# **\*\*\*\*\*\*\*\*基础总结\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# **1.1数据类型及判断**

***1. js中数据类型的分类(2大类)  
 \* 基本(值)类型：  
 \* Number: 任意数值 1  
 \* String: 任意文本 '1' "1"  
 \* Boolean: true/false  
 \* undefined: undefined  
 \* null: null  
   
 \* 对象(引用)类型：  
 \* Object: 任意对象（一般对象） {}  
 \* Array: 一种特别的对象类型(1.可以通过下标取值。2.内部数据有序)  
 \* Function: 一种特别的对象类型(可执行)  
 备注：准确来说Array和Function属于一种特别的Object类型。  
   
2. 判断：  
 \* typeof：注意typeof的返回值是类型的字符串名称（注意：首字母是小写的）  
 \* 可以区别: 数值, 字符串, 布尔值, undefined, function  
 \* 不能区别: null与对象  
 \* 对于typeof：  
 \* 一个不正确：typeof null的结果是object  
 \* 一个不精确：typeof []的结果是object  
 \* 备注：typeof {} 的返回值是object  
 \* 以上三种数据类型两两组合都不能用typeof区分  
   
 \* instanceof：  
 \* 专门用来判断对象数据的具体类型: Object, Array与Function  
 \* A instanceof B ==> 判断A是否是B这个类型的实例对象(B是构造函数名)***

***\* ===  
 \* 可以判断: undefined和null  
 \* 备注：因为undefined类型只有一个值、null类型也只有一个值,所以可以用===判断***

***/\*基本数据类型的判断\*/*var *a*console.log(*a*,typeof *a* === 'undefined',*a* === undefined,*a*==='undefined')  
*//undefined true true false  
//判断一个数据是否为undefined可以使用：typeof a === 'undefined',a === undefined  
a* = 3  
console.log(typeof *a* === 'number')*//true  
a* = 'atguigu'  
console.log(typeof *a* === 'string')*//true  
a* = true  
console.log(typeof *a* === 'boolean')*//true  
a* = null  
console.log(typeof *a*)*//object  
//注意这特别的,typeof null返回的是'object',由于js底层设计“缺陷”造成的  
  
/\*对象类型的判断\*/*var *b1* = {  
 b2:[2,'atguigu',console.log],  
 b3:function () {  
 console.log('b3')  
 }  
}  
console.log(*b1*) *//{b2: Array(3), b3: ƒ}*console.log(*b1*.b2) *//(3) [2, "atguigu", ƒ]*console.log(*b1*.b3) *// ƒ () {console.log('b3')}*console.log(*b1* instanceof Object,typeof *b1*) *//true "object"*console.log(*b1*.b2 instanceof Array,typeof *b1*.b2)*//true "object"*console.log(*b1*.b3 instanceof Function,typeof *b1*.b3)**

***//true "function"  
//扩展*console.log(typeof *b1*.b2[2]) *//'function'*console.log(typeof *b1*.b2[2]('atguigu')) *//'atguigu' 'undefined'  
 //console.log的返回值是undefined*function *demo*(a) {  
 console.log(a)  
}  
console.log(*demo*('atguigu'))  
*//'atguigu' undefined 注:这里的undefined是值,不是字符串,因为demo函数未定义return,所以他的返回值是undefined,是undefined类型的数据***

# **1.2数据类型的3个问题**

***1. undefined与null的区别?  
 \* undefined代表没有赋值  
 \* null是定义了，且赋值了，值为null  
2. 什么时候给变量赋值为null呢?  
 \* var a = null //开始时候赋值为null,意味着以后这个a将保存一个对象, 但对象此时还没有确定  
 \* a = null //让a指向的对象成为垃圾对象,从而被回收  
3. 严格区别【变量类型】与【数据类型】  
 \* js的变量本身是没有类型的, 程序员所说的变量的类型实际上是变量内存中数据的类型  
 \* 数据类型*  
  *\* 基本类型  
 \* 对象类型  
 \* 变量类型  
 \* 基本类型: 保存着基本类型数据的变量  
 \* 引用类型: 保存着对象地址值的变量***

# **2.数据-变量-内存的关系**

***1. 什么是数据?  
 \* 存储在内存中代表特定信息的‘东东’，本质是二进制数据：010101.....  
 \* 万物皆数据***

***2. 什么是内存?  
 \* 内存条通电后产生的存储空间（临时的）  
 \* 产生：内存条==>通电===>产生的存储空间  
 \* 死亡：断电  
 \* 硬盘中的数据是永久的  
 \* 一块小内存由2个部分组成：***

***\* 1.该块内存的地址；2.该块内存所存储的数据  
 \* 内存的分类：  
 \* 栈内存（相对小）  
 \* 堆内存（相对大）  
3. 什么是变量?  
 \* 值可以变化的量,一个变量：1.变量名；2.变量值  
 \* 每一个变量都对应一块小内存，变量名用来查找存储内容的那块内存，变量值就是内存中所保存的内容  
 \* 与之相对立的一个叫做“常量”  
4. 内存,数据, 变量三者之间的关系  
 \* 内存是一个容器，用来保存数据  
 \* 变量是内存的标识，我们通过变量的名找到对应的内存，进而进行操作（读、写）***

## **2.1问题一**

***这行语句：var a = xxx, a对应的那块内存中到底保存的是什么?  
\* xxx是一个基本数据  
\* xxx是一个对象  
\* xxx是一个变量***

**var *a* = 3 *//保存的就是基本类型的数据*var *b* = {} *//保存的是对象对应堆内存中的内存地址***

**var *c* = *a //保存的是变量a里的数据***

## **2.2关于引用变量赋值问题**

***1). 两个引用变量指向同一个对象, 通过一个引用变量修改对象内部数据, 另一个引用变量也看得见***

**var *o1* = {m:1} *//o1里存储的是对象在堆内存中的地址* var *o2* = *o1 //这里是把o1里存储的地址赋值给了o2,o1和o2同时指 向同一个对象***

***o1*.m = 2  
 console.log(*o2*,*o1* === *o2*) *//{m: 2} true  
 2). 两个引用变量指向同一个对象,让一个引用变量指向另一个对象, 另一个引用变量还是指向原来的对象* var *o3* = {m:1}  
 var *o4* = *o3  
 o3* = {m:4} *//o3指向另一个对象在堆内存中的地址* console.log(*o4*.m,*o4* === *o3*)*//1 false o3和o4存储的地址都同***

***//扩展* var *o5* = {m:5}  
 var *o6* = *o5* var *demo* = function (o) {  
 *//o.m = 6 //控制台输出 6 6 true o和o5指向同一个对象* o = {m:7} *//控制台输出 5 5 true o指向另一个对象* }  
 *demo*(*o5*) *//这里只是把o5所存储的地址传过去了* console.log(*o5*.m,*o6*.m,*o5*===*o6*)**

## **2.3问题二**

***在js【调用函数】传递变量参数时, 是值传递？还是引用传递？  
 对于函数传参，只有值传递, 没有引用传递, 传递的都是变量的值,  
 只是这个值可能是基本数据, 也可能是地址(引用)数据***

**function *demo*(a) {  
 console.log(a)  
 }  
 *demo*(1) *//1* var *obj* = {name:'kobe'}  
 *demo*(*obj*) *//{name: "kobe"} 将demo所储存的对象的地址传过去了***

## **2.4问题三**

***JS引擎如何管理内存?  
 1. 内存生命周期  
 1). 分配需要的内存: 定义变量 / 创建对象  
 2). 使用(存/取)分配到的内存: 赋值 / 创建对象  
 3). 不需要时将其释放/归还  
 2. 释放内存  
 \* 为执行函数分配的栈空间内存: 函数执行完自动释放  
 \* 存储对象的堆空间内存: 当内存没有引用指向时, 对象成为垃圾对象, 垃圾回收器后面就会回收释放此内存***

***/\*全局代码\*/  
//定义一个变量时，分配对应的内存* var *a  
//将数据保存到a对应的内存中，赋值  
 a* = 3  
*//{}会在堆内存中开辟一块空间，以后将数据保存到这块空间  
// =属于一个读取+赋值的操作  
 a* = {}  
*// .属于读取的操作  
 a*.b = 3  
*/\*函数代码\*/* function *fn*() {  
 var a = 1  
 var b = 2  
 }**

# **3.对象**

***1. 什么是对象?  
 \* 代表现实中的某个事物, 是该事物在编程中的抽象  
 \* 多个数据的集合体(封装体)  
 \* 用于保存多个数据的容器  
2. 为什么要用对象?  
 \* 便于对多个数据进行统一管理  
3. 如何创建对象?  
  
4. 对象的组成  
 \* 属性  
 \* 代表现实事物的状态数据  
 \* 由属性名和属性值组成  
 \* 属性名都是字符串类型, 属性值是任意类型  
 \* 方法  
 \* 代表现实事物的行为数据  
 \* 备注：方法是一种特别的属性==>属性值是函数  
5. 如何访问对象内部数据?  
 \* 对象.属性名: 编码简单, 但有时不能用  
 \* 对象['属性名']: 编码麻烦, 但通用***

***6. 什么是空对象***

***\* 没有我们自己定义的属性的对象叫做空对象***

## **3.1问题一**

***什么时候必须使用['属性名']的方式?***

***//1.属性名不是合法的标识名*var *p* = {}  
*//p.en-name = 'kobe' //Invalid left-hand side in assignment  
p*['en-name'] = 'kobe'  
console.log(*p*)*//{en-name: "kobe"}  
//2.属性名是变量(不确定)*var *a* = 'name'  
var *b* = 'age'  
*p*[*a*] = 'kobe'  
console.log(*p*)*//{en-name: "kobe", name: "kobe"}***

## **3.2面试题**

**var *a* = {}  
 var *obj1* = {m:2}  
 var *obj2* = {n:2}  
 *a*[*obj1*] = 4**

***a*[*obj2*] = 5**

**console.log(*a*)*//是多少?***

**var *a* = {}  
 var *obj1* = {m:2}  
 var *obj2* = {n:2}  
 console.log(*obj1*.toString()) *//[object Object]* console.log(*obj2*.toString()) *//[object Object]  
 a*[*obj1*] = 4 *//因为给对象添加属性时中括号里只能放字符串,所以要先把obj1和obj2转换成字符串类型  
 a*[*obj2*] = 5 *//对象类型转换成字符串类型是[object Object]* console.log(*a*)*//{[object Object]: 5} 因为两个对象转换成字符串之后相等,所以对象a里只多了一个属性,它的值为后赋的值* var *arr1* = [1,2,3]  
 var *arr2* = [1,2,4]  
 console.log(*arr1*.toString()) *//1,2,3* console.log(*arr2*.toString()) *//1,2,4  
 a*[*arr1*] = 6 *//数组类型也要先转换成字符串类型  
 a*[*arr2*] = 7 *//数组转字符串是将每一项用逗号拼接起来* console.log(*a*) *//{[object Object]: 5, 1,2,3: 6, 1,2,4: 7}***

***//如果arr1和arr2相等,那么对象里只添加一个属性***

***//输出结果为 {[object Object]: 5, 1,2,3: 7}***

# **4.函数**

***1. 什么是函数?  
 \* 具有特定功能的n条语句的封装体  
 \* 只有函数是可执行的, 其它类型的数据是不可执行的  
 \* 函数也是对象  
2. 为什么要用函数?  
 \* 提高代码复用  
 \* 便于阅读和交流  
3. 如何定义函数?  
 \* 函数声明***

**function *f1*(){} *\* 表达式* var *f2* = function (){} *4. 函数的2种角色  
 \* 函数: 通过()使用  
 \* 对象: 通过.使用 ==> 称之为: 函数对象  
5. 如何调用(执行)函数?  
 \* test()-------直接调用  
 \* new test()---实例调用  
 \* obj.test()---对象调用  
 \* test.call/apply(obj)----通过call和apply调用* var *obj* = {  
 fn3:function () {  
 console.log('我是函数fn3',this)  
 this.m = 123  
 }  
 }  
 var *fn3* = *obj*.fn3  
 *fn3*() *//我是函数fn3 Window {…}  
 //在这里fn3 = window.fn3,所以this指向是window,***

***this.m是把m添加到了window上面***

**console.log(m) *//直接打印window.m,简写成m* var *a* = new *fn3*() *//我是函数 fn3***

***//fn3{} 因为a是一个实例,所以this直接指向它自身*  
 *obj*.fn3() *//直接调用* console.log(*obj*) *//打印它自身* var *obj2* = {info:'快下课了'}  
 *fn3*.call(*obj2*) *//我是函数fn3 {info: "快下课了"}***

***//this指向obj2,this.m给obj2添加了属性m* console.log(*obj2*) *//{info: "快下课了", m: 123}* function *sum*(a,b) {  
 var c = a+b  
 console.log(this)}  
 *sum*(1,2) *//Window {…}  
 sum*.call(*obj2*,1,2) *//{info: "快下课了", m: 123}***

***//call有几个参数不确定,第一个是想让this指向的对象,后面的是实参  
 sum*.apply(*obj2*,[1,2]) *//{info: "快下课了", m: 123}***

***//apply只有两个参数,后面的参数是一个数组,里面存放所有的实参  
   
 /\*  
 \* 使用new操作符的时候，都发生了什么？  
 \* 1.将构造函数中的this强制变为，新生成的实例。  
 \* 2.执行代码  
 \* 3.自动返回新的实例  
 \* \*/* function *Person*(name,age) {  
 this.name = name  
 this.age = age  
 console.log('@@@@',this)  
 }  
 new *Person*('kobe',18)*//@@@@ Person {name: "kobe", age: 18}***

## **4.1回调函数**

***1. 什么函数才是回调函数?  
 \* 你定义的  
 \* 你没有直接调用  
 \* 但最终它执行了(在特定条件或时刻)  
2. 常见的回调函数?  
 \* DOM事件函数  
 \* 定时器函数  
 \* ajax回调函数(后面学)  
 \* 生命周期回调函数(后面学)***

## **4.2 IIFE**

***1. 理解  
 \* 全称: Immediately-Invoked Function Expression 立即调用函数表达式  
 \* 别名: 匿名函数自调用(自调用函数)  
2. 作用  
 \* 隐藏内部实现  
 \* 不污染外部命名空间***

## **4.3 this关键字**

***1. this是什么?  
 \* 一个关键字, 一个内置的引用变量  
 \* 在函数中都可以直接使用this  
 \* this代表调用函数的当前对象  
 \* 在定义函数时, this还没有确定, 只有在执行时才动态确定(绑定)的。(不考虑箭头函数)  
2. 如何确定this的值?  
 \* test()  
 \* obj.test()  
 \* new test()  
 \* test.call/apply(obj)  
 总结: 函数的调用方式决定了this是谁***

***//请说出以下代码的输出*function *fun1*() {  
 function *fun2*() {  
 console.log(this)  
 }  
 *fun2*() *//输出window*}  
*fun1*()**

# **\*\*\*\*\*\*\*\*函数高级\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# **1.原型与原型链**

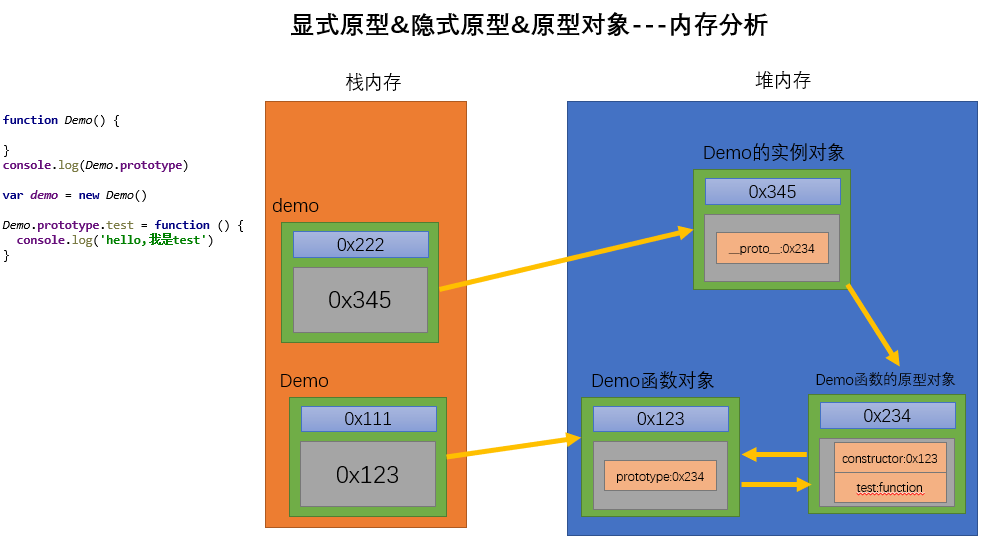
***1. 函数的prototype属性(图)  
 \* 每个函数都有一个prototype属性, 它默认指向一个Object空对象(即称为: 原型对象)  
 \* 原型对象中有一个属性constructor, 它指向函数对象（函数本身）***

***\* 备注：实例能自动看到,缔造它的构造函数的原型对象上,所有的属性  
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)***

***\* 函数（更多的是构造函数）原型对象上的方法是给实例用的***

## **1.1显示原型与隐示原型**

***1. 每个函数function都有一个prototype属性，即显式原型属性（我们常常简称：显式原型）  
 函数的原型对象上有一个constructor属性，它指向了这个函数本身  
2. 每个实例对象都有一个\_\_proto\_\_属性，即隐式原型属性（我们常常简称：隐式原型）  
3. 实例对象的隐式原型属性的值【等于】该实例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, 默认值为它构造函数的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)***

***明确：  
 1.程序员写代码的时候经常操作的是显式原型，因为操作\_\_proto\_\_没意义（ES6之前根本不能操作）  
 2.隐式原型属性是js引擎在执行查找时，内部在用的东西，且程序员是不能操作的（ES6能操作了）***

### **1.1.1关于显式原型(prototype)的问题**

***(1) 什么时候添加的？ 定义函数/创建函数对象 ===>js创建函数对象后，自动给函数添加了prototype属性。  
 (2) 值为多少？ 值是一个Object空对象的地址值 ===> Demo.prototype = {} Demo.prototype.constructor = Demo。  
 (3) 什么时候用？ 给函数的实例对象用***

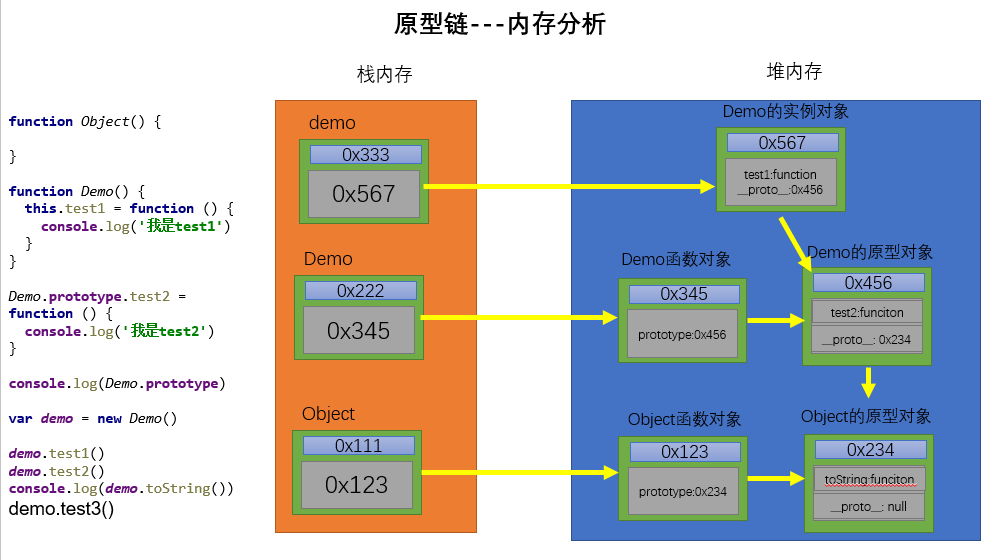
### **1.1.2关于隐式原型(\_\_proto\_\_)的问题**

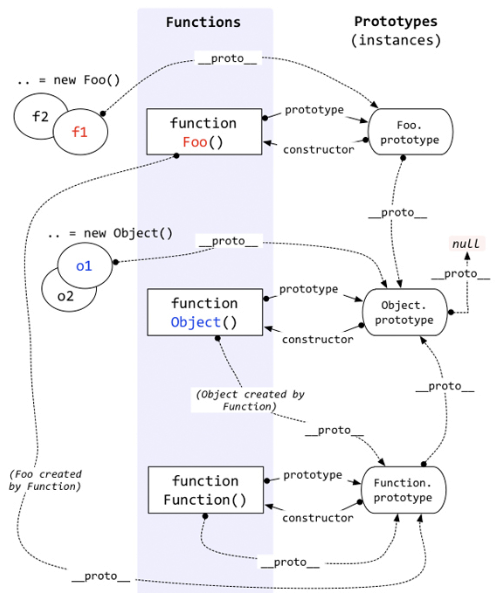
***(1) 什么时候添加的？***

***创建实例的时候，js创建实例之后自动给实例添加\_\_proto\_\_属性。  
(2) 值为多少？ 值为其构造函数显式原型对象的地址，===> this.\_\_proto\_\_ = Demo.prototype  
(3) 什么时候用？ 当查找实例身上的某个属性的时候用，***

## **1.2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\_\_这条链一直找下去, 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/实例对象的关系(图解)  
3. 构造函数/原型/实例对象的关系2(图解)***

****

****

## **1.3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***

## **1.4 instanceof**

***1. instanceof是如何判断的?  
 \* 表达式: A instanceof B  
 \* 如果B函数的原型对象在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 Object)*//true* console.log(Function instanceof Function) *//true* console.log(Object instanceof *Foo*) *//false  
// Function.prototype.\_\_proto\_\_ = Object.prototype* console.log(Function.prototype.\_\_proto\_\_===Object.prototype)**

## **1.5面试题**

***/\*测试题1\*/*var *A* = function() {  
}  
*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) *// 1 undefined 2 3  
/\*测试题2\*/*var *F* = function(){};  
Object.prototype.a = function(){  
 console.log('a()')  
};  
Function.prototype.b = function(){  
 console.log('b()')  
};  
var *f* = new *F*();  
*f*.a() *//a()  
f*.b() *//报错 f.b is not a function  
F*.a() *//a()  
F*.b() *//b()***

# **2.执行上下文与执行上下文栈**

## **2.1变量提升与函数提升**

***1. 变量声明提升  
 变量提升：将变量的声明提前执行，全称: 变量声明提升  
2. 函数声明提升  
 函数提升：将函数的声明（定义）提前执行，全称：函数声明提升  
3. 问题: 变量提升和函数提升是如何产生的?***

***js预解析***

***特别注意：对于同名的变量和函数，预解析提升最后的结果，把函数提升了。***

***1.变量声明必须用var,才能实现变量声明提升   
2.函数必须是使用函数声明的方式去定义函数，才能实现函数提升。***

***/\*面试题: 输出什么?\*/*var *a* = 4  
function *fn* () {  
 console.log(a)  
 var a = 5 *//函数内部声明提升*  
}  
*fn*() *//undefined***

## **2.2执行上下文**

***1. 代码分类(位置)  
 \* 全局代码  
 \* 函数代码  
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  
 \* 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***

## **2.3 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***

***//1. 进入全局执行上下文*var *a* = 10  
var *bar* = function (x) {  
 var b = 5  
 *foo*(x + b) *//3. 进入foo执行上下文*}  
var *foo* = function (y) {  
 var c = 5  
 console.log(*a* + c + y)  
}  
*bar*(10) *//2. 进入bar函数执行上下文***

***1. 依次输出什么?  
2. 整个过程中产生了几个执行上下文?***

**console.log('global begin: '+ *i*)  
var *i* = 1  
*foo*(1);  
function *foo*(i) {  
 if (i == 4) {  
 return;  
 }  
 console.log('foo() begin:' + i);  
 *foo*(i + 1); *//执行到这之后,调用它自身进行跳转,后面的代码放在执行上下文栈里.每一次调用都生成一个函数执行上下文,等最后一次执行上下文结束后,从后往前依次执行没执行的语句.* console.log('foo() end:' + i);  
}  
console.log('global end: ' + *i*)  
*//global begin: undefined'  
//foo() begin:1'  
//foo() begin:2'  
//foo() begin:3'  
//foo() end:3  
//foo() end:2  
//foo() end:1  
//global end: 1***

## **2.4面试题**

***/\*测试题1: 先预处理函数, 后预处理变量, 如果已经存在就会被忽略\*/*console.log(typeof *a*) *// 'function'*function *a*() {}  
var *a  
/\*测试题2: 变量预处理, in操作符\*/*if (!(*b* in window)) {  
 var *b* = 1; *//b的声明直接提升到全局,if无影响*}  
console.log(*b*) *//undefined b提升到全局定义,所以 b in window 为true,if判断进不去  
/\*测试题3: 预处理, 顺序执行\*/*var *c* = 1  
function *c*(c) {  
 console.log(c)  
 var c = 3  
}  
*c*(2) *//js预解析先声明函数,然后是变量,c=1直接把函数覆盖了,所以c()会报错***

# **3.作用域与作用域链**

## **3.1什么是作用域**

***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***

## **3.2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  
 \* 函数执行上下文环境是在调用函数时, 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  
 \* 上下文环境是动态的, 调用函数时创建, 函数调用结束时上下文环境就会被释放  
3. 联系  
 \* 上下文环境(对象)是从属于所在的作用域  
 \* 全局上下文环境(对象)==>全局作用域  
 \* 函数上下文环境（对象）==>对应的函数使用域***

***问题:  
1. 有几个作用域? ---- 1+n（n是定义了几个函数）---- 1+2 ----3  
2. 产生过几个上下文环境对象? ---- 1+n（n是函数调用了几次）---- 1+3 ----4  
3. 最终有几个上下文对象---- 1(window)----其他的在函数调用结束后都释放了***

## **3.3作用域链**

***1. 理解  
 \* 由内向外的多个作用域形成的链  
 \* 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***

## **3.4面试题**

***//问题1: 结果输出多少?  
/\*函数的嵌套：  
\* 1.嵌套着调用,不影响作用域,定义时作用域在哪就在哪  
\* 2.嵌套着定义  
\* \*/* var *x* = 10;  
 function *fn*() {  
 console.log(*x*);  
 }  
 function *show*(f) {  
 var x = 20;  
 f();  
 }  
 *show*(*fn*); *//10  
 //问题2：输出多少* var *obj2* = {  
 fn2: function () {  
 console.log(*fn2*)*//这里报错 fn2 is not defined*  
 }  
 }  
 *obj2*.fn2()**

**console.log(*obj2*.fn2)**

***//问题3***

**var *fn* = function () {  
 console.log(*fn*)  
}  
*fn*() *//ƒ () {console.log(fn)}  
//问题4*  
var *obj2* = {  
 fn2: function () {  
 console.log(this.fn2) *//ƒ () {***

***console.log(this.fn2)***

***console.log(fn2) // 报错***

***}* console.log(*fn2*) *// 报错* }  
}  
*obj2*.fn2()**

# **4.闭包**

## **4.1 什么是闭包**

***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?  
 \* 理解一: 闭包是嵌套的内部函数(绝大部分人)  
 \* 理解二: 包含被引用变量的对象(或者说是一个“容器”)(极少数人) ------ 使用chrome调试查看  
 \* 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)  
 \* 执行外部函数***

## **4.2常见的闭包**

***1. 将函数作为另一个函数的返回值***

**function *fn1*() {  
 var a = 1  
 function *fn2*() {  
 a++  
 console.log(a)  
 }  
 *//return fn2 //使用return在外面需要用变量接收* window.fn2 = *fn2 //添加到window上可以直接使用*}  
*// var f = fn1()  
// f()  
fn2*()  
*fn2*()**

***2. 将函数作为实参传递给另一个函数调用***

**function *delayShow*(msg,time) {  
 setTimeout(function () {  
 console.log(msg)  
 },time)  
}  
*delayShow*('ok',2000)**

## **4.3闭包的作用**

***1. 使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. 让函数外部可以间接操作(读/写)到函数内部的数据  
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? ---- 不存在  
 2. 在函数外部能直接访问函数内部的局部变量吗? ---- 不能***

## **4.4闭包的生命周期**

***1. 产生: 内部函数被定义后就立即产生了闭包(并不是内部函数调用时才产生)  
2. 死亡: 在嵌套的内部函数成为垃圾对象时***

## **4.5闭包的应用**

***闭包的应用2 : 定义JS模块(自调用函数封装)  
 \* 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)  
 \* 只向外暴露一个包含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***

## **4.6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***

## **4.7内存溢出和内存泄露**

***//让浏览器崩溃的一段代码* var *obj* = {}  
 for (var *i* = 0; *i* <1000000; *i*++) {  
 *obj*[*i*] = new Array(1000000)  
 }  
  
function *fn*() {  
 *// 意外的全局变量* a = new Array(10000)  
}  
*fn*()  
  
 *// 没有及时清理的计时器或回调函数* var *intervalId* = setInterval(function () {  
 console.log('-----')  
 }, 200)  
  
 console.log('intervalId', *intervalId*)  
 *// 后面执行* clearInterval(*intervalId*)**

## **4.8面试题**

***//代码片段一*var *name* = "The Window";  
var *object* = {  
 name: "My Object",  
 getNameFunc: function () {  
 return function () {  
 return this.name;  
 }  
 }  
};  
console.log(*object*.getNameFunc()());*//'The Window'***

***//等于window调用  
  
//代码片段二*var *name2* = "The Window";  
var *object2* = {  
 name2: "My Object",  
 getNameFunc: function () {  
 *//缓存this* var that = this;  
 return function () {  
 return that.name2;  
 };  
 }  
};  
console.log(*object2*.getNameFunc()());*// 'My Object'***

***//代码片段三***

**function *Foo*() {  
 *getName* = function () { console.log(1) };  
 return this;  
}  
*Foo*.getName = function () { console.log(2)};*//构造函数有,实例没有*  
*Foo*.prototype.getName = function () { console.log(3)};  
var *getName* = function () { console.log(4)};*//声明提升*  
function *getName*() { console.log(5)} *//声明提升  
  
//请写出以下输出结果：  
Foo*.getName(); *//2  
getName*(); *//4  
Foo*().getName();*//1 执行Foo()第一句之后getName重新赋值,再return  
getName*();*//1 getName 已经改变了*new *Foo*.getName();*//2*new *Foo*().getName();*//3 先执行new Foo()构造实例,实例上没有getName, 去原型链上找*new new *Foo*().getName();*//3***

# **\*\*\*\*\*\*\*\*对象高级\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# **1.对象创建方式**

## **1.1 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（创建多个对象的时候）***

**var *p* = new Object()  
*p*.name = 'kobe'  
*p*.age = 18  
*p*.setName = function (name) {  
 this.name = name  
}**

## **1.2对象字面量**

***\* 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(方法)***

**var *p* = {  
 name:'kobe',  
 age:18,  
 setName:function (name) {  
 this.name = name  
 }  
}**

## **1.3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 *creatPerson*(name,age) {  
 var p = {  
 name:name,  
 age:age,  
 setName:function (name) {  
 this.name = name  
 }  
 }  
 return p  
}**

**var *p1* = *creatPerson*('wade',18)  
var *p2* = *creatPerson*('peiqi',18)**

**console.log(typeof *p1*)  
console.log(typeof *p2*)**

## **1.4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***

**function *Person*(name,age) {  
 this.name = name  
 this.age = age  
 this.setName = function (name) {  
 this.name = name  
 }  
}  
function *Dog*(name,age) {  
 this.name = name  
 this.age = age  
 this.setName = function (name) {  
 this.name = name  
 }  
}  
var *p1* = new *Person*('kobe',18)  
var *p2* = new *Person*('wade',19)  
var *d1* = new *Dog*('wc',19)  
var *d2* = new *Dog*('xh',19)**

## **1.5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) {  
 this.name = name  
 this.age = age  
}  
*Person*.prototype.setName = function (name) {  
 this.name = name  
}**

**var *p1* = new *Person*('kobe',18)  
var *p2* = new *Person*('wade',19)  
*p1*.setName('haha')  
console.log(*p1*)**

# **2.继承模式**

## **2.1原型链继承**

***1. 套路  
 1. 定义父类型的构造函数  
 2. 给父类型的原型对象添加方法  
 3. 定义子类型的构造函数  
 4. 创建父类型的实例赋值给子类型的原型  
 5. 将子类型原型的constructor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***

***//1.定义一个父类的构造方法*function *Parent*() {  
 this.name = '建林'  
}  
  
*//2.给父类型的原型对象添加方法  
Parent*.prototype.makeMoney = function () {  
 console.log(this.name+':小目标，赚个2亿!')  
}  
  
*//3.定义子类型的构造函数*function *Child*() {  
 this.name = '思聪'  
}  
  
*//4.创建父类型的实例赋值给子类型的原型  
Child*.prototype = new *Parent*()  
  
*//5.将子类型原型的构造属性设置为子类型  
Child*.prototype.constructor = *Child  
  
//6.给子类型原型添加方法  
Child*.prototype.spendMoney = function () {  
 console.log(this.name+':不好意思，花了一个亿')  
}  
  
var *c* = new *Child*()  
var *p* = new *Parent*()  
console.log(*p*.\_\_proto\_\_)  
console.log(*c*.\_\_proto\_\_ )  
*c*.spendMoney()  
*c*.makeMoney()**

## **2.2借用构造函数继承**

***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过call()调用父类型构造函数***

***//1.父类的构造函数*function *Parent*(name,age) {  
 this.name = name  
 this.age = age  
 this.getName = function () {  
 console.log(this.name,1)  
 }  
}  
  
  
*//2.子类的构造函数*function *Child*(name,age,price) {  
 */\*this.name = name  
 this.age = age  
 this.getName = function () {  
 console.log(this.name)  
 }\*/***

***//3. 在子类型构造函数中调用父类型构造  
 Parent*.call(this,name,age)  
 this.price = price  
 this.getPrice = function () {  
 console.log(this.price)  
 }  
}  
  
var *c* = new *Child*('haonan',18,16)  
console.log(*c*)  
console.log(*c*.name,*c*.age)  
*c*.getName()  
*c*.getPrice()**

## **2.3组合继承**

***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call()借用父类型构造函数初始化相同属性***

***//定义父类的构造函数*function *Person*(name,age) {  
 this.name = name  
 this.age = age  
}  
*//原型上追加父类方法  
Person*.prototype.setName = function (name) {  
 this.name = name  
}  
*//定义子类的构造函数*function *Student*(name,age,price) {  
 *Person*.call(this,name,age)  
 this.price = price  
}  
*//让子类的构造函数指向父类的实例  
Student*.prototype = new *Person*('demo',23)  
*//给新的原型对象追加一个constructor属性，值为子类构造函数  
Student*.prototype.constructor = *Student  
Student*.prototype.setPrice = function (price) {  
 this.price = price  
}  
  
var *s* = new *Student*('kobe',18,1)  
console.log(*s*)  
console.log(*s*.name,*s*.age,*s*.price);  
*s*.setPrice(2)  
*s*.setName('wade')  
console.log(*s*)**

# **\*\*\*\*\*\*\*\*进程机制与线程机制\*\*\*\*\*\*\*\*\*\*\***

# **1.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  
 \* 老版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***

# **2.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***

# **3.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(比如有开销比较大的代码,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)***

# **4.JS是单线程的**

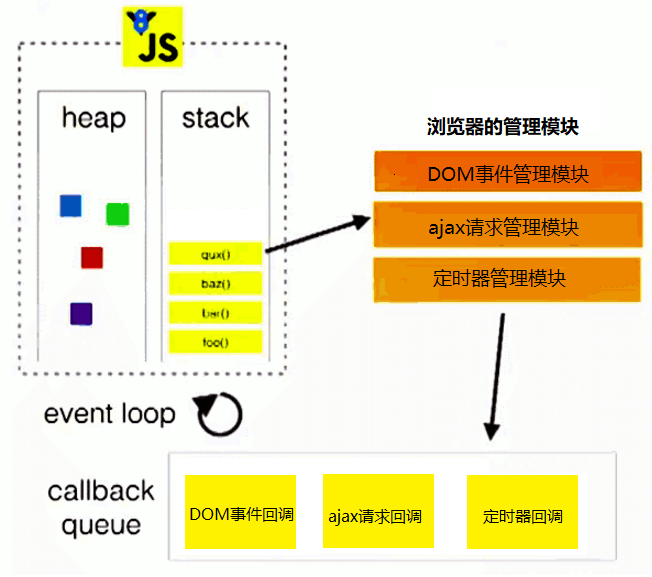
***1. 如何证明js执行是单线程的?  
 \* setTimeout()的回调函数是在主线程执行的  
 \* 定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行  
2. 为什么js要用单线程模式, 而不用多线程模式?  
 \* JavaScript的单线程，与它的用途有关。  
 \* 作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。  
 \* 这决定了它只能是单线程，否则会带来很复杂的同步问题  
3. 代码的分类:  
 \* 初始化代码  
 \* 回调代码  
4. js引擎执行代码的基本流程  
 \* 先执行初始化代码: 包含一些特别的代码  
 \* 设置定时器  
 \* 绑定监听  
 \* 发送ajax请求  
 \* 后面在某个时刻才会执行回调代码***

***/\*  
1. 先执行初始化代码  
2. 后执行回调代码  
\*/*setTimeout(function () {  
 console.log(1)  
},0)  
setTimeout(function () {  
 console.log(2)  
},0)  
setTimeout(function () {  
 console.log(3)  
},0)  
console.log('alert之前')  
*//alert('-------') // 暂停当前主线程的执行, 暂停计时*console.log('alert之后')**

# **5.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***

**function *fn1*() {  
 console.log('fn1()')  
}  
*fn1*()  
document.getElementById('btn').onclick = function () {  
 console.log('处理点击事件')  
}  
setTimeout(function () {  
 console.log('到点了')  
},0)  
function *fn2*() {  
 console.log('fn2()')  
}  
*fn2*()**

****

# **6.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***

***<!--  
计算得到fibonacci数列中第n个数的值  
在主线程计算: 当位数较大时, 会阻塞主线程, 导致界面卡死  
在分线程计算: 不会阻塞主线程  
-->***

***//计算斐波那契数列中第n项的值*function *fibonacci*(n) {  
 return n<3 ? 1 : *fibonacci*(n-2) + *fibonacci*(n-1)  
}  
*//console.log(fibonacci(8));*var *inputData* = document.getElementById('number')  
var *btn1* = document.getElementById('btn1')  
var *btn2* = document.getElementById('btn2')  
*//在主线程上运行开销比较大的代码  
btn1*.onclick = function () {  
 var number = *inputData*.value  
 var result = *fibonacci*(number)  
 alert(result)  
}  
*//在分线程上运行开销比较大的代码  
btn2*.onclick = function () {  
 var number = *inputData*.value  
 *//1.创建web worker实例对象* var worker = new Worker('demo.js')  
 *//2.给分线程发送数据* worker.postMessage(number)  
 *//3.接收分线程执行的结果* worker.addEventListener('message',function (event) {  
 var result = event.data  
 console.log('主线程接收到了分线程返回的结果',result)  
 alert(result)  
 })  
}**

***//demo.js文件***

**function *fibonacci*(n) {  
 return n<3 ? 1 : *fibonacci*(n-2) + *fibonacci*(n-1)  
}  
var *onmessage* = function (event) {  
 var number = event.data  
 console.log('分线程接收到了主线程过来的数据',number)  
 var result = *fibonacci*(number)  
 postMessage(result)  
}**