基础进阶(内存,执行上下文,变量对象,作用域链和闭包,this,函数)

2020年1月13日

8:39

**一、内存**

* 1. 三种数据结构 : 栈 ,堆, 队列

栈: 先进后出的数据结构,JS没有严格区分栈内存和堆内存,因此可以粗暴地理解所有的数据都保存在堆内存中

堆: 是一种树状结构,顺序的不同不影响我们的使用

队列: 是一种先进先出的数据结构,主要是为了理解事件循环的机制

* 1. 变量对象和基础数据类型

JS执行上下文生成后,会创建一个叫做变量对象的特殊对象,JS的基础数据类型都保存在变量对象中(JS的基础数据类型有--undefined,Null,Boolean,Number,String;ES6添加一种新的数据类型Symbol)

* 1. 引用数据类型和堆内存

JS的引用数据类型的值的大小是不固定的.引用数据类型的值是保存在堆内存中的.操作对象时,实际上是在操作对象的引用而不是实际的对象

* 1. 内存空间管理

在JS中,最常用的是通过标记清除的算法来找到那些对象是不再使用的

手动释放内存 :

var a = 20; 在内存中分配空间

a=null;//使用完毕,释放内存

注意 : 局部作用域中,当函数执行完,局部变量就没有存在的作用了,垃圾收集器很容易进行判断并回收.但是全局变量什么时候回收内存难以判断,因此应避免使用全局变量

**二、执行上下文(execution context)**

每当控制器转到可执行代码的时候,就会进行一个执行上下文.执行上下文可以理解为当前代码的执行环境,它会形成一个作用域.JS的运行环境大概包括以下三种:

--全局环境 : JS代码运行起来首先进入该环境

--函数环境 : 函数被调用时,会进行当前函数中执行代码

--eval(不推荐使用,可忽略)

JS程序中,必定生成多个执行上下文,JS引擎会以栈的方式来处理它们,该栈为函数调用栈(call stack).栈底永远是全局上下文,栈顶是正在执行的上下文.

**函数调用栈的生成方式 :**

**代码 :**

var color = 'blue';

function changeColor() {

var anotherColor = 'red';

function swapColors() {

var tempColor = anotherColor;

anotherColor = color;

color = tempColor;

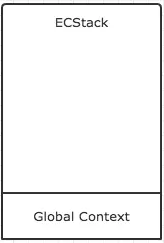
}

swapColors();

}

changeColor();

**图解 :**



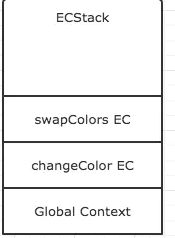
**第一步,全局上下文入栈**

入栈后开始执行可执行代码,直到遇到了changeColor(),激活函数changeColor,创建它自己的执行上下文.



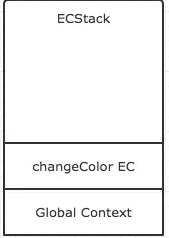
**第二步 : changeColor的执行上下文入栈**

入栈后开始执行其中的可执行代码,直到遇到了swapColors()之后又激活了一个执行上下文。



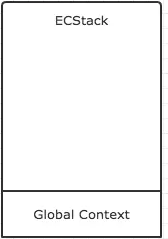
**第三步 : swapColors的执行上下文入栈**

该段代码没有生成其他的上下文,因此顺利执行完毕,swapColors的上下文从栈中弹出。



**第四步：swapColors的执行上下文出栈**

继续执行changeColor的可执行代码，也没有再遇到其他执行上下文，顺利执行完毕之后弹出。



**第五步 : changeColor的执行上下文出栈**

全局上下文在关闭窗口后出栈

注意: 函数中，return能直接终止可执行代码的执行，因此会直接将当前上下文弹出栈。

**总结 :**

* 1. 单线程
  2. 同步执行,只有栈顶的上下文处于执行中,其他上下文需要等待
  3. 全局上下文只有一个,在浏览器关闭时出栈
  4. 函数的执行上下文的个数没有限制
  5. 每次函数被调用,都有新的执行上下文被创建,即使是调用自身的函数,也是如此

**三、变量对象详解**

* 1. **执行上下文的生命周期**

创建阶段 : 执行上下文开始创建**变量对象**,确定**作用域链,**确定**this的指向**

代码执行阶段 : 创建结束后, 开始执行代码,会完成变量赋值,函数引用,执行其他代码

* 1. **变量对象**

变量对象的创建,依次经历以下的过程

* 1. 建立arguments对象,
  2. 检查当前上下文的函数声明.在变量对象中以函数名建立一个属性,**属性值指向该函数所在内存地址的引用**
  3. 检查当前上下文中的变量声明,每找到一个变量,就在变量对象以变量名建立一个属性,**属性值为undefined**

注意: 如果变量名和函数名同名,则在这个阶段,以函数名为准

console.log(foo); // function foo

function foo() { console.log('function foo') }

var foo = 20;

**函数声明** : 类似 function f(){};var f = function(){}为变量声明

未进入执行阶段之前,变量对象中的属性都不能访问.进入执行阶段,变量对象转变为活动对象,里面的属性都能被访问了,然后开始了执行阶段的操作

**活动对象 :** 处于函数调用栈栈顶的执行上下文中的变量对象

* 1. **全局上下文的变量对象**

全局上下文的变量对象和this指向都是window对象.

全局上下文的生命周期和程序的生命周期一致,只要程序运行不结束,全局上下文就一直存在

其他所有的上下文环境都能直接访问全局上下文的属性.

* 1. **let 和 const 声明**

let 和 const声明的变量,**仍被提前收集到变量对象中**,但和var 不同的是,let/const定义的变量，不会在这个时候给他赋值undefined。

console.log(a);

let a = 10;// Cannot access 'a' before initialization

**暂时性死区**（ temporal dead zone，简称TDZ ） : 如果区块中包含let ,const命令,那么区块对使用了这些命令的变量形成封闭作用域,凡是在声明之前使用该变量都会报错.

let a = 20;

if(true){console.log(a);}//20

let a = 20;

if(true){console.log(a);let a = 10}// Cannot access 'a' before initialization

**四、作用域链与闭包**

* 1. **词法环境(Lexicle Enviroment)**

词法环境,其实就是作用域

--在JS中我们可以将作用域定义做一套规则,用来管理JS引擎如何在当前作用域以及子作

用域根据标记符名称进行变量查找

--JS中有全局作用域和函数作用域

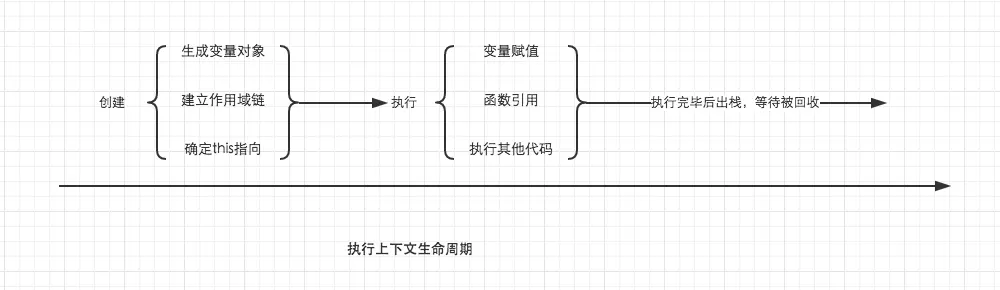
一个词法环境,是由**环境记录(Environment Records)**,和一个外部指向outer组成,环境记录对应的是变量对象,outer对应的就是**作用域链(scopeChain)**

* 1. **作用域链**

作用域和作用域链的区别 :

--作用域是一套规则

--作用域链是代码执行过程中,会动态变化的一条索引路径



作用域链,是由当前对象变量和上层环境的一系列变量对象组成,它保证了当前执行环境对符合访问权限的变量和函数的有序访问.

**例子 :**

var a = 20;

function test() {

var b = a + 10;

function innerTest() {

var c = 10;

return b + c;

}

return innerTest();

}

test();

innerTest的执行上下文表示如下 :

innerTestEC = {

VO: {...}, // 变量对象

scopeChain: [VO(innerTest), VO(test), VO(global)], // 作用域链

}

使用一个数组来模拟作用域链，数组的第一项scopeChain[0]为作用域链的最前端，而数组的最后一项，为作用域链的最末端，所有的最末端都为全局变量对象。

var a = 20;

function foo() {

if (!a) { //先在当前的变量对象中查询,结果为初始值undefined,执行块内内容

a = 100;

}

var a = 10; //对变量对象进行赋值

return a; //返回变量对象中的属性a

}

console.log(foo());

**总结:** 作用域链是由一系列变量对象组成,我们在这个单向的通道中,查询变量对象的标记符,就可以访问到上一层作用域中的变量

**闭包**

闭包是一种特殊的对象.它由两部分组成.执行上下文(代号A)以及在该执行上下文中创建的函数(代号B).当B执行时,如果访问了A中变量对象的值,那么闭包就会产生.在大多数理解中,都以函数B的名字代指这里生成的闭包。而在chrome中，则以执行上下文A的函数名代指闭包。

**例子 :**

function foo() {

var a = 20;

var b = 30;

function bar() {

return a + b;

}

return bar;

}

var bar = foo();

bar();

JS具有自动的垃圾回收机制,当一个值失去引用时,垃圾回收机制会根据特殊的算法找到它并回收.函数执行上下文在执行完毕后,生命周期结束,那么函数的执行上下文就会失去引用.其占用的内存很快会被垃圾回收器释放.但是,闭包阻止了这一过程.

**五、this**

this的指向,是函数调用时确定的,即执行上下文被创建时确定的.

**由于调用方式的不同,this可以指向不同的对象**

var a = 10;

var obj = {

a: 20

}

function fn () {

console.log(this.a);

}

fn(); // 10

fn.call(obj); // 20

**在函数执行过程中,this一旦被确定,就不可更改**

**结论 :**

在一个函数上下文中，this由调用者提供，由调用函数的方式来决定。

如果调用者函数，被某一个对象所拥有，那么该函数在调用时，内部的this指向该对象。

如果函数独立调用，那么该函数内部的this，则指向undefined。

在非严格模式中，当this指向undefined时，它会被自动指向全局对象。

**例子 :**

var a = 20;

var obj = {

a: 10,

c: this.a + 20,

fn: function () {

return this.a;

}

}

console.log(obj.c); //40

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

**解释 : 单独的{}是不会形成新的作用域的,因此,this指向的仍是window对象**

**例子 :**

function foo() {

console.log(this.a)

}

function active(fn) {

fn(); // 真实调用者，为独立调用,this指向全局,打印结果为20

}

var a = 20;

var obj = {

a: 10,

getA: foo

}

active(obj.getA);

**更改this的指向 :**

使用call或者apply方法,绑定方法内部的this

**使用场景 :**

**--将类数组对象转换为数组**

var arg = [].slice.call(arguments);

**--根据自己的需要灵活修改this的指向**

var foo = {

name: 'joker',

showName: function() {

console.log(this.name);

}

}

var bar = {

name: 'rose'

}

foo.showName.call(bar);

**--实现继承**

var Student = function(name, age, high) {

// use call

Person.call(this, name, age);

this.high = high;

}

**--向其他执行上下文的传递中,保证this的指向不变**

**构造函数和原型方法上的this**

通过new调用构造函数,经历了以下4个阶段

--创建一个新的对象

--将构造函数的this指向这个新对象

--指向构造函数的代码,为这个对象添加属性方法等

--返回新的对象

因此,在调用构造函数时,this指向的是新创建的对象.

原型方法上的this,根据调用方式p1.getName(),this也指向了p1;如果调用方式为

let x = p1.getName;

x(); //为独立调用,this指向undefined,非严格模式下指向window

**七、函数与函数式编程**

**模块**

利用闭包,我们可以访问到执行上下文内部的变量和方法,可以实现模块的共有变量和共有方法.

(function() {

// 私有变量

var age = 20;

var name = 'Tom';

// 私有方法

function getName() {

return `your name is ` + name;

}

// 共有方法

function getAge() {

return age;

}

// 将引用保存在外部执行环境的变量中，形成闭包，防止该执行环境被垃圾回收

window.getAge = getAge;

})();

**利用闭包实现简单的状态管理器**

<<状态管理器.txt>>

**函数式编程**

对较为复杂并且使用频率较高的运算进行函数封装,之后每次需要进行类似操作,只需要调用函数即可

**函数是第一等公民**

函数可以作为其他函数的参数,也可以赋值给其他变量,或者作为函数的返回值

var timer = setTimeout(function() {

delay();

}, 5000);//复杂,不正确

var timer = setTimeout(delay, 5000);//正确

**只用表达式,不用语句**

function setBackgroundColor(ele, color) { //输入使用ele,能应用于多个dom对象

ele.style.backgroundColor = color;

return color; //函数有输入有输出

}

// 多处使用

var ele = document.querySelector('.test');

setBackgroundColor(ele, 'red');

setBackgroundColor(ele, '#ccc');

**纯函数**

相同的输入总会得到相同的输出，并且不会产生副作用的函数