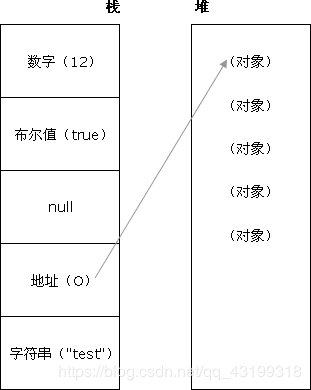
# 前端知识点汇总

### 一、JavaScript

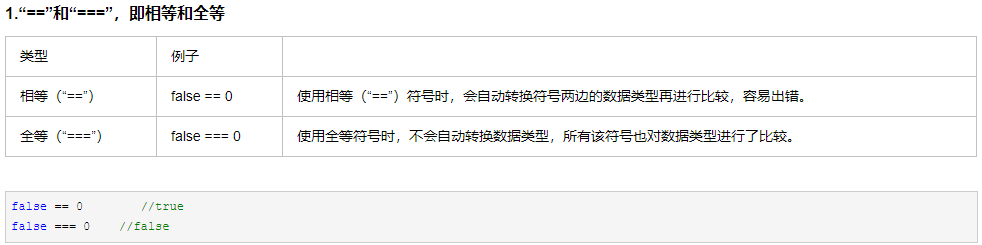
1. 原始值和引用值类型及区别

原始值是存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

引用值是存储在堆中的对象，这个时候变量中存放的是对象的指针，指向对象的内存空间。（注意引用值是一个对象，并不是一个指针）







undefined表示系统级的、出乎意料的或类似错误的值的空缺；表示缺少值，此处应该有值，但没有定义。例如：var a;  a就会显示undefined

null表示程序级的、正常的或在意料之中的值的空缺； 一般多使用null。

2. 判断数据类型typeof、instanceof、Object.prototype.toString.call()、constructor

typeof  检查一个变量的数据类型，共返回6种数据格式：

1、object

2、undefined

3、string

4、number

5、boolean

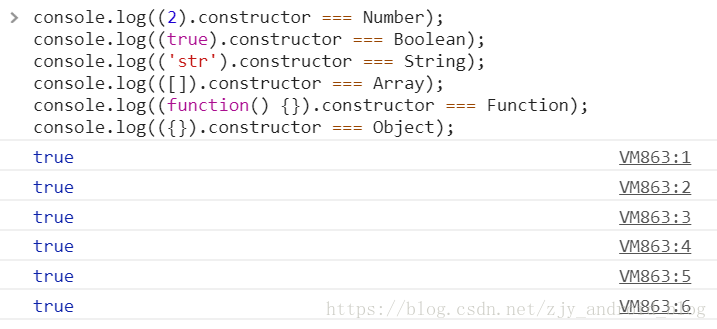
6、function



instanceof，判断一个变量是否是某个对象的实例

instanceof 运算符用来检测 constructor.prototype 是否存在于参数 obj 的原型链上（或者说：检测obj的原型链上是否存在constructor.prototype ）

用 typeof 不能准确判断一个对象变量，所以使用Object.prototype.toString.call(obj)进行精确判断。



所有原型对象都会自动获得一个 constructor（构造函数）属性，这个属性包含一个指向 prototype 属性所在函数的指针。

用costructor来判断类型看起来是完美的，然而，如果我创建一个对象，更改它的原型，这种方式也变得不可靠了。

3. 类数组与数组的区别与转换

  
4. 数组的常见API  
5. bind、call、apply的区别

在JS中，这三者都是用来改变函数的this对象的指向的。  
相似之处：  
1、都是用来改变函数的this对象的指向的。  
2、第一个参数都是this要指向的对象。  
3、都可以利用后续参数传参。  
区别：

call和apply的不同点只有一点，那就是向函数传递参数的方式不同。call是逐个的向函数传递参数，apply是通过将参数以数组的形式传递到函数中。

函数调用bind方法之后返回了一个函数，这个函数必须手动调用一下。还有就是bind向函数传递参数也是逐个传递的，这点类似call。

1. new的原理

  
7. 如何正确判断this？

### **全局环境**

无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this 都指向全局对象。

### **函数（运行内）环境**

简单调用（此调用未设置this）  
非严格模式下，this 的值默认指向全局对象。  
在严格模式下，this将保持他进入执行环境时的值。如果 this 没有被执行环境（execution context）定义，那它将保持为 undefined。  
当一个函数在其主体中使用 this 关键字时，可以通过使用函数继承自Function.prototype 的 call 或 apply 方法将 this 值绑定到调用中的特定对象。（如果传递给 this 的值不是一个对象，JavaScript 会尝试使用内部 ToObject 操作将其转换为对象。）

bind方法  
ES5 引入了 Function.prototype.bind。调用f.bind(someObject)会创建一个与f具有相同函数体和作用域的函数，在这个新函数中，this将永久地被绑定到了bind的第一个参数，无论此函数是如何被调用的。

箭头函数  
在箭头函数中，this与封闭词法环境的this保持一致。在全局代码中，它将被设置为全局对象。

作为对象的方法  
当函数作为对象里的方法被调用时，它们的 this 是调用该函数的对象。（此概念适用于原型链中的this和getter 与 setter 中的 this）

作为构造函数  
当一个函数用作构造函数时（使用new关键字），它的this被绑定到正在构造的新对象。（手动返回其他对象时，则绑定到返回的这个对象上）

作为一个DOM事件处理函数  
当函数被用作事件处理函数时，它的this指向触发事件的元素（一些浏览器在使用非addEventListener的函数动态添加监听函数时不遵守这个约定）。

作为一个内联事件处理函数  
当代码被内联on-event 处理函数调用时，它的this指向监听器所在的DOM元素。

8. 闭包及其作用

它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中，不会在f1调用后被自动清除。

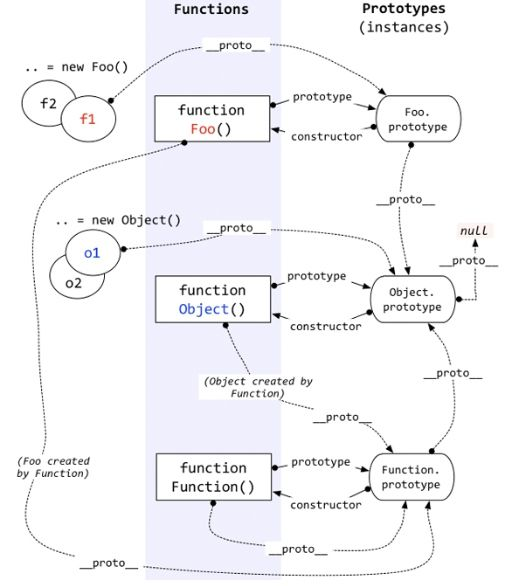
闭包就是能够读取其他函数内部变量的函数。由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包就是能够读取其他函数内部变量的函数，只有函数内部的子函数才能读取局部变量。

作用：一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中，不会在f1调用后被自动清除。

原因就在于f1是f2的父函数，而f2被赋给了一个全局变量，这导致f2始终在内存中，而f2的存在依赖于f1，因此f1也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

9. 原型和原型链



每当定义一个函数对象时候，对象中都会包含一些预定义的属性。其中每个****函数对象****都有一个prototype 属性，这个属性指一个****原型对象，****而这个原型对象中拥有的属性和方法可以被所以实例共享。

原型链是实现继承的主要方法，基本思想就是利用原型让一个引用类型继承另一个引用类型的属性和方法。

要理解原型链就要明白原型对象、构造函数、实例，三者之间的关系。

我们先来梳理三者的关系：

每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，实例也有一个指向原型对象的内部指针。

当这个原型对象是另一个类型的实例，那么这个原型对象就有一个内部指针指向另一个原型，以此类推就构成了一条原型链。原型链的根就是Object.prototype。

每个对象都有 \_\_proto\_\_ 属性，但只有函数对象才有 prototype 属性

所有函数对象proto都指向 Function.prototype，它是一个空函数（Empty function）

原型和原型链是JS实现继承的一种模型。

原型链的形成是真正是靠\_\_proto\_\_ 而非prototype

1、Object是作为众多new出来的实例的基类 function Object(){ [ native code ] }

2、Function是作为众多function出来的函数的基类 function Function(){ [ native code ] }

3、构造函数的proto(包括Function.prototype和Object.prototype)都指向Function.prototype

4、原型对象的proto都指向Object.prototype

5、Object.prototype.proto指向null

10. prototype与\_\_proto\_\_的关系与区别

在JS里，万物皆对象。方法（Function）是对象，方法的原型(Function.prototype)是对象。因此，它们都会具有对象共有的特点。即：对象具有属性proto，可称为隐式原型，一个对象的隐式原型指向构造该对象的构造函数的原型，这也保证了实例能够访问在构造函数原型中定义的属性和方法。

方法(Function)方法这个特殊的对象，除了和其他对象一样有上述proto属性之外，还有自己特有的属性——原型属性（prototype），这个属性是一个指针，指向一个对象，这个对象的用途就是包含所有实例共享的属性和方法（我们把这个对象叫做原型对象）。原型对象也有一个属性，叫做constructor，这个属性包含了一个指针，指回原构造函数。

11. 继承的实现方式及比较

原型链继承

基本思想：子类原型等于父类的实例。

缺点: 子类实例共享属性，造成实例间的属性会相互影响



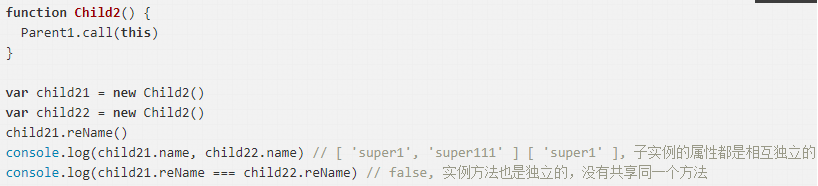
构造函数继承

基本思想：在子类型构造函数的内部调用超类型构造函数（通过使用apply()和call()方法也可以在（将来）新创建的对象上执行构造函数）；

缺点: 父类的方法没有被共享，造成内存浪费。

方法都在构造函数中定义，函数复用变得没有意义

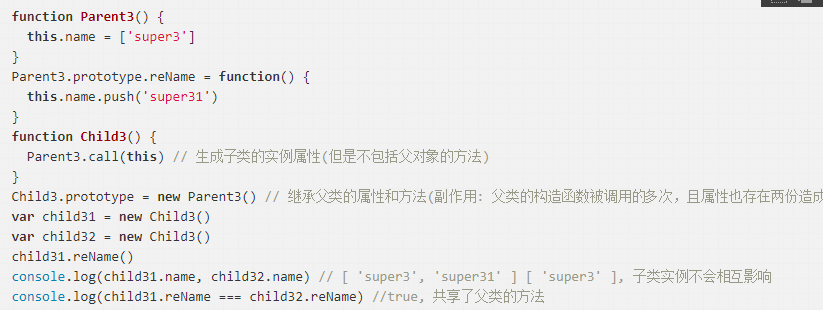
通过call()或者appla()调用父类型时，可以进行传参



组合继承

基本思想：使用原型链实现对原型属性和方法的继承（主要想继承方法），而通过借用构造函数来实现对实例属性的继承（子类型的实例内部存在同名属性，从而对父类型的同名属性进行屏蔽）；最后同时避免了原型链会被继承时会共享同一个父类型属性和借用构造函数的函数复用的缺陷

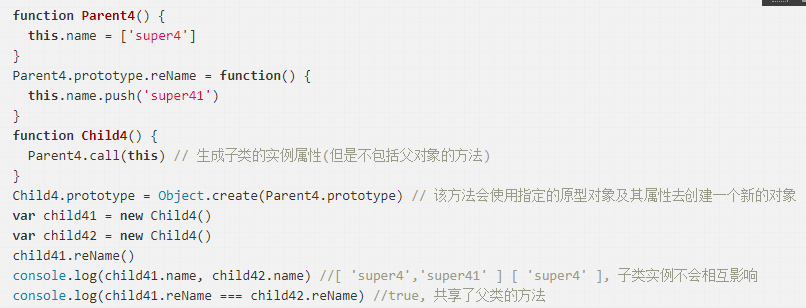
缺点: 父类构造函数被调用两次,子类实例的属性存在两份。造成内存浪费



寄生继承

基本思想：创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。

完美：子类都有各自的实例不会相互影响，且共享了父类的方法。



ES6 class

和寄生继承实现的效果一致

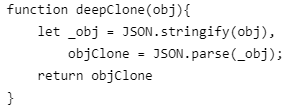


12. 深拷贝与浅拷贝

递归



****借用JSON对象的parse和stringify****



**利用Lodash，解决值为function深拷贝丢失的问题。**

\_.clone(value)：浅拷贝。浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间。

\_.cloneDeep(value)：深拷贝。深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

13. 防抖和节流

在前端开发的过程中，我们经常会需要绑定一些持续触发的事件，如 resize、scroll、mousemove、键盘输入等等，但有些时候我们并不希望在事件持续触发的过程中那么频繁地去执行函数。

1、什么是函数防抖

函数防抖（debounce）:就是让某个函数在上一次执行之后，满足等待某个时间内不再触发此函数后再执行，而在这个等待时间内再次触发函数，等待时间可以重新计算，知道该函数在一定间隔内没有被调用时，才开始执行被调用方法（所谓防抖，就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。）

2、应用场景

搜索框

function fangdou(fn,wait){

    var timer=null;

    return function(){

        if(timer){

            clearTimeout(timer);

            timer=null;

        }

        // console.log(timer)

        timer=setTimeout(function(){

            fn()

        },wait)

    }

}

函数节流（throttle）:是让一个函数无法在很短的时间间隔内连续调用，当上一次执行完之后过了规定的时间间隔，才能进行下一次的函数调用。（所谓节流，就是指连续触发事件但是在N秒中只执行一次函数）节流会稀释函数的执行频率。

即每间隔某个事件去执行某个函数，避免函数的过多执行，这个方式就叫函数节流

2、应用场景

window对象的resize、scroll事件，拖拽时候的mousemove

1. 通用写法

function  jieliu(fn,wait){

    let lastTime=null;

    return function(){

        let newTime=+new Date();

        if(newTime-lastTime>wait||!lastTime){

            fn();

            lastTime=newTime;

        }

    }

}

14. 作用域和作用域链、执行期上下文

**作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。**

**块级作用域、全局作用域、函数作用域。**

**作用域是分层的，内层作用域可以访问外层作用域的变量，反之则不行**。

输出一个自由变量的值，自身作用域如果没有，向父级作用域寻找。一层一层向上寻找，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链。

 JavaScript 属于解释型语言。解释型语言：程序不需要[编译](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91/1258343" \t "https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%9E%8B%E8%AF%AD%E8%A8%80/_blank)，程序在运行时才翻译成[机器语言](https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E8%AF%AD%E8%A8%80/2019225" \t "https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E5%9E%8B%E8%AF%AD%E8%A8%80/_blank)，每执 行一次都要翻译一次。

执行期上下文

　　　　1.创建阶段

　　　　　　(1).生成变量对象

　　　　　　(2).建立作用域链

　　　　　　(3).确定 this 指向

　　　　2.执行阶段

　　　　　　(1).变量赋值

　　　　　　(2).函数引用

　　　　　　(3).执行其他代码

　　　　3.销毁阶段

　　　　　　执行完毕出栈，等待回收被销毁

15. DOM常见的操作方式

DOM的四个基本接口

在DOM接口规范中，有四个基本的接口：Document, Node, NodeList, NamedNodeMap。

Document:Document接口是对文档进行操作的入口，它是从Node接口继承过来的。

Node:Node接口是其他大多数接口的父类。 在DOM树中，Node接口代表了树中的一个节点。

常见的DOM操作

1.查找节点

document.getElementById('id属性值'); 返回拥有指定id的第一个对象的引用

document/element.getElementsByClassName('class属性值'); 返回拥有指定class的对象集合

document/element.getElementsByTagName('标签名'); 返回拥有指定标签名的对象集合

document.getElementsByName('name属性值'); 返回拥有指定名称的对象结合

document/element.querySelector('CSS选择器'); 仅返回第一个匹配的元素

document/element.querySelectorAll('CSS选择器'); 返回所有匹配的元素

document.documentElement; 获取页面中的HTML标签

document.body; 获取页面中的BODY标签

document.all['']; 获取页面中的所有元素节点的对象集合型

2.创建节点

document.createElement('元素名'); 创建新的元素节点

document.createAttribute('属性名'); 创建新的属性节点

document.createTextNode('文本内容'); 创建新的文本节点

document.createComment('注释节点'); 创建新的注释节点

document.createDocumentFragment( ); 创建文档片段节点

3.删除节点

parentNode.removeChild( existingChild );删除已有的子节点，返回值为删除节点

element.removeAttribute('属性名');删除具有指定属性名称的属性，无返回值

element.removeAttributeNode( attrNode );删除指定属性，返回值为删除的属性

4.修改节点

parentNode.replaceChild( newChild, existingChild );用新节点替换父节点中已有的子节点

element.setAttributeNode( attributeName );若原元素已有该节点，此操作能达到修改该属性值的目的

element.setAttribute( attributeName, attributeValue );若原元素已有该节点，此操作能达到修改该属性值的目的

5.插入节点

parent.appendChild( element/txt/comment/fragment );向父节点的最后一个子节点后追加新节点

parent.insertBefore( newChild, existingChild );向父节点的某个特定子节点之前插入新节点

element.setAttributeNode( attributeName );给元素增加属性节点

element.setAttribute( attributeName, attributeValue );给元素增加指定属性，并设定属性值

6.设置样式

ele.style.styleName = styleValue;设置ele元素的CSS样式

16. Array.sort()方法与实现机制

 var test = [444,33,1111,222];

            test.sort(function(a,b){

                return a-b;

            });

冒泡排序  
17. Ajax的请求过程

Ajax是什么

我理解Ajax 是一种异步请求数据的一种技术，对于改善用户的体验和程序的性能很有帮助。

Ajax的使用

(1)创建`XMLHttpRequest`对象,也就是创建一个异步调用对象.

(2)创建一个新的`HTTP`请求,并指定该`HTTP`请求的方法、`URL`及验证信息.

(3)设置响应`HTTP`请求状态变化的函数.

(4)发送`HTTP`请求.

(5)获取异步调用返回的数据.

(6)使用JavaScript和DOM实现局部刷新.

var xmlHttp = new XMLHttpRequest();

xmlHttp.open('GET','demo.php','true');

xmlHttp.send()

xmlHttp.onreadystatechange = function(){

if(xmlHttp.readyState === 4 & xmlHttp.status === 200){

}

}  
18. JS的垃圾回收机制

## **2. 变量的生命周期**

当一个变量的生命周期结束之后它所指向的内存就应该被释放。JS有两种变量，全局变量和在函数中产生的局部变量。局部变量的生命周期在函数执行过后就结束了，此时便可将它引用的内存释放（即垃圾回收），但全局变量生命周期会持续到浏览器关闭页面。

3. JS垃圾回收方式

JS执行环境中的垃圾回收器怎样才能检测哪块内存可以被回收有两种方式：标记清除（mark and sweep）、引用计数(reference counting)。

3.1 标记清除（mark and sweep）

大部分浏览器以此方式进行垃圾回收，当变量进入执行环境（函数中声明变量）的时候，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”，在离开环境之后还有的变量则是需要被删除的变量。标记方式不定，可以是某个特殊位的反转或维护一个列表等。

垃圾收集器给内存中的所有变量都加上标记，然后去掉环境中的变量以及被环境中的变量引用的变量的标记。在此之后再被加上的标记的变量即为需要回收的变量，因为环境中的变量已经无法访问到这些变量。

3.2 引用计数(reference counting)

这种方式常常会引起内存泄漏，低版本的IE使用这种方式。机制就是跟踪一个值的引用次数，当声明一个变量并将一个引用类型赋值给该变量时该值引用次数加1，当这个变量指向其他一个时该值的引用次数便减一。当该值引用次数为0时就会被回收。

19. JS中的String、Array和Math方法

https://www.cnblogs.com/3yleaves/p/9603993.html  
20. addEventListener和onClick()的区别

1. onclick事件在同一时间只能指向唯一对象  
       2.addEventListener给一个事件注册多个listener  
       3.addEventListener对任何DOM都是有效的，而onclick仅限于HTML  
       4.addEventListener可以控制listener的触发阶段，（捕获/冒泡）。对于多个相同的事件处理器，不会重复触发，不需要手动使用removeEventListener清除  
       5.IE9使用attachEvent和detachEvent   
   21. new和Object.create的区别

  
22. DOM的location对象

Location 对象包含有关当前 URL 的信息。

Location 对象是 Window 对象的一个部分，可通过 window.location 属性来访问。

23. 浏览器从输入URL到页面渲染的整个流程（涉及到计算机网络数据传输过程、浏览器解析渲染过程）

从输入url到渲染出整个页面的过程包括三个部分：

1、dns解析url的过程

2、浏览器发送请求与服务器交互的过程

3、浏览器对接收到的html页面渲染的过程

一、dns解析url的过程

      dns解析的过程就是寻找哪个服务器上有请求的资源。因为ip地址不容易记忆，一般会使用url域名（如www.baidu.com）作为网址。dns解析就是将域名翻译成ip地址的过程。

具体过程：

       1）浏览器缓存：浏览器会按照一定的频率 缓存dns记录

 　　2）操作系统缓存：如果浏览器缓存中找不到需要的dns记录，就会取操作系统中找

　　 3）路由缓存：路由器也有dns缓存

　　 4）isp的dns服务器：isp有专门的dns服务器应对dns查询请求

　　 5）根服务器：isp的dns服务器找不到之后，就要向根服务器发出请求，进行递归查询

二、浏览器与服务器交互过程

　　1）浏览器根据解析到的ip地址和端口号发起http请求

　　http请求包括header和body。header中包括请求的方式（get和post）、请求的协议 （http、https、ftp）、请求的地址ip、缓存cookie。body中有请求的内容。

　　2）http请求到达传输层，利用tcp协议与服务器建立连接（三次握手）

　　3）服务器接收到http请求之后，开始搜索html页面，并发送响应报文

　　4）浏览器接收到返回的html页面之后，若状态码显示成功，开始进行页面的渲染

三、浏览器页面渲染过程

　　1）浏览器通过html parse根据深度遍历的方式把html节点遍历成dom 树

　　2）将css解析成css dom树

　　3）将dom树和css dom树构造成render树

　　4）根据得到的render树 计算所有节点在屏幕中的位置进行布局

　　5）遍历render树并调用硬件api绘制所有节点

24. 跨域、同源策略及跨域实现方式和原理

同源策略限制了从同一个源加载的文档或脚本如何与来自另一个源的资源进行交互。这是一个用于隔离潜在恶意文件的重要安全机制。

所谓同源是指，域名，协议，端口均相同。

三种解决方案：jsonp,cors,反向代理nginx

浏览器的回流（Reflow）和重绘（Repaints）

html页面的加载理解

浏览器接收到html代码，可能是一份完整的文档，也可能是一个chunk，即开始解析。解析过程是先构建dom树，再根据dom树构建渲染树，最后浏览器将渲染树绘制到页面上。

构建dom树的过程即根据html代码自上而下进行构建，当遇到script文件加载／执行会阻塞后面dom树的构建（javascript可能会改变dom树），而遇到css文件则会阻塞渲染树的构建，即dom树依然继续构建（除非遇到script标签并且css文件依旧未加载完成），但不会渲染绘制到页面上。而无论哪个阻塞，该加载的文件还是会加载，例如html文档中的其他css／js／图片文件。

另外javascript被加载后就会被执行，执行的过程也阻塞树的构建。是执行完了才解析其他内容，而不是执行完了才加载其他内容。

回流（reflow）：指的是网络浏览器为了重新渲染部分或全部的文档而重新计算文档中元素的位置和几何结构的过程。

页面上节点是以树的形式展现的，我们通过js将页面上的一个节点删除，此时为了不让剩下的节点脱节，将断点结合起来重新形成一棵树。而这个重新结合过程就是回流。就是由于某些修改，要将元素回过头来重新“流”一遍。

重绘（repaints）：是一个元素外观的改变所触发的浏览器行为，例如改变vidibility、outline、背景色等属性。浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。

你应该能感觉到，回流的代价要远大于重绘。且回流必然会造成重绘，但重绘不一定会造成回流。

减少回流的几点建议：

1. 减少不必要的DOM深度。因为无论你改变DOM节点树上任何一个层级都会影响节点树的每个层级——从根结点一直到修改的子节点。不必要的节点深度将导致执行回流时花费更多的时间。  
2. 精简css，去除没有用处的css  
3. 如果你想让复杂的表现发生改变，例如动画效果，那么请在这个流动线之外实现它。使用position-absolute或position-fixed来实现它，也即是让其脱离文档流，不影响父级；现代浏览器也可以使用CSS3 transition实现动画效果，比改变像素值来的高性能。  
4. 避免不必要的复杂的css选择符，尤其是使用子选择器，或消耗更多的CPU去做选择器匹配。

5.页面的元素适当定高，例如如果div内容可能有高度差异的动态内容载入； 页面刷新载入的时候，应避免页面元素的晃动、位移等，这些都是额外的重绘，会让你的CPU和风扇兴奋的

6.图片设定不响应重绘的尺寸，如果你的<img>不设定尺寸、同时外部容器没有定死高宽，则图片在首次载入时候，占据空间会从0到完全出现，左右上下都可能位移，发生大规模的重绘。可以参见新浪微博载入时候页面高度随着图片显示不断变高的问题，这些都让浏览器重绘了，一是体验可能不好，二是烧CPU的。

26. JavaScript中的arguments

arguments是什么？

在JavaScript的函数内部，参数用数组表示，arguments就是用来访问这个数组的的对象。

arguments与数组类似，但不是Array的实例。

arguments.length可以用来确定传递的参数个数。

arguments[i]也可以像数组一样访问每一个传递的参数，i从0开始，必须为正整数。

27. EventLoop事件循环

javascript是单线程。单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。  
于是js所有任务分为两种：同步任务，异步任务。

具体来说，异步运行机制如下：

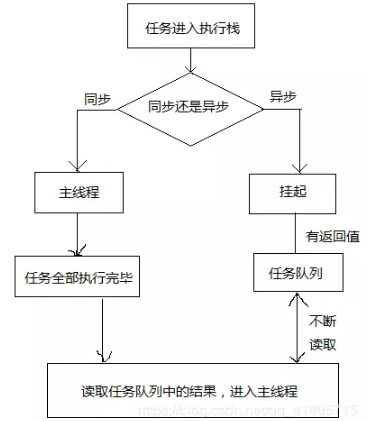
（1）所有同步任务都在主线程上执行，形成一个[执行栈]

（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为Event Loop（事件循环）



28. 宏任务与微任务

* **微任务(microtask)**
  + process.nextTick
  + promise
  + Object.observe(曾经是提案，如今已经废除)
  + MutationOberver
* **宏任务(macrotask)**
  + script
  + setTimeout
  + setInterval
  + setImmediate
  + I/O
  + UI渲染

**执行顺序如下：**

1. 执行同步代码，这是宏任务
2. 执行栈为空，查询是否有微任务要执行
3. 必要时渲染UI
4. 进行下一轮的 **EventLoop** ，执行宏任务中的异步代码

29. BOM属性对象方法

BOM

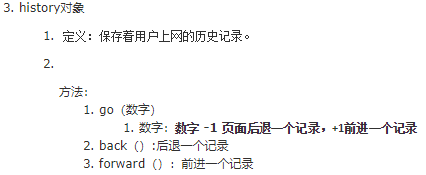
 1.window对象

2.location对象

3.history对象

BOM，即JavaScript可以进行操作的浏览器的各个功能部件的接口。





30. 函数柯里化及其通用封装

柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数的新函数。

用处：

封装函数，减少需传入的参数数量（确定不变参数，减少可变参数）

分步骤传参（语义化）

function add(...args) {

return args.reduce((sums, i) => sums + i);

}

function carrying(fn, ...args1) {

return function (...args2) {

return fn.call(this, ...args1, ...args2);

}

}

carrying(add,1)(2);

1. JS的map()和reduce()方法

js中的map是映射的意思，即原数组被映射成新的数组，而这个数组是由原数组中的每个元素调用一个特定的方法返回值组成的新数组。

js中的reduce（）方法 把函数执行后的结果再进行累加执行，这个函数必须接收两个参数，第一个参数是一个callback，用于针对数组项的操作；第二个参数则是传入的初始值，这个初始值用于单个数组项的操作。reduce()把结果继续和序列的下一个元素做累积计算。最终计算出一个值。

**var** items = [10, 120, 1000];

// our reducer function**var** reducer = **function** **add**(sumSoFar, item) { **return** sumSoFar + item; };

// do the job**var** total = items.reduce(reducer, 0);

console.log(total); // 1130

32. “==”和“===”的区别

等于vs全等  
33. setTimeout用作倒计时为何会产生误差？

setTimeout 误差

上面讲了定时器是属于 宏任务(macrotask) 。如果当前 执行栈 所花费的时间大于 定时器 时间，那么定时器的回调在 宏任务(macrotask) 里，来不及去调用，所有这个时间会有误差。

### 二、ES6

1. let、const和var的概念与区别

#### **var声明的变量会挂载在window上，而let和const声明的变量不会：**

#### **var声明变量存在变量提升，let和const不存在变量提升**

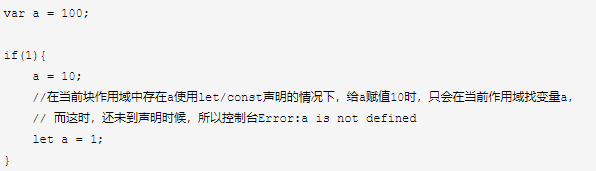
#### **三、let和const声明形成块作用域**

#### **同一作用域下let和const不能声明同名变量，而var可以。** 2. 变量提升与暂时性死区

变量提升即将变量声明提升到它所在作用域的最开始的部分

js中创建函数有两种方式：函数声明式和函数字面量式。只有函数声明才存在函数提升！

var命令会发生“变量提升”现象，即变量可以在声明之前使用，值为undefined。

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。  
3. 变量的结构赋值  
4. 箭头函数及其this问题

箭头函数的this是在定义函数时绑定的，不是在执行过程中绑定的。简单的说，函数在定义时，this就继承了定义函数的对象。箭头函数没有this 等于外部的this。

所以，这会很好的解决匿名函数和setTimeout和setInterval的this指向问题。我们不用再去给其用that变量存储this。  
5. Symbol概念及其作用  
6. Set和Map数据结构

 Set数据结构：它类似于数组，但是成员的值都是唯一的，没有重复的值。Set 本身是一个构造函数，用来生成 Set 数据结构。Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。

Map:类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

1. Proxy  
   8. Reflect对象  
   9. Promise（手撕Promise A+规范、Promise.all、Promise相关API和方法）

Promise流行异步编程解决方案，使用promise对象之后可以使用一种**链式**调用的方式来组织代码；让代码更加的直观。也就是说，有了Promise对象，就可以将异步操作以同步的操作的流程表达出来，避免了层层嵌套的回调函数**（回调地狱）**。

三个状态

pending（执行中）

Resolved（成功，又称Fulfilled）

rejected（拒绝）

一、Promise.all()

类方法，多个 Promise 任务同时执行。  
如果全部成功执行，则以数组的方式返回所有 Promise 任务的执行结果。 如果有一个 Promise 任务 rejected，则只返回 rejected 任务的结果。

Promise.all方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.all([p1, p2, p3]);

二、Promise.race

Promise.race方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。

const p = Promise.race([p1, p2, p3]);

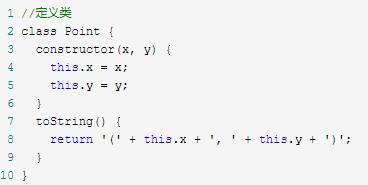
上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

10. Iterator和for...of（Iterator遍历器的实现）  
11. 循环语法比较及使用场景（for、forEach、for...in、for...of）

for...of是ES6新引入的特性。修复了ES5引入的for...in的不足。

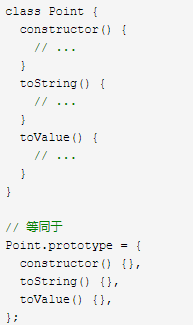
作用于数组的for-in循环除了遍历数组元素以外,还会遍历自定义属性。  
for...of循环不会循环对象的key，只会循环出数组的value，因此for...of不能循环遍历普通对象,对普通对象的属性遍历推荐使用for...in。  
12. Generator及其异步方面的应用

Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。  
13. async函数  
14. 几种异步方式的比较（回调、setTimeout、Promise、Generator、async）  
15. class基本语法及继承



上面代码定义了一个“类”，可以看到里面有一个constructor方法，这就是构造方法，而this关键字则代表实例对象。也就是说，ES5 的构造函数Point，对应 ES6 的Point类的构造方法。

Point类除了构造方法，还定义了一个toString方法。注意，定义“类”的方法的时候，前面不需要加上function这个关键字，直接把函数定义放进去了就可以了。另外，方法之间不需要逗号分隔，加了会报错。ES6 的类，完全可以看作构造函数的另一种写法。



在类的实例上面调用方法，其实就是调用原型上的方法。

Class可以通过extends关键字实现继承

子类必须在constructor方法中调用super方法,否则新建实例时会报错.这是因为子类没有自己的this对象,而是继承父类的this对象,然后对其进行加工.如果不调用super方法,子类就得不到this对象.

super(x, y); // 调用父类的constructor(x, y)

因此super()在这里相当于A.prototype.constructor.call(this)

第二种情况，super作为对象时，在普通方法中，指向父类的原型对象；

16. 模块加载方案比较（CommonJS和ES6的Module）

传统模块解决方案中，CommonJS常应用于服务器端，AMD 规范常用于浏览器端，ES6中的模块加载方案是一种浏览器和服务器通用的解决方案。

ES6 模块的设计思想： 是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。

ES6 模块与 CommonJS 模块完全不同。

CommonJS 模块输出的是值的拷贝            ES6 模块输出的是值的引用。

CommonJS 模块是运行时加载            ES6 模块是编译时输出接口。

 CommonJS 加载的是一个对象（即module.exports属性），该对象只有在脚本运行完才会生成

 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成

17. ES6模块加载与CommonJS加载的原理

### 三、HTML/CSS

1. CSS权重及其引入方式  
2. <a></a>标签全部作用  
3. 用CSS画三角形  
4. 未知宽高元素水平垂直居中（方案及比较）  
5. 元素种类的划分  
6. 盒子模型及其理解  
7. 定位方式及其区别（文档流）  
8. margin塌陷及合并问题  
9. 浮动模型及清除浮动的方法  
10. CSS定位属性  
11. display及相关属性  
12. IFC与BFC  
13. 圣杯布局和双飞翼布局的实现  
14. Flex布局  
15. px、em、rem的区别  
16. Less预处理语言  
17. 媒体查询  
18. vh与vw  
19. H5的语义化作用及语义化标签  
20. Web Worker和Web Socket  
21. CSS3及相关动画  
22. 如何实现响应式布局  
23. SEO的概念及实现  
24. HTML5的新特性  
25. Less和Sass使用

### 四、HTTP与计算机网络

1. TCP/IP协议分层管理  
2. 三次握手四次挥手机制及原因  
3. HTTP方法  
4. GET和POST的区别  
5. HTTP建立持久连接的意义  
6. HTTP报文的结构  
7. HTTP状态码  
8. Web服务器及其组成  
9. HTTP报文首部  
10. HTTP通用首部字段  
11. HTTP请求首部字段、响应首部字段、实体首部字段  
12. Cookie相关首部字段  
13. HTTPS与HTTP区别及实现方式  
14. Cookie与Session  
15. 基于HTTP的功能追加协议（SPY、WebSocket、HTTP）  
16. 常见的Web攻击分类  
17. TCP与UDP区别  
18. 存储机制localStorage、sessionStorage与Cookie存储技术  
19. XSS攻击及防御  
20. CSRF攻击及防御

### 五、前端工程化

1. 前端工程化的流程（架构选型、业务开发、测试、打包构建、部署上线、项目监控）  
2. Webpack基本概念与配置  
3. loader与plugin原理与实现  
4. Webpack的模块热替换及实现  
5. Webpack的优化问题  
6. SPA及其优缺点  
7. SSR实现及优缺点  
8. 设计模式（工厂模式、单例模式、原型模式、代理模式、适配器模式、观察者模式等...）

### 六、React

1. React自身特点及选型时考虑  
2. React与VUE的异同  
3. Virtual DOM  
4. React生命周期  
5. Diff算法  
6. 受控组件与非受控组件  
7. 高阶组件  
8. Flux架构模式（涉及MVC/MVVM、Flux）  
9. Redux设计概念、设计原则、方法、redux实现异步流的库  
10. 纯组件（Pure Component）与shouldComponentUpdate关系  
11. Redux中的<Provider/>组件与connect函数  
12. React Fiber架构  
13. React Hooks的作用及原理

### 七、NodeJS

1. NodeJS基本概念与特点  
2. CommonJS规范、核心模块  
3. Node的异步I/O  
4. Node的内存控制  
5. Node构建网络服务（TCP、HTTP、Web Socket服务等）  
6. Node的进程

### 八、需要会手撕的代码部分

1. Promise（A+规范）、then、all方法  
2. Iterator遍历器实现  
3. Thunk函数实现（结合Generator实现异步）  
4. async实现原理（spawn函数）  
5. class的继承  
6. 防抖和节流  
7. Ajax原生实现  
8. 深拷贝的几种方法与比较  
9. 继承的几种实现与比较  
10. 未知宽高的元素水平垂直居中  
11. 三栏布局的实现  
12. 两栏布局的实现  
13. React高阶组件  
14. 数组去重

Array.from(new Set(array))  
15. 几种排序算法的实现及其复杂度比较  
16. 前序后序遍历二叉树（非递归）  
17. 二叉树深度遍历（分析时间复杂度）  
18. 跨域的实现（JSONP、CORS）

### 九、数据可视化

1. Canvas和SVG的区别  
2. 在考虑设计可视化图表时，结合Canvas和SVG特性会怎么取舍  
3. 常见的可视化组件库  
4. 可视化组件库如Echarts的设计思路  
5. 一些偏向底层的可视化组件库和前端框架结合方面需要考虑哪些问题  
6. 可视化组件如何做到数据驱动？

### 十、计算机基础

1. 计算机系统  
2. 线程与进程  
3. 常见的git指令  
4. Linux相关指令  
5. 其他类型的编程语言（如Java）  
6. 数据库