

谦太祥和（北京）科技有限公司

**老马-经典前端教程-JavaScript高级]**

[JavaScript高级]

# 目录

[目录 2](#_Toc8952)

[1． 联系老马 3](#_Toc8659)

[2. 值类型与引用类型及参数传递 3](#_Toc9657)

[2.1 值类型与引用类型的内存模型 3](#_Toc20821)

[2.2 值类型的复制赋值和引用类型的赋值的区别 4](#_Toc16011)

[2.3 函数参数的引用传递和值传递 5](#_Toc9468)

[2.4 函数参数详解 6](#_Toc11488)

# 联系老马

老马联系方式

QQ：515154084

邮箱：malun666@126.com

微博：<http://weibo.com/flydragon2010>

百度传课：

<https://chuanke.baidu.com/s5508922.html>

微信：



# 2. 值类型与引用类型及参数传递

## 2.1 值类型与引用类型的内存模型

值类型（基本类型、简单类型）：数值、布尔值、null、undefined。

引用类型（复杂类型）：对象、数组、函数。

变量对象存放变量名字和值

堆内存：存放JavaScript的数据和方法

例如代码：

var a1 = 0; // 变量对象

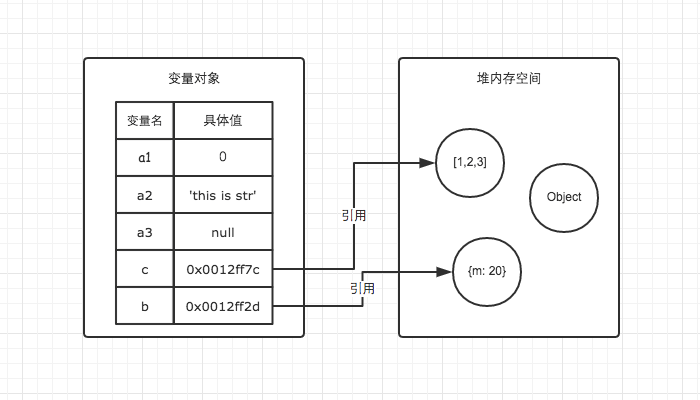
var a2 = 'this is string';

var a3 = null;

var b = { m: 20 }; // 变量b存在于变量对象中，{m: 20} 作为对象存在于堆内存中

var c = [1, 2, 3]; // 变量c存在于变量对象中，[1, 2, 3] 作为对象存在于堆内存中

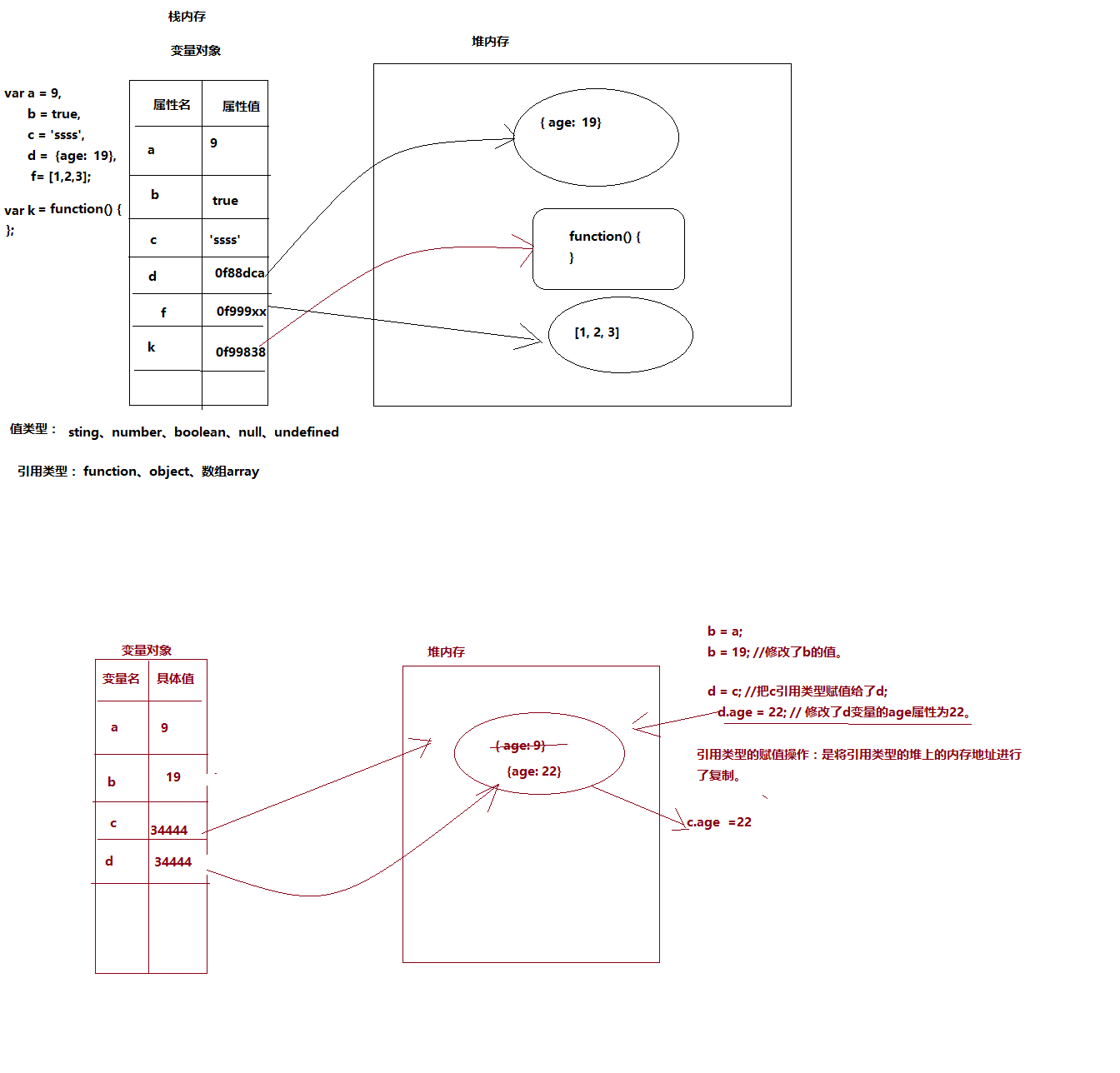
在内存中的存储情况为：



## 2.2 值类型的复制赋值和引用类型的赋值的区别

值类型的赋值，只是简单的值的副本拷贝。

引用类型的赋值操作，是复制的引用的地址，而堆栈上的数据并没有赋值。是浅拷贝。



var a = 9, b;

var c = { age: 9}, d;

// 把a值类型赋值给了b；

b = a;

b = 19; //修改了b的值。

console.log(a); // a=>9

console.log(b); // b=>19

// 如果是引用类型的赋值后操作

d = c; //把c引用类型赋值给了d;

d.age = 22; // 修改了d变量的age属性为22。

// 那么c的age属性是多少呢？

console.log(c.age); // c.age => 22

console.log(d.age); // d.age => 22

c.age = 99;

console.log(c.age); // c.age => 99

console.log(d.age); // d.age => 99

## 2.3 函数参数的引用传递和值传递

函数参数的引用传递和值传递

如果实参是值类型，会复制一个值类型的副本给函数，不会影响原来的传递参数的值类型变量。

如果实参是引用类型，传递只是引用类型的一个地址值，在函数内部操作参数对应的引用对象会影响到传递的参数。

var a = 9, // 简单类型，值类型

b = { name: 'laoma', age: 18 };// 引用类型

// 把 a\b分别传给 c1和c2.

// 函数的参数：如果是简单类型会做一个值类型的数值副本传到函数内部。

// 如果是一个参数是引用类型，会讲引用类型的地址值复制给传入函数的参数。

function demo(c1, c2) {

c1 = 29; // 对c1做了更改。

c2.name = '6666'; // 更改了c2引用类型的顺序name

}

// 调用函数执行

demo(a, b);

console.log(a); // a => 9;

console.log(b); // b.name => '6666'

## 2.4 函数参数详解

* 形参：函数定义的参数
* 实参：函数调用时实际传递的参数。
* 参数匹配

是从左向右进行匹配。如果实参个数少于形参，后面的参数对应赋值undefined。

实参的个数如果多于形参的个数，可以通过arguments访问。

* 函数对象的length属性就是函数形参的个数。

【案例】模拟封装Math的max方法。

// 函数在未定义之前就可以直接使用，因为声明的函数会提示到代码的顶部。

var m = myMax(10, 9, 2, 33, 22, 18);

console.log(m); // m => 33

// 要你实现max方法，可以接受任意多个参数。返回这些参数中最大的那个值？

function myMax() {

// 如果调用函数的地方没有传递参数，那么直接返回NaN

if(arguments.length <=0 ) {

return NaN; // return语句执行，当前函数立即结束。

}

var max = arguments[0];

// 获取实参：使用arguments,它是一个类数组。length属性是实参的个数。

for(var i = 1; i < arguments.length; i++) {

if(arguments[i] > max) {

max = arguments[i];

}

}

return max;

}

如果参数多于4个，那么开发人员很难记忆，最好将参数封装成对象来接受，对象的属性是无序的，可以方便开发人员使用。

// 函数参数的封装

// 一个函数：封装一个矩形。矩形：x y坐标， width、 height、背景色、文字信息、文字的坐标、文字字体颜色

// 全部用形参来处理矩形的参数

function rect(x, y, width, height, bgColor, text, text\_color, text\_x, text\_y) {

}

// 如果函数的形参非常多问题：

// 1、开发人员很难记忆形参的具体情况

// 2、传递参数的时候，如果顺序不小心写错了，那么就会导致函数内部出现错误。

// 3、编写代码不方便。

// 解决办法：把这些形参封装成一个对象进行传递。

function rect2(rectObj) {

// 拿到矩形的x、y坐标

console.log(rectObj.x + ' ' + rectObj.y);

}

// rectObj

var rectObj = {

x: 19,

y: 20,

width: 200,

height: 300,

bgColor: '#ccc',

text: 'laoma'

};

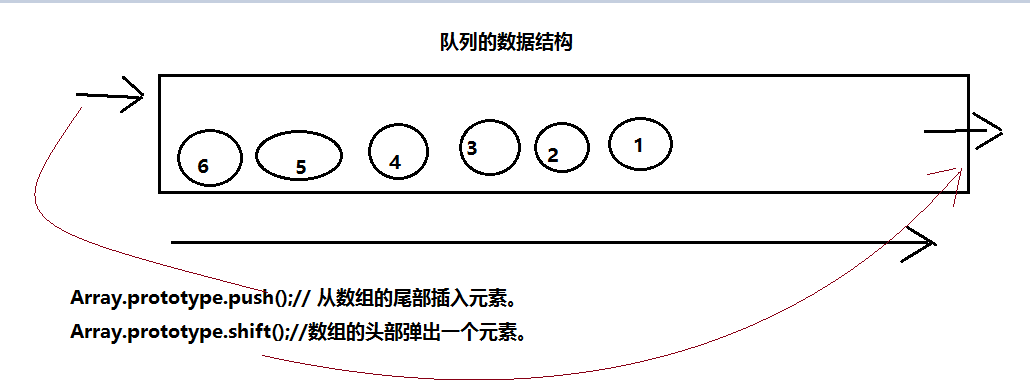
rect2(rectObj);

# 函数高级内幕

## 3.1 JavaScript事件循环机制

### 3.1.1 复习队列数据结构

队列就是现进先出的数据结构。



### 3.1.2 JavaScript是单线程的。

浏览器：

JavaScript执行线程：负责执行js代码

UI线程：负责UI展示，负责展示给用户看到的页面的。

JavaScript事件循环线程。

**结论：JavaScript中的代码都是排队执行，不会同步执行多个任务。**

### 3.1.3 JavaScript的任务

JavaScript中的任务分为同步任务和异步任务。

同步任务：一般的赋值操作、循环、分支语句等都是同步的任务。

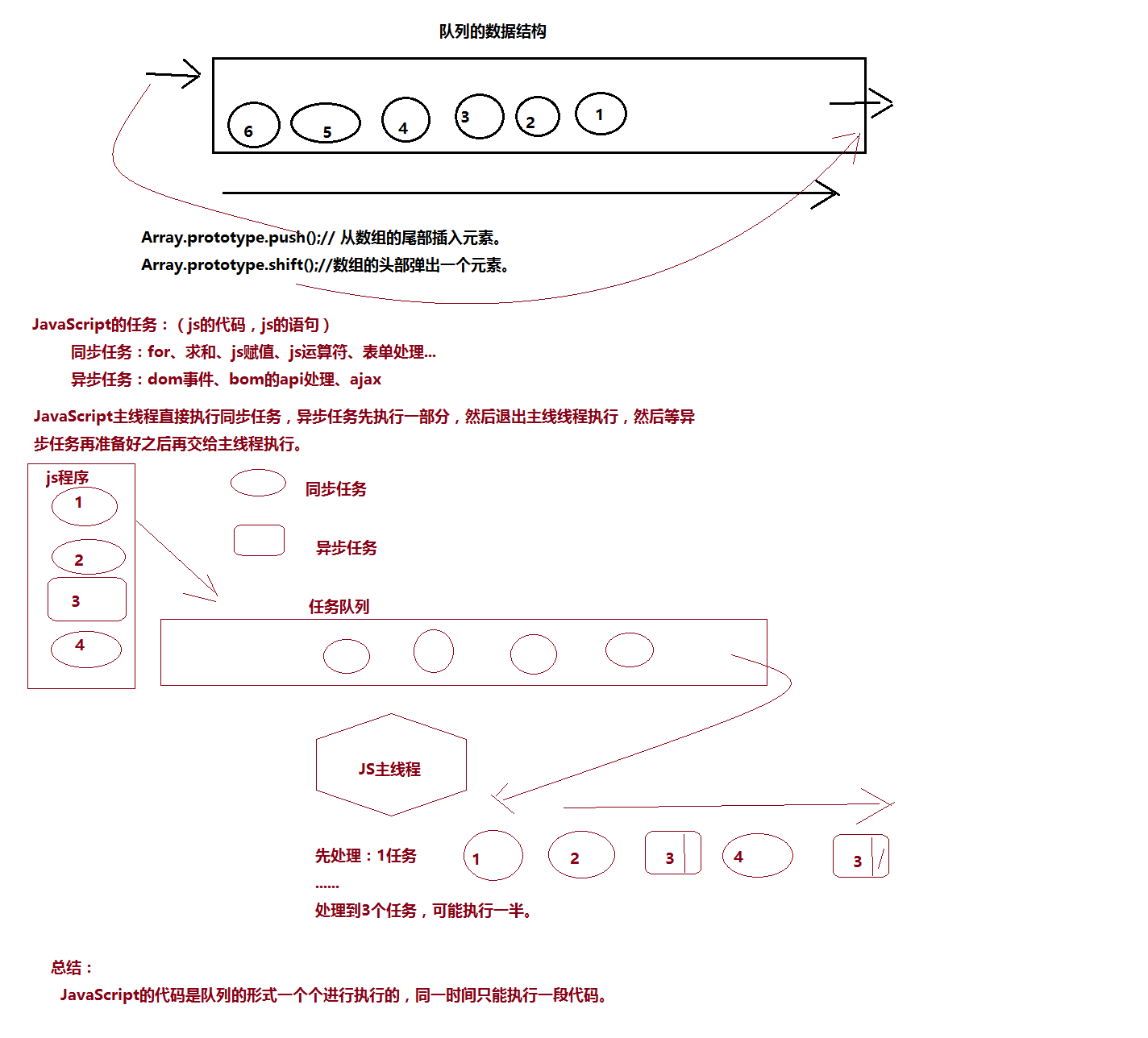
异步任务：DOM事件、Ajax、BOM的一些API等

### 3.1.4 事件循环机制（Event Loop）

JavaScript的执行引擎的主线程 从任务队列中获取任务执行

如果任务是异步任务，那么运行到异步任务时，异步任务就退出主线程，主线程进行下一个任务的获取处理。

如果异步任务完成，就插入到任务队列的末尾，等待主线程处理



## 3.2 执行上下文

### 3.2.1 复习栈的数据结构

栈的数据结构:先进后出。

类比于弹夹。

可以使用Array数组的push和pop方法配合实现数组的栈数据结构。



### 3.2.2 函数执行上下文

EC：函数执行环境（或执行上下文），Execution Context

ECS：执行环境栈，Execution Context Stack

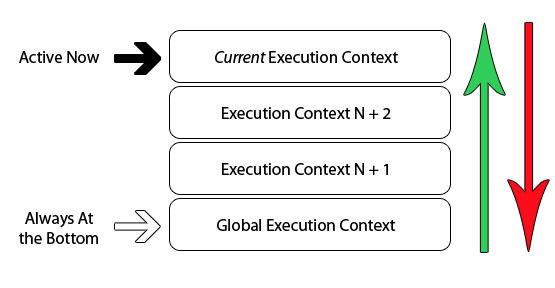
JavaScript执行在单线程上，所有的代码都是排队执行

一开始浏览器执行全局的代码时，首先创建全局的执行上下文，压入执行栈的顶部。

每当进入一个函数的执行就会创建函数的执行上下文，并且把它压入执行栈的顶部。当前函数执行完成后，当前函数的执行上下文出栈，并等待垃圾回收。

浏览器的Js执行引擎总是访问栈顶的执行上下文。

全局上下文只有唯一的一个，它在浏览器关闭时出栈



例如代码：

<script>

function f1() {

console.log('f1.....')

f2();

}

function f2() {

console.log('f2.....')

f3();

}

function f3() {

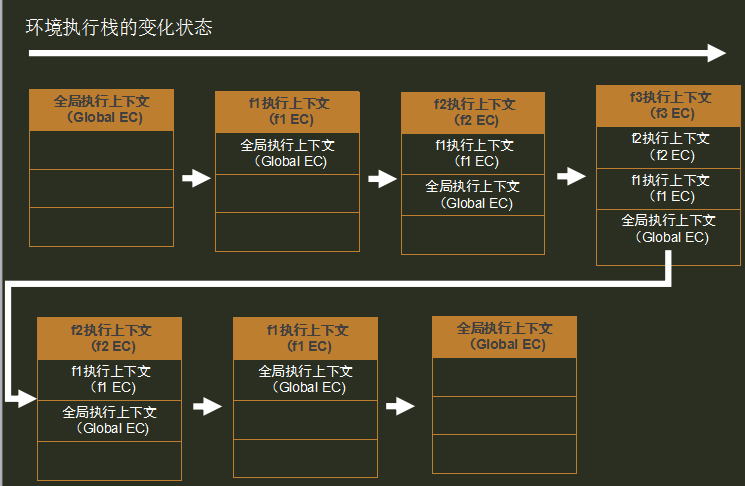
console.log('f3 .....');

}

f1();

</script>

环境执行栈的内存状态的变化如下图：



### 3.2.3执行上下文生命周期

总的的生命周期：创建→执行→出栈等待销毁

* **创建阶段：**

1. 创建作用域链（Scope Chain）
2. 创建变量对象（或者AO）

首先初始化函数的参数arguments，初始化函数声明， 初始化变量（undefined）。函数的优先级要高于变量，如果变量和函数重名，变量会被忽略。

* + - 创建arguments对象，检查上下文，初始化参数名称和值并创建引用的复制。
    - 扫描上下文的函数声明（而非函数表达式）：

为发现的每一个函数，在变量对象上创建一个属性并指向函数在内存中的引用。

如果函数的名字已经存在，引用指针将被重写。

* + - 扫描上下文的变量声明：

为发现的每个变量声明，在变量对象上创建一个属性——就是变量的名字，并且将变量的值初始化为undefined

如果变量的名字已经在变量对象里存在，将不会进行任何操作并继续扫描。

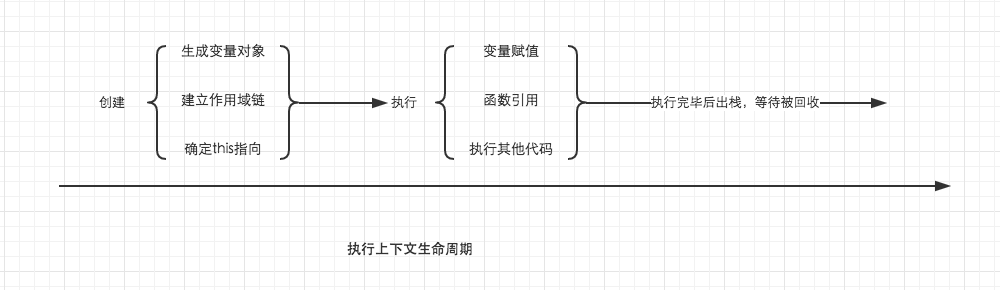
3）求出上下文内部“this”的值。

* **执行阶段：**

执行变量赋值、代码执行

* **回收阶段：**

执行上下文出栈等待虚拟机回收执行上下文



例如代码：

<script>

var a1 = 19,

a2 = 20,

a3 = 'sss',

b1 = { name: 'laoma'};

a1 = f1(a1, a2);

function f1(a, b) {

var t = 0,

m = 10;

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

console.log(i);

}

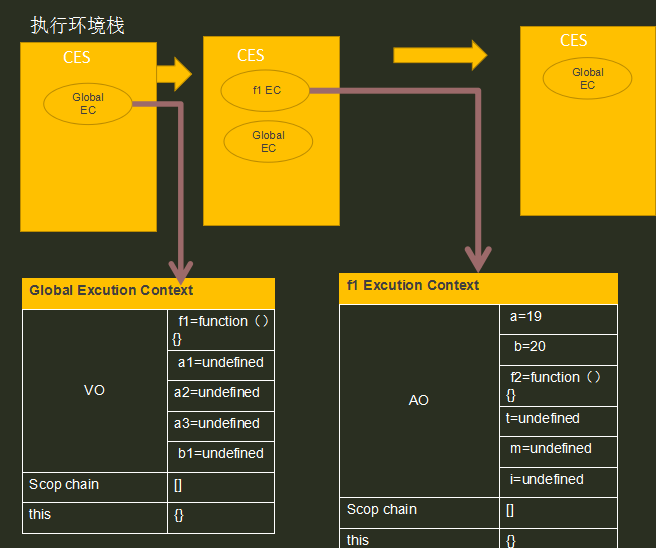
function f2() {

console.log(f2);

}

return a + b;

}</script>



## 3.3 JavaScript解释和执行阶段

JavaScript的执行分为： 解释 和 执行两个阶段。

* 解释阶段：
  + 词法分析
  + 语法分析
  + **作用域规则确定**
* 执行阶段：
  + 创建执行上下文
  + 执行函数代码
  + 垃圾回收

所以在js的解释阶段就已经确定了变量的作用域的规则！！！

## 3.4 变量的作用域

作用域：就是变量声明的区域,就是变量和函数的可访问范围。在全局声明的变量为全局可见可访问的就是全局变量，如果在函数内部声明的变量只能在函数内部访问

函数的参数只能在函数内部访问，是局部变量。

JavaScript没有块级作用域，只有函数作用域和全局作用域。for循环内部定义的变量是函数级别的作用域。

变量没有在函数内声明或者声明的时候没有带var就是全局变量，拥有全局作用域。特殊：var a = b = c = 0; b与c是全局变量。

全局作用域的变量可以在js中任何地方调用，函数作用域的变量只能在自己函数内部调用，包括自己内部定义的其他函数都可以直接调用。

变量的作用域是以它声明时的为准，因为变量的作用域在JS代码的解释阶段就已经完成规则的制定。

### 3.4.1 作用域分析

<script>

var t = 9; // 全局作用域，全部都可以访问

function f1() { // f1 函数全局作用域

var t2 = 10; // t2 是f1函数内部可访问

console.log(t);

function f2() { // f2函数式 f1函数的作用域

var t3 = 200; // t3 只能在f2函数内部访问

console.log(t2);

return t2 \* t2; // f2函数可以访问f1函数的作用域的变量及 f2自己内部的变量。

}

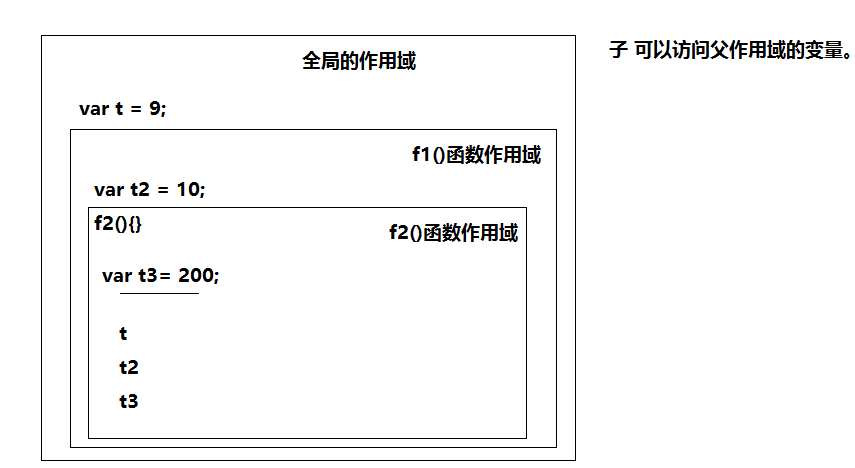
return f2();

}

var m = f1();

console.log(m);

</script>



### 3.4.2 没有块级作用域

JavaScript没有块级作用域。

for循环、while循环中定义的变量的作用域是函数级别的作用域。

C、C#、C++、Java： 在for循环内部定义的变量，只能在for循环内部访问。但是在js中，因为没有块级作用域，所以在for循环内部定义的变量在整个所在的函数内部是可以访问的。

// 没有块级作用域

function f1() {

var t1 = 9; // 只能在f1函数内部访问到。

for(var i = 0; i < 10; i++) { // i变量在for循环中定义的。

console.log(i); // java/c#/c/c++

}

console.log(i);// 因为js没有块级作用域，所以可以直接访问i变量。

}

f1();

// console.log(t1); //全局是访问不到f1函数内部的变量的。

### 3.4.3 变量提升

如果一个声明的变量在函数体内，那么它的作用域就是函数内部。如果是在全局环境下声明的，那么它的作用域就是全局的。

通过var声明的变量是无法用delete删除的。

函数内部的声明的变量会被提升到函数的头部。函数在解析执行的时候，先进行变量声明处理，然后再运行函数内部的代码。

变量和赋值语句一起书写，在js引擎解析时，会将其拆成声明和赋值2部分，声明置顶，赋值保留在原来位置

变量重复声明不会出错，后面的会覆盖前面的。

var a = 10; // 全局变量，全局都可以访问。

// 先执行f1函数，然后再定义f1函数的内容。

f1();

function f1() {

// 函数的变量提升： 因为在函数执行之前，先创建了函数的EC，在创建

// EC的时候已经把函数里面声明的变量都已经初始化成了undefined

// 所以：hositing存在：很多人有这样的习惯。把函数内部所有的变量声明都放在函数的

// 头部

console.log(a); // a => undefined

var a = 19; // 给局部变量a赋值19

console.log(a); // a=> 19

// 特殊情况：变量声明和函数声明 同时拥有一个名字的时候，函数优先级高。

console.log(b); // b=> function

var b = 9; // 因为js是动态语言，把b重新赋值9，把之前的function覆盖。

function b() {

}

console.log(b); // b=>9

}

console.log(a); // a => 10

f();

function f() {

for(var k = 0; k <10; k++) {

console.log(k);

}

console.log(k);

}

案例：

案例1：

var a = 18;

function d() {

console.log(a); // undefined

var a = { age: 19};

console.log(a); // {age: 19}

}

d(); // 输出？

console.log(a); //18

案例2：

// 在浏览器中全局对象就是window

// 在全局作用域中声明的变量和函数声明都会作为window的属性存在。

// var a = 9;

// function b() {

// console.log(a);

// }

// console.log('a' in window);

// console.log(window.a);

// console.log(window['a']);

// console.log(window.b());

// ("a" in window) => true

if (!!("a" in window)) {

var a = 1; // 定义了一个a变量，a =undefined，js没有块级作用域

}

console.log(a);

// 案例3

// console.log(a); // a => function a() {}

// var a = 20; // 对a进行重新赋值

// console.log(a); // a => 20

// function a() { // 创建global EC的VO的时候，函数的声明优先级要高于变量的声明。

// }

// // 案例4：

// f();

// console.log('b =' + b); // 9

// console.log('c =' + c); // 9

// console.log(a); // 出一个异常错误，使用未定义的变量a

// function f() {

// var a = b = c = 9; // a是局部变量，b\c是全局变量。

// // var a = 9, b = 9, c = 9; //定义三个局部变量

// console.log(a); // 9

// console.log(b); // 9

// console.log(c); // 9

// }

// 案例5：

f();

function f() {

for(var k = 0; k < 10; k++) {

console.log(k);// 0-9

}

console.log(k);// 10，js没有块级作用域

}

## 3.5 作用域链

作用域链是一个数组

作用域链是控制变量作用域的有序访问的js内部实现。

作用域链存储在函数的执行上下文中。

当前函数的作用域对象都是在最前端，而且全局的在最末端

变量（标识符）的搜索都是从作用域链的最前端向后搜索，直到全局作用域 ，标识符的解析是沿着作用域链一级一级搜索的过程，从第一个对象开始，逐级向后回溯，直到找到同名标识符为止，找到后不再继续遍历，找不到就报错。

例如代码：

var t = 0, m = 10;  
var res = fParent(t, m);

console.log(res);

function fParent(a, b) {

var vParent = 'parent';

function fChild(c) {

var vChild = 'child';

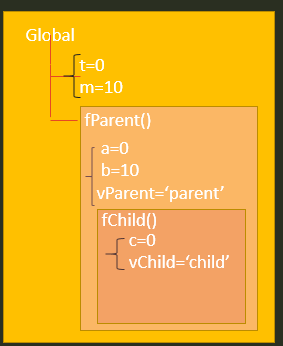
return c + vChild;

}

return a + b + vParent +fChild(a);

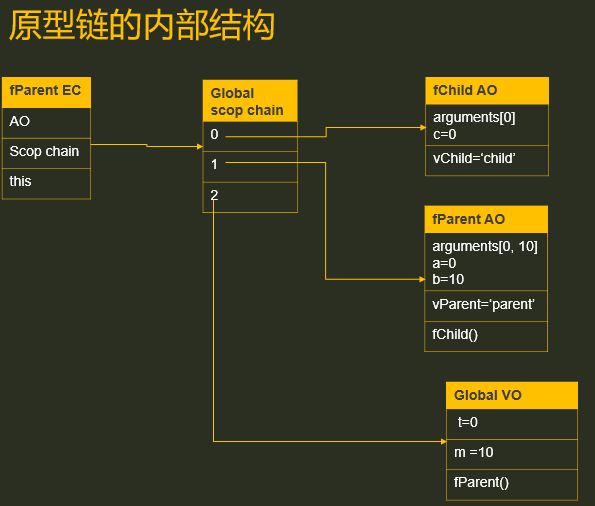
}

变量的作用域为：



作用域链的结构为：





## 3.6 函数的四种调用模式

### 3.6.1 方法调用模式

如果一个函数作为对象的一个方法属性调用，那么它的调用模式就是方法调用模式。

var a = {};

a.toString(); // 方法调用模式

方法调用模式中的this指向调用对象。

### 3.6.2函数调用模式

如果一个函数被直接调用。那么调用者其实就是全局对象：window

函数调用模式this指向全局对象。

// 定义构造函数

// 定义一个Dog类的构造函数。

function Dog(dogName) {

// 如果函数当做构造函数来用。

// 第一步：创建一个空对象（新对象），函数上下文===this

// 第二步：把空对象赋值给函数的上下文，this=新对象。

this.name = dogName;

this.age = 0;

this.run = function() {

console.log(this.name + ' is running..');

};

// 如果函数当做构造函数调用，并没有返回任何数据的时候。默认就会返回 新对象（this）

}

// 使用构造函数创建一个Dog类型的实例

var d = new Dog('lddd');// 构造函数的返回值就是默认的新对象this

d.run(); // 调用d对象实例的 run方法。 这就是对象的方法调用模式。

// 在方法调用模式中，方法内部的this指向当前调用者的对象=> d

// this === d // true

### 3.6.3 构造器调用模式

构造器调用模式就是构造函数调用。

构造器模式调用必须有关键字new的存在。

构造器模式调用的this指向创建出来的新对象。

var t = new Dog(); // 构造器调用模式

构造函数可以返回一个值，但是如果是简单类型会被忽略。如果是引用类型会替换掉新创建的对象返回。

### 3.6.4 apply/call调用模式(借用方法模式)

apply和call可以改变函数调用的内部this的指向。

apply和call的功能一样，只不过参数不一样。第一个参数都是改变函数内部的this的指向。

第一个参数如果是null、undefined会被全局对象替代，如果是简单类型会被包装类型替代。

call第二个参数开始后面的都是传给函数的参数，可以有多个，用逗号隔开。

apply第二个参数是一个传给函数的参数数组

调用：

function m(a) {

};

m.call(window,2);

m.apply(window, [2]);

### 3.6.5 案例

1、定义按钮类，要求按钮类的构造函数可以接受参数初始化按钮的宽度、高度、坐标xy

2、借用Math的min方法实现求数组[2,9,33]中的最小值

3、把类数组转换成真正的数组。

var t = {}; t[0] = 1; t[1] = true; t.length = 2;

4、判断代码输出的内容：

function Dog() {

console.log(this);

}  
Dog();

var d = new Dog();

Dog.call(null);

参考代码：

// 1.定义按钮类，要求按钮类的构造函数可以接受参数初始化按钮的宽度、高度、坐标xy

/\*

function Btn(width, height, x, y) {

// 构造函数内部初始化值。

this.width = width; //给this对象上的width属性赋值 width参数的参数值。

this.height = height;

this.px = x;

this.py = y;

}

var b = new Btn(100, 100, 30, 30);

\*/

// 2、借用Math的min方法实现求数组[2,9,33]中的最小值

// var m = Math.min(10, 29, 3);

// console.log(m);

// var m = Math.min.apply(null, [2, 9, 33]);

// console.log(m);

// 3.把类数组转换成真正的数组。

// var t = {}; // 类数组，跟数组很相似，但不是数组。

// t[0] = 1;

// t[1] = true;

// t[2] = 'laoma';

// t.length = 3;

// t = [1, true, 'laoma'];

// var m = [1, 2, 3];

// m.slice();// 如果什么都不传，默认是从0索引开始截取到数组最后。

// 第一个参数是：截取开始的位置，startIndex

// 第二个参数是： 截取结束的位置+1。endeIndex

// 如果我借用 数组的slice方法，然后把this指向到t对象，那么slice方法就会返回t对象的

// 对应的数组。

// var k = Array.prototype.slice.call(t,0);

// // 通过一个slice方法可以把类数组转成真正的数组。

// console.log(k);

## 3.7 函数没有重载

JavaScript中函数不能重名，如果重名后面的会把前面的覆盖。

原因：所有的函数声明都会创建在Vo或者AO上的一个属性。而后面声明的函数会把前面vo或者ao中的同名属性覆盖。

重载的概念:在程序中可以定义相同名字，不同参数的形式的不同函数。函数在调用的时候，自动识别不同参数对应的函数，实现了相同函数名不同的函数调用。

JavaScript中通过arguments实现函数重载

数组的slice方法、splice方法等都可传递一个参数或者多个参数，不同参数方法内部实现不一样，就是方法重载。

案例参考：

创建一个矩形的类型。构造函数接受一个参数，返回一个正方形，接受两个参数返回一个矩形。

// arguments模拟函数重载

// 创建一个矩形的类型。构造函数接受一个参数，返回一个正方形，接受两个参数返回一个矩形。

function React() {

//如果一个参数，返回一个正方形

if(arguments.length == 1) {

this.width = arguments[0];

this.height = arguments[0];

}

//如果两个参数，返回一个矩形

if(arguments.length > 1) {

this.width = arguments[0];

this.height = arguments[1];

}

// 由于跟原型上的toString方法重名，那么会覆盖Object原型上的toString方法

this.toString = function() {

// console.log('width: ' + this.width + ' height: ' + this.height)

return 'width: ' + this.width + ' height: ' + this.height;

}

}

var r1 = new React(10);

console.log(r1.toString());

var r2 = new React(10, 9);

console.log(r2.toString());

## 3.8 函数的递归调用

函数的递归调用就是指：函数调用自身。

arguments.callee就是指向函数自身的变量，所以可以直接用它来代替函数名，在匿名函数中非常有用。但是在严格模式下会报错。

函数表达式方式定义的时候，还可以用命名函数表达式

例如：

var f = function s(){};// 在s函数内部，可以直接使用s变量。

案例：

1. 求1到100的和，请使用递归。

// 常规的写法

// var sum = 0,

// n = 100;

// for(var i = 1; i <= n; i++) {

// sum += i;

// }

// console.log(sum);

// 用递归实现。函数调用函数自己。

function sumNum(num) {

// 递归一定要有个结束自己调用自己的出口。

if(num <= 1) {

return num; // 函数只要执行到return当前函数立即结束执行。

}

// 实现函数调用函数自己。

return sumNum(num -1) + num;

}

console.log( sumNum(100) );

1. 求num的阶乘，用递归实现。

求num的阶乘，用递归实现。

function factorial(num) {

// 定义一个出口。

if(num == 1) {

return 1;

}

return num \* factorial(num -1);

}

console.log(factorial(4));

1. 求f(n)的斐波那契数列的值。 f(n) = f(n-1) + f(n-2); n>=3

// f(0) = 0, f(1) = 1 f(2) = 1, f(3) = 2....f(4)= 3 f(5) = 5 f(6）= 8)

function fibonacci(n) {

if(n == 1) {

return 1;

}

if(n == 0) {

return 0;

}

return fibonacci(n-1) + fibonacci(n-2);

}

console.log(fibonacci(6));

## 3.9 函数式编程

### 3.9.1函数是JavaScript的一等公民

所谓"第一等公民"（first class），指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

### 3.9.2 数组的sort方法回顾。

// 复习： Array.prototype.sort 可以对数组中的元素进行排序。

// 排序的算法是根据字符串比较大小的算法进行排序。

// 不适合于数字的排序。

var t = [ 48, 2, 33, 4, 55, 9, 8];

console.log(t);

console.log(t.sort());

// 2, 33, 4, 48, 55, 8 ,9

// 按照数值的大小进行排序。

t.sort(function(a, b) { // 函数式编程。一个函数作为另外一个函数的输入。

// 如果 a > b return 大于0的值

// a===b return 0

// a <b return 小于0的值

return a - b;

});

console.log(t);

### 3.9.3数组的map方法（IE9+支持）

返回新数组

方法接受一个回调函数，回调函数三个参数：当前项、索引、操作的数组

不影响原来的数组

//map 方法

var t = [ 1, 3, 9, 10, 20 ];

// for(var i in t) {

// console.log(t[i]);

// }

// 对数组中的每个元素都进行\*2，打印结果数组。

// map方法，返回一个新数组，每个项处理完成后的结果组成的新数组。

// 对原数组没有影响。

var m = t.map(function(item, index, array){

// item: 就是当前的选项。

// index: 当前选项的索引

// array：当前数组

// return： 返回每个项处理的结果。

return item \* 2;

});

console.log(t);

console.log(m);

### 3.9.4数组的forEach方法(IE9+支持)

返回：undefined

forEach 方法按升序为数组中含有效值的每一项执行一次callback 函数，那些已删除（使用delete方法等情况）或者未初始化的项将被跳过（但不包括那些值为 undefined 的项）（例如在稀疏数组上）。

回调函数的参数：

* 数组当前项的值
* 数组当前项的索引
* 数组对象本身

// 22-数组的forEach方法.html

var m = ['22', true, 1, 99, 98];

// 打印m中的每个元素。

m.forEach(function(item, index) {

// item是遍历的每个项。

// index 项对应数组的索引。

console.log('index：' + index + ' ' + item);

});

## 3.10 函数也是对象及它的属性和方法

函数本身也是一种对象。构造函数是Function

f instanceof Object;// true

语法：

object instanceof constructor

参数

object：要检测的对象.

constructor：某个构造函数

instanceof 运算符用来检测 constructor.prototype 是否存在于参数 object 的原型链上。

函数有内部属性：arguments，可以在函数内部使用。

函数的自身的属性：length，函数定义的形参个数。

另外我可以自定义函数的其他属性和方法。【一般用于全局变量、静态变量、公共存储等】

// 函数的自有属性

function fun1(a, b) {

console.log(a + b);

}

console.log(fun1.length); // 函数都有默认的length=形参的个数

// fun1是函数也是一个对象。就可以当对象使用。

fun1.age = 19;

console.log( fun1.age );

# 4. JavaScript内存管理及垃圾回收

## 4.1 垃圾回收算法

Javascript具有自动垃圾回收机制(GC:Garbage Collecation)。我们不用关心变量的内存申请和回收。

在C与C++等语言中，开发人员可以直接控制内存的申请和回收。但是在Java、C#、JavaScript语言中，变量的内存空间的申请和释放都由程序自己处理，开发人员不需要关心。

垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存。

js中最常用的垃圾回收方式就是标记清除。当变量进入环境时，例如，在函数中声明一个变量，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。被标记离开环境的变量在下一次垃圾回收启动的时候会被释放掉占用的内存空间。

Javascript引擎基础GC方案是（simple GC）：mark and sweep（标记清除），即

（1）从根对象开始遍历所有可访问的对象。

（2）回收已不可访问的对象。

IE6的垃圾回收是根据内存分配量运行的，当环境中存在256个变量、4096个对象、64k的字符串任意一种情况的时候就会触发垃圾回收器工作。bug！！！

IE7中做了调整，触发条件不再是固定的，而是动态修改的，初始值和IE6相同，如果垃圾回收器回收的内存分配量低于程序占用内存的15%，说明大部分内存不可被回收，设的垃圾回收触发条件过于敏感，这时候把临界条件翻倍，如果回收的内存高于85%，说明大部分内存早就该清理了，这时候把触发条件置回

var m = 0, n = 19; // 把 m , n add() 标记为进入环境。

add(m, n); // 把 a, b, c标记为进入环境。

console.log(n); // a,b,c标记为离开环境，等待垃圾回收。

function add(a, b) {

a++;

var c = a + b;

return c;

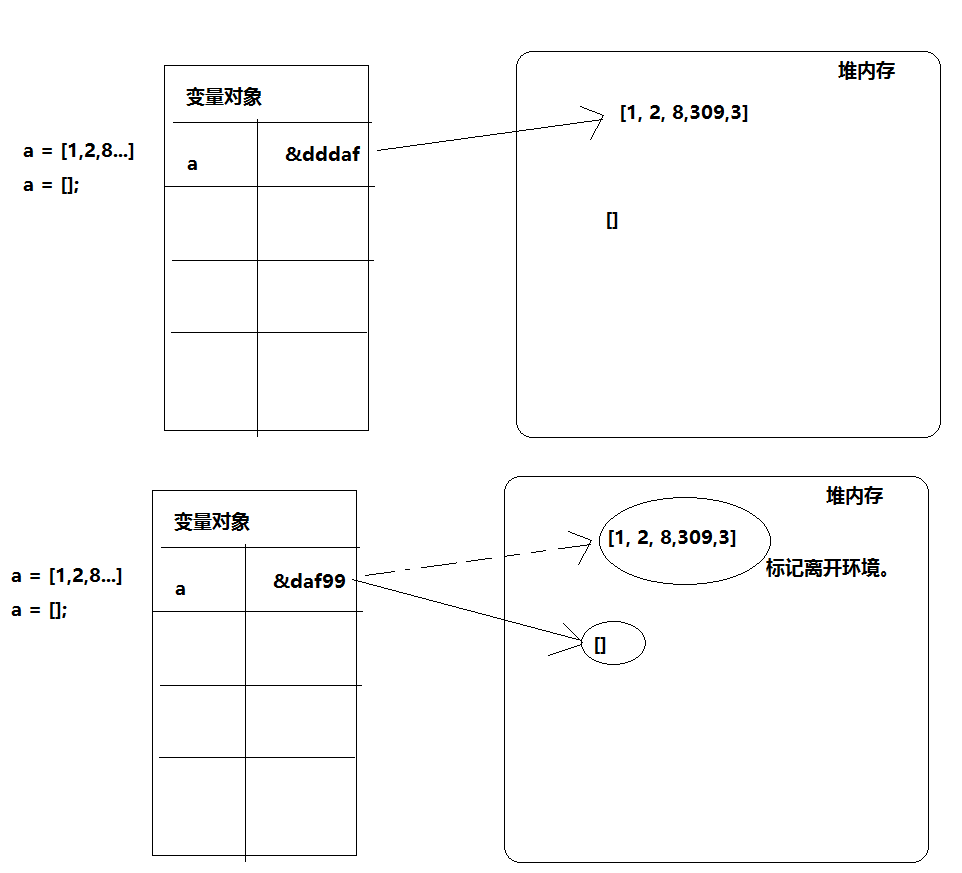
}

## 4.2 垃圾回收应用

### 4.2.1 数组的清零操作

arr = []; //虽然能清空arr数组，但是新建了一个[]空数组对象。

最好的办法是：arr.length = 0;//即可清除数组内容，还不额外开辟新内存。



// 数组的清零。

var a = [1, 9, 20, 333, 4, 43];

a.forEach(function(item) {

console.log(item);

});

// 数组清零，让gc尽快的回收a数组的内存的空间。

// a = null; // 首先，把a转成null的类型。

// a = []; // 确实让a变量成一个空数组。但是在堆上重新申请了一个空数组对象。

a.length = 0; // 可以直接让数字清空，而且数组类型不变。

### 4.2.2对象尽量复用

尤其是在循环等地方出现创建新对象，能复用就复用。不用的对象，尽可能设置为null，尽快被垃圾回收掉。

// 对象复用

var t = {};// 每次循环都会创建一个新对象。

for(var i = 0; i < 10; i++) {

// var t = {};// 每次循环都会创建一个新对象。

t.age = 19;

t.name = '123';

t.index = i;

console.log(t);

}

t = null; //对象如果已经不用了，那就立即设置为null；等待垃圾回收。

### 4.2.3 循环中的函数表达式，

循环中的函数表达式，能复用最好放到循环外面。

// 在循环中最好也别使用函数表达式。

// for(var k = 0; k < 10; k ++) {

// var t = function(a) { // 创建了10次 函数对象。

// console.log(a);

// };

// t(k);

// }

// 推荐用法

// function t(a) {

// console.log(a);

// }

// for(var k = 0; k < 10; k ++) {

// t(k);

// }

// t = null;

# 原型链

## 5.1 原型链基础链接关系

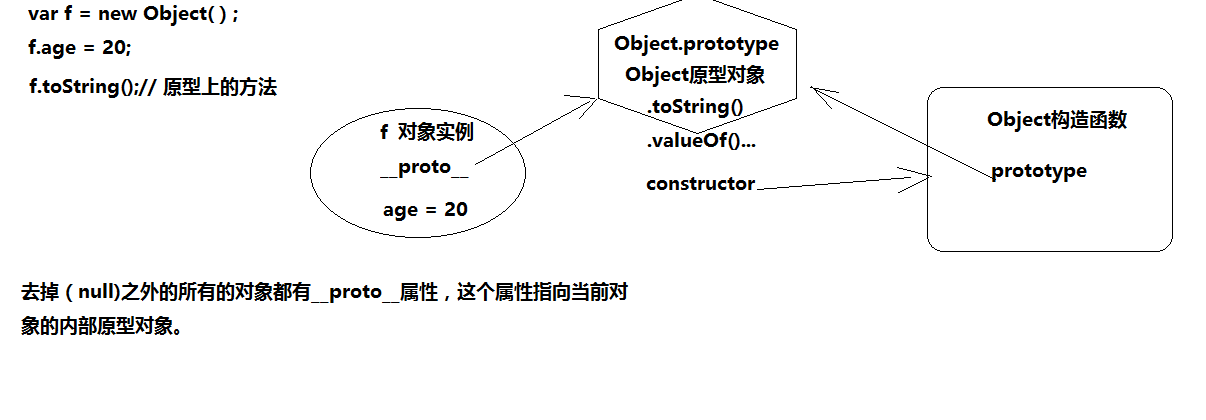
构造函数都有prototype属性，称为构造函数原型。

通过构造函数创建出来的对象都会继承构造函数原型上的方法和属性。

原型对象上都有constructor指向对应的构造函数。

var f = new Object(）;

f.age = 20;



## 5.2 构造函数的原型及内部原型

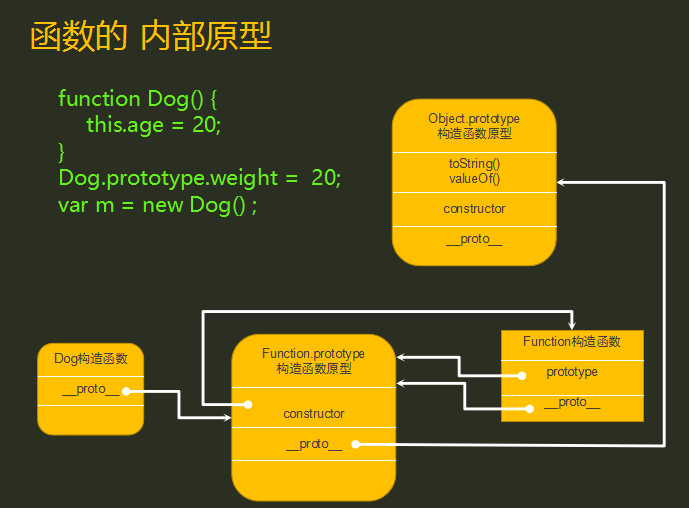
构造函数都有prototype属性，称为构造函数原型。

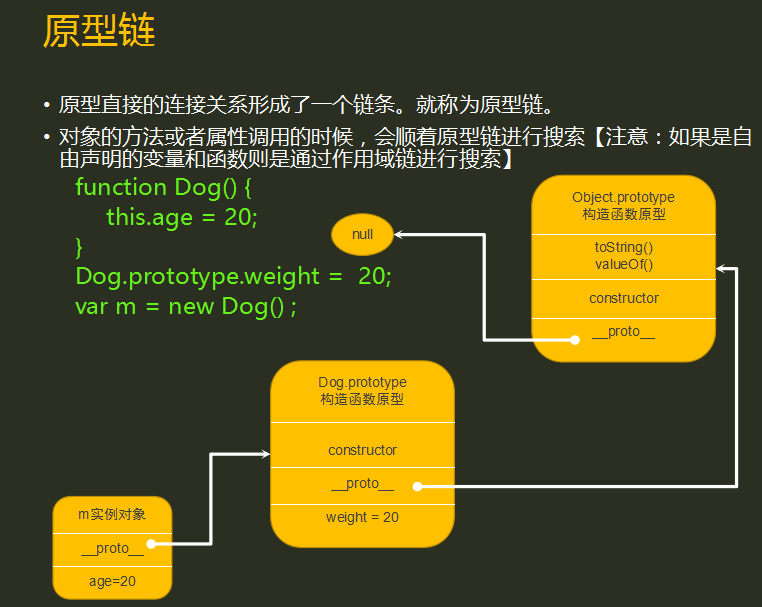
通过构造函数创建出来的对象都会继承构造函数原型上的方法和属性。

原型对象上都有constructor指向对应的构造函数。

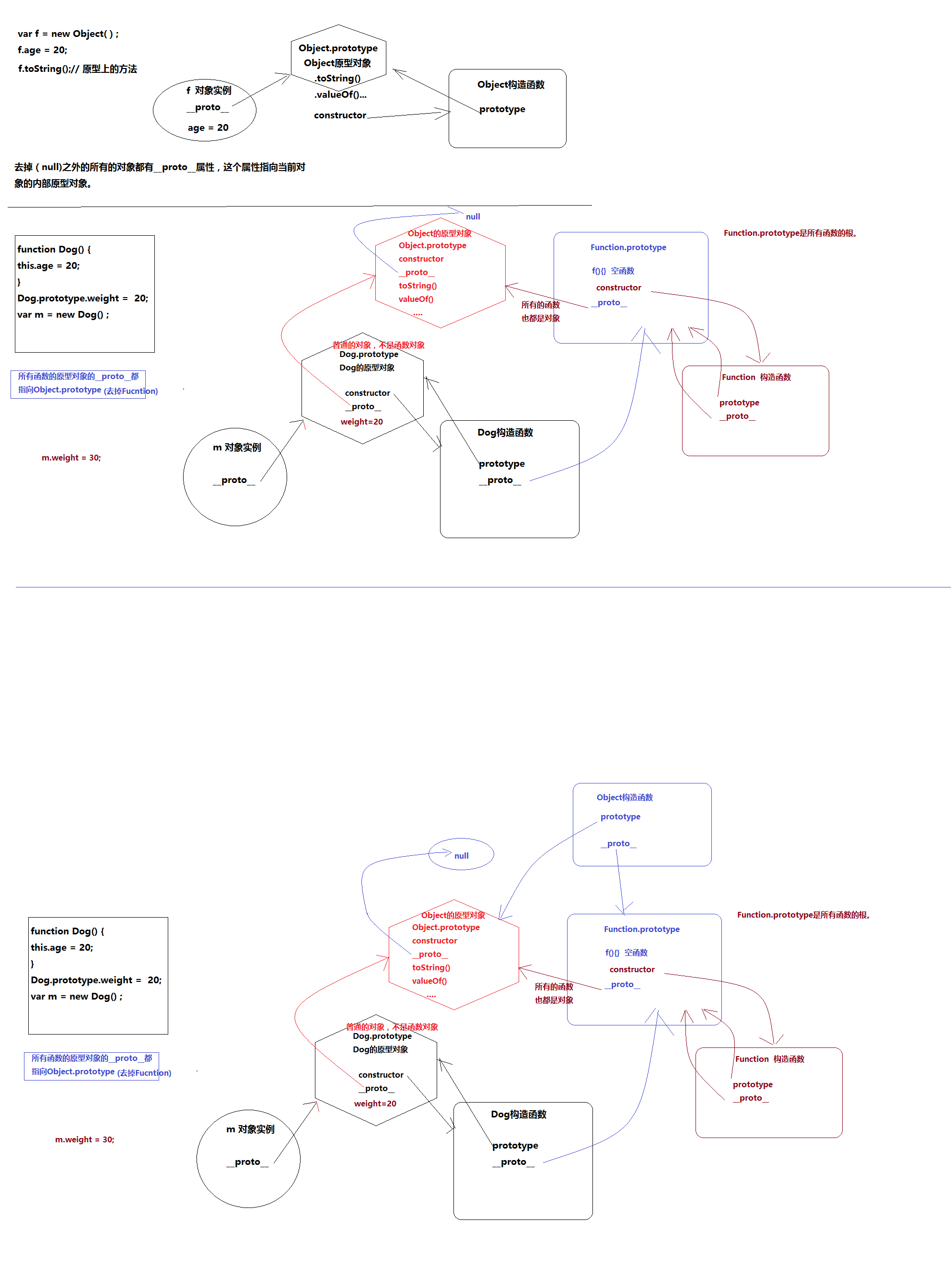
var f = new Object(）;

f.age = 20;





## 5.3 原型链总图



# 闭包

## 6.1 什么是闭包

闭包（英语：Closure）是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。

闭包是一个函数和函数所声明的词法环境的结合。

结论：JavaScript中任何一个函数都是闭包。

特殊情况：由于函数嵌套调用的时候，如果内部函数访问外部函数的变量。

function foo(x) {

    var tmp = 3;  
    return function (y) {  
        alert(x + y + (++tmp));  
    }  
}  
var bar = foo(2);   
bar(10);

//分析：

1、函数可以作为一个变量返回值

2、bar变量作为foo函数的返回值，指向的是一个函数。

3、虽然foo函数已经执行完成，temp变量已经离开了它的定义的环境，但是它内部的变量temp被bar一直引用，所以temp不会被释放。导致foo函数的闭包环境一直驻留内存中。bar是闭包对象。

## 6.2 闭包的应用

### 6.2.1 匿名自执行函数模拟块级作用域

由于JavaScript没有块级作用域，匿名自执行函数可以直接来模拟块级作用域。

(function(){

// 块级作用域环境

})();

// ;(function(a) {

// console.log(a);

// }(8));

### 6.2.2 循环注册dom事件中的index

// 从文档结构中获取所有的li标签。

var lis = document.querySelectorAll('li');

console.log(lis);

// // 循环注册事件的index的典型错误。

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

// lis[i].onclick = function(e) { //事件的方法执行是：当事件触发的时候执行。

// // 变量i是父函数里面的变量。

// console.log(i); // 把当前的索引i的值打印出来。

// };

// }

// // i == 5

// 把循环注册事件做成 自执行函数来传递变量i。

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

// lis[i].onclick = function(e) { //事件的方法执行是：当事件触发的时候执行。

// // 变量i是父函数里面的变量。

// console.log(i); // 把当前的索引i的值打印出来。

// };

(function(a) {

lis[a].onclick = function(e) {

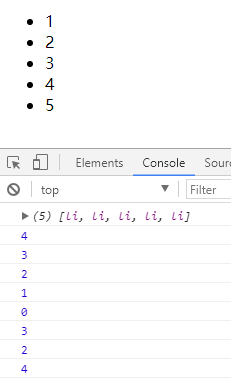
console.log(a);

};

})(i);

}

运行结果：



### 6.2.3 setTimeOut中的闭包应用

// 经过多少毫秒后，执行回调函数。

// bom的一个方法。接受两个参数：第一个参数是回调函数。

// 第二个参数是经过的毫秒数。

// setTimeout(function(){

// // 当1000毫秒后，执行当前的函数体。

// console.log('sss');

// }, 1000);

// for(var i = 0; i < 10; i++) {

// setTimeout(function() { // 注册1秒后的代码执行很快。

// console.log(i); // 函数体是在1秒后才执行。

// }, 1000);

// }

// 用匿名自执行函数解决

for(var i = 0; i < 10; i++) {

(function(a){

setTimeout(function() { // 注册1秒后的代码执行很快。

console.log(a); // 函数体是在1秒后才执行。

}, 1000);

})(i);

}

## 6.3 闭包的缺点

闭包会导致JavaScript执行效率下降。（目前V8引擎已经对闭包做了很多性能优化，基本不用考虑）

闭包导致内存会驻留，如果是大量对象的闭包环境注意内存消耗。

# 面向对象与继承

## 7.1 面向对象的概念

面向对象就是一种思考问题和组织代码的方式。

把任何的数据和行为抽象成一个形象的对象，类似于人生活中思考的方式，就是面向对象。

继承：相类似的对象进行公共抽象，并公共复用就是继承。

## 7.2 对象创建模式的演变

### 7.2.1 工厂模式创建对象

工厂创建对象的方法：

var a = {}; // 不能重复的利用 设置公共属性的代码。

// 工厂模式

// 我需要创建10个Cat对象，每个对象都有 年龄、姓名的属性，包括run方法。

function createCat(age, name) { // 工厂模式就是标准化的生成默认对象的函数。

var o = new Object();

o.age = age;

o.name = name;

o.run = function() {

console.log(o.name + ' running....');

};

return o;

}

var c = createCat(19, 'dddgg');

// 优点：

// 可以进行批量的创建 都有公共默认值和属性的对象。

// 缺点：

// 1、对象的方法不能跟其他对象共享，多占用内存。

// 2、不能识别对象的原型及构造函数。

// c instanceOf Cat;// false

优点： 可以重复创建相类似的对象。

缺点：

1、对象的原型不能确认具体的类型、对象的构造函数也不明确。

2、对象的方法不能重用，每个对象的内存中都存储一份函数对象的内存。

### 7.2.2 构造函数创建对象

构造函数模式



* 优点：

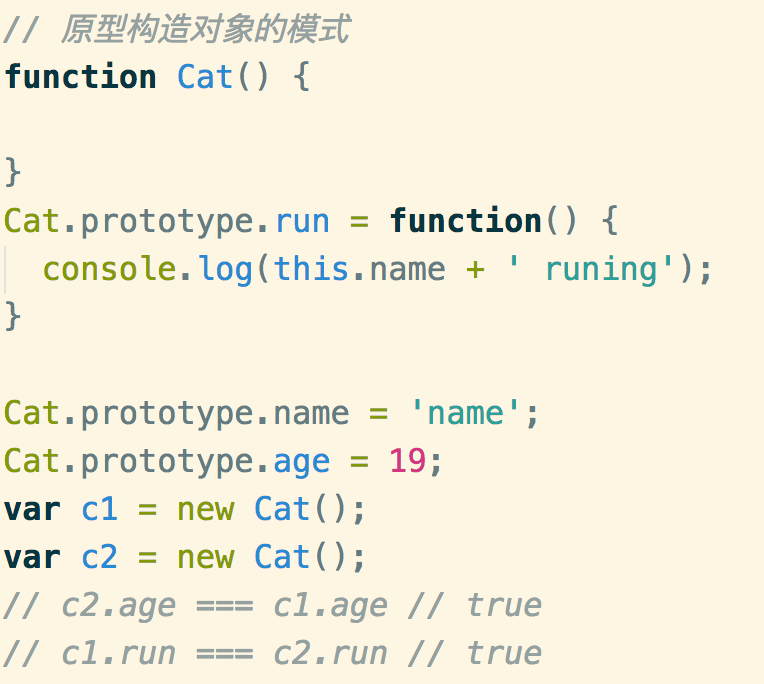
所有创建出来的对象，都可以找到它的原型和构造函数。

公共的属性和方法也可以在创建的时候统一创建和维护

* 缺点：

对象的函数，每个对象都会拥有一份内存拷贝。浪费内存

### 7.2.3 原型创建对象模式



优点：

所有原型上的属性和方法在所有的新对象中可以进行共享。

缺点：

如果对象需要自己特有的属性值，不与其他对象共享，则必须跟构造函数模式进行配合

### 7.2.4 组合模式创建对象（原型+构造函数）

组合使用构造函数模式与原型模式

公共的属性和方法放到 原型上，独有的属性使用构造函数模式，放到对象自己身上。



优点：

既保证了方法等共享的属性能只在内存中保存一份，节省内存。

又可以实现每个对象有自己单独存放的属性。是一种经典的构建对象的方法

### 7.2.5 稳妥构造函数模式

为了方便开发人员进行创建对象，对于有些可能忘记使用new的场景，兼容处理让开发人员可以使用new创建对象也可以不用new，达到相同的创建对象的效果。那么就需要使用稳妥构造函数模式。



优点:

可以共享属性和方法的初始化代码。

无论用户是否用了new还是没有使用new都会被正确的返回新的对象

缺点：

无法追溯对象的原型和构造函数，默认没有公共的属性和方法，内存浪费。

## 7.3 JavaScript继承

### 7.3.1 原型继承模式

原型继承模式，就是给子类的原型设置成父类的一个实例。

子类的对象实例就可以通过原型继承的方式获得父类中的属性和方法。



缺点：

不能给父类的构造函数传递参数。

父类中的引用类型的属性，会成为子类中共享所有。

解决：配合借用构造函数模式

### 7.3.2 组合继承模式

组合模式就是经典模式，解决了原型模式的两大问题。

组合继承模式：用原型继承原型属性和方法。用构造函数模式继承实例的属性和方法，非常经典的用法。

缺点：父类的构造函数需要执行两次。一次：设置原型对象，二次：构造函数借用时需要再执行一次。



// 组合继承模式

// 组合的原型继承和借用构造函数继承

// 父类

function Animal(age, name) {

this.age = age;

this.name = name;

this.foods = ['水', '苹果'];

}

// 在父类的原型上创建一个run方法

Animal.prototype.run = function() {

console.log(this.name + ' running');

};

// 定义子类

function Cat(age, name) {

// Animal(age, name); // this === window; 函数执行模式，this === window

// this == c

// 第一次执行父类的构造函数。

Animal.call(this, age, name); //借用父类的构造函数，给子类创建实例属性。

}

// 第二次执行父类的构造函数。

Cat.prototype = new Animal(); // 组合原型继承模式

Cat.prototype.constructor = Cat;

var c = new Cat(19, 'deail');

// 原型式继承模式

function object(o) {

function F() {}

F.prototype = o;

return new F();

}

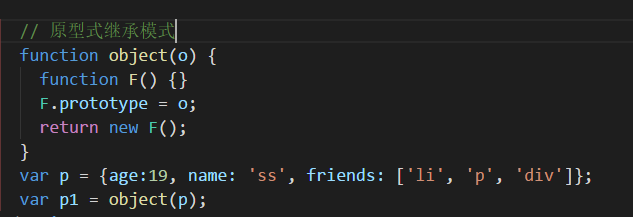
var p = {age:19, name: 'ss', friends: ['li', 'p', 'div']};

var p1 = object(p);

### 7.3.3原型式继

原型式继承是避免调用父类构造函数的一种巧妙的方式。本质就是借用对象来构造另外一个对象。

缺点：原型对象上的引用类型的属性会造成子类对象进行共享。



// o就是要借用的对象。

function object(o) {

function F() {}

F.prototype = o; // 让空函数的原型指向 o对象。

return new F(); // 创建一个f实例，f的内部原型指向o对象。

}

var m = {age = 19, name: 'laoma', friends: ["laoma2", "jimik" ]};

var m1 = object(m);

console.log(m1.friends);

m1.age = 20;

// 优点：

// 不需要使用new构造函数就可以直接 构造另外其他对象。

// 缺点：

// 所有构造出来的实例会共享 原型对象上的引用类型的属性。

### 7.3.4寄生继承模式

寄生继承模式是在原型式继承模式上增强原型对象的增强模式。只是对原型式继承的扩展而已。

寄生继承类似一个工厂模式，工厂内部把原型对象进行构造出另外一个实例，并对构造出来的实例进行增强，最后返回这个实例。

// 原型式继承的方法，传一个对象，

// 内部新构造一个对象，新对象的原型指向o对象

function object(o) {

function F() {}

F.prototype = o;

return new F();

}

// 寄生继承方式： 其实就是传一个对象到一个方法（工厂方法），方法内部

// 根据传来的对象构造一个新对象，并对新对象进行扩展增强，

// 返回新对象。

function createPersion(p) {

var o = object(p); // 同p对象构造一个新对象o

o.say = function() { // 对我新构造出来的对象o进行扩展

console.log('hi');

}

return o;

}

### 7.3.5 寄生组合继承模式

// 寄生组合继承模式

// 组合了：寄生继承模式和借用构造函数继承模式。

// 父类

function Animal(age, name) {

this.age = age;

this.name = name;

this.foods = ['水果','ls'];

}

// 父类原型上的方法：通过寄生继承的方式进行继承.

Animal.prototype.run = function() {

console.log(this.name + ' runnging');

};

function Cat(age, name) {

// 使用借用构造函数继承模式来构建对象的实例属性。

Animal.call(this, age, name);

}

// Cat.prototype = new Animal(); // 多执行了一次父类的构造函数。

// 寄生继承的方法

Cat.prototype = inheritFrom(Animal.prototype);

var c1 = new Cat(19, 'laom');

var c2 = new Cat(19, 'laom2');

c1.run();

c2.run();

// 寄生继承模式。

function inheritFrom(o) {

var t = object(o);

t.constructor = Cat; // 把Cat原型的构造函数指回Cat构造函数。

return t;

}

// 原型式继承的方法。

function object(o) {

function F() {}

F.prototype = o;

return new F();

}

## 7.4 私有变量

# 8、模块化演进

高级内容会包括：

* 函数高级
* 作用域深入
* 原型链
* 闭包
* 垃圾回收机制
* 异常处理
* 正则表达式
* JavaScript面向对象继承
* JavaScript模块化开发

请大家继续关注老马的传课官网的JavaScript高级课程。

老马联系方式

QQ：515154084

邮箱：malun666@126.com

微博：<http://weibo.com/flydragon2010>

百度传课：

<https://chuanke.baidu.com/s5508922.html>

微信：

