目 录

[第一部分 JavaScriptCore 20](#_Toc337)

[第一章 正则表达式 20](#_Toc26254)

[1. 概述 20](#_Toc12490)

[1.1. 定义 20](#_Toc31404)

[1.2. 何时 20](#_Toc8176)

[2. 使用方式 20](#_Toc29015)

[2.1. 明文 20](#_Toc7100)

[2.2. 字符集 20](#_Toc23094)

[2.3. 预定义字符集 21](#_Toc22909)

[2.4. 量词 21](#_Toc4218)

[2.4.1. 有明确数量边界的 22](#_Toc24123)

[2.4.2. 没有明确数量边界的 22](#_Toc10559)

[2.5. 选择和分组 22](#_Toc17932)

[2.5.1. 选择 22](#_Toc26117)

[2.5.2. 分组 22](#_Toc4912)

[2.6. 匹配特殊位置 23](#_Toc21471)

[2.7. 密码强度 23](#_Toc10991)

[3. 正则表达式汇总 23](#_Toc17749)

[第二章 String 24](#_Toc5777)

[1. 概念 24](#_Toc1029)

[1.1. 与数组相比 24](#_Toc4325)

[2. API 24](#_Toc16511)

[2.1. 强调 24](#_Toc27954)

[2.2. 大小写转换 25](#_Toc14848)

[2.3. 获取指定位置字符 25](#_Toc2702)

[2.4. 选取子字符串 25](#_Toc9455)

[2.5. 查找关键词 25](#_Toc24287)

[2.5.1. 查找一个固定的关键词出现的位置 25](#_Toc20037)

[2.5.1.1. 语法 25](#_Toc16162)

[2.5.1.2. 优缺点 26](#_Toc11087)

[2.5.1.3. 专门查找最后一个关键词的位置 26](#_Toc30377)

[2.5.1.4. php中 26](#_Toc16492)

[2.5.2. 判断是否包含符合规则的关键词 26](#_Toc3427)

[2.5.2.1. 语法 26](#_Toc24235)

[2.5.2.2. 优缺点 27](#_Toc23933)

[2.5.3. 获取所有关键词的内容 27](#_Toc30587)

[2.5.3.1. 语法 27](#_Toc23248)

[2.5.3.2. 返回包含所有关键词的数组 27](#_Toc32466)

[2.5.3.3. 优缺点 27](#_Toc17147)

[2.5.3.4. PHP中 27](#_Toc22375)

[2.5.4. 即获得每个关键词的内容，又获得每个关键词的位置 27](#_Toc25372)

[2.6. 替换 28](#_Toc25087)

[2.6.1. 简单替换 28](#_Toc18564)

[2.6.2. 高级替换 28](#_Toc20440)

[2.6.3. 衍生操作 28](#_Toc7134)

[2.6.3.1. 删除 28](#_Toc6584)

[2.6.3.2. 格式化 28](#_Toc8853)

[2.7. 切割 29](#_Toc18477)

[2.7.1. 简单切割 29](#_Toc138)

[2.7.2. 复杂切割 29](#_Toc24311)

[2.7.3. 返回值 29](#_Toc25684)

[2.7.4. 固定套路 29](#_Toc3264)

[2.7.4.1. 语法 29](#_Toc23200)

[3. split与slice区别 30](#_Toc2440)

[3.1. split 30](#_Toc10601)

[3.1.1. 定义 30](#_Toc8555)

[3.1.2. 语法 30](#_Toc23075)

[3.1.3. 提示和注释 30](#_Toc5810)

[3.2. slice 30](#_Toc4623)

[3.2.1. 定义 30](#_Toc7499)

[3.2.2. 语法 30](#_Toc4685)

[3.2.3. 提示和注释 31](#_Toc793)

[第三章 RegExp 31](#_Toc17771)

[1. 定义 31](#_Toc29635)

[2. 适用情况 31](#_Toc11905)

[3. 创建 31](#_Toc31715)

[3.1. 直接量 31](#_Toc28188)

[3.2. 用new 32](#_Toc20164)

[4. API 32](#_Toc29569)

[4.1. 验证 32](#_Toc11034)

[4.2. 查找关键词 32](#_Toc25875)

[4.2.1. 即查找内容又查找位置 32](#_Toc24140)

[4.2.2. 返回值 32](#_Toc24801)

[4.2.3. exec做三件事 33](#_Toc9041)

[4.2.4. 固定套路 33](#_Toc32421)

[第四章 Function 33](#_Toc16685)

[1. 什么是 33](#_Toc1724)

[2. 为什么 33](#_Toc11915)

[3. 何时 33](#_Toc11742)

[4. 创建 34](#_Toc27408)

[4.1. 创建的方式（3种） 34](#_Toc25509)

[4.1.1. 声明 34](#_Toc1112)

[4.1.1.1. hoist（声明提前） 34](#_Toc29209)

[4.1.2. 直接量 34](#_Toc11193)

[4.1.3. 用new 34](#_Toc24288)

[4.1.4. 三种定义方式的对比 35](#_Toc11073)

[4.2. 参数 35](#_Toc17101)

[4.3. 返回值 35](#_Toc15245)

[5. 调用 35](#_Toc13011)

[6. 重载(overload) 36](#_Toc2406)

[6.1. 什么是 36](#_Toc28074)

[6.2. 为什么 36](#_Toc29009)

[6.3. 何时 36](#_Toc23479)

[6.4. 如何 36](#_Toc26726)

[6.4.1. 问题 36](#_Toc6885)

[6.4.2. 解决 36](#_Toc1332)

[6.4.3. arguments 36](#_Toc24211)

[6.4.3.1. 类数组对象vs数组 36](#_Toc28799)

[6.4.4. 如何实现 37](#_Toc27600)

[6.4.5. 延伸 37](#_Toc1296)

[6.4.5.1. caller 37](#_Toc31851)

[6.4.5.2. callee 37](#_Toc26541)

[6.4.5.3. 打针的三种方式apply、bind、call 38](#_Toc24045)

[7. 匿名函数 39](#_Toc9652)

[7.1. 定义 39](#_Toc31674)

[7.2. 为什么 39](#_Toc10948)

[7.3. 何时 39](#_Toc7879)

[7.3.1. 回调callback 39](#_Toc16908)

[7.3.2. 自调 39](#_Toc4676)

[7.3.2.1. 自调函数的多种写法 40](#_Toc2995)

[8. 作用域和作用域链（重要） 41](#_Toc22673)

[8.1. 作用域(scope) 41](#_Toc8232)

[8.1.1. 分类 41](#_Toc29121)

[8.1.1.1. 全局作用域 41](#_Toc11014)

[8.1.1.2. 函数作用域 41](#_Toc16249)

[8.1.2. 函数的生命周期 42](#_Toc8718)

[8.1.2.1. 程序开始执行前 42](#_Toc14451)

[8.1.2.2. 定义函数时 42](#_Toc8455)

[8.1.2.3. 调用函数时 43](#_Toc29038)

[8.1.2.4. 函数调用后 44](#_Toc31108)

[8.1.3. 作用域链(scope chain) 44](#_Toc28245)

[8.1.4. 函数中没有对象/this时 44](#_Toc15831)

[9. 闭包（closure）（重要） 45](#_Toc1503)

[9.1. 适用情况 45](#_Toc2248)

[9.2. 语法 45](#_Toc13054)

[9.3. 闭包如何形成 49](#_Toc4030)

[9.4. 闭包的缺点 49](#_Toc19142)

[9.5. 鄙视：判断闭包程序的输出结果（画简图） 49](#_Toc15233)

[9.6. 结论 50](#_Toc711)

[9.7. 返回内层函数的对象的方法（3种） 50](#_Toc9152)

[第五章 OOP（重点） 51](#_Toc21222)

[1. OOP面向对象编程概述 51](#_Toc27638)

[2. 封装 51](#_Toc22320)

[2.1. 概述 51](#_Toc13425)

[2.2. 何时 51](#_Toc14672)

[2.3. 如何（3种） 52](#_Toc10366)

[2.3.1. 直接量（方法1） 52](#_Toc9856)

[2.3.1.1. 语法 52](#_Toc15114)

[2.3.1.2. 何时 52](#_Toc15916)

[2.3.1.3. 问题 52](#_Toc27959)

[2.3.1.4. 解决 53](#_Toc1708)

[2.3.2. 用new创建一个对象（方法2） 54](#_Toc5819)

[2.3.2.1. 创建空对象 54](#_Toc8374)

[2.3.2.2. 为空对象添加新属性 54](#_Toc9723)

[2.3.2.3. 何时 55](#_Toc30586)

[2.3.3. 前两种方法对比 55](#_Toc15939)

[2.3.4. 用构造函数（方法3） 55](#_Toc23175)

[2.3.4.1. 概念 55](#_Toc3216)

[2.3.4.2. 如何：2步 55](#_Toc6118)

[2.3.4.3. 优点 56](#_Toc12223)

[2.3.4.4. 问题 56](#_Toc8214)

[3. 继承 57](#_Toc14111)

[3.1. 概念 57](#_Toc19141)

[3.2. 原型对象 58](#_Toc4799)

[3.2.1. 如何 58](#_Toc19070)

[3.2.1.1. 不用创建 58](#_Toc27368)

[3.2.1.2. new的第2步（何时如何继承） 58](#_Toc3752)

[3.2.1.3. 向原型对象中添加共有成员 58](#_Toc16543)

[3.2.2. 自有属性和共有属性 58](#_Toc3714)

[3.2.3. 内置类型/对象 60](#_Toc1843)

[3.2.3.1. 问题 60](#_Toc3331)

[第六章 内置类型 60](#_Toc10005)

[1. 概念 60](#_Toc9422)

[2. 分类 60](#_Toc7306)

[3. 包装类型 61](#_Toc9359)

[3.1. 原始类型与引用类型 62](#_Toc14598)

[3.1.1. 两种访问方式 62](#_Toc18303)

[3.1.2. 两种类型复制 63](#_Toc1061)

[3.1.3. 两种变量类型检测 64](#_Toc29420)

[3.2. 原型链（prototype chain） 64](#_Toc24990)

[3.2.1. 原型链 vs 作用域链 64](#_Toc10470)

[3.3. 判断对象的类型 65](#_Toc2982)

[3.3.1. 类型1typeof 65](#_Toc8587)

[3.3.2. 类型2类型.prototype.isPrototypeOf 65](#_Toc11679)

[3.3.3. 类型3child instanceof构造函数 65](#_Toc18313)

[3.3.4. 类型4class 65](#_Toc2311)

[3.3.5. 类型5Array.isArray(obj) 66](#_Toc31179)

[3.3.5.1. 鄙视 66](#_Toc7929)

[3.4. toString()、toLocalString()、valueOf()三种方法之间的区别 66](#_Toc17259)

[3.4.1. valueOf() 66](#_Toc27344)

[3.4.2. toString() 68](#_Toc1122)

[3.4.3. toLocaleString() 69](#_Toc24220)

[3.4.4. 三种对比 70](#_Toc6286)

[3.4.4.1. Array 70](#_Toc6285)

[3.4.4.2. Boolean 70](#_Toc14335)

[3.4.4.3. Date 70](#_Toc16522)

[3.4.4.4. Math 71](#_Toc25269)

[3.4.4.5. Number 71](#_Toc14207)

[3.4.4.6. String 71](#_Toc31534)

[3.4.4.7. toString()方法与toLocalString()方法区别 72](#_Toc20016)

[4. 多态 72](#_Toc3694)

[5. 静态 73](#_Toc10130)

[5.1. 静态方法vs实例方法 73](#_Toc32123)

[5.1.1. 实例方法 73](#_Toc19190)

[5.1.2. 静态方法 73](#_Toc6992)

[6. 自定义继承 74](#_Toc19599)

[6.1. 仅修改一个对象的父对象 74](#_Toc11226)

[6.2. 修改所有子对象的父对象 74](#_Toc26389)

[6.3. 仅基于现有父对象，创建子对象，并扩展自有属性 74](#_Toc21378)

[6.3.1. 鄙视：模拟实现Object.create() 75](#_Toc30611)

[6.4. 两种类型间的继承 75](#_Toc17478)

[6.4.1. 何时 75](#_Toc5288)

[6.4.2. 如何 75](#_Toc7293)

[6.4.2.1. 定义抽象父类型 75](#_Toc1216)

[6.4.2.2. 让子类型继承父类型 75](#_Toc19625)

[第七章 ES5 77](#_Toc13063)

[1. 严格模式 77](#_Toc17682)

[1.1. 如何 77](#_Toc306)

[1.2. 要求 77](#_Toc28924)

[2. 保护对象 78](#_Toc1104)

[2.1. 概述 78](#_Toc29870)

[2.2. 对象属性的保护（仅保护一个属性） 78](#_Toc27956)

[2.2.1. 内部属性 78](#_Toc3247)

[2.2.2. 命名属性 78](#_Toc8518)

[2.2.2.1. 数据属性 78](#_Toc8157)

[2.2.2.2. 访问器属性 80](#_Toc28340)

[2.2.2.3. 特性的默认值 81](#_Toc25419)

[2.3. 保护对象结构——防篡改 81](#_Toc24415)

[2.3.1. 防扩展 81](#_Toc9741)

[2.3.2. 密封 83](#_Toc2037)

[2.3.3. 冻结 83](#_Toc19542)

[3. Object.create() 83](#_Toc23832)

[3.1. 替换函数中的this 84](#_Toc31893)

[3.2. 关于Object类型 85](#_Toc9860)

[4. 数组API 86](#_Toc22135)

[4.1. 查找元素在数组中的位置 86](#_Toc10380)

[4.2. 判断 86](#_Toc2805)

[4.3. 遍历 87](#_Toc7789)

[4.4. 过滤 88](#_Toc9690)

[4.5. 汇总 88](#_Toc13381)

[4.6. bind函数 89](#_Toc7263)

[第八章 ES6 89](#_Toc26030)

[1. let 89](#_Toc4566)

[2. 参数增强 90](#_Toc3666)

[2.1. 默认值 90](#_Toc2679)

[2.2. 剩余参数(rest) 91](#_Toc9196)

[2.3. 散播(spread) 92](#_Toc17869)

[3. 箭头函数 92](#_Toc3481)

[4. 模板字符串 93](#_Toc1368)

[5. 解构 93](#_Toc27237)

[5.1. 数组解构 93](#_Toc26964)

[5.2. 对象解构 94](#_Toc26991)

[5.3. 参数解构 94](#_Toc7383)

[6. for of简化遍历 96](#_Toc10278)

[6.1. 遍历（索引）数组：3种 96](#_Toc5012)

[6.2. for of vs for in 96](#_Toc28518)

[7. class 97](#_Toc28427)

[7.1. 对对象直接量的简化 97](#_Toc11902)

[7.2. class 97](#_Toc3568)

[7.3. 继承extends 98](#_Toc19302)

[7.4. 静态方法 100](#_Toc8205)

[7.4.1. 原理 101](#_Toc22962)

[7.4.2. vs原型对象方法 101](#_Toc10729)

[8. 访问器属性 101](#_Toc5632)

[9. Promises（重点） 102](#_Toc32508)

[9.1. 定义函数时 104](#_Toc12786)

[9.2. 串联前后两个函数 104](#_Toc11838)

[9.3. promise延伸 106](#_Toc7422)

[9.3.1. promise.all 106](#_Toc31812)

[9.3.2. Promise.race() 106](#_Toc6284)

[第九章 ES7 106](#_Toc8023)

[1. 小知识 108](#_Toc6719)

[1.1. id属性值 108](#_Toc21792)

[1.2. 函数名是变量 108](#_Toc21856)

[1.3. 参数赋值问题 108](#_Toc26339)

[1.4. concat 108](#_Toc19650)

[1.5. 从局部到全局作用域的顺序，查找变量所在位置 109](#_Toc6238)

[1.6. 关于对象与数组的比较 109](#_Toc6292)

[第二部分 DOM & BOM 109](#_Toc7873)

[第一章 DOM 110](#_Toc10240)

[1. 什么是DOM 110](#_Toc26723)

[2. DOM Tree 110](#_Toc17545)

[2.1. 节点对象 110](#_Toc796)

[3. 查找 111](#_Toc3455)

[3.1. 不用查找就可直接获得 111](#_Toc7790)

[3.2. 按节点间关系查找 111](#_Toc5133)

[3.2.1. 节点数 112](#_Toc20443)

[3.2.1.1. 父子关系 112](#_Toc17572)

[3.2.1.2. 兄弟关系 112](#_Toc13089)

[3.2.2. 元素树 112](#_Toc28882)

[3.2.2.1. 父子关系 112](#_Toc3384)

[3.2.2.2. 兄弟关系 112](#_Toc9417)

[3.2.2.3. 鄙视：递归遍历指定父节点下的所有后代节点 113](#_Toc4157)

[3.3. 按HTML特征查找（4种） 113](#_Toc3614)

[3.3.1. 按id查找 113](#_Toc4014)

[3.3.2. 按标签名查找 114](#_Toc9615)

[3.3.3. 按name属性查找 114](#_Toc12205)

[3.3.4. 按class属性查找 114](#_Toc7109)

[3.4. 按选择器查找 114](#_Toc12057)

[3.4.1. 如果只找一个元素 114](#_Toc18805)

[3.4.2. 如果找多个元素 114](#_Toc11206)

[3.5. 鄙视:按HTML查找 vs 按选择器查找 115](#_Toc9557)

[4. 修改 115](#_Toc31001)

[4.1. 内容 115](#_Toc5802)

[4.2. 属性（3种） 116](#_Toc9421)

[4.2.1. HTML标准属性 116](#_Toc29072)

[4.2.1.1. 用核心DOM API 116](#_Toc25271)

[4.2.1.2. HTML DOM API 116](#_Toc4640)

[4.2.2. 状态属性 116](#_Toc8121)

[4.2.3. 自定义扩展属性 116](#_Toc30239)

[4.2.3.1. 添加 117](#_Toc15691)

[4.3. 样式 118](#_Toc28898)

[5. 添加/删除元素 120](#_Toc16955)

[5.1. 添加元素（3步） 120](#_Toc13188)

[5.1.1. 创建空元素 120](#_Toc29341)

[5.1.2. 设置关键属性 120](#_Toc29607)

[5.1.3. 将新元素对象添加到DOM树上 120](#_Toc4092)

[5.1.3.1. 优化 122](#_Toc3409)

[5.1.3.2. 页面加载原理 122](#_Toc19530)

[5.2. 删除 124](#_Toc8359)

[6. HTML DOM常用对象: 125](#_Toc3366)

[6.1. Image对象 125](#_Toc29335)

[6.2. select/option 125](#_Toc19049)

[6.2.1. select对象 125](#_Toc20397)

[6.2.2. option对象 125](#_Toc27173)

[6.3. table/子元素 127](#_Toc27298)

[6.3.1. table管着行分组 127](#_Toc19823)

[6.3.1.1. 添加行分组 127](#_Toc15470)

[6.3.1.2. 删除行分组 127](#_Toc7343)

[6.3.1.3. 获取 127](#_Toc23011)

[6.3.2. 行分组管着行 127](#_Toc5968)

[6.3.2.1. 添加 127](#_Toc4889)

[6.3.2.2. 删除 128](#_Toc2561)

[6.3.2.3. 获取 128](#_Toc4910)

[6.3.3. 行管着格 128](#_Toc26599)

[6.3.3.1. 添加 128](#_Toc11055)

[6.3.3.2. 删除 128](#_Toc9836)

[6.3.3.3. 获取 128](#_Toc2342)

[6.3.4. form/表单元素 130](#_Toc14488)

[6.3.4.1. HTML 表单标签 130](#_Toc25749)

[6.3.4.2. form对象 130](#_Toc11793)

[6.3.4.3. 表单元素 131](#_Toc13301)

[第二章 BOM 131](#_Toc23348)

[1. 什么是BOM 131](#_Toc23272)

[2. window对象 131](#_Toc23273)

[3. 打开和关闭新链接 132](#_Toc32576)

[3.1. 在当前窗口打开，可后退 132](#_Toc28)

[3.2. 在当前窗口打开，不可后退 132](#_Toc25360)

[3.3. 在新窗口打开，可打开多个 132](#_Toc15368)

[3.4. 在新窗口打开，只能打开一个 132](#_Toc21593)

[4. history 132](#_Toc26538)

[5. location 133](#_Toc1816)

[5.1. 功能 133](#_Toc4642)

[5.1.1. 在当前窗口打开新链接，可后退 133](#_Toc5303)

[5.1.2. 在当前窗口打开新链接，禁止后退 134](#_Toc17100)

[5.1.3. 重新加载页面：刷新（2种） 134](#_Toc29756)

[5.2. 总结（常用的属性和方法） 134](#_Toc21838)

[6. navigator 134](#_Toc6308)

[6.1. cookie 135](#_Toc1778)

[6.2. .plugins 135](#_Toc3908)

[6.3. .userAgent 135](#_Toc18442)

[7. 定时器（2种） 136](#_Toc1590)

[7.1. 周期性定时器 136](#_Toc14635)

[7.2. 一次性定时器 137](#_Toc8977)

[8. event 138](#_Toc15231)

[8.1. 如何绑定事件处理函数（3种） 138](#_Toc5248)

[8.2. 事件模型：3个阶段 139](#_Toc29834)

[8.3. 冒泡 139](#_Toc23926)

[8.3.1. 事件对象 139](#_Toc26887)

[8.3.2. 取消事件 140](#_Toc3849)

[8.3.3. 2个难题 141](#_Toc12239)

[8.3.4. 典型应用场景 142](#_Toc3657)

[8.3.5. 鼠标位置（3组） 142](#_Toc1261)

[8.3.6. 页面滚动 143](#_Toc11646)

[第三部分 jQuery 143](#_Toc1603)

[第一章 概述 143](#_Toc23737)

[1. 什么是jQuery 143](#_Toc28276)

[2. 为什么 144](#_Toc5027)

[3. 下载 144](#_Toc2020)

[3.1. 版本 144](#_Toc26337)

[3.1.1. 未压缩版 144](#_Toc8474)

[3.1.2. 压缩版 145](#_Toc3471)

[4. 3.x的新特性（10个） 145](#_Toc21698)

[5. 引入网页（2种） 145](#_Toc32670)

[5.1. 原理 146](#_Toc19229)

[5.1.1. 引入jquery.js文件时 146](#_Toc23577)

[5.1.2. 使用jQuery简化API 146](#_Toc1961)

[5.1.2.1. 创建jQuery类型子对象(2种) 146](#_Toc32414)

[6. jQuery API的三大特点 146](#_Toc11583)

[7. 查询 147](#_Toc9614)

[7.1. 基本选择器：同CSS 147](#_Toc13312)

[7.2. 层次选择器：同CSS 147](#_Toc22202)

[7.3. 子元素过滤：同CSS 147](#_Toc10182)

[7.4. 基本过滤(位置过滤)：jQuery新增 147](#_Toc24229)

[7.5. 属性过滤选择器：同CSS 148](#_Toc31473)

[7.6. 内容过滤选择器：jQuery新增： 148](#_Toc26246)

[7.7. 可见性过滤： 148](#_Toc6344)

[7.8. 表单元素选择器：jQuery新增 149](#_Toc16308)

[7.9. 状态过滤选择器：同CSS 149](#_Toc31691)

[8. 修改 149](#_Toc14118)

[8.1. 内容 149](#_Toc28581)

[8.2. 属性 150](#_Toc6691)

[8.2.1. HTML标准属性: 150](#_Toc9124)

[8.2.2. 状态属性 150](#_Toc7523)

[8.2.3. 自定义扩展属性 150](#_Toc6613)

[8.2.4. 鄙视：.attr vs.prop 151](#_Toc8737)

[8.3. 样式 152](#_Toc8258)

[8.3.1. 获取和修改 152](#_Toc28926)

[8.3.2. 用class操作样式 153](#_Toc11492)

[9. 按节点间关系查找 153](#_Toc151)

[9.1. 父子关系 153](#_Toc31292)

[9.2. 兄弟关系 153](#_Toc20035)

[10. 添加/删除/替换/克隆 154](#_Toc29114)

[10.1. 用html片段创建新元素 154](#_Toc16730)

[10.2. 将元素添加到DOM树上 154](#_Toc6748)

[10.3. 删除 154](#_Toc5602)

[10.4. 替换 154](#_Toc13600)

[10.5. 克隆 154](#_Toc29093)

[11. 事件绑定 155](#_Toc15944)

[11.1. jQuery共有几种事件绑定方式： 155](#_Toc7931)

[11.2. 页面加载后自动执行：2种 155](#_Toc384)

[11.2.1. 仅DOM内容加载后就提前执行： 155](#_Toc710)

[11.2.2. 等全部页面内容都加载完才执行 156](#_Toc8289)

[11.2.3. 鄙视：jQuery中$的原理 156](#_Toc20344)

[11.3. 鼠标事件 156](#_Toc9708)

[11.4. 模拟操作 157](#_Toc17687)

[12. 动画API 158](#_Toc22252)

[12.1. 简单动画 158](#_Toc3743)

[12.1.1. 显示隐藏 158](#_Toc11874)

[12.1.2. 上滑下滑 158](#_Toc3888)

[12.1.3. 淡入淡出 158](#_Toc11888)

[12.2. 万能动画: 对任意css属性应用动画效果: 160](#_Toc17895)

[12.3. 排队和并发 160](#_Toc12623)

[12.4. 停止动画：$().stop() 160](#_Toc25947)

[12.4.1. 动画结束后自动执行 160](#_Toc862)

[13. 类数组对象操作 161](#_Toc25611)

[14. 添加自定义API 162](#_Toc5713)

[15. \*\*\*\*\*封装自定义插件 163](#_Toc13641)

[15.1. 官方插件（jQuery UI） 163](#_Toc5726)

[15.2. 第三方插件 164](#_Toc14913)

[15.3. 自定义插件 164](#_Toc31235)

[15.3.1. 定义jQuery UI风格的插件 164](#_Toc23403)

[15.3.1.1. 定义插件样式文件.css 164](#_Toc11358)

[15.3.1.2. 为jQuery添加插件API 164](#_Toc5233)

[15.3.2. 定义Bootstrap风格的插件 164](#_Toc23586)

[16. 跨域 165](#_Toc4926)

[16.1. $.ajax() 165](#_Toc22923)

[16.2. \*\*\*\*\*跨域访问 165](#_Toc16690)

[16.2.1. 服务端允许跨域——CORS跨域 166](#_Toc8771)

[16.2.2. 客户端跨域：JSONP——JSON With Padding 167](#_Toc18810)

[16.2.2.1. 使用script src发送请求 168](#_Toc5530)

[16.2.2.2. 将数据显示方式变活 169](#_Toc1462)

[16.2.2.3. 将封装的函数名写活 169](#_Toc27613)

[16.2.2.4. 动态创建script元素 170](#_Toc14885)

[16.2.2.5. 总结 170](#_Toc8324)

[第四部分 框架（VUE） 171](#_Toc16246)

[第一章 框架概述 171](#_Toc23394)

[1. 原生API vs 函数库 vs 框架 171](#_Toc4229)

[第二章 VUE概述 171](#_Toc15575)

[1. 基于MVVM 172](#_Toc24926)

[1.1. 下载并在网页中引入vue.js ——前三天 172](#_Toc4417)

[1.2. 安装VUE脚手架工具 —— 最后两天 172](#_Toc7011)

[第三章 MVVM框架原理 173](#_Toc6412)

[1. 传统前端三大文件 173](#_Toc20377)

[2. 框架将前端重新进行划分 173](#_Toc21063)

[3. ViewModel的原理 173](#_Toc30827)

[4. new Vue()内包含两大子系统 174](#_Toc22506)

[4.1. 响应系统 174](#_Toc28251)

[4.2. 虚拟DOM树 174](#_Toc19500)

[4.3. 为什么快 174](#_Toc20354)

[4.4. 鄙视：为什么框架使用虚拟DOM树 174](#_Toc11924)

[第四章 绑定语法 174](#_Toc22478)

[第五章 指令（directive） 175](#_Toc17338)

[1. v-bind 175](#_Toc22617)

[2. v-for 175](#_Toc9158)

[3. v-if 176](#_Toc11856)

[4. v-show 177](#_Toc28209)

[4.1. 鄙视：VUE中的 v-show 和 v-if的区别 178](#_Toc24130)

[5. v-on 178](#_Toc21596)

[5.1. 如何获得事件对象e： 179](#_Toc24283)

[5.2. 阻止默认行为和取消冒泡： 180](#_Toc31211)

[5.3. 键盘事件： 180](#_Toc13891)

[6. v-html和v-text： 181](#_Toc24852)

[7. v-cloak 182](#_Toc26564)

[8. v-pre 182](#_Toc2620)

[9. v-once 182](#_Toc23401)

[10. 双向绑定 v-model 183](#_Toc27028)

[10.1. text, textarea 184](#_Toc30633)

[10.2. radio 185](#_Toc10908)

[10.3. select 185](#_Toc9567)

[11. 绑定style和class 186](#_Toc14511)

[11.1. 绑定style 186](#_Toc30885)

[11.2. 绑定class 188](#_Toc32008)

[12. computed计算属性 190](#_Toc19775)

[12.1. computed vs methods 191](#_Toc1508)

[13. 自定义指令 192](#_Toc23493)

[13.1. 钩子函数（可选） 192](#_Toc10149)

[13.2. 创建自定义指令 192](#_Toc13480)

[13.3. 使用指令 192](#_Toc28711)

[13.4. 创建new Vue实例对象，监视HTML元素 192](#_Toc14528)

[第六章 过滤器(filter) 193](#_Toc17831)

[第七章 Axios 195](#_Toc5605)

[第八章 组件 196](#_Toc24433)

[1. 定义组件模板 196](#_Toc27997)

[2. 创建组件 197](#_Toc28226)

[3. 在页面中使用组件 197](#_Toc27722)

[4. 组件属性 199](#_Toc11773)

[4.1. Data 199](#_Toc15279)

[4.2. Props 199](#_Toc18341)

[4.3. Computed 199](#_Toc15445)

[4.4. Methods 200](#_Toc11647)

[4.5. Watch 200](#_Toc24959)

[第九章 组件化开发 200](#_Toc3958)

[第十章 组件间传递数据（重点） 203](#_Toc18010)

[1. 父->子 203](#_Toc17221)

[1.1. 子组件要定义一个外部可访问到的属性 203](#_Toc31726)

[1.2. 父组件的模板中 203](#_Toc18214)

[2. 子->父 205](#_Toc15256)

[2.1. 父组件 205](#_Toc31458)

[2.2. 子组件 205](#_Toc19030)

[3. 兄弟间传参 209](#_Toc19056)

[第十一章 组件生命周期 215](#_Toc31596)

[1. 4个阶段 215](#_Toc7819)

[2. 8个生命周期钩子函数 216](#_Toc29990)

[第十二章 SPA(Single Page Application) 216](#_Toc19167)

[第十三章 Vue脚手架 217](#_Toc12488)

[1. Vue脚手架工具 217](#_Toc11969)

[2. 安装/使用Vue脚手 217](#_Toc14061)

[2.1. 安装脚手架命令行工具 217](#_Toc9361)

[2.2. 用脚手架命令行生成脚手架项目代码 217](#_Toc25568)

[3. 运行测试 218](#_Toc32049)

[4. 脚手架项目结构 219](#_Toc7092)

[5. 打包发布 219](#_Toc28730)

[6. Vue-router 219](#_Toc21542)

[6.1. 根组件<template>中 219](#_Toc28376)

[6.2. 在views文件夹下，定义多个组件.vue： 219](#_Toc19089)

[6.3. router.js中编写路由字典: 220](#_Toc30315)

[6.4. 浏览时 220](#_Toc11897)

[6.5. 嵌套路由 220](#_Toc10602)

[6.5.1. 浏览时 220](#_Toc3449)

[6.5.2. 路由跳转 221](#_Toc26645)

[6.5.3. 路由传参 221](#_Toc29593)

# JavaScriptCore

## 正则表达式

### 概述

正则表达式：(Regular Expression)

推荐网站：regexper.com

#### 定义

规定字符串中字符出现规律的规则

#### 何时

1. 按规则模糊查找多种关键词时
2. 用规则验证用户输入的格式时

### 使用方式

#### 明文

最简单的规则就是关键词的原文

#### 字符集

1. 定义：规定一位字符上所有备选字符列表的集合
2. 适用情况：一位字符有多种备选时
3. 语法：[备选字符列表]

注意：一个字符集只能匹配一位字符

1. 简写：如果备选字符列表中部分字符是连续的，可用-省略中间字符
2. 取值：

* 一位数字：[0-9]
* 一位小写字母：[a-z]
* 一位大写字母：[A-Z]
* 一位字母：[A-Za-z]
* 一位字母或数字：[0-9A-Za-z]
* 一位汉字：[\u4e00-\u9fa5]

例：

比如：车牌号：一个汉字大写字母点5位大写字母或数字

[\u4e00-\u9fa5][A-Z]·[0-9A-Z][0-9A-Z][0-9A-Z][0-9A-Z][0-9A-Z]

1. 除了：[^47]除了4或7

#### 预定义字符集

1. 定义：对常用字符集的简写
2. 取值

* 一位数字：\d[0-9]
* 一位字母数字或下划线：\w[0-9A-Za-z\_]
* 一位空字符：\s包括：空格、Tab...
* 通配符：**.**

注意：字符集无法灵活定义出现的次数

#### 量词

1. 定义：定义一个字符集出现次数的规则
2. 适用情况：只要定义字符集出现的次数
3. 语法：量词一般紧跟在要修饰的字符集之后，默认仅修饰相邻的前一个字符集。

##### 有明确数量边界的

1. {n,m} 至少n次，最多m次
2. {n,} 至少n次，多了不限
3. {n} 必须n次，不能多也不能少

##### 没有明确数量边界的

1. \* 可有可无，多个不限
2. ? 可有可无，最多1次
3. + 至少一次，多了不限

#### 选择和分组

##### 选择

1. 定义：多个规则，选其一匹配即可！
2. 语法：规则1|规则2

注意：|优先级最低

例：微信weixin w x Wei Xin wechat

(微|w(ei)?)\s\*(信|x(in)?)

##### 分组

1. 定义：将多个规则合并为一组
2. 适用情况：

希望一个量词可修饰之前多个规则时

希望分段获取或处理字符串中部分子内容时

1. 语法：(多个规则)

例：身份证号

15位数字2位数字1位数字或X或x

后三位整体可有可无，最多1次

\d{15}(\d{2}[0-9Xx])?

例：手机号

+86或0086 可有可无，最多1次

空字符 可有可无，多了不限

1

3~8

9位数字

(\+86|0086)?\s\*1[3-8]\d{9}

例：电子邮件

用户名：字母、数字或下划线1位以上

@

域名：字母、数字或-1位以上

.字母、数字或-整体1位以上

\w+@[0-9A-Za-z-]+(\.[0-9A-Za-z-]+)+

#### 匹配特殊位置

1. 字符串开头：^

例：仅匹配开头的空字符：^\s+

1. 字符串结尾：$

例：仅匹配结尾的空字符：\s+$

适用情况：只要仅匹配开头或结尾的内容时

例：同时匹配开头或结尾的空字符：^\s+|\s+$

1. 单词边界：\b

取值：字符串开头和结尾，空字符，标点符号……

例：匹配每个单词首字母？\b[a-z]

#### 密码强度

6~8位字母，数字的组合，至少包含一个大写字母和一位数字

例：^(?![a-z0-9]+$)(?![A-Za-z]+$)[A-Za-z0-9]{6,8}$

### 正则表达式汇总

|  |  |  |  |
| --- | --- | --- | --- |
| **分类** | **语法** | **含义** | **示例** |
| **普通字符** | 明文 | 匹配指定集合内的任一个字符 |  |
| **字符集** | [abc] | 匹配指定集合内的任一个字符 | /[123]/ |
| [^abc] | 匹配不在指定集合内的任意字符 | /[^5623]/ |
| [0-9] | 匹配任意一个数字 | /[0-9]/ |
| [a-z] | 匹配任意一个小写字符 | /[a-z]/ |
| [A-Z] | 匹配任意一个大写字符 | /[A-Z]/ |
| [A-Za-z] | 匹配匹配A-z内的所有字符  注：大小写之间有[\]^\_`所间隔  ASCII表为64-90，97-122 | /[A-Za-z]/ |
| [\u4e00-\u9fa5] | 匹配任意一个汉字  一~龥（yù） | [\u4e00-\u9fa5] |
| **预定义字符集** | \d | 匹配一个数字 | /\d/等价于/[0-9]/ |
| \D | 匹配一个非数字 | /\D/等价于/[^0-9]/ |
| \w | 匹配一个数字/字母/下划线 | /\w/等价于/[0-9a-zA-Z]/ |
| \W | 匹配一个非数字/字母/下划线 | /\W/等价于/[^0-9a-zA-Z]/ |
| \s | 匹配一个空白字符 | /\s/等价于/[\n\r\t\v\f]/ |
| \S | 匹配一个非空白字符 | /\S/等价于/[^\n\r\t\v\f]/ |
| . | 匹配除了回车和换行外的任何单个字符 | /./等价于/[^\n\r]/ |
| **没有明确数量边界** | ? | 可有可无，最多一次 |  |
| \* | 可有可无，多了不限 |  |
| + | 至少一次，多了不限 |  |
| **有明确数量边界** | {n} | 必须n次，不能多也不能少 |  |
| {n,m} | 至少n次，最多m次 |  |
| {n,} | 至少n次，多了不限 |  |
| **选择** | 规则1|规则2 | 多个规则，选其一匹配即可 |  |
| **分组** | (多个规则) | 将多个规则合并为一组 |  |
| **限制字符** | [^] | 除了 | [^abc] |
| **匹配位置** | ^ | 字符串开头 | /^a/ |
| $ | 字符串结尾 | /a$/ |
| \b | 单词边界  包括：^、$、空格、标点符合 | /\bhis\b/ |

## String

### 概念

多个字符组成的只读字符数组

#### 与数组相比

|  |  |
| --- | --- |
| 相同 | 不同 |
| 下标、.length、for遍历、slice() | 两者类型不同，API不通用 |

### API

#### 强调

所有String API都无权修改原字符串，只能**返回新字符串**

#### 大小写转换

只要不区分大小写时，都要先转换为一致的大小写，再比较

str.toUpperCase()

str.toLowerCase()

#### 获取指定位置字符

1. var char=str.charAt(i) str[i]
2. php中

substr($string,$starti,1)

1. var unicode=str.charCodeAt(i)

* 简写：省略i,默认为0
* var char=String.fromCharCode(unicode)

#### 选取子字符串

1. var subStr=str.slice(starti,endi+1)

简写：同数组的slice

1. PHP中

substr($string,$starti,n)

1. str.substring(starti,endi+1)

* 简写：同数组的slice
* 不支持负数参数：变通：str.length-n

1. str.substr(starti,n)

* 不考虑含头不含尾
* str.substring(starti,starti+n)

#### 查找关键词

##### 查找一个固定的关键词出现的位置

###### 语法

var i=str.indexOf("关键词",starti)

1. 在str中从starti位置开始查找下一个关键词的位置
2. 返回值

* 返回找到的关键词所在的位置
* 如果找不到返回-1

1. 固定套路

查找所有关键词出现的位置

var i=-1;

while((i=str.indexOf("关键词",i+1))!=-1){

i//本次找到的关键词位置

}

1. 简写

省略fromi，默认为0

###### 优缺点

优：可以指定开始位置，可以找所有

缺：不支持正则，一次只能找一种关键词

###### 专门查找最后一个关键词的位置

var i=str.lastIndexOf("关键词")

###### php中

1. indexOf

* strpos
* stripos

1. lastIndexOf

* strrpos
* strripos

##### 判断是否包含符合规则的关键词

###### 语法

var i=str.search(/正则表达式/i)

1. 如果返回-1，说明不包含，如果返回不是-1，说明包含
2. 忽略大小写

/正则表达式/i

1. 永远只找第一个关键词的位置
2. 强调

不支持g

###### 优缺点

优：支持正则

缺：不能设置开始查找位置，只能找第一个，不能找所有

只能返回位置，不能返回关键词内容

##### 获取所有关键词的内容

###### 语法

var kwards=str.match(/正则表达式/ig)

强调：省略g，只找第1个

加g，才找所有

###### 返回包含所有关键词的数组

如果没找到，返回null

如果一个函数可能返回null，都要先验证，再使用结果

###### 优缺点

优：获得所有关键词的内容

缺：无法返回每个关键词的位置

###### PHP中

1. search

preg\_match

1. match

preg\_match\_all

1. 强调：

* PHP中正则不支持g

很多函数默认就是查找所有和替换所有

* PHP中正则依然要用i忽略大小写

##### 即获得每个关键词的内容，又获得每个关键词的位置

regexp.exec(str)

见4.2。

#### 替换

str=str.replace("关键词","替换值")

##### 简单替换

1. 将所有关键词替换为统一的新值
2. str=str.replace(/正则/ig,"新值")

将str中所有符合正则要求的关键词替换为统一的“新值”

如果替换一个固定的关键词，则第一个参数可以不用正则，只要写死关键词即可

1. 问题：无法根据不同的关键词，选择不同的值替换

##### 高级替换

可根据每次找到的关键词不同，动态替换为不同的新值

str=str.replace(/正则/ig,function(kword){

//每找到一个关键词就自动调用一次

//每次调用时，kword参数都自动获得本次找到的关键词

return替换后的新值;

})

鄙视：将每个单词首字母转为大写！

var str="you can you up!"

str=str.replace(

/\b[a-z]/ig,

function(kword){

return kword.toUpperCase();

}

)

console.log(str);

##### 衍生操作

###### 删除

替换为空字符串

###### 格式化

1. 用正则对原始字符串分组

var reg=/(\d{4})(\d{2})(\d{2})/

正则中每个分组都会自动获得一个分组序号，从1开始

1. 在replace的替换值中使用$n，重新拼接新的格式

birth.replace(reg,"$1年$2月$3日")

#### 切割

##### 简单切割

var subs=str.split("分隔符")

##### 复杂切割

var subs=str.split(/正则表达式/)

##### 返回值

多段子字符串组成的数组

切割后的结果中不包含分隔符

##### 固定套路

将字符串打散为字符数组

var chars=str.split("")

定义和用法

str\_split() 函数把字符串分割到数组中。

###### 语法

str\_split(string,length)

取值：

string 必需。规定要分割的字符串。

length 可选。规定每个数组元素的长度。默认是 1。

返回值：

如果 length 小于 1，则 str\_split() 函数将返回 FALSE。

如果 length 大于字符串的长度，则整个字符串将作为数组的唯一元素返回。

PHP 版本：5+

### split与slice区别

#### split

##### 定义

split() 方法用于把一个字符串分割成字符串数组。

##### 语法

stringObject.split(separator,howmany)

取值：

separator 必需。字符串或正则表达式，从该参数指定的地方分割 stringObject。

howmany 可选。该参数可指定返回的数组的最大长度。如果设置了该参数，返回的子串不会多于这个参数指定的数组。如果没有设置该参数，整个字符串都会被分割，不考虑它的长度。

返回值：

一个字符串数组。该数组是通过在 separator 指定的边界处将字符串 stringObject 分割成子串创建的。返回的数组中的字串不包括 separator 自身。

但是，如果 separator 是包含子表达式的正则表达式，那么返回的数组中包括与这些子表达式匹配的字串（但不包括与整个正则表达式匹配的文本）。

##### 提示和注释

注释：如果把空字符串 ("") 用作 separator，那么 stringObject 中的每个字符之间都会被分割。

注释：String.split() 执行的操作与 Array.join 执行的操作是相反的。

#### slice

##### 定义

slice() 方法可从已有的数组中返回选定的元素。

##### 语法

arrayObject.slice(start,end)

取值：

start 必需。规定从何处开始选取。如果是负数，那么它规定从数组尾部开始算起的位置。也就是说，-1 指最后一个元素，-2 指倒数第二个元素，以此类推。

end 可选。规定从何处结束选取。该参数是数组片断结束处的数组下标。如果没有指定该参数，那么切分的数组包含从 start 到数组结束的所有元素。如果这个参数是负数，那么它规定的是从数组尾部开始算起的元素。

返回值：

返回一个新的数组，包含从 start 到 end （不包括该元素）的 arrayObject 中的元素。

说明：

请注意，该方法并不会修改数组，而是返回一个子数组。如果想删除数组中的一段元素，应该使用方法 Array.splice()。

##### 提示和注释

注释：您可使用负值从数组的尾部选取元素。

注释：如果 end 未被规定，那么 slice() 方法会选取从 start 到数组结尾的所有元素。

## RegExp

### 定义

专门封装一条正则表达式，并提供使用正则表达式执行验证和查找的API的对象

### 适用情况

只要在js中使用正则表达式，都用正则表达式对象

1. 使用正则表达式验证字符串格式
2. 即查找关键词内容，又查找关键词位置

### 创建

#### 直接量

用var reg=/正则/ig

正则表达式是固定的！

字符冲突：/->\/

#### 用new

用new：var reg=new RegExp("正则","ig")

正则表达式不是固定的，需要运行时动态生成

字符冲突：\->\\\"\'

new RegExp("\\d{6}")

### API

#### 验证

1. var bool=reg.test(str)

用reg验证str是否符合要求

返回值：bool

1. 问题

正则默认只要在str中找到匹配的内容，就返回true，不要求完全匹配！

1. 解决

今后返回验证，必须前加^后加$

表示从头到尾必须完整匹配

#### 查找关键词

var arr=reg.exec(str)

##### 即查找内容又查找位置

既查找每个关键词内容，又查找每个关键词位置

在str中查找下一个满足reg要求的关键词

##### 返回值

1. arr:[0:"完整关键词",1:$1,2:$2,...,index:本次找到关键词的位置]
2. reg.lastIndex

下次开始位置

1. 如果没找到，返回null

都要先判断不是null，再使用

##### exec做三件事

将本次找到的关键词，放入数组第0个元素，将每个分组的子内容放入后续元素

修改数组的index属性，记录本次找到关键词的位置

修改reg.lastIndex属性=index+关键词的长度

##### 固定套路

1. 找全部

var arr=null;

while((arr=reg.exec(str))!=null){

arr[0]//完整关键词

arr[n]//第n个分组的子内容

arr.index//本次找到关键词的位置

reg.lastIndex//下次开始查找的位置

}

1. 简写：如果只获得某个分组的子内容

while(reg.exec(str)!=null){

RegExp.$n//第n个分组的子内容

}

## Function

### 什么是

function是封装一段可重用的代码段的对象/程序结构，再起一个名字

函数名其实仅是引用函数对象的一个普通变量。

### 为什么

代码重用

### 何时

只要一项任务，可能被反复使用，都要定义为函数，反复使用函数

### 创建

#### 创建的方式（3种）

##### 声明

1. function函数名(参数列表){函数体;return返回值}
2. 会被声明提前——hoist

###### hoist（声明提前）

什么是

在开始执行程序前

引擎会首先查找var声明的变量和function声明的函数

将其提前到当前作用域的顶部，集中创建；赋值留在原地。

鄙视时

凡是看到先使用，后声明，都是在声明提前

先改为声明提前之后的程序，再判断输出

解决问题

1. 所有变量和函数的声明都放在当前作用域的顶部
2. ES6

可用let代替var

要求，当前作用域中let a之前不允许出现未声明的a

1. 函数

var函数名=function(参数列表){函数体;return返回值}

##### 直接量

var函数名=function(参数列表){

函数体;return返回值

}

不会被声明提前

##### 用new

var函数名=new Function(

"参数名1",

"参数名2",

...,

"函数体;

return返回值"

)

##### 三种定义方式的对比

1. 仅以晟敏方式定义的方法，湖北声明提前。
2. 代码动态执行：

以声明的方式和直接量方式定义的方法，方法体固定。无法再运行时动态执行。

以创建对象方式定义的方法，方法体是字符串。可以在运行时动态创建，修改，并执行字符串格式的方法。

1. 效率：

以创建对象方式定义的方法，因为方法体是字符串，需要再次解析。所以执行效率低。

其他两种方法定义方式不存在效率问题。

#### 参数

1. 接收函数执行时必须的数据的变量
2. 为什么

可让函数更灵活

1. 何时

只要函数必须某些数据才能正常执行

#### 返回值

1. 函数的执行结果
2. 何时

如果调用者需要获得函数的返回值

### 调用

1. 让引擎按照函数的步骤清单，执行任务
2. 强调

函数不调用不执行

只有调用才执行

### 重载(overload)

#### 什么是

相同函数名，不同参数列表的多个函数

在调用时，根据传入参数的不同，自动选择匹配的函数执行

#### 为什么

减少API的数量，减轻调用者的负担

#### 何时

只要一项任务，需要根据不同的参数，执行不同的操作时

#### 如何

##### 问题

js语法不支持重载

js不允许多个同名函数同时存在；若多个同名函数出现，仅会执行最后一个。

##### 解决

每一个函数内，都有一个arguments对象专门接入所有传入函数的参数值的类数组对象。

根据arguments的元素内容或元素个数，判断执行不同的操作。

##### arguments

函数调用时，是自动创建

自动接收所有传入函数的参数值的

类数组对象，长的像数组的对象

###### 类数组对象vs数组

|  |  |  |
| --- | --- | --- |
|  | 类数组 | 数组 |
| 相同 | 下标、length、for/for of遍历 | |
| 不同 | Object类型  类数组对象无法使用数组的API | Array类型 |

##### 如何实现

1. 只定义一个函数，不定义形参列表；
2. 在函数内，通过判断arguments对象的内容，决定执行何种操作。

##### 延伸

###### caller

返回一个对函数的引用，该函数调用了当前函数。

functionName.caller

functionName 对象是所执行函数的名称。

说明：对于函数来说，caller 属性只有在函数执行时才有定义。假如函数是由顶层调用的，那么 caller 包含的就是 null 。假如在字符串上下文中使用 caller 属性，那么结果和 functionName.toString 相同，也就是说，显示的是函数的反编译文本。

###### callee

返回正被执行的 Function 对象，也就是所指定的 Function 对象的正文。

[function.]arguments.callee

可选项 function 参数是当前正在执行的 Function 对象的名称。

说明：callee 属性的初始值就是正被执行的 Function 对象。

callee 属性是 arguments 对象的一个成员，他表示对函数对象本身的引用，这有利于匿名函数的递归或确保函数的封装性，例如下边示例的递归计算1到n的自然数之和。而该属性仅当相关函数正在执行时才可用。更有需要注意的是callee拥有length属性，这个属性有时候用于验证还是比较好的。arguments.length是实参度，arguments.callee.length是形参长度，由此能够判断调用时形参长度是否和实参长度一致。

//callee能够打印其本身

function calleeDemo() {

alert(arguments.callee);

}

调用时：alert(sum(100));

其中函数内部包含了对sum自身的引用，函数名仅仅是个变量名，在函数内部调用sum即相当于调用一个全局变量，不能很好的体现出是调用自身，这时使用callee会是个比较好的方法。

###### 打针的三种方式apply、bind、call

apply and call

他们的作用都是将函数绑定到另外一个对象上去运行，两者仅在定义参数方式有所区分：

apply(thisArg,argArray);

call(thisArg[,arg1,arg2…] ]);

即任何函数内部的this指针都会被赋值为thisArg，这可实现将函数作为另外一个对象的方法运行的目的

apply的说明：假如 argArray 不是个有效的数组或不是 arguments 对象，那么将导致一个 TypeError。

假如没有提供 argArray 和 thisArg任何一个参数，那么 Global 对象将被用作 thisArg，并且无法被传递任何参数。

call的说明：call 方法可将一个函数的对象上下文从初始的上下文改变为由thisArg指定的新对象。

假如没有提供thisArg参数，那么Global对象被用作thisArg

相关技巧：应用call和apply有一个技巧在里面，就是用call和apply应用另一个函数（类）以后，当前的

函数（类）就具备了另一个函数（类）的方法或是属性，这也能够称之为“继承”。看下面示例:

上面的例子能够看出，通过call之后，extend能够继承到base的方法和属性。

顺便提一下，在javascript框架prototype里就使用apply来创建一个定义类的模式，

解析：从代码看,该对象仅包含一个方法：Create，其返回一个函数，即类。但这也同时是类的构造函数，其中调用initialize，而这个方法是在类创建时定义的初始化函数。通过如此途径，就能够实现prototype中的类创建模式

### 匿名函数

#### 定义

定义函数时，不指定函数名的函数

使用后自动释放

好处：节约内存(凡是不起名的对象，用完立刻释放！)

#### 为什么

节约内存

划分临时作用域，避免全局变量

#### 何时

只要只是用一次的函数，都用匿名函数

只有需要重用的函数，才需要命名

##### 回调callback

自己定义了函数，自己不调用，传给别的函数去调用

例：arr.sort(function(a,b){return a-b})

ES6：arr.sort((a,b)=>a-b)

例：str.replace(/正则/,function(kword){return...})

ES6：str.replace(/reg/g,(kw,$1,$2,...)=>{return替换值})

##### 自调

定义函数后，立刻调用自己！

使用全局变量会长期驻留内存，造成全局污染，应禁止使用全局变量。

语法：

(function(参数列表){

函数体;

return返回值

})(参数值列表)

例：

var i = 0, tmp = [1,2,3];

for(i,len=tmp.length;i<len;i++) {

setTimeout(

(function(txt){

return function(){

console.log(txt)

}

})(tmp[i]),

1000\*i);

}

(function(a){

console.log(a); //123

})(123);

###### 自调函数的多种写法

1. 写法一：赋值给变量

var a = function(){return 1}();

相当于 var a = 1;

1. 写法二：返回给void

void function() {return 1}();

相当于void 1;

1. 写法三：作为参数传递

func(function() {return 1}());

相当于 func(1);

1. 写法四：()分组

(function() {return 1} )();

相当于1

(function() {return 1} ());

相当于(1)

1. 写法五：躲进if里

if (function(){return true}) {}

相当于 if( true ) {}

1. 写法六：一元操作符

!function(){return true}();

相当于 !true

~function(){return true}();

相当于 !true

-function(){return true}();

相当于 !true

+function(){return true}();

相当于 !true

1. 写法七：new 关键字

new function(){return } ();

相当于 {}

### 作用域和作用域链（重要）

#### 作用域(scope)

用法：一个变量的可用范围

本质：存储变量的一个对象

为什么：为了避免不同范围的变量间互相干扰！

##### 分类

###### 全局作用域

保存所有全局变量的对象——window

保存全局变量

优缺点：

1. 优点：可反复使用
2. 缺点：随处可用，容易造成全局污染

建议：尽量少用全局变量

###### 函数作用域

保存函数中局部变量的对象——AO

保存局部变量

包括：参数变量

函数内var出的变量

优缺点：

1. 优点：仅函数内使用

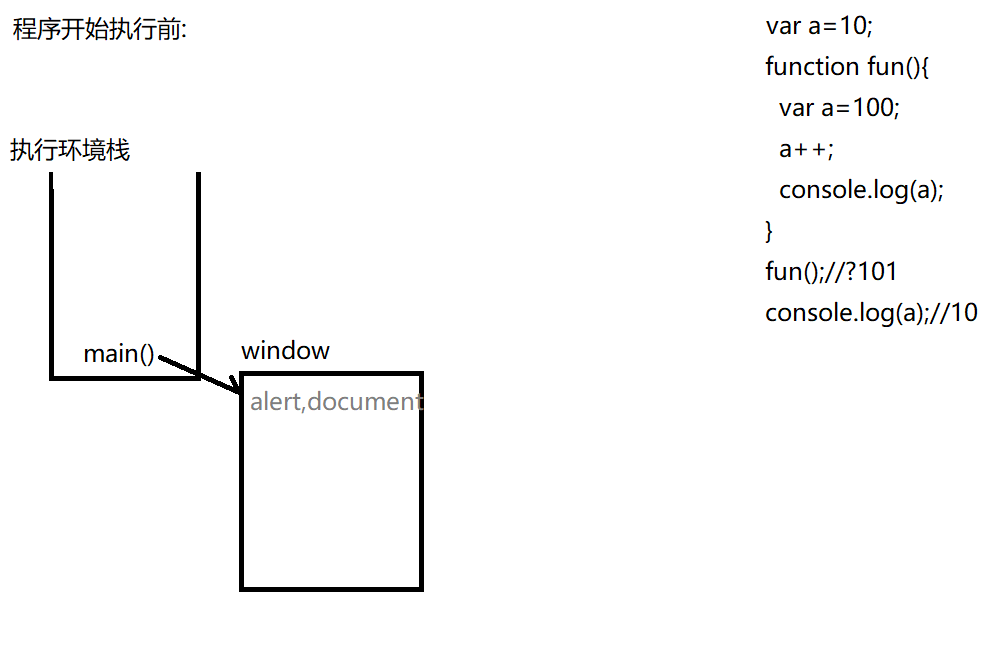
不会被污染

1. 缺点：不可重用

##### 函数的生命周期

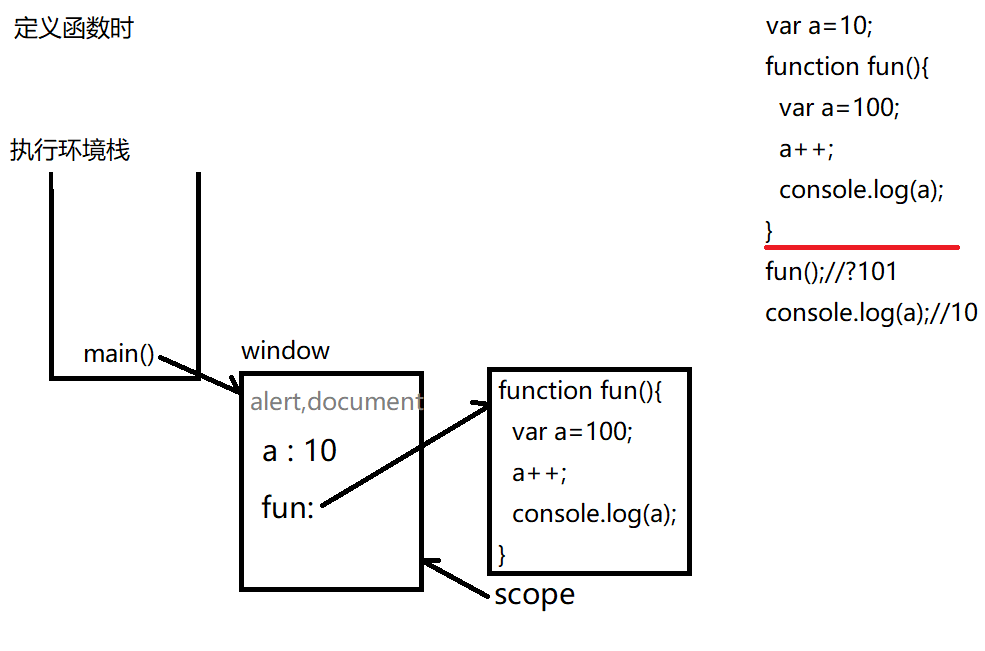
###### 程序开始执行前

1. 创建执行环境栈数组（ECS），用于保存将来要执行的每项任务的名称
2. 执行浏览器的主程序main()
3. 主程序创建全局作用域对象（window），所有全局变量都是保存在window对象中，main执行环境引用window



###### 定义函数时

1. 在window中创建函数名变量
2. 创建函数对象，封装函数定义
3. 函数名变量引用函数对象
4. 函数对象的scope属性，指向创建函数时所在的所用域(window)



###### 调用函数时

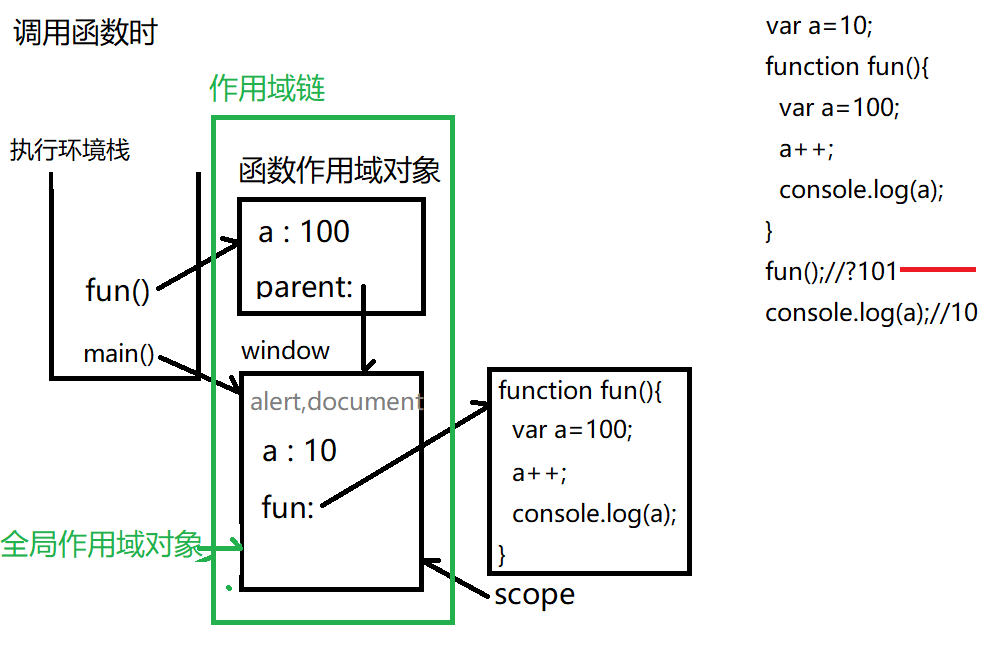
1. 在执行环境栈数组(ECS)中新增本次函数调用的记录（执行环境元素）
2. 为本次函数调用创建临时的函数作用域对象(AO)
3. 在函数作用域对象(AO)中保存所有局部变量

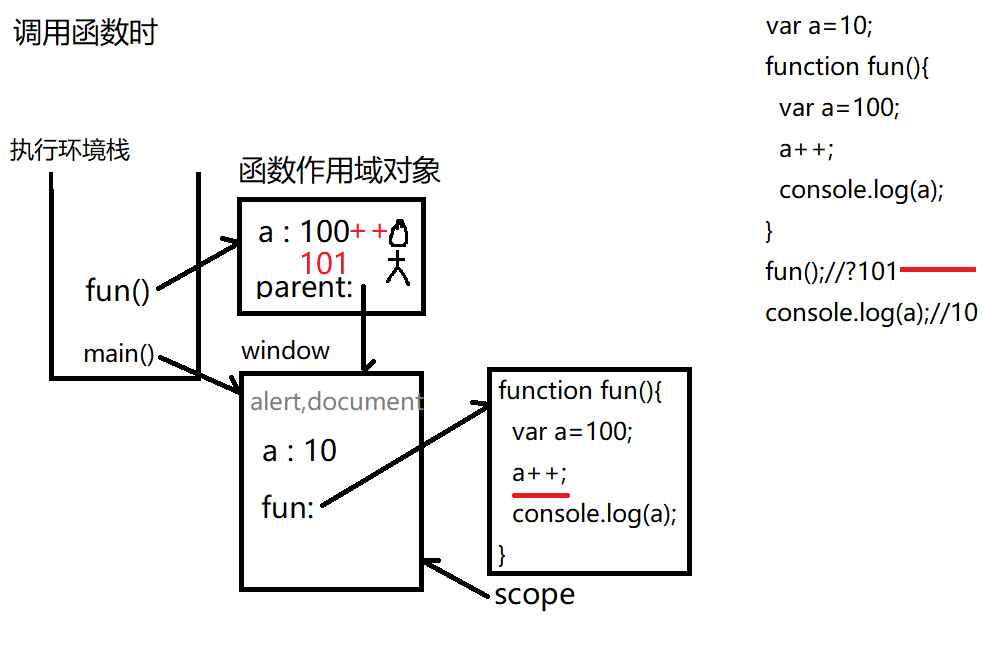
包含，形参变量与函数内用var声明的变量

1. 设置函数作用域对象(AO)的parent属性引用函数的scope属性指向的父级作用域对象
2. 函数的执行环境引用函数作用域对象(AO)
3. 变量的使用顺序：

执行时，优先使用函数作用域对象(AO)中的局部变量；

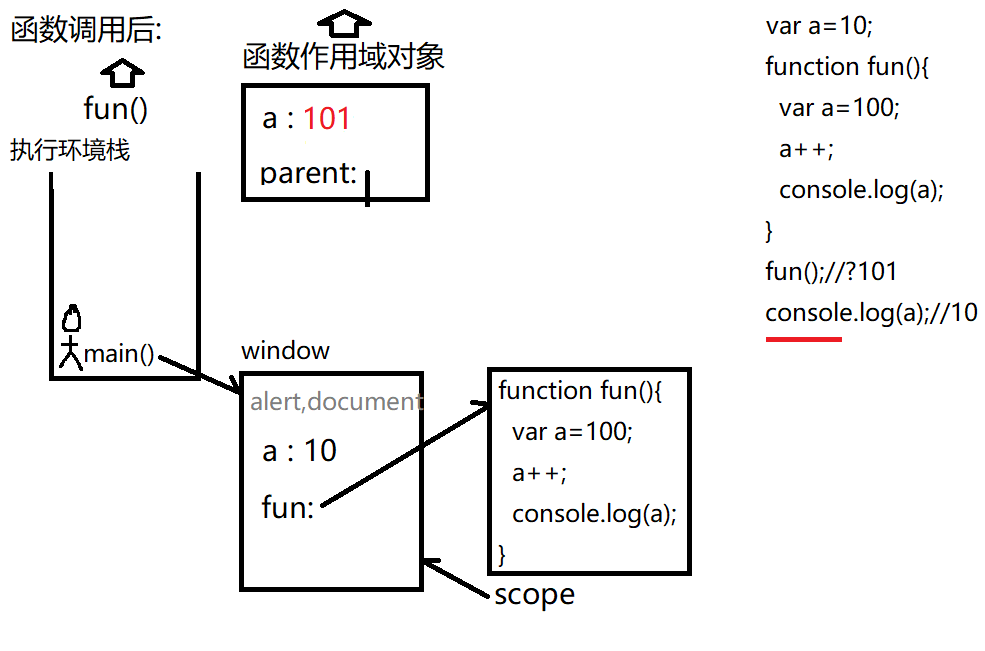
当函数作用域对象(AO)中没有时，才去父级作用域中找。





###### 函数调用后

1. 执行环境栈中本次调用的记录出栈
2. 导致函数作用域对象无人使用，被释放
3. 导致所有局部变量一同释放



##### 作用域链(scope chain)

由多级作用域逐级引用形成的链式结构

保存着所有变量

控制着所有变量的使用顺序（先在自身找，找不到在去父级找）

##### 函数中没有对象/this时

函数中，没有用任何对象/this就直接访问的变量，在作用域链中找。

### 闭包（closure）（重要）

既重用变量又保护变量不被污染的机制

全局变量，局部变量都有不可兼得的优缺点：

全局变量：优：可重用！

缺：极易被污染

局部变量：优：不会被污染

缺：不可重用

#### 适用情况

既重用变量又保护变量不被污染

只要希望给一个函数永久绑定一个专属的变量时

#### 语法

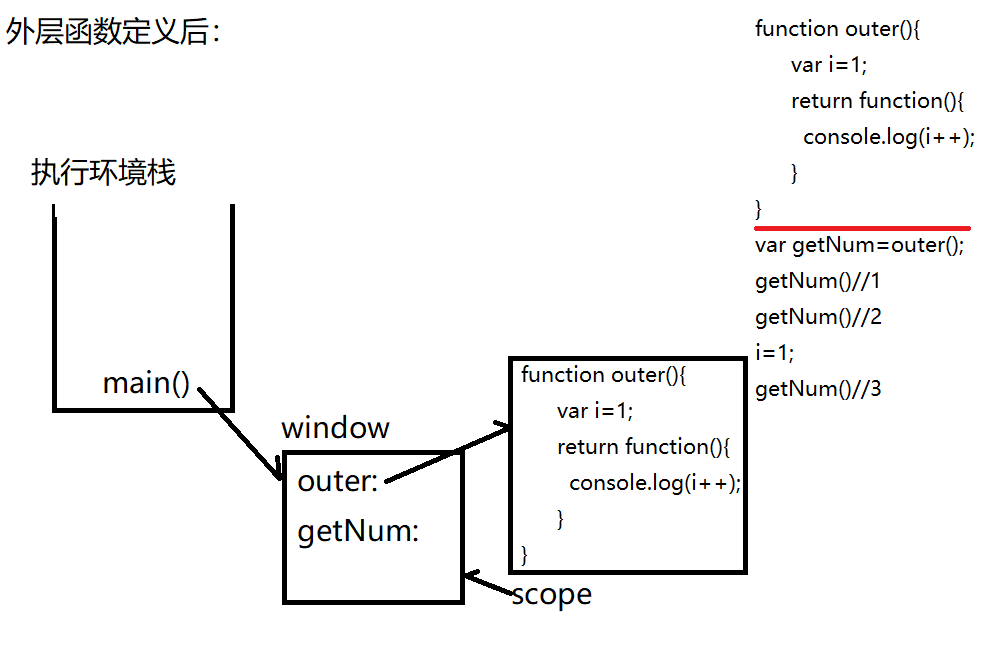
1. 定义外层函数包裹受保护的变量和内层函数
2. 外层函数返回内层函数的对象，3种：

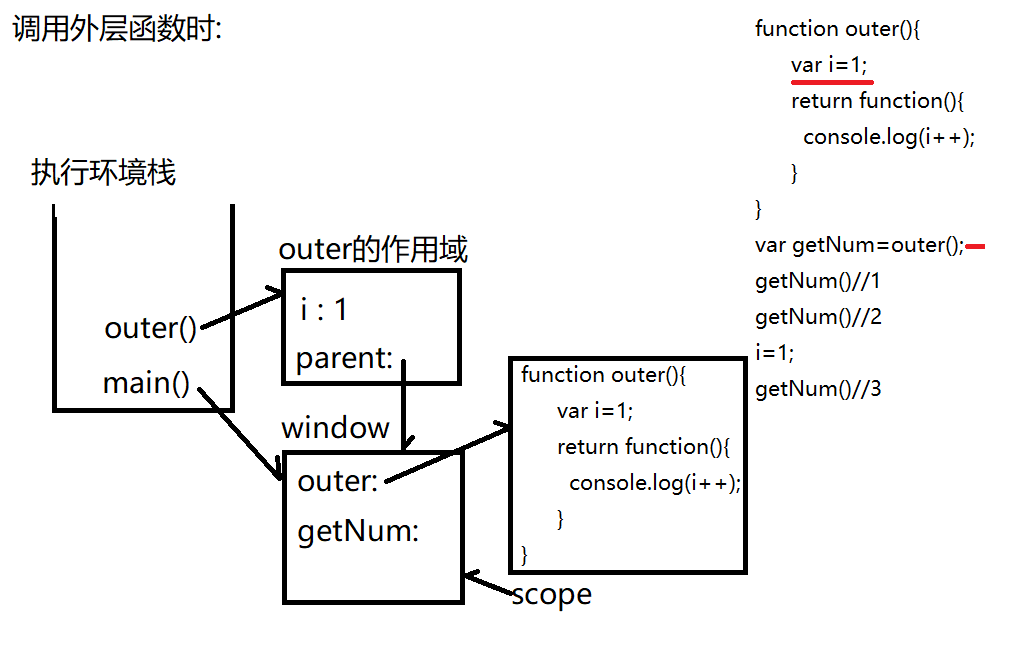
* return function(){...}
* 直接给全局变量赋值一个内部function
* 将内部函数保存在一个对象的属性或数组元素中

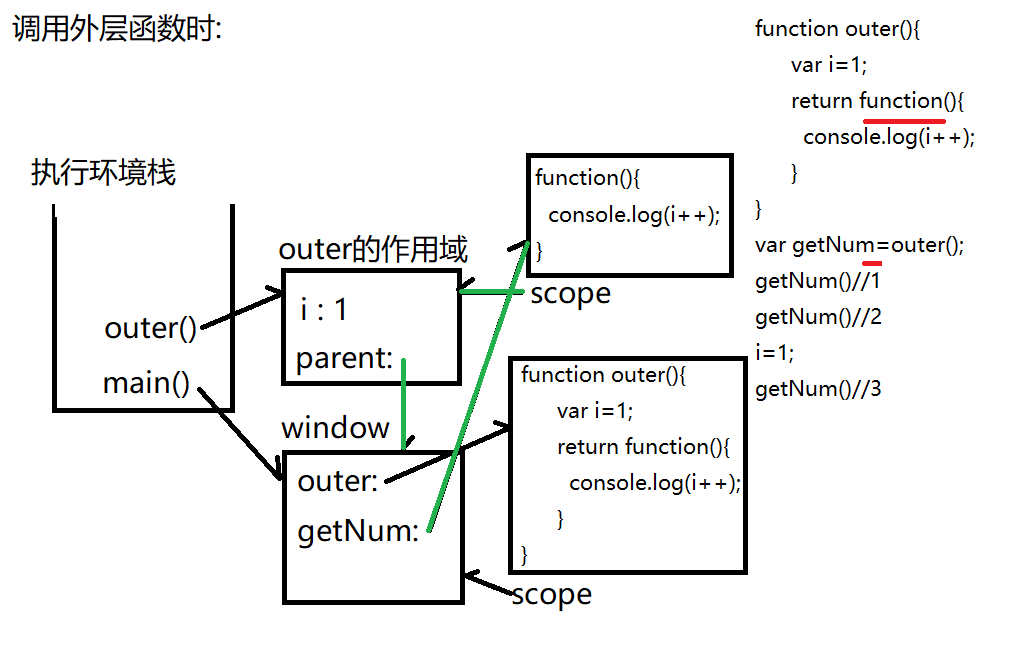
return[function,function,function]

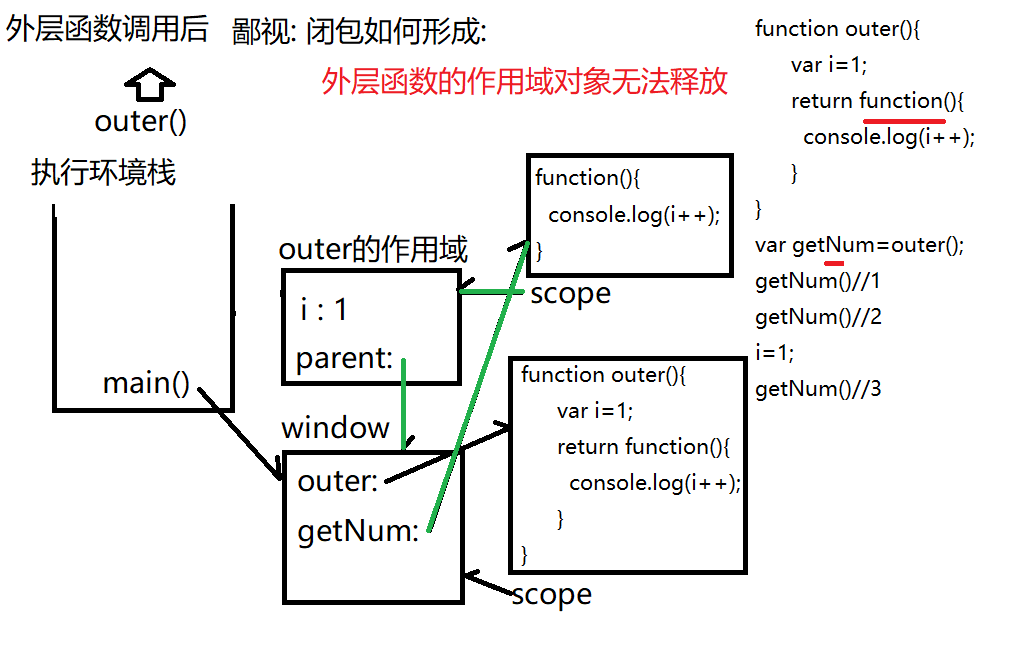
return{fun:function(){...}}

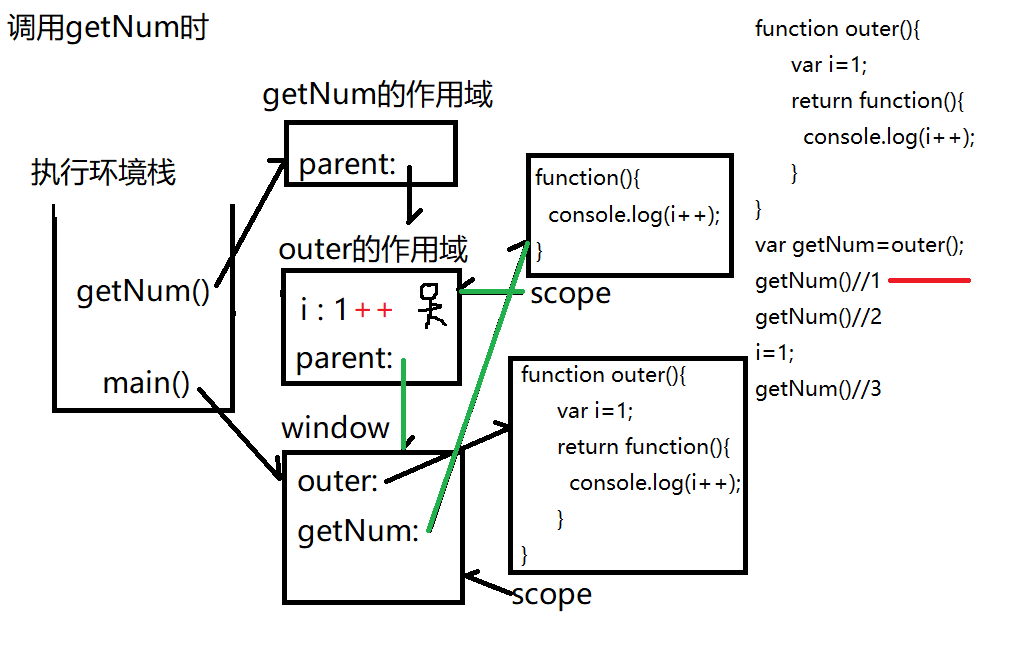
1. 调用外层函数获得内层函数对象保存在全局变量——形成了闭包！

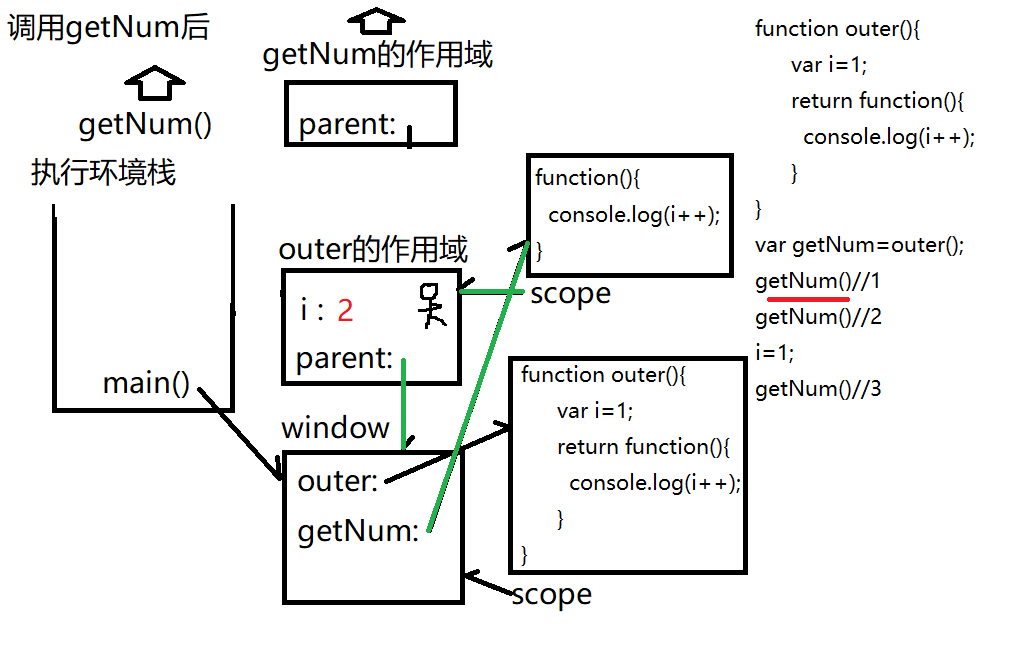


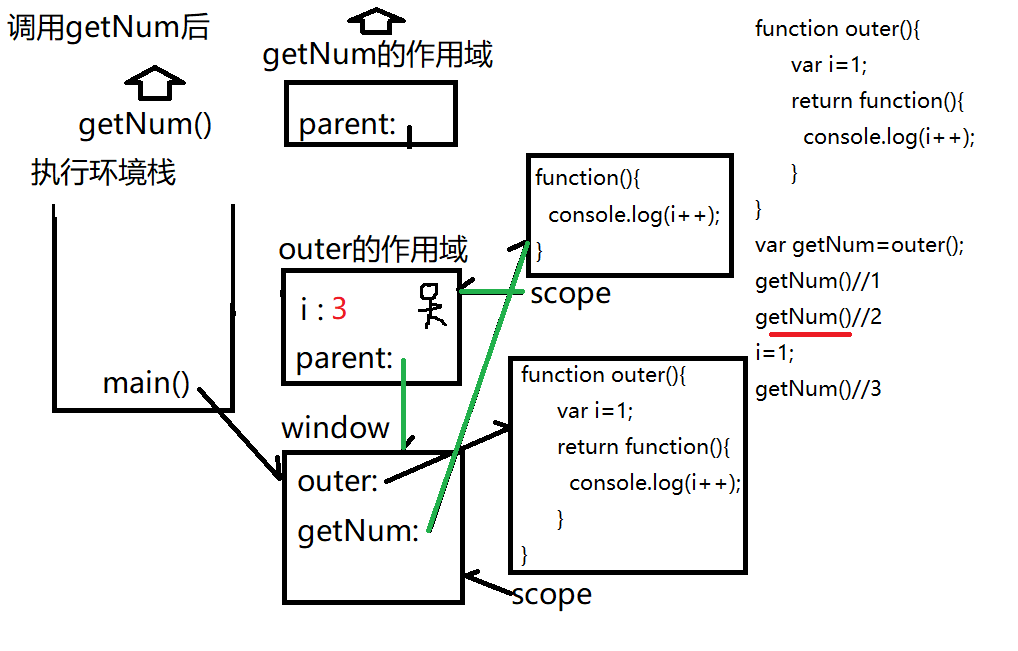


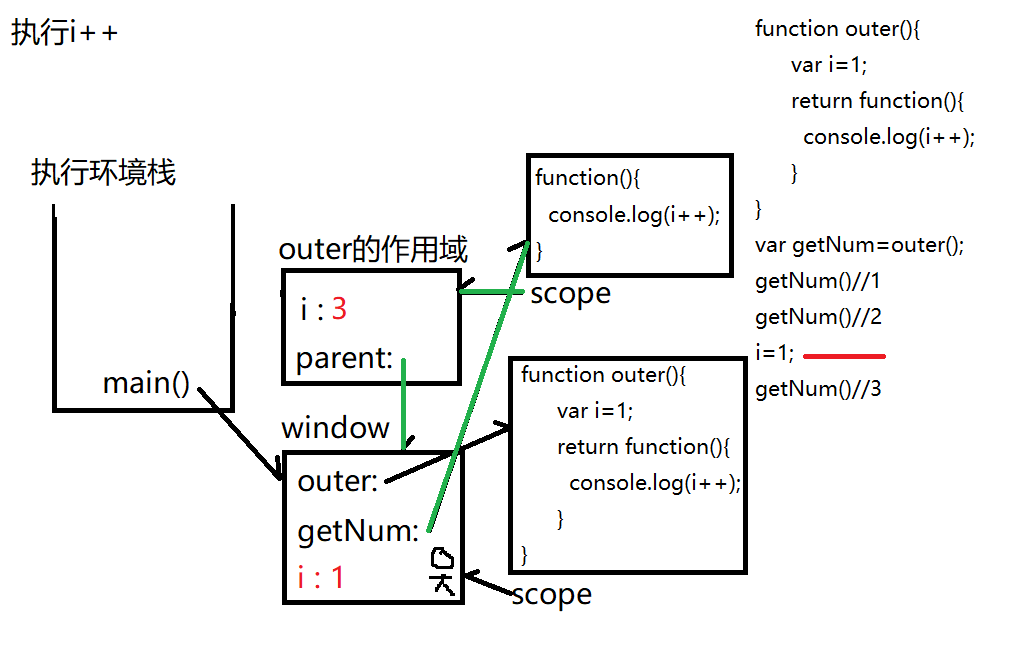


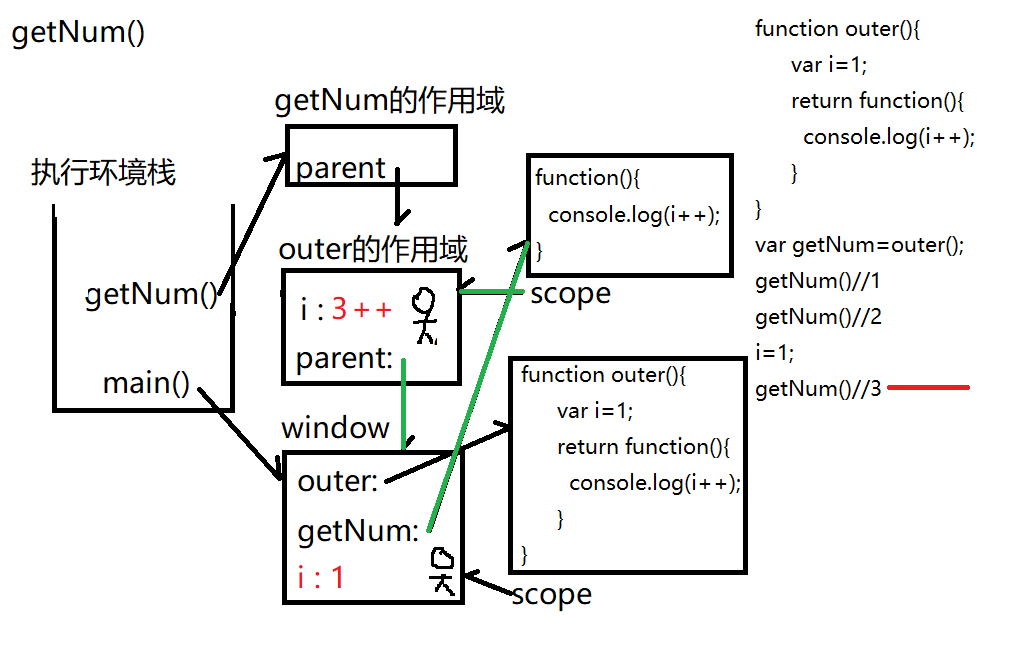












参考下例题看图

例：

//1.用外层函数包裹受保护的变量和内层函数

function outer(){

var i=1; //要保护的变量

//2.外层函数将内层函数对象返回到外部

return function(){

console.log(i++);

}

}

//3.调用外层函数，获得内层函数的对象

var getNum=outer();

//getNum:function(){console.log(i++);}

getNum(); //1

getNum(); //2

i=1;

getNum(); //3

getNum(); //4

#### 闭包如何形成

外层函数调用后，外层函数的作用域对象(AO)，无法释放，被内层函数对象的scope引用着。

#### 闭包的缺点

1. 无法自动释放！闭包中的函数比普通函数多占用一块内存：外层函数的作用域对象。

只能手动释放：将引用闭包的变量赋值为null

导致内层函数对象释放，导致外层函数的作用域对象(AO)的作用域释放

1. 容易造成内存泄漏

#### 鄙视**：**判断闭包程序的输出结果（画简图）

1. 找受保护的变量是谁？

外层函数的局部变量

1. 找外层函数共返回几个内层函数对象：

* return function(){...}
* 直接给全局变量赋值一个内部function
* 将内部函数保存在一个对象的属性或数组元素中

return[function,function,function]

return{ fun:function(){...} }

例：

function fun(){

var n=999;

//强行给未声明的变量赋值，结果会自动在全局创建该变量

nAdd=function(){n++};

return function(){

console.log(n);

}

}

//nAdd=function(){n++}; //为全局变量

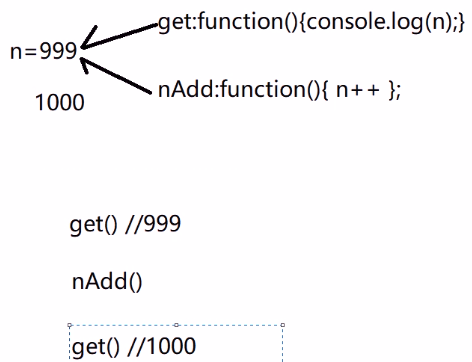
var get=fun();

//get:function(){console.log(n);}

get(); //999

nAdd(); //n++

get(); //1000



例：

function fn(){

for(var i=0,arr=[];i<3;i++){

arr[i]=function(){console.log(i)}

}

return arr;

}

var funs=fn();

//[function(){console.log(i)},function(){console.log(i)},function(){console.log(i)}]

//return返回的为一个函数,此时i已经过自加1为3

//console.log(funs);

funs[0](); //[function(){console.log(i)}],3

funs[1](); //[function(){console.log(i)}],3

funs[2](); //[function(){console.log(i)}],3

#### 结论

同一次外层函数调用，返回的多个内层函数对象，共用同一个受保护的变量

多次外层函数调用，返回的多个内层函数对象，各自拥有自己的受保护的变量，互不影响

#### 返回内层函数的对象的方法（3种）

return function(){...}

直接给全局变量赋值一个内部function

将内部函数保存在一个对象的属性或数组元素中（见上面的例题）

return[function,function,function]

return{fun:function(){...}}

## OOP（重点）

### OOP面向对象编程概述

对象：程序中描述现实中一个事物的属性和功能的程序结构

面向对象：程序中都是用对象来描述现实中一个具体事物的属性和功能。

为什么：现实中，任何数据，必须属于一个具体事物才有意义！

便于大量数据的维护和使用！

何时：今后所有程序都用面向对象方式

三大特点：封装，继承，多态

1. 封装：

创建一个对象，集中存储一个事物的属性和功能

1. 继承

父对象中的成员，子对象无需重复创建，就可直接使用

1. 多态

同一事物，在不同情况下，表现出不同的状态

### 封装

#### 概述

封装：创建一个对象集中存储一个事物的属性和功能

为什么：便于大量数据的维护和使用！

事物的属性会成为对象的属性，其实就是保存在对象中的普通变量

事物的功能会成为对象的方法，其实就是保存在对象中的普通函数

#### 何时

只要使用面向对象编程，都要先封装若干对象

#### 如何（3种）

##### 直接量（方法1）

###### 语法

var obj={

属性名:值,

...:...,

方法名:function(){...}

}

ES6：

var obj={

属性名1:属性值,

属性名...:...,

方法名(){

return ...this.属性名...

}

}

对象的成员：属性+方法=成员

访问属性：对象.属性名——单个属性的用法和普通变量完全一样！

访问方法：对象.方法名()——单个方法的用法和普通函数完全一样！

注：方法是保存在对象中的函数。

###### 何时

在创建对象时，已知对象的成员

###### 问题

对象自己的方法，访问自己的属性，如果不加this，仅会在作用域链中找，不会在对象中找

注：一个没有前缀的变量，则只能在作用域链中查找，无权进入对象。

例：//输出不随变量的改变而改变

var lilei={

sname:"Li Lei",

sage:11,

intr:function(){

console.log(`I'm Li Lei, I'm 11`);

}

}

console.log(lilei.sage);

lilei.sage++;

console.log(lilei.sage);

lilei.intr();

###### 解决

写死函数名，为紧耦合，即当函数名发生改变时，程序依旧无效。

例：//使变量与输出 紧耦合，一旦方法名改变，会报未定义

var lilei={

sname:"Li Lei",

sage:11,

intr:function(){

console.log(`I'm ${lilei.sname}, I'm ${lilei.sage}`);

}

}

console.log(lilei.sage);

lilei.sage++;

console.log(lilei.sage);

lilei.intr();

不写死函数名，为松耦合，即当函数名发生改变时，程序依旧有效。使用.this

例：//使变量与输出松耦合，一旦方法名改变，.前的对象也会改变，达到实时改变的效果

//使用window自有的this的属性

var lilei={

sname:"Li Lei",

sage:11,

intr:function(){

console.log(`I'm ${this.sname}, I'm ${this.sage}`);

}

}

console.log(lilei.sage);

lilei.sage++;

console.log(lilei.sage);

lilei.intr();

var obj=lilei;

lilei=null;

obj.intr(); //this->obj obj.name

1. this.属性名

今后，只要对象自己的方法要访问自己的属性，必须用this.前缀

this：指向正在调用当前函数的**.**前的对象的关键字

1. 强调

不带this.的变量，在作用域链中查找

this.属性在当前对象和当前对象的原型链中找

例：鄙视题

var a=10;

var obj={

a:20,

fun:function(){

var a=30;

console.log(this.a);

}

}

obj.fun();//this->this.a->obj.a->20

var fun=obj.fun;

//fun:function(){var a=30;console.log(this.a);}

/\*windiow.\*/fun(); //this->window window.a->10

注：如果一个函数前不用任何前缀调用时，其this都默认指向window

##### 用new创建一个对象（方法2）

2步

###### 创建空对象

var obj=new Object();{}

简写：new可省略，()也可省略，但至少保留其一，可推广到其它内置对象

###### 为空对象添加新属性

obj.属性名=值;

obj.方法名=function(){...this.属性名...}

例：

var emp1=new Object();

emp1.ename="Tom";

emp1.salary=3500;

emp1.hiredate=new Date(2014,0,15);

emp.getInfo=function(){

return this.ename+this.salary

};

###### 何时

在创建对象时，暂时不知道对象的成员

因为不符合使用习惯，所以一般不使用，

##### 前两种方法对比

以上两种方法存在的问题：一次只能创建一个对象

反复创建多个相同结构对象时，代码冗余量大，不便于维护。

解决：用构造函数反复创建多个相同结构的对象

##### 用构造函数（方法3）

###### 概念

构造函数：专门描述一类对象统一结构的函数

为什么：重用结构的定义

何时：只要反复创建多个相同结构的对象时，都要先定义构造函数

作用：

描述统一的结构

将空对象构建成要求的结构

###### 如何：2步

定义构造函数

function类型名(属性参数列表){

this.属性名=属性参数;

...=...;

this.方法名=function(){

...this.属性名...

} //js中强烈不推荐将方法定义在构造函数中

}

this引用始终指向js解释器当前正在处理的实例

方法声明在构造方法内，则每创建一个实例都会创建一个函数对象

方法声明在构造方法外，则所有实例共用一个函数对象

例：

function Student(sname,sage){

this.sname=sname;

this.sage=sage;

this.intr=function(){

console.log(`I'm ${this.sanme},I'm ${this.sage}`);

}

}

用new调用构造函数

实例化一个对象

var obj=new类型名(属性值,...);

例：

var lilei=new Student("Li Lei",11);

var hmm=new Student("Han Meimei",12);

console.log(lilei);

console.log(hmm);

new：4件事

1. 创建空对象
2. 设置新对象的\_\_proto\_\_继承构造函数的prototype对象
3. 用新对象调用构造函数：

构造函数中的this->新对象

通过强行赋值的方式，为新对象添加新属性和方法

1. 返回新对象的地址保存在变量中

注：

new可让构造函数的this指向新对象

形参变量也是局部变量

###### 优点

代码重用

###### 问题

虽然可重用结构，但放在构造函数中的方法定义，每new一次，都会创建函数对象副本，浪费内存

解决：继承

例：上面2部分例题完整版

function Student(sname,sage){

this.sname=sname;

this.sage=sage;

this.intr=function(){

console.log(`I'm ${this.sanme},I'm ${this.sage}`);

}

}

var lilei=new Student("Li Lei",11);

var hmm=new Student("Han Meimei",12);

console.log(lilei);

console.log(hmm);

例：

function Student(sname,sage){

this.sname=sname;

this.sage=sage;

}

Student.prototype.intr=function(){

console.log(`I'm ${this.sanme},I'm ${this.sage}`);

}

var lilei=new Student("Li Lei",11);

var hmm=new Student("Han Meimei",12);

console.log(lilei);

console.log(hmm);

lilei.intr();

hmm.intr();

console.log(lilei.\_\_proto\_\_==hmm.\_\_proto\_\_);

console.log(lilei.\_\_troto\_\_==Student.protosype);

### 继承

#### 概念

继承：父对象中的成员，子对象无需重复创建就可直接使用！

为什么：代码重用，节约内存！

何时：只要多个子对象拥有相同的属性值或方法时，仅需要集中定义在父对象中一份，所有子对象共用即可

如何：js中的继承都是通过继承原型对象实现的

#### 原型对象

原型对象：保存所有子对象共有成员的父对象

何时：今后只要多个子对象，拥有共同的成员时，都要定义在原型对象中

每个函数中都有一个prototype属性，该属性引用的就是原型对象。

原型对象是保存共享属性值和共享方法的对象。

但，JavaScript并没有规定一个函数的原型类型，因此原型可以是任何类型。

##### 如何

###### 不用创建

1. 买一赠一：每定义一个构造函数，都附赠一个空的原型对象——构造函数.prototype
2. 构造函数.prototype指向原型对象（父级对象）
3. 原型对象.constructor指向构造函数

###### new的第2步（何时如何继承）

每创建一个新子对象，都让新子对象的\_\_proto\_\_自动继承构造函数的原型对象

子对象.\_\_proto\_\_=构造函数的prototype

如何将共有成员放入原型对象中：强行赋值！

###### 向原型对象中添加共有成员

构造函数.prototype.共有成员=值

总结：自定义一种类型：

1. 构造函数：包含属性结构的定义，用于创建子对象
2. 原型对象：包含所有共有方法/属性的定义

##### 自有属性和共有属性

自有属性：保存在子对象本地的属性

共有属性（原型属性）：保存在原型对象中，一旦原型对象中属性值改变，所有继承自该原型的对象属性均改变

获取：即可用构造函数.prototype.属性；

又可用子对象.属性（使用专门的getPrototypeOf方法）

修改：共有属性只能用原型对象修改，不能用子对象修改

自有属性可修改共有属性的效果，但一经修改，将不会跟着共有属性的改变而改变；即相同属性，自有属性的优先级大于共有属性的优先级。

自有属性：必须用子对象.属性名=值

var bool=obj.hasOwnProperty("属性名")

共有属性：必须用构造函数.prototype.属性名=值

不是自有

1. obj.属性名!==undefined
2. 属性名in obj

判断属性名是否在obj的原型链中

例：

function Student(sname,sage){

this.sname=sname;

this.sage=sage;

}

Student.prototype.intr=function(){

console.log(`I'm ${this.sanme},I'm ${this.sage}`);

}

Student.prototype.className="初一2班";

var lilei=new Student("Li Lei",11);

var hmm=new Student("Han Meimei",12);

console.log(lilei);

console.log(hmm);

console.log(lilei.className,hmm.className);

Student.prototype.className="初二2班";

console.log(lilei.className,hmm.className);

lilei.className="初三2班";

console.log(lilei.className,hmm.className);

Student.prototype.className="高一2班";

console.log(lilei.className,hmm.className);

lilei.intr();

hmm.intr();

console.log(lilei.\_\_proto\_\_==hmm.\_\_proto\_\_);

console.log(lilei.\_\_troto\_\_==Student.protosype);

注：

当在原型对象（构造函数.prototype）添加一个新的属性时，所有继承该原型对象的子对象都将继承该属性，在调用时可得；

例：student.prototype.className="高一二班";

console.log(lilei.className); //高一二班

若在子对象上添加一个新的属性时，该对象可获得性的属性；

但，当在原型对象上所添加的属性与子对象上所添加的属性名相同时，以子对象只会显示子对象上新增的属性，而不是原型对象上新增的属性

##### 内置类型/对象

其实内置对象类型Array,Date...都是构造函数

每种类型都有自己的原型对象：Array.prototype,Date.prototype,..

###### 问题

旧浏览器不支持新的API

解决：在该类型的原型对象中添加自定义API弥补缺失

判断

if(typeof内置类型.prototype.API!=="function")

if(!("API"in内置类型.prototype))

添加

内置类型.prototype.方法=function(参数){

...this//获得将来调用方法的.前的对象

}

## 内置类型

### 概念

什么是：ES标准中规定的，浏览器已经实现的类型/对象

### 分类

包括：11个

String Number Boolean （包装类型）

Array Date Math RegExp

Error

Function Object

global（在浏览器中被window代替！）

除Math和global外，其余都是一种类型：

都包含2部分：

1. 构造函数：创建该类型的子对象
2. 原型对象：保存该类型所有子对象共有的API

其实内置对象类型Array,Date...都是构造函数

每种类型都有自己的原型对象：

问题：就浏览器不支持新API

解决：在该类型的原型对象中添加自定义API弥补缺失

### 包装类型

什么是：专门封装原始类型的值，并提供操作原始类型值的API

手机号即为原始类型的值，手机即为包装类型

为什么：原始类型的值本身不具有任何属性和功能

何时：只要试图用原始类型的值调用API时，都会自动创建该类型的包装类型对象，来辅助原始类型的值完成操作。

1. 封装原始类型的值
2. 调用包装类型中的方法操作原始类型的值

例：

var price=12.5;

price.toFixed(2) //typeof price number

"12.50" //new Number(price).toFixed(2)

//new Number释放！

例：

str.charCodeAt()

typeof str

string

//new String(str).charCodeAt();

//new String(str)自动释放

例：

var chl="成亮";

chl.money=10;

//new String(chl).money=10 //系统自动new的对象的地址没有保存到chl里，但将值保存到了chl中

//释放 //因为没有讲地址保存到一个变量中，就找不到（释放）了

console.log(chl.money); //undefined

console.log(chl); //10

console.log(typeof(chl)); //string

//new String(chl).money

var chl=new String("成亮");

chl.money=10;

console.log(chl.money); //10

console.log(typeof(chl)); //object

var obj={ename:"李然"}; //object

obj.money=10;

console.log(obj.money); //10

鄙视：判断一个对象是不是数组类型，共有几种方式

typeof不行！

#### 原始类型与引用类型

基本类型值：指的是保存在栈内存中的简单数据段

引用类型值：指的是那些保存在堆内存中的对象，意思是，变量中保存的实际上只是一个指针，这个指针执行内存中的另一个位置，由该位置保存对象

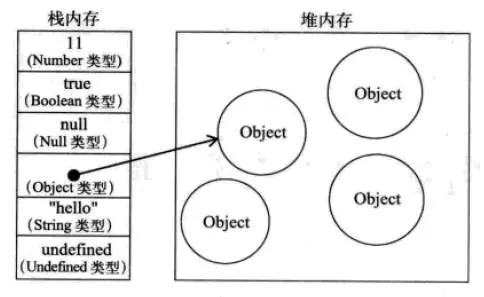
原始类型：undefined、Null、Boolean、Number、string

引用类型：Function，Array，Object

##### 两种访问方式

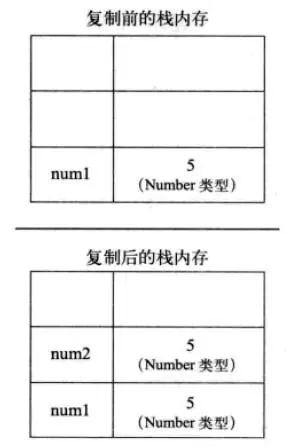
基本类型值：按值访问，操作的是他们实际保存的值；

引用类型值：按引用访问，当查询时，我们需要先从栈中读取内存地址，然后再顺藤摸瓜地找到保存在堆内存中的值；

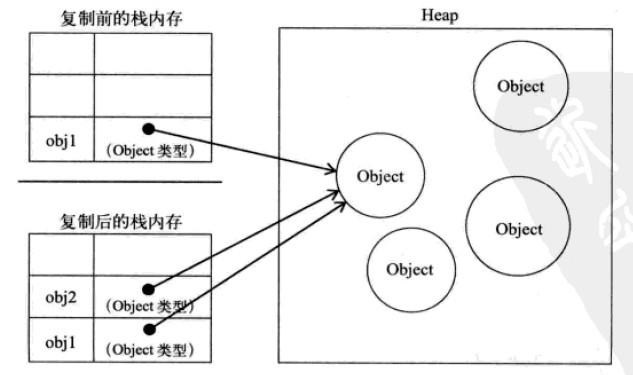


##### 两种类型复制

1. 基本类型变量的复制：从一个变量向一个变量复制时，会在栈中创建一个新值，然后把值复制到为新变量分配的位置上



1. 引用类型变量的复制：复制的是存储在栈中的指针，将指针复制到栈中未新变量分配的空间中，而这个指针副本和原指针执行存储在堆中的同一个对象；
2. 复制操作结束后，两个变量实际上将引用同一个对象；因此改变其中的一个，将影响另一个；



##### 两种变量类型检测

Typeof操作符是检测基本类型的最佳工具,判断一个值是否表示一种原始类型，并返回是哪种原始类型（注意：对Null值使用typeof运算符，返回的是object）；

如果变量值是nul或者对象，typeof 将返回“object”；

Instanceof用于检测引用类型，可以检测到具体的，它是什么类型的实例；

如果变量是给定引用类型的实例，instanceof操作符会返回true;

#### 原型链（prototype chain）

由多级父对象逐级继承形成的链式结构。

保存着所有对象的属性。

控制着对象成员的使用顺序和共享范围。

注：先用自有属性；

若自己没有，才沿原型链向父级找，原型链上没有，返回undefined。

##### 原型链 vs 作用域链

|  |  |  |
| --- | --- | --- |
|  | 原型链 | 作用域链 |
| 概念 | 由多级父对象逐级继承形成的链式结构 | 由多级作用域逐级引用形成的链式结构 |
| Object | Function |
| 存储 | 保存着所有对象的属性 | 保存着所有变量 |
| 控制 | 控制着对象成员的使用顺序和共享范围 | 控制着所有变量的使用顺序 |
| 特点 | 先在自身找，找不到再去父级找；  若父级没有，报undefined | 先在局部作用域找，找不到再去父级（全局作用域）找；  若父级没有，报错 |
| 总结 | 所有对象.访问的属性都保存在原型链上  原型链上的属性必须用“对象.”才能访问 | 所有不用.的变量都在作用域链上找  作用域链中的变量不用.，可直接访问 |

#### 判断对象的类型

##### 类型1typeof

typeof

只能区分基础类型和function

不能进一步区分对象的类型

##### 类型2类型.prototype.isPrototypeOf

var bool=类型.prototype.isPrototypeOf(child)

不仅检查直接父对象，而且检查整个原型链

##### 类型3child instanceof构造函数

var bool=child instanceof构造函数

不仅检查直接父对象，而且检查整个原型链

问题：

1. 检查整个原型链
2. 继承关系可能被修改

##### 类型4class

检查内置class属性

1. Object.prototype.toString.call(obj)==“[object class]”

obj.toString()

1. 更严格

class属性直接保存在对象本地

只在创建对象时确定类型

对象创建后，不随继承关系的改变而改变

##### 类型5Array.isArray(obj)

如果检查数组类型

var bool=Array.isArray(obj)

1. ES5

IE9+

1. 原理和Object.prototype.toString.call一样

###### 鄙视

方法定义在原型对象中，还是定义在构造函数对象上

答：

1. 如果方法仅限当前类型的子对象可用，其他类型的对象不可用，就定义在原型对象中，但必须当前类型的子对象才能调用——实例方法
2. 如果方法不确定将来调用它的对象类型，就定义在构造函数对象上，不需要任何对象实例，即可用构造函数名直接调用——静态方法

#### toString()、toLocalString()、valueOf()三种方法之间的区别

Array、Boolean、Date、Number等对象都具有toString()、toLocaleString()、valueOf()三个方法。

##### valueOf()

valueOf()用于返回指定对象的原始值

//Array：返回数组对象本身

var array=["CodePlayer",true,12,-5];

document.writeln(array.valueOf()===array);//true

//Date：当前时间距1970年1月1日午夜的毫秒数

var date=new Date(2013,7,18,23,11,59,230);

document.writeln(date.valueOf());//1376838719230

//Number：返回数字值

var num=15.26540;

document.writeln(num.valueOf());//15.2654

//布尔：返回布尔值true或false

var bool=true;

document.writeln(bool.valueOf()===bool);//true

//new一个Boolean对象

var newBool=new Boolean(true);

//valueOf()返回的是true，两者的值相等

document.writeln(newBool.valueOf()==newBool);//true

//但是不全等，两者类型不相等，前者是boolean类型，后者是object类型

document.writeln(newBool.valueOf()===newBool);//false

//Function：返回函数本身

function foo(){

}

document.writeln(foo.valueOf()===foo);//true

var foo2=new Function("x","y","return x+y;");

document.writeln(foo2.valueOf()===foo2);//true

//Object：返回对象本身

var obj={name:"张三",age:18};

document.writeln(obj.valueOf()===obj);//true

//String：返回字符串值

var str="http://www.365mini.com";

document.writeln(str.valueOf()===str);//true

//new一个字符串对象

var str2=new String("http://www.365mini.com");

//两者的值相等，但不全等，因为类型不同，前者为string类型，后者为object类型

document.writeln(str2.valueOf()===str2);//false

##### toString()

toString()将当前对象以字符串的形式返回,返回值为String类型

//数组

var array=["Code",true,1,-2];

document.writeln(array.toString());//Code,true,1,-2

//日期

var date=new Date(2013,7,18,23,11,59,230);

document.writeln(date.toString());//Sun Aug 18 2013 23:11:59 GMT+0800(中国标准时间)

//数字

var num=15.26540;

document.writeln(num.toString());//15.2654

//布尔

var bool=true;

document.writeln(bool.toString());//true

//Object返回"[object ObjectName]"，其中ObjectName是对象类型的名称

var obj={name:"张三",age:18};

document.writeln(obj.toString());//[object Object]

//HTML DOM节点

var eles=document.getElementsByTagName("body");

document.writeln(eles.toString());//[object NodeList]

document.writeln(eles[0].toString());//[object HTMLBodyElement]

//function返回如下格式的字符串，其中functionname是一个函数的名称，此函数的toString方法被调用："function functionname(){[native code]}"

##### toLocaleString()

toLocaleString()函数用于将当前对象以字符串值的形式返回，该字符串的格式适合当前宿主环境的当前区域设置。

//数组

var array=["CodePlayer",true,12,-5];

document.writeln(array.toLocaleString());//CodePlayer,true,12,-5

//日期

var date=new Date(2013,7,18,23,11,59,230);

document.writeln(date.toLocaleString());//2013年8月18日下午11:11:59

//数字

var num=15.26540;

document.writeln(num.toLocaleString());//15.265

//布尔

var bool=true;

document.writeln(bool.toLocaleString());//true

//Object

var obj={name:"张三",age:18};

document.writeln(obj.toLocaleString());//[object Object]

//HTML DOM节点

var eles=document.getElementsByTagName("body");

document.writeln(eles.toLocaleString());//[object NodeList]

document.writeln(eles[0].toLocaleString());//[object HTMLBodyElement

##### 三种对比

###### Array

例：

var array = new Array("niu","li","na");

console.log(array.valueOf());

console.log(array.toString());

console.log(array.toLocaleString());

结果：

Array[3]

niu,li,na

niu,li,na

valueOf：返回数组本身

toString()：把数组转换为字符串，并返回结果，每一项以逗号分割。

toLocalString()：把数组转换为本地数组，并返回结果。

###### Boolean

例：

var boolean = new Boolean();

console.log(boolean.valueOf());

console.log(boolean.toString());

结果：

false

fales

valueOf：返回Boolean对象的原始值。

toString()：根据原始布尔值或者booleanObject对象的值返回字符串"true"或"false"。默认为"false"。

toLocalString()：Boolean对象没有toLocalString()方法。但是在Boolean对象上使用这个方法也不会报错。

###### Date

例：

var date = new Date();

console.log(date.valueOf());

console.log(date.toString());

console.log(date.toLocaleString());

结果：

1469261342234

Sat Jul 23 2016 16:09:02 GMT+0800(CST)

2016/7/23 下午4:09:02

valueOf：返回Date对象的原始值，以毫秒表示。

toString()：把Date对象转换为字符串，并返回结果。使用本地时间表示。

toLocalString()：可根据本地时间把Date对象转换为字符串，并返回结果，返回的字符串根据本地规则格式化。

###### Math

例：

console.log(Math.PI.valueOf());

结果：

3.141592653589793

valueOf：返回 Math 对象的原始值。

###### Number

例：

var num = new Number(1337);

console.log(num.valueOf());

console.log(num.toString());

console.log(num.toLocaleString());

结果：

1337

1337

1,337

valueOf：返回一个Number对象的基本数字值。

toString()：把数字转换为字符串，使用指定的基数。

toLocalString()：把数字转换为字符串，使用本地数字格式顺序。

###### String

例：

var string = new String("abc");

console.log(string.valueOf());

console.log(string.toString());

结果：

abc

abc

valueOf：返回某个字符串对象的原始值。

toString()：返回字符串。

###### toString()方法与toLocalString()方法区别

toLocalString()是调用每个数组元素的toLocaleString()方法，然后使用地区特定的分隔符把生成的字符串连接起来，形成一个字符串。

toString()方法获取的是String(传统字符串),而toLocaleString()方法获取的是LocaleString(本地环境字符串)。

如果你开发的脚本在世界范围都有人使用，那么将对象转换成字符串时请使用toString()方法来完成。

LocaleString()会根据你机器的本地环境来返回字符串，它和toString()返回的值在不同的本地环境下使用的符号会有微妙的变化。

所以使用toString()是保险的，返回唯一值的方法,它不会因为本地环境的改变而发生变化。如果是为了返回时间类型的数据，推荐使用LocaleString()。若是在后台处理字符串，请务必使用toString()。

### 多态

多态：同一个函数在不同情况下表现出不同的状态

重载：

重写(override)：如果子对象觉得父对象继承来的成员不好用，可在子对象本地定义同名成员，来覆盖父对象成员。

为什么：子对象觉得父对象继承来的成员不好

何时：只要子对象觉得父对象继承来的成员不好

如何：在子对象本地定义同名成员，来覆盖父对象成员。

注：每个对象中都隐藏着一个class属性（call），记录着对象创建时的类型名，不随继承关系的改变而改变。

即：Object.prototype.toString.call(obj1)

Object.prototype——要抢方法的全名（的对象名）

.toString——要抢方法的全名（的方法名）

.call——用来进行“抢”的函数名

(obj1)——要使用该方法的对象名

其在调用的一瞬间，相当于obj1.toString()，可避免“重载”所带来的影响。

### 静态

#### 静态方法vs实例方法

##### 实例方法

什么是：必须先创建对象实例，再用对象实例调用的方法

何时：要求必须该类型的子对象才能调用的方法

如何：所有定义在原型对象中的方法，都是实例方法

存储：原型对象中

使用范围：只允许当前类型的子对象使用

调用：子对象.方法（）

##### 静态方法

什么是：无需创建对象实例，就可用类型名直接调用的方法

何时：不确定/不限制调用该方法的对象类型时

如何：直接定义在构造函数对象上的方法

存储：构造函数对象中

使用范围：任何对象都可以使用

调用：类型名.方法（）

类型名.静态方法=function(){...}

例：

function Product(pid,pname){

this.pid=pid;

this.pname=pname;

this.getById=function(pid){

console.log(`将当前编号为:${pid}的一个商品`);//该函数放在构造函数，将传递给子级，由子级使用

}

}

Product.getById=function(pid){//在妈妈身上

console.log(`将当前编号为:${pid}的一个商品`);

}//该函数由放在原型对象，由构造函数使用，子级不使用

Product.prototype.save=function(){//强行添加save的function

//在爹身上

console.log(`将当前商品:${this.pname}保存到数据库中...`);//该this指的是下面.save()前的macbook

}

//按id查询一个商品：hetById(pid)——静态()（不需要实例就能调用，为返回一个对象）

//保存一个商品到数据库：save()——实例()（需要有对象才能储存）

Product.getById(1001);

var macbook=new Product(1001,"苹果本");

macbook.save();

### 自定义继承

即为改父对象

#### 仅修改一个对象的父对象

获得子对象的父对象

var father=Object.getPrototypeOf(child);

设置子对象继承指定的父对象

child.\_\_proto\_\_=father;

Object.setPrototypeOf(child,father)

上下两句代码效果一样，但由于上面使用\_\_proto\_\_可能不保险，所有建议使用下面的。

#### 修改所有子对象的父对象

修改构造函数原型对象，来修改所有子对象的父对象

构造函数.prototype=father

时机：必须紧跟在构造函数定义之后

必须在创建子对象之前修改

#### 仅基于现有父对象，创建子对象，并扩展自有属性

Object.create()

创建新子对象，继承父对象，扩展子对象自有属性

var child=Object.create(father,{

//defineProperties

属性名:{四大特性},

...:...

})

##### 鄙视：模拟实现Object.create()

Object.create=function(father,props){

//var child=new Object();

//Object.setPrototypeOf(child,father);

var Fun=function(){};

Fun.prototype=father;

var child=new Fun();

//Object.defineProperties(child,props);

if(props!==undefined){

for(var key in props){

child[key]=props[key].value;

}

}

return child;

}

#### 两种类型间的继承

##### 何时

如果发现多个类型拥有部分相同的属性结构和方法定义

都要抽象父类型。

##### 如何

问题：两种类型间包含部分相同的属性结构和方法定义

解决：定义抽象父类型

如何：2步

###### 定义抽象父类型

1. 父类型的构造函数中定义相同的属性
2. 父类型的原型对象中定义相同的方法

###### 让子类型继承父类型

在子类型构造函数中借用父类型构造函数

1. extends

让父类型构造函数帮助添加相同部分的属性定义

子类型构造函数仅负责添加独有的属性定义即可

1. 错误

直接调用父类型构造函数

this->window

父类型中的属性都泄露到全局

1. 正确

父类型构造.call(this,参数1,参数2,...)

简写：父类型构造.apply(this,arguments)

例：

function Enemy(fname,speed){

this.fname=fname;

this.speed=speed;

}

Enemy.prototype.fly=function(){

console.log(`${this.fname} 以时速 ${this.speed} 飞行！`);

}

function Plane(fname,speed,score){

//Enemy(fname,speed);//因为Enemy调用时前面没有.，则会导致其指向window，从而使得其参数泄露至全局作用域

Enemy.call(this,fname,speed);//call将替换this的对象，后面紧跟着是要给函数的参数值

//此处的this是Plane中的，而Plane是new出来的，所以其this始终是对的

this.score=score;

}

Plane.prototype.getScore=function(){

console.log(`击落 ${this.fname} 得 ${this.speed} 分`);

}

Object.setPrototypeOf(

Plane.prototype,Enemy.prototype

);

var f16=new Plane("F16",1000,5);

f16.fly();

f16.getScore();

console.log(f16);

1. 鄙视：

call vs apply

相同：都是强行借用一个本来无法调用的函数，并临时替换函数中this为指定对象

不同：call：传入借用函数的参数，必须单独传入，逗号分隔

apply：传入借用函数的参数，放在一个数组中整体传入

可自动打散数组类型参数

让子类型原型对象继承父类型原型对象

inherits

Object.setPrototypeOf(

子类型构造.prototype,父类型构造.prototype

)

## ES5

ECMAScript：JavaScript的核心语法标准

简称：ES

版本：3.1、5、6、……

### 严格模式

什么是：比普通js运行机制要求更严格的模式

为什么：普通js存在很多广受诟病的缺陷

何时：今后所有js程序都应该运行在严格模式下

#### 如何

1. 在js文件/<script>元素的顶部，添加："use strict"，就可在整个js文件或<script>元素内启用严格模式
2. 仅在一个函数顶部添加："use strict"，仅在一个函数内启用严格模式

#### 要求

1. 禁止给未声明的变量赋值
2. 静默失败升级为错误

静默失败：执行不成功，还不报错！

1. 不再推荐使用arguments.callee和caller

arguments.callee可自动获得当前正在执行的函数本身。

专门用于实现递归调用。

递归的问题：效率极低。重复计算量太大。

解决：所有递归都可用循环代替！

1. 匿名函数调用中的this不再指向window，而指向undefined。

### 保护对象

#### 概述

什么是：限制对对象中属性值和结构的修改

为什么：js中的对象毫无自保能力！

如何：仅保护一个属性

保护对象的整体结构

#### 对象属性的保护（仅保护一个属性）

##### 内部属性

不能用**.**访问的属性

比如：class、extensible:true

例：

Object.prototype.toString.call(obj)

var bool=Object.isExtensible(obj)

Object.preventExtensions(obj)

##### 命名属性

可直接用.访问的属性

细分为：

数据属性：直接存储属性值的属

访问器属性：实际存储属性值，仅提供对另一个数据属性的保护

###### 数据属性

数据属性：直接存储一个属性值的属性

如何保护数据属性：

ES5将每个数据属性又变为一种小对象。

每个属性对象中包含一个值和三个开关：

{

value: //实际存储属性值,

writable:true/false //控制是否可修改,

enumerable:true/false //控制是否可被for in遍历

//隐藏不彻底，用**.**可访问

configurable:true/false //控制是否可删除该属性

//控制是否可修改前两个开关

//一旦改为false，不可逆

}

注：

delete：为关键字，可删除任意对象的任意属性（由于过于危险，不建议使用）

delete eric.ename;

获得一个属性的四大特性

var attrs=Object.getOwnPropertyDescriptor(obj,"属性")

修改一个属性的四大特性

Object.defineProperty(obj,"属性名",{

开关名:true/false,

...:...

})

强调：通常修改writable和enumerable时都会带上configurable为false，作为双保险

同时修改多个属性的特性

Object.defineProperties(obj,{

属性名:{

开关:true/false

...:...

},

属性名:{

开关:true/false

...:...

},

})

问题：无法使用自定义规则保护属性

解决：用访问器属性

###### 访问器属性

访问器属性：不实际存储属性值，仅提供对另一个数据属性的保护。而是包含一对儿get/set方法。

何时：只要用自定义规则保护属性使

例：age属性必须介于18~65之间

var eric={

eid:1001,

\_age:22 //将元素改名，使其不容易被找到

};

//访问器属性

Object.defineProperty(eric,"age",{

get:function(){

console.log("自动调用age的get");

return this.\_age;

},

set:function(newVal){

console.log("自动调用age的set");

if(newVal>=18&&newVal<=65){

this.\_age=newVal;

}else{

throw new Error("年龄必须介于18~65之间");

}

},

enumerable:true,//允许被遍历

configurable:false//不许被删除

});

console.log(eric.age);

eric.age=23;

console.log(eric.age);

eric.age=70;

如何：

先定义隐藏的受保护的数据属性实际存储数据；

（其实应该用闭包保护局部变量，实际存储属性值）

不能用直接量添加，只能用API添加定义、添加访问器属性。

即：

Object.defineProperty(obj,"访问器属性名",{

//四大特性

**get**:function(){ //get每次试图取值时自动调用

return this.数据属性

},

**set**:function(val){ //set每次试图赋值时自动调用

//val自动获得要赋的新值

if(val符合规则)

this.数据属性=val

}else{

throw new Error("报错！")

},

enumerable:true,

configurable:false

});

使用访问器属性：访问器属性的用法和普通数据属性完全一样，只不过：

console.log(obj.访问器属性)时，自动调用get方法

obj.访问器属性=新值时，自动调用set()，newVal自动获得新值。

get用来访问数据，set用来修改数据

如省略set方法，给访问器属性相当于只读

###### 特性的默认值

用.添加的新属性，特性的默认值为true

defineProperty/defineProperties添加的新属性，特性的默认值为false

#### 保护对象结构——防篡改

##### 防扩展

禁止给对象添加新属性

Object.preventExtensions(obj);

原理：每个对象都有一个内部属性：extensible:true

preventExtensions()可将extensible:false

Object.Extensible(obj):获取对象是否可扩展

例：

function Emp(eid,ename,salary,age){

//this添加保护

Object.defineProperties(this,{

eid:{

value:eid,

writable:false,

enumerable:true,

configurable:false

},

ename:{

value:ename,

writable:true,

enumerable:true,

configurable:false

},

salary:{

value:salary,

writable:true,

enumerable:false,

configurable:false

},

\_age:{

writable:true,

enumerable:false,

}

});

this.age=age;//红色的age是同一个age，黄底的age是同一个

Object.preventExtensions(this);//防扩展

//Object.seal(this);//密封//和防扩展二选一，但要删除所有的configurable:false

}

//Emp.prototype.age={//访问器不能用.添加

Object.defineProperty(Emp.prototype,"age",{//因为访问器属性过于特殊，所以只能这么使用，虽然上下两句意思相同

get:function(){return this.\_age},

set:function(val){

if(val>=18&&val<=65){

this.\_age=val;

}else{

throw new Error("年龄必须介于18~65之间")

}

},

enumerable:true,

configurable:false

});

var eric=new Emp(1001,"埃里克",12000,22);

console.log(eric);//打印

for(var key in eric){//遍历

console.log(key+":"+eric[key]);

}

//eric.age=-2;

eric.Age=-2;

console.log(eric);//打印

##### 密封

在兼顾防扩展的同时，又禁止删除现有属性

Object.seal(obj)

原理：将extensible:false

自动将每个属性的configurable:false

Object.isSealed(obj)：判断obj对象是否为密封对象。

##### 冻结

在兼顾密封基础上，禁止修改一切属性值！

Object.freeze(obj)

原理：将extensible:false

自动将每个属性的configurable:false

自动将每个属性的writable:false

何时：在一些公用的配置信息对象上

Object.isFrozen（obj)：判断obj是否被冻结

### Object.create()

什么是：仅有父对象，没有构造函数，也想创建子对象。

create原理：

1. 创建空对象
2. 让空对象继承father
3. 为空对象扩展自有属性

何时：如果没有构造函数，也想创建子函数时

如何：

var child=Object.create(father,{

属性名:{四大特性},

...:...

})

例：

var father={ bal:100000000, car:"infiniti" }

var hmm=Object.create(father,{

//相当于defineProperties

bao:{

value:"LV",

writable:true,

enumerable:true,

configurable:false

},

phone:{

value:"iPhoneXR",

writable:true,

enumerable:true,

configurable:false

}

});

console.log(hmm); //{bao:"LV",phone:"iPhoneX MAX"}

console.log(hmm.bao,hmm.car); //100000000 "infiniti"

#### 替换函数中的this

何时：如果一直函数执行时，其中的this不是想要的

如何：2种

1. 在调用函数时，临时替换一次 **this:.call**

函数**.call**(obj,参数值,...)

调用函数，并用obj代替函数中的this，并传入参数值

打散数组参数

何时：如果函数需要传入多个参数值，但参数是**放在数组中**给定的。

如何：函数.apply(替换this的对象,数组)

apply可先打散数组为单个元素，再传入函数

注：当发现给的参数列表与函数规定的参数列表不一致时，使用apply

例：

function Plane(fname,speed,score){

//Enemy.call(this,fname,speed);

Enemy.apply(this,arguments)

}

1. 创建新函数副本，永久绑定this

基于原函数，为某个函数创建一个新的函数副本，并永久绑定this（回调函数）为指定对象：**.bind()**

var新函数=函数.bind(替换this的对象)

强调：bind不是调用函数的意思，而是创建一个和fun一模一样的新函数副本。

新函数中永久绑定this为指定的obj对象，同时还可绑定部分参数值为指定值。

固定场景：今后只要替换回调函数中的this时，只能用bind！

例：

function calc(base,bonus1,bonus2){

console.log(`${this.ename} 的总工资是:${base+bonus1+bonus2}`);

}

var lilei={ename:"Li Lei"};

var hmm={ename:"Han Meimei"};

calc.call(lilei,10000,2000,3000); //Li Lei 的总工资是:15000

var sals=[4000,5000,6000];

calc.apply(hmm,sals); //Han Meimei 的总工资是:15000

//apply要求传入的数据必须放在一个数组中

//calc(4000,5000,6000)

//为李雷创建专属计算器函数，并永久绑定this为liliei，永久绑定第一个参数base为10000

var l\_calc=calc.bind(lilei,10000);

console.log(l\_calc);

//1.鼠标放在f calc()看函数定义

//2.bound this->lilei

//3.bound args->0:10000

console.log(l\_calc==calc);

//今后调用，再不用重复输入lilei和10000，this永久绑定为lilei，部分参数也永久绑定

l\_calc(2000,3000);//Li Lei 的总工资是:15000

l\_calc(1000,6000);//Li Lei 的总工资是:17000

#### 关于Object类型

Object是Object类型的构造函数，所以typeof Object返回的是function。

而Object.prototype对象才是所有对象的父对象。

Object作为构造函数，有三种情况：

1. Object()：如果传入参数null，undefined，或者不传入，则相当于new Object()，创建空对象；
2. Object（原始类型值）：将原始类型的值转华为Object类型，也等效于new Object（原始类型值）；
3. Object（引用类型的对象）：什么也不做，返还原对象。

Object.getPrototyprOf(obj)：

Object.getOwnPropertyDescriptor(obj,"属性名")：返回对象中指定属性的特性；

Object.getOwnPropertyNames(obj)：获得对象中所有属性名的数组；

Object.keys(obj)：仅返回可被枚举的属性名组成的数组。

### 数组API

#### 查找元素在数组中的位置

arr.indexOf()用法同str.indexOf()

var i=arr.indexOf(elem[,fromi]);

返回值：elem的位置i，如果找不到，返回-1.

#### 判断

判断数组的元素是否符合要求

arr.every()判断数组中是否所有元素都符合条件，返回bool值

arr.some()判断数组中是否包含符合条件的元素，返回bool值

例：

var bool=arr.every(function(elem,i,arr){

//function会自动在每个元素上调用一次回调函数

//elem:获得当前遍历到的元素值

//i:获得当前所在的元素位置

//arr:获得当前数组对象

//不能用this，因为function(elem,i,arr)前没有**.**，所以this->window

//故只能用arr->当前数组

//arr1.every()的this->arr1，every在调用时将this传给了arr

return判断条件

})

例：

var arr1=[1,2,3,4,5];

var arr2=[2,4,6,4,2];

console.log("偶数组成");

//判断哪个数组全由偶数组成

console.log(

arr1.every(function(elem){

return elem%2==0;

}), //false

arr2.every(function(elem){

return elem%2==0;

}) //true

);

例：//判断哪个数组是升序排列

console.log(

arr1.every(

// this->arr1

// ↓

function(elem,i,arr){

//this->window

return i<arr.length-1?arr[i+1]>=elem:true;

}

),

arr2.every(

function(elem,i,arr){

return i<arr.length-1?arr[i+1]>=elem:true;

}

)

);

#### 遍历

arr.forEach():对原数组中每个元素执行相同的操作

（在原数组上修改）

如何：

arr.forEach(function(elem,i,arr){

arr[i]=新值

})

arr.map():取值原数组中每个元素值，执行相同操作后，放入新数组返回。

（保护原数组）

var newArr=arr.map(function(elem,i,arr){

return新值;

})

例：

var arr=[1,2,3,4,5];

arr.forEach(function(elem,i,arr){

//elem仅是原元素值的副本，即原始类型的按值传递

//elem\*2;//[1,2,3,4,5]

arr[i]\*2;//[2,4,6,8,10]

arr[i]=elem\*2;//[2,4,6,8,10]

});

console.log(arr);//[2,4,6,8,10]

var arr=[{x:1},{x:2},{x:3},{x:4},{x:5}];

arr.forEach(function(elem){//引用类型

elem.x\*=2;//elem获得了原元素对象的地址值

});

console.log(arr);

#### 过滤

过滤：复制出原数组中符合条件的元素，放入新数组返回。

原数组保持不变。

如何：

var subArr=arr.filter(function(elem,i,arr){

return条件

});

#### 汇总

汇总：对数组中的元素进行统计

如何：

var result=arr.reduce(function(prev,elem,i,arr){

//prev:截止到当前元素之前的临时汇总值

returnprev+elem; //将当前elem汇总到prev中的新汇总值

},起始值);

#### bind函数

bind函数用于基于现有函数创建一个新函数对象。同时为新函数对象提前绑定一个新函数对象。同时为新函数对象提前绑定this值或这只参数值。

bind函数在调用时，需要指定this的值，同时还可以指定若干个形式参数的实际值。

## ES6

ES6不改变原理的基础上，将程序变得最简化！

学ES6，更重要的是学兼容性

ES6中，变量的作用域分为三种：

局部作用域：只能在当前函数内使用；

全局作用域：可以在任意函数内使用——是全局对象的成员；

块级作用域：只能在当前快内使用——ES6新增

### let

什么是：代替var，用于声明变量，切该声明仅在当前块中有效。

var的两大缺陷：

1. 会被声明提前

避免了声明提前

原理：在同一作用域内，let前不允许提前使用未声明的同名变量

1. 没有块级作用域

为js添加了块级作用域

匿名函数自调，来划分临时作用域

let的原理：2步

1. 自动给变量起别名
2. 用匿名函数自调封装后续代码

即：(function(){

var \_a=10;

console.log(\_a);

})();

注意：同一作用域内，不允许重复let同一个变量

例：

var t=0;

function conn(){

console.log("0.3s");

t+=0.3;

}

function query(){

//var t; //声明提前，

console.log("0.8s");

t+=0.8;

var err=false;

if(err){

let \_t=new Date(); //若用var，则会造成声明提前，上面t+=0.8;的t会调用局部作用域的t而不是全局作用域的t，所以要改成let

console.log(

"is error at:"+\_t.toLocaleString()

);

/\*let原理：

(function(){//作用域

var \_t=new Date();

console.log(

"is error at:"+\_t.toLocalesString()

);

});

\*/

}

}

conn();

query();

console.log(`sum:${t}s`);

### 参数增强

#### 默认值

允许为函数的参数列表末尾的几个参数变量，预先定义默认值。

function fun(参数1,...,参数n=默认值)

调用时，如果没有给形参n传参，则用默认值代替。

兼容：if（形参n===undefined）

形参n=默认值

允许给函数参数设置默认值，将来即使不传入参数值，也可使用默认值代替

参数3=参数3||默认值

强调：有默认值的参数，必须放在参数列表的末尾

例：

Array.prototype.indexOf=function(val,starti=0){

//if(starti===undefined) starti=0;

for(var i=starti;i<this.length;i++){

if(this[i]===val)

return i;

}

return -1;

}

var arr=[1,2,3,4,3,2,1];

console.log(

arr.indexOf(3),//2

arr.indexOf(3,3)//4

);

#### 剩余参数(rest)

代替arguments接收不确定个数的参数值

arguments的问题：

1. 不是纯正的数组
2. 无法有选择的获得参数值，只能获得全部参数值

如何：

定义：function fun(参数1,...,...数组名){

//“数组名”可接住除之前参数外，剩余的参数值

//是一个纯正的数组

}

兼容：用arguments将类数组对象转为数组（见例2）

var arr=Array.prototype.slice.call(arguments,starti)

可简写为：[].slice.call(arguments,starti)

例：

function calc(ename,...arr){

//"Li Lei",[10000,2000]

//"Han Meimei",[4000,2000,3000,5000]

var sum=arr.reduce(function(prev,elem){

return prev+elem

},0);

console.log(`${ename}的总工资为：${sum}`);

}

calc("Li Lei",10000,2000);

calc("Han Meimei",4000,2000,3000,5000);

例：

function calc(ename){

//arguments["Li Lei",10000,2000]

var arr=//如何将类数组对象转为数组

Array.prototype.slice.call(arguments,1);//slice不是arguments的API，所以要用call抢

//[].slice.call(arguments,1) //简版

//arr:[10000,2000]

var sum=arr.reduce(function(prev,elem){

return prev+elem

},0);

console.log(`${ename}的总工资是：${sum}`);

}

calc("Li Lei",10000,2000);

calc("Han Meimei",4000,2000,3000,5000);

#### 散播(spread)

代替apply用于打散数组类型参数

何时：如果函数需要单独传入参数，但给定的参数值却是一个数组时

调用函数时：fun(**...**数组)

**...**可将数组打散为单个元素，再传给fun()

例：Math.max(...arr);

兼容：fun.apply(null,数组)

### 箭头函数

对所有回调函数的简写

如何：

1. 去function改为=>
2. 如果只有一个参数，可省略()
3. 如果函数体中只有一句话，可省略{}

如果仅有的一句话还是return，必须省略return。

强调：箭头函数内外的this，通用！（所以构造函数是设立块级作用域，而箭头函数将内外this的通用，使得块级作用域失效）

所以，如果反而希望内外this不同时，就不能用箭头函数简化

反之，如果希望内外this始终一致时，就必须用箭头函数

事件处理函数不能改为箭头函数。

兼容：若不兼容，则用回function。

### 模板字符串

1. 代替字符串拼接
2. 反引号包裹的字符串`xxx`
3. 支持换行
4. 支持动态生成内容

在``中使用${js表达式}

模板字符串会自动执行js表达式的结果，并拼接到最终生成的普通字符串中

ESLint中规定不允许使用+拼接字符串，必须使用模板字符串

### 解构

解构：将大的对象中的成员，提取出来，单独使用从而简化变量的批量赋值

#### 数组解构

提取出数组中的个别元素，单独使用。

如何：下标对下标 (交换两个变量值）

var[变量1,变量2,...]=数组;

结果：变量1=数组[0],变量2=数组[1]

例：var[a,b,c]=[1,2,3]

[a,b]=[b,a]

例：

var date=[2018,11,6];

var [y,m,d]=date; //[y,m,d]不是数组，仅仅是外形与[2018,11,6]的外形相似。从而达到下标对下标的效果

console.log(y); //2018

console.log(m); //11

console.log(d); //6

#### 对象解构

提取出对象中的个别成员，单独使用

如何：属性对属性

var{属性1:变量1,属性2:变量2}=对象

结果：变量1=对象[属性1]，...

简写：如果属性名和变量名刚好相同，可只写属性名即可。

var{属性1,属性2,...}=对象

结果：属性名自动成为变量名

属性不用起别名

例：

var time={ year:2018, month:11, date:6 };

//原简写方式 ↓ ↓ ↓

//var {year:y, month:m, date:d}=time;

//ES6推荐简写，当属性名和变量名相同，只用写一个

//即

var {year,month,date}=time;

console.log(year);

console.log(month);

console.log(date);

#### 参数解构

其实就是对象解构在函数中的应用

代替apply打散数组或对象中的成员值为单个值，再分别传入。

何时：只要不确定参数的个数和顺序

如何：2步

1. 定义函数时：将参数定义在对象结构中

function fun({形参1,形参2,...}){...}

1. 调用时：

fun({

形参1:值1,

形参2:值2,

...

})

其中，变量名和参数名必须保持一致，但顺序和个数不必一致。

优点：调用时，参数的个数和顺序与函数个数定义时无关。

例：

//users.js

var user={

signin:function(){

console.log("登录...");

},

islogin:function(){

console.log("判断登录状态...");

},

signout:function(){

console.log("注销...");

}

}

//app.js

var {islogin,signout}=user;

islogin();

signout();

例：

function ajax({ //将参数定义为对象

url,//url:url, //必须

type,//type:type, //必须

data,//data:data, //可选

dataType//dataType:dataType, //可选

}){

console.log(`向${url}发送${type}请求`);

if(data!=undefined){

console.log(`携带参数${data}`);

}

if(dataType==="json"){

console.log("将响应结果的json字符串自动转为js对象");

}

}

ajax({ //将参数定义为对象

url:"http://localhost:3000/users/signin",

type:"post",

data:"uname=dingding&upwd=123456"//data,

//dataType:undefined

})

ajax({ //将参数定义为对象

url:"http://localhost:3000/index",

type:"get",

//data:undefined,

dataType:"json"

})

ajax({ //将参数定义为对象

url:"http://localhost:3000/users/signout",

type:"get",

//data:undefined,

//dataType:undefined

})

### for of简化遍历

#### 遍历（索引）数组：3种

1. 最基础，最灵活，语法稍麻烦

for(var i=0;i<arr.length;i++){

arr[i]

}

1. 在不会改变遍历顺序时

arr.forEach((elem,i,arr)=>{

elem 或者 arr[i]

})

1. 在不关心下标的时，对所有元素都做同一件事时使用

for of 会自动取出每个元素值，且只认数字下标的索引数组

for(var elem of arr){

elem

}

注：for of只能遍历数字下标

for of用来代替for循环

for in是专门用来遍历对象用的

#### for of vs for in

|  |
| --- |
| for of专门遍历索引数组或类数组对象——对象都是数字  返回遍历的元素的值  for in专门遍历关联数组或对象——下标是无规则的自定义名称（即认识一切下标）  返回遍历元素的下标（自定义下标） |

### class

#### 对对象直接量的简化

var obj={

属性名,//属性名:变量名,如果属性名和变量名相同

方法名(){...}//省略:function

}

#### class

封装构造函数和原型对象方法的程序结构

如何：

1. 用class{}包裹原来的构造函数和原型对象
2. 将构造函数名提升为类型名

将所有构造函数名去掉function,改为constructor

1. 放在class中的"函数名(){...}"自动就是原型对象方法

简写：放到clss中的原型对象方法，不需要function。

注：写在class中的方法，默认都是放在原型对象中的实例方法

例：

原函数：

function Student(sname,sage){

this.sname=sname;

this.sage=sage;

}

Student.prototype.intr=function(){

console.log(`I'm ${this.sname},I'm ${this.sage}`);

}

var lilei=new Student("Li Lei",11);

console.log(lilei);

简写：SE6写法,class封装

class Student{//构造函数

constructor (sname,sage){

this.sname=sname;

this.sage=sage;

}

intr(){//原型对象

console.log(`I'm ${this.sname},I'm ${this.sage}`);

}

}

var lilei=new Student("Li Lei",11);

console.log(lilei);

#### 继承extends

让一个子类型继承父类型

如何：

class子类型extends父类型{ //super class

构造函数(){

super(...) //调用父类型构造函数

}

}

强调：不用setPrototypeOf了

重写时：可用super.方法()，来调用父类型的方法

例：

//原函数

function Enemy(fname,speed){

this.fname=fname;

this.speed=speed;

}

Enemy.prototype.fly=function(){

console.log(`${this.fname} 以时速 ${this.speed} 飞行！`);

}

function Plane(fname,speed,score){

//Enemy(fname,speed);//因为Enemy调用时前面没有.，则会导致其指向window，从而使得其参数泄露至全局作用域

Enemy.call(this,fname,speed);//call将替换this的对象，后面紧跟着是要给函数的参数值

//此处的this是Plane中的，而Plane是new出来的，所以其this始终是对的

this.score=score;

}

Plane.prototype.getScore=function(){

console.log(`击落 ${this.fname} 得 ${this.speed} 分`);

}

Object.setPrototypeOf(

Plane.prototype,Enemy.prototype

);

var f16=new Plane("F16",1000,5);

f16.fly();

f16.getScore();

console.log(f16);

//class封装结构函数和原型对象

class Enemy{

constructor (fname,speed){

this.fname=fname;

this.speed=speed;

}

fly(){

console.log(`${this.fname} 以时速 ${this.speed} 飞行！`);

}

}

function Plane(fname,speed,score){

//Enemy.call(this,fname,speed);

Enemy.applu(this,arguments)

this.score=score;

}

Plane.prototype.getScore=funciton(){

console.log(`击落${this.fname}得${this.score}分！`);

}

Object.setPrototypeOf(

Plane.prototype,Enemy.prototype

);

var f16=new Plane("F16",1000,5);

f16.fly();

f16.getScore();

console.log(f16);

//继承

class Plane extends Enemy{ //Plane需继承父类型Enemy

constructor(fname,speed,score){

super(fname,speed,score)//Enemy.call(this,...)

//super class调用父类型构造函数

this.score=score;

}

getScore(){

console.log(`击落 ${this.fname} 得 ${this.speed} 分`);

}

}

var f16=new Plane("F16",1000,5);

f16.fly();

f16.getScore();

console.log(f16);

extends代替了Object.setPrototypeOf(Plane.prototype,Enemy.prototype);

#### 静态方法

不需要实例化对象，就可用类型名直接调用的方法。

class类名{

constructor(){...}

方法名(){...}//保存在原型对象中的实例方法

**static**方法名(){...}//静态方法

}

例：

class Product{

constructor (pid,pname){

this.pid=pid;

this.pname=pname;

this.getById=function(pid){

console.log(`将当前编号为:${pid}的一个商品`);//该函数放在构造函数，将传递给子级，由子级使用

}

}

**static** getById(pid){ //静态

console.log(`将当前编号为:${pid}的一个商品`);

}//该函数由放在原型对象，由构造函数使用，子级不使用

save(){ //实例

console.log(`将当前商品:${this.pname}保存到数据库中...`);

}

}

//按id查询一个商品：hetById（pid）——静态

//保存一个商品到数据库：save（）——实例

Product.getById(1001);

var macbook=new Product(1001,"苹果本");

macbook.save();

##### 原理

静态其实就是直接定义在构造函数上，谁都能用。

##### vs原型对象方法

1. 保存在原型对象中
2. 仅限于当前类型的子对象才能使用
3. 必须先实例化对象，再用对象调用

### 访问器属性

class类名{

constructor(){

this.数据属性=...

}

get访问器属性(){return this.数据属性}

set访问器属性(value){判断value，并赋值}

}

例：

原函数：

function Emp(age){

//this.\_age=undefined;

this.age=age;

}

Object.defineProperty(Emp.prototype,"age",{

get:function(){

return this.\_age;

},

set:function(value){

if(value>=18&&value<=65){

this.\_age=value;

}else{

throw new Error("age must be 18~65");

}

},

enumerable:true,

configurable:false

});

访问器属性：

class Emp{

constructor (age){

this.age=age;

}

get age(){

return this.\_age;

}

set age(value){

if(value>=18&&value<=65){

this.\_age=value;

}else{

throw new Error("that age must be 18~65");

}

}

}

var eric=new Emp(22);

console.log(eric);

eric.age=-2;

//new Emp(-2);

### Promises（重点）

专门解决回调地狱：callbackhell

什么是回调地狱：因为回调函数嵌套层级太深导致代码可读性差。

什么是：保证多个异步函数，必须顺序执行的一种机制

为什么：使用回调函数参数方式，虽然可实现多个异步函数顺序执行，但语法形式上会形成"回调地狱"现象

如何：

例：异步函数按照先后顺序执行（语法上形成回调地狱）

function liang(task){

console.log("成亮起跑...");

setTimeout(function(){

console.log("成亮到达终点！");

if(task!==undefined)

task();

},6000)

}

function ran(task){

console.log("李然起跑...");

setTimeout(function(){

console.log("李然到达终点！");

if(task!==undefined)

task();

},4000)

}

function dong(task){

console.log("东东起跑...");

setTimeout(function(){

console.log("东东到达终点！");

if(task!==undefined)

task();

},2000)

}

//将函数串起来

liang(function(){//6s后调用task()

ran(function(){//4s后调用task()

dong(function(){

console.log("比赛结束！");

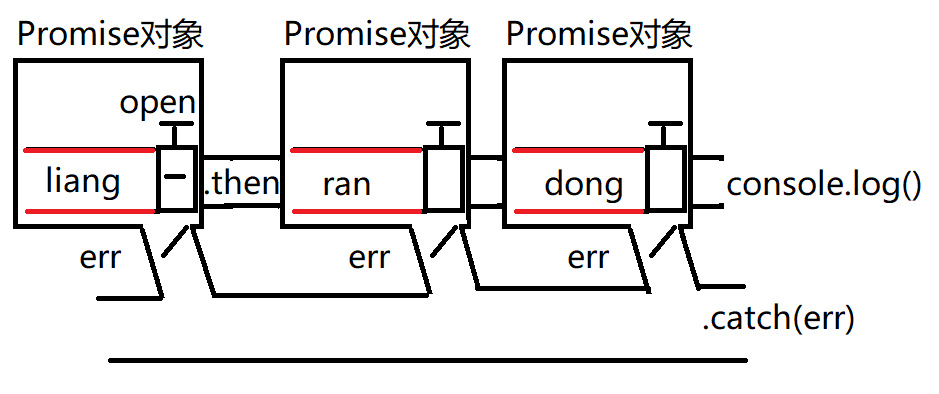
});//2s后执行完

});

});

三步：

1. 原函数的代码包裹在function（盒子）中
2. 该function（盒子）拥有open、err两个开关，
3. 同时该function（盒子）拥有then可接下一个盒子



#### 定义函数时

function函数名(){//不需要回调函数参数

return new Promise(

function(open){

//原函数的内容

//在原函数异步执行完成后，自动调用open()

}

)

}

#### 串联前后两个函数

调用前一个函数，返回Promise对象

前一个函数() //return new Promise()

.then(后一个函数) //不要加()，因为不是立刻调用！

事件：open->.then

错误处理：err->.catch

1. 定义时：

function fun(){

return new Promise(

function(open,err){

//原函数的内容

//在原函数异步调用完成后，自动调用open()

//如果出错则打开err("错误消息")

}

)

}

1. 调用时：

前一个函数()

.then(后一个函数)

...

.catch(function(err){错误处理代码})

等待多个异步任务完成才执行：

Promise.all([

任务函数1(),//return new Promise()

...(),

...

]).then(function(){...})

多个函数需要连续调用，每个函数，都要返回Promise对象，才能连续用the多个函数需要连续调用，每个函数，都要返回Promise对象，才能连续用then调用

例：

function liang(){

return new Promise(//open开关

function(open,err){//原任务

console.log("成亮起跑...");

setTimeout(function(){

if(Math.random()<0.5){

console.log("成亮到达终点！");

open();

}else err("成亮摔倒了");

},6000);

}

);

}

function ran(){

return new Promise(//open开关

function(open,err){//原任务

console.log("李然起跑...");

setTimeout(function(){

if(Math.random()<0.5){

console.log("李然到达终点！");

open();

}else err("李然摔倒了");

},4000);

}

);

}

function dong(){

return new Promise(//open开关

function(open,err){//原任务

console.log("东东起跑...");

setTimeout(function(){

if (Math.random()>0.5){

console.log("东东到达终点！");

open();

}else err("东东摔倒了");

},2000);

}

);

}

liang()//return new Promise;

.then(ran)//return new Promise;

.then(dong)//return new Promise;

.then(function(){console.log("比赛结束");})

.catch(function(err){console.err(err);})

#### promise延伸

Promise构造函数只有一个参数，该参数是一个函数，被称作执行器，执行器有2个参数，分别是resolve()和reject()，一个表示成功的回调，一个表示失败的回调。

记住，Promise实例只能通过resolve或者reject函数来返回，并且使用then()或者catch()获取，不能在new Promise里面直接return，这样是获取不到Promise返回值的。

##### promise.all

Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。

##### Promise.race()

race根据传入的多个Promise实例，只要有一个实例resolve或者reject，就只返回该结果，其他实例不再执行。

## ES7

将必须顺序调用的多个异步任务都封装在一个async function自调中

在async function中，可用await关键词设置阻塞方式运行异步任务函数

如果可能出错：try{...}catch(errMsg){...}

例：

function liang(){

return new Promise(

function(open,err){

console.log("成亮起跑...");

setTimeout(function(){

if(Math.random()<0.5){

console.log("成亮到达终点！");

open();

}else err("摔倒了！");

console.log("成亮到达终点！");

open();

},Math.random()\*6000);

}

);

}

function ran(){

return new Promise(

function(open,err){

console.log("李然起跑...");

setTimeout(function(){

if(Math.random()<0.5){

console.log("李然到达终点！");

open();

}else err("摔倒了！");

},Math.random()\*4000);

}

);

}

function dong(){

return new Promise(

function(open,err){

console.log("东东起跑...");

setTimeout(function(){

if(Math.random()<0.5){

console.log("东东到达终点！");

open();

}else err("摔倒了！");

},Math.random()\*2000);

}

);

}

(async function(){

try{

await liang() //return new Promise()

await ran() //return new Promise()

await dong() //return new Promise()

console.log("比赛结束")

}catch(errMsg){

console.error(errMsg)

}

})();

### 小知识

#### id属性值

id属性的值，会自动转为全局变量，不需查找，可直接获得元素对象。

一般不见于项目中，仅在案例中快速演示或测试时使用。

#### 函数名是变量

函数名其实仅是引用函数对象的一个普通变量

#### 参数赋值问题

不见到等号就不会赋值

#### concat

定义和用法

concat() 方法用于连接两个或多个数组。

该方法不会改变现有的数组，而仅仅会返回被连接数组的一个副本。

语法

arrayObject.concat(arrayX,arrayX,......,arrayX)

取值：

arrayX 必需。该参数可以是具体的值，也可以是数组对象。可以是任意多个。

返回值：

返回一个新的数组。该数组是通过把所有 arrayX 参数添加到 arrayObject 中生成的。如果要进行 concat() 操作的参数是数组，那么添加的是数组中的元素，而不是数组。

#### 从局部到全局作用域的顺序，查找变量所在位置

只要找到，就不再继续查找

如果找到全局都没有，就报错

在任何位置，为一个未声明的变量赋值，都会在全局创建该变量。

#### 关于对象与数组的比较

js中一切对象底层都是关联数组

|  |  |
| --- | --- |
| 相同 | 不同 |
| * obj.属性名等效于obj["属性名"]   只要访问对象的属性时，属性名是变化的变量，就只能用[变量]   * for in遍历每个成员   for(var key in obj){  key//当前属性名  obj[key]//当前属性值  }   * 访问不存在的属性，不报错，返回undefined * 强行给不存在的属性赋值   不报错，自动添加该属性 | 类型不同，API不通用 |

# DOM & BOM

|  |  |
| --- | --- |
| BOM | DOM |
| Browser Object Model  浏览器对象模型，用来访问和操纵浏览器窗口，是JS 有能力与浏览器交互。  通过使用BOM，可移动窗口，更改状态栏文本，执行其他不与页面内容发生直接联系的操作  没有相关标准，但被广泛支持 | Document Object Model  文档对象模型，用来操作当前HTML文档的内容  定义了访问和操作HTML文档的标准方法  通过对DOM树的操作，实现对HTML文档数据的操作  W3C制定了相关标准 |

## DOM

### 什么是DOM

Document Object Mode

专门操作网页内容的API标准——W3C制定

为什么：为了统一不同浏览器操作网页内容的API标准

何时：

1. 只要操作网页内容，都用DOM API；
2. 几乎所有浏览器100%兼容DOM API；
3. IE8比DOM标准出现的早，所以兼容性有问题。

包括：增，删，改，查，事件绑定

### DOM Tree

什么是：内存中保存所有网页内容的树形结构

为什么：树形结构是最直观的保存上下级包含关系的结构

如何：

1. 当浏览器加载一个html网页时，会先创建树根节点document。将来所有网页内容都是document下的后代节点
2. 浏览器每读取到HTML中的一项内容(元素，文本，属性)，就会创建一个节点对象，并将节点对象挂载到DOM树上。

#### 节点对象

节点对象：网页中每项内容都是DOM树上的一个节点对象

所有节点对象都有三个共同的属性：

1. nodeType:

节点类型：值为一个数字

何时：判断节点类型时

document 9

element 1

attribute 2

text 3

问题：无法进一步区分元素的标签

1. nodeName:

节点名：

何时：如果想进一步区分节点名称时

document #document

element 全大写的标签名

attribute 属性名

text #text

1. nodeValue:

节点值：

何时：几乎不用

document null

element null

attribute 属性值（通常不会把属性当做节点看待）

text 文本内容

### 查找

#### 不用查找就可直接获得

<html> document.documentElement

<head> document.head

<body> document.body

<form> document.forms[i]

#### 按节点间关系查找

何时：如果已经获得一个节点，找周围附近的节点时

如何：节点数，元素树

注：在页面上所有都是节点，但只有**带尖括号**的才是元素。

即：节点包含元素。

##### 节点数

节点树：保存所有网页内容的完整树结构

2大类

###### 父子关系

node.parentNode 获得node节点的父节点

node.childNodes 获得node下所有直接子节点的集合（类数组对象）

node.firstChild 获得node下第一个直接子节点

node.lastChild 获得node下最后一个直接子节点

###### 兄弟关系

node.previousSibling 获得node的前一个平级兄弟节点

node.nextSibling 获得node的后一个平级兄弟节点

问题：包含看不见的空字符，干扰查找

一般不建议使用，容易被坑死

##### 元素树

元素树：仅包含元素节点的树结构

说明：元素树不是一棵新树，仅仅是节点树上的子集

2大类

###### 父子关系

node.parentElement 获得node的父元素

node.children 获得node下所有直接子元素的集合(类数组对象)

node.firstElementChild 获得node下第一个直接子元素

node.lastElementChild 获得node下最后一个直接子元素

###### 兄弟关系

node.previousElementSibling 获得node的前一个平级兄弟元素

node.nextElementSibling 获得node的后一个平级兄弟元素

###### 鄙视：递归遍历指定父节点下的所有后代节点

2步：

1. 先输出直接子节点
2. 对每个直接子节点递归调用和父节点完全相同的函数

例：遍历父元素下所有子元素

//1.定义仅查找直接子元素的函数

function getChildren(parent){

var children=parent.children;

for(var elem of children){

console.log(elem.nodeName);

//2.对每个直接子元素调用和父元素相同的处理函数

arguments.callee(elem);

}

}

window.onload=function(){

getChildren(document.body);

}

注：arguments.callee调用自身

|  |
| --- |
| 所有DOM操作的固定四步:   1. 查找触发事件的元素 2. 绑定事件处理函数 3. 查找要修改的元素   处理函数中的this，默认指向.前的元素   1. 修改元素 |

#### 按HTML特征查找（4种）

##### 按id查找

var elem=document.getElementById("id");

返回值：一个元素对象

强调：只能用document调用

##### 按标签名查找

var elems=parent.getElementsByTagName("标签名")

返回值：符合条件的子元素的集合（类数组对象）

强调：

1. 可用任何父元素调用，限制查找的范围；
2. 不仅查找直接子元素，还查找所有后代。

##### 按name属性查找

var elems=document.getElementsByName("name")

返回值：多个元素对象的集合

强调：只能用document调用

注：表单元素才有name属性，如单选、多选按钮

##### 按class属性查找

var elems=parent.getElementsByClassName("类名")

返回值：多个元素对象的集合

强调：

1. 可在任意父元素上调用，缩小查询范围
2. 不仅查找直接子元素，还查找所有后代。
3. 如果一个元素被多个class修饰时，只用其中一个class名就可找到该元素。

问题：按HTML查找，一次只能按一个条件查找元素

#### 按选择器查找

##### 如果只找一个元素

var elem=parent.querySelector("选择器")

##### 如果找多个元素

var elems=parent.querySelectorAll("选择器")

兼容性：选择器的兼容性，依赖于浏览器的限制。

#### 鄙视:按HTML查找 vs 按选择器查找

1. 返回值

按HTML查找返回“动态集合”

动态集合：不实际存储属性值，每次访问集合内容，都重新查找DOM树

按选择器查找返回“非动态集合”

非动态集合：实际存储所有属性值，每次访问集合内容，不会导致重复查找DOM树

1. 效率

首次查找：

按HTML效率高

按选择器效率低

1. 易用性

按HTML查找，繁琐！

按选择器查找，简单！

总结：

如果只用一个条件就可查找到想要的元素时，首选按HTML查找；

如果查找条件复杂时，首选按选择器查找。

### 修改

#### 内容

获取或修改原始HTML代码片段：elem.innerHTML

获取或修改纯文本内容：elem.textContent

比innerHTML多做2件事：

1. 去掉内嵌标签
2. 转义字符翻译为正文

获取或修改表单元素的值：elem.value

#### 属性（3种）

##### HTML标准属性

HTML标准中规定的属性

2种

###### 用核心DOM API

最初DOM标准制定时的API（4个）

elem.getAttribute("属性名");

elem.setAttribute("属性名","新值")

elem.hasAttribute("属性名")

elem.removeAttribute("属性名")

###### HTML DOM API

专门支持HTML内容的简化版API

HTML DOM已经将所有标准属性做到了内存中的元素对象上，只不过默认值为""

只要：elem.属性名

特例：<ANY class="...">

↑

elem.className

##### 状态属性

enabled、disabled、selected、checked

共同特点：

1. 都是bool类型的值
2. 不能用核心DOM访问！
3. 只能用.访问

选择器（C3独有的）： :checked、:selected、:disabled、:enabled

##### 自定义扩展属性

什么是：HTML标准中没有规定的，程序员自己添加的属性

何时：2个

1. 在客户端缓存业务数据
2. 代替id,class,元素选择器，用于为元素添加行为

如何：

###### 添加

强烈建议:<ANY data-属性名="值"

js中：2种

1. 核心DOM API(4个)

不能用**.**访问（自定义扩展属性），因为是后添加的属性，无法提前保存在内存中的元素对象上

例：

a1.getAttribute("data-toggle")

//"tab"

简写

a1.dataset.toggle

//"tab"

1. HTML5标准

所有data-\*的自定义扩展属性，都可用

elem.dataset.属性名

配套，查找时使用属性选择器

例：

btn=document.querySelectorAll("li>[data-toggle=tab]")

<li><\* data-toggle="tab">

例：

//1.

var as=document.querySelectorAll(

"#tab>li>[data-toggle=tab]"

);

console.log(as);

//2.

for (var a of as){

a.onclick=function(){

var a=this;

//3.

var id=a.getAttribute("data-id");//#content1

//a.dataset.id;

var div=document.querySelector(id);

var divs=document.querySelectorAll(

"#container>div"

);

//4.

for (var d of divs){

d.style.zIndex="";

}

div.style.zIndex=10;

}

}

#### 样式

修改内联样式

elem.style.css属性="值"

强调：css属性名，要去-变驼峰

获取样式：

1. 不能用elem.style获取样式，因为仅包含内联样式
2. 必须获得计算后的样式：最终应用到当前元素上的所有样式的总和。

如何：2步

1. 获得计算后的样式的集合:

var style=getComputedStyle(元素对象)

1. 获得css属性值:

style.css属性值

强调：计算后的样式

1. 将属性值都计算为数值
2. 将相对单位换算为绝对单位
3. 只读!属性的来源不确定

问题：elem.style.css属性名一次只能修改一个css属性

解决：企业项目中，都是用class来批量修改元素的样式

例：

//1.

var txtName=document.getElementsByName("username")[0];

var txtPwd=document.getElementsByName("pwd")[0];

//2.

txtPwd.onfocus=txtName.onfocus=function(){

var txt=this;

//3.

var div=txt.parentNode.nextElementSibling.children[0];

//4.

div.classNmae="";

}

//2.

txtName.onblur=function(){

vali(this,/^\w{1,10}$/);

//var reg=/^\w{1,10}$/;

//var div=txt.parentNode.nextElementSibling.children[0];

//if(reg.test(txt.value)){

//div.className="vali\_success";

//}else{

//div.className="vali\_fail";

//}

}

//封装构造函数

function vali(txt,reg){

var div=txt.parentNode.nextElementSibling.children[0];

if(reg.test(txt.value)){

div.className="vali\_success";

}else{

div.className="vali\_fail";

}

}

txtPwd.onblur=function(){

vali(this,/^\d{6}$/);

//var reg=/^\d{6}$/;

//var div=txt.parentNode.nextElementSibling.children[0];

//if(reg.test(txt.value)){

//div.className="vali\_success";

//}else{

//div.className="vali\_fail";

//}

}

注：按name属性名查找：（document.getElementsByName），其返回的为一个类数组对象，若该类数组对象中只有一个对象，则需在数组后面加[0]。

### 添加/删除元素

#### 添加元素（3步）

##### 创建空元素

var a=document.createElement("a");

<a></a>

##### 设置关键属性

a.href="http://tmooc.cn"

a.innerHTML="go to tmooc";

<a href="http://tmooc.cn">go to tmooc</a>

##### 将新元素对象添加到DOM树上

parent.appendChild(a) //将a元素追加到parent下最后一个元素之后

parent.insertBefore(a,child) //将a元素插入到parent下的child子元素之前

parent.replaceChild(a,child) //用a元素代替parent下的child

例：

<div id="data">123</div>

<script>

//设置一个空的a标签

var a1=document.createElement("a");

//给a标签的href赋值为#

a1.href="#";

//给a标签添加元素

a1.innerHTML="lalalala";

console.log(a1);

//将a标签添加到位置

document.body.insertBefore(a1,data);

document.body.appendChild(a1,data);

//document.body.replaceChild(a1,data);

</script>

例：在上一个例子的div标签中添加一个table，并将json的内容写入

<script>

var json=[

{"ename":"Tom", "salary":11000, "age":25},

{"ename":"John", "salary":13000, "age":28},

{"ename":"Mary", "salary":12000, "age":25}

];

//创建table标签

var table =document.createElement("table");

// table标签添加在id="data"标签的子级

document.getElementById("data").appendChild(table);

// 创建thead标签

var thead=document.createElement("thead");

// 将thead标签挂载在table标签下

table.appendChild(thead);

// 建立tr标签

var tr=document.createElement("tr");

// 将tr标签挂载在thead下

thead.appendChild(tr);

// 将json的第一个值赋值给变量emp

var emp=json[0];

// 遍历emp

for(var key in emp){

// 使每遍历一次,都将得到json第一条的所有属性,或者称为下标

//创建一个th标签

var th=document.createElement("th");

//将该标签挂载在tr标签下

tr.appendChild(th);

// 在th标签下添加每次emp遍历的内容

th.innerHTML=key;

}

var th=document.createElement("th");

th.innerHTML="操作";

tr.appendChild(th);

//创建tbody

var tbody=document.createElement("tbody");

//添加到table下

table.appendChild(tbody);

//遍历每个员工

for(var emp of json){//for of遍历下标

var tr=document.createElement("tr");//创建tr

//tbody.appendChild(tr); //将tr追加到tbody下

tbody.insertBefore(tr,tbody.children[0]);

//遍历当前员工对象的每个属性

for(var key in emp){//for in遍历内容

var td=document.createElement("td");//创建td

tr.appendChild(td);//将td追加到tr下

td.innerHTML=emp[key];//由下标寻找值

}

var td=document.createElement("td");//创建td

var btn=document.createElement("button");

btn.innerHTML="x";

td.appendChild(btn);

tr.appendChild(td);//将td追加到tr下

}

//(json中后面的员工，反而出现在表格前)

document.getElementById("data").appendChild(table);

</script>

注意：该例题末尾对for in 循环的用法

###### 优化

优化：尽量减少操作DOM树的次数

原因：反复操作DOM树，会导致重排重绘

注：每建立一个元素都只能挂载到元素树上一次，

若挂载多个相同的元素，请使用遍历

###### 页面加载原理

HTML -> DOM Tree

↓

Render Tree ->\*\*\*layout\*\*\*->\*\*\*paint\*\*\*

↑

CSS -> COM

如何：2种情况

1. 如果同时添加父元素和子元素时，应先在内存中把子元素添加到父元素上，最后再将父元素一次性添加到DOM树
2. 如果父元素已经在页面上，应使用文档片段添加子元素

什么是：内存中，临时保存多个平级子元素的虚拟父元素

何时：只要向页面中添加多个平级子元素时

如何：3步：

1. 创建文档片段对象

var frag=document.createDocumentFragment()

1. 将子元素临时添加到文档片段中

frag.appendChild(子元素)

1. 将文档片段一次性挂载到DOM树

parent.appendChild(frag)

例：

<select name="provs">

<option>—请选择—</option><!--0-->

<option>北京市</option><!--1-->

<option>天津市</option>

<option>河北省</option>

</select>

<select name="cities" class="hide"></select>

<script>

/\*实现“省”和“市”的级联下拉列表\*/

var cities=[

[{"name":'东城区',"value":101},

{"name":'西城区',"value":102},

{"name":'海淀区',"value":103},

{"name":'朝阳区',"value":104}],

[{"name":'河东区',"value":201},

{"name":'河西区',"value":202},

{"name":'南开区',"value":303}],

[{"name":'石家庄市',"value":301},

{"name":'廊坊市',"value":302},

{"name":'保定市',"value":303},

{"name":'唐山市',"value":304},

{"name":'秦皇岛市',"value":304}]

];

//1.

var selProvs=document.getElementsByName("provs")[0];

//2.

selProvs.onchange=function(){

var selProvs=this;

//3.

var selCts=document.getElementsByName("cities")[0];

// 每一个select对象都有一个selectedIndex属性，可自动获得当前选中项的下标

var i=selProvs.selectedIndex;//获得当前选中的i

if(i>0){

var cts=cities[i-1];

// 创建文档片段对象frag

var frag=document.createDocumentFragment();

var opt=document.createElement("option");

opt.innerHTML="—请选择—";

frag.appendChild(opt);

for (var c of cts){

var opt=document.createElement("option");

opt.innerHTML=c.name,opt.value=c.value;

// 先将子元素加载到 文档片段对象frag 上

frag.appendChild(opt);

}

//4.

selCts.innerHTML="";

// 再将文档片段对象frag一次性挂载到selCts上

// 之后，该文档片段对象frag会自动释放

selCts.appendChild(frag);

selCts.className="";

}else{

selCts.className="hide";

}

}

</script>

#### 删除

parent.removeChild(child)

例：

<ul id="myList">

<li>Coffee</li>

<li>Tea</li>

<li>Milk</li>

</ul>

<script>

var list=document.getElementById("myList");

list.removeChild(list.childNodes[0]);

</script>

### HTML DOM常用对象:

核心DOM：针对一切结构化文档(XML，HTML)

优：万能

缺：繁琐

HTML DOM：针对HTML文档专门提供的简化API

优：简单

缺：不是万能——仅对部分常用的HTML元素提供了简化版API

#### Image对象

代表页面上一个<img>元素

唯一的简化：var img=new Image();

注：只有image和option能够new，其余的都不行

#### select/option

##### select对象

select对象：代表页面上一个<select>元素

1. 属性：

.selectedIndex 获得当前选中项的下标位置i

.value 获得当前选中项的值

如果选中的option有vaule属性，则返回value属性值，如果没有value属性，返回option的内容代替！

.options 获得当前select元素下所有option元素的集合。

.options.length 当前select元素下所有option元素的个数

.length => .options.length的简写

1. 方法：

.add(option) 向当前select元素中添加一个option

问题：不支持文档片段！

.remove(i) 移出当前selelct元素中i位置的一位option。

##### option对象

option对象：代表select元素下一个option元素

创建：var opt=new Option(内容,值)

创建一个新的option元素，并设置元素的内容为"内容"，设置元素的value属性为"值"

<select name="provs">

<option>—请选择—</option><!--0-->

<option>北京市</option><!--1-->

<option>天津市</option>

<option>河北省</option>

</select>

<select name="cities" class="hide"></select>

<script>

/\*实现“省”和“市”的级联下拉列表\*/

var cities=[

[{"name":'东城区',"value":101},

{"name":'西城区',"value":102},

{"name":'海淀区',"value":103},

{"name":'朝阳区',"value":104}],

[{"name":'河东区',"value":201},

{"name":'河西区',"value":202},

{"name":'南开区',"value":203}],

[{"name":'石家庄市',"value":301},

{"name":'廊坊市',"value":302},

{"name":'保定市',"value":303},

{"name":'唐山市',"value":304},

{"name":'秦皇岛市',"value":305}]

];

//1.查找触发事件元素

var selProvs=document.getElementsByName("provs")[0];//因为会返回一个类数组对象，所以设置[0]。且该元素内为option元素

//2.绑定事件触发函数

//onchange当（选中项）改变时，自动执行

selProvs.onchange=function(){

var selProvs=this;

var i=selProvs.selectedIndex;//i=selProvs的下标

//alert(i);

//3.查找要修改的元素

var selCts=document.getElementsByName("cities")[0];

if (i>0){

var arr=cities[i-1];//防止选择“—请选择—”而产生错误，因为“i-1”，当i=0时，即cities[-1] -> undefined

// 拿到子数组

//console.log(arr);

//4.修改元素

var frag=document.createDocumentFragment();

frag.appendChild(new Option("-请选择-"));

for(var c of arr){//of依次取出ar中每个元素的值

frag.appendChild(new Option(c.name,c.value));

}

selCts.innerHTML="";// 先移出之前的内容，在重新加载新内容

selCts.appendChild(frag);// 将文档片段挂载到selCts下

selCts.className="";// 每次都应先清除再添加

}else{

selCts.className="hide";

}

}

</script>

#### table/子元素

主机管理

##### table管着行分组

###### 添加行分组

var thead=table.createTHead()

相当于：document.createElement(“thead”)

table.appendChild(thead)

var tbody=table.createTBody();

var tfoot=table.createTFoot();

###### 删除行分组

table.deleteTHead()

table.deleteTFoot()

###### 获取

table.tHead

table.tBodies[i]

table.tFoot

##### 行分组管着行

###### 添加

var tr=行分组.insertRow(i)

相当于2句话：.createElement(tr)

行分组.appendChild(tr)

**固定用法**

在开头插入一行：行分组.insertRow(0)

在末尾追加一行：行分组.insertRow()

默认表示末尾追加

###### 删除

行分组.deleteRow(i) //i要求是行分组内的相对下标位置

问题：无法自动获得行在行分组内的相对下标位置

解决：tr.rowIndex可自动获得行在整个表内的下标位置

问题：行分组.deleteRow()不能用tr.rowIndex

解决：table.deleteRow(tr.rowIndex)

注：为删除整个table表中的最后一个tr

###### 获取

行分组.rows[i]

##### 行管着格

###### 添加

var tr=tr.insertCell(i) //问题：只能创建td, 不能创建th

相当于：.createElement(td)

.appendChild(tr)

**固定套路**：tr.insertCell() //末尾追加

###### 删除

tr.deleteCell(i) //极少用

###### 获取

tr.cells[i]

例：

<div id="data"></div>

<script>

var json=[

{"ename":"Tom", "salary":11000, "age":25},

{"ename":"John", "salary":13000, "age":28},

{"ename":"Mary", "salary":12000, "age":25}

];

var table=document.createElement("table");

var thead=table.createTHead();

var tr=thead.insertRow();

for(var key in json[0]){

//给行添加格，其内容为json中遍历的下标

tr.insertCell().innerHTML=key;

//td

}

//给行添加个，其内容为“操作”

tr.insertCell().innerHTML="操作";

var tbody=table.createTBody();

for (var emp of json){

var tr=tbody.insertRow(); //.insertRow(0)倒叙

for(var key in emp){

tr.insertCell().innerHTML=emp[key];//emp的值

//td

}

var btn=document.createElement("button");

btn.innerHTML="x";

tr.insertCell().appendChild(btn); //在tr后面挂载按钮（btn）

}

table.onclick=function(e){

//alert("ding!");

if(e.target.nodeName==="BUTTON"){

var btn=e.target;

var tr=btn.parentNode.parentNode;

//所按的按钮 td tr

var ename=tr.cells[0].innerHTML;

if(confirm("是否继续删除ename为："+ename+" 的条目吗？")){

// tr->所按的键的父级td的父级tr所在的位置

// 若不加.前tr作为具体限制限制，将删除行的最后一行

// confirm()为alert样式的弹出窗口的返回值，为bool属性

// 确定时，返回true；取消时，返回false

table.deleteRow(tr.rowIndex);

console.log(tr.rowIndex)

}

}

}

</script>

##### form/表单元素

###### HTML 表单标签

|  |  |
| --- | --- |
| 标签 | 描述 |
| [<form>](http://www.runoob.com/tags/tag-form.html" \t "http://www.runoob.com/html/_blank) | 定义供用户输入的表单 |
| [<input>](http://www.runoob.com/tags/tag-input.html" \t "http://www.runoob.com/html/_blank) | 定义输入域 |
| [<textarea>](http://www.runoob.com/tags/tag-textarea.html" \t "http://www.runoob.com/html/_blank) | 定义文本域 (一个多行的输入控件) |
| [<label>](http://www.runoob.com/tags/tag-label.html" \t "http://www.runoob.com/html/_blank) | 定义了 <input> 元素的标签，一般为输入标题 |
| [<fieldset>](http://www.runoob.com/tags/tag-fieldset.html" \t "http://www.runoob.com/html/_blank) | 定义了一组相关的表单元素，并使用外框包含起来 |
| [<legend>](http://www.runoob.com/tags/tag-legend.html" \t "http://www.runoob.com/html/_blank) | 定义了 <fieldset> 元素的标题 |
| [<select>](http://www.runoob.com/tags/tag-select.html" \t "http://www.runoob.com/html/_blank) | 定义了下拉选项列表 |
| [<optgroup>](http://www.runoob.com/tags/tag-optgroup.html" \t "http://www.runoob.com/html/_blank) | 定义选项组 |
| [<option>](http://www.runoob.com/tags/tag-option.html" \t "http://www.runoob.com/html/_blank) | 定义下拉列表中的选项 |
| [<button>](http://www.runoob.com/tags/tag-button.html" \t "http://www.runoob.com/html/_blank) | 定义一个点击按钮 |
| [<datalist>](http://www.runoob.com/tags/tag-datalist.html" \t "http://www.runoob.com/html/_blank) | 指定一个预先定义的输入控件选项列表（html5新增） |
| [<keygen>](http://www.runoob.com/tags/tag-keygen.html" \t "http://www.runoob.com/html/_blank) | 定义了表单的密钥对生成器字段（html5新增） |
| [<output>](http://www.runoob.com/tags/tag-output.html" \t "http://www.runoob.com/html/_blank) | 定义一个计算结果（html5新增） |

###### form对象

代表页面上一个<form>元素

获取：var form=document.forms[i]

属性：

.elements 获得form中所有表单元素的集合

.elements.length 获得表单中所有表单元素的个数

简写：.length => .elements.length

方法：.submit() 提交

###### 表单元素

获取：.elements[i/name]

可简写：如果表单元素有name属性，则可用name属性直接打.获得元素对象

form.name

方法：.focus() //移入焦点

.blur() //移出焦点

## BOM

### 什么是BOM

BOM：Browser Object Model

专门操作浏览器窗口的API——没有标准

包括：

window:2个角色：

1. 包裹所有DOM和BOM的API
2. 代替ES中的global充当全局作用于对象

包括：

1. history:
2. location:
3. document:
4. navigator:
5. screen:
6. event:

### window对象

常用属性

获得浏览器窗口大小：2组

完整窗口大小：outerHeight; outerWidth

文档显示区大小：innerHeight; innerWidth

### 打开和关闭新链接

4种：

#### 在当前窗口打开，可后退

Html：<a href=”url” target=”\_self”>

Js：open(“url”,”\_self”)

#### 在当前窗口打开，不可后退

Html：

Js：location.replace(“新url”)

原理：用新url代替history中旧url

#### 在新窗口打开，可打开多个

Html：<a href=”url” target=”\_blank”>

Js：open(“url”,”\_blank”)

#### 在新窗口打开，只能打开一个

Html：<a href=”url” target=”自定义窗口名”>

Js：open(“url”,”自定义窗口名”)

其实：浏览器中每个窗口都有唯一的name属性

浏览器规定，相同name属性的窗口只能打开一个

其实：<a>的target属性和open()的第二个参数，都是在指定新窗口的name值

内置窗口名：

\_self：自动获得当前窗口名，作为新窗口的名字

\_blank：不指定窗口名，让浏览器随机生成窗口名

### history

什么是：当前窗口打开后，保存当前窗口成功访问过的url的历史记录栈。

何时：用于控制前进，后退，刷新

如何：

history.go(i) .go(1) 前进一步

.go(-1) 后退一步 .go(-2) 后退2步

.go(0) 刷新

|  |  |
| --- | --- |
| **属性名** | **含义** |
| length | 返回浏览器历史列表中的URL数量 |
| **方法名** | **含义** |
| back() | 加载history列表中的前一个URL |
| forward() | 加载history列表中的下一个URL |
| go() | 加载history列表中的某个具体页面 |

### location

什么是：保存当前窗口正在打开的url的对象

何时：获取url中的信息，以及操作url时

属性：分段获得url中的各个部分信息

协议：.protocol

主机名+端口号：.host

主机名：.hostname

端口号：.port

相对路径：.pathname

锚点地址：.hash

查询字符串：.search

#### 功能

##### 在当前窗口打开新链接，可后退

location.href=”新url”——等效于直接修改浏览器地址栏

location.assign(“新url”)

##### 在当前窗口打开新链接，禁止后退

location.replace(“新url”)——替换history中当前url记录

##### 重新加载页面：刷**新（**2种）

1. 普通刷新：优先使用浏览器本地缓存中的资源

history.go(0)

F5

location.reload()

1. 强制刷新：跳过本地缓存，总是从服务端下载资源

location.reload(true) //强迫（force）

#### 总结（常用的属性和方法）

|  |  |
| --- | --- |
| **属性名** | **含义** |
| hash | 设置或返回从#开始的URL（锚） |
| host | 设置或返回主机名和当前URL的端口号 |
| hostname | 设置或返回当前URL的主机名 |
| href | 设置或返回完整的URL |
| pathname | 设置或返回当前URL的路径部分 |
| port | 设置或返回当前URL的端口号 |
| portocol | 设置或返回当前URL的协议 |
| search | 设置或返回从？开始的URL（查询部分） |
| **方法名** | **含义** |
| assgin() | 加载新的文档 |
| reload() | 重新加载当前文档 |
| replace() | 用新的文档替换当前文档 |

### navigator

什么是：保存浏览器配置信息的对象

何时：只要访问浏览器配置信息时

如何：

.cookieEnabled:

判断当前浏览器是否启用cookie

#### cookie

什么是cookie：在客户端本地持久保存客户私密信息的小文件

为什么：因为程序内存中的变量/数组/对象都是临时的！

原理：

程序员可通过js程序在客户端本地创建cookie文件，保存用户信息

每次请求网站时，都自动携带到服务端，服务端可通过服务端语言读取cookie中的内容

注：在未设置保存时间时，cookie的生命周期直到浏览器关闭为止。

#### .plugins

什么是：保存浏览器已经安装的插件信息的集合

何时：判断当前浏览器是否已经安装了某个插件

如何：

navigator.plugins[“插件名”]!==undefined

#### .userAgent

什么是：保存浏览器名称和版本号的字符串

何时：判断浏览器名称和版本号

如何：

document.write(navigator.userAgent+"<br>");

var browser,version,ua=navigator.userAgent;

if(ua.indexOf("MSIE")!=-1){

browser="IE"

}else if(ua.indexOf("Trident")!=-1){

browser="IE",version=11

}else if(ua.indexOf("Firefox")!=-1){

browser="Firefox"

}else if(ua.indexOf("Edge")!=-1){

browser="Edge"

}else if(ua.indexOf("OPR")!=-1){

browser="OPR"

}else if(ua.indexOf("Chrome")!=-1){

browser="Chrome"

}else if(ua.indexOf("Safari")!=-1){

browser="Safari"

}

document.write(browser+"<br>")//浏览器名称

if(version===undefined){

var i=ua.indexOf(browser)+browser.length+1;

version=parseInt(ua.slice(i,i+3));

}

document.write(version+"<br>")//浏览器版本号

### 定时器（2种）

#### 周期性定时器

周期性定时器：让程序每隔一段时间间隔就自动执行一项任务。

如何：3件事：

1. 任务函数：让定时器反复调用的函数
2. 启动定时器：var定时器序号=setInterval(任务函数,间隔ms)
3. 停止定时器：clearInterval(定时器序号)

2种：

用户手动停止定时器

自动停止定时器：在任务函数中判断临界值，一旦达到临界值，就自动调用clearInterval

例：

function calc(){

var end=new Date("2018/11/12 14:11:50");

var now=new Date();

var s=parseInt((end-now)/1000);

if(s>0){//当时间未到达临界时间时的操作（继续倒计时

var d=parseInt(s/3600/24);

if(d<10) d="0"+d;

//s/3600/24,再下取整

var h=parseInt(s%(3600\*24)/3600);

if(h<10) h="0"+h;

//s/(3600\*24)的余数,再/3600,再下去整

var m=parseInt(s%3600/60);

if(m<10) m="0"+m;

//s/3600的余数,再/60，再下取整

s%=60;//s/60的余数

if(s<10) s="0"+s;

//距离下一个假期还有: ?天?小时?分?秒

var span=document.getElementById("time");

span.innerHTML=d+"天"+h+"小时"+m+"分"+s+"秒";

}else{//当时间到达临界时间时的方法

var span=document.getElementById("time");

span.innerHTML="放学了";

clearInterval(timer);

}

}

calc();

var timer=setInterval(calc,1000)

//timer:1 2 3 4 5 6 ...

btn.onclick=function(){

clearInterval(timer);

}

#### 一次性定时器

一次性定时器：setTimeout

什么是：让程序延迟执行一项任务，后自动停止

如何：3件事

1. 任务函数
2. 启动定时器：var timer=setTimeout(任务函数,延迟ms)
3. 停止定时器：clearTimeout(timer)

例：

<div id="msg">

<a href="#">X</a>

</div>

<script>

//首先要在css中为msg添加transition属性

//当（整个页面）加载完成

window.onload=function(){

msg.style.bottom="-200px";

//一次性定时器

setTimeout(function(){

msg.style.bottom="0px";

},5000)

}

</script>

### event

什么是：浏览器默认触发的或人为触发的页面内容状态的改变

什么是事件处理函数：当事件发生时，自动调用的一个函数

何时：只要希望触发事件时，能自动执行一项任务

#### 如何绑定事件处理函数（3种）

1. 在HTML中绑定

<ANY on事件名=”js语句/函数调用()”

问题：不符合内容与行为分离的原则，不便于维护

1. 在js中，用赋值方式绑定

elem.on事件名=function(){

this->当前触发事件的元素对象

}

问题：一个事件，只能绑定一个处理函数

1. 在js中，添加事件监听对象

elem.addEventListener(“事件名”,处理函数)

事件监听对象：保存元素+事件名+处理函数的组合的一个对象

addEventListener其实是将一个事件监听对象添加到浏览器的事件队列中

浏览器触发事件监听对象，是通过遍历的方式查找并触发的。

例：

<button id="btnShoot">shoot</button><br>

<button id="btnAward">获得跟踪导弹</button><br>

<button id="btnBreak">失去跟踪导弹</button><br>

<script>

btnShoot.onclick=function(){

console.log("发射普通子弹·····");

}

//匿名函数

btnAward.onclick=function(){

//btnShoot.onclick=function{...}

btnShoot.addEventListener(

"click",

function(){

alert("发射导弹=>=>=>");

}

);

}

</script>

elem.removeEventListener(“事件名”,处理函数)

强调：移除处理函数时，必须找到原处理函数对象

例：

//具名函数

function shoot2(){

alert("发射导弹=>=>=>");

}

//若想要继续触发事件，则必须使用具名函数

btnAward.onclick=function(){

//btnShoot.onclick=function{...}

btnShoot.addEventListener(

"click",shoot2

);

}

btnBreak.onclick=function(){

btnShoot.removeEventListener(

"click",shoot2

);

}

#### 事件模型：3个阶段

捕获：由外向内，依次记录各级父元素上绑定的处理函数

目标触发：先触发目标元素上的处理函数：

目标元素：最初触发事件的元素

#### 冒泡

冒泡：由内向外，依次触发各级父元素上绑定的事件处理函数

##### 事件对象

什么是：事件发生时自动创建的，保存事件信息的对象。

何时：

1. 获得事件的信息
2. 修改事件的行为

如何：获得事件对象：事件对象e总是作为处理函数的第一个参数自动传入！

function(e){ //e 自动获得事件对象 }

方便动态监听新添加的元素。

##### 取消事件

取消冒泡：e.stopPropagation()

利用冒泡：

优化：尽量减少事件监听的个数！

原因：浏览器是采用遍历方式查找监听对象并触发事件。所以，事件监听对象的个数，决定了遍历的速度。

何时：只要发现，多个子元素，需要绑定相同的处理函数时，必须利用冒泡

如何：只将处理函数在父元素上绑定一次即可

例：

<div id="d1">

<div id="d2">

<div id="d3">

</div>

</div>

</div>

<script>

d1.onclick=function(e){ //e->event

alert(this.id);

e.stopPropagation();

}

d2.onclick=function(){ //e->event

alert(this.id);

// e.stopPropagation();

}

d3.onclick=function(e){ //e->event

alert(this.id);

e.stopPropagation();

}

</script>

注：

在没有e.stopPropagation()时，时由内向外侵入。

即：点击最外层的d1时，只有d1弹出；

点击d2时，d2首先弹出，之后d1弹出；

点击d3时，d3首先弹出，之后依次d2，d1弹出。

在有e.stopPropagation()时

仅在d1和d3使用了阻止默认行为后：

点击最外层的d1时，只有d1弹出；

点击d2时，d2首先弹出，之后蔓延至外层的d1，d1弹出；

点击d3时，d3弹出，之后被阻止蔓延。

##### 2个难题

1. 如何获得目标元素

不能用this->父元素，改为e.target

1. 排除不想要的元素的干扰

阻止默认行为：e.preventDefault()

e.target不随冒泡而改变，始终指向子元素。

例：

<div id="keys">

<button>1</button>

<button>2</button>

<button>3</button>

<button>4</button><br>

<button>C</button>

<button>+</button>

<button>-</button>

<button>=</button>

</div>

<textarea id="sc" style="resize:none;width:200px; height:50px;" readonly></textarea>

<script>

keys.onclick=function(e){

//this->.前的触发事件的当前元素

if(e.target.nodeName=="BUTTON"){ //专门用于区别于需要触发的元素

var btn=e.target;

//查找id为sc的文本域对象，保存在变量sc中(可省略)

switch(btn.innerHTML){ //判断btn的内容

case "C": //如果是"C"

sc.value=""; //清空sc的内容

break;

case "=": //如果是"="

try{

//将sc的内容放入eval()中执行，将结果再保存回sc的内容中

sc.value=eval(sc.value)

}catch(err){

sc.value=err;

}

break;

default: //以上条件均不满足时，即其余6个按钮

//将当前按钮的内容追加到sc的内容结尾

sc.value+=btn.innerHTML;

}

}

}

</script>

##### 典型应用场景

1. 用a当按钮用时，阻止添加#xxx到地址栏；

a.onclick=funciton(e){e.preventDefault()}

1. 用submit按钮时，也想阻止表单提交

form.onsubmit=function(e){...e.preventDefault()}

1. 使用HTML5的拖拽API时（下个老师讲）

##### 鼠标位置（3组）

1. 相对于屏幕左上角：e.screenX，e.screenY
2. 相对于文档显示区左上角：e.clientX，e.clientY
3. 相对于元素左上角：e.offsetX，e.offsetY

##### 页面滚动

事件：window.onscroll=funciton(){

获得页面滚动的距离：

var scrollTop=document.body.scrollTop || document.documentElement.scrollTop;

}

方法：用程序控制页面滚动到指定位置

window.scrollTo(left,top) 让页面滚动到top的位置

window.scrollBy(left增量,top增量) 让页面滚动过指定距离

# jQuery

## 概述

### 什么是jQuery

jQuery是第三方的，执行DOM操作的，极简化的函数库

第三方:必须先下载，才能使用

执行DOM操作:学习jQuery还是在学习DOM操作

极简化:jQuery是对DOM API的终极简化

函数库:jQuery中只有函数，没有属性。用函数来解决一切问题！



### 为什么

1. 简单：5个方面都进行了简化

增，删，改，查，事件绑定

动画

ajax

1. 解决了绝大多数兼容性问题:

凡是jQuery让用的！都没有兼容性问题。

何时:今后几乎所有项目和框架都是jQuery做的。

### 下载

官网：jquery.com

#### 版本

1.x支持旧浏览器

2.x不再支持旧浏览器

3.x不再支持旧浏览器，添加了新特性

##### 未压缩版

未压缩版：jquery-1.11.3.js

1. 具有完备的注释
2. 清晰的代码结构
3. 见名知义的变量名

优：可读性好

缺：体积大，下载慢，加载慢

何时：学习和开发阶段

##### 压缩版

压缩版：jquery-1.11.3.min.js

1. 删掉所有注释
2. 去掉所有代码格式(换行，缩进，空格)
3. 将变量名最简化

优：体积小，下载快，加载快

缺：可读性差！

何时：生产环境下

### 3.x的新特性（10个）

（标红的要记）

1. 移出就的IE工作区（从2.0起就不支持）
2. jQuery3.0所有代码都运行在Strict Mode（运行在严格模式下）
3. 引入了for...of循环（遍历）
4. 新的动画API：requestAnimationFrame()
5. 支持Promise（语法）
6. 对包含特殊含义的字符串提供转义的新方法
7. 类操作方法支持SVG
8. jQuery.when()对于多种参数的不同解读
9. 新的显示/隐藏逻辑
10. 对于防止XSS攻击的额外保护

### 引入网页（2种）

1. 将js文件保存在本地，通过相对路径引入本地js文件
2. 不要将js文件保存在本地，而是直接引用CDN网络上的js文件

<script src=”cdn网址”

#### 原理

##### 引入jquery.js文件时

其实是在全局定义一种新类型jQuery：

1. 构造函数：创建该类型的子对象

jQuery=function(…){

return new jQuery.fn.init(…) //自己不用new

}

window.$=window.jQuery=jQuery; //所以用$等于用jQuery

1. 原型对象：保存该类型子对象才能使用的共有方法

##### 使用jQuery简化API

想用jQuery简化版API必须先创建jQuery类型的子对象，又因为使用jQuery简化版API主要操作DOM元素，所以，创建jQuery对象时，需要找到要操作的DOM元素，封装进jQuery对象中。

总结：

1. 普通DOM对象不能使用jQuery简化版API
2. 想用jQuery API必须先创建jQuery子对象，并提供选择器用于查找到要操作的DOM对象，封装进来。

var $elem=$("选择器");

1. 对jQuery对象调用API，等效于对内部封装的DOM元素调用对等的API

var $elem=$(DOM元素)

###### 创建jQuery类型子对象(2种)

var $xxx=$(“选择器”)

var $xxx=$(DOM元素对象) //this, e.target…

### jQuery API的三大特点

1. 一个API两用：如果没给新值，默认就读取。如果给了新值，就变为修改。
2. 自带遍历：对jQuery对象调用一次API，等效于对jQuery对象内所有DOM元素，都调用一次API。
3. 几乎所有API，都返回正在使用的jQuery对象本身

如果对同一个元素执行多个操作时，可在上一个API之后，链式调用下一个API。

### 查询

查询：用选择器查询

jQuery支持所有css选择器，并少量添加了部分新选择器

var $elems=$(“选择器”)

#### 基本选择器：同CSS

#id .class element \* selector1,selector2,…

ID选择器、类选择器、元素选择器、通配符选择器、复合选择器

#### 层次选择器：同CSS

父 后代 父>直接子 兄+弟 兄~之后所有

#### 子元素过滤：同CSS

选择各个父元素内，相对位置上的子元素

:first-child :last-child :nth-child(index/even/odd) :only-child

#### 基本过滤(位置过滤)：jQuery新增

先查询出符合条件的元素，集中放在一个集合中，在集合中获取指定位置的元素！

:first/last :even/odd :gt/lt/eq(i)

:first 第一个元素

:last 最后一个元素

:even 选择偶数值的元素

:odd 选择奇数值的元素

:eq() 选择器选取 index 值等于指定数的元素。

:gt() 选择器选取 index 值高于指定数的元素。

:lt() 选择器选取带有小于指定 index 值的元素。

filter() 方法允许您规定一个标准。不匹配这个标准的元素会被从集合中删除，匹配的元素会被返回。

:not() 方法返回不匹配标准的所有元素。

提示：not() 方法与 filter() 相反。

索引和index都是由0开始

#### 属性过滤选择器：同CSS

[属性名]

[属性名=值]

[属性名^=值]

[属性名$=值]

[属性名\*=值]

[属性名!=值]

[属性名1][属性名2]...

选择器中的“而且”：两个选择器连用，中间不加任何分割

#### 内容过滤选择器：jQuery新增：

:contains(文本) 根据元素内容中是否包含指定“文本”来选择元素

:has(selector) 根据元素内子元素的特征来选择父元素

:parent 选择器选取所有包含子节点或文本节点的父元素

:empty 选择内容为空的元素（空元素指的是不包含子元素或文本的元素）

#### 可见性过滤：

:visible 匹配所有可见元素

:hidden 匹配所有不可见元素，或type为hidden的元素

通常:hidden只有2种情况：display:none和input type="hidden"

#### 表单元素选择器：jQuery新增

:input 可选择所有表单元素：input、select、textarea、button

:[type] 每种type都对应一种专门选择器：

:text 选取所有的单行文本框

:password 选取所有的密码框

:checkbox 选取所有的复选框

:radio 选取所有的单选框

:submit 选取所有的提交按钮

:reset 选取所有的重置按钮

:image 选取所有的图像按钮

:button 选取所有的按钮

:file 选取所有的上传域

:hidden 选取所有不可见元素

#### 状态过滤选择器：同CSS

:enabled 选取所有可用元素

:disabled 选取所有不可用元素

:checked 选取所有被选中的元素（单选框或多选框）

:selected 选取所有被选中的元素（下拉列表）

### 修改

#### 内容

1. 原始HTML片段： $("").html([“html片段”]) .innerHTML

读取或修改节点的HTML内容，类似于JavaScript中的innerHTML属性

1. 纯文本内容： $("").text([“纯文本”]) .textContent

读取或修改节点的文本内容，类似于JavaScript中的textContent属性

1. 表单元素： $("").val([“值”]) .value

读取或修改节点的value属性值，类似于JavaScript中的value属性

1. 清空元素： $("").empty()
2. 属性内容 $(""). ()

读取或者修改节点的属性

1. 删除属性内容 $("").removeAttr()

删除节点的属性

注：都是一个API两用：给新值，就修改；不给值，就获取

#### 属性

##### HTML标准属性:

$(…).attr(“属性名”,[”属性值”])

代替：getAttribute()/setAttribute()

可访问出现在元素开始标签上的所有属性

##### 状态属性

enabled disabled checked selected

$(…).prop(“属性名”,[”属性值”])

代替**.**访问所有内存中的属性！

##### 自定义扩展属性

$(…).attr(“属性名”,[”属性值”])

如果同时修改多个属性：

$(…).attr({

属性名:值,

… : …

})

例：

<h1>练习：查找元素&操作元素</h1>

<h3>使用附加导航(affix)实现内容切换</h3>

<div class="affix">

<ul>

<li><a data-toggle="item" href="#my-order">我的订单</a></li>

<li class="active"><a data-toggle="item" href="#user-setting">用户设置</a></li>

<li><a data-toggle="item" href="#account-security">账户安全</a></li>

</ul>

</div>

<div class="container">

<div id="my-order" class="box" style="background:#fee">我的订单的详细内容</div>

<div id="user-setting" class="box active" style="background:#efe">用户设置的详细内容</div>

<div id="account-security" class="box" style="background:#eef">账户安全的详细内容</div>

</div>

<script src="js/jquery-1.11.3.js"></script>

<script>

$(".affix>ul")

.on("click","a",function(){

var a=$(this);

a.parent().toggleClass("active")

.siblings("li").removeClass("active")

a.attr("href").toggleClass("active")

.siblings("div").removeClass("active")

// var i=a.index();

// $(".box").eq(i).toggleClass("active")

// .siblings("div").removeClass("active")

})

</script>

##### 鄙视：.attr vs.prop

attr可访问出现在元素开始标签上的所有属性，即提前保存在元素的对象中

对于HTML元素本身就带有的固有属性，在处理时，使用attr方法。

例：<a href="http://www.baidu.com" target="\_self" class="btn">百度</a>

href、target、class为固有属性（标准属性），可用attr访问，也可用prop访问，建议使用attr

prop代替.访问所有内存中的属性

对于HTML元素我们自己自定义的DOM属性，在处理时，使用prop方法。

例：<a href="#" id="link1" action="delete">删除</a>

href、id为固有属性（标准属性），建议用attr访问

action为自定义属性，建议用prop访问

状态属性（enabled、disabled、checked、selected）只能用prop访问

jquery中用attr()方法来获取和设置元素属性,attr是attribute（属性）的缩写，在jQuery DOM操作中会经常用到attr()，attr()有4个表达式。

1. attr( 属性名 ) //获取属性的值（取得第一个匹配元素的属性值。通过这个方法可以方便地从第一个匹配元素中获取一个属性的值。如果元素没有相应属性，则返回 undefined ）

2. attr( 属性名, 属性值 ) //设置属性的值 （为所有匹配的元素设置一个属性值。）

注：在原属性上加值，为：$("...").attr("属性名",[$("...").attr("属性名" + " 新增值"])

3. attr( 属性名 , 函数值 ) //设置属性的函数值 （为所有匹配的元素设置一个计算的属性值。不提供值，而是提供一个函数，由这个函数计算的值作为属性值。）

4. attr(properties) //给指定元素设置多个属性值，即：{属性名一: “属性值一” , 属性名二: “属性值二” , … … }。(这是一种在所有匹配元素中批量设置很多属性的最佳方式。 注意，如果你要设置对象的class属性，你必须使用'className' 作为属性名。或者你可以直接使用'class'或者'id'。)

#### 样式

##### 获取和修改

$(…).css(“css属性名”,[”属性值”])

$(…).css({

css属性:值,

… : …

})

##### 用class操作样式

.addClass(“class1 class2”) 加一个属性

.removeClass(“class”) 去掉一个属性

.removeClass() 去掉所有属性

.hasClass(“class”) 判断是否包含这个属性

.toggleClass(“class”) 在有和没有某个”class”之间切换

css("") 读取css的值

css("","") 设置多个样式

if($(…).hasClass(“class”))

$(…).removeClass(“class”)

else

$(…).addClass(“class”)

### 按节点间关系查找

#### 父子关系

$().parent() .parentNode

$().children([“selector”]) .children

$().find(“selector”) 可在所有后代中查找符合条件的

注：find()，的括号不能为空

$().children(":first-child") .firstElementChild

$().children(":last-child") .lastElementChild

#### 兄弟关系

$().prev() 之前的兄弟 .previousElementSibling

$().prevAll() 之前所有

$().next() 之后的兄弟 .nextElementSibling

$().nextAll() 之后所有

$().siblings([“selector”]) 除自己之外所有

### 添加/删除/替换/克隆

#### 用html片段创建新元素

var $elem=$(“html片段”)

#### 将元素添加到DOM树上

$parent.append($child) 末尾追加，返回$parent

.prepend($child) 开头插入，返回$parent

$child.appendTo($parent) 末尾追加，返回$child

.prependTo($parent) 开头插入，返回$child

$child.before($elem) 将$elem插入到$child之前

.after($elem) 将$elem插入到$child之后

#### 删除

$elem.remove(); 移除

$elem.remove(selector); 按选择器定位后删除

$elem.empty(); 清空节点

#### 替换

$旧.replaceWith($新) 返回$旧

$新.replaceAll($旧) 返回$新

#### 克隆

var $新=$旧.clone()

问题：默认是浅克隆：仅复制属性和样式，不复制行为

解决：$旧.clone(true) 深克隆：既复制属性和样式，又复制行为

### 事件绑定

#### jQuery共有几种事件绑定方式：

1. .bind/unbind() 用法和addEventListener()完全一样
2. .one() 用法和bind()一样，只不过，只能触发一次，就自动解绑。
3. .delegate() 简化版的利用冒泡——被on代替
4. .live/die() 早已废弃
5. .on/off: 2种：
6. 代替addEventListener()，用法完全一样
7. 专门用于简化利用冒泡：

$parent.on(“事件名”, ”selector” ,function(){

//on自动用”selector”，判断元素是否是想要的

//凡是进入函数内，都说明$tar一定是想要的

var $tar=$(this); //this->e.target

})

1. 部分简写(21个)：

$().事件名(function(){ //$().on(“事件名”,function(){ … })

})

冒泡中不建议使用。

#### 页面加载后自动执行：2种

##### 仅DOM内容加载后就提前执行：

仅等待HTML和JS，不必等待CSS和图片

何时：只要和CSS、图片无关的操作(事件绑定)，都应该在DOM内容加载后，就提前执行！

如何：DOMContentLoaded——有兼容性问题

$(document).ready(function(){ … })

简化：$().ready(function(){ … })

更简化：$(function(){ … })

##### 等全部页面内容都加载完才执行

必须等待HTML，JS，CSS，图片…

何时：只要依赖于CSS、图片的操作就只能放在整个页面加载后才能执行。

如何：window.onload=function(){ … } ——会有冲突

$(window).on(“load”, function(){ … })

总结：今后只要使用jQuery开发，几乎所有的代码都要放在$(function(){ … })中

##### 鄙视：jQuery中$的原理

$是jQuery类型的别名，也代表jQuery类型的构造函数用于创建jQuery类型的子对象。

4种用途:

1. 查找DOM元素并封装jQuery对象

$(“选择器”)

优化：

如果选择器简单: #id .class element 自动调用按HTML查找

如果选择器复杂才自动调用getComputedStyle()

1. 直接封装DOM元素为jQuery对象

$(DOM元素)

1. 创建新元素: $(“html片段”)
2. 绑定DOM内容加载后提前出发的事件

$(function(){ … })

#### 鼠标事件

.mouseover .mouseout

问题：频繁进出子元素，也会反复触发父元素上的处理函数——效率低！

解决：用.mouseenter .mouseleave代替

即使反复进出子元素，也不会反复触发父元素上的处理函数——效率高！

如果同时绑定mouseenter和mouseleave

可简写为：

.hover(

function(){ … }, //给mouseenter

function(){ … } //给mouseleave

)

如果两个处理函数可合并为一个，可进一步简写为:

.hover( function(){ … } )

//既给mouseenter

//又给mouseleave

#### 模拟操作

模拟触发：即使没有点击按钮，也能执行按钮的处理函数！

如何：$().trigger(“事件名”) 触发找到的元素上的指定事件的处理函数

可简写为：$().click() 一个函数两用

例：

<h1>与服务器进行通信的三步</h1>

<button id="bt0">连接到服务器</button>

<button id="bt1">给服务器发消息</button>

<button id="bt2">断开与服务器的链接</button>

<hr/>

<button id="bt99">一次性触发“连接-发消息-断开”三个处理函数</button>

<script src="js/jquery-1.11.3.js"></script>

<script>

$('#bt0').click(function(){

console.log('开始连接服务器....')

})

$('#bt1').click(function(){

console.log('开始向服务器发消息....')

})

$('#bt2').click(function(){

console.log('断开到服务器的连接....')

})

/\*$('#bt99').click(function(e){

//使用JS顺序触发前三个按钮的单击事件即可

})

\*/

$("#bt99").click(function(){

console.log(132)

//使用JS顺序触发前三个按钮的单击事件

//调用#bt0的处理函数

$("#bt0").trigger("click");

//调用#bt1的处理函数

$("#bt1").click();

//调用#bt2的处理函数

$("#bt2").click();

})

</script>

### 动画API

凡是在控制台，可以看见变化效果的，都是通过JS修改的；而看不见变化的，则是通过CSS来实现的。

#### 简单动画

写死的三种动画效果

##### 显示隐藏

.show() .hide() .toggle()

问题：在没有参数时，默认使用display:none/block实现瞬间显示隐藏。

解决：必须添加时间参数：.show(ms) .hide(ms)

##### 上滑下滑

.slideUp() .slideDown() .slideToggle()

##### 淡入淡出

.fadeIn() .fadeOut() .fadeToggle()

问题： 1.写死的动画效果，无法维护。

2.用js的定时器模拟的动画，效率不如CSS动画高

简单动画API，几乎都用css transition代替。

特例：.show() .hide() .toggle()不加参数时，是推荐使用的

注：在控制台中，能够看见变换过程的，均是js制作的；而看不见变化过程的则是css制作的。

例：

<h1>使用“高度动画”实现“手风琴”组件</h1>

<div class="accordion">

<div class="title">《西游记》简介</div>

<div class="content fade in">一个和尚和四个动物的故事: ?</div>

<div class="title">《水浒传》简介</div>

<div class="content fade">105个男人和三个女人的故事:!</div>

<div class="title">《红楼梦》简介</div>

<div class="content fade">一个男人和一群女人的故事: .</div>

</div>

<script src="js/jquery-1.11.3.js"></script>

<script>

$(".content:gt(0)").hide();

$(".accordion")

.on("click",".title",function(){

var $title=$(this);

//如果自己的兄弟是开着

if($title.next().is(":visible")){

$title.next().slideUp()//就关闭自己旁边的兄弟

}else{//否则

//将自己旁边的兄弟content打开

$title.next()//旁边的.content

.slideDown()//旁边的.content

//并将其余.content都关上

.siblings(".content")//旁边.content的兄弟中其余content

.slideUp();

}

})

</script>

#### 万能动画: 对任意css属性应用动画效果:

$().animate({

css属性: 目标值,

… : …

}, 动画持续时间)

问题：只支持单个数值的属性

#### 排队和并发

1. 多个属性并发变化

如何：一个animate内的多个css属性并发变化

1. 多个属性排队变化

如何：对一个元素反复调用多次animate是排队执行

#### 停止动画：$().stop()

问题：.stop()默认只能停止正在播放的一个动画

队列中其余动画依然执行

解决：.stop(true) 停止动画并清空队列！

选择器：:animated 选择正在播放动画的元素

##### 动画结束后自动执行

每个动画API都有最后一个参数: function(){ … } 会在动画结束后自动被调用！

例：

$("#s4").click(function(){

alert("ding");

$(this).animate(//异步

{

width:256,

height:256,

opzaity:0

},2000,function(){//会在动画播放结束后自动执行

$(this).hide();

})

})

### 类数组对象操作

length属性

查找jQuery对象的存储位置

var i=$(所选元素).index(要查找的一个元素)

简写：如果只在一个父元素内查找:

$(要查找的元素).index(); //获取要查找的元素都在其父元素内的下标位置

例：

var i=$as.index($a); // 原式

var i=$a.index(); // 简写，但要确定是否在一个父元素下

eq(index)：将下标等于index的DOM对象取出来

get()：返回DOM或Query对象在类数组中的下标

$(…).each( 遍历.前的jQuery对象中每个DOM元素

function(i,elem){ //会在每个DOM元素上自动调用一次

i -> 当前正在遍历的位置

elem -> 当前正在遍历的DOM元素

}

)

例：

$("#list>li").each(function(i,elem){

var $li=$(elem);

if($li.html()>=90){

$li.css("background","#009688")

}else if($li.html()<60){

var score=parseFloat($li.html());

score+=10;

$li.html(score);

}

})

$.each( //遍历任意一个类数组对象

任意类数组对象/数组,function(i,val){

静态方法

})

//jq 3.x中，$.each被 for of代替

例：

function add(){

var sum=0;

//for(var i=0;i<argument.length;i++)

//var val=arguments[i];

$.each(arguments,function(i,val){

sum+=val;

})

return sum;

}

console.log(

add(1,2,3,4), //10

add(1,2,3,4,5,6,7) //28

);

### 添加自定义API

在jQuery类型的原型对象中添加自定义API

例：

<script>

//自定义函数库myLib.js，为jQuery扩展sum函数，计算找到的所有元素内容的和。

//封装一个sum方法在jQuery

//jQuery.prototype

jQuery.fn.sum=function(){

var $elems=this; //this=$(ul>li)

var sum=0;

$elems.each(function(i,elem){

sum+=parseFloat(elem.innerHTML);

})

return sum;

}

console.log(

$("ul>li").sum()

);

</script>

问题：单机按钮后，每一行都变成了bold样式

原因：jQuery的选择器会匹配0/1、多个元素。我们在调用.hasClass()的时候，只会检查最先匹配的元素。

解决：独立检察和操作每个元素，在方法的环境上调用each方法，这样就会隐式迭代。

### \*\*\*\*\*封装自定义插件

插件：页面中可重用的独立功能

为什么：避免重复劳动

何时：只要一个功能可能被重复使用时

如何：

来源：3类

#### 官方插件（jQuery UI）

如何使用

1. 在页面中按插件要求编写html内容，不用加任何class
2. 在页面中引入插件的.css文件和插件的.js文件

前提：jQuery UI是基于jQuery实现的，所以必须先引jQuery.js，再引jQuery UI.js

1. 用jQuery查找要应用插件的父元素，调用插件同名函数

jQuery UI插件特点：侵入性

插件根据自身的需要，自动隐式添加class和自定义扩展属性

优点：极其简单

缺点：不可维护

Bootstrap插件特点：DIY

完全由程序员为元素指定class

缺点：繁琐！

优点：可维护

#### 第三方插件

#### 自定义插件

前提：自定义插件都不是从0开始做，都是将现有css和js提取到独立的文件中。

##### 定义jQuery UI风格的插件

###### 定义插件样式文件.css

直接复制插件相关的css到.css文件中

强调：一个插件内的选择器为了避免和其他插件中同名选择器冲突，都必须以当前插件名开头

###### 为jQuery添加插件API

自动侵入样式类和自定义扩展属性

绑定事件：使用jQuery UI风格的插件：同上

##### 定义Bootstrap风格的插件

1. 提取css到独立的.css文件中：
2. 在插件的js文件中，查找自定义扩展属性，直接绑定事件——要求使用者将来必须手动先添加自定义扩展属性

使用Bootstrap风格的插件：

1. 引入插件的css
2. 按插件要求编写HTML内容

手动添加所有必须的class和data-\*属性（自定义扩展属性）

//测试：有样式，没行为

1. 引入插件的js——自动查找data-\*的元素绑定事件

### 跨域

#### $.ajax()

$.ajax({

url:”...”,

type:”get/post”,

data:{变量:值,…}, //jQuery支持对象语法

dataType:“json”,

/\* \*/

success:function(res){ //jq1.x 2.x使用

//回调函数：在请求成功后自动执行

//res：就是服务端返回的响应结果

}

/\* \*/

}).then(res=>{ //jq3.x+支持

…

})

#### \*\*\*\*\*跨域访问

什么是跨域：一个域名下的网站，向另一个域名下的服务端发送请求。

包括：

1. 域名不一样：

<http://www.a.com> -> <http://www.b.com>

1. 二级域名不一样：

<http://hr.tedu.cn> -> <http://oa.tedu.cn>

1. 端口号不一样：

[http://localhost: 5500](http://localhost:3000) -> http://localhost: 3000

因为端口号是区分不同应用程序的唯一标示

操作系统不允许夸应用程序端口号发送请求

1. 协议不一样：

<http://tedu.cn> -> <https://tedu.cn>

http端口：80 https端口：443

1. 即使是本机：localhost -> 127.0.0.1 算跨域

问题：

只有ajax(xhr请求)禁止跨域：

报错：Access-Control-Allow-Origin

可以跨域：

img的src，link的href，scripe的src…

解决：2种

##### 服务端允许跨域——CORS跨域

CORS：cross-origin resource sharing

在服务端响应结果中，添加响应头:

"Access-Control-Allow-Origin":"<http://127.0.0.1:5500>"

划线处若如此改为"http://127.0.0.1:5500"，则可允许"http://127.0.0.1:5500"访问；

若改为"\*"，则可允许任意地址访问

注：本机虚拟服务器为5500端口，所以只能允许5500端口访问，若要测试，需要别的电脑辅助

例：在服务器.js文件中

const http=require("http");

http.createServer(function(req,res){

var data="七月七日 晴";

res.writeHead(200,{

"Content-Type":"text/plain;charset=utf-8",

// 在此加如下响应头可使http://127.0.0.1:5500访问成功

"Access-Control-Allow-Origin":"http://127.0.0.1:5500"

})

res.write(data);

res.end();

}).listen(8080);

// http://localhost:8080

// 127.0.0.1:8080

问题：每个服务端接口都要重复添加

解决：使用cors模块:

（在命令窗口中输入：npm i -save cors）

npm i –save cors

在服务器端app.js文件中：

const cors=require(“cors”)

//静态资源配置后

app.use(cors({

origin:”http://来源地址:端口号”不能用\*

}))

在服务器端的index.js文件中，就可以使用res.send()不用单独加响应头了

res.send(result);

// res.writeHead(200,{

// "Content-Type":"application/json;charse=utf-8",

// "Access-Control-Allow-Origin":"\*"

// });

// res.write(JSON.stringify(result));

// res.end();

结果：所有接口依然可使用res.send()

注意：一定要先安装模块cors

##### 客户端跨域：JSONP——JSON With Padding

例：该部分练习均在如下代码内进行修改，原代码如下：

index.html文件如下：

<script>

$.ajax({

url:"http://localhost:8080",

type:"get",

success:function(res){

document.write(res);

}

})

</script>

sever.js文件如下：

const http=require("http");

http.createServer(function(req,res){

var data="七月七日 晴";

res.writeHead(200,{

"Content-Type":"text/plain;charset=utf-8",

"Access-Control-Allow-Origin":"\*"

})

res.write(data);

res.end();

}).listen(8080,function(){

console.log("服务器启动成功")

});

###### 使用script src发送请求

前提：绝对不能用xhr请求

客户端：使用<script src发送请求

服务端：将要返回的数据，填充在一条可执行的js语句字符串中返回。

例：

在index.html文件中，除去$.ajax方法，改为

<script src="http://localhost:8080"></script>

在sever.js文件中，在res.write()中用模板字符串拼接一条语句，如下

const http=require("http");

http.createServer(function(req,res){

var data="七月七日 晴";

res.writeHead(200,{

"Content-Type":"text/plain;charset=utf-8","Access-Control-Allow-Origin":"http://127.0.0.1:5500"

})

// res.write(data);

// documeny.write('天气预报')

res.write(`document.write('${data}')`);

res.end();

}).listen(8080,function(){

console.log("服务器启动成功")

});

注：ajax对于跨域有限值，但script的src对于跨域没有限值，但要求必须是一条可执行的js语句

问题：服务端返回的js语句是写死的，众口难调

###### 将数据显示方式变活

客户端：先定义一个带参函数，用于处理服务端的数据

服务端：改js语句为函数调用语句并传参

例：

在index.html文件中，增加一个函数

function show(data){

alert (data);

document.write(data);

console.log(data);

}

在sever.js文件中，在res.write()中用模板字符串拼接一条语句，如下

res.write(`show('${data}')`);

问题：服务端的函数名是写死的

###### 将封装的函数名写活

客户端：发送请求时，带一个参数。参数是函数名

服务端：获得客户端发来的函数名

将函数名拼接到返回的函数调用语句中

例：

在index.html文件中，增加一个函数

function getWeather(data){

alert (data);

document.write(data);

console.log(data);

}

并在script的src中传递参数

<script src="http://localhost:8080?callback=getWeather"></script>

在sever.js文件中，引入模块url

const url=require("url");

在http.createServer(function(req,res){}中写入如下代码：

引入模块后，先将url编译为对象

req=url.parse(req.url,true);

// parse编译了req的url字符串，之后才会有req.query、req.body

// true 表示参数也转为对象

// 所以才可以req.query.callback、req.query.uname

之后读取出函数名

var callback=req.query.callback;

最后，在输出中将函数名修改为动态的

res.write(`${callback}('${data}')`);

可由http://localhost:5500/?callback=getWeather，进行验证

问题：script是写死的，无法动态发送请求

###### 动态创建script元素

客户端：按需动态创建script元素

问题：script元素会越积越多

解决：在回调函数结尾，删除body中最后一个script

例：

在sever.js文件中，加入按钮

<button id="btn">今日天气</button>

将法3的script src注释掉

以以下代码代替

<script>

$("#btn").click(function(){

$(`<script src="http://localhost:8080?callback=getWeather">`)

.appendTo("body");

})

</script>

注：由于jQuery语法中规定，在script标签内不得出现</script>标签，所以，可不用写结束标签，只写开始标签不影响其效果。

在http://localhost:5500?callback=getWeather，可访问。

在点击按钮后，可执行函数。

但由于只有加入没有撤销，使得网页中script不断增加

故在事件完成之后，增加如下一条代码，如下：

function getWeather(data){

alert (data);

document.write(data);

console.log(data);

// 4.法

$("body>script:last").remove();

}

###### 总结

由于jsonp请求十分的常用，所以jQuery已经封装了jsonp的方式：代码如下

$.ajax({

url:"http://localhost:8080",

type:"get",

dataType:"jsonp", // 该xhr请求为script请求

success:function(res){

document.write(res);

}

})

注意：该最终解决方案是在第四种方案的基础上修改而来的，即仍需前端-后端同时修改才能实现。但通常在企业中，后端均会写好jsonp的相关代码。从而，前端只需加dataType:"jsonp",便可达到需求。

# 框架（VUE）

## 框架概述

### 原生API vs 函数库 vs 框架

**原生API**：浏览器/平台已经实现，咱们可直接使用的

比如：ES, DOM, BOM

问题：繁琐

**函数库**：基于原生API，进一步封装简化的一组函数的集合

比如：jQuery

优点：简单

缺点：仅简化单个API，对生产流程没有优化

**框架**：前人将多次成功的项目经验总结起来，形成的半成品项目

优：从生产流程上避免重复操作和手工劳动

缺：需要改变观念，放弃旧的做事方法

框架将功能都封装起来，调试不方便

比如：VUE, AngularJS, React

## VUE概述

什么是：渐进式的，基于MVVM的纯前端js框架

渐进式：虽然提供了很多组件，但可有选择的使用，还可跟其他技术混用。

相对：“全家桶”要用就必须使用所有组件，且不能掺杂其他技术。React

### 基于MVVM

纯前端js框架：单靠浏览器就可执行，无需服务端支持

何时：**以数据为主的项目**

如何：

官网：cn.vuejs.org

下载：2种：

#### 下载并在网页中引入vue.js ——前三天

最新稳定版本：2.5

开发版：包含完备的警告和错误提示信息

生产版：移除警告和错误提示信息，并压缩

http://cn.vuejs.org/v2/guide/installation.html

#### 安装VUE脚手架工具 —— 最后两天

命令行工具（CLI）

Vue.js提供的官方命令行工具（http://cli.vuejs.org/），可用于快速搭建大型单页面应用。

步骤如下：

1. 全局安装vue-cli

npm install -g @vue/cli

1. 创建一个新的空白项目，并下载依赖的模块

vue create my-project

1. 进行项目目录，并进行该项目

cd my-project

npm run serve

注：不同版本的Vue-CLI工具的使用方法差别较大，最新版本的使用过程请参考官方手册。

## MVVM框架原理

### 传统前端三大文件

.html 专门定义网页内容 —— 都是静态

.css 专门定义网页样式 —— 都是静态

.js 专门定义网页交互行为 —— 动态

### 框架将前端重新进行划分

View 视图：包括以前的HTML和CSS

Model 数据模型：内存中(ajax请求回来的)包含数据的对象

ViewModel 控制器：能自动将model中的数据，填充到View中的中间程序。（前端任务）

### ViewModel的原理

引入vue.js：其实是在全局添加一种新的类型Vue

包括：

1. 构造函数：创建一个Vue对象的实例

var vm=new Vue({ … })

1. 原型对象：保存所有Vue对象公用的方法

new Vue({ 创建一个ViewModel控制器对象

el:”选择器”, //建立对页面元素的监视

data:{ 模型变量 } //将模型数据中的数据自动绑定到受监视的页面元素中

})

例：

var vm=new Vue({ // ViewModel控制器

el:"#app",

data:{ // Model

//ajax:

uname:"dingding",score:3000,

}

})

// setInterval(code,millisec)

// setInterval() 方法可按照指定的周期（以毫秒计）来调用函数或计算表达式。直至clearInterval() 被调用或窗口被关闭

// code 要调用的函数或要执行的代码串

// millisec 周期性执行或调用 code 之间的时间间隔，以毫秒计

setInterval(function(){

vm.score+=500;

},1000);

### new Vue()内包含两大子系统

#### 响应系统

将data中每个变量，都重新封装为new Vue()对象的访问器属性。

只要修改new Vue()对象的属性，就会触发set()和get()方法。

set()除了修改模型数据外，还通知Vue，该模型变量发生了变化，需要DOM树修改。

#### 虚拟DOM树

什么是：new Vue()对受监视的页面元素，重新扫描后构建的简化版DOM树

何时：

1. 模型数据发生变化，会通知虚拟DOM树
2. 虚拟DOM树会通过遍历自己的方式查找到受影响的元素
3. 用自己封装好的DOM API仅更新受影响的元素

#### 为什么快

1. 局部更新真实DOM树，避免大范围重排重绘。
2. 无需人工执行增删改查操作干预，自动化程度高

#### 鄙视：为什么框架使用虚拟DOM树

1. 遍历快：仅保存了受影响的元素和属性
2. 仅更新受影响的DOM元素，效率比整体替换高！

## 绑定语法

{{}}：双花括号语法，插值(interpolation)绑定

大胡子语法(Mustashe)

同ng(AngularJS简称)

何时：仅限于绑定内容，不能绑定属性

如何：

1. {{模型变量}}
2. {{算数/比较/三目运算}}
3. {{方法(实参)}}
4. {{对象.属性}}
5. {{数组[i]}}

不能写程序结构：for if while…

## 指令（directive）

什么是：增强HTML功能的特殊自定义属性

为什么：HTML本身是静态的，无法动态生成内容或属性值

注意：指令写在属性标签中

包括：13个

### v-bind

专门用于让HTML的属性也能动态绑定数据

何时：只要绑定属性值，都用v-bind

如何：<ANY v-bind:属性名=”变量/表达式…”

简写：<ANY :属性名=… 不用写v-bind

### v-for

专门根据数组反复生成多个相同结构的元素

何时：...

如何：<ANY v-for=”value of 数组|对象|字符串|整数”

结果：数组中有几个元素，就会反复创建几个<any>

在<ANY>元素及其子元素中,value可当做模型变量绑定

如果希望同时获得元素值和下标i时（注意书写顺序）：

<ANY v-for=”(value,i) of 数组”

强调：v-for必须写在要反复生成的子元素上，而不能写下父元素上！

注：v-for中的of和in的效果一样，但ng中设定的是of，所以建议使用of。

注：v-for遍历一个整数就是迭代，相当于数数。

例：

<div id="app">

<ul>

<!-- v-for 并不是像dom、jQuery一样绑在要遍历的父元素，而是绑在要遍历的元素上 -->

<b>遍历数组</b>

<li v-for="(task,i) of tasks">{{i+1}} - {{task}}</li>

<hr>

<b>遍历对象</b>

<li v-for="(value,key) of a">{{key}} - {{value}}</li>

</ul>

<hr>

<b>遍历数值</b>

<div>

<a href="javascript:;" v-for="i of pageCount"> {{i}} </a>

</div>

</div>

<script>

var vm=new Vue({

el:"#app",

data:{

tasks:["吃饭","睡觉","打豆豆"],

a:{

math:91,

chs:69,

eng:96

},

pageCount:3

}

})

</script>

### v-if

专门根据条件控制元素的显示隐藏

单用：<ANY v-if=”条件”

如果满足条件就显示<ANY>，否则不显示

如果多个元素选其一显示：

<ANY v-if=”条件1”

<ANY v-else-if=”条件2”

… …

<ANY v-else> //当前边所有条件都不满足时，选用

强调：v-if v-else-if v-else之间不能插入其他元素，必须紧邻

例：

<h1 v-if="!isLogin">登录 | 注册</h1>

<h1 v-else="isLogin">welcome dingding,注销</h1>

<script>

var vm=new Vue({

el:"#app",

data:{

isLogin:false, // vm.isLogin=true

}

})

</script>

当输入：vm.islogin=true，显示：welcome dingding，注销

当输入：vm.islogin=false，显示：登录|注册

例：

<h1 v-if="orderStatus==10">未付款</h1>

<h1 v-else-if="orderStatus==20">已付款</h1>

<h1 v-else-if=orderStatus==30>待发货</h1>

<h1 v-else-if=orderStatus==40>已发货</h1>

<h1 v-else>已签收</h1>

<script>

var vm=new Vue({

el:"#app",

data:{

orderStatus:10, // vm.orderStatus+=10

}

})

</script>

当orderStatus的值不断增加时，显示的数据由“未付款”——“已签收”不断变化

### v-show

专门根据条件控制元素的显示隐藏

是通过display属性控制显示隐藏

如何：用法和v-if完全一样

vs v-if：是通过是否创建元素到DOM树来控制显示隐藏

如何选择：

只是控制一个元素的显示隐藏，首选v-show

如果控制多个元素选其一显示，首选v-if…

#### 鄙视：VUE中的 v-show 和 v-if的区别

v-if：是通过动态的在DOM树中创建该元素，来控制它的显示和隐藏；

v-show：是通过改变元素属性（display:none;）的方式来改变元素的显示和隐藏。

v-if有更高的切换消耗；v-show有更高的初始渲染消耗；

v-if适合运营条件不大可能改变；v-show适合频繁切换。

当有多个元素时，建议使用v-if；

如只有一元素时，建议使用v-show。

但当对安全有要求时，建议使用v-if

### v-on

专门绑定事件处理函数

如何：<ANY v-on:事件名=”处理函数(参数值)”

简写：<ANY @事件名=”…”

例：

<div id="app">

<button v-on:click="change(-1)">-</button>

<span>{{n}}</span>

<button @click="change(+1)">+</button>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

n: 0

},

methods: {

change(i) {

this.n += i;

}

}

})

</script>

#### 如何获得事件对象e：

<ANY @事件名=”处理函数($event,参数值)”

其中：$event是Vue中的关键词，专门表示事件对象

new Vue({

methods:{

处理函数(e,形参){

e和DOM中的e是完全一样的

}

}

})

例：

<div id="app">

<a href="#" @click="change($event,-1)">-</a>

<span>{{m}}</span>

<a href="#" @click="change($event,+1)">+</a>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

n: 0

},

methods: {

change(e,i) {

e.preventDefault();

this.n += i;

}

}

})

</script>

可得到，以a标签为按钮，不想让其作为“锚点”，不应用href="#"，而是href="javascript:;"，可称为真正的死链接。

但，不改href="#"时，需要使用$event和e.preventDefault()

#### 阻止默认行为和取消冒泡：

除了e.preventDefault()外，还可用事件修饰符（vue独有）：

@事件名.prevent 代替 e.preventDefault()

.stop 代替 e.stopPropagation()

例：

<div id="app">

<a href="#" @click.prevent="change(-1)">-</a>

<span>{{m}}</span>

<a href="#" @click.prevent="change(+1)">+</a>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

n: 0

},

methods: {

change(i) {

this.n += i;

}

}

})

</script>

可视为上式的简写。

#### 键盘事件：

@keyup.按键号=”处理函数()”

只有按得键是规定的键时，才出发keyup事件

enter（13）

左（37），上（38），右（39），下（40）

例：

<div>

<h3>键盘事件</h3>

<h4>使点击事件与键盘“enter”键绑定</h4>

<input type="text" @keyup.13="search()">

<!-- 13为enter键的键值 -->

<!-- keyup表示当键盘按键抬起时,触发事件 -->

<!-- keydown表示当键盘按键按下时,触发事件 -->

<button @click="search()">百度一下</button>

</div>

<script>

var vm = new Vue({

el: "#app",

methods: {

search(){

alert("ding");

}

}

}

})

</script>

### v-html和v-text：

v-html：专门以替换方式绑定innerHTML

何时：只要绑定HTML片段

结果：会被浏览器解析为网页内容

v-text：专门以替换方式绑定textContent

何时：只要绑定的纯文本内容

结果：努力保持文本原貌，不进行任何编译

vs {{}}：v-text不支持过滤器；{{}}支持过滤器

例：

<div id="app">

<ul>

<li>{{msg}}</li>

<li v-html='msg'></li>

<li v-text="msg"></li>

</ul>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

msg:`来自&lt;&lt;新华社&gt;&gt;的消息`,

}

</script>

效果：

来自&lt;&lt;新华社&gt;&gt;的消息

来自<<新华社>>的消息

来自&lt;&lt;新华社&gt;&gt;的消息

### v-cloak

专门在new Vue对象加载完之前，临时隐藏{{}}语法

问题：如果new Vue加载慢，客户可能短暂看到{{}}

解决：用v-cloak，在new Vue加载完之前临时隐藏元素

如何：

1. 在要隐藏的元素上添加自定义属性v-cloak，不用赋值

<ANY v-cloak

1. 在当前页面的样式表中，手动添加：

[v-cloak]{ display:none }

1. 当new Vue加载完，会自动查找所有v-cloak的元素，移除v-cloak属性

### v-pre

保留当前元素中的{{}}语法，不绑定数据

### v-once

仅在加载时动态绑定一次，之后自动解绑。

即使模型变量再更新也不会改变页面

如何：<ANY v-once>{{}}

原理：在绑定一次后，将该元素从虚拟DOM树中移除

例：

<div id="app">

<h1 v-once>加载时间：{{time.toLocaleString()}}</h1>

<h1>当前时间：{{time.toLocaleString()}}</h1>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

time:new Date()

}

})

setInterval(function(){

vm.time=new Date();

},1000)

</script>

“加载时间”时间只加载一次，不再改变；

“当前时间”时间随定时器的时间的改变而改变。

注：

绑定内容{{}}

绑定属性：

绑定事件@

data中写变量

methods中写处理函数

注：vue在修改页面时，对于href、class、src属性的值的修改需要在属性前加“:”，且不能加“{{}}”，属性内的值的修该直接在“引号”（""）内修改，或用模板字符串${}的形式修改。

### 双向绑定 v-model

单向绑定：仅能将Model中的数据绑定到View中

**双向绑定**：既能将Model中的数据绑定到View中

当用户修改View中的内容时，也能反向更新内存中的model数据

今后，只要绑定可修改的表单元素的值时，都用v-model

如何：<ANY v-model:value=”模型变量/表达式”

原理：

Vue将所有v-model的元素单独组成一个监视队列watch

用死循环方式反复监视队列中的元素值变化

当监视到发生变化，则自动修改对应的模型变量

包括：text,textarea、radio、select

#### text, textarea

<ANY v-model:value=”模型变量/表达式”

可简写为：<ANY v-model =”模型变量/表达式”

注：watch函数，只要当前模型变量改变，就立刻触发

new Vue({

el:"xxx"

data:{...},

watch:{

模型变量名(){

this.模型变量名...

}

}

)}

例：

<div id="app">

<input type="text" placeholder="请输入关键词" v-model="kwords">

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

kwords:"macbook"

},

watch:{

// 只要当前模型变量改变，就立刻触发

kwords(){

console.log(`去服务器查找关键词：${this.kwords}`);

}

}

})

</script>

#### radio

<input type=”radio” value=”写死的值” v-model=”模型变量”

用模型变量的值和value作比较，如果相等就checked，否则就不checked

选中时，会将value更新到模型变量中。

例：

<div id="app">

性别：

<label><input type="radio" name="sex" value="1" v-model="sex">男</label>

<label><input type="radio" name="sex" value="0" v-model="sex">女</label>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

sex:0,

}

})

</script>

#### select

<select v-model=”模型变量”>

<option value=”值1”>xxx

<option value=”值2”>xxx

绑定时，用模型变量的值和每个value比较，只有相等的才选中。

选项改变时，会将选中的option的value反响更新回模型变量

例：显示内容随orderStatus的值的变化而变化

<div id="app">

<select v-model="orderStatus">

<option value="0">-请选择-</option>

<option value="10">未付款</option>

<option value="20">已付款</option>

<option value="30">已发货</option>

<option value="40">已签收</option>

</select>

</div>

<script>

var vm = new Vue(

el: "#app",

data: {

orderStatus:10

}

})

</script>

将10赋值给orderStatus中，select会拿v-model和每一个value作比较，当符合时，显示内容。

例：当“同意”按钮未点击时，“注册”按钮为只读状态。

<div id="app">

<label>

<input type="checkbox" v-model="agree">同意

</label>

<br>

<button :disabled="!agree">注册</button>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

agree:false

}

})

</script>

### 绑定style和class

#### 绑定style

1. 将style按字符串绑定：问题：不便于修改其中一个css属性；
2. 将style按对象绑定：

style中黑可以直接绑定一个对象语法：

style="{ 'css属性':值, 'css属性':值, ... }

new Vue({

data:{

popStyle:{ // 对象语法

// 方便修改一个值

css属性名:”属性值px”,

css属性名:”属性值px”

}

}

})

例：实现鼠标拖拽效果

<style>

\* {padding: 0;margin: 0;}

#app {

margin: 20px;

width: 100%;

height: 734px;

background: #aaddcc;

}

#pop {

position: fixed;

top: 0;

left: 0;

width: 150px;

height: 150px;

background: pink;

}

</style>

<div id="app" @mousemove="mmHandler($event)">

<div id="pop" :style="popStyle" @mousedown="mdHandler($event)" @mouseup="muHandler"></div>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

canMove: false,

offsetX: 0,

popStyle: {

top: "100px",

left: "100px"

}

},

methods: {

mmHandler(e) {

if (this.canMove) {

var left = e.clientX - this.offsetX;

var top = e.clientY - this.offsetY;

this.popStyle.left = left + "px";

this.popStyle.top = top + "px";

}

},

mdHandler(e) {

this.offsetX = e.offsetX;

this.offsetY = e.offsetY;

this.canMove = true;

},

muHandler() {

this.canMove = false;

}

}

})

</script>

offset为鼠标点击处相对于其所点击元素的坐标；

client为鼠标点击处相对于窗口的坐标；

muHandler为鼠标按键抬起时；

mdHandler为鼠标按键摁下时；

canMove为变量，阈值，可否移动；

mmHandler为鼠标移动的坐标；

由于所点击的元素在未设置时，默认基准点为该元素的左上角，故需要那 鼠标点击时相对于窗口的坐标 — 鼠标点击时现对于所点击元素的坐标 = 该元素左上角的基准点的坐标

又由于样式中设置为position: fixed;，所以通过left和top可改变所点击元素在窗口中的位置。

#### 绑定class

1. 将class按字符串绑定：问题：不便于修改其中一个css属性；
2. 将class按对象绑定。

模型数据是一个对象，对象中包含每个class的名字，每个class对应一个bool值控制是否应用该class。

注意：如果class名或css属性名中间有“-”，必须用""包裹起来。

写死的class和:class=""是可以共存的

结果：:class绑定后的class会和写死的class合并为一处

2种：

1. 用字符串方式绑定
2. 用对象方式绑定

new Vue({

data:{

classObj:{

样式类名：true/false,

样式类名：true/false

}

}

})

结果：样式类对应的值为true，则出现在元素的class中，否则不出现

注：最简单的绑定方式是“字符串绑定方式”；当需要精确操控class或CSS某一个属性时，使用“对象绑定方式”（“对象绑定方式”访问单个属性较为方便）。

例：

<style>

\* {padding: 0;margin: 0;}

#app {margin: 20px;}

.msg{padding: 10px 5px;}

.fail{

background: red;

color: #fff;

}

.success{

background: green;

color: #fff;

}

</style>

<body>

<div id="app">

<input type="password" v-model="pwd">

<span class="msg" :class="spanClass">{{msg}}</span>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

pwd:"",

msg:"请输入6位数字",

// spanClass:"" //字符串方式绑定

spanClass:{ //对象方式绑定

fail:false,

success:false

}

},

watch:{

pwd(){

//字符串方式绑定

if(/^\d{6}$/.test(this.pwd)){

this.msg="密码可用！";

this.spanClass="success";

}else{

this.msg="密码不可用";

this.spanClass="fail";

}

//对象方式绑定

// test()，用于检测一个字符串是否匹配某个模式

if(/^\d{6}$/.test(this.pwd)){

this.msg="密码可用！";

this.spanClass.success=true;

this.spanClass.fail=false;

}else{

this.msg="密码不可用";

this.spanClass.success=false;

this.spanClass.fail=true;

}

}

}

})

</script>

注：在元素检查中，不会出现:class=""的形式，写死的class和绑定的:class会通过拼接合并显示为class=""的形式。

### computed计算属性

什么是：实际不存在的属性，都要依靠现有属性动态计算获得

如何：

1. 定义时：

new Vue({

computed:{

计算属性名(){

return计算后的值

}

}

})

1. 绑定时：同普通模型变量的绑定完全一样

其实methods也能实现！

例：

<div id="app">

<h1>总价：${{total}}</h1>

<h1>总价：${{total}}</h1>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

cartItems: [

{ pname: "macbook", price: 4455, count: 2 },

{ pname: "xiaomi", price: 3399, count: 3 },

{ pname: "lenove", price: 5500, count: 2 }

]

},

computed: { // 计算型

// methods: { // 函数型

total() {

console.log("计算了一次总价")

return this.cartItems.reduce((prev, item) => prev + item.price \* item.count,0)

}

}

})

</script>

#### computed vs methods

computed的值可被vue缓存，仅计算一次，反复使用

除非依赖的模型变量发生变化，才重新计算

methods是只要调用一次就计算一次，反复调用会导致反复计算——效率低

如何选择：

1. 如果侧重返回值，首选计算属性
2. 如果侧重执行一项任务，才选择methods

### 自定义指令

#### 钩子函数（可选）

bind：只调用一次，指令第一次绑定到元素时调用，用这个钩子函数可以定义一个在绑定时执行一次的初始化动作。

inserted：被绑定元素插入父节点时调用（父节点存在即可调用，不必存在与document中）。

update：被绑定元素所在的模板更新时调用，而不论绑定值是否变化。通过比较更新前后的绑定值，可以忽略不必要的模板更新。

unbind：只调用一次，指令与元素解绑时调用。

#### 创建自定义指令

使用Vue.创建静态方法创建指令，使每个页面的小写vm共用。

Vue.directive(“指令名”,{ // 指令名不要加v-

inserted(el){ // 会在当前元素被添加到DOM树上时自动调用

// 不得修改，为处理函数

// el：当前DOM元素

// 可对el执行原生DOM API

}

})

inserted：当元素被加载（挂）在DOM树上时，自动触发

#### 使用指令

<ANY v-指令名> //指令名必须加v-

#### 创建new Vue实例对象，监视HTML元素

注：只有创建new Vue实例对象监视指定的div，才能够将原生的HTML和指令关联起来，HTML中Vue专用的指令才能被识别出来。即指令是被new Vue识别出来。

注：所有指令在定义时，均不加v-；在调用时，均加v-。（包括基础的13种指令）

例：让文本框自动获得焦点

<script>

Vue.directive("focus",{//定义focus指令时，不加前缀-v

inserted(elem){ //当前input被加载到DOM树时自动执行 //不得修改，为处理函数

elem.focus(); // 原生DOM

}

})

</script>

<div id="app">

<input type="text" placeholder="输入关键词" v-focus>

<!-- 调用focus指令时，加前缀-v -->

</div>

<script>

new Vue({

el: "#app"

})

</script>

## 过滤器(filter)

什么是：接收原始值，进行再加工

何时：只要原始值无法直接使用时

比如：性别：0，1——>女男

如何：

1. 创建:

Vue.filter(“过滤器名”,function(val){

return加工后的新值

})

数据由参数进，return出

1. 使用自定义过滤器：

在{{}}中使用：{{模型变量|过滤器|…}}

在属性绑定时使用：:属性名=”表达式|过滤器|…”

v-text不能用过滤器！

注：每一个过滤器接收到的均是前一个过滤器传过来的数据。

例：自定义过滤

<script>

Vue.filter("sexConvert",function(val){

return val==1?"男":"女"; })

</script>

<div id="app">

<h1>性别1：{{sex1 | sexConvert}}</h1>

<h1>性别2：{{sex2 | sexConvert}}</h1>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

sex1:0,

sex2:1

}

})

</script>

例：自定义多种条件的过滤

<script>

Vue.filter("sexConvert", function (val, lang) {// 静态函数

if (lang =="zh-ch") {

return val == 1 ? "男" : "女";

} else {

return val == 1 ? "Male" : "Female";

}

})

Vue.filter("sexIcon", function (val) {

return val == "男" || val == "Male" ? val + "♂" : val + "♀"

})

</script>

<div id="app">

<h1>性别1：{{sex1 | sexConvert("zh-cn") | sexIcon}}</h1>

<h1>性别2：{{sex2 | sexConvert("zh-cn") | sexIcon}}</h1>

<h1>sex1：{{sex1 | sexConvert("en-us") | sexIcon}}</h1>

<h1>sex2：{{sex2 | sexConvert("en-us") | sexIcon}}</h1>

</div>

<script>

var vm = new Vue({

el: "#app",

data: {

sex1: 0,

sex2: 1

}

})

</script>

注：在多种条件的过滤，第一个形参为默认条件，传递的第一个参数从第二个形参开始赋值。

## Axios(重点)

什么是：专门发送HTTP请求的支持Promise的函数库

何时：今后，没有jQuery也想发送ajax请求时

如何：

1. Get:

axios.get(“url”,{

params:{ 参数:值, … }

}).then(res=>{

res.data才是服务端返回的数据

})

例：

axios.get(

"http://localhost:3000/index"

).then(res => {

console.log(res);

// 因为6个对象被包装在data的数组中

var data = res.data;

console.log(data);

});

例：

var lid = 5;

axios.get(

"http://localhost:3000/details",

{ // 配置对象

params: { lid }

}

).then(res => {

console.log(res.data);

});

注：在url后面是一个巨大的配置对象

注：params配置项，由它携带lid参数

1. Post:

借助qs.min.js（NodeJS的模块）

axios.post(“url”,Qs.stringify({参数1,参数2,…}))

//”参数1=值1&参数2=值2&…”

注：Qs（querystring）：专门用于将url中？后的参数转为对象，或将对象转回来。

注：将对象转为字符串，stringify()

例：

var uname="ding",upwd="123456";

axios.post(

"http://localhost:3000/users/signin",

// 将对象转为字符串

Qs.stringify({uname,upwd})//Qs专门将？后的参数转为对象，或者将对象转为参数

// "uname=dingding&upwd=123456" - querystring

).then(res=>{

console.log(res.data);

})

## 组件

什么是：拥有专属HTML，CSS，数据 的，可重用的页面独立区域。

为什么：可重用，松耦合，便于协作和分工

何时：今后所有的页面都是用组件组成的。

如何：

注：组件的本质就是一个可重用的标签，组件名就是标签名

### 定义组件模板

<template id=”tplxxx”>

<div> 必须只有一个父元素

组件可重用的html片段(包含绑定语法，事件绑定)

</div>

</template>

### 创建组件

Vue.component(“组件名/标签名”,{ //组件其实是可重用的标签而已

template: “选择器找到<template>”,

data(){ //data:function(){ … }——要为每个组件实例创建独有的模型数据

return { 模型变量 }

},

methods:{

处理函数(){ … }

}

})

### 在页面中使用组件

<div id=”#app”>

…<组件名></组件名>

<script>

new Vue({ el:”#app”, … })

原理：new Vue识别出组件标签：

1. 找到组件模板的HTML片段<template>替换<组件名>标签
2. 创建组件实例，绑定组件内容，提供数据和方法支持。

例：

<script>

Vue.component("my-counter",{

template:"#tpl",//new Vue({el})

data:function(){

return {

count:0// 动态绑定一个对象，且每次都会返回独立的一个对象

}

},

methods:{

change(i){

this.count+=i;

}

}

});

</script>

<!-- HTML5:template中专门保存一段隐藏的html片段 -->

<template id="tpl" style="display:none">

<div>

<button @click="change(-1)">-</button>

<span>{{count}}</span>

<button @click=change(+1)>+</button>

</div>

</template>

<div id="app">

<ul>

<li><my-counter></my-counter></li>

<li><my-counter></my-counter></li>

<li><my-counter></my-counter></li>

<li><my-counter></my-counter></li>

</ul>

</div>

<script>

new Vue({

el: "#app"

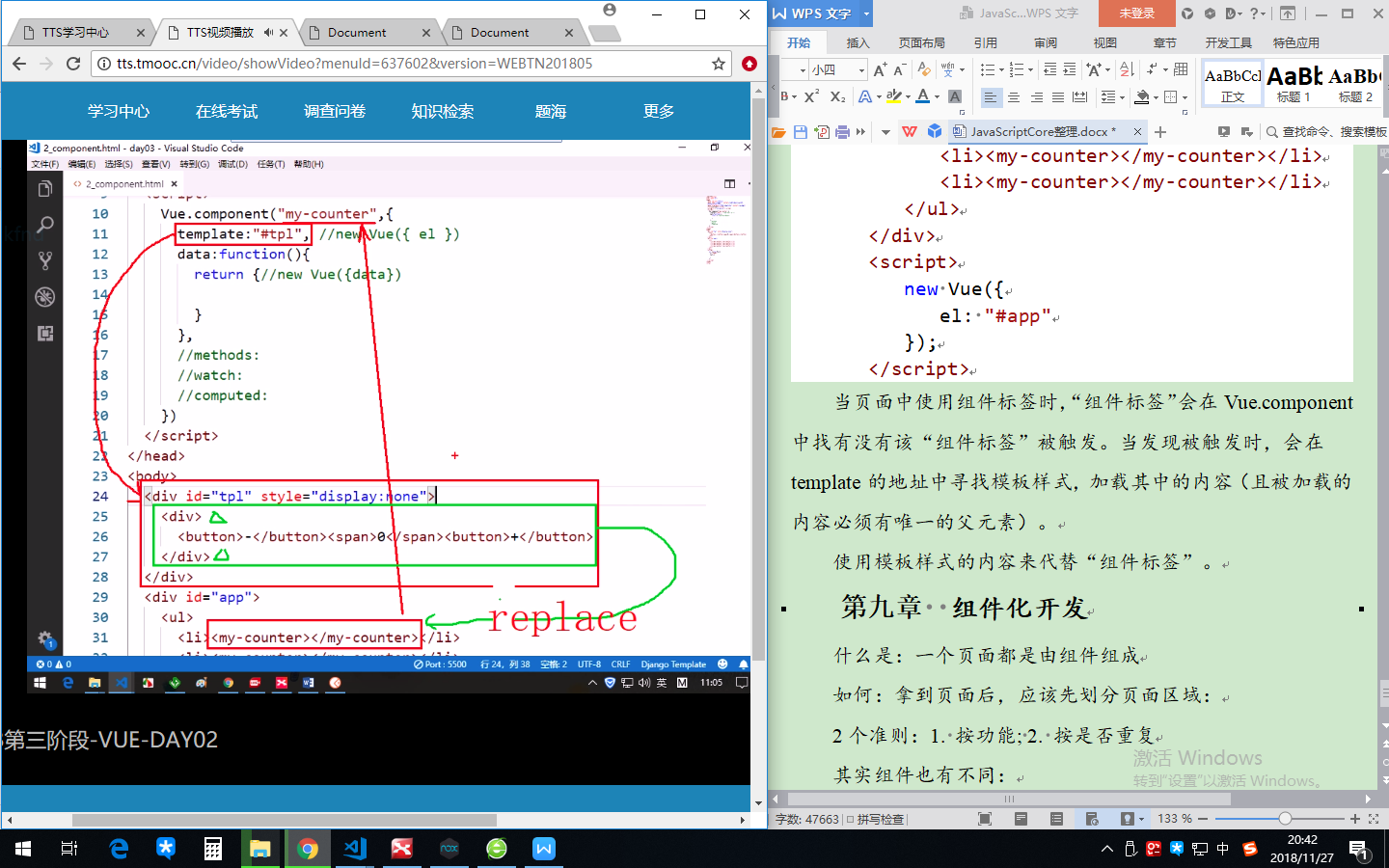
});

</script>

当页面中使用组件标签时，“组件标签”会在Vue.component中找有没有该“组件标签”被触发。当发现被触发时，会在template的地址中寻找模板样式，加载其中的内容（且被加载的内容必须有唯一的父元素）。

使用模板样式的内容来代替“组件标签”。

如下图所示：



### 组件属性

#### Data

Vue实例的数据对象。Vue将会递归将data的属性转换为getter/setter，从而让data的属性能够响应数据变化。

对象必须是纯粹的对象（含有零个或多个的key/value对））

注意：不应该对data属性使用箭头函数。理由是因为箭头函数绑定了父级作用域的上下文，所以this将不会按照期望指向Vue实例，this.myProp将是undefined。

#### Props

props可以使数组或对象，用于接受来自父组件的数据。

props可以使简单的数组，或者使用对象作为替代，对象允许配置高级选项，如类型检测、自定义校验和设置默认值。

注意那些 prop 会在一个组件实例创建之前进行验证，所以实例的属性 (如 data、computed 等) 在 default 或 validator 函数中是不可用的。

#### Computed

计算属性间被混入到Vue实例中。所有getter和setter的this上下文自动地绑定为Vue实例。

注意如果你为一个计算属性使用了箭头函数，则 this 不会指向这个组件的实例，不过你仍然可以将其实例作为函数的第一个参数来访问。

计算属性的结果会被缓存，除非依赖的响应式属性变化才会重新计算。注意，如果某个依赖 (比如非响应式属性) 在该实例范畴之外，则计算属性是不会被更新的。

##### 示例

var vm = new Vue({

data: { a: 1 },

computed: {

// 仅读取

aDouble: function () {

return this.a \* 2

},

// 读取和设置

aPlus: {

get: function () {

return this.a + 1

},

set: function (v) {

this.a = v - 1

}

}

}

})

vm.aPlus // => 2

vm.aPlus = 3

vm.a // => 2

vm.aDouble // => 4

注：多方资料表示，set方法只用了解即可，不需要掌握...

##### 注意事项

###### 不应该使用箭头函数来定义计算属性函数

理由是因为箭头函数绑定了父级作用域的上下文，所以this将不会按照期望指向Vue实例，this.myProp将是undefined。

###### 计算属性可同时监控多个数值

###### 在DOM加载后立刻执行

###### computed创建的变量只能由其set方法修改

#### Methods

methods将被混入到Vue实例中。可以直接通过VM实例访问这些方法，或者在指令表达式中使用。方法中的this自动绑定为Vue实例。

#### Watch

一个对象，键是需要观察的表达式，值是对应回调函数。值也可以是方法名，或者包含选项的对象。Vue实例将会在实例化时调用$watch()，遍历watch对象的每一个属性。

## 组件化开发(重点)

什么是：一个页面都是由组件组成

如何：拿到页面后，应该先划分页面区域：

2个准则：1. 按功能; 2. 按是否重复

其实组件也有不同：

1. 每个页面中可能有多个组件，但多数情况只有一个new Vue()——称为根组件
2. 全局组件：可在页面任何位置使用

Vue.component(“组件名/标签名”,{ … })

1. 局部组件：只能用于特定父组件内的组件，也成为子组件

将全局组件降级为一个普通组件对象

var 组件对象名/驼峰命名={ … }

将所有组件对象提前到父组件之前定义

在父组件中：

Vue.component(“父组件”,{

… ,

components:{

组件对象名, …

}

})

例：基本组件开发代码，之后所有修改、加强均是在以下代码基础上进行

<div id="app">

<todo></todo>

</div>

<template id="tplTodo">

<div>

<h1>待办事项</h1>

<todo-add></todo-add>

<todo-list></todo-list>

</div>

</template>

<template id="tplTodoAdd">

<div>

<input type="text" placeholder="请输入新的代办是想"><button>添加</button>

</div>

</template>

<template id="tplTodoList">

<ul>

<li><todo-item></todo-item></li>

<li><todo-item></todo-item></li>

<li><todo-item></todo-item></li>

</ul>

</template>

<template id="tplTodoItem">

<span>

1 - 吃饭 <a href="javascript:;">X</a>

</span>

</template>

<script src="./11\_VUE/day\_01/js/vue.js"></script>

<script>

Vue.component("todo",{

template:"#tplTodo"

});

Vue.component("todo-add",{

template:"#tplTodoAdd"

});

Vue.component("todo-list",{

template:"#tplTodoList"

});

Vue.component("todo-item",{

template:"#tplTodoItem"

});

</script>

<script>

var vm = new Vue({

el:"#app"

})

</script>

程序说明：

<todo>为组件插入位置；

<template ....>为定义组件，标签内包裹着组件的内容。根据要求，每个组件的内容都必须有唯一父元素；

Vue.component()为创建组件及其所包含的属性、方法，全局组件；

new Vue为识别出组件标签，找到组件模板，创建组件实例。

例：防止组件泄露，

首先，将全局组件降级为一个普通组件对象

其次，将所有组件对象提前到父组件之前定义

// 由于程序是顺序读取，要将子组件放在父组件之前

var todoAdd={ // 将全局组件降级为局部组件

template: "#tplTodoAdd"

};

var todoItem={

template: "#tplTodoItem"

};

var todoList={

template: "#tplTodoList",

components:{ // 将所有子组件对象提前到父组件之前定义

todoItem

}

};

Vue.component("todo", {

template: "#tplTodo",

components:{ // 将所有子组件对象提前到父组件之前定义

todoAdd,

todoList

}

});

注：组件与对象不同，其“父子”之间不是继承的关系。组件拥有专属的数据。

## 组件间传递数据（重点）

问题：组件中的数据是专属的，其它组件默认无法访问。

解决：（不同的方式传参有不同的解决方法）

### 父->子

#### 子组件要定义一个外部可访问到的属性

var 子组件={

template:””,

data(){ return { 模型变量 } }, //仅自己内部可用

props:[ “自定义属性/模型变量” ] //既可被内部访问，又可被外部绑定赋值

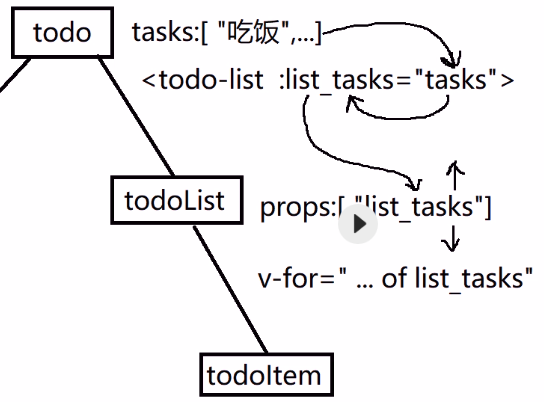
}

#### 父组件的模板中

<template id=”tpl父”> //tasks

…<子组件 :自定义属性=”父模型变量”></子组件>

//将父组件模型变量绑定给子组件的自定义属性



上图概述：

由于组件之间的属性不能相互传递，所以todoList想要父级的data中的tasks。

所以，使用props属性，该属性可以两用，一个可以对外，一个可以对内。

对外：通过props属性抛出list\_tasks到todo的template（定义组件）上的<todo-list ...>标签上，使其与tasks绑定。

使用“:”的方式将父组件的模型变量绑定给子组件的自定义属性。

对内：则可以使用v-for=" (task,i) of list\_tasks "

所以，props属性是“勾连内外”的“元素”。

例：

<template id="tplTodo">

<div>

<h1>待办事项</h1>

<todo-add></todo-add>

<todo-list :list\_tasks="tasks"></todo-list>

</div>

</template>

<template id="tplTodoList">

<ul>

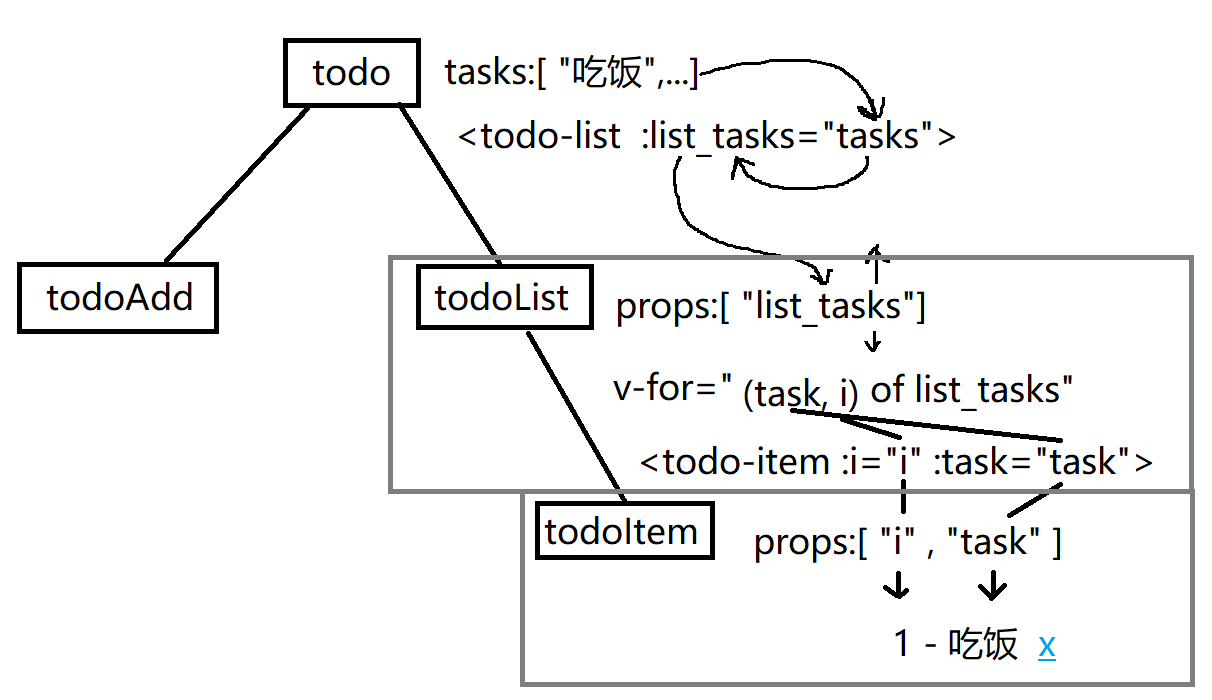
<li v-for="(task,i) of list\_tasks">

<todo-item></todo-item>

</li>

</ul>

</template>



上图概述：每个item中都需要动态获得两个值。使用props属性。过程同上，todoItem向todoList索要"i"和"task"。

v-for=" (task,i) of list\_tasks "中的task和i可以在自己的函数内被当做模型变量去绑定。

例：

<template id="tplTodoList">

<ul>

<li v-for="(task,i) of list\_tasks">

<todo-item :i="i" :task="task"></todo-item>

</li>

</ul>

</template>

<template id="tplTodoItem">

<span>

{{i+1}} - {{task}}

<a href="javascript:;">X</a>

</span>

</template>

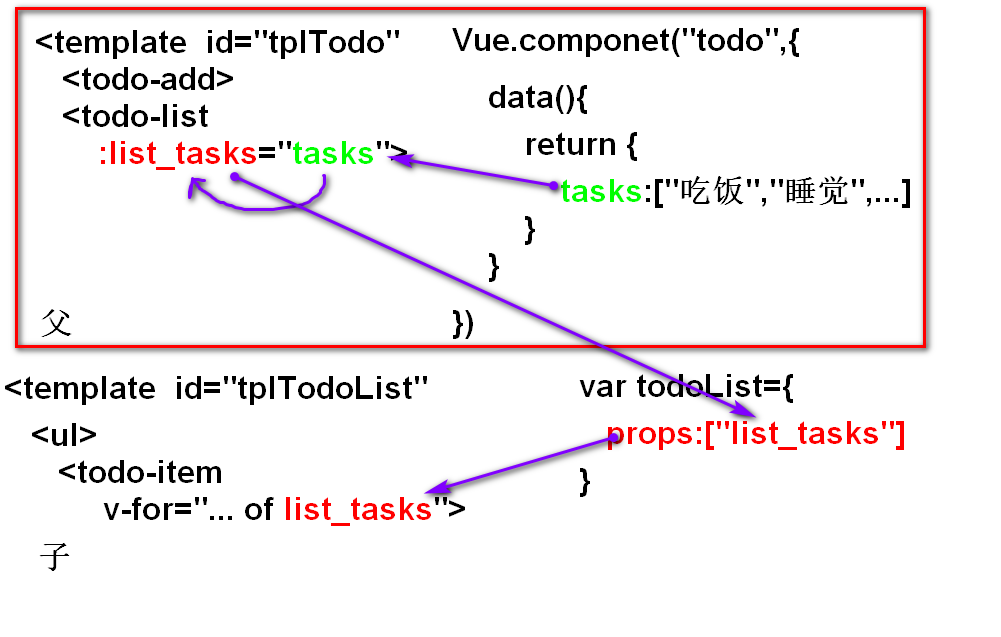
var todoItem={

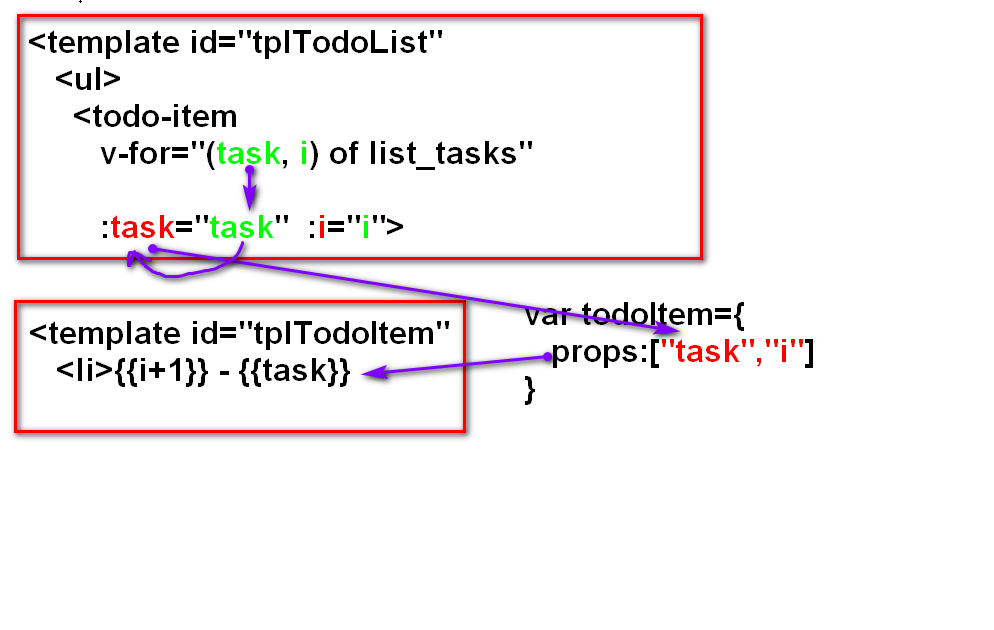
template: "#tplTodoItem",

props:["i","task"]

};

注：tasks与list\_tasks绑定是通过给引用地址而不是复制的方式，两边可以同时改变。





### 子->父

#### 父组件

<template id=”父”

<子 @自定义事件=”处理函数” //不加()

Vue.component(“父”,{

methods:{

处理函数(形参){ … }

}

})

#### 子组件

<ANY @click=”处理函数()”

var 子={

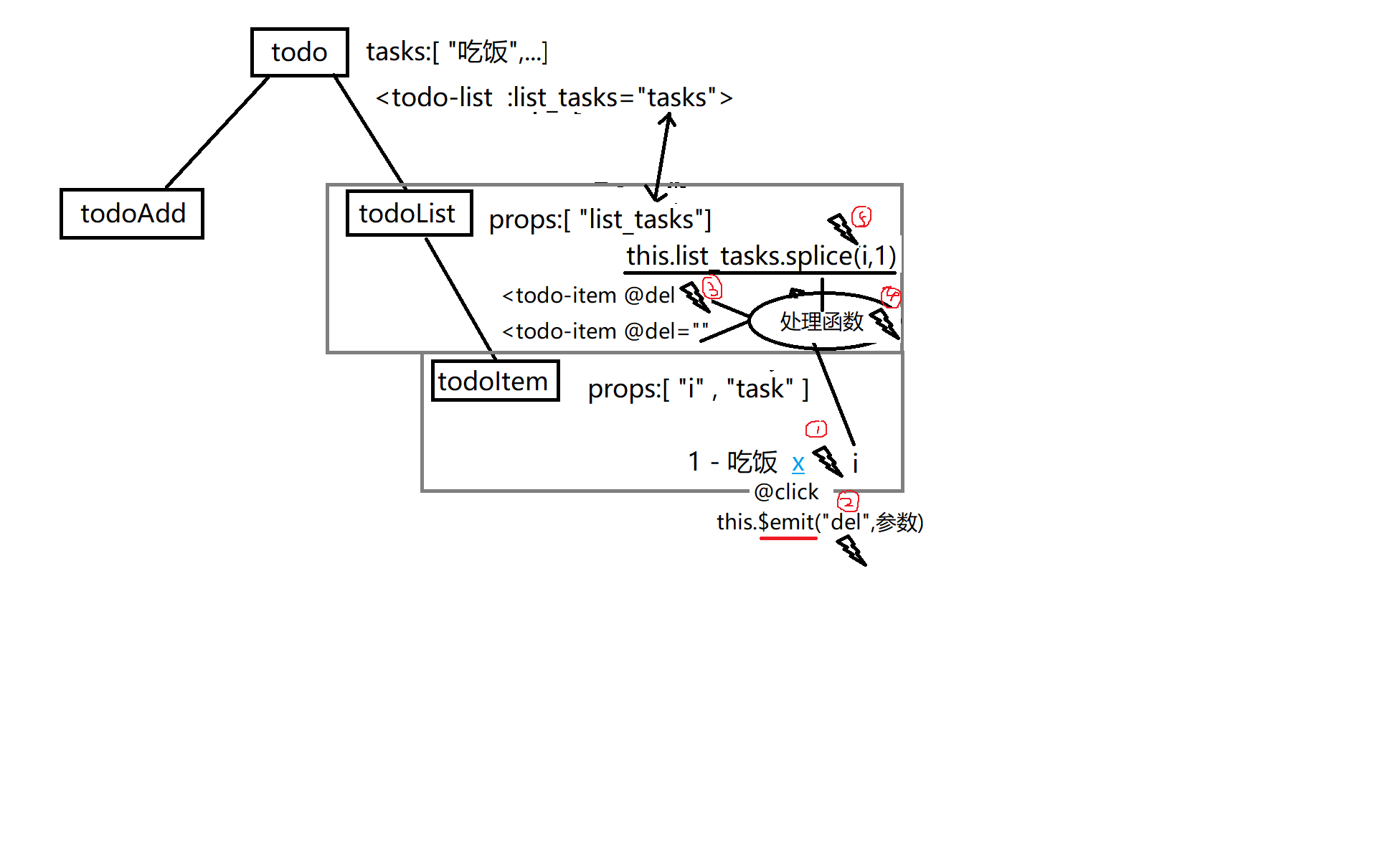
methods:{

处理函数(){ this.$emit(“自定义事件”,参数值) }

}

}

this.$emit->触发相应事件



点击触发①，①传给②this.$emit，由②触发自己身上的del，并将del传给父组件位置③，③通过之前绑定好的处理函数④，造成了父组件输出的结果⑤。

例：

<template id="tplTodoList">

<ul>

<li v-for="(task,i) of list\_tasks">

<todo-item :i="i" :task="task" @del="del"></todo-item>

<!-- del 绑定，相当于@del=del 绑定语法 有没有参数不一定，视传参而定 -->

<!-- del() 函数调用，()表示无参数 -->

</li>

</ul>

</template>

<template id="tplTodoItem">

<span>

{{i+1}} - {{task}}

<a href="javascript:;" @click="del(i)">X</a>

</span>

</template>

var todoItem={

template: "#tplTodoItem",

props:["i","task"],

methods:{

del(i){this.$emit("del",i)}

}

};

var todoList={

template: "#tplTodoList",

props:["list\_tasks"],

components:{todoItem},

methods:{

del(i){this.list\_tasks.splice(i,1)}

}

};

注：先在item和list上分别绑定处理事件和触发事件。

由于在id为tplTodoItem的定义组件内的a标签上绑定@click="del(i)"。所以在script的todoItem参数内添加一个methods(方法)。该方法为触发函数。并通过this.$emit()将"del",i传递给id为tplTodoList的定义组件内的@del="del"。（注："del"与"del"是同一del）

再在id为tplTodoList的定义组件内的todo-item标签上绑定@del="del。所以在script的todoList参数内添加一个methods(方法)。该方法为删除效果的处理函数。

注：@del为methods(方法)中的del(i)。

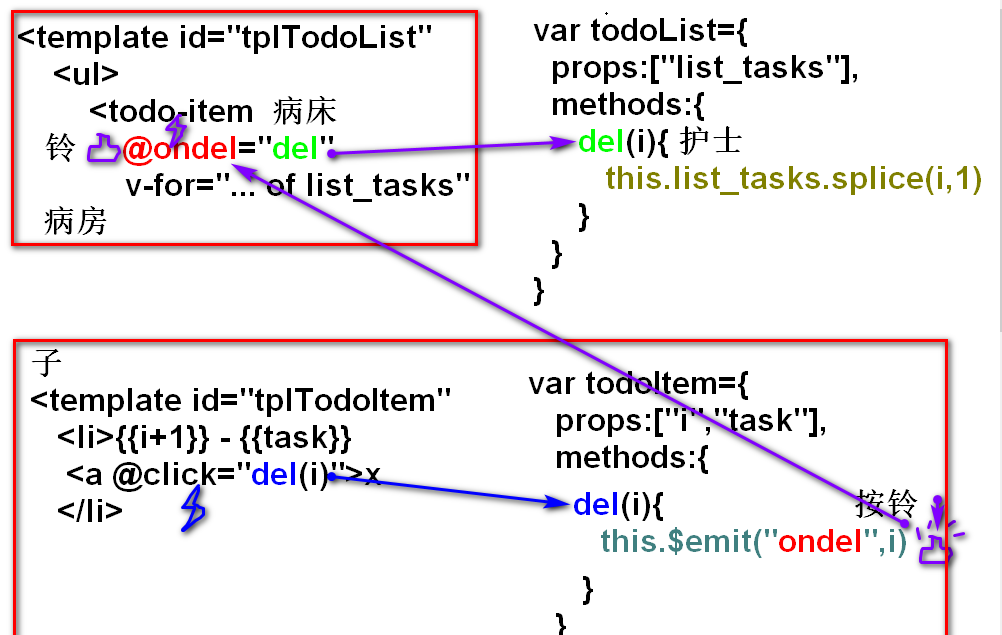


step1:手动触发是事件，由于click事件在tplTodoItem定义组件内绑定了del(i)方法；

step2:导入todoItem参数，触发其预先内设置好的方法，该方法由$emit()将del函数和参数i传送出去；

step3:在事件发生的标签上设置@del函数来绑定step2传出的del函数，该函数通过tplTodoList定义组件传入todoList参数中；

step4:在todoList参数中触发预先设置好的方法，该方法将通过splice()实现删除效果。参数i一直由触发的函数传送至此。



### 兄弟间传参

1. 先给父组件，再由父元素自动同步到子组件

问题：如果子组件离父组件层级较多，则逻辑复杂

1. 借助中间Vue实例对象：

1. 定义空vue对象实例：var bus=new Vue()

2. 接收数据的组件：

var 接收者={

mounted(){

//this->接收者

bus.$on(“自定义事件名”,this.处理函数.bind(this))

},

methods:{

处理函数(参数){ … }

}

}

1. 发送者：

<ANY @click=”处理函数”>

var 发送者={

methods:{

处理函数(){

bus.$on(“自定事件名”,参数值)

}

}

}

例：给程序添加加入列表功能

方法1，子组件a通过父组件将数据传送给子组件b

// 该方法适用于层级较为简单、方便查找的组件

<template id="tplTodo">

<div>

<h1>待办事项</h1>

<todo-add @add="addHandler"></todo-add>

<todo-list :list\_tasks="tasks"></todo-list>

</div>

</template>

<template id="tplTodoAdd">

<div>

<input type="text" placeholder="请输入新的代办是想" v-model="newTask">

<button @click="add">添加</button>

</div>

</template>

var todoAdd={

template: "#tplTodoAdd",

data(){return { newTask:"" }},

methods:{

add(){

this.$emit("add",this.newTask);

this.newTask=""; // 加载完成后清空

}

}

};

Vue.component("todo", {

template: "#tplTodo",

data(){

return {tasks:["吃饭","睡觉","打豆豆"]}

},

methods:{

addHandler(task){

this.tasks.push(task)

}

},

components:{

todoAdd,

todoList

}

});

概述：

step1：

表单元素要想获得页面上的“值”，必须与该“值”所在的标签进行绑定，使用v-model进行双向绑定。

在todoAdd子组件中返回：return { newTask:"" }

在tplTodo的模板组件中的input标签上进行绑定：v-model="newTask"

在button按钮上添加触发事件：@click="add"

step2：

在todoAdd子组件的方法（methods）中添加事件名为：add的触发事件，

add(){

this.$emit("add",this.newTask);

this.newTask=""; // 加载完成后清空

}

并通过$emit将参数传给父级todo的组件的@add自定义事件

注：$emit中的事件名必须与父级组件中的自定义事件名相同

step3：在父级todo的组件的上对add事件进行绑定。@add="addHandler"

step4：在父组件todo中添加方法（methods），

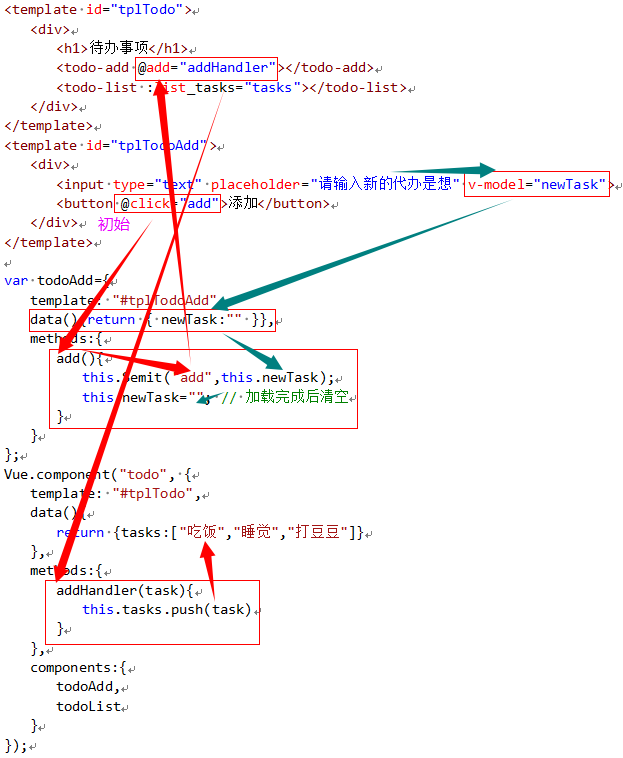
addHandler(task){

this.tasks.push(task)

}

//push() 方法可向数组的末尾添加一个或多个元素，并返回新的长度

完成数组tasks增加新元素的效果。



方法2，子组件a通过添加绑定 将数据传送给子组件b

当嵌套十分的复杂时

<template id="tplTodoAdd">

<div>

<input type="text" placeholder="请输入新待办事项" v-model="newTask">

<button @click="add">添加</button>

</div>

</template>

var bus=new Vue();

var todoAdd={

template:"#tplTodoAdd",

data(){ return { newTask:"" }},

methods:{

add(){

bus.$emit("add",this.newTask);

this.newTask="";

}

}

};

var todoList={

template:"#tplTodoList",

props:["list\_tasks"],

mounted(){//当前组件加载完自动执行

//this->todoList

bus.$on(

"add",

this.addHandler.bind(this)

); //function(){...}

//addHandler();

},

methods:{

addHandler(newTask){

this.list\_tasks.push(newTask);

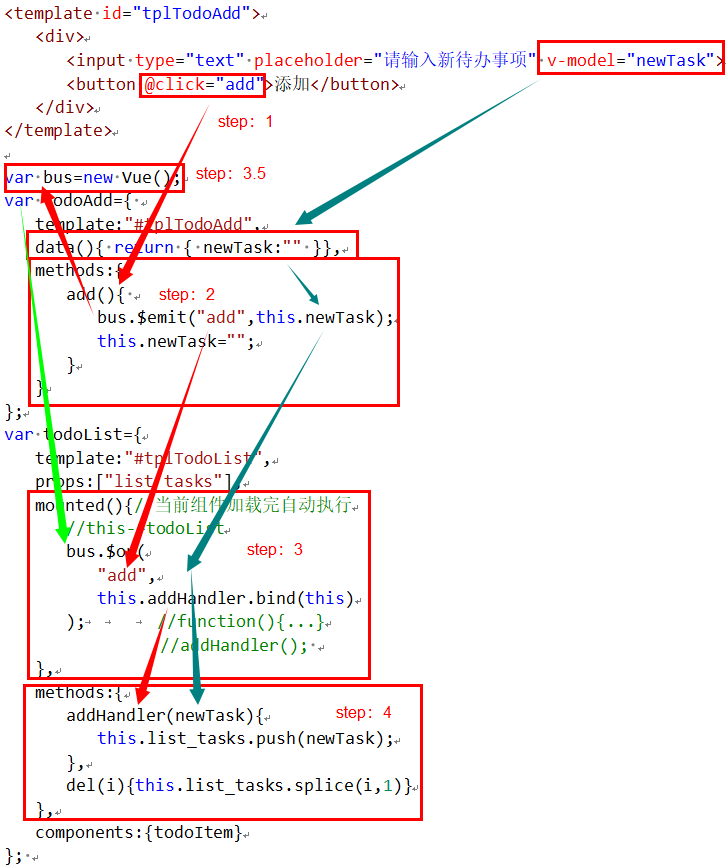
},

del(i){this.list\_tasks.splice(i,1)}

},

components:{todoItem}

};



step1:当点击“添加”按钮，触发单击事件add，将传递至todoAdd子组件参数，同时对input标签双向绑定，获取元素；

step2:在其方法中，add事件为：当bus事件被触发时，将this.newTask参数传递至add事件；

step3:在todoList子组件中有mounted函数，该函数为：当前组件加载完成时自动触发。

即触发bus事件的add事件后，运行this.addHandler.bind(this);

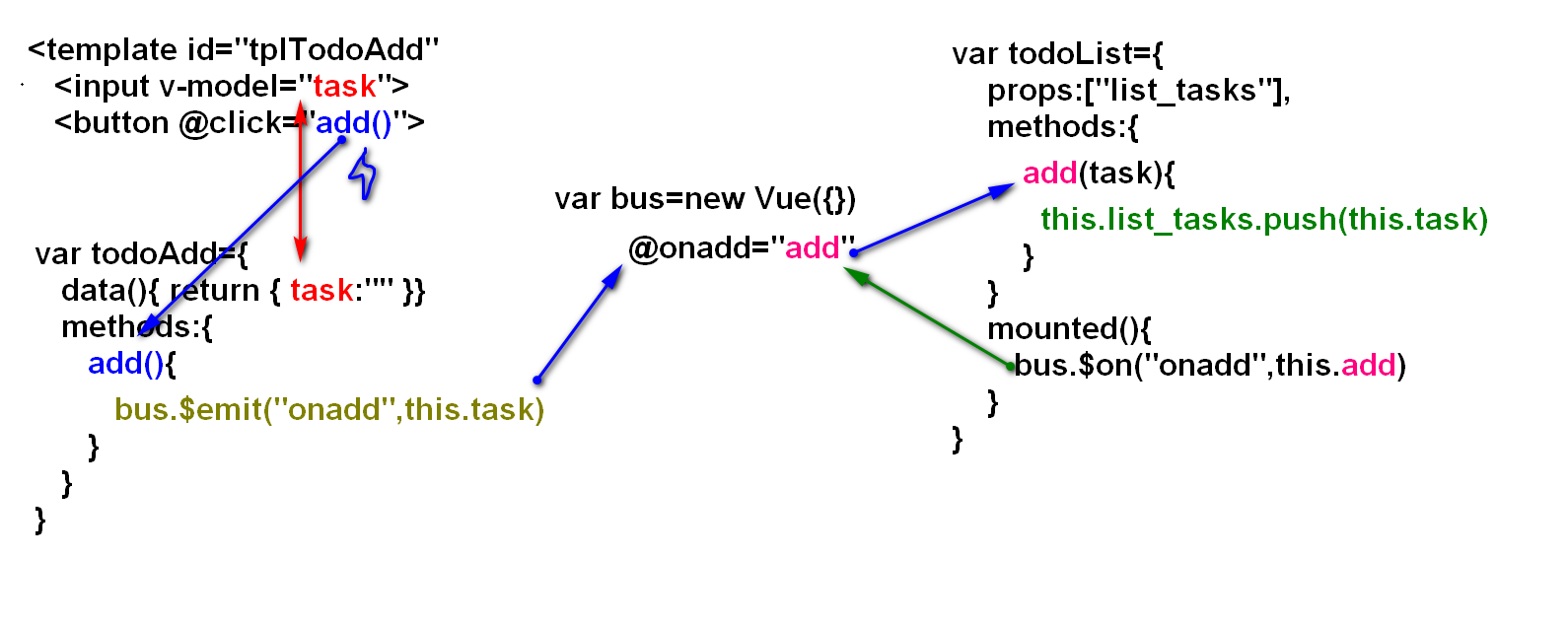
注：关于this.addHandler.bind(this)：其中this.addHandler能够保证在当前对象中找到addHandler，但addHandler本质为function，当该function被add所拿走时，并未记录该function的所有者，所以在调用addHandler()时其this指向window。（回调函数传参，不会传递函数的所有者）所以需要强行更改this的指向，使用call apply bind中的bind，永久的将this指向正确的对象todolist。虽然this.addHandler在vue下能够指向正确的this，但为了保险人为增加了bind(this)，使this指向todolist。

step4:addHandler事件将newTask加至list\_tasks中。

step3.5:$emit和$on实际应该称为发送者和监听者之间的关系。

$emit()当触发bus实例上的事件时，将this.newTask传至监听器add上；

$on()当监听到bus上的自定义事件add时，this.addHandler.bind(this)将回调this.newTask。



## 组件生命周期

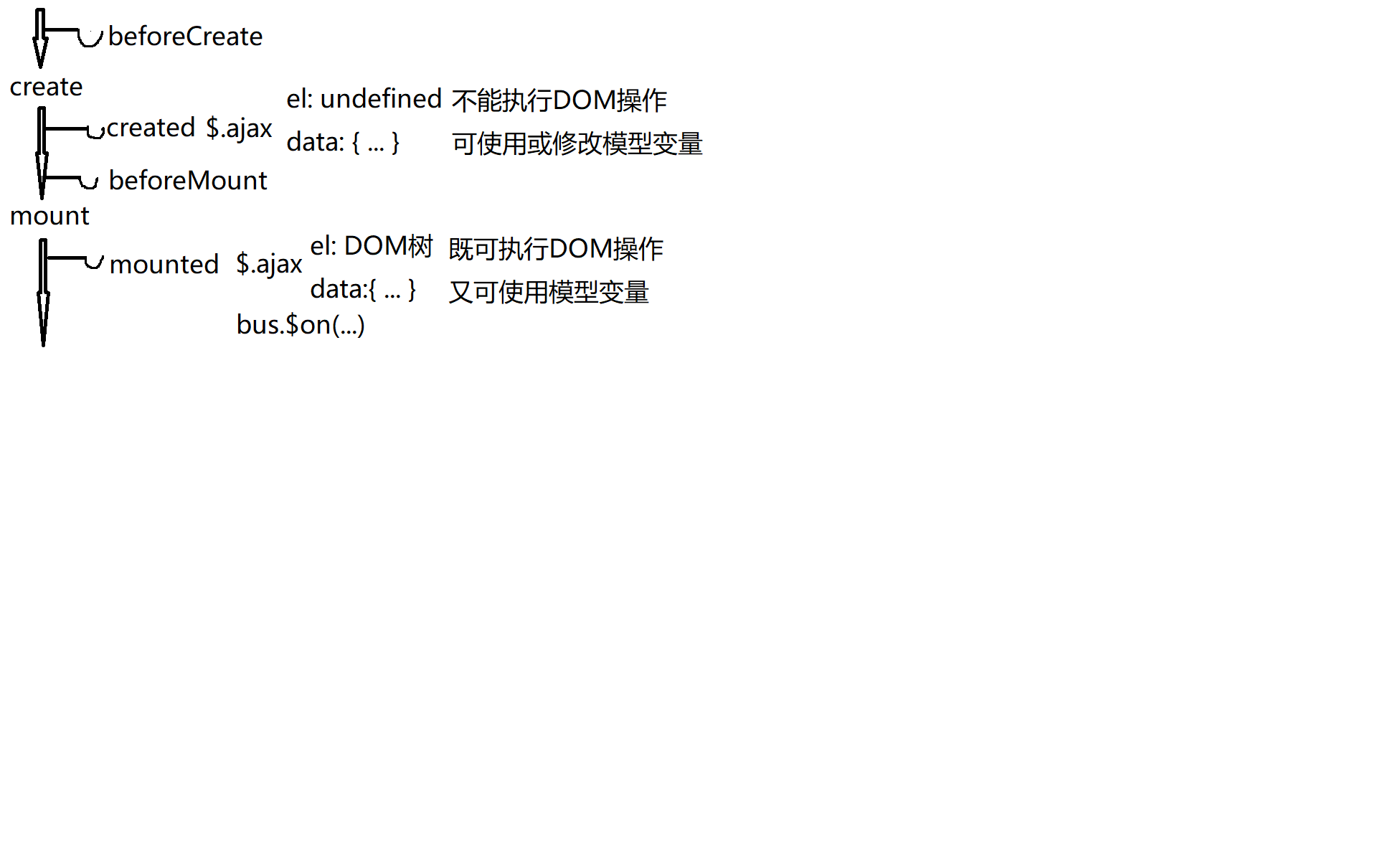
### 4个阶段

Create: 创建Vue组件对象实例，并创建模型对象data

Mount: 将组件模板替换到DOM树，并替换组件内的绑定语法为data中的具体值

Update: 组件模型数据发生改变时

Destory: 主动销毁组件对象实例时



### 8个生命周期钩子函数

1. beforeCreate
2. created

el: undefined 暂时不能执行DOM操作

data: { … } 可使用data中的模型变量

axios.get/post(…) 和data有关，与DOM无关

bus.$on(…)

1. beforeMount
2. mounted

el: DOM树 可执行DOM操作

data: { … 也可使用data中的模型变量

axios.get/post(…) 和data有关，与DOM无关

bus.$on(…)

1. beforeUpdate

数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

你可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。

该钩子在服务器端渲染期间不被调用。

1. Updated

由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态，因为这可能会导致更新**无限循环**。

该钩子在服务器端渲染期间不被调用。

1. beforeDestroy

在当前组件销毁前，通常为vue组件间的跳转时触发。

1. Destroyed

|  |
| --- |
| this.nextTick(callback)，当数据发生变化，更新后执行回调。  this.$nextTick(callback)，当dom发生变化，更新后执行的回调。 |

### 其他

#### 监听是否刷新或关闭

created () {

// 监听刷新或关闭

window.addEventListener('beforeunload', e => this.beforeunloadFn(e))

},

methods: {

beforeunloadFn(e) { // 监听刷新或关闭

console.log('监听刷新或关闭')

}

},

destroyed() {

window.removeEventListener('beforeunload', e => this.beforeunloadFn(e))

}

## SPA(Single Page Application)

什么是单页面应用: 整个应用程序只有一个完整的HTML页面。所谓页面跳转，是通过替换网页中指定区域的内容，来模拟实现的跳转。

为什么：

|  |  |
| --- | --- |
| 多页面应用 | 单页面应用 |
| 包含多个独立的完整的HTML页面  缺: 请求次数多 | 整个项目只有一个完整的HTML页面。其它所谓的“页面”只不过是HTML片段而已  优: 请求少 |
| 页面跳转的本质:  删除就的整个DOM树，重新下载另外整个DOM树，重新加载。  缺: 效率低 | 页面跳转的本质:  只是把当前DOM树中局部片段替换为另一个HTML片段  优: 效率高 |
| 如果两个页面都需要一个相同的资源，每个页面都要重复请求 | 如果多个“页面”都要用到相同的资源，注定只需要下载一次即可 |
| 无法实现页面切换动画 | 有机会实现页面切换动画 |

实现过程：

1. 定义一个完整的HTML页面
2. 定义多个“页面”的HTML片段<template>
3. 定义路由字典：(routes)

一组相对路径和对应”页面”之间的对应关系的集合

var routes=[

{path:”/index”, component: Index },

{path:”/details”, component: Details},

{…}

]

1. 编写路由器程序

获得url中的相对路径

去路由字典中查询要加载的对应组件片段

用找到的组件片段，替换完整页面中占位符元素所在的位置

## Vue脚手架

### Vue脚手架工具

什么是脚手架工具：用于创建脚手架代码的命令行工具

什么是脚手架代码：已经包含最核心功能的半成品项目结构。

为什么：避免重复劳动，标准化项目结构

### 安装/使用Vue脚手

#### 安装脚手架命令行工具

npm i -g @vue/cli

#### 用脚手架命令行生成脚手架项目代码

vue create xz\_vue

step：1

选Manually select feature——手动选择配置

step：2

选组件:空格选中/取消选中

选Babel: 编译器，将浏览器不认识的变种js，转化为标准的浏览器认识的js。

比如: Vue使用了ES6中的模块语法: 浏览器不认

-export default { … } ——抛出模块对象

等效于node中的module.exports={ … }

-Import 组件 from ‘./views/组件.vue’ ——引入模块

等效于var 组件=require(“./views/组件.js”)

(TypeScript：Vue中不要求使用，而ng中要求使用)

(Progressive Web App (PWA) Support：渐进式web app)

选Router——实现单页面应用中的路由组件

选Vuex——实现客户端状态存储

(CSS Pre-processors：使用sass，scss等css预编译器时，使用)

(Linter / Formatter：代码质量检查工具；ng中强制启用；公司中会用到，个人使用没必要)

(Unit Testing 和 E2E Testing ：为单元测试使用)

step：3

Use history mode for router 是否使用history模式作为路由方式：选N 不启用history模式

Vue默认的路由模式：域名/#/路径

History模式：域名/路径 ——需要修改服务器配置才能实现

step：4

将配置保存在各自独立的配置文件中，还是集中保存在package.json文件中：选In package.json

step：5

是否保存当前配置为今后项目的模板：选N

完成：

完成如上操作后，会在xz\_vue文件夹生成vue脚手架。

安装完成后，会出现如下代码提示：



cd xz\_vue 进入xz\_vue文件夹

nmp run serve 运行该vue脚手架

### 运行测试

进入脚手架项目文件夹内, 运行：npm run serve

编译脚手架中的零散的项目为一个完整的项目，并启动一个简版开发服务器，临时宿主调试的网页。

### 脚手架项目结构



注：每个组件就是一个.vue文件

### 打包发布

npm run build

结果：生成dist文件夹，包含编译后的.js .css .html等

将来只将dist文件夹部署到云服务器即可。

### Vue-router

#### 根组件<template>中

用<router-view>占位

#### 在views文件夹下，定义多个组件.vue：

<template>

<div>HTML片段</div>

<script>

export default {

组件内容

}

<style scoped>

组件自用的css样式

#### router.js中编写路由字典:

1. 引入组件.vue：

import 组件名 from “./views/组件.vue”

1. 注册.vue文件

routes=[

{path:”/”, component: 组件名},

…

]

#### 浏览时

1. 用根组件App.vue的<template>替换.html文件中的<div id=”app”>
2. 用路由器解析URL地址中的相对路径，去router.js中的routes列表中查找匹配的component，用找到的component的template替换App.vue中的占位符<router-view>

#### 嵌套路由

何时：多个页面之间包含相同的部分

如何：

1. 定义公共父组件：

包含相同的部分和<router-view>

1. 每个子组件仅包含不同的部分
2. 在路由字典中：

{path:”/父级路径”,component: 父组件, children:[

{path:”/父级路路径/子级路径”, component: 子组件},

… …

]}

##### 浏览时

1. App.vue代替<div id=”app”>
2. 父组件代替App.vue中的<router-view>
3. 子组件代替父组件中的<router-view>

##### 路由跳转

1. 在html中写死的跳转：（强烈推荐的方法）

<router-link to=”/路径”>文本</router-link>

运行时—> <a href=”/#/路径”>文本</a>

1. 程序跳转

##### 路由传参

1. 在router.js中routes内为接收参数的path添加参数定义:

path:”/路径/:变量”

强调：变量前必须加：

1. <router-link to=”/路径/变量值”

强调：不要加：

1. 接收者组件中

this.$route.params.变量

例：1

router.js中routes内为接收参数的path添加参数定义

注：必须有“：”，其为变量名的特殊标识

export default new Router({

routes: [

{ path: '/', component: Home,

children:[

{path:'/details/:lid',component:Details}

]

}

]

})

例：2

在跳转标签处的路径增加变量值

index.html中

注：不能添加“：”

<router-link to="/details/5">查看详情</router-link>

例：3

接收者组件

Details.vue中

<template>

<div>

<h3>lid:{{lid}}</h3>

</div>

</template>

<script>

export default{ // vue实例对象

data(){ // 在组件中，data应为函数

return {

lid:-1

}

},

created() {

this.lid=this.$route.params.lid;

// $route相当于地址栏里的地址，相当于localtion

// localtion可由.search拿到查询字符串，在vue中$route用.params拿到lid的查询字符串

}

}

</script>

## VueX

### 介绍

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

### 状态管理模式

让我们从一个简单的 Vue 计数应用开始：

new Vue({

// state

data() {

return {

count: 0

}

},

// view

template: `<div>{{ count }}</div>`,

// actions

methods: {

increment() {

this.count++

}

}

})

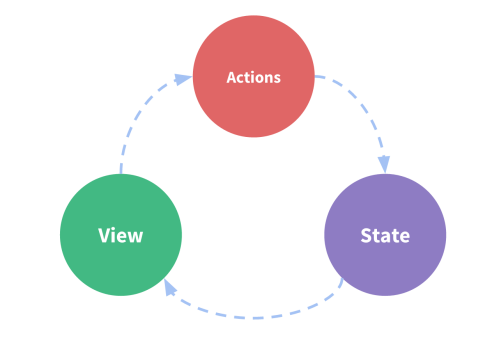
这个状态自管理应用包含以下几个部分：

state，驱动应用的数据源；

view，以声明方式将 state 映射到视图；

actions，响应在 view 上的用户输入导致的状态变化。

以下是一个表示“单向数据流”理念的简单示意：



但是，当我们的应用遇到多个组件共享状态时，单向数据流的简洁性很容易被破坏：

1. 多个视图依赖于同一状态。
2. 来自不同视图的行为需要变更同一状态。

对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。

对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

### Vuex的特点

Vuex 和单纯的全局对象有以下两点不同：

Vuex 的状态存储是响应式的。

当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

### 属性

#### State

由于 Vuex 的状态存储是响应式的，从 store 实例中读取状态最简单的方法就是在计算属性中返回某个状态：每当 **store.state.count** 变化的时候, 都会重新求取计算属性，并且触发更新相关联的 DOM。

然而，这种模式导致组件依赖全局状态单例。在模块化的构建系统中，在每个需要使用 state 的组件中需要频繁地导入，并且在测试组件时需要模拟状态。

Vuex 通过 store 选项，提供了一种机制将状态从根组件“注入”到每一个子组件中（需调用 Vue.use(Vuex)）：

const app = new Vue({

el: '#app',

// 把 store 对象提供给 “store” 选项，这可以把 store 的实例注入所有的子组件

store,

components: { Counter },

template: `

<div class="app">

<counter></counter>

</div>

`

})

通过在根实例中注册 store 选项，该 store 实例会注入到根组件下的所有子组件中，且子组件能通过 this.$store 访问到。

让我们更新下 Counter 的实现：

const Counter = {

template: `<div>{{ count }}</div>`,

computed: {

count () {

return this.$store.state.count

}

}

}

##### mapState 辅助函数

当一个组件需要获取多个状态时候，将这些状态都声明为计算属性会有些重复和冗余。为了解决这个问题，我们可以使用 mapState 辅助函数帮助我们生成计算属性。

// 在单独构建的版本中辅助函数为 Vuex.mapState

import { mapState } from 'vuex'

export default {

// ...

computed: mapState({

// 箭头函数可使代码更简练

count: state => state.count,

// 传字符串参数 'count' 等同于 `state => state.count`

countAlias: 'count',

// 为了能够使用 `this` 获取局部状态，必须使用常规函数

countPlusLocalState (state) {

return state.count + this.localCount

}

})

当映射的计算属性的名称与 state 的子节点名称相同时，我们也可以给 mapState 传一个字符串数组。

computed: mapState([

// 映射 this.count 为 store.state.count

'count'

])

##### 对象展开运算符

mapState 函数返回的是一个对象。

通常，我们需要使用一个工具函数将多个对象合并为一个，以使我们可以将最终对象传给 computed 属性。使用对象展开运算符，我们可以极大地简化写法：

computed: {

localComputed () { /\* ... \*/ },

// 使用对象展开运算符将此对象混入到外部对象中

...mapState({

// ...

})

}

###### 注释

computed: mapState({

count: 'count', // 第一种写法

sex: (state) => state.sex, // 第二种写法

from: function (state) { // 用普通函数this指向vue实例,要注意

return this.str + ':' + state.from

},

// 注意下面的写法看起来和上面相同,事实上箭头函数的this指针并没有指向vue实例,因此不要滥用箭头函数

// from: (state) => this.str + ':' + state.from

myCmpted: function () {

// 这里不需要state,测试一下computed的原有用法

return '测试' + this.str

}

}),

...mapState({ //这里的...不是省略号了,是对象扩展符

count:'count'

})

##### 组件仍然保有局部状态

使用 Vuex 并不意味着你需要将所有的状态放入 Vuex。

虽然将所有的状态放到 Vuex 会使状态变化更显式和易调试，但也会使代码变得冗长和不直观。如果有些状态严格属于单个组件，最好还是作为组件的局部状态。你应该根据你的应用开发需要进行权衡和确定。

#### Getter

有时候我们需要从 store 中的 state 中派生出一些状态，例如对列表进行过滤并计数：

computed: {

doneTodosCount () {

return this.$store.state.todos.filter(todo => todo.done).length

}

}

如果有多个组件需要用到此属性，我们要么复制这个函数，或者抽取到一个共享函数然后在多处导入它——无论哪种方式都不是很理想。

Vuex 允许我们在 store 中定义“getter”（可以认为是 store 的计算属性）。就像计算属性一样，getter 的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。

Getter 接受 state 作为其第一个参数：

const store = new Vuex.Store({

state: {

todos: [

{ id: 1, text: '...', done: true },

{ id: 2, text: '...', done: false }

]

},

getters: {

doneTodos: state => {

return state.todos.filter(todo => todo.done)

}

}

})

##### 通过属性访问

Getter 会暴露为 store.getters 对象，你可以以属性的形式访问这些值：

store.getters.doneTodos // -> [{ id: 1, text: '...', done: true }]

Getter 也可以接受其他 getter 作为第二个参数：

getters: {

// ...

doneTodosCount: (state, getters) => {

return getters.doneTodos.length

}

}

store.getters.doneTodosCount // -> 1

我们可以很容易地在任何组件中使用它：

computed: {

doneTodosCount () {

return this.$store.getters.doneTodosCount

}

}

**注意**，getter 在通过属性访问时是作为 Vue 的响应式系统的一部分缓存其中的。

##### 通过方法访问

你也可以通过让 getter 返回一个函数，来实现给 getter 传参。在你对 store 里的数组进行查询时非常有用。

getters: {

// ...

getTodoById: (state) => (id) => {

return state.todos.find(todo => todo.id === id)

}

}

store.getters.getTodoById(2) // -> { id: 2, text: '...', done: false }

**注意**，getter 在通过方法访问时，每次都会去进行调用，而不会缓存结果。

##### mapGetters 辅助函数

mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性：

import { mapGetters } from 'vuex'

export default {

// ...

computed: {

// 使用对象展开运算符将 getter 混入 computed 对象中

...mapGetters([

'doneTodosCount',

'anotherGetter',

// ...

])

}

}

如果你想将一个 getter 属性另取一个名字，使用对象形式：

mapGetters({

// 把 `this.doneCount` 映射为 `this.$store.getters.doneTodosCount`

doneCount: 'doneTodosCount'

})

#### Mutation

更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。Vuex 中的 mutation 非常类似于事件：每个 mutation 都有一个字符串的 **事件类型 (type)**和一个 **回调函数 (handler)**。这个回调函数就是我们实际进行状态更改的地方，并且它会接受 state 作为第一个参数：

const store = new Vuex.Store({

state: {

count: 1

},

mutations: {

increment (state) {

// 变更状态

state.count++

}

}

})

你不能直接调用一个 mutation handler。这个选项更像是事件注册：“当触发一个类型为 increment 的 mutation 时，调用此函数。”要唤醒一个 mutation handler，你需要以相应的 type 调用 store.commit 方法：

store.commit('increment')

##### 提交载荷（Payload）

你可以向 store.commit 传入额外的参数，即 mutation 的 载荷（payload）：

// ...

mutations: {

increment (state, n) {

state.count += n

}

}

store.commit('increment', 10)

在大多数情况下，载荷应该是一个对象，这样可以包含多个字段并且记录的 mutation 会更易读：

// ...

mutations: {

increment (state, payload) {

state.count += payload.amount

}

}

store.commit('increment', {

amount: 10

})

##### 对象风格的提交方式

提交 mutation 的另一种方式是直接使用包含 **type** 属性的对象：

store.commit({

type: 'increment',

amount: 10

})

当使用对象风格的提交方式，整个对象都作为载荷传给 mutation 函数，因此 handler(handler) 保持不变：

mutations: {

increment (state, payload) {

state.count += payload.amount

}

}

##### Mutation 需遵守 Vue 的响应规则

既然 Vuex 的 store 中的状态是响应式的，那么当我们变更状态时，监视状态的 Vue 组件也会自动更新。这也意味着 Vuex 中的 mutation 也需要与使用 Vue 一样遵守一些注意事项：

1. 最好提前在你的 store 中初始化好所有所需属性。
2. 当需要在对象上添加新属性时，你应该

* 使用 Vue.set(obj, 'newProp', 123), 或者
* 以新对象替换老对象。例如，利用 stage-3 的对象展开运算符我们可以这样写：

state.obj = { ...state.obj, newProp: 123 }

##### 使用常量替代 Mutation 事件类型

使用常量替代 mutation 事件类型在各种 Flux 实现中是很常见的模式。这样可以使 linter 之类的工具发挥作用，同时把这些常量放在单独的文件中可以让你的代码合作者对整个 app 包含的 mutation 一目了然：

// mutation-types.js

export const SOME\_MUTATION = 'SOME\_MUTATION'

// store.js

import Vuex from 'vuex'

import { SOME\_MUTATION } from './mutation-types'

const store = new Vuex.Store({

state: { ... },

mutations: {

// 我们可以使用 ES2015 风格的计算属性命名功能来使用一个常量作为函数名

[SOME\_MUTATION] (state) {

// mutate state

}

}

})

用不用常量取决于你——在需要多人协作的大型项目中，这会很有帮助。但如果你不喜欢，你完全可以不这样做。

##### Mutation 必须是同步函数

一条重要的原则就是要记住 mutation 必须是同步函数。

为什么？请参考下面的例子：

mutations: {

someMutation (state) {

api.callAsyncMethod(() => {

state.count++

})

}

}

现在想象，我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。

##### 在组件中提交 Mutation

你可以在组件中使用 **this.$store.commit('xxx')** 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。

import { mapMutations } from 'vuex'

export default {

// ...

methods: {

...mapMutations([

'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`

// `mapMutations` 也支持载荷：

'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.commit('incrementBy', amount)`

]),

...mapMutations({

add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`

})

}

}

##### 下一步：Action

在 mutation 中混合异步调用会导致你的程序很难调试。例如，当你调用了两个包含异步回调的 mutation 来改变状态，你怎么知道什么时候回调和哪个先回调呢？这就是为什么我们要区分这两个概念。在 Vuex 中，**mutation 都是同步事务**：

store.commit('increment')

// 任何由 "increment" 导致的状态变更都应该在此刻完成。

为了处理异步操作，让我们来看一看 Action。

#### Action

Action 类似于 mutation，不同在于：

1. Action 提交的是 mutation，而不是直接变更状态。
2. Action 可以包含任意异步操作。

让我们来注册一个简单的 action：

const store = new Vuex.Store({

state: {

count: 0

},

mutations: {

increment (state) {

state.count++

}

},

actions: {

increment (context) {

context.commit('increment')

}

}

})

Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters。当我们在之后介绍到 Modules 时，你就知道 context 对象为什么不是 store 实例本身了。

实践中，我们会经常用到 ES2015 的 参数解构 来简化代码（特别是我们需要调用 commit 很多次的时候）：

actions: {

increment ({ commit }) {

commit('increment')

}

}

##### 分发 Action

Action 通过 **store.dispatch** 方法触发：

store.dispatch('increment')

乍一眼看上去感觉多此一举，我们直接分发 mutation 岂不更方便？实际上并非如此，还记得 mutation 必须同步执行这个限制么？Action 就不受约束！我们可以在 action 内部执行异步操作：

actions: {

incrementAsync ({ commit }) {

setTimeout(() => {

commit('increment')

}, 1000)

}

}

Actions 支持同样的载荷方式和对象方式进行分发：

// 以载荷形式分发

store.dispatch('incrementAsync', {

amount: 10

})

// 以对象形式分发

store.dispatch({

type: 'incrementAsync',

amount: 10

})

来看一个更加实际的购物车示例，涉及到**调用异步 API** 和**分发多重 mutation**：

actions: {

checkout ({ commit, state }, products) {

// 把当前购物车的物品备份起来

const savedCartItems = [...state.cart.added]

// 发出结账请求，然后乐观地清空购物车

commit(types.CHECKOUT\_REQUEST)

// 购物 API 接受一个成功回调和一个失败回调

shop.buyProducts(

products,

// 成功操作

() => commit(types.CHECKOUT\_SUCCESS),

// 失败操作

() => commit(types.CHECKOUT\_FAILURE, savedCartItems)

)

}

}

注意我们正在进行一系列的异步操作，并且通过提交 mutation 来记录 action 产生的副作用（即状态变更）。

##### 在组件中分发 Action

你在组件中使用 this.$store.dispatch('xxx') 分发 action，或者使用 mapActions 辅助函数将组件的 methods 映射为 store.dispatch 调用（需要先在根节点注入 store）：

import { mapActions } from 'vuex'

export default {

// ...

methods: {

...mapActions([

'increment', // 将 `this.increment()` 映射为 `this.$store.dispatch('increment')`

// `mapActions` 也支持载荷：

'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.dispatch('incrementBy', amount)`

]),

...mapActions({

add: 'increment' // 将 `this.add()` 映射为 `this.$store.dispatch('increment')`

})

}

}

##### 组合 Action

Action 通常是异步的，那么如何知道 action 什么时候结束呢？更重要的是，我们如何才能组合多个 action，以处理更加复杂的异步流程？

首先，你需要明白 **store.dispatch** 可以处理被触发的 action 的处理函数返回的 Promise，并且 **store.dispatch** 仍旧返回 Promise：

actions: {

actionA ({ commit }) {

return new Promise((resolve, reject) => {

setTimeout(() => {

commit('someMutation')

resolve()

}, 1000)

})

}

}

现在你可以：

store.dispatch('actionA').then(() => {

// ...

})

在另外一个 action 中也可以：

actions: {

// ...

actionB ({ dispatch, commit }) {

return dispatch('actionA').then(() => {

commit('someOtherMutation')

})

}

}

最后，如果我们利用 **async / await**，我们可以如下组合 action：

// 假设 getData() 和 getOtherData() 返回的是 Promise

actions: {

async actionA ({ commit }) {

commit('gotData', await getData())

},

async actionB ({ dispatch, commit }) {

await dispatch('actionA') // 等待 actionA 完成

commit('gotOtherData', await getOtherData())

}

}

一个 **store.dispatch** 在不同模块中可以触发多个 action 函数。在这种情况下，只有当所有触发函数完成后，返回的 Promise 才会执行。

#### Module

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。

为了解决以上问题，Vuex 允许我们将 store 分割成**模块（module）**。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割：

const moduleA = {

state: { ... },

mutations: { ... },

actions: { ... },

getters: { ... }

}

const moduleB = {

state: { ... },

mutations: { ... },

actions: { ... }

}

const store = new Vuex.Store({

modules: {

a: moduleA,

b: moduleB

}

})

store.state.a // -> moduleA 的状态

store.state.b // -> moduleB 的状态

##### 模块的局部状态

对于模块内部的 mutation 和 getter，接收的第一个参数是模块的局部状态对象。

const moduleA = {

state: { count: 0 },

mutations: {

increment (state) {

// 这里的 `state` 对象是模块的局部状态

state.count++

}

},

getters: {

doubleCount (state) {

return state.count \* 2

}

}

}

同样，对于模块内部的 action，局部状态通过 context.state 暴露出来，根节点状态则为 context.rootState：

const moduleA = {

// ...

actions: {

incrementIfOddOnRootSum ({ state, commit, rootState }) {

if ((state.count + rootState.count) % 2 === 1) {

commit('increment')

}

}

}

}

对于模块内部的 getter，根节点状态会作为第三个参数暴露出来：

const moduleA = {

// ...

getters: {

sumWithRootCount (state, getters, rootState) {

return state.count + rootState.count

}

}

}

##### 命名空间

默认情况下，模块内部的 action、mutation 和 getter 是注册在全局命名空间的——这样使得多个模块能够对同一 mutation 或 action 作出响应。

如果希望你的模块具有更高的封装度和复用性，你可以通过添加 **namespaced: true** 的方式使其成为带命名空间的模块。当模块被注册后，它的所有 getter、action 及 mutation 都会自动根据模块注册的路径调整命名。例如：

const store = new Vuex.Store({

modules: {

account: {

namespaced: true,

// 模块内容（module assets）

state: { ... }, // 模块内的状态已经是嵌套的了，使用 `namespaced` 属性不会对其产生影响

getters: {

isAdmin () { ... } // -> getters['account/isAdmin']

},

actions: {

login () { ... } // -> dispatch('account/login')

},

mutations: {

login () { ... } // -> commit('account/login')

},

// 嵌套模块

modules: {

// 继承父模块的命名空间

myPage: {

state: { ... },

getters: {

profile () { ... } // -> getters['account/profile']

}

},

// 进一步嵌套命名空间

posts: {

namespaced: true,

state: { ... },

getters: {

popular () { ... } // -> getters['account/posts/popular']

}

}

}

}

}

})

启用了命名空间的 getter 和 action 会收到局部化的 getter，dispatch 和 commit。换言之，你在使用模块内容（module assets）时不需要在同一模块内额外添加空间名前缀。更改 namespaced 属性后不需要修改模块内的代码。

##### 在带命名空间的模块内访问全局内容（Global Assets）

如果你希望使用全局 state 和 getter，rootState 和 rootGetter 会作为第三和第四参数传入 getter，也会通过 context 对象的属性传入 action。

若需要在全局命名空间内分发 action 或提交 mutation，将 { root: true } 作为第三参数传给 dispatch 或 commit 即可。

modules: {

foo: {

namespaced: true,

getters: {

// 在这个模块的 getter 中，`getters` 被局部化了

// 你可以使用 getter 的第四个参数来调用 `rootGetters`

someGetter (state, getters, rootState, rootGetters) {

getters.someOtherGetter // -> 'foo/someOtherGetter'

rootGetters.someOtherGetter // -> 'someOtherGetter'

},

someOtherGetter: state => { ... }

},

actions: {

// 在这个模块中， dispatch 和 commit 也被局部化了

// 他们可以接受 `root` 属性以访问根 dispatch 或 commit

someAction ({ dispatch, commit, getters, rootGetters }) {

getters.someGetter // -> 'foo/someGetter'

rootGetters.someGetter // -> 'someGetter'

dispatch('someOtherAction') // -> 'foo/someOtherAction'

dispatch('someOtherAction', null, { root: true }) // -> 'someOtherAction'

commit('someMutation') // -> 'foo/someMutation'

commit('someMutation', null, { root: true }) // -> 'someMutation'

},

someOtherAction (ctx, payload) { ... }

}

}

}

##### 在带命名空间的模块注册全局 action

若需要在带命名空间的模块注册全局 action，你可添加 root: true，并将这个 action 的定义放在函数 handler 中。例如：

{

actions: {

someOtherAction ({dispatch}) {

dispatch('someAction')

}

},

modules: {

foo: {

namespaced: true,

actions: {

someAction: {

root: true,

handler (namespacedContext, payload) { ... } // -> 'someAction'

}

}

}

}

}

##### 带命名空间的绑定函数

当使用 mapState, mapGetters, mapActions 和 mapMutations 这些函数来绑定带命名空间的模块时，写起来可能比较繁琐：

computed: {

...mapState({

a: state => state.some.nested.module.a,

b: state => state.some.nested.module.b

})

},

methods: {

...mapActions([

'some/nested/module/foo', // -> this['some/nested/module/foo']()

'some/nested/module/bar' // -> this['some/nested/module/bar']()

])

}

对于这种情况，你可以将模块的空间名称字符串作为第一个参数传递给上述函数，这样所有绑定都会自动将该模块作为上下文。于是上面的例子可以简化为：

computed: {

...mapState('some/nested/module', {

a: state => state.a,

b: state => state.b

})

},

methods: {

...mapActions('some/nested/module', [

'foo', // -> this.foo()

'bar' // -> this.bar()

])

}

而且，你可以通过使用 createNamespacedHelpers 创建基于某个命名空间辅助函数。它返回一个对象，对象里有新的绑定在给定命名空间值上的组件绑定辅助函数：

import { createNamespacedHelpers } from 'vuex'

const { mapState, mapActions } = createNamespacedHelpers('some/nested/module')

export default {

computed: {

// 在 `some/nested/module` 中查找

...mapState({

a: state => state.a,

b: state => state.b

})

},

methods: {

// 在 `some/nested/module` 中查找

...mapActions([

'foo',

'bar'

])

}

}

##### 给插件开发者的注意事项

如果你开发的插件（Plugin）提供了模块并允许用户将其添加到 Vuex store，可能需要考虑模块的空间名称问题。对于这种情况，你可以通过插件的参数对象来允许用户指定空间名称：

// 通过插件的参数对象得到空间名称

// 然后返回 Vuex 插件函数

export function createPlugin (options = {}) {

return function (store) {

// 把空间名字添加到插件模块的类型（type）中去

const namespace = options.namespace || ''

store.dispatch(namespace + 'pluginAction')

}

}

##### 模块动态注册

在 store 创建之后，你可以使用 store.registerModule 方法注册模块：

// 注册模块 `myModule`

store.registerModule('myModule', {

// ...

})

// 注册嵌套模块 `nested/myModule`

store.registerModule(['nested', 'myModule'], {

// ...

})

之后就可以通过 store.state.myModule 和 store.state.nested.myModule 访问模块的状态。

模块动态注册功能使得其他 Vue 插件可以通过在 store 中附加新模块的方式来使用 Vuex 管理状态。例如，vuex-router-sync 插件就是通过动态注册模块将 vue-router 和 vuex 结合在一起，实现应用的路由状态管理。

你也可以使用 store.unregisterModule(moduleName) 来动态卸载模块。

**注意**，你不能使用此方法**卸载静态模块**（即创建 store 时声明的模块）。

##### 保留 state

在注册一个新 module 时，你很有可能想保留过去的 state，例如从一个服务端渲染的应用保留 state。你可以通过 **preserveState** 选项将其归档：store.registerModule('a', module, { preserveState: true })。

当你设置 preserveState: true 时，该模块会被注册，action、mutation 和 getter 会被添加到 store 中，但是 state 不会。这里假设 store 的 state 已经包含了这个 module 的 state 并且你不希望将其覆写。

##### 模块重用

有时我们可能需要创建一个模块的多个实例，例如：

1. 创建多个 store，他们公用同一个模块 (例如当 runInNewContext 选项是 false 或 'once' 时，为了在服务端渲染中避免有状态的单例)
2. 在一个 store 中多次注册同一个模块

如果我们使用一个纯对象来声明模块的状态，那么这个状态对象会通过引用被共享，导致状态对象被修改时 store 或模块间数据互相污染的问题。

实际上这和 Vue 组件内的 data 是同样的问题。因此解决办法也是相同的——使用一个**函数来声明模块状态**（仅 2.3.0+ 支持）：

const MyReusableModule = {

state () {

return {

foo: 'bar'

}

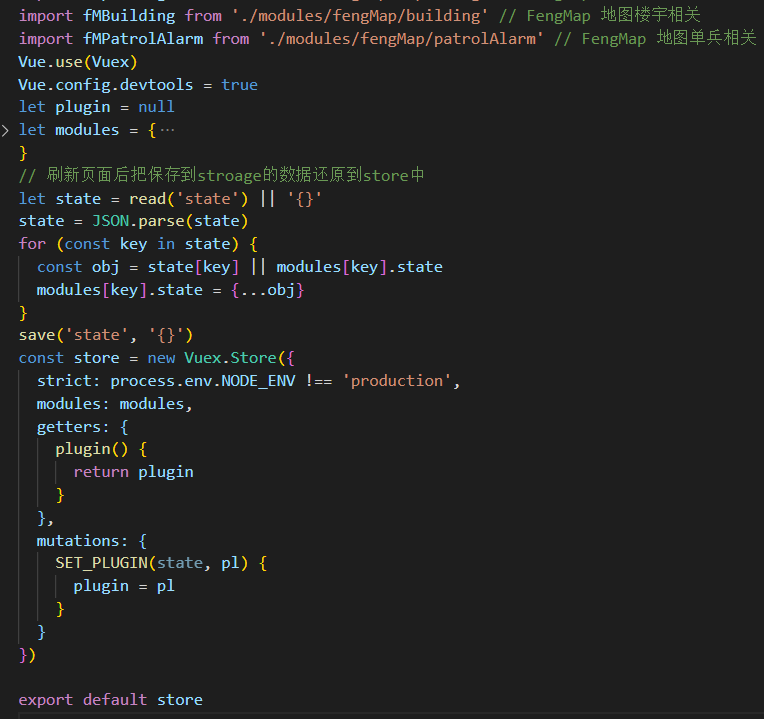
},

// mutation, action 和 getter 等等...

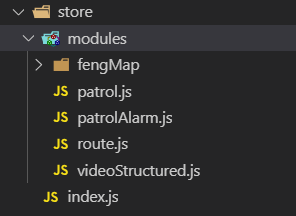
}

### 其他

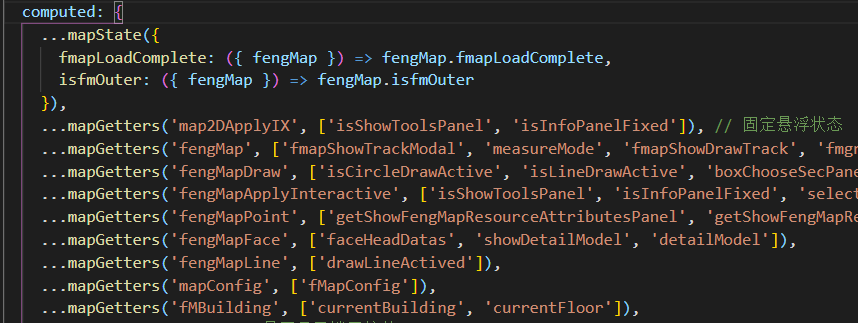
当项目十分庞大复杂时，需要配置vuex的路由，如下图：

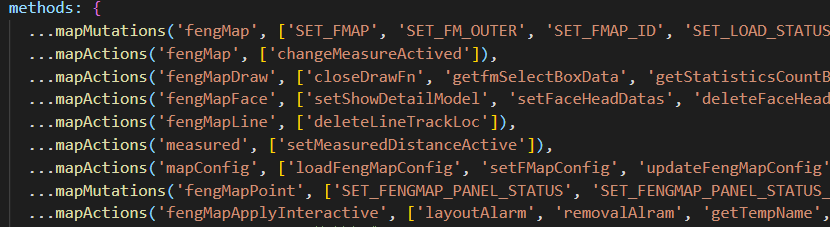


文件结构如下图：



在vue中使用时，引入如下图：





mapState用法不变，花括号内为store文件的index.js中modules所有保存的自命名的引入文件名，如第一图中的fMPatrolAlarm、fMBuilding。

mapGetters用法，如第二图中，若方法在patrol.js文件中，则：

...mapGetters(['方法名', ...])

若方法在如fengMap文件中的js文件中，则：

...mapGetters('文件名', ['方法名', ...])

mapMutations、mapActions用法同上。

在store中使用时，

state、getters、mutations、actions的用法不变

需要指出的是，actions会传2个参数。

第一个参数为Object，包含当前文件的所有state、getters、mutations、actions；

第二个参数为传参。

当actions在使用dispatch方法(调用actions方法)或commit方法(调用mutations方法)时，方法会指向当前文件下的方法

若要调用其他文件的方法，操作如下：

dispatch('patrol2DAlarm/getMap2DSingleAlarm', res.data, { root: true })

第一个参数，方法路径；

第二个参数，传参；

第三个参数，是否从根开始。

该例子会从调用Store/modules/patrol2DAlarm文件夹下的getMap2DSingleAlarm方法。

当mutations在使用时，会传2个参数。

第一个参数为当前文件的state；

第二个参数为传参。

当mutations互调时，使用方式如下：

this.commit('SORT\_PLAYBACK\_LIST', data )

但需要注意的是，该this其会指向当前Store实例，即在index.js所看到的。

若要调用的方法是保存在某个文件夹中，该直接调用方法名可能会找不到或指向其他同名方法(如直接存放在modules/\*\*.js中的同名方法或存放在modules/文件名/\*\*.js中的同名方法)。

此时在调用方法是建议

this.commit('fengMapDraw/SORT\_PLAYBACK\_LIST', { main: 'poinList' }, { root: true })

情况同上actions，只是更强制。

# 汇总

## JavaScript

### 关于url传递特殊字符or中文

#### 转译

将url转译为Unicode编码

可使用，escape()、encodeURI()、encodeURIComponent()

##### Escape()

escape()不能直接用于URL编码，它的真正作用是返回一个字符的Unicode编码值。比如"春节"的返回结果是%u6625%u8282，，escape()不对"+"编码 主要用于汉字编码，现在已经不提倡使用。

##### encodeURI()

encodeURI()是Javascript中真正用来对URL编码的函数。 编码整个url地址，但对特殊含义的符号"; / ? : @ & = + $ , #"，也不进行编码。对应的解码函数是：decodeURI()。

##### encodeURIComponent()

encodeURIComponent() 能编码"; / ? : @ & = + $ , #"这些特殊字符。对应的解码函数是decodeURIComponent()。

假如要传递带&符号的网址，所以用encodeURIComponent()

#### 判断

##### 用正则表达式判断

let reg = new RegExp('[\\u4E00-\\u9FFF]+', 'g')

if (reg.test(value)) { alert('用户名不能包含汉字')}

##### 用 Unicode 字符范围判断

chkstrlen: function (str) {

letr strlen = 0;

for (let i = 0; i < str.length; i++) {

if (str.charCodeAt(i) > 255) //如果是汉字，则字符串长度加2

strlen += 2;

else

strlen++;

}

return strlen;

}