# JavaScript

JavaScript作为一门脚本语言，他主要的目的是与用户进行交互，请求数据，性能优化等等。如果说HTML，CSS组成了一个人那个人就是植物人，加入JavaScript后就有了灵魂，成为了一个能动的人。JavaScript有几个典型的特点，弱类型，单线程这些特点催生了他与其他编程语言的不同之处。下面就是JavaScript的知识。

## 基础语法部分

**这部分很基础，是写代码最基本的**

### 变量与数据类型

JS的数据类型有很多比较常用的：Number,boolean,String,Null,undefind,object(储存键值对），symbol通常用于object的属性建。

变量的声明有三种：let，var，const。

Let：块级作用域，在同一作用域里不能多次声明。

Var：全局作用域，可以多次声明，并且可以做到变量提升，先用后定义是可以的。

Const:常量，块级作用域，一样不能多次在同一作用域里面声明，而且声明时必须赋值，且之后不能再对值进行更改。

### 运算符与表达式

算数运算符常见的加减乘除：加+减-乘\*除 / 取余% 幂运算算指数\*\*

比较运算符：==比较值是否相等（不考虑类型），===比较两个值是否完全相等（考虑类型的），！=是否不等，！==是否完全不等，>大于，<小于，>=大于等于，<=小于等于。

逻辑运算符：比较常见的就是！反转条件，&&两个条件同时为真，||两个条件中的一个为真。

赋值运算符：=简单赋值，右侧值赋值给左边的变量，+=左边值加右边的值赋给左边的变量，

-=左边的值减右边的值赋值给左边的变量，\*=左边的值乘以右边的值赋值给左边的变量，/=左边的值除以右侧表达式的值赋给左边变量。%=当前值取余于右边的值赋给左边变量。

一元运算符：

++:递增运算符，将变量值增加，分为前置++i，先增后用和后置i++先用后增。

--:递减运算符，将变量值减少，同样分为前置和后置。

+：简单的加法

-：简单的减法

表达式：就是由变量。运算符，函数调用组成的表达式

比如：let a=b+10;这就是一个典型的简单的表达式，复杂点的可能let a=(10>5)&&(20<30)，这个a就是true。

### 条件分支:

If语句：典型的分支结构，if,else,else if这种是编程必备不说了。

Switch语句：switch(a){

Case 88:执行代码，break；

Default:代码

}，a是表达式，下面的意思是当表达式等于值时执行下面代码，break跳出，default是指都不满足时执行的代码。

三元运算符：a>b?a:b这就是一个典型”的三元运算符，a>b是条件，如果满足的话这个表达式的值就是a不满足值就是一个b，可以理解成一个阉割版的if，else。有时候真的很好用。

### 循环:

原始的for循环:for（let i=0;i<10;i++）{}这样最原始的for循环。主要就是初始化表达式。条件表达式，迭代表达式三部分组成的。

原始的while循环：while（表达式）{}，他的意思就是表达式为true时会一直重复执行循环代码直至表达式值为false，如果一直是true就会陷入死循环。do{}while(表达式)，这个是基于while基础上的，管他表达式是不是true的先执行一次，下一次再考虑

for in:for(let i in obj){console.log(i)}这个循环的i其实就是拿到对象或数组键值对中的值，如果是对象拿到的是属性名，如果是索引就是属性值。

for of:这个通常是遍历可迭代对象的，map，set，字符串，数组这样的，拿到的直接是值而不是索引。

上面这两个要做一个明显的区分。语法上差不多，但用途是不咋一样的。

### 数组:

JavaScript中的数组是一个动态的，他会自动扩容。一样分为一维数组和二维数组。

let arr=[1,2,3]这就算声明了一个数组了arr[1]的值就是2，数组索引是从0开始的。

也可以用new Array，ES6提供Array.of（3）这种方法构造数组其目的是防止出现new Array(3)这样的混淆语言，本意是在数组中加入3却创建了一个长度为3的数组，也可以将map，set这种可迭代对象用Array.from转换为数组。具体语法：array.from(map)这个方法就是将可迭代的元素转换为数组。

数组提供了很多方法：

push():arr.push(666)这个方法就是在数组中添加数据666,在最后添加元素并返回数组长度。

pop()：arr.pop()删除数组最后一个元素，并返回删除元素的值。

shift()：arr.shift（），移除数组的第一个元素并返回移除元素的值。

unshift()：arr.unshift()在数组的开头添加元素，并返回数组的长度。

splice():语法arr.splice(1,2,w,k)其中1是删除的起始索引，2是结束索引，w，k是添加的元素。

slice():截取数组内容，arr.slice(1,2)1开始索引，2结束索引，返回的是包含截取数据的数组。

concact():主要用于合并数组或是合并数组和值比如说，arr.concat(1,2,3)这是合并值，arr.concat(arr1,arr2)这是合并数组，最后返回值为数组。

Join()：将所有数组元素连接成一个字符串。

indexof()：返回数组中第一个与给定元素相等的元素的索引，找不到返回-1

lastIndexof()：返回数组中最后一个与给定元素相等的元素的索引，找不到返回-1；

forEach()：对数组进行遍历每一个元素进行一次操作，但没有返回值。

map()：对数组每一个元素进行遍历，操作，最后返回操作后的值形成一个新数组

filter()：对数组元素进行过滤，返回满足条件的元素形成一个新数组。

reduce()：一个累加器 arr.reduce((sum,current)=>{

Sum+current

})，sum为上次处理的结果，current为现在正在处理的值。

every():数组中所有元素都满足条件。

some()：数组中有元素满足条件。

find()：找到第一个满足条件的元素，返回值。

findeIndex()：找到第一个满足条件的元素，返回索引。

includes()：数组是否包含一个指定值。

split:方法将字符串分割成数组，let str=’wc，nmb’ str.split(‘,’)结果为[‘wc’,’nmb’]，他与join是一对相反功能

### 函数:

函数可以很好的解决代码的复用性。

函数声明：

function main(a,b){

return a+b;

}

函数表达式：constant main=function(x,y){return x+y)

箭头函数：ES6提供的比较方便的方法 const main=(x，y)=>{return x+y}

函数调用:let a=main(a,b)

作用域：在函数内部的数据无法在外部调用，例：

constant main=function(x,y){

let wrap=114514;

return x+y}

Console.log(wrap)这样会在控制台报错，因为这是在函数作用域外，函数内部的作用域是不生效的。

参数是能有默认值的。constant main=function(x=114,y){return x+y)，在没传参时会优先用默认值，如果传参就不用默认值了

### 对象:

JS中的对象是一个复合数据类型，允许储存多个值和更复杂的实体。对象可以包含数据，和函数多种方法。

原始对象创建：对象通常是以键值对方向存在的所有我们创建对象就是

let obj={

Name:1

Age:!

Move:()=>{

Console.log(‘恭喜牛魔升天’)；

}

}

用函数配合new 的方法创建对象

function person(name,age){

this.name=name;

this.age=age;

this.fuc=()=>{}

}

let person=new person（牛，22）

用Object.create()进行一个深拷贝

let person1**=**Object.create(person)。

访问对象

访问对象通常用.或[]进行访问

用.就是person.name

用[]就是person[“age”]用键对值对进行访问。

对象遍历

用for（let key in obj）{}这个在上面的循环中有提到过

对象提供了很多方法hasOwnproperty(‘键值对中的键’)，检查是否存在某一属性，valueof这种的方法。

## JavaScript之WebAPI

**这部分是开发中很常用的部分**

### DOM与BOM：

DOM是文档对象模型：将文档HTML文档表现为树结构，通过JS更改元素。

DOM负责的是节点操作，事件处理，样式和类操作，文档查询。

BOM是浏览器对象模型：浏览器厂商提供的方法，或得window窗口的信息如大小，回退这样的，以及定时器setTimeout这样的东西。

### 事件驱动：

主要是实现用户交互的，像鼠标，键盘这种的，在HTML标签中用onload,onclick配合函数用，或者在DOM中用.addEventListener，.on=()=>{}这样的有很多事件，像鼠标移动，键盘之类的去MDN找就能找到完整的，照着文档编程好吧。记住事件是异步的，通过事件的好处是，模块化，异步的同时还能实现用户交互。

### 回调函数：

其实在上面JS的数组时已经有很多回调了，数组的方法都带回调的，其实就是执行一个异步操作的函数，回调函数没有名字，通常喜欢用箭头函数写回调（）=》{}当然如果想要用function（）{}这种老登写法写回调也是没问题的。

### 冒泡与捕获：

说道冒泡与捕获就离不开数据的传播也就是事件流，也就是说给一个元素添加事件，他的父元素子元素都可以响应。给子元素添加事件触发了事件处理，然后会冒泡到父元素，然后再到祖父元素一直到document跟节点，捕获是相反的。

默认都是冒泡的，不支持捕获，以addEventListener（事件名，回调，布尔）冒泡捕获的核心在布尔值上，bool值默认为false就是冒泡，点下面上面响应，点上面下面不响应，如果是true的话就是捕获，点上面下面响应，点下面上面不响应。

### 事件委托:

给父元素绑定元素，然后操作子元素。例如：

Document.getElementById(‘w’).addEvenetListener(‘click’,(e)=>{e.tagName},false)对子元素操作。依赖于事件冒泡，必要时可以e.stopPropagation()这样来阻止冒泡。

### 本地存储:

保存本地数据，常用的是localstorage，sessionStorage

localStorage：存储本地数据，在浏览器关闭后依然保存，除非手动删除，不然一直存在。

localStorage.setItem(‘key’,’value’)；

let value=localStorage.getItem(‘key’)；

localStorage.removeItem(‘key’)；

localStorage.clear()清除所有数据

sesstionStorage:只在当前会话窗口生效的存储，窗口关闭清空

sessionStorage.setItem(‘key’,’value’)；

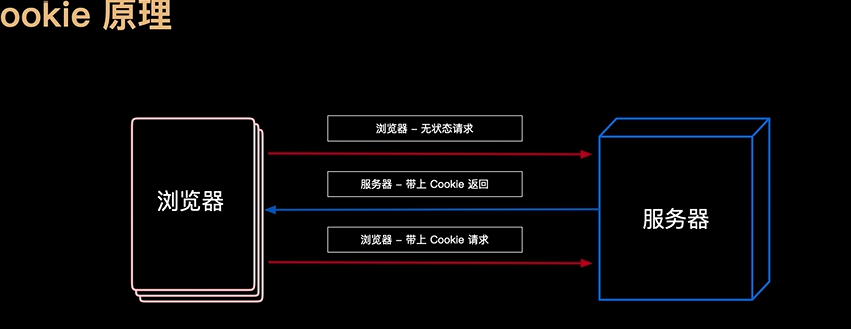
let value=sessionStorage.getItem(‘key’)；

session.removeItem(‘key’)；

session.clear()清除所有数据。

**还有一个细节：不管是localstorage还是sessionstorage，他们存储数据都是用的键值对的方法，值都是字符串。如果我们储存对象类型的数据，或者数组类型的数据需要用JSON.stringfy（）转换为JSON字符串，读出的时候再用JSON.parse解析。**

这里和cookie做一个区分，cookie不能用来存储，cookie最大只能存储4K的数据，他是优化客户端和服务端通信用的，客户端创建document.cookie=”name=’baga”，客户端第一次请求服务端就会返回cookie然后客户端就带着cookie和服务端进行交互。一般是通过服务端在httpresponseheader上的set-cookie，他主要用于的场景就是一个请求。所以缺点也很明显，请求多了自然请求的速度会变慢，并且请求消耗的流量和宽带也会变大。他的设计初衷就是维持http状态，让他更加优化。



## JavaScript高级

**这部分是带点特色的有点难度和技巧的，但本质上还是上面的webAPI和基础语法，只不加了一点点底层原理，半理论半实践吧。**

### 全局对象与活动对象

**全局对象：**在浏览器中的全局对象其实就是window对象

他是全局作用域的顶级对象，

包含了全局属性和全局方法，比如setInterval这样的方法。

他是全局函数的宿主对象，

他是全局变量的宿主对象

**活动对象：**指的是在函数执行上下文中的对象。

他是函数执行上下文的一部分，

包含了参数，局部变量和函数声明

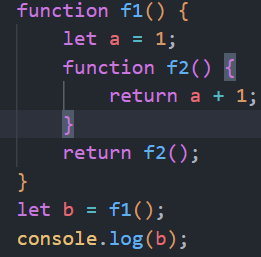
函数执行时创建。

函数执行完毕后销毁。

作用域与闭包

**作用域：**作用域分为全局作用域和局部作用域，全局作用域中的变量任何地方都能用，局部作用域中的变量只能在局部用，函数执行完会销毁所以外面调用会报错。

**闭包：**闭包的本质就是函数嵌套函数例如：

代码与控制台如上。

闭包之后内部变量不会销毁，所以我们在外面也可以调用。

其实原理就是：以上面代码为例f2访问的是内部的局部变量，在外面调用的时候用的就是f2赋给全局变量所以不会被销毁而f2依赖于f1所以f1也在内存中，执行完毕后都不会被销毁。闭包结合了全局变量和局部变量的优点，但也有很明显的缺点，就是函数内部的变量执行完之后不进行删除会一直存在于内存上，影响运行速度。

闭包通常用于封装计时器这类的场景中，防止变量的污染。但是闭包会改变内部变量所以用的时候要谨慎使用。

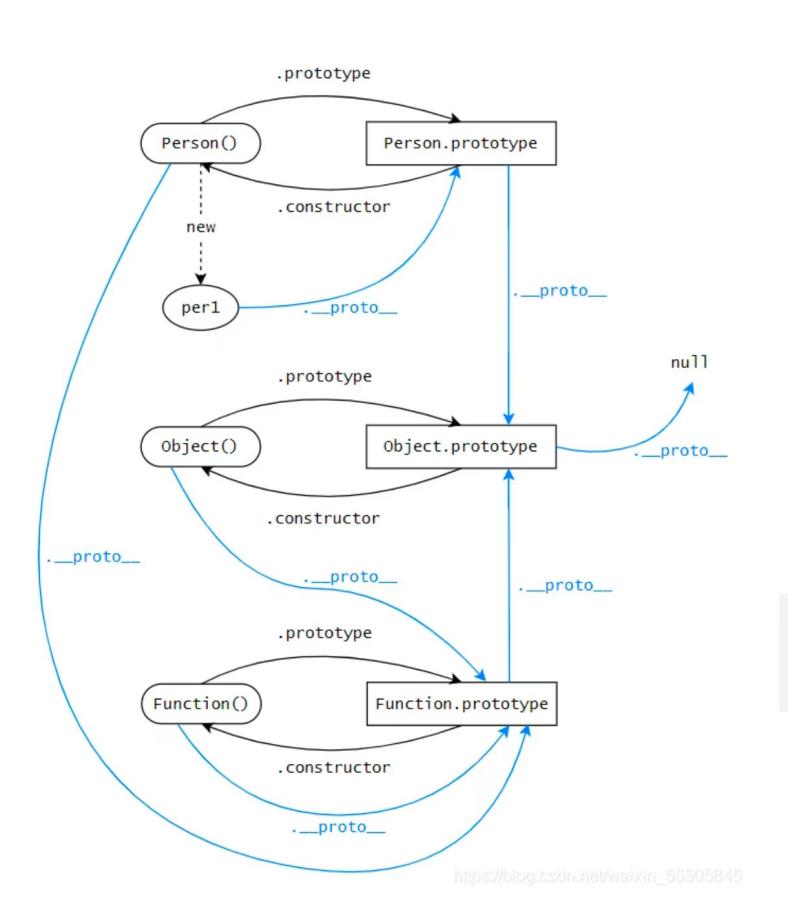
另：个人对闭包的理解就是：函数嵌套函数，变量参数不会被回收，外部能访问内部变量。

原型与原型链

JS中的对象本质都是通过函数创建的，这种情况也是原型和原型链很多人混淆的原因，作者李某学了好几遍也是搞明白了，下面解释一下吧。

其实原型的作用主要在继承上，防止一个属性写多次，所以采用继承方法来优化程序。

每个函数都有prototype，他指向的是原型对象，而函数的原型对象也是对象我们可以通过\_\_proto\_\_访问他的上一级原型，我创建函数的new其实就是construct构造属性实现的而construct是从原型上继承下来的，prototype（也就是函数的原型对象）的construct构造属性指向的是函数自身，我们的对象也是通过new函数创建的，也就意味着，他也有原型。这也就形成了三个互相依赖的“三角恋”。  
关系图如下：



我们上面创建对象的方法 let obj={name:”李某”,age:99}其实就是一个编程语言的语法糖。

最原始的创建方法就是通过函数创建的。function luguan(name){

this.name=name;

}

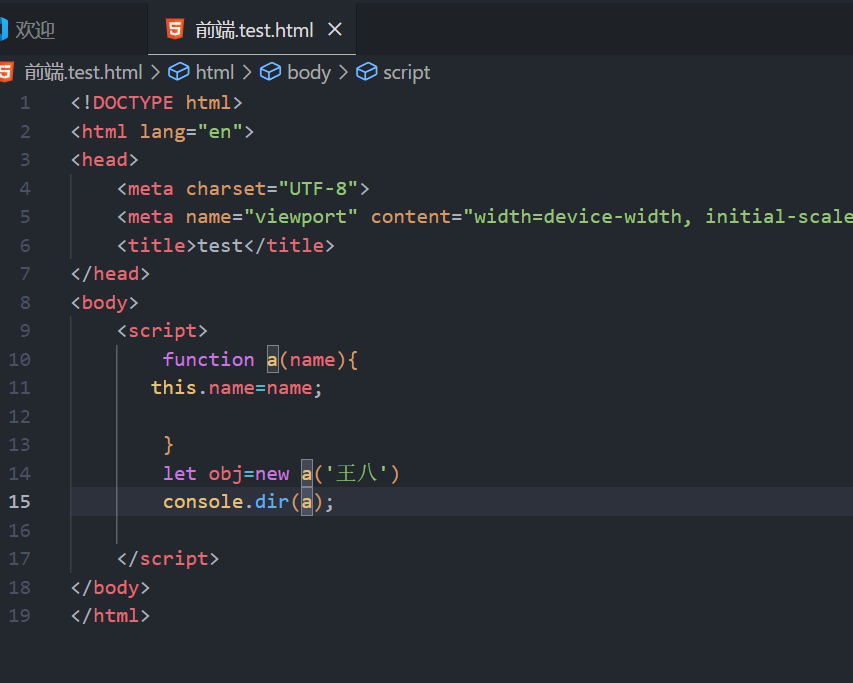
let obj=new luguan(‘李某’)，这种方法创建的函数。他默认什么不写是包含了一个constructor和[[prototype]]这两个属性的。

从这里也能看出我们函数本质也是一个对象。下面就是原型链的核心了。

我们上面创建的obj是一个对象，那么他就有原型我们通过obj.\_\_proto\_\_能拿到他的原型，而obj.\_\_proto\_\_=luguan.prototype，且luguan.prototype也是一个对象我们就能通过luguan.pototype.\_\_proto\_\_继续拿到上一级对象的原型，就是这两个一直循环就构成了原型链。一级一级的查找，直至查到object之后无法再往上找返回undefined。

其实现在很多浏览器都不支持\_\_proto\_\_了其实现在\_\_proto\_\_被obj.getprototype()这个方法替代掉了，这个知道就好了，比较底层的东西，编程实际中用不着，但是面试可能会是一个拷打的点。

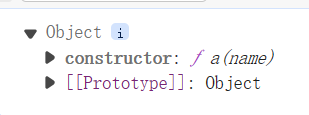
光看字可能不好理解，上代码：



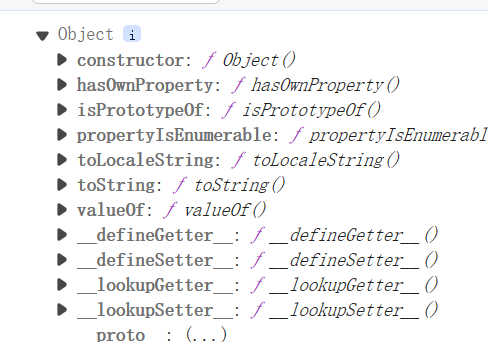
上面代码我们在控制台中显示是这样的，一般用log，查属性用dir



可以看出来a是一个函数有[[prototype]]属性，我们用.prototype拿到函数的prototype如图可以看到是一个object对象



接下来我们一直通过\_\_proto\_\_拿到对象的原型，一直往上拿原型

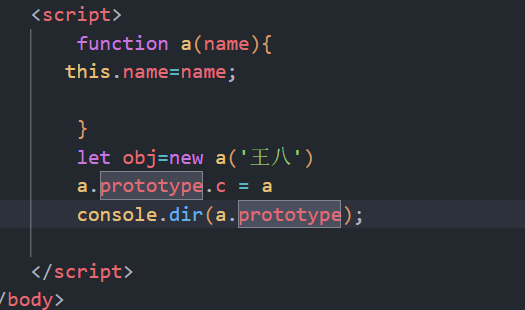


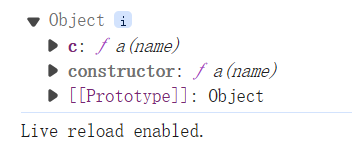
到这一步已经是object了最顶层了，已经看到object提供的方法了。

如果继续调用原型就会是这样IMG_256。因为他已经到顶了。

再举一个比较典型的例子，上面写的数组提供了forEach,map这样的迭代方法，我们new Array的时候并没有给他，其实就是从原型链上拿的。也就是说，当我们在一个对象里找不到就会往原型对象上找，一直往上一直找到object，如果找不到那再返回null就是了没有。这就是原型链的查找原理。

比较绕的一条:我们可以通过原型链给上一级的原型添加属性



我这是通过prototype给函数加的也可以通过\_\_proto\_\_给对象加。

简单一句话：函数对象一层层，往上级级找爹爹，函数prototype找爹爹，对象\_\_proto\_\_找爹爹。  
instanceof是对象是不是以这个为原型的返回boolean ，arr instanceof Array这就是true，还有一个

typeof：  1 typeof Number这也是个true。  
总结：构造函数是使用了new关键字的函数，用来创建对象，所有函数都是Function()的实例  
原型对象是用来存放实例对象的公有属性和公有方法的一个公共对象，所有原型对象都是Object()的实例  
原型链又叫隐式原型链，是由\_\_proto\_\_属性串联起来，原型链的尽头是Object.prototype，这就是原型链的查找机制。

### let和const与var

var:最初的JavaScript只有var一个变量，他的名字叫声明式变量，他存在一个变量提升就是说不管声明还是没声明，只要全局中有，那就不分前面还是后面他都能调用。他默认就放在程序的最顶上，变量提升虽然方便了程序员的开发，但是在项目后期维护时，作用域全部都乱了，项目自然而然的成了史山，很难维护，所以到现在var基本都被废弃掉了，写程序基本都用let和const尽可能的避开用var声明。

let:ES6提供的新的变量声明办法，他的声明是块级作用域也通常是{}里面的东西。在作用域外面没办法调用。

const:ES6提供的新的变量声明办法，他的声明是块级作用域也通常是{}里面的东西。在作用域外面没办法调用。这些与let相似，但他赋值后就不能再赋值了，再赋值就会报错。另外一个与let不同的点是他声明后必须同时进行初始化。

let可以写做let a; a=10。但const只可以是const a=10，且不能重新赋值，初始化成什么就是什么。

### 箭头函数：

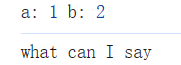
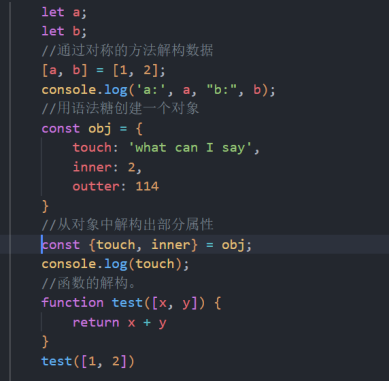
ES6中提供的新语法，其替代了原来的function(){}这种函数，他可以写为（）=>{return a}如果return的值只有一点点可以简化为（）=>a大括号和return都省去了，如果（）内部只有一个参数可以写成item=>item.a这样的极大提高了代码的可读性，同时也方便代码编写。

这两个表示的都是一个东西，上面的箭头函数就是替换下面的老逼登写法的。

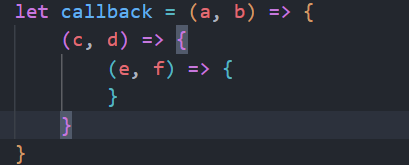
### 解构

解构赋值为ES6提供的对数组，对象，的一种语法。其简化了大量代码，避免了频繁的通过.方法调用对象属性，避免了频繁用索引查数组内容。其实就是一个左右对称赋值。  
下面是一个对象。由于我比较懒，这里就用语法糖来写一下吧。let  obj={  
name:  
age:  
move:}  
我们要拿到对象里的某些属性。  
let {name,move}=obj；这种情况就是典型的结构我们从obj中解构出了两个属性。  
还可以不解构对象，单独赋值变量。  
let ［a,b］=[1,2],结果为：a=1,b=2,  
let  [a,,b]=[1,2,3],结果为：a=1,b=3  
let  [a,...b]=[1,2,3],结果为：a=1,b=[1,2,3]  
解构支持默认值，let ［a=1,b=1］=[1,2],这种情况a=1,b=2,但如果没有对应值时会启用备用值。  
函数参数也可以用解构function a([x,y]){}  
a([1,2])。

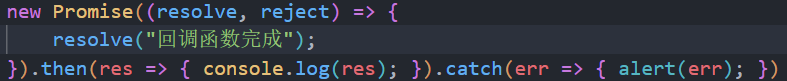
下面是解构的代码示例 ，包含对象解构数组解构，函数解构。

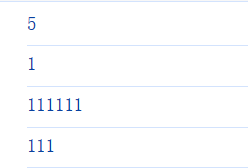
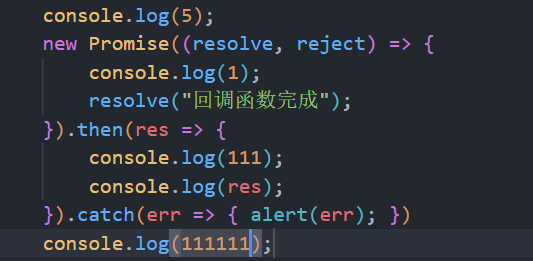
这是控制台  
总结：解构就是一个简化读取对象，数组内容的一个语法。

### Promise

关于promise  
说到promise就不得不说回调函数的缺陷，前面对回调函数的优点解释了，但他存在一个极大的缺陷就是大量嵌套后代就变成了史，根本无法维护，专业名词为回调函数地狱。下面就是一个回调函数地狱的例子：  


这里只嵌套了三个最简单的看起来都有点头大，如果业务复杂到五层六层甚至十层，那是完全没法维护了，为了解决这个问题promise应运而生。

下面是一个语法实例：这个示例请求异步操作是没出错的，所以控制台打印的是回调函数完成，不信可以去试试。  
  
promise为ES6提供的一个语法，他的内部不嵌套回调，创建立即进入pending（执行中）状态，直至resolve（成功），reject（失败），开始就无法更改，具有不可逆性。promise内部函数为同步的，.then为异步的。下面这个代码就可以验证这个说法。



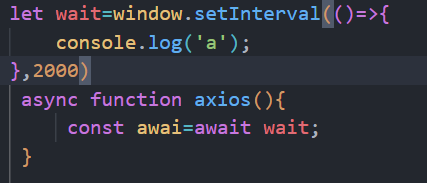
promise.then.then这种链式语法是支持的，这也是后面async和await出现的原因就是为了解决这个问题的。

之所以要catch是因为他内部的错误无法在外部显示。另外.then返回的也是一个promise对象。

promise.all(a,b,c)相当于把a,b,c三个promise对象整合成一个promise对象。如果a,b,c中有不是promise对象的，会被包装成resolve的promise对象。

### async await

上面讲了promise解决了回调函数的嵌套问题，但是promise.then()返回的依然是promise对象他还是可以.then()这样层数多了会一下出现一堆.then()这种情况下，依然是代码可读性可维护性大大降低，所以我们现在将promise再次封装提供了async和await。这就是async和await的代码，他主要是实现异步请求用的，同时使得异步请求更加语义化。

使用：

async放在函数的外面，表示内部是异步的，声明这里面可以用await，同时说明他的返回值会是一个promise可以直接.then调用。await表示后面紧跟的表达式是要等待的结果。async函数默认会返回一个promise对象但如果内部throw一个错误，他会把promise默认为reject请求失败。但如果抛出异常下面就不会执行了直接相当于break，所以我们通常用try，catch来实现请求不成功的处理，这样可以将函数下面的代码块执行完。async函数必须要内部所有函数请求完毕返回的promise对象才能确定状态。await后面跟的都是promise对象，如果不是编译器会将他包装成一个promise对象。多说无益上代码：

这里是在webGL代码隔离中用async和await的例子：

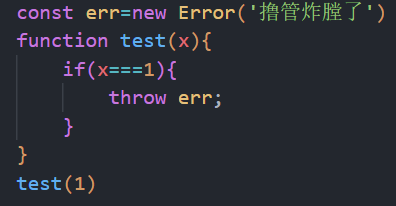




这里就通过了await请求了两个着色器。这里强调一个细节，不存在父子关系的await最好要同级声明。这两个请求在这个函数的内部。同时await得到的数据依然是promise对象，依然可以执行.then操作。

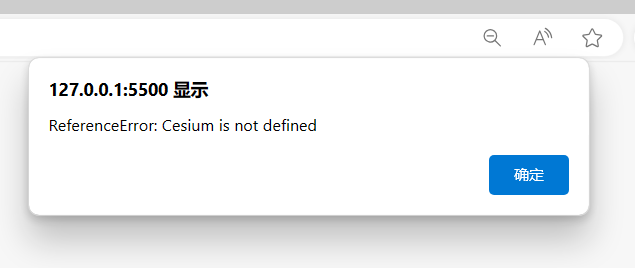
### 异常处理

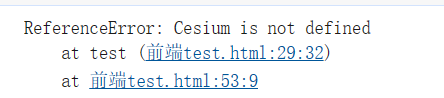
JavaScript异常处理  
很多代码是不在全局报错的，所以会出现代码不正常运行，控制台也不报错的情况，这种情况就需要用到异常处理。异常处理有两种。一种是自定义异常抛出，另一种是try和catch。  
**自定义异常：**let error=new Error()我们自己创建了异常，配合throw抛出error这就是自定义异常。​下面举一个简单的例子：代码与控制台就是下面这样的。



**try和catch处理异常：**  
try｛  
需要检查是否报错的代码  
｝catch（error）｛  
对错误进行操作的代码  
｝  
这个方法类似于promise.then.catch不过不需要回调罢了。下面是一个代码示例。







这里初始化cesium没有CDN引入包，所以报错了，我们通过try和catch可以对异常进行了控制台打印，这种方法通常是给用户使用的，而前面的抛出错误则多是给开发者使用的。

### EventLoop

EventLoop事件循环  
开头处说过，JavaScript是一个单线程的编程语言。这也决定了他的代码就分为了同步代码任务和异步代码任务，而异步代码任务又分为宏任务和微任务，而执行顺序是，同步任务>微任务>宏任务。  
promise内部代码是同步任务，计时器属于宏任务，.then内部函数是微任务。  
下面说一下执行过程。首先代码识别同步任务。放入栈中，同时识别异步任务进入任务队列，在同步任务执行完后再从任务队列选异步任务入栈。就像是一条容易堵车的路，谁先过谁后过，这样可以把一条窄路走流畅。  
下面是一个例子：  
​  
