老马经典JavaScript高级教程

@马伦-flydragon

课程内容简介

- 第一章：开发环境搭建
- **第二章：HTML5各种标签及语义化**
- 第三章：CSS3及整站开发
- 第四章：JavaScript基础
- 第五章：JavaScript高级
- 第六章：移动端开发[css3 + html5]
- 第七章：前端高级框架
- 第八章：NodeJs
- 第九章：前端工作流及工程化

老马自我介绍

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

课程内容介绍

- JavaScript基础课程视频
  - 如果您没有基础请先观看老马的JS的基础视频教程
  - 地址：https://chuanke.baidu.com/v5508922-234347-1696511.html
  - 如果您Js基础ok，请越过。
  - 基础内容包括：
    - Js变量、类型、类型转换、运算符
    - 表达式、直接量、语句
    - 函数、函数表达式、函数参数构造函数
    - 对象创建、对象字面量、引用类型
    - 数组、数组排序、栈、队列
    - 包装类型、字符串操作、Math对象、全局对象
- JavaScript高级课程内容
  - 变量提升、函数提升
  - JavaScript函数深入、作用域链
  - 原型链、执行上下文
  - 引用传递、内存管理、垃圾回收
  - 闭包、构造函数、函数四种执行方式
  - JavaScript面向对象、继承
  - 模块化演进
  - 正则表达式、异常处理
  - 其他

第一节：值类型与引用类型及参数传递

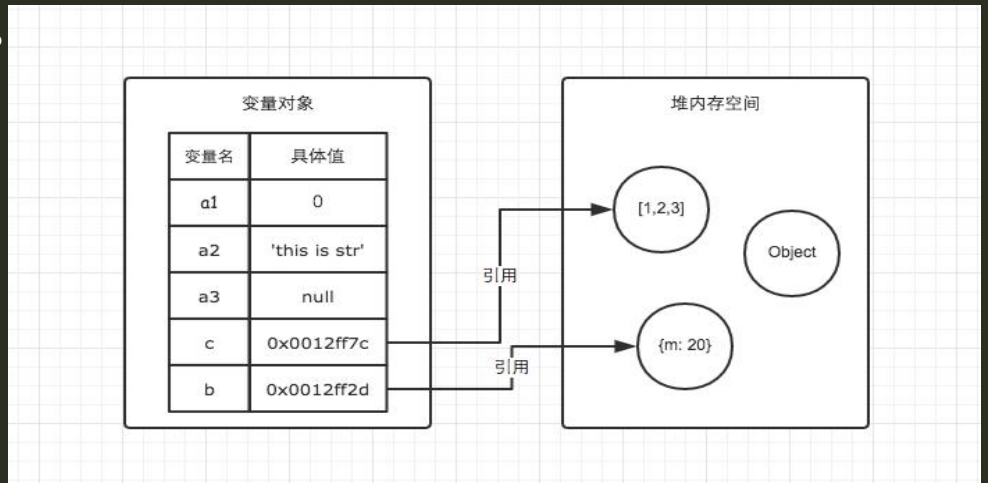
- 值类型与引用类型
- 值类型复制
- 引用类型复制
- 值传递和引用传递
- arguments
- 函数的lenght属性
- 参数对象化封装

变量的内存的存储结构

- 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：

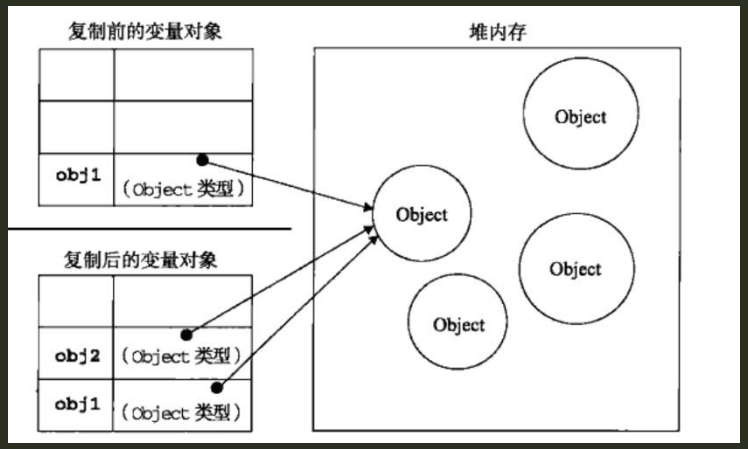
```
var a1 = 0; // 变量对象
var a2 = 'this is string';
var a3 = null;
var b = { m: 20 }; // 变量b存在于变量对象中，{m: 20} 作为对象存在于堆内存中
var c = [1, 2, 3]; // 变量c存在于变量对象中，[1, 2, 3] 作为对象存在于堆内存中
```

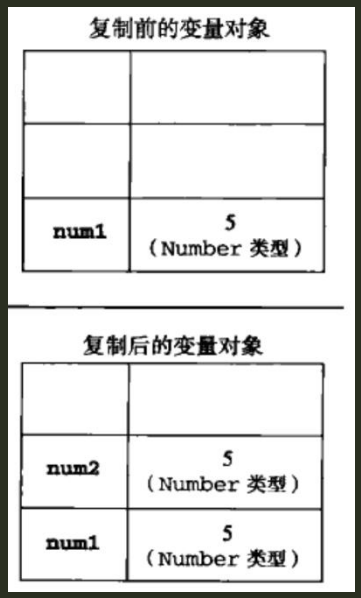
- 在内存中的存储情况为：
-



值类型和引用类型复制与传递

- 值类型的复制赋值和引用类型的赋值的区别。
- 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。





函数的参数

- 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。
- 【案例】模拟封装Math的max方法。
- 函数对象的length属性就是函数形参的个数。
- 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。
- 函数参数的值传递和引用传递
  - 引用传递的参数，是传递引用对象的地址。函数内部修改会影响传递参数的引用对象。
  - 值传递的是一个值类型的副本，函数内部不影响函数外部传递的参数变量。

第二节：函数高级内幕

- JavaScript事件循环机制
- 变量作用域
- 变量提升与函数提升
- 作用域链
- 函数的arguments
- 函数执行上下文
- 构造函数内部执行过程
- 函数的四种调用模式
- 没有重载
- 函数的属性和方法
  - length
  - prototype

JavaScript事件循环机制（难点）

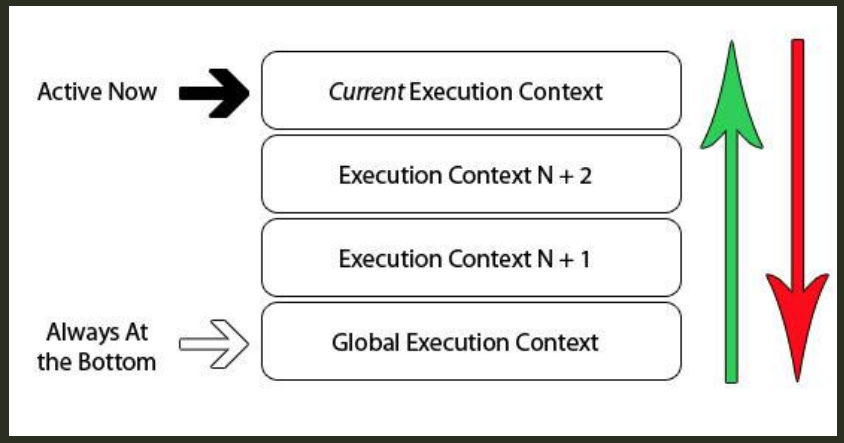
- 复习队列数据结构
- 事件循环机制： Event Loop
- JavaScript是单线程 的。
- 浏览器：
  - JavaScript执行线程：负责执行js代码
  - UI线程：负责UI展示，负责展示给用户看到的页面的。
  - JavaScript事件循环线程。
- 为什么是单线程的？JavaScript执行线程和UI线程是互斥的。
- JavaScript中的代码都是排队执行，不会同步执行多个任务。
- JavaScript中的任务分为同步任务和异步任务。
- 同步任务：一般的赋值操作、循环、分支语句等都是同步的任务。
- 异步任务：DOM事件、Ajax、BOM的一些API等
- 事件循环机制
  - 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

执行上下文相关的概念

- 栈的数据结构:先进后出。
- EC：函数执行环境（或执行上下文），Execution Context
- ECS：执行环境栈，Execution Context Stack
- **VO变量对象**（Variable object）是说JS的执行上下文中都有个对象用来存放执行上下文中可被访问但是不能被delete的*函数标示符、形参、变量声明*等。它们会被挂在这个对象上，对象的属性对应它们的名字对象属性的值对应它们的值但这个对象是规范上或者说是引擎实现上的不可在JS环境中访问到活动对象
- **AO激活对象**（Activation object）有了变量对象存每个上下文中的东西，但是它什么时候能被访问到呢？就是每进入一个执行上下文时，这个执行上下文儿中的变量对象就被激活，也就是该上下文中的函数标示符、形参、变量声明等就可以被访问到了
- scope chain：作用域链

执行上下文的执行栈

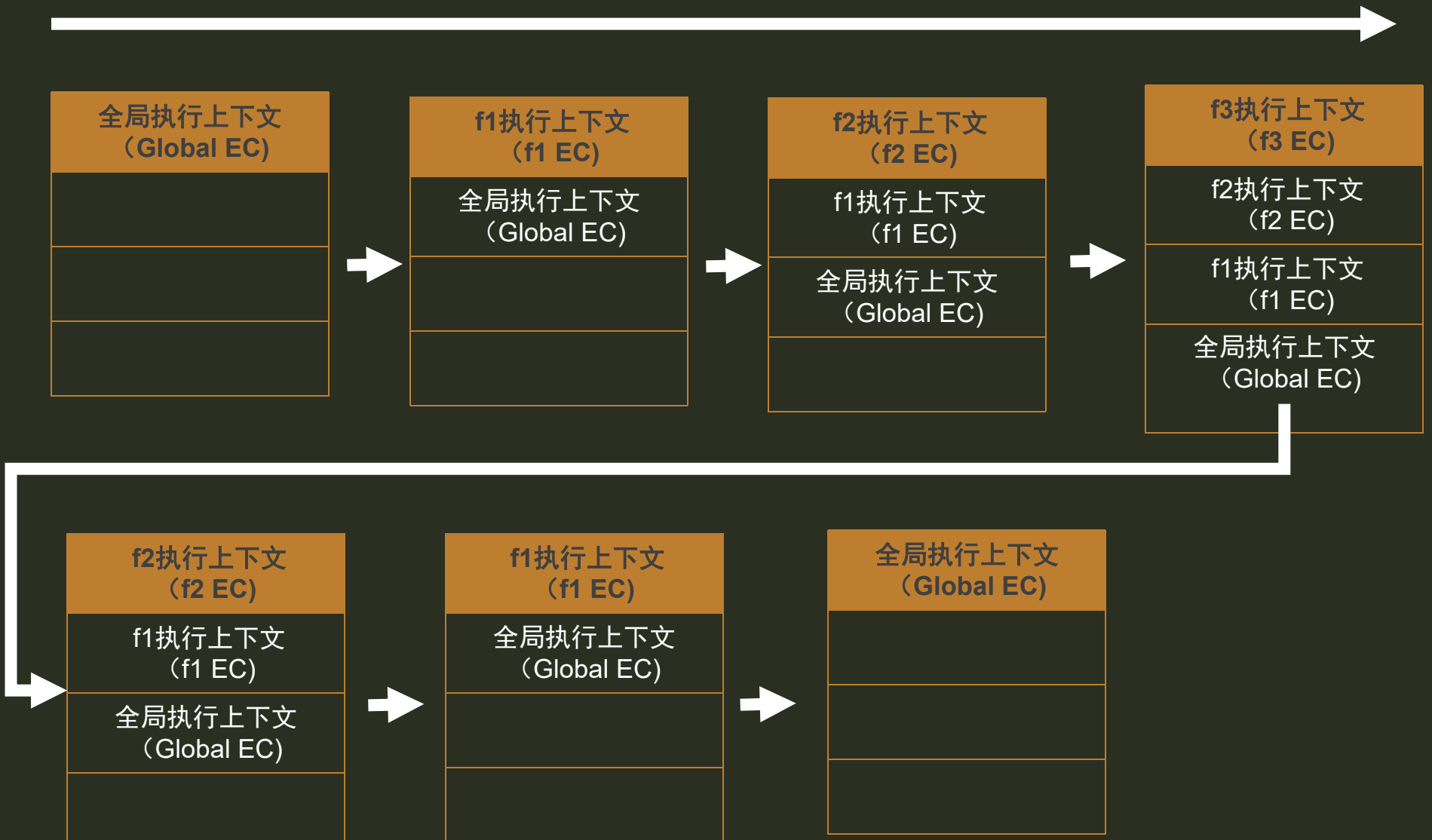
- JavaScript执行在单线程上，所有的代码都是排队执行
- 一开始浏览器执行全局的代码时，首先创建全局的执行上下文，压入执行栈的顶部。
- 每当进入一个函数的执行就会创建函数的执行上下文，并且把它压入执行栈的顶部。当前函数执行完成后，当前函数的执行上下文出栈，并等待垃圾回收。
- 浏览器的Js执行引擎总是访问栈顶的执行上下文。
- 全局上下文只有唯一的一个，它在浏览器关闭时出栈



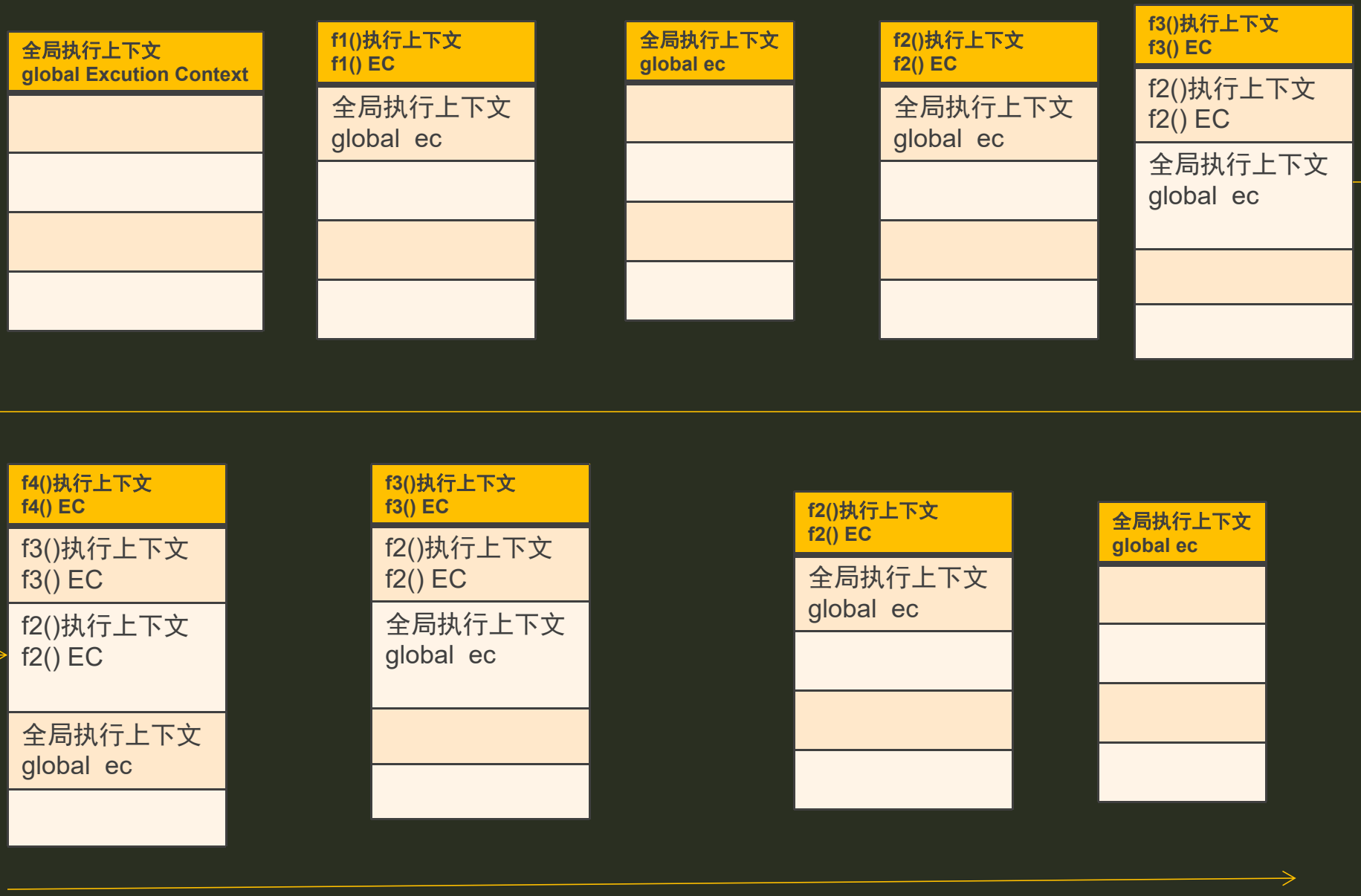
执行上下文的执行栈过程

```
<script>
function f1() {
console.log('f1.....')
f2();
}

function f2() {
console.log('f2.....')
f3();
}

function f3() {
console.log('f3 .....');
}

f1();
</script>
```

环境执行栈的变化状态

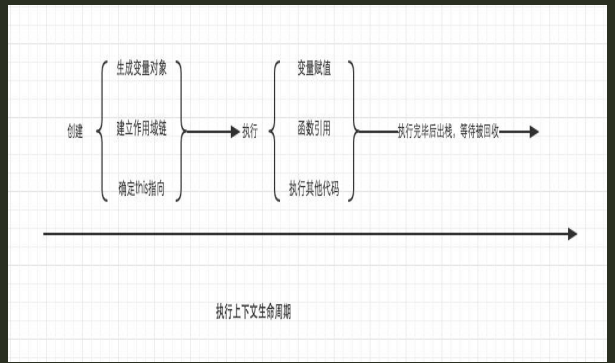

全局执行上下文
（Global EC)
f1执行上下文
（f1 EC)
全局执行上下文
（Global EC)
f2执行上下文
（f2 EC)
f1执行上下文
（f1 EC)
全局执行上下文
（Global EC)
f3执行上下文
（f3 EC)
f2执行上下文
（f2 EC)
f1执行上下文
（f1 EC)
全局执行上下文
（Global EC)
f2执行上下文
（f2 EC)
f1执行上下文
（f1 EC)
全局执行上下文
（Global EC)
f1执行上下文
（f1 EC)
全局执行上下文
（Global EC)
全局执行上下文
（Global EC)

执行环境栈ECS


全局执行上下文
global Excution Context
f1()执行上下文
f1() EC
全局执行上下文
global ec
全局执行上下文
global ec
f2()执行上下文
f2() EC
全局执行上下文
global ec
f3()执行上下文
f3() EC
f2()执行上下文
f2() EC
全局执行上下文
global ec
f4()执行上下文
f4() EC
f3()执行上下文
f3() EC
f2()执行上下文
f2() EC
全局执行上下文
global ec
f3()执行上下文
f3() EC
f2()执行上下文
f2() EC
全局执行上下文
global ec
f2()执行上下文
f2() EC
全局执行上下文
global ec
全局执行上下文
global ec

执行上下文的生命周期

- 总的的生命周期：创建→执行→出栈等待销毁
- 创建阶段：
  - 创建作用域链（Scope Chain）
  - 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——确切的说是函数的名字——其有一个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。
    - 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

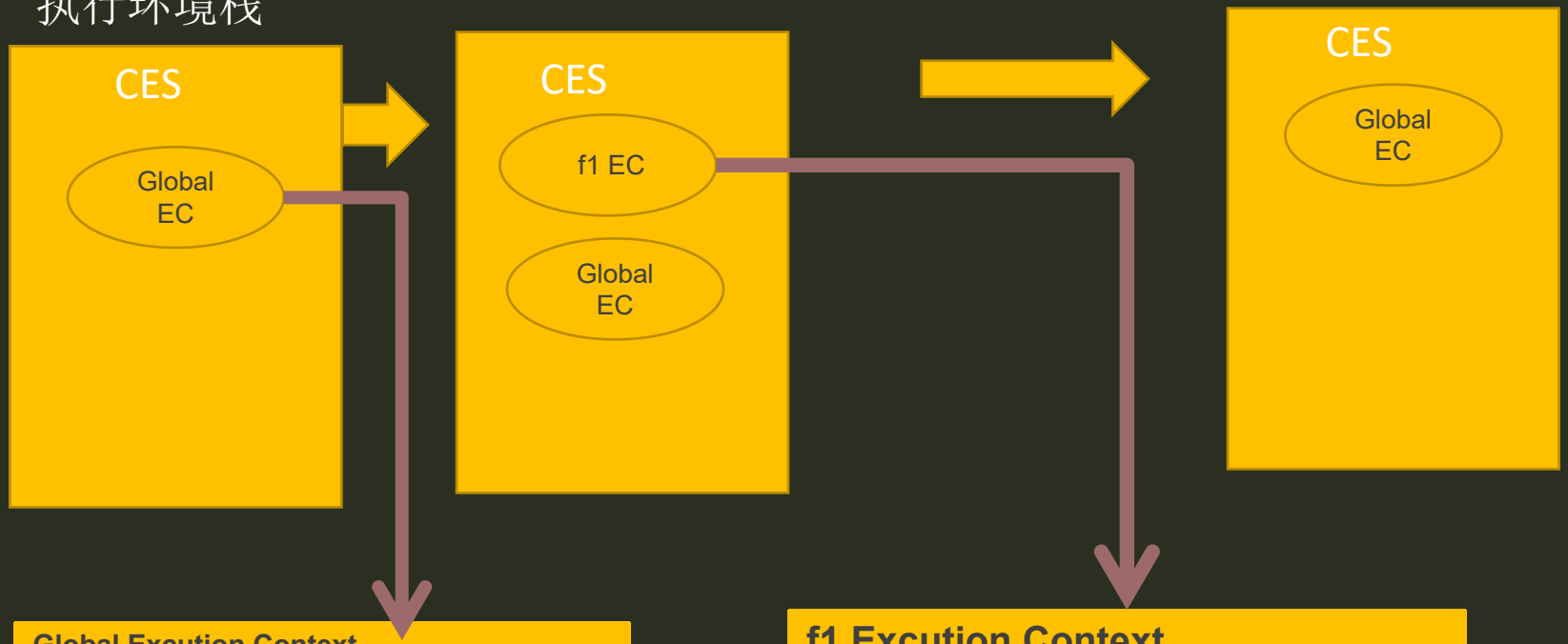


执行上下文的分析

- 例如代码：

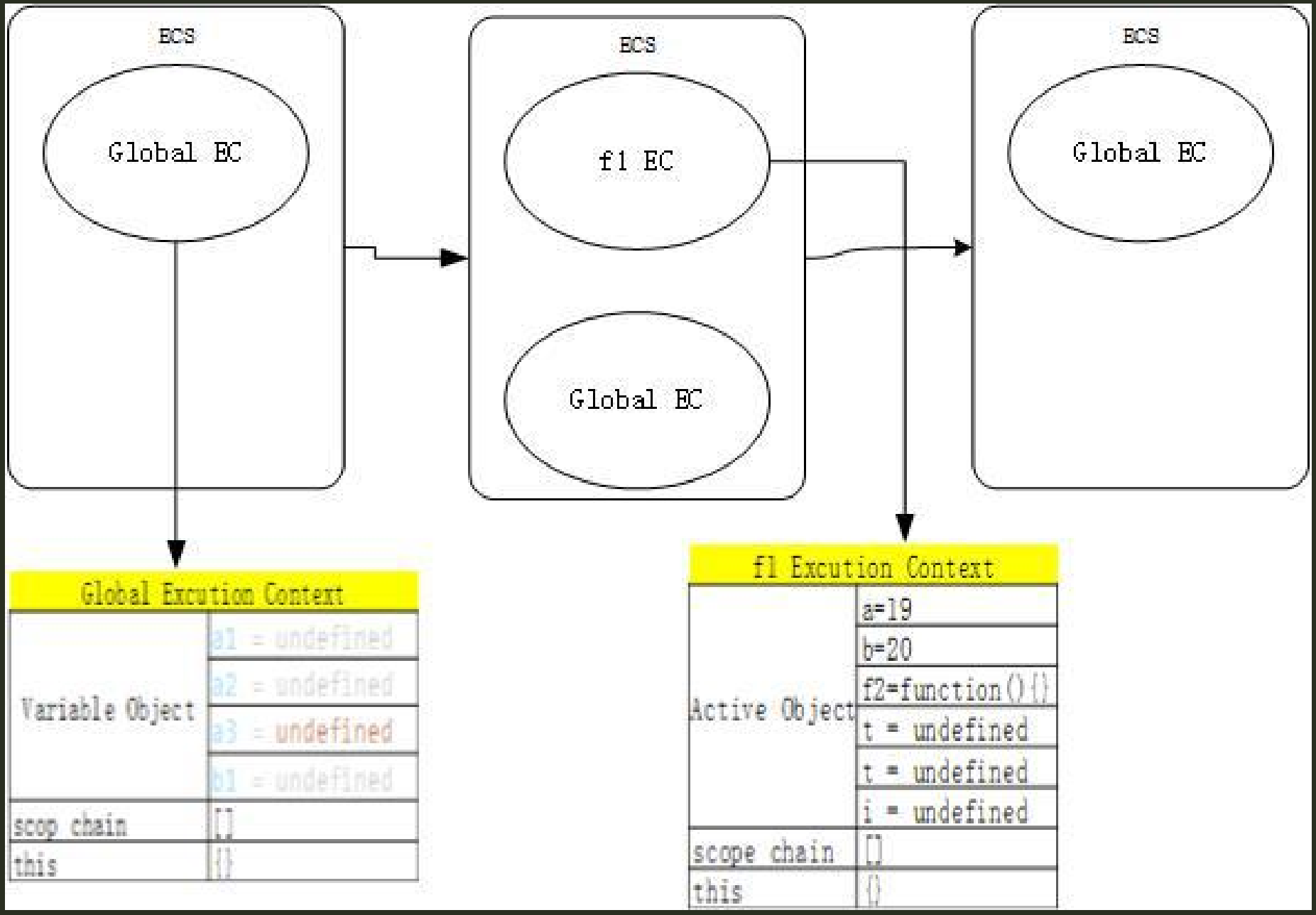
```
<script>
var a1 = 19,
    a2 = 20,
    a3 = 'sss',
    b1 = { name: 'laoma'};

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>
```

执行环境栈



Global Excution Context	
VO	f1=function（）{}
	a1=undefined
	a2=undefined
	a3=undefined
	b1=undefined
Scop chain	[]
this	{}

f1 Excution Context	
AO	a=19
	b=20
	f2=function（）{}
	t=undefined
	m=undefined
	i=undefined
Scop chain	[]
this	{}


ECS
Global EC
ECS
f1 EC
Global EC
ECS
Global EC
Global Excution Context
Variable Object
a1 = undefined
a2 = undefined
a3 = undefined
b1 = undefined
scop chain
[]
this
{}
f1 Excution Context
Active Object
a=19
b=20
f2=function(){}
t = undefined
t = undefined
i = undefined
scope chain
[]
this
{}

JavaScript的解释和执行阶段

- 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

函数变量的作用域

- 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。

分析作用域

```
<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>
```

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。

变量提升（hositing）

- 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。
- 解读代码：

```
if (!("a" in window)) {
    var a = 1;
}
console.log(a);
```

变量提升案例

- 案例1：

```
var a = 18;
function d() {
  console.log(a);
  var a = { age: 19};
  console.log(a);
}
d();  //  输出？
console.log(a);
```

案例2：

```
if (!("a" in window)) {
        var a = 1;
}
console.log(a);
```

案例3：

```
console.log(a);
var a = 20;
console.log(a);
function a() {
}
```

案例4：

```
f();
console.log(a);
console.log(b);
console.log(c);
function f() {
    var a = b = c = 9;
    console.log(a);
    console.log(b);
    console.log(c);
}
```

案例5：

```
f();
function f() {
    for(var k = 0; k <10; k++)
    {
        console.log(k);
    }
    console.log(k);
}
```

函数提升

- 只有声明式函数才会被提升，字面量函数不会被提升
- 总结：

  1)函数声明会置顶
  2)变量声明也会置顶
  3)**函数声明比变量声明更置顶：(函数在变量上面)**
  4)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
  5)声明过的变量不会重复声明
  6)遗漏声明的变量是全局变量

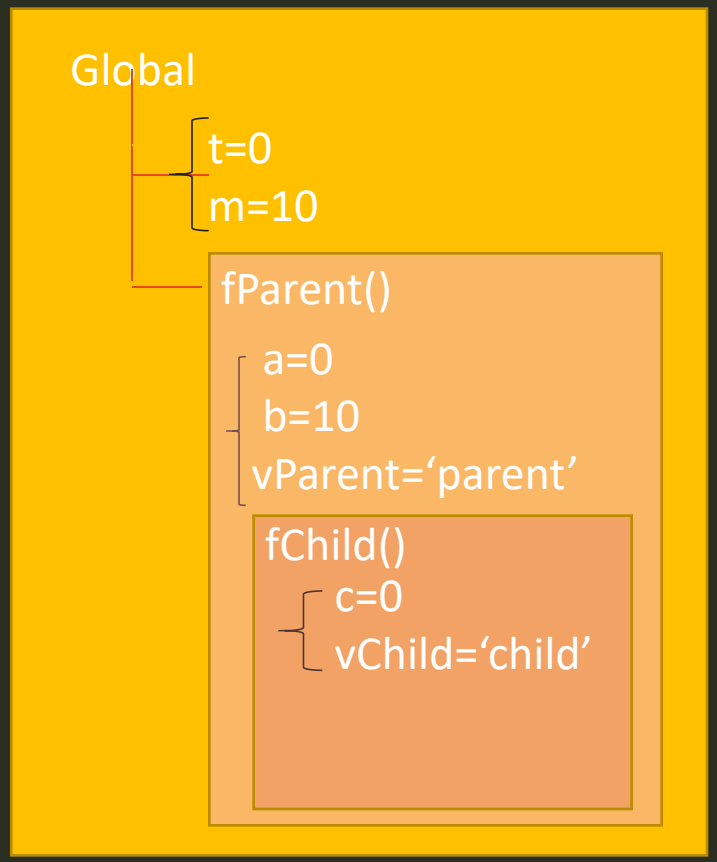
作用域链

- 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，作用域链中存放的是执行环境中的vo或者ao。
- 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。

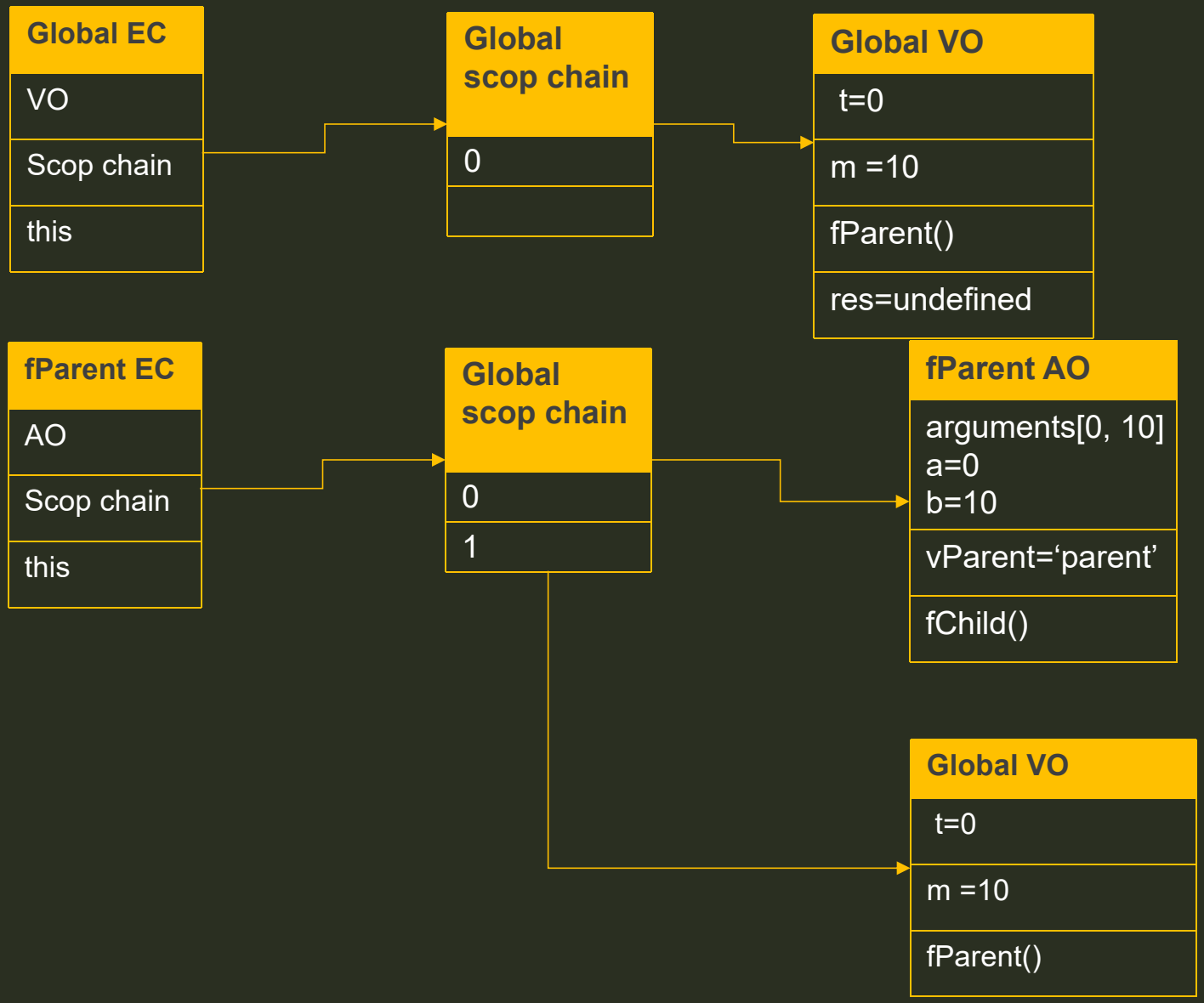
作用域链的代码分析

比如代码：

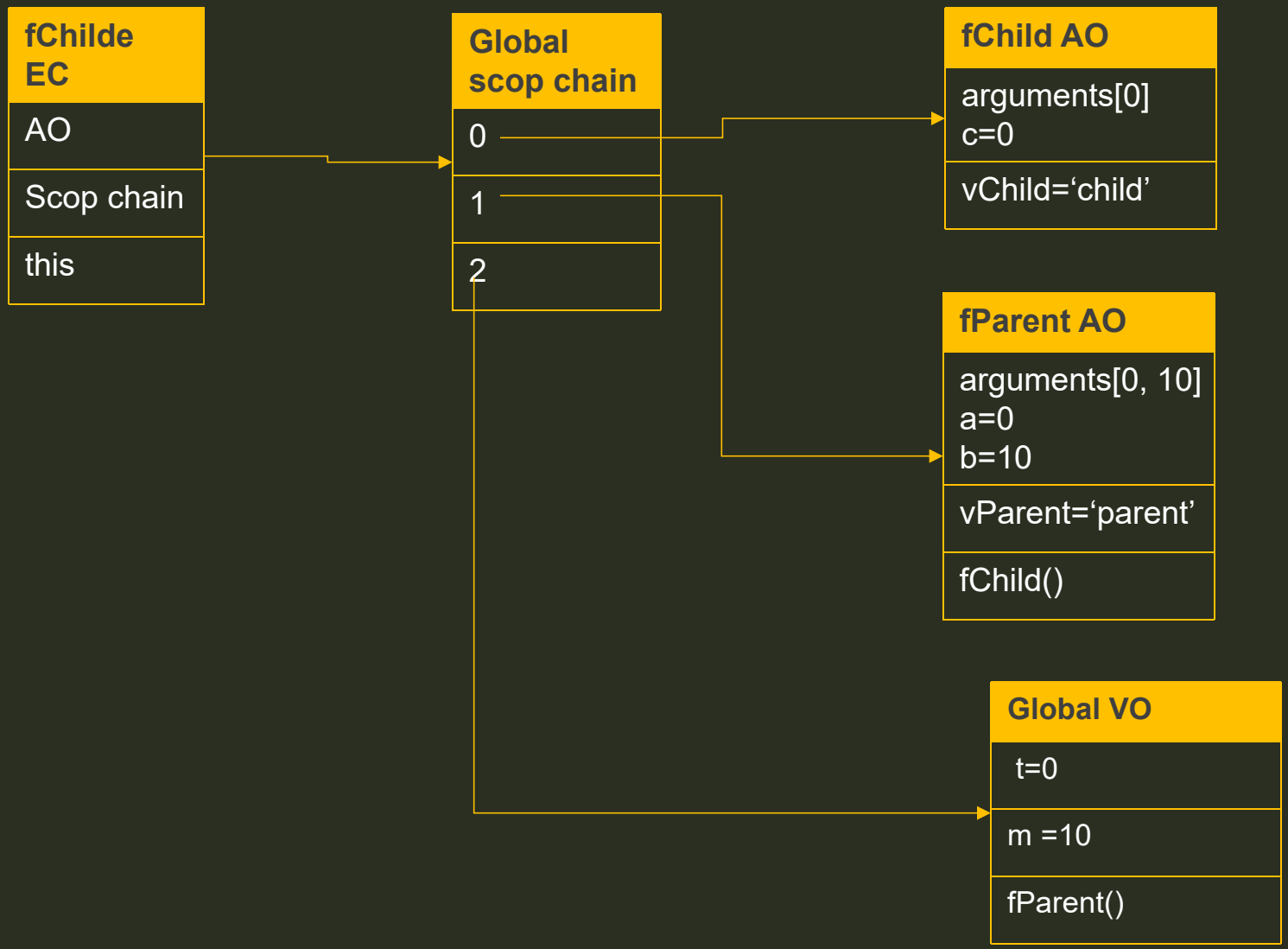
```
var t = 0, m = 10;

var res = fParent(t, m);
console.log(res);

function fParent(a, b) {
        var vParent = 'parent';

        function fChild(c) {
                var = 'child';
                returnvChild c + vChild;
        }
        return a + b + vParent +fChild(a);
}
```



作用域链的内部结构


Global EC
VO
Scop chain
this
Global scop chain
0
Global VO
t=0
m =10
fParent()
res=undefined
fParent EC
AO
Scop chain
this
Global scop chain
0
1
fParent AO
arguments[0, 10]
a=0
b=10
vParent='parent'
fChild()
Global VO
t=0
m =10
fParent()

原型链的内部结构


fChilde EC
AO
Scop chain
this
Global scop chain
0
1
2
fChild AO
arguments[0]
c=0
vChild='child'
fParent AO
arguments[0, 10]
a=0
b=10
vParent='parent'
fChild()
Global VO
t=0
m =10
fParent()

函数四种调用模式与this

- 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
- 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
  - 方法调用模式的this指向调用对象。
- 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。
  - 不管在什么地方定义的，直接调用函数都属于这种。如果是严格模式this==undefined
  - function f() {}; f();//函数调用模式
- 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]);

函数四种调用模式案例

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();// 函数调用模式
var d = new Dog(); //构造函数调用模式  this == d
Dog.call(null);  // 借用调用模式 window
```

JavaScript中函数没有重载

- 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。*

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：
  - 求1到100的和，请使用递归。
  - 求num的阶乘，用递归实现。
  - 求f(n)的斐波那契数列的值。 f(n) = f(n-1) + f(n-2); n>=3

函数式编程

- 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。
- 数组的sort方法回顾。
- 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
- 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**

函数的属性和方法

- 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】

第三节：垃圾回收

- 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

垃圾回收的应用

- 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。
- 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。
- 在循环中的函数表达式，能复用最好放到循环外面。

第四节：原型链与闭包

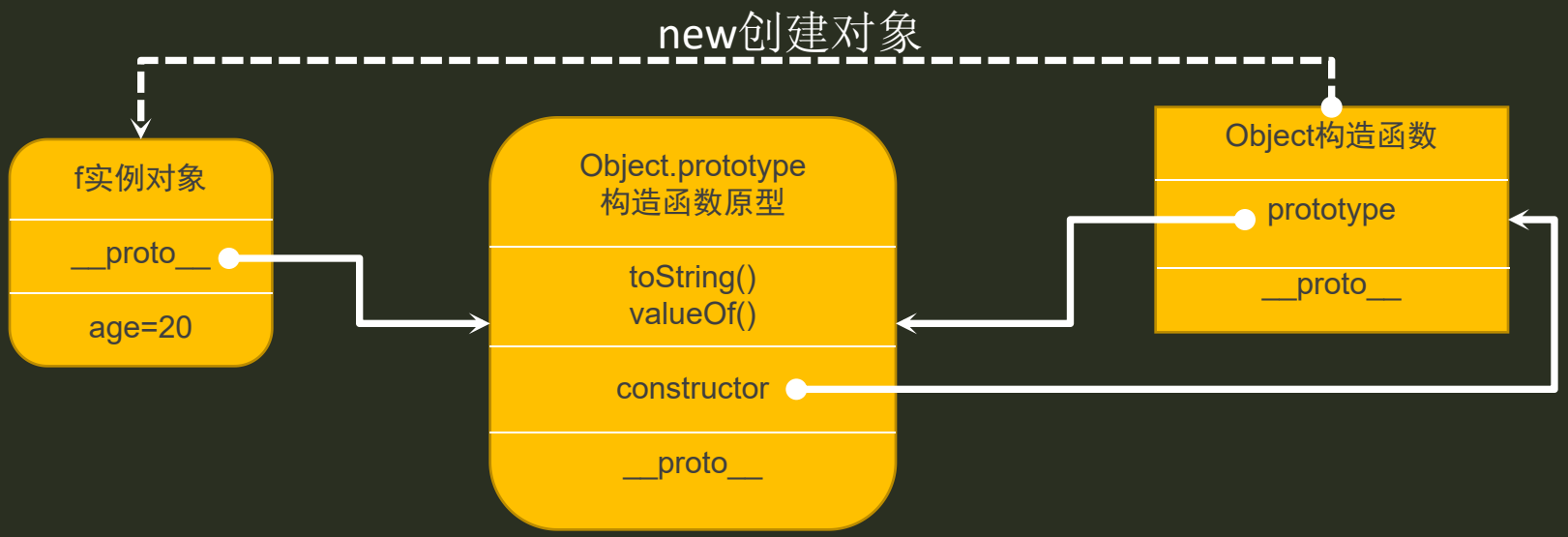
- 原型链
  - 函数的原型对象(prototype)
  - 原型对象的构造函数(constructor)
  - 内部原型(__proto__)
- 闭包
  - 函数的作用域：函数作用域与全局作用域
  - 没有块级作用域
  - 闭包的使用
  - 沙箱模式

原型链

- JavaScript是基于原型的面向对象语言。也就是说所有对象都是基于原型上进行创建，而不像Java和C#等是基于一个类型的模板创建。
- 函数都有prototype属性指向函数的原型对象。【只有函数根除外】
- 所有的对象都有__proto__属性（非标准属性，但是所有的浏览器都实现了。【null除外】
- Object是构造函数，也是对象。Object的prototype是所有对象的根。
- Function是函数对象的构造函数。Function的原型对象是所有函数的根。而它的内部原型是Object的原型对象，这就是关键点了。

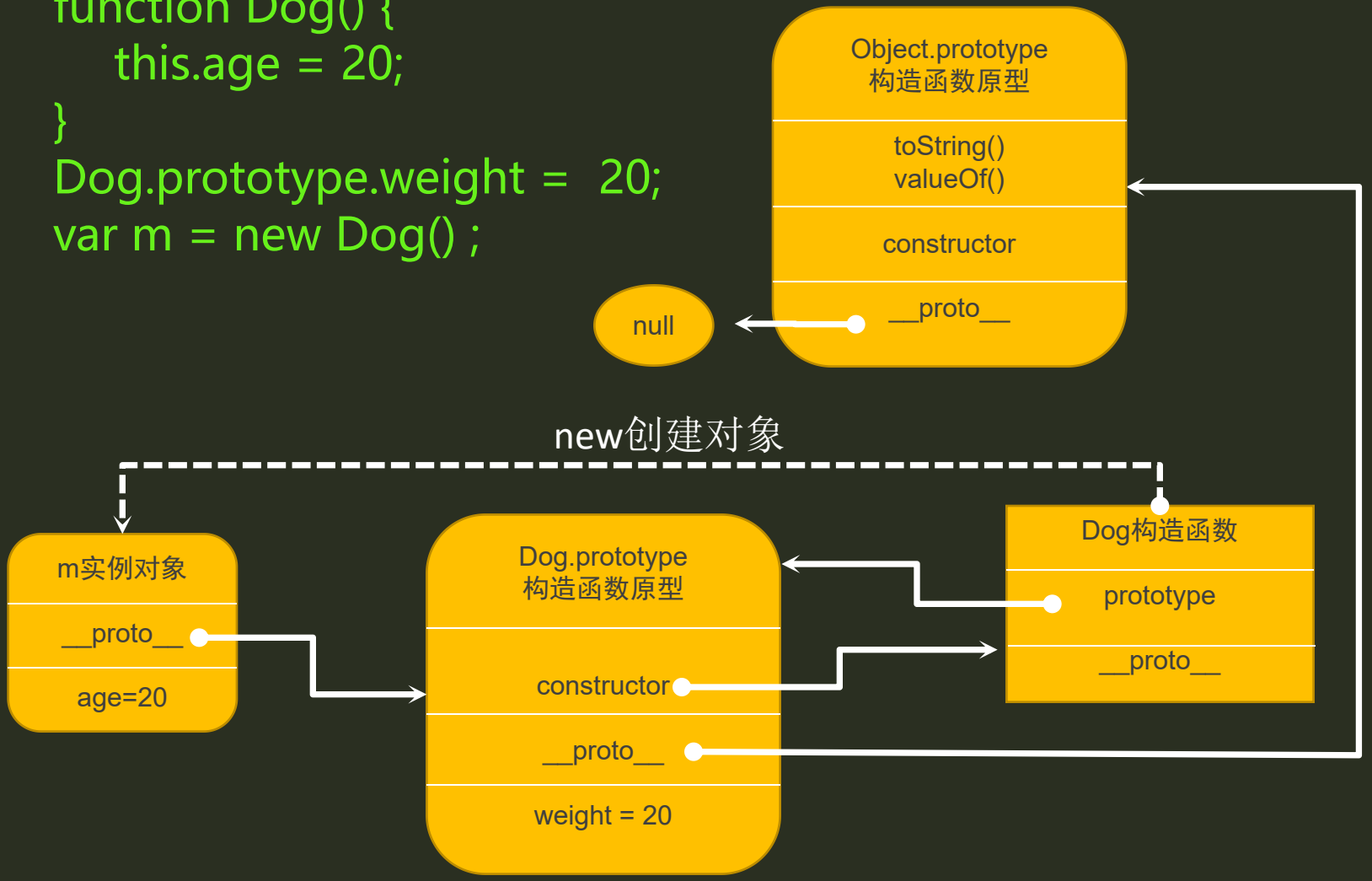
构造函数的原型对象

- 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;



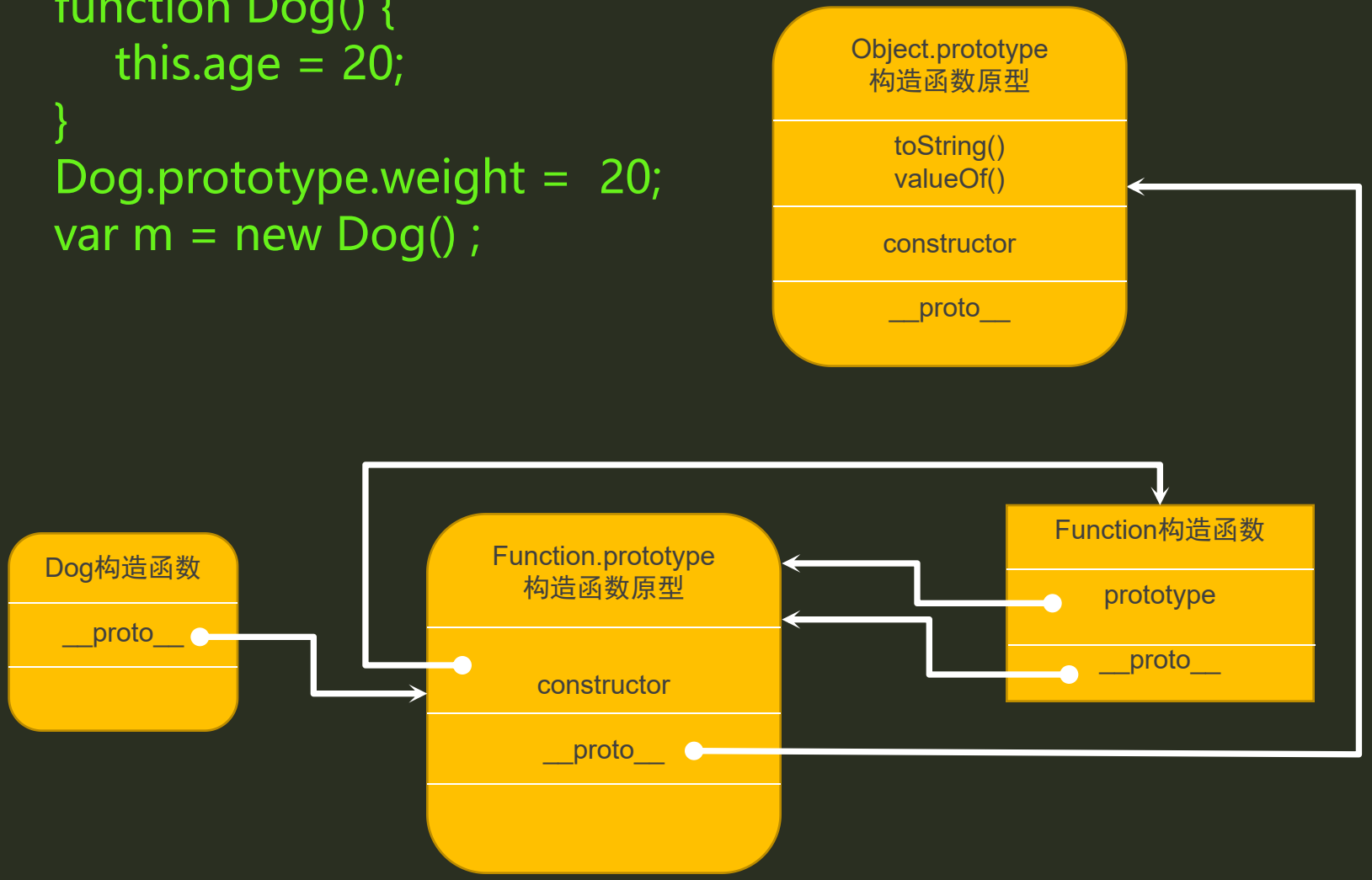
构造函数的原型对象

```
function Dog() {
    this.age = 20;
}
Dog.prototype.weight =  20;
var m = new Dog() ;
```



函数的 内部原型

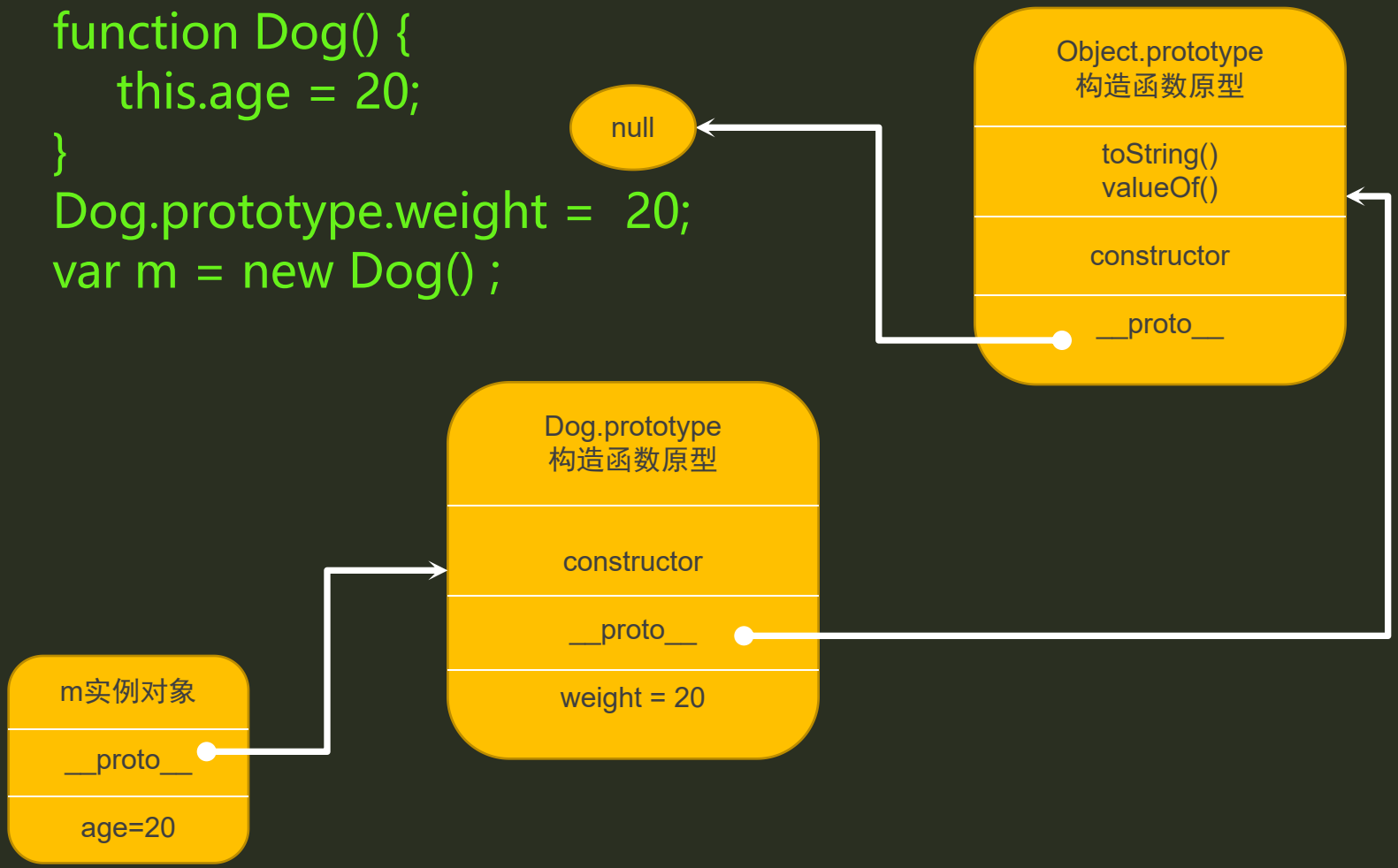
```
function Dog() {
    this.age = 20;
}
Dog.prototype.weight =  20;
var m = new Dog() ;
```



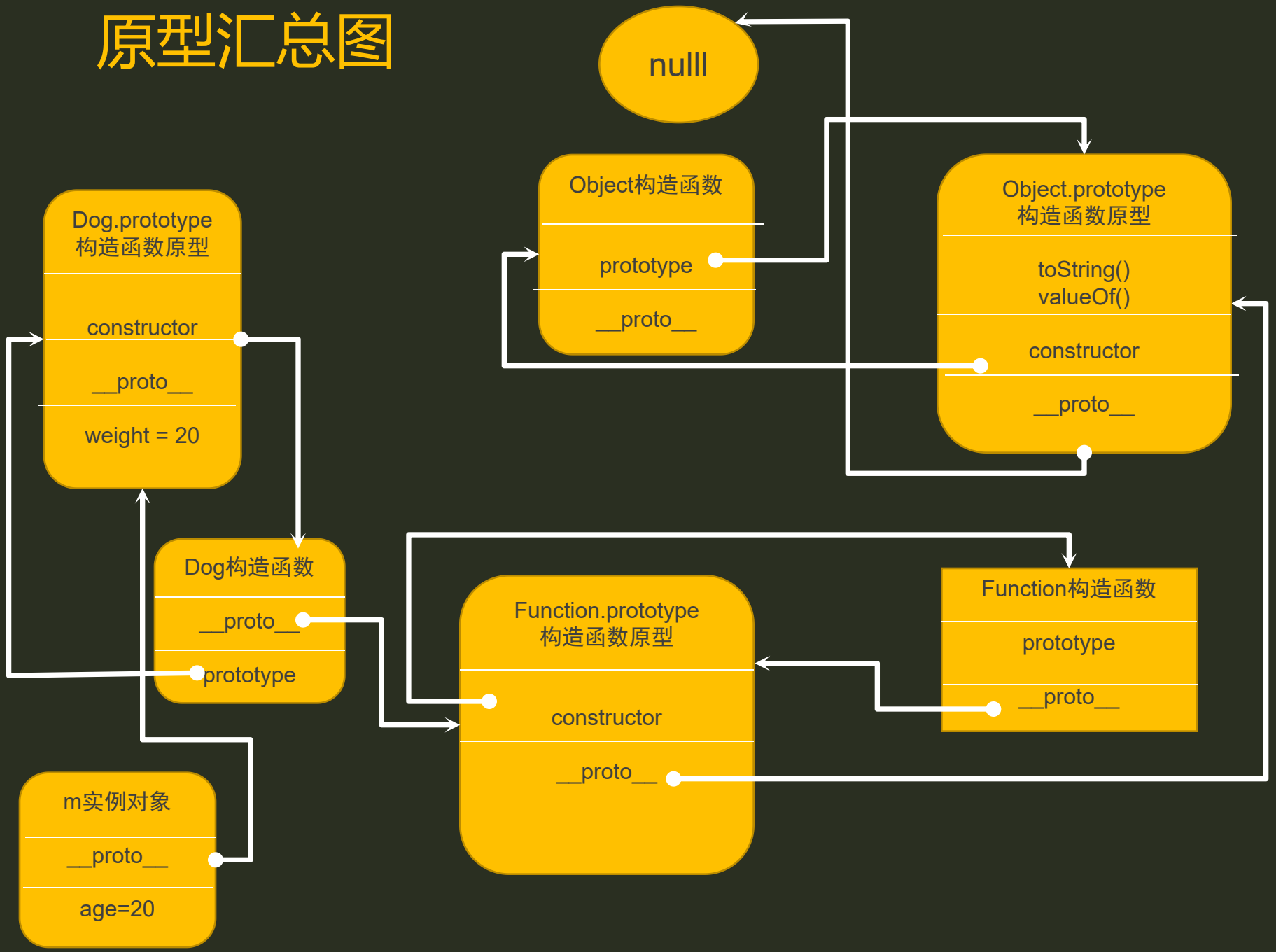
原型链

- 原型直接的连接关系形成了一个链条。就称为原型链。
- 对象的方法或者属性调用的时候，会顺着原型链进行搜索【注意：如果是自由声明的变量和函数则是通过作用域链进行搜索】

```
function Dog() {
    this.age = 20;
}
Dog.prototype.weight =  20;
var m = new Dog() ;
```



原型汇总图


nulll
Object构造函数
prototype
__proto__
Object.prototype
构造函数原型
toString()
valueOf()
constructor
__proto__
Dog.prototype
构造函数原型
constructor
__proto__
weight = 20
Dog构造函数
__proto__
prototype
Function.prototype
构造函数原型
constructor
__proto__
Function构造函数
prototype
__proto__
m实例对象
__proto__
age=20

闭包（Closures）

- **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。

闭包的应用

```
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。

闭包的应用

- 匿名自执行函数模拟块级作用域

```
(function(){
    // 块级作用域环境
})();
```

- 循环注册dom事件中的index

- setTimeOut中的闭包应用

```
for(var i = 0 ; i < 10; i++) {
    setTimeout(function(){
            console.log(i);
    },1000);
}
```

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。

第五节：面向对象

- 面向对象的概念
- 对象的创建
- 面向对象继承

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。

对象属性和行为复用

- 工厂的方式创建对象
- 构造函数创建对象
- 原型创建对象
  - 原型动态性
  - 赋值新对象的原型创建方式及constructor处理
- 混合模式：原型+构造函数
- 寄生构造函数模式
- 稳妥构造函数模式

工厂方式创建对象

- 工厂创建对象的方法：
- 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。

```
function createCat() {
	var o = {};
	o.age = 19;
	o.name = 'cat';
	o.run = function() {
		console.log('');
	};

	return o;
}

var c1 = createCat();
var c2 = createCat();
c1.run();
c2.run();
```

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

```
// 构造函数模式
function Cat(age, name) {
  this.age = 19;
  this.name = 'ss';
  this.run = function() {
    console.log(name + ' runing...');
  };

  // 构造函数内部（在new的前提下）
  // 创建新对象
  // 将新对象的内存空间赋值给this
  // 执行构造函数，给this初始化属性和方法
  // 返回新创建的对象。
}

var c1 = new Cat(19, 'sss');
c1.run();
```

原型构建对象

- 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

```
// 原型构造对象的模式
function Cat() {

}
Cat.prototype.run = function() {
  console.log(this.name + ' runing');
}

Cat.prototype.name = 'name';
Cat.prototype.age = 19;
var c1 = new Cat();
var c2 = new Cat();
// c2.age === c1.age // true
// c1.run === c2.run // true
```

组合构造函数模式与原型模式构建对象

- 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

```
// 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。
function Cat(age, name) {
  this.age = age;
  this.name = name;
}

Cat.prototype.run = function() { // 给原型添加方法
  console.log(this.name + ' runing....');
}

var c1 = new Cat(19, "c1");
var c2 = new Cat(20, "c2");
// c1.run === c2.run  // 共享原型上的方法
// c1.age == 19;
// c2.age == 20;
// 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。
```

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。*

```
// 稳妥构造函数模式
function Persion(age, name) {
  var o = {};
  o.age = age;
  o.name = name;
  o.run = function() {
    console.log(o.name + ' runing....');
  }
  return o;
  // 当构造函数有返回值的时候，如果是简单类型直接忽略，还是返回新对象（this）.
  // 如果是引用类型，则直接返回次引用类型替换掉新对象（this）
}

var c1 = new Persion();
var c2 = Persion();       // 不使用new也能正确获得对象
// 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。
```

对象的继承

- 原型继承的方式
- 借用构造函数继承
- 组合继承
- 原型方法继承（原型式）
- 寄生继承方式
- 终极方式：寄生组合的方式

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。
  - 父类中的属性都会成原型属性，是所有实例共享的。而部分熟悉我们希望做成实例的属性。
  - 解决：配合借用构造函数模式

```
// 原型对象继承模式
function Animal(age, name) {
  this.age = age;
  this.name = name;
  this.needs = ['空气','水'];
}

function Cat(age, name) {
  this.age = age;
  this.name = name;
}
Cat.prototype = new Animal();
// 把Cat的原型的构造函数属性设置回Cat
Cat.prototype.constructor = Cat;
var c = new Cat();

// 问题:
// 1、初始化的代码，在子类和父类中重复。不能向父类的构造函数传参数。
// 2、父类中的引用类型的属性，所有子类对象都是公共的，相互之间相互影响。
```

组合借用构造函数模式与原型继承模式

- 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。

```
// 借用构造函数模式配合原型继承模式
function Animal(age, name) {
  this.age = age;
  this.name = name;
  this.needs = ['空气','水'];
}

function Cat(age, name) {
  // 关键点：借用父类的构造函数并，this换成当前新对象。
  //这样可以解决原型继承中的两大问题
  // 1、给父类的构造函数传参数
  // 2、父类中的引用类型的对象子类共享的问题。
  Animal.call(this, age, name);
}
Cat.prototype = new Animal();
// 把Cat的原型的构造函数属性设置回Cat
Cat.prototype.constructor = Cat;
var c = new Cat();
```

原型式继承

- 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。

```
// 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);
```

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。

```
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;
}
```

寄生组合继承模式

- 由于组合继承模式需要执行两次的父类的构造函数，使用寄生继承模式替换原型继承模式，那么就可以实现父类的构造函数只执行一次的效果，是目前认为最经典的继承模式，最好的继承模式。YUI库也是大量使用了这种继承模式。

私有变量

- JavaScript中并没有私有变量的概念，但是我们可以通过某些方式进行模拟。所谓的私有变量就指：对象的某个属性只能通过对象的方法进行访问，不能直接通过对象进行访问。
- 构造函数模拟私有变量的方式
- 其他模拟方法

第六节：模块化演变

- js的全局变量
- var a = b = c = 9; // b 和c 是全局变量。
- JS开发的灾难：
  - 全局变量的相互污染？
  - 不同开发人员不同js文件相互干扰
  - 不同框架中的全局变量相互干扰
  - 没有模块、没有命名空间等手段
- 解决多人合作和模拟模块的尝试
  - 模拟命名空间
  - 自执行函数模拟局部作用域
  - 模块化尝试封装

第七节：正则表达式

- 什么是正则表达式？
  - 正则表达式：用于匹配字符串规律规则的表达式。正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。
- 正则表达式的用途
  - 给定的字符串是否符合正则表达式的过滤逻辑(匹配)
  - 可以通过正则表达式，从字符串中获取我们想要的特定部分(提取)
  - 强大的字符串替换能力(替换)
- 正则表达式学习工具
  - 在线正则表达式工具： https://c.runoob.com/front-end/854
  - sublime、vscode

元字符

元字符	说明
\d	匹配数字
\D	匹配任意非数字的字符
\w	匹配字母或数字或下划线
\W	匹配任意不是字母，数字，下划线
\s	匹配任意的空白符
\S	匹配任意不是空白符的字符
.	匹配除换行符以外的任意单个字符
^	表示匹配行首的文本(以谁开始)
$	表示匹配行尾的文本(以谁结束)
\n	换行

限定符

限定符	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次 == {0， 1}
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

括号相关

- [] 字符串用中括号括起来，表示匹配其中的任一字符，相当于或的意思 。中间有横线表示范围。
  - [0-9] 等价 \d　[abc] [a-z]
- [^] 匹配除中括号以内的内容
  - [^0-9] [^a-zA-Z]
- \ 转义符
  - \\ => \　\. => .
- | 或者，选择两者中的一个。注意|将左右两边分为两部分，而不管左右两边有多长多乱
  - \w+|\s+
- () 从两个直接量中选择一个，分组
  - gr(a|e)y匹配gray和grey
- [\u4e00-\u9fa5] 匹配所有的汉字

常见验证案例

- 验证手机号：
  - ^\d{11}$
- 验证邮编：
  - ^\d{6}$
- 验证日期 2012-5-01
  - ^\d{4}-\d{1,2}-\d{1,2}$
- 验证邮箱 xxx@aicoder.com：
  - ^\w+@\w+\.\w+$
- 验证IP地址
  - 192.168.1.10 5.5.5.5 122.33.44.2
  - ^\d{1,3}(\.\d{1,3}){3}$

贪婪模式

- 贪婪模式：
  - 就是尽可能多的匹配字符串。
  - 默认就是贪婪模式
- 非贪婪模式
  - 就是只要满足正则规则就结束匹配。
  - 在限定符号(+ *) 后面加上？配合就表示非贪婪模式。
- 案例：
  - 字符串：12345ss3355
  - \d+ =》 12345 3355
  - \d+? => 1 2 3 4 5 3 3 5 5
  - 正则： \d+?
    - 结果： 1 2 3 4 5
  - 正则: \d+
    - 结果： 12345

JavaScript 中使用正则表达式

- 创建正则对象RegExp方式1：
  - var reg = new RegExp('\d', 'i');
  - var reg = new RegExp('\d', 'gi');
  - RegExp构造函数接受 两个参数，第一个是模式表达式，第二个是修饰符。
- 方式2：
  - var reg = /\d/i;
  - var reg = /\d/gi;　　/\d+/gi
- 关于gi说明

标志	说明
i	忽略大小写
g	全局匹配
gi	全局匹配+忽略大小写
m	多行

正则对象的属性和方法

- 属性
  - **RegExp.prototype.global 是否全局搜索。 Boolean**
  - **RegExp.prototype.ignoreCase**在匹配字符串时是否要忽略字符的大小写。
  - RegExp.prototype.lastIndex下次匹配开始的字符串索引位置，只有设置为g全局模式的时候才有用，要么设置为0或者匹配下一个的起始位置。
  - **RegExp.prototype.multiline 是否多行**
  - **RegExp.prototype.source 模式的文本,规则的字符串。**
- 方法
  - **RegExp.prototype.test(str)**
    - 方法执行一个检索，用来查看正则表达式与指定的字符串是否匹配。返回 true 或 false。
    - var str = 'hello world!'; var result = /^hello/.test(str);
  - **RegExp.prototype.exec(str)**
    - 在一个指定字符串中执行一个搜索匹配。返回一个结果数组或 null
    - 此方法执行完成后，将会自动更改正则对象中的lastIndex属性
    - 方法返回一个数组。第0个元素是匹配的字符串，其后是分组选择的数据，其中增加了一个input属性（放原始字符串），还有一个index属性：是匹配元素字符串的起始位置的索引。
    - 如果匹配上的有多个结果，那么每一次执行都会返回后一个结果，直到最后返回null

查找所有匹配

- 请实现查找所有的连续两个数字“12,34,56,78”

字符串中支持正则的方法

- str.search(exp);
  - 参数：正则对象，如果传入一个非正则表达式对象，则会隐式地使用 new RegExp(obj) 将其转换为一个 RegExp
  - 如果匹配成功，则 search() 返回正则表达式在字符串中首次匹配项的索引。否则，返回 -1
- str.match(exp)；
  - 参数正则对象，非正则对象就会隐式转换。
  - 如果正则表达式没有 g 标志，则 str.match() 会返回和 RegExp.exec() 相同的结果。而且返回的 Array 拥有一个额外的 input 属性，该属性包含被解析的原始字符串。另外，还拥有一个 index 属性，该属性表示匹配结果在原字符串中的索引（以0开始）。
  - 如果正则表达式包含 g 标志，则该方法返回一个 Array ，它包含所有匹配的子字符串而不是匹配对象。捕获组不会被返回(即不返回index属性和input属性)。如果没有匹配到，则返回 null 。
- *str*.replace(*regexp*|*substr*, *newSubStr*)
  - 参数：第一个可以是正则表达式也可以是字符串。
  - 第二个是要替换的字符串

正则案例

- 将字符串“1392945632000,mss,Date(1392945632000)”，其中绿色部分的数字转成日期对象。
- 将 字符串： 12,34,56 转成： “12”,” 34”,” 56”

- 补充：
  - string的replace方法的第二个参数，可以用特殊符号。

变量名	代表的值
$$	插入一个 "$"。
$&	插入匹配的子串。
$`	插入当前匹配的子串左边的内容。
$'	插入当前匹配的子串右边的内容。
$n	假如第一个参数是 RegExp对象，并且 n 是个小于100的非负整数，那么插入第 n 个括号匹配的字符串。

第七节：JavaScript的异常处理

- 代码时常会报错，当代码出现异常后，JS的解析引擎就会停止解析工作，停止执行后续的js代码。导致整个页面的动态效果就会出问题。
- 可以通过try catch语句进行捕获异常，并处理异常信息，使得网页能正常运行。
- 语句格式：

```
try {
    // 语句块
    // 有可能出现异常。
} catch(e) {
    // 异常处理块
} finally {     // 可以省略
    // 最后处理块
}
```

自定义抛出错误

- throw可以让开发人员抛出一个自定义的异常信息。
- throw代码执行后，程序会中断到catch块，如果没有catch则程序终止执行。
- 用例：
  - throw "Error2"; // 抛出了一个值为字符串的异常
  - throw 42; // 抛出了一个值为整数42的异常
  - throw true; // 抛出了一个值为true的异常
  - throw new Error( )

错误信息对象Error

- 抛出错误信息可以直接使用Error对象。
- 属性：

  name: 错误名
  number: 错误号
  description: 描述
  message: 错误信息,多同description
- 使用

```
new Error([message[, fileName[,lineNumber]]])
```

- 标准浏览器参数如上图所示。
- ** ie和edge浏览器：第一个参数数lineNumber，二个参数是描述和message

- 其他错误对象

  EvalError: 错误发生在eval()中
  SyntaxError: 语法错误,错误发生在eval()中,因为其它点发生SyntaxError会无法通过解释器
  RangeError: 数值超出范围
  ReferenceError: 引用不可用
  TypeError: 变量类型不是预期的
  URIError: 错误发生在encodeURI()或decodeURI()中

老马自我介绍

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

Thanks

@马伦-flydragon