闭包和作用域链

**闭包的概念:**

函数执行,形成一个不销毁的局部作用域或者局部栈内存,不仅可以**保护**局部变量不受外界干扰,还可以在不销毁的局部栈内存中**保存局部变量**,供后期使用

**产生闭包的条件:**

函数嵌套

内部函数应用了外部函数的数据(函数或者变量)

**function fn(){**

**return function(){}**

**}**

**var f = fn(); //是闭包 => 柯理化函数**

**var utils = (function() {**

**return {}**

**})(); //也是闭包 => 惰性函数**

**闭包的两个作用:**

**延长外部函数局部变量的生命周期**

**保护:保护局部变量不受外界干扰**

**保存:形成不销毁的栈内存,把一些保存下来,方便后面的调取使用**

理解:  
真实项目中为了保证js的性能,应该尽可能减少闭包的使用

(堆栈内存不及时销毁,是很耗性能的)

我们在封装类库插件的时候,也会把自己的程序都放到闭包中保护起来,防止和用户的程序冲突,但是我们有时候又需要暴露一些方法给用户使

**①jq的方式==>把他们定义为window的全局属性**

(function() {

function jQuery() {

//....

}

//....

window.jQuery=window.$=jQuery;

})();

**②Zepto的方式==>基于return把需要的方法抛到全局**

var Zepto = (function() {

//....

var $ = {

xxx: function() {}

//....

}

return $;

})();

Zepto.xxx(); //外部这样使用

作用域:

传统的js(es6):

只支持函数级作用域,函数里面定义的变量是局部的,不在函数里面定义就是全局的

es6:

除了支持函数级作用域,还支持块级作用域( 语法块 {} )

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

tabList[i].onclick = function() {

console.log(i);

}

}

第一种解决方案:自定义属性

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

tabList[i].myIndex = i; //每次把当前的所以存到自己的自定义属性中

tabList[i].onclick = function() {

console.log(this.myIndex);

//方法执行的时候,方法中的this是当前操作的元素

}

}  
第二种方式:闭包

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

tabList[i].onclick = (function (n){

//自执行函数执行,形参一个局部作用域,不释放,返回值(return)给这个onclick事件

//局部作用域不释放的原因:返回的函数对应的堆地址被外面的事件占用了

//此处onclick绑定的是返回的小函数,点击到时候,执行的是小函数,自执行函数在给事件赋值的时候就已经自执行了

var i = n;

return function() {

console.log(i);

}

})(i); //把本次全局的i当做实参传给自执行函数的形参n

}

第三种方式:

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

(function(n) {

tabList[n].onclick = function() {

console.log(n);

}

})(i);

}

第四也是最好的一种方式:

//基于es6中的let来创建变量,是存在块级作用域的

//作用域:全局作用域 私有作用域 块级作用域

//(一般用{}包起来的都是块级作用域,前提是es6的语法规范下,如let)

//对象的大括号不是块级作用域

for(let i=0;i<tabList.length;i++) {

**//循环体也是块级作用域,初始值设置的变量是当前本次块级作用域的变量**

**//循环执行了多少次,就形成多少个块级作用域,各个块级作用域的变量都是局部的**

tabList[i].onclick = function() {

console.log(i);

}

}

典型例题:

for(var i=0;i<5;i++) {

setTimeout(function(){

console.log(i);

},100)

}

for(let i=0;i<5;i++) {

setTimeout(function(){

console.log(i);

},100)

}

for(var i=0;i<5;i++) {

setTimeout((function(i) { **//这里第一步是形参赋值, 定义了一个局部的i等于全局i**

return function() { //因为返回的函数被外部占用了,所以局部i不释放

setTimeout(function(){

console.log(i);

},100)

}

})(i),1000);

}

for(var i=0;i<5;i++) {

(function(i) { **//这里第一步是形参赋值, 定义了一个局部的i等于全局i**

setTimeout(function(){

console.log(i);

},100)

})(i);

}

变量提升:

带 var 的只声名,不定义,为undefined

**(var fn = function(){})匿名函数只声名,不定义,为undefined**

**普通的函数 function 同时进行声名和定义**

变量提升只发生在当前作用域(例如: 开始加载页面的时候,只对全局作用域下的进行提升,因为此时函数中存储的都是字符串而已)

console.log(a); //undefined

console.log(window.a); //undefined

console.log(‘a’ in window); //true,在全局作用域中声名了a,此时a当做属性值赋值给window了,只不过此时还没有给a赋值,默认值为undefined

var a = 12; //全局变量修改,window的属性值也跟着修改,window改,全局也改!!!

console.log(a); //报错

console.log(window.a); //undefined,因为全局对象下没有这个属性

console.log(‘a’ in window); //false,全局对象下没有这个属性

**a = 12; //window.a = 12 的简写,没有变量提升**

console.log(a); //12

console.log(window.a); //12

**整理:  
加了var, 是全局变量,由于全局变量和window对象有映射关系,就给了window对象加了一个属性a**

**不加 var, 这个a 本质是window下的一个属性**

var a = 12,

b = 12; //这样写b是带var的

var a = b = 12; //这样写a是带var的, b是不带var的

**重要例题:**

console.log(a); //undefined

console.log(b); //undefined

var a = 12,

b = 12;

function fn() {

console.log(a); //undefined,局部变量,有变量提升,此时不使用全局变量

console.log(b); //12,没有变量提升,使用外部的全局变量

var a = b = 13;

//私有作用域中,带var是局部变量,与外界没有任何关系

**//局部作用域中,不带var,不是局部变量, 会像上级作用域查找,看是否为上级的变量,不是就继续向上查找,一直找到window为止(我们把这种查找机制叫做:作用域链)**

**//**当前函数执行,a的上级作用域和在哪执行没关系,**而是和定义时的上级作用域相关**

**//在作用域链查找的过程中,如果找到window也没有找到这个变量,就给window设置了一个属性,值为变量的值window.b = 13**

console.log(a); //13

console.log(b); //13

}

fn();

console.log(a); //12

console.log(b); //13

**重要的题目:**

var ary = [12,23];

function fn(ary) {

console.log(ary); // [12,23]

ary[0] = 100;

ary = [100]; //开辟了一个新的空间

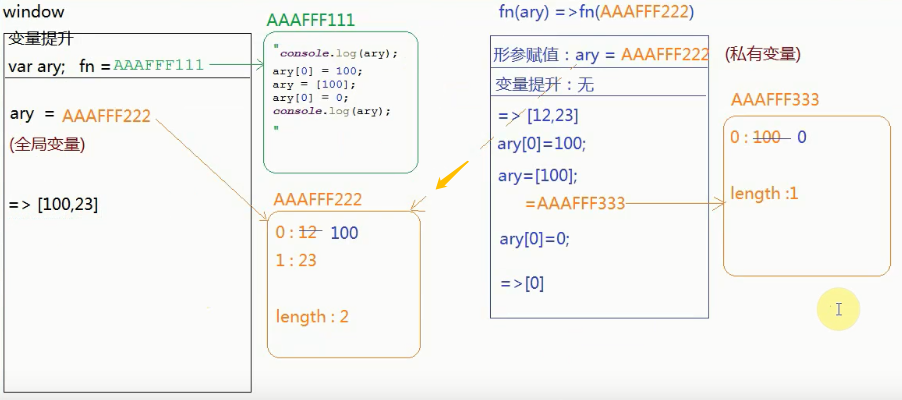
ary[0] = 0;

console.log(ary); //[0]

}

fn(ary);

console.log(ary);  **//[100,23]**



**条件判断下的变量提升规则:**

①在当前作用域下,不管条件是否成立,都要进行变量提升

②带var的,还是只声名不定义

②普通的function函数,老版本渲染机制下,既声名又定义

但是为了迎合es6的块级作用域,新版本浏览器都只是声名,没有定义,类似var

**重要例题:(新版本报错, 老版本是false,false)**

f = function () {return true;};

g = function () {return false;};

~function() {

// g()变量提升,只声名,不定义,为undefined

if(g() && [] == ![]) { /**/执行 undefined() 报错**,undefined is not a function

f = function() {return false;};

function g() {return true;}; //变量提升,只声名,不定义,是局部变量

}

}();

console.log(f());

console.log(g());

**[] == ![] 具体判断过程:**

![] 数组取反,[]为ture,取反为false

原式变为[] == false

对象和布尔值比较,两边变成 0 == 0

所以最终结果为ture

**下面两道重要的题目,对比分析(默认新版本浏览器,水很深):  
第一题:**

console.log(fn); //undefined

if(1 === 1) {

function fn() {

console.log(‘ok’);

}

}

console.log(fn); //fn函数的代码

**第二题:**

console.log(fn); //还是undefined

if(1 === 1) {

**console.log(fn) ; //fn函数的代码**

**//理由:当条件成立,进入到判断题当中(es6中{}是一个块级作用域),第一件事不是代码的执行,而是类似变量提升那样,在{}以内将fn既声名又定义了**

function fn() {

console.log(‘ok’);

}

}

console.log(fn); //还是fn函数的代码

**重要题目:**

//变量提升

// fn = ...1 | ...2 | undefined | ...3 | ...4

fn(); //4

function fn() {console.log(1);} //函数变量提升,声名和定义过了,以后不再执行

fn(); //4

function fn() {console.log(2);}

fn(); //4

var fn = 100; //函数变量提升只声名,没有赋值,此时进行一次赋值操作

fn(); //报错

function fn() {console.log(3);}

fn();

function fn() {console.log(4);}

fn();

es6的let没有变量提升,但是在当前作用域代码自上而下执行之前,浏览器会做一个语法检测:自上而下查找当前作用域下的所有变量,一旦发现有重复的,直接抛出异常(在重复的变量处报错,注意不是发生重复的时候,而是在这之前 => a(报错) a(第二个a) )

==>老语法:变量提升 新语法:语法检查(重复性检查)

console.log(a); //undefined

console.log(typeof a); //undefined

var a; //变量提升,走新语法规范,可以理解为浏览器处理的一个死区(bug):暂时性死区

console.log(a); //报错, a is not defined

let a; //语法检查,发现let,走新语法规范,es6解决了暂时性死区的问题

console.log(typeof a); //报错, a is not defined

let a; //语法检查,发现let,走新语法规范,es6解决了暂时性死区的问题

**重要的题目:(最难的题目)**

var n = 10;

function fn() {

var n = 20;

function f() {

n++;

console.log(n);

}

f();

return f;

}

var x = fn(); //21

x(); //22

x(); //23

console.log(n); //10

**画图思路:**

全局作用域:变量提升(普通函数创建堆内存,把函数的代码字符串放进去)=>代码自上而下执行(变量赋予值,函数赋予地址)=>函数执行(创建局部作用域:形参赋值=>变量提升=>代码自上而下执行)

全局作用域,局部作用域=>栈内存 函数定义=>堆内存(暴露地址)

**js中的内存分为堆内存和栈内存:**

堆内存:存储引用数据类型值(对象:键值对 函数:代码字符串)

栈内存:提供js代码执行的环境和存储基本类性质

**堆内存释放:**

让所有引用堆内存空间地址的变量赋值为null即可

(没有变量占用这个堆内存了,浏览器会在空闲的时候把它释放掉)

**栈堆存释放:**

一般情况下,当函数执行完成,所形成的局部作用域(栈内存)都会自动释放掉,在栈内存中存储的基本值也会释放掉,但是也有特殊情况不销毁的情况:

1. 函数执行完成,当前形成的栈内存中,某些内容被栈内存以外的变量占用了
2. 全局栈内存只有在页面关闭的时候,才会被释放掉
3. ......
4. 如果当前栈内存没被释放,那么之前在栈内存中存储的基本值也不会被释放,能保存下来

注意:  
一个函数执行,返回一个小函数

如果这个函数再次执行,返回的是代码字符串相同的小函数,但是是全新的,暴露的地址不同,**不是和之前同一个小函数**了!!!

**对比分析:**

var name = "the window";

var object = {

name: "obj",

getNameFunc: function() {

console.log(this); //obj对象

return function() {

console.log(this); //window

return this.name;

}

}

}

console.log(object.getNameFunc()()); //the window,函数自执行,this指向window

var name = "the window";

var object = {

name: "the obj",

getNameFunc: function() {

console.log(this); //obj对象

var that = this; //缓存this,好处是保证最开始的this指向不发生改变

return function() {

console.log(this); //window

console.log(that); //obj对象

return that.name;

}

}

}

console.log(object.getNameFunc()()); //the obj