# 数据类型

1. 分类

\* 基本(值)类型

\* String: 任意字符串

\* Number: 任意的数字

\* boolean: true/false

\* undefined: undefined

\* null: null

\* 对象(引用)类型

\* Object: 任意对象

\* Function: 一种特别的对象(可以执行)

\* Array: 一种特别的对象(数值下标, 内部数据是有序的)

2. 判断

\* typeof:

\* 可以判断: undefined/ 数值 / 字符串 / 布尔值 / function

\* 不能判断: null与object object与array

\* instanceof:

\* 判断对象的具体类型

\* ===

\* 可以判断: undefined, null

1. undefined与null的区别?

\* undefined代表定义未赋值

\* null定义并赋值了, 只是值为null

2. 什么时候给变量赋值为null呢?

\* 初始赋值, 表明将要赋值为对象

\* 结束前, 让对象成为垃圾对象(被垃圾回收器回收)

3. 严格区别变量类型与数据类型?

\* 数据的类型

\* 基本类型

\* 对象类型

\* 变量的类型(变量内存值的类型)

\* 基本类型: 保存就是基本类型的数据

\* 引用类型: 保存的是地址值

# 数据\_变量\_内存

1. 什么是数据?

\* 存储在内存中代表特定信息的'东东', 本质上是0101...

\* 数据的特点: 可传递, 可运算

\* 一切皆数据

\* 内存中所有操作的目标: 数据

\* 算术运算

\* 逻辑运算

\* 赋值

\* 运行函数

2. 什么是内存?

\* 内存条通电后产生的可储存数据的空间(临时的)

\* 内存产生和死亡: 内存条(电路版)==>通电==>产生内存空间==>存储数据==>处理数据==>断电==>内存空间和数据都消失

\* 一块小内存的2个数据

\* 内部存储的数据

\* 地址值

\* 内存分类

\* 栈: 全局变量/局部变量

\* 堆: 对象

3. 什么是变量?

\* 可变化的量, 由变量名和变量值组成

\* 每个变量都对应的一块小内存, 变量名用来查找对应的内存, 变量值就是内存中保存的数据

4. 内存,数据, 变量三者之间的关系

\* 内存用来存储数据的空间

\* 变量是内存的标识

问题: var a = xxx, a内存中到底保存的是什么?

\* xxx是基本数据, 保存的就是这个数据

\* xxx是对象, 保存的是对象的地址值

\* xxx是一个变量, 保存的xxx的内存内容(可能是基本数据, 也可能是地址值)

问题: 在js调用函数时传递变量参数时, 是值传递还是引用传递

\* 理解1: 都是值(基本/地址值)传递

\* 理解2: 可能是值传递, 也可能是引用传递(地址值)

问题: JS引擎如何管理内存?

1. 内存生命周期

\* 分配小内存空间, 得到它的使用权

\* 存储数据, 可以反复进行操作

\* 释放小内存空间

2. 释放内存

\* 局部变量: 函数执行完自动释放

\* 对象: 成为垃圾对象==>垃圾回收器回收

# 对象

1. 什么是对象?

\* 多个数据的封装体

\* 用来保存多个数据的容器

\* 一个对象代表现实中的一个事物

2. 为什么要用对象?

\* 统一管理多个数据

3. 对象的组成

\* 属性: 属性名(字符串)和属性值(任意)组成

\* 方法: 一种特别的属性(属性值是函数)

4. 如何访问对象内部数据?

\* .属性名: 编码简单, 有时不能用

\* ['属性名']: 编码麻烦, 能通用

问题: 什么时候必须使用['属性名']的方式?

1. 属性名包含特殊字符: - 空格

2. 属性名不确定

# 函数

1. 什么是函数?

\* 实现特定功能的n条语句的封装体

\* 只有函数是可以执行的, 其它类型的数据不能执行

2. 为什么要用函数?

\* 提高代码复用

\* 便于阅读交流

3. 如何定义函数?

\* 函数声明

\* 表达式

4. 如何调用(执行)函数?

\* test(): 直接调用

\* obj.test(): 通过对象调用

\* new test(): new调用

\* test.call/apply(obj): 临时让test成为obj的方法进行调用

1. 什么函数才是回调函数?

1). 你定义的

2). 你没有调

3). 但最终它执行了(在某个时刻或某个条件下)

2. 常见的回调函数?

\* dom事件回调函数 ==>发生事件的dom元素

\* 定时器回调函数 ===>window

\* ajax请求回调函数(后面讲)

\* 生命周期回调函数(后面讲)

匿名函数

1. 理解

\* 全称: Immediately-Invoked Function Expression

2. 作用

\* 隐藏实现

\* 不会污染外部(全局)命名空间

\* 用它来编码js模块

# 原型

1. 函数的prototype属性(图)

\* 每个函数都有一个prototype属性, 它默认指向一个Object空对象(即称为: 原型对象)

\* 原型对象中有一个属性constructor, 它指向函数对象

2. 给原型对象添加属性(一般都是方法)

\* 作用: 函数的所有实例对象自动拥有原型中的属性(方法)

// 每个函数都有一个prototype属性, 它默认指向一个Object空对象(即称为: 原型对象)

console.log(Date.prototype, typeof Date.prototype)

function Fun () {//alt + shift +r(重命名rename)

}

console.log(Fun.prototype) // 默认指向一个Object空对象(没有我们的属性)

// 原型对象中有一个属性constructor, 它指向函数对象

console.log(Date.prototype.constructor===Date)

console.log(Fun.prototype.constructor===Fun)

//给原型对象添加属性(一般是方法) ===>实例对象可以访问

Fun.prototype.test = function () {

console.log('test()')

}

var fun = new Fun()

fun.test()

## 显示原型和隐式原型

1. 每个函数function都有一个prototype，即显式原型(属性)

2. 每个实例对象都有一个\_\_proto\_\_，可称为隐式原型(属性)

3. 对象的隐式原型的值为其对应构造函数的显式原型的值

4. 内存结构(图)

5. 总结:

\* 函数的prototype属性: 在定义函数时自动添加的, 默认值是一个空Object对象

\* 对象的\_\_proto\_\_属性: 创建对象时自动添加的, 默认值为构造函数的prototype属性值

\* 程序员能直接操作显式原型, 但不能直接操作隐式原型(ES6之前)

//定义构造函数

function Fn() { // 内部语句: this.prototype = {}

}

// 1. 每个函数function都有一个prototype，即显式原型属性, 默认指向一个空的Object对象

console.log(Fn.prototype)

// 2. 每个实例对象都有一个\_\_proto\_\_，可称为隐式原型

//创建实例对象

var fn = new Fn() // 内部语句: this.\_\_proto\_\_ = Fn.prototype

console.log(fn.\_\_proto\_\_)

// 3. 对象的隐式原型的值为其对应构造函数的显式原型的值

console.log(Fn.prototype===fn.\_\_proto\_\_) // true

//给原型添加方法

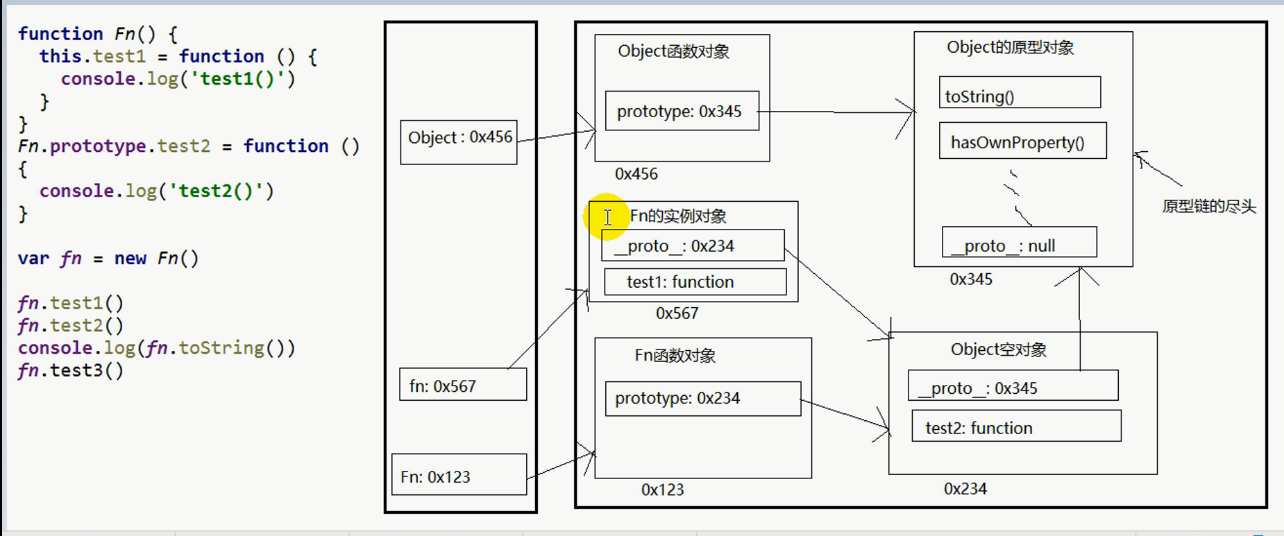
Fn.prototype.test = function () {

console.log('test()')

}

//通过实例调用原型的方法

fn.test()



## 原型链

1. 原型链(图解)

\* 访问一个对象的属性时，

\* 先在自身属性中查找，找到返回

\* 如果没有, 再沿着\_\_proto\_\_这条链向上查找, 找到返回

\* 如果最终没找到, 返回undefined

\* 别名: 隐式原型链

\* 作用: 查找对象的属性(方法)

2. 构造函数/原型/实体对象的关系(图解)

3. 构造函数/原型/实体对象的关系2(图解)

// console.log(Object)

//console.log(Object.prototype)

console.log(Object.prototype.\_\_proto\_\_) // null

function Fn() {

this.test1 = function () {

console.log('test1()')

}

}

console.log(Fn.prototype)

Fn.prototype.test2 = function () {

console.log('test2()')

}

var fn = new Fn()

fn.test1()

fn.test2()

console.log(fn.toString())

console.log(fn.test3)

// fn.test3()

/\*

1. 函数的显示原型指向的对象默认是空Object实例对象(但Object不满足)

\*/

console.log(Fn.prototype instanceof Object) // true

console.log(Object.prototype instanceof Object) // false

console.log(Function.prototype instanceof Object) // true

/\*

2. 所有函数都是Function的实例(包含Function)

\*/

console.log(Function.\_\_proto\_\_===Function.prototype) // true

/\*

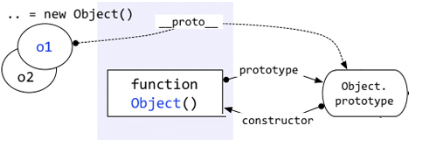
3. Object的原型对象是原型链尽头

\*/

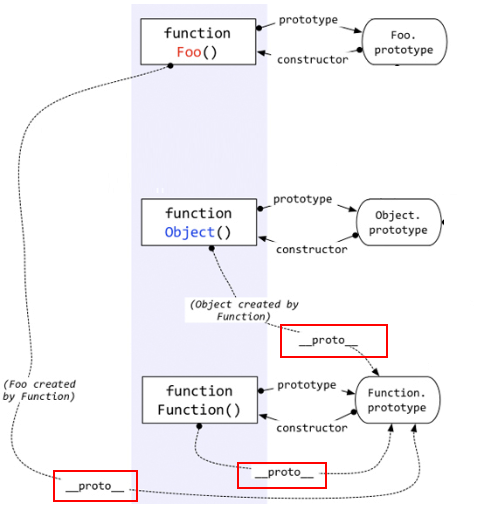
console.log(Object.prototype.\_\_proto\_\_) // null

var o1 = new Object();

var o2 = {};



function Foo(){ }



## 原型链属性问题

1. 读取对象的属性值时: 会自动到原型链中查找

2. 设置对象的属性值时: 不会查找原型链, 如果当前对象中没有此属性, 直接添加此属性并设置其值

3. 方法一般定义在原型中, 属性一般通过构造函数定义在对象本身上

## 探索instanceof

1. instanceof是如何判断的?

\* 表达式: A instanceof B

\* 如果B函数的显式原型对象在A对象的原型链上, 返回true, 否则返回false

2. Function是通过new自己产生的实例

/\*

案例1

\*/

function Foo() { }

var f1 = new Foo()

console.log(f1 instanceof Foo) // true

console.log(f1 instanceof Object) // true

/\*

案例2

\*/

console.log(Object instanceof Function) // true

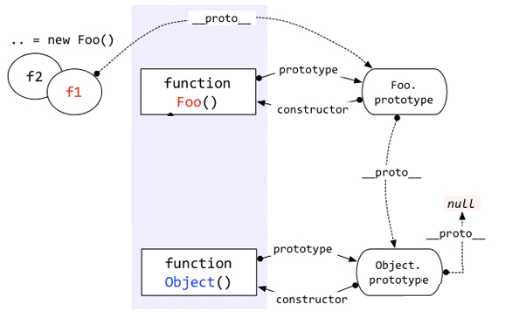
console.log(Object instanceof Object) // true

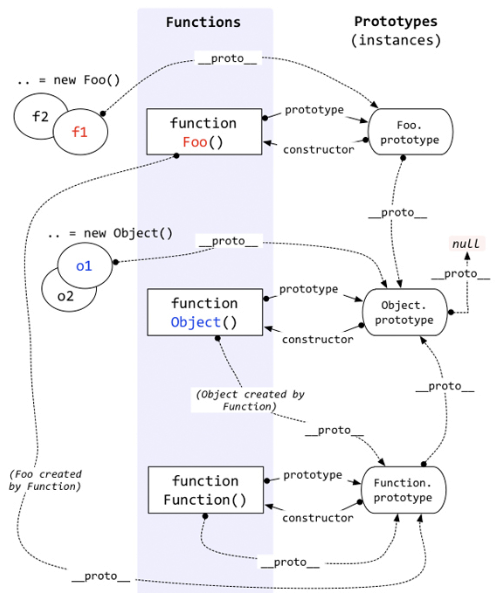
console.log(Function instanceof Function) // true

console.log(Function instanceof Object) // true

function Foo() {}

console.log(Object instanceof Foo) // false





/\*

测试题1

\*/

function A () {

}

A.prototype.n = 1

var b = new A()

A.prototype = {

n: 2,

m: 3

}

var c = new A()

console.log(b.n, b.m, c.n, c.m)

/\*

测试题2

\*/

function F (){}

Object.prototype.a = function(){

console.log('a()')

}

Function.prototype.b = function(){

console.log('b()')

}

var f = new F()

f.a()

// f.b()

F.a()

F.b()

console.log(f)

console.log(Object.prototype)

console.log(Function.prototype)

# 执行上下文与执行上下文栈

## 变量提升与函数提升

1. 变量声明提升

\* 通过var定义(声明)的变量, 在定义语句之前就可以访问到

\* 值: undefined

2. 函数声明提升

\* 通过function声明的函数, 在之前就可以直接调用

\* 值: 函数定义(对象)

3. 问题: 变量提升和函数提升是如何产生的?

/\*

面试题 : 输出 undefined

\*/

var a = 3

function fn () {

console.log(a)

var a = 4

}

fn()

console.log(b) //undefined 变量提升

fn2() //可调用 函数提升

// fn3() //不能 变量提升

var b = 3

function fn2() {

console.log('fn2()')

}

var fn3 = function () {

console.log('fn3()')

}

## 执行上下文

1. 代码分类(位置)

\* 全局代码

\* 函数(局部)代码

2. 全局执行上下文

\* 在执行全局代码前将window确定为全局执行上下文

\* 对全局数据进行预处理

\* var定义的全局变量==>undefined, 添加为window的属性

\* function声明的全局函数==>赋值(fun), 添加为window的方法

\* this==>赋值(window)

\* 开始执行全局代码

3. 函数执行上下文

\* 在调用函数, 准备执行函数体之前, 创建对应的函数执行上下文对象(虚拟的, 存在于栈中)

\* 对局部数据进行预处理

\* 形参变量==>赋值(实参)==>添加为执行上下文的属性

\* arguments==>赋值(实参列表), 添加为执行上下文的属性

\* var定义的局部变量==>undefined, 添加为执行上下文的属性

\* function声明的函数 ==>赋值(fun), 添加为执行上下文的方法

\* this==>赋值(调用函数的对象)

\* 开始执行函数体代码

## 执行上下文栈

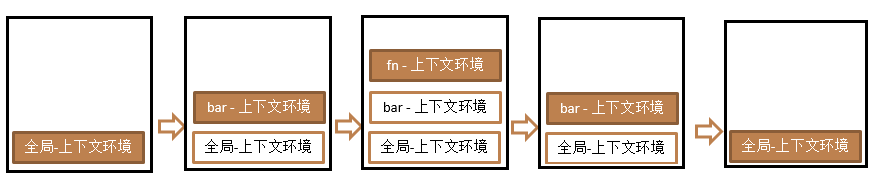
1. 在全局代码执行前, JS引擎就会创建一个栈来存储管理所有的执行上下文对象

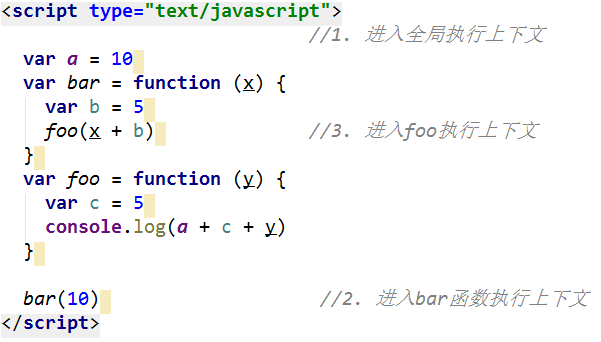
2. 在全局执行上下文(window)确定后, 将其添加到栈中(压栈)

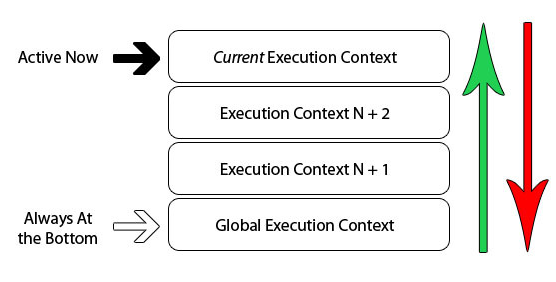
3. 在函数执行上下文创建后, 将其添加到栈中(压栈)

4. 在当前函数执行完后,将栈顶的对象移除(出栈)

5. 当所有的代码执行完后, 栈中只剩下window







/\*

测试题1: 先执行变量提升, 再执行函数提升

\*/

function a() {}

var a

console.log(typeof a) // 'function'

/\*

测试题2:

\*/

if (!(b in window)) {

var b = 1

}

console.log(b) // undefined

/\*

测试题3:

\*/

var c = 1

function c(c) {

console.log(c)

var c = 3

}

c(2) // 报错

# 作用域和作用域链

## 作用域

1. 理解

\* 就是一块"地盘", 一个代码段所在的区域

\* 它是静态的(相对于上下文对象), 在编写代码时就确定了

2. 分类

\* 全局作用域

\* 函数作用域

\* 没有块作用域(ES6有了)

3. 作用

\* 隔离变量，不同作用域下同名变量不会有冲突

## 作用域与执行上下文

1. 区别1

\* 全局作用域之外，每个函数都会创建自己的作用域，作用域在函数定义时就已经确定了。而不是在函数调用时

\* 全局执行上下文环境是在全局作用域确定之后, js代码马上执行之前创建

\* 函数执行上下文是在调用函数时, 函数体代码执行之前创建

2. 区别2

\* 作用域是静态的, 只要函数定义好了就一直存在, 且不会再变化

\* 执行上下文是动态的, 调用函数时创建, 函数调用结束时就会自动释放

3. 联系

\* 执行上下文(对象)是从属于所在的作用域

\* 全局上下文环境==>全局作用域

\* 函数上下文环境==>对应的函数使用域

## 作用域链

1. 理解

\* 多个上下级关系的作用域形成的链, 它的方向是从下向上的(从内到外)

\* 查找变量时就是沿着作用域链来查找的

2. 查找一个变量的查找规则

\* 在当前作用域下的执行上下文中查找对应的属性, 如果有直接返回, 否则进入2

\* 在上一级作用域的执行上下文中查找对应的属性, 如果有直接返回, 否则进入3

\* 再次执行2的相同操作, 直到全局作用域, 如果还找不到就抛出找不到的异常

## 作用域\_面试题

var x = 10;

function fn() { // 作用域在定义时就绑定了，绑定在全局作用域内

console.log(x); // 在fn中查找，找不到，于是到全局作用域中找

}

function show(f) {

var x = 20;

f();

}

show(fn); // 10

-----------------------------------------------------------

var fn = function () {

console.log(fn)

}

fn() // 输出 fn 函数

var obj = {

fn2: function () {

console.log(fn2) // 在fn2 函数内找不到，又到全局作用域找，还是找不到

//console.log(this.fn2)

}

}

obj.fn2() // 报错

# 闭包

## 引入

需求: 点击某个按钮, 提示"点击的是第n个按钮"

//遍历加监听

/\*

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

var btn = btns[i]

btn.onclick = function () {

alert('第'+(i+1)+'个')

}

}\*/

/\*

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

var btn = btns[i]

//将btn所对应的下标保存在btn上

btn.index = i

btn.onclick = function () {

alert('第'+(this.index+1)+'个')

}

}\*/

//利用闭包

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

(function (j) {

var btn = btns[j]

btn.onclick = function () {

alert('第'+(j+1)+'个')

}

})(i)

}

## 理解闭包

1. 如何产生闭包?

\* 当一个嵌套的内部(子)函数引用了嵌套的外部(父)函数的变量(函数)时, 就产生了闭包

2. 闭包到底是什么?

\* 使用chrome调试查看

\* 理解一: 闭包是嵌套的内部函数(绝大部分人)

\* 理解二: 包含被引用变量(函数)的对象(极少数人)

\* 注意: 闭包存在于嵌套的内部函数中

3. 产生闭包的条件?

\* 函数嵌套

\* 内部函数引用了外部函数的数据(变量/函数)

function fn1 () {

var a = 2

var b = 'abc'

function fn2 () { //执行函数定义就会产生闭包(不用调用内部函数)

console.log(a)

}

// fn2()

}

fn1()

function fun1() {

var a = 3

var fun2 = function () {

console.log(a)

}

}

fun1()

## 常见的闭包

1. 将函数作为另一个函数的返回值

2. 将函数作为实参传递给另一个函数调用

// 1. 将函数作为另一个函数的返回值

function fn1() {

var a = 2

function fn2() {

a++

console.log(a)

}

return fn2

}

var f = fn1()

f() // 3

f() // 4

// 2. 将函数作为实参传递给另一个函数调用

function showDelay(msg, time) {

setTimeout(function () {

alert(msg)

}, time)

}

showDelay('atguigu', 2000)

## 闭包的作用

1. 使用函数内部的变量在函数执行完后, 仍然存活在内存中(延长了局部变量的生命周期)

2. 让函数外部可以操作(读写)到函数内部的数据(变量/函数)

问题:

1. 函数执行完后, 函数内部声明的局部变量是否还存在? 一般是不存在, 存在于闭中的变量才可能存在

2. 在函数外部能直接访问函数内部的局部变量吗? 不能, 但我们可以通过闭包让外部操作它

function fn1() {

var a = 2

function fn2() {

a++

console.log(a)

// return a

}

function fn3() {

a--

console.log(a)

}

return fn3

}

var f = fn1()

f() // 1

f() // 0

## 闭包的生命周期

1. 产生: 在嵌套内部函数定义执行完时就产生了(不是在调用)

2. 死亡: 在嵌套的内部函数成为垃圾对象时

function fn1() {

//此时闭包就已经产生了(函数提升, 内部函数对象已经创建了)

var a = 2

function fn2 () {

a++

console.log(a)

}

return fn2

}

var f = fn1()

f() // 3

f() // 4

f = null //闭包死亡(包含闭包的函数对象成为垃圾对象)

## 闭包的应用\_自定义JS模块

闭包的应用2 : 定义JS模块

\* 具有特定功能的js文件

\* 将所有的数据和功能都封装在一个函数内部(私有的)

\* 只向外暴露一个包信n个方法的对象或函数

\* 模块的使用者, 只需要通过模块暴露的对象调用方法来实现对应的功能

function myModule() {

//私有数据

var msg = 'My atguigu'

//操作数据的函数

function doSomething() {

console.log('doSomething() '+msg.toUpperCase())

}

function doOtherthing () {

console.log('doOtherthing() '+msg.toLowerCase())

}

//向外暴露对象(给外部使用的方法)

return {

doSomething: doSomething,

doOtherthing: doOtherthing

}

}

var module = myModule()

module.doSomething()

module.doOtherthing()

(function () {

//私有数据

var msg = 'My atguigu'

//操作数据的函数

function doSomething() {

console.log('doSomething() '+msg.toUpperCase())

}

function doOtherthing () {

console.log('doOtherthing() '+msg.toLowerCase())

}

//向外暴露对象(给外部使用的方法)

window.myModule2 = {

doSomething: doSomething,

doOtherthing: doOtherthing

}

})()

myModule2.doSomething()

myModule2.doOtherthing()

## 闭包的缺点及解决

1. 缺点

\* 函数执行完后, 函数内的局部变量没有释放, 占用内存时间会变长

\* 容易造成内存泄露

2. 解决

\* 能不用闭包就不用

\* 及时释放

function fn1() {

var arr = new Array[100000]

function fn2() {

console.log(arr.length)

}

return fn2

}

var f = fn1()

f()

f = null //让内部函数成为垃圾对象-->回收闭包

## 面试题

//代码片段一

var name = "The Window";

var object = {

name : "My Object",

getNameFunc : function(){

return function(){

return this.name;

};

}

};

alert(object.getNameFunc()()); //? the window

//代码片段二

var name2 = "The Window";

var object2 = {

name2 : "My Object",

getNameFunc : function(){

var that = this;

return function(){

return that.name2;

};

}

};

alert(object2.getNameFunc()()); //? my object

//代码片段三

function fun(n,o) {

console.log(o)

return {

fun:function(m){

return fun(m,n)

}

}

}

var a = fun(0)

a.fun(1)

a.fun(2)

a.fun(3)//undefined,0,0,0

var b = fun(0).fun(1).fun(2).fun(3)//undefined,0,1,2

var c = fun(0).fun(1)

c.fun(2)

c.fun(3)//undefined,0,1,1

# 对象创建模式

## Object构造函数模式

方式一: Object构造函数模式

\* 套路: 先创建空Object对象, 再动态添加属性/方法

\* 适用场景: 起始时不确定对象内部数据

\* 问题: 语句太多

/\*

一个人: name:"Tom", age: 12

\*/

// 先创建空Object对象

var p = new Object()

p = {} //此时内部数据是不确定的

// 再动态添加属性/方法

p.name = 'Tom'

p.age = 12

p.setName = function (name) {

this.name = name

}

//测试

console.log(p.name, p.age)

p.setName('Bob')

console.log(p.name, p.age)

## 对象字面量

方式二: 对象字面量模式

\* 套路: 使用{}创建对象, 同时指定属性/方法

\* 适用场景: 起始时对象内部数据是确定的

\* 问题: 如果创建多个对象, 有重复代码

var p = {

name: 'Tom',

age: 12,

setName: function (name) {

this.name = name

}

}

//测试

console.log(p.name, p.age)

p.setName('JACK')

console.log(p.name, p.age)

var p2 = { //如果创建多个对象代码很重复

name: 'Bob',

age: 13,

setName: function (name) {

this.name = name

}

}

## 工厂模式

方式三: 工厂模式

\* 套路: 通过工厂函数动态创建对象并返回

\* 适用场景: 需要创建多个对象

\* 问题: 对象没有一个具体的类型, 都是Object类型

function createPerson(name, age) { //返回一个对象的函数===>工厂函数

var obj = {

name: name,

age: age,

setName: function (name) {

this.name = name

}

}

return obj

}

// 创建2个人

var p1 = createPerson('Tom', 12)

var p2 = createPerson('Bob', 13)

// p1/p2是Object类型

function createStudent(name, price) {

var obj = {

name: name,

price: price

}

return obj

}

var s = createStudent('张三', 12000)

// s也是Object

## 自定义构造函数模式

方式四: 自定义构造函数模式

\* 套路: 自定义构造函数, 通过new创建对象

\* 适用场景: 需要创建多个类型确定的对象

\* 问题: 每个对象都有相同的数据, 浪费内存

//定义类型

function Person(name, age) {

this.name = name

this.age = age

this.setName = function (name) {

this.name = name

}

}

var p1 = new Person('Tom', 12)

p1.setName('Jack')

console.log(p1.name, p1.age)

console.log(p1 instanceof Person)

function Student (name, price) {

this.name = name

this.price = price

}

var s = new Student('Bob', 13000)

console.log(s instanceof Student)

var p2 = new Person('JACK', 23)

console.log(p1, p2)

## 构造函数+原型的组合模式

方式六: 构造函数+原型的组合模式

\* 套路: 自定义构造函数, 属性在函数中初始化, 方法添加到原型上

\* 适用场景: 需要创建多个类型确定的对象

function Person(name, age) { //在构造函数中只初始化一般函数

this.name = name

this.age = age

}

Person.prototype.setName = function (name) {

this.name = name

}

var p1 = new Person('Tom', 23)

var p2 = new Person('Jack', 24)

console.log(p1, p2)

# 继承模式

## 原型链继承

方式1: 原型链继承

1. 套路

1. 定义父类型构造函数

2. 给父类型的原型添加方法

3. 定义子类型的构造函数

4. 创建父类型的对象赋值给子类型的原型

5. 将子类型原型的构造属性设置为子类型

6. 给子类型原型添加方法

7. 创建子类型的对象: 可以调用父类型的方法

2. 关键

1. 子类型的原型为父类型的一个实例对象

//父类型

function Supper() {

this.supProp = 'Supper property'

}

Supper.prototype.showSupperProp = function () {

console.log(this.supProp)

}

//子类型

function Sub() {

this.subProp = 'Sub property'

}

// 子类型的原型为父类型的一个实例对象

Sub.prototype = new Supper()

// 让子类型的原型的constructor指向子类型

Sub.prototype.constructor = Sub

Sub.prototype.showSubProp = function () {

console.log(this.subProp)

}

var sub = new Sub()

sub.showSupperProp()

// sub.toString()

sub.showSubProp()

console.log(sub) // Sub

## 借用构造函数继承

方式2: 借用构造函数继承(假的)

1. 套路:

1. 定义父类型构造函数

2. 定义子类型构造函数

3. 在子类型构造函数中调用父类型构造

2. 关键:

1. 在子类型构造函数中通用call()调用父类型构造函数

function Person(name, age) {

this.name = name

this.age = age

}

function Student(name, age, price) {

Person.call(this, name, age) // 相当于: this.Person(name, age)

/\*this.name = name

this.age = age\*/

this.price = price

}

var s = new Student('Tom', 20, 14000)

console.log(s.name, s.age, s.price)

## 组合继承

方式3: 原型链+借用构造函数的组合继承

1. 利用原型链实现对父类型对象的方法继承

2. 利用super()借用父类型构建函数初始化相同属性

function Person(name, age) {

this.name = name

this.age = age

}

Person.prototype.setName = function (name) {

this.name = name

}

function Student(name, age, price) {

Person.call(this, name, age) // 为了得到属性

this.price = price

}

Student.prototype = new Person() // 为了能看到父类型的方法

Student.prototype.constructor = Student //修正constructor属性

Student.prototype.setPrice = function (price) {

this.price = price

}

var s = new Student('Tom', 24, 15000)

s.setName('Bob')

s.setPrice(16000)

console.log(s.name, s.age, s.price)

# 线程机制与事件机制

## 进程与线程

1. 进程：程序的一次执行, 它占有一片独有的内存空间

2. 线程： CPU的基本调度单位, 是程序执行的一个完整流程

3. 进程与线程

\* 一个进程中一般至少有一个运行的线程: 主线程

\* 一个进程中也可以同时运行多个线程, 我们会说程序是多线程运行的

\* 一个进程内的数据可以供其中的多个线程直接共享

\* 多个进程之间的数据是不能直接共享的

4. 浏览器运行是单进程还是多进程?

\* 有的是单进程

\* firefox

\* 老版IE

\* 有的是多进程

\* chrome

\* 新版IE

5. 如何查看浏览器是否是多进程运行的呢?

\* 任务管理器==>进程

6. 浏览器运行是单线程还是多线程?

\* 都是多线程运行的

## 浏览器内核

1. 什么是浏览器内核?

\* 支持浏览器运行的最核心的程序

2. 不同的浏览器可能不太一样

\* Chrome, Safari: webkit

\* firefox: Gecko

\* IE: Trident

\* 360,搜狗等国内浏览器: Trident + webkit

3. 内核由很多模块组成

\* html,css文档解析模块 : 负责页面文本的解析

\* dom/css模块 : 负责dom/css在内存中的相关处理

\* 布局和渲染模块 : 负责页面的布局和效果的绘制

\* 布局和渲染模块 : 负责页面的布局和效果的绘制

\* 定时器模块 : 负责定时器的管理

\* 网络请求模块 : 负责服务器请求(常规/Ajax)

\* 事件响应模块 : 负责事件的管理

## 定时器引发的思考

1. 定时器真是定时执行的吗?

\* 定时器并不能保证真正定时执行

\* 一般会延迟一丁点(可以接受), 也有可能延迟很长时间(不能接受)

2. 定时器回调函数是在分线程执行的吗?

\* 在主线程执行的, js是单线程的

3. 定时器是如何实现的?

\* 事件循环模型(后面讲)

document.getElementById('btn').onclick = function () {

var start = Date.now()

console.log('启动定时器前...')

setTimeout(function () {

console.log('定时器执行了', Date.now()-start)

}, 200)

console.log('启动定时器后...')

// 做一个长时间的工作

for (var i = 0; i < 1000000000; i++) {

}

## JS是单线程的

1. 如何证明js执行是单线程的?

\* setTimeout()的回调函数是在主线程执行的

\* 定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行

2. 为什么js要用单线程模式, 而不用多线程模式?

\* JavaScript的单线程，与它的用途有关。

\* 作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。

\* 这决定了它只能是单线程，否则会带来很复杂的同步问题

3. 代码的分类:

\* 初始化代码

\* 回调代码

4. js引擎执行代码的基本流程

\* 先执行初始化代码: 包含一些特别的代码 回调函数(异步执行)

\* 设置定时器

\* 绑定事件监听

\* 发送ajax请求

\* 后面在某个时刻才会执行回调代码

## 事件循环模型

1. 所有代码分类

\* 初始化执行代码(同步代码): 包含绑定dom事件监听, 设置定时器, 发送ajax请求的代码

\* 回调执行代码(异步代码): 处理回调逻辑

2. js引擎执行代码的基本流程:

\* 初始化代码===>回调代码

3. 模型的2个重要组成部分:

\* 事件(定时器/DOM事件/Ajax)管理模块

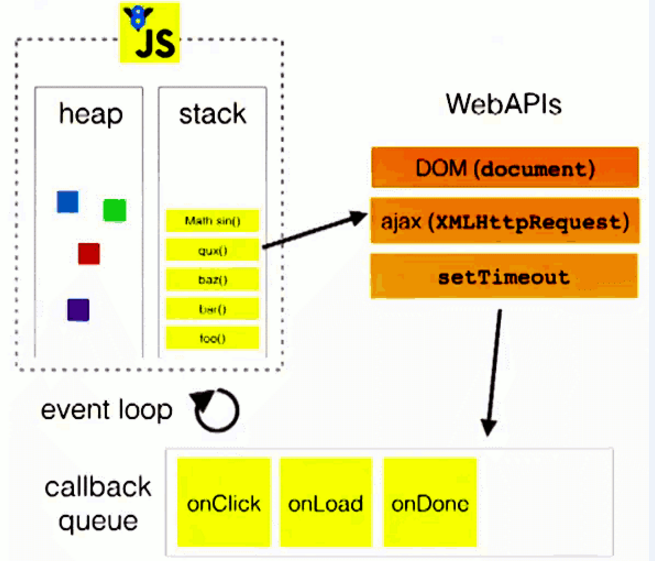
\* 回调队列

4. 模型的运转流程

\* 执行初始化代码, 将事件回调函数交给对应模块管理

\* 当事件发生时, 管理模块会将回调函数及其数据添加到回调列队中

\* 只有当初始化代码执行完后(可能要一定时间), 才会遍历读取回调队列中的回调函数执行



## Web Workers

1. H5规范提供了js分线程的实现, 取名为: Web Workers

2. 相关API

\* Worker: 构造函数, 加载分线程执行的js文件

\* Worker.prototype.onmessage: 用于接收另一个线程的回调函数

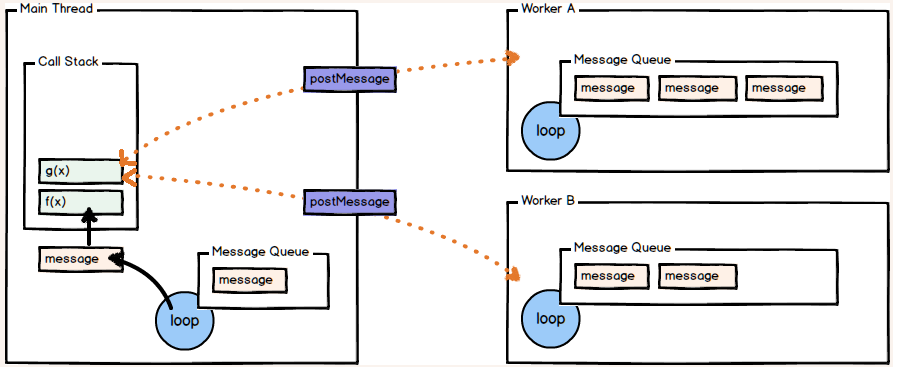
\* Worker.prototype.postMessage: 向另一个线程发送消息

3. 不足

\* worker内代码不能操作DOM(更新UI)

\* 不能跨域加载JS

\* 不是每个浏览器都支持这个新特性



var input = document.getElementById('number')

document.getElementById('btn').onclick = function () {

var number = input.value

//创建一个Worker对象

var worker = new Worker('worker.js')

// 绑定接收消息的监听

worker.onmessage = function (event) {

console.log('主线程接收分线程返回的数据: '+event.data)

alert(event.data)

}

// 向分线程发送消息

worker.postMessage(number)

console.log('主线程向分线程发送数据: '+number)

}

// console.log(this) // window

Worker.js

function fibonacci(n) {

return n<=2 ? 1 : fibonacci(n-1) + fibonacci(n-2) //递归调用

}

console.log(this)

this.onmessage = function (event) {

var number = event.data

console.log('分线程接收到主线程发送的数据: '+number)

//计算

var result = fibonacci(number)

postMessage(result)

console.log('分线程向主线程返回数据: '+result)

// alert(result) alert是window的方法, 在分线程不能调用

// 分线程中的全局对象不再是window, 所以在分线程中不可能更新界面

}