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