数据缓存&队列&动画**



**4.出列 - 入列**



1. **函数式编程underscore源码分析**

**重点：迭代器设计 模板函数 数据流的处理**

**Underscore设计篇**

**3.1.1函数式编程思想概述**

 纯函数：

        定义：对于相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用，也不依赖外部

         环境的状态

        在JavaScript中，对于数组的操作有些是纯的，有些是不纯的

  var arr = [1,2,3,4,5]

    // 纯函数

    arr.slice(0.3) // [1,2,3]   arr.slice(0.3) // [1,2,3]

    // 非纯函数

    arr.splice(0,3) // [1,2,3] arr.splice(0,3) // [4,5]

函数式编辑为何排斥不纯的函数？

         非纯函数中，函数的行为需要由外部的系统环境决定。也就是说此函数行为不仅取决于输入的参数age，还取决于一个外部的变量 timeOfLife，这种对于外部状态的依赖，是造成系统复杂性大大提高的主要原因

var timeOfLife = 20

   // 纯函数

   function test(age){ return age > 20 }

   // 非纯函数

   function test(age){ return age > timeOfLife }

 函数柯里化：

        定义： 向函数传递一部分参数来调用它，让它返回一个函数去处理剩下的参数

             事件上柯里化是一种，‘预加载‘函数的方法，通过传递较少的参数，得到一个已经记住了这些参数的新函数，某种意义上讲，这是一种对参数的’缓存‘，是一种非常高效的编写函数的方法

var timeOfLife = 20

    function test(timeOfLife){

       return function(age){ return age > timeOfLife }

    }

    var testing = test(20)

    testing(18) // false

 函数组合：

        为避免写出不优雅的包菜式代码h(g(f(x))),我们需要用到函数组合

        我们定义的compose就像双面胶一样，可以把任何两个纯函数结合到一起，也可以扩展出组合n个函数的n面胶

  // 两个函数的组合

   var compose = function(f,g){ return function(x){  return f(g(x)) } }

   var mult = function(x){ return x\*5 }

   var add = function(x){ return x+1  }

   compose(mult,add)(2) // 5

命令代码：

        通过编写一条又一条指令，让计算机执行一些动作，其中一般会涉及许多繁杂的细节声明式代码

        通过写表达式的方式，声明我们想干什么，而不是通过一步一步的指示

        声明式代码，是函数式编程的一个明显好处一一编写、优化代码时能更专注

  // 命令式

    var rest = [] , arr = [4,9,16,25,4,16]

    for(var i = 0;i<arr.length;i++){

       if(rest.indexOf(arr[i]) === -1){  rest.push(arr[i]) }

    }

    // 声明式

    var rest = arr.map(Math.sqrt)

 总结：

        函数对于外部状态的依赖，是造成系统复杂性大大提高换主要原因

        代码书写中让函数尽可能地纯净

        函数式编程不是万能，它与opp一样，只是一种编程范式

        为降低软件复杂度，opp的方式是靠良好的封装、继承、多态以及接口定义。函数式编程则是靠纯函数以及它们的组合、柯里化等技术

**3.1.2underscore整体结构&面向对象**

 \_对象：

underscore有下划线的意思，所以underscore通过一个下划线变量\_来标识自身

\_是一个函数对象，之后所有的api都会被挂载到这个对象上，如\_.each , \_.map

 \_( )：

虽然underscore推崇函数式编程，但也支持面向对象风格的函数调用，仅需要通过\_()来包裹对象即可

\_([2,3,4])会创建一个新的underscore对象（从而能够调用underscore提供的方法）, 并在this.\_wrapped中存储传入的数据

mixin：

mixin模式是增加代码复用度的一个广泛使用的设计模式

链接式调用：

开启链接式调用方法：\_.chain( )

关启链接式调用方法：\_.value( )

**3.1.3undefined的处理/函数式编程(回调)iteratee设计**

void 运算符通常只用于获取 undefined的原始值

void 0和undefined是等价，因此可以用来判断undefined

**3.1.4 rest参数/underscore创建对象方式**

\_ . restArguments：

underscore的官方实现，它暴露了一个\_.restArguments方法，通过给该方法传递一个函数，能够使得函数支持rest参数

\_ . baseCreate(创建对象)：

underscore利用baseCreate创建对象时，会先检查当前环境是否已支持Object.create，如不支持，会创建一个简易的polyfill

**Underscore辅助功能篇**

**3.2.1Map-reduce真值检测函数**

**3.2.2 indexOf lastIndexOf条件查询-背后索引查询器实现**

\_.createdIndexFinder:

underscore中通过内置的工厂函数createIndexFinder来创建一个索引查询器

\_.indexOf及\_.lastIndexOf 正是由该函数所创建的

createdIndexFinder( dir ,predicateFind , sortedIndex )接受3个参数

dir查询方向 \_.indexOf是正向查询 \_.lastIndexOf是反向查询

**predicateFind真值检测函数，该函数只有在查询元素不是数字(NaN)时才会使用**

sortedIndex有序数组的索引获得函数。如果设置了该参数，将假定数组已有序，从而更加高效地通过针对有序数组的查询函数（比如二分查找等）来优化查询性能

**3.2.3乱序数组-洗牌算法**

洗牌算法：

算法思路在宏观上可以概括为：将集合视为牌堆，不停地从牌堆中抽牌构成新的牌堆，直至新牌堆的牌数到达预设数量

从underscore 1.9版本开始，洗牌算法通过\_.sample实现

\_.sample(array,n):从array随机取出n个样本

underscore中的抽样函数正基于洗牌算法

**Underscore数组篇**

**3.3.1 underscore数组定位&摊平数组**

数组定位

\_.initial( array,n ) : 获得array的除了最后n个元素以外的元素

\_.rest( array ,n ) : 返回array中除了前n个元素外的所有元素

摊平数组

.flatten(array,shallow) : 摊平array，通过shallow指明是深度摊平还是浅摊平

**3.3.2 数组运算uniq去重函数 原生对象扩展Underscore函数篇**

\_.unique: 重写数组去重函数

\_.computed: 去掉数组中所有的假值

\_.range: 返回某一个范围内的数值组成的数组

**Underscore函数篇**

**3.4.1偏函数(partial)-缓存(memoize)使用场景源码分析**

\_.partial : 反映了新函数是原函数的一部分

应用一个函数，填充在任意个数的参数，不改变其动态this值，和bind方法很相近

\_.memoize : memoize方法可以缓存某函数的计算结果（我搞不懂）

**3.4.2延时执行函数- compose-escape字符串逃逸**

字符串逃逸：防止用户注入代码，造成xss攻击，所以使用正则把字符转为实体名称

**3.4.3throttle与debounce 概念解析源码实现**

throttle节流函数：在一定时间内，调用了n次，只会执行一次

debounce防抖函数：在一定时间内，调用了n次，只会根据最后一次调用重新计算时间

**Underscore对象篇**

**3.5.1 属性操作 & createAssigner**

\_.has(object,key): 检测object对象是否包含给定的key属性 == Object.hasOwnProperty(key)

\_.keys(object): 获取object对象所有的属性名称 == Object.keys(obj)(但不兼容IE)

\_.invert(object)：返回一个object副本，使其键(keys)和值(values)对象

createAssigner( ): 对象属性的扩展 == Object.assign

**3.5.2 deepclone 深度克隆 & pick**

**3.5.3template模板引擎设计**

1. **模块化编程自研模块加载器**

**模块加载器设计概要**

**4.1.1 模块系统概述-自定义模块规范-书写约定**

**4.1.2 加载器结构设计导论**

****

模块部分：每个模块创建都先初始化数据，存储在缓存对象中

数据初始化：加载器中设计了一个名为module的构造函数，每个模块都是此构造函数实例对象

模块存储：加载器中设计了一个名为cache缓存对象，每个文件（模块）都会存储在cache对象中

具体存储方式：{ 当前模块绝对路径：new Module( ) }

注意：当前模块的绝对路径是通过资源部分，资源定位方法实现的

资源部分: 资源定位、依赖管理是加载器设计两大核心

资源定位：加载器中设计了一个resolve()的方法把模块名解析成绝对路径格式

模块名的获取：startUp.use( ['a.js','b.js ')

startUp.use()加载器启动方法，启动时会去调用传入数组列表中的模块

动态加载script文件：通过加载器resolve()方法把模块名解析成绝对路径格式

动态创建script document.create('script') src指向当前模块绝对路径地址

加载文件同时，模块加载器解析当前模块所依赖的模块以数组的形式存储

依赖管理：已知当前模块在cache中的形态，{ '当前模块绝对路径' : new Module( ) }

换算： {'当前模块绝对路径' : { url:"当前模块绝对路径" , deps:[ ] } }

deps存储当前模块的依赖列表，依赖列表通过动态加载script文件正则解析获取

重点：解析依赖 -》 获取依赖模块绝对路径地址 -》 动态加载 -》 提取依赖 -》解析依赖

递归方式加载所有模块，直至模块全部加载完毕

**模块加载器设计核心**

**4.2.1 Module构造器设计 模块数据初始化**

**4.2.2 模块资源定位-异步加载**

**4.2.3 依赖解析-依赖管理处理方案**

**4.2.4 模块(路径)短名称配置解决方案**