前端学习笔记

1. 执行上下文

定义：简而言之，执行上下文评估和执行javascript代码环境的抽象概念；

类型：全局执行上下文、函数执行上下文、eval上下文；

执行栈：调用栈，一种拥有LIFO（先进后出）数据结构的栈，被用来存储代码执行时创建的所有上下文；

创建上下文：两个阶段

1. 创建阶段
   1. this值的决定，即this绑定；
   2. 创建词法组件环境；
      1. 词法环境定义：简单来说词法环境是一种持有标识符—变量映射的结构。（这里的标识符指的是变量/函数的名字，而变量是对实际对象[包含函数类型对象]或原始数据的引用）；
      2. 组成：环境记录器（存储变量和函数声明的实际位置）、一个外部环境的引用（意味着它可以访问其父级词法环境（作用域））；
   3. 创建变量环境变量
   4. 不同：在 ES6 中，词法环境组件和变量环境的一个不同就是前者被用来存储函数声明和变量（let 和 const）绑定，而后者只用来存储 var 变量绑定。
2. 执行阶段：完成对所有这些变量的分配，最后执行代码。
3. 作用域

定义：程序中定义变量的区域，它规定了如何查找变量，也就是确定了当前执行的代码对变量的访问权限

词法作用域：静态作用域，函数的作用域在函数定义的时候就已经决定了。javascript采用的就是词法作用域；

动态作用域：函数的作用域是在函数执行时才决定的，如.bash文件执行；

代码示例1：

var value = 1;

function foo() {

console.log(value);

}

function bar() {

var value = 2;

foo();

}

bar();

打印结果：1；因为js是词法作用域，定义时就已经决定了：函数执行时，函数内部没有value，查找到定义位置的上一级，所以等于1，而不是2；

分类：

1. 全局作用域
2. 函数作用域：一般情况下，函数之前的作用域独立，不能互相访问；
3. 块级作用域：es6引用，使用let和const创建的变量拥有块级作用域

暂时性死区：块级作用域；

示例1：

function foo() {

console.log(bar)

let bar = 3

}

foo() // 会报错：Uncaught ReferenceError: bar is not defined。括号 { 与变量生命前的区域就是暂时性死区；

示例2：

function foo(arg1 = arg2, arg2) {

console.log(`${arg1} ${arg2}`)

}

foo(undefined, 'arg2')

// Uncaught ReferenceError: arg2 is not defined 当第一个参数缺省时，执行 arg1 = arg2 会当作暂时性死区处理：

注：第一个参数换成null就不会报错

作用域链：变量查找时，优先在当前作用域进行查找，当找不到时，就会向上一层作用域查找，直到查到到全局作用域，这样作用域之间就形成一条链，就是作用域链

1. 闭包

定义：在一个外层函数中，返回另一个函数，而这个返回的函数使用了外层函数的变量，这样，我们就可以通过这个返回的函数去访问原外层函数内部的变量，这个返回的函数就是闭包；

作用：封装具有私有变量；

弊端：内存泄漏，用完及时清除

1. this

定义：是在调用函数执行时根据上下文决定的

规则：

1. 简单调用函数时，严格模式是指向undefined，否则指向window/global；
2. 利用new构造函数执行，绑定到新创建的对象上；
3. bind/call/apply显示调用，绑定到指向参数的对象上；

注：非严格模式下，如绑定值为null和undefined，会自动绑定到window，严格模式，制定值本身；如为其他简单数据类型，则指向该原始值自动包装对象

1. 箭头函数没有自己的this，根据外层上下文绑定this；

箭头函数和普通函数的重要区别:

1. 没有自己的this.super\arguments\和new。target绑定
2. 不能使用new来调用；
3. 没有原型对象；
4. 不可以改变this的绑定；
5. 形参名称不能重复

代码示例：

var student = {

name: '若川',

doSth: function(){

console.log(this.name);

return () => {

console.log('arrowFn:', this.name);

}

}

}

var person = {

name: 'person',

}

student.doSth().call(person); // '若川' 'arrowFn:' '若川'

student.doSth.call(person)(); // 'person' 'arrowFn:' 'person'

1. 一般上下文的对象调用，this指向该对象；

new的过程：

1. 创建一个新的对象；
2. 这个对象会被执行原型链接；
3. 生成的新对象会绑定到函数调用的this；
4. 通过new创建的每个对象都最终被[[prototype]]链接到这个函数的 prototype对象上；
5. 如果函数没有返回对象类型Object(Function, Array, Date, RegExg), 那么new表达式中的函数调用会自动返回这个新的对象
6. 代码示例：

function MyNew( fn, ...args ) {

Let obj = Object.create(fn.prototype)

Let res = fn.apply(obj, args)

Return typeof res === ‘object’ ? res : obj

}

Function Person(name) {

This.name = name

}

Let newP = myNew(Person, ‘kelvin’’)

Apply实现：就是把调用函数添加到目标对象中去

Function.prototype.myApply = function(context) {

Var context = context || window

Let fn = Symbol();

Context[fn] = this

Var args = arguments[1]

If( args === undefined ) {

Context[fn]()

}else {

Context[fn](…args)

}

Delete context.fn

}

Apply/call/bind总结:

1. 三者都可以改变this指向，指向第一个参数；
2. Apply/call会立即执行函数，bind是返回一个函数，可待后续执行；
3. Call的参数需要逐个列出，apply是一个数组
4. 拷贝

浅拷贝：只是复制值的引用，而非复制真正的值；

方法：

1. ...展开运算符；
2. Object.assign
3. Concat
4. Slice

深拷贝：真正复制值，复制后改变值不会影响原有值；

深拷贝方法：

1. JSON.stringify/parse:undefined、function、Symbol在转化时会被 忽略；
2. 递归拷贝复制：

Function clone(x) {

// 如果不是对象或者数组，直接返回

If( typeof x !== ‘object ) return x

// 如果是数组

Let res;

If( x instanceof Array ) {

For(let i=0;i<x.length; i++) {

// 避免一层死循环 a.b = a

Res[i] = x[i] === x ? res : clone[x]

}

}

If() { // 如果是object

For(let key in x) {

If(x.hasWonProperty(key)) {

// 避免一层死循环 a.b = a

Res[key] = x[key] === x ? res : clone[x]

}

}

}

Return res

}

1. 循环克隆：

Function cloneLoop(x) {

Const t = type(x) // 一个类型判断函数

Let root = x;

If(t = ‘array’’ ) {

Root = []

}else if(t === ‘object’) {

Root = {}

}

Const loopList = [

{

Parent: node,

Key: undefined,

Data: x

}

]

While(loopList.length) {

// 深度优先

Const node = loopList.pop();

Const parent = node.parent;

Const key = node.key;

Const data = node.data;

Const tt = type(data)

// 初始化赋值目标，key为undefined则复制到赋元素，否则复制到子元素

Let res = parent;

If(typeof key !== ‘undefined’) {

Res = parent[key] = tt === ‘array’ ? [] : {}

}

If(tt === ‘array’) {

For( let i = 0; i<data.length; i++ ) {

If(data[i] === data) {

Res[i] = res

}else if(isClone(data[i])) { // 是数组或者对象

// 下一次循环

loopList.push({

Parent: res,

Key: i,

Data: data[i]

})

}else {

Res[i] = data[i]

}

}

}

else if( tt === ‘object’) {

For(let k in data) {

If(hasOwnProperty(data, k)) {

// 循环引用

If(data[k] === data) {

Res[k] = res

}else if(isClone(data[k])) {

// 下一次循环

loopList.push({

Parent: res,

Key: i,

Data: data[k]

})

}else {

Res[k] = data[k]

}

}

}

}

}

Return root

}

1. Eventloop

定义：event loop是让js做到即是单线程，又绝对不会阻塞的核心机制，也是 js并发模型的基础，是用来协调各种事件、用户交互、脚本执行、UI渲染、 网络请求等的一种机制

执行&运行：

执行：一般执行依赖与环境，比如node/浏览器等，js在不同环境的执行 机制可能不相同；

运行：指的是js的解释引擎，这是统一的；

什么是EventLoop：

1. 主线程至上而下执行所有代码；
2. 同步任务直接进入主线程执行，异步任务则进入event table 并注册 回调函数；
3. 异步任务完成后，event table会将这个回调函数移入event queue；
4. 主线程任务执行完毕后， 会从event queue中读取任务，进入到主线 程执行；
5. 在这个event queue中又会分宏任务和微任务两种，先完成微任务队 列再去执行宏任务队列
6. 上述动作会不断轮询，就形成了eventloop；



浏览器event loop与nodejs环境的区别：

（1）浏览器的微任务队列是在每个宏任务队列执行完之后执行；nodejs会 在事件循环的各个阶段之间执行，也就是一个阶段执行完毕，就会去 执行微任务队列；

（2）举例：有两个settimeout代码里面嵌套promise微任务，在浏览 器里面 的话，是逐步执行完每一个settimeout的所有任务，才会执行 下一个 settimeout的代码；在nodejs，就会按照队列类型来执行（同 步->微 任务->宏任务）

1. 函数式编程

定义：是一种编程规范。比起命令式编程，更注重执行的结果而非执行的过程，倡导利用若干简单的执行单元让计算结果不断推进，而不是直接设计一个复杂的执行过程；

特点：

1. 函数是一等公民：函数跟其他数据类型一样，可以作为其他函数 的参数进行输入；
2. 数据不可变：要求所有的数据都是不可变的，如果你想修改一个 对象，应该是新建一个对象，而不是在原有的对象上进行修改；
3. 引用透明：函数不依赖外部的变量；
4. 纯函数：就是没有副作用的函数，相同的输入，相同的输出，不 修改外部数据
5. 惰性计算：函数只在需求时执行，不产生无意义的中间变量；

好处：

1. 代码简洁、开发快速：函数复用高；
2. 接近自然语言：易于理解：使用大量声明式代码，没有乱七八槽 的嵌套循环；
3. 更少出错率：相同的输入，相同的输出，便于测试调试；

缺陷：

1. 性能：一些方法的过度包装，产生一些上下文切换的开销。而且 js本身不属于函数式语言，函数式比直接的指令语句要慢，因为需要 引擎对指令做优化，如map就比纯粹的循环慢；
2. 资源占用：数据不可变，所有需要新建对象，导致垃圾回收压力 大；
3. Web worker

介绍：js一直是单线程环境，我们无法同时运行连个js脚本；html5新规 范中，实现了web worker来引入多线程技术，能让我们在页面主运行 的hs线程加载运行单独的一个或者多个js线程，用来处理一些复杂的 运算，而不会影响主线程UI渲染等，从而可以提高性能效率。但是这 不意味改变js本身单线程的性质，这种事浏览器提供的一个api/能力；

分类：

1. 专用线程：仅能被创建它的脚本使用；
2. 共享线程：能够在不同的脚本使用，一个共享线程对应多个主线 程；

创建：

Var worker = new worker(‘./worker.js’)

注意：加载的worker脚本跟当前创建的脚本需同域；

通信：

1. 传递信息： postMessage
2. 接受信息： onmessage
3. 举例：

Main.js:

var worker = new Worker('worker.js')

worker.onmessage = function(e) {

console.log(e)

}

Worker.js:

console.log('123')

self.postMessage('heelo work')

终止：worker.terminate();

错误监控： worker.onerroe = function(err) {};

环境与作用域：

没有window全局对象，无法访问DOM，可以通过self来访问当前worker实例，一般只能执行纯js运算；

可以使用的部门浏览器api：

setTimeout/setInerval等；

XMLHttpRequest

Navigator

Location

worker中加载外部脚本：使用 importScripts函数



Subworker： worker中的worker：一个worker中再去创建一个worker，目前部 分浏览器还没实现；

共享worker：

New ShareWorker();

应用：

1. 对于图像、视频、饮品的解析处理；
2. canvas中的图像计算处理；
3. 大量的ajax情趣或者网络服务轮询；
4. 大量数据计算处理（排序、检索、过滤、分析）；
5. Vue

mvvm原理：数据劫持+发布订阅模式

数据劫持：通过Object.defineProperty()递归对data的数据每一层都添加上get 和set：oberve函数；

数据编译：complie函数：正则匹配到关键数据插值，然后替换成真实数据，利用replace递归遍历节点，替换成真实数据

发布订阅：当获取数据自动调用get方法时，执行addSubs方法将watcher添加到订阅事件中，当修改数据时，自动调用set方法，执行notify方法将watcher中的update方法执行；

双向数据绑定：v-model: 利用 :value与@input组合的语法糖；

总结步骤：

1. 通过Object.defineProperty的get和set进行数据劫持；
2. 通过遍历data数据进行数据代理到this上；
3. 通过{{}} 对数据进行编译；
4. 通过发布订阅模式进行数据与视图的同步更新；

nextTick：

异步队列：Vue异步执行DOM更新。只要观察到数据变化，Vue将开启一个队列，并缓冲在同一事件循环中发生的所有数据变化。如果一个watcher被触发多次，只会被推入队列一次。这种在缓冲中去除重复数据对于避免不必要的计算和DOM操作非常重要。然后，在下一次事件循环的tick中，Vue刷新队列并执行实际工作。Vue在异步队列中的实现使用原生的Promise.then以及MessageChannel，如果执行环境不支持，会采用setTimeout来代替；

Object.defineProperty Vs Proxy：

1. Object.defineProperty只能劫持对象的属性：需要遍历每一个属性，如果属性也是对象，需要深度遍历，proxy是直接代理对象：不需要遍历操作；
2. Object.defineProperty只能监听原有的属性，无法监听新增属性，需要调用$set（observer），proxy不需要；
3. Proxy对数组监听支持也友好，不需要改写数组方法

Virtual DOM：

定义：是对DOM的抽象，本质上是javascript对象；

为什么需要：

1. 框架在patch过程中尽可能一次性将差异更新到DOM中，而不是更 新整个试图减少DOM的频繁操作；
2. 多人开发过程中在代码层面无需过多关注DOM操作，可以大大提高 开发效率，同时避免因为代码不严谨，过多操作DOM降低性能；
3. Virtual DOM最初的目的，就是为了更好的跨平台，转换成js对象， 再转换成其他原生组件等容易实现，或者SSR等；

关键要素：参照snabbdom.js

Type： 类型，div；

Key： vnode唯一的id；

Data： 包括属性，事件等

Children：子vnode

Text：文本

Elm： 对应的dom

DOM diff 节点对比：

1. 只会在同层级进行比较，不会跨层级比较；
2. 旧节点不存在，插入新节点，新节点不存在，删除旧节点；
3. 新旧节点都是VNode，且新旧节点tag相同

A：对比新旧节点的属性；

B：对比新旧节点的子节点差异，通过key值进行重排序，key值相同 节点继续往下便利

1. 新旧节点如果都是VText，怕大怒你两者文本是否发生变化；
2. 其他情况直接新节点代替旧节点