**JS 基础知识点与高频考题解析**

**变量类型**

**1，介绍js的基本数据类型。**

Undefined、Null、Boolean、Number、String

**2，类型判断用到哪些方法？**

**typeof**

typeof xxx得到的值有以下几种类型：undefined boolean number string object function、symbol ，比较简单，不再一一演示了。这里需要注意的有三点：

\* typeof null结果是object ，实际这是typeof的一个bug，null是原始值，非引用类型

\* typeof [1, 2]结果是object，结果中没有array这一项，引用类型除了function其他的全部都是object

\* typeof Symbol() 用typeof获取symbol类型的值得到的是symbol，这是 ES6 新增的知识点

**instanceof**

用于实例和构造函数的对应。例如判断一个变量是否是数组，使用typeof无法判断，但可以使用[1, 2] instanceof Array来判断。因为，[1, 2]是数组，它的构造函数就是Array。同理：

function Foo(name) {

   this.name = name

}

var foo = new Foo('bar’)

console.log(foo instanceof Foo) // true

**3，JavaScript有几种类型的值？，你能画一下他们的内存图吗？**

栈：原始数据类型（Undefined，Null，Boolean，Number、String）

堆：引用数据类型（对象、数组和函数）

两种类型的区别是：存储位置不同；

1，原始数据类型直接存储在栈(stack)中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储；

2，引用数据类型存储在堆(heap)中的对象,占据空间大、大小不固定,如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体

在参数传递方式上，原始类型是按值传递，引用类型是按共享传递

JS 中这种设计的原因是：按值传递的类型，复制一份存入栈内存，这类类型一般不占用太多内存，而且按值传递保证了其访问速度。按共享传递的类型，是复制其引用，而不是整个复制其值（C 语言中的指针），保证过大的对象等不会因为不停复制内容而造成内存的浪费。

**4，介绍js有哪些内置对象？**

Object 是 JavaScript 中所有对象的父对象

数据封装类对象：Object、Array、Boolean、Number 和 String

其他对象：Function、Arguments、Math、Date、RegEx、Error

**5，如何区分数组和对象？**

1、从原型入手，Array.prototype.isPrototypeOf(obj);  利用isPrototypeOf()方法，判定Array是不是在obj的原型链中，如果是，则返回true,否则false。Array.prototype.isPrototype([]) //true

2、也可以从构造函数入手，利用对向的constructor属性

3、根据对象的class属性(类属性)，跨原型链调用toString()方法。Object.prototype.toString.call(Window);

4、Array.isArray()方法。

**6，null，undefined 的区别？**

null        表示一个对象被定义了，值为“空值”；

undefined   表示不存在这个值。

typeof undefined                      //"undefined"

undefined :是一个表示"无"的原始值或者说表示"缺少值"，就是此处应该有一个值，但是还没有定义。当尝试读取时会返回 undefined；

例如变量被声明了，但没有赋值时，就等于undefined

typeof null        //"object"

null : 是一个对象(空对象, 没有任何属性和方法)；

例如作为函数的参数，表示该函数的参数不是对象；

注意：

在验证null时，一定要使用　=== ，因为 == 无法分别 null 和　undefined

undefined表示"缺少值"，就是此处应该有一个值，但是还没有定义。典型用法是：

1）变量被声明了，但没有赋值时，就等于undefined。

2) 调用函数时，应该提供的参数没有提供，该参数等于undefined。

3）对象没有赋值的属性，该属性的值为undefined。

4）函数没有返回值时，默认返回undefined。

null表示"没有对象"，即该处不应该有值。典型用法是：

1） 作为函数的参数，表示该函数的参数不是对象。

2） 作为对象原型链的终点。

7，数组和对象有哪些原生方法，列举一下？

8，字符串有哪些原生方法，列举一下？

**9，声明变量和声明函数的提升有什么区别?**

(1) 变量声明提升：变量申明在进入执行上下文就完成了。

只要变量在代码中进行了声明，无论它在哪个位置上进行声明， js引擎都会将它的声明放在范围作用域的顶部；

(2) 函数声明提升：执行代码之前会先读取函数声明，意味着可以把函数申明放在调用它的语句后面。

只要函数在代码中进行了声明，无论它在哪个位置上进行声明， js引擎都会将它的声明放在范围作用域的顶部；

(3) 变量or函数声明：函数声明会覆盖变量声明，但不会覆盖变量赋值。

同一个名称标识a，即有变量声明var a，又有函数声明function a() {}，不管二者声明的顺序，函数声明会覆盖变量声明，也就是说，此时a的值是声明的函数function a() {}。注意：如果在变量声明的同时初始化a，或是之后对a进行赋值，此时a的值变量的值。eg: var a; var c = 1; a = 1; function a() { return true; } console.log(a);

10，== 和 === 的区别

**原型，原型链**

**1，JavaScript原型，原型链 ? 有什么特点？**

**原型**

每个对象都会在其内部初始化一个属性，就是prototype(原型)

使用hasOwnProperty() 可以判断这个属性是不是对象本身的属性

**问题：Javascript中，有一个函数，执行时对象查找时，永远不会去查找原型，这个函数是？**

hasOwnProperty

javaScript中hasOwnProperty函数方法是返回一个布尔值，指出一个对象是否具有指定名称的属性。此方法无法检查该对象的原型链中是否具有该属性；该属性必须是对象本身的一个成员。

使用方法：

object.hasOwnProperty(proName)

其中参数object是必选项。一个对象的实例。

proName是必选项。一个属性名称的字符串值。

如果 object 具有指定名称的属性，那么JavaScript中hasOwnProperty函数方法返回 true，反之则返回 false。

**原型链**

当我们在访问一个对象的属性时，如果这个对象内部不存在这个属性，那么他就会去prototype里找这个属性，这个prototype又会有自己的prototype，于是就这样一直找下去，找到Object.\_\_proto\_\_为止，找不到就返回unde也就是我们平时所说的原型链的概念。

关系：instance.constructor.prototype = instance.\_\_proto\_\_

特点：

JavaScript对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

当我们需要一个属性的时，Javascript引擎会先看当前对象中是否有这个属性， 如果没有的话，就会查找他的Prototype对象是否有这个属性，如此递推下去，一直检索到 Object 内建对象。

所有的引用类型（数组、对象、函数），都具有对象特性，即可自由扩展属性（null除外）

所有的引用类型（数组、对象、函数），都有一个\_\_proto\_\_属性，属性值是一个普通的对象

所有的函数，都有一个prototype属性，属性值也是一个普通的对象

所有的引用类型（数组、对象、函数），\_\_proto\_\_属性值指向它的构造函数的prototype属性值

**原型链中的this**

所有从原型或更高级原型中得到、执行的方法，其中的this在执行时，就指向了当前这个触发事件执行的对象。

**闭包**

闭包的形成与变量的作用域以及变量的生存周期有密切的关系

**1，变量的作用域**

* 在js中我们把作用域分为全局作用域和局部作用域，全局作用域就是window，在没有块级作用域概念的时候，每一个函数都是一个局部作用域。
* 其实变量的作用域，就说指变量的有效范围。我们最长说的就是在函数中声明的变量作用域。
* 当在函数中声明一个变量的时候，如果改变量没有用var关键字去定义那么该变量就是一个全局变量，但是这样做最容易造成命名冲突。
* 另一种情况就是使用var声明的变量，这时候的变量就是局部变量，只有在该函数内部可以访问，在函数外面是访问不到的
* 在javascript中，函数可以用来创造函数作用域。在函数中搜索变量的时候，如果该函数当中没有这个变量，那么这次搜索过程会随着代码执行环境创建的作用域链往外层逐层搜索，一直搜索到window对象为止，找不到就会抛出一个为定义的错误。而这种从内到外逐层查找的关系在js中我们称为**作用域链**

**2，变量的生存周期**

除了变量作用域之外，另外一个跟闭包有关的概念就是变量的生存周期，对于全局变量来说，全局变量的生存周期是永久的，除非我们主动销毁这个全局变量，而对于函数内部的使用var声明的变量来说，当退出函数是，这些变量就会随着函数的结束而销毁。

**3，闭包的形成**

Javascript允许使用内部函数，可以将函数定义和函数表达式放在另一个函数的函数体内。而且，内部函数可以访问它所在的外部函数声明的局部变量、参数以及声明的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。常见的闭包写法就是简单的函数套函数，通过另一个函数访问这个函数的局部变量,利用闭包可以突破作用域链，将函数内部的变量和方法传递到外部，延续变量的生命。使用闭包可以减少全局环境的污染，也可用延续变量的生命。

**4，闭包的适用场景**

闭包的适用场景非常广泛，首先从闭包的优点出发就是：

**减少全局环境的污染生成独立的运行环境**

模块化就是利用这个特点对不同的模块都有自己独立的运行环境，不会和全局冲突，模块和模块之间通过抛出的接口进行依赖使用

以及像我们常用的jquery类库（避免和全局冲突使用闭包实现自己独立的环境）

**可以通过返回其他函数的方式突破作用域链**

可以利用这个功能做一些值的缓存工作，例如常见的设计模式（单例模式），以及现在比较火的框架vue中的计算属性

其实当遇到以下场景的时候都可以使用闭包

1.维护函数内的变量安全,避免全局变量的污染。

2.维持一个变量不被回收。

3.封装模块

**5，闭包的缺点**

由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大。所以在闭包不用之后，将不使用的局部变量删除，使其被回收。在IE中可能导致内存泄露，即无法回收驻留在内存中的元素，这时候需要手动释放。

**6，内存泄露**

内存泄漏指一块被分配的内存既不能使用，又不能回收，直到浏览器进程结束。

出现原因：

1.循环引用：含有DOM对象的循环引用将导致大部分当前主流浏览器内存泄露。循环 引用，简单来说假如a引用了b,b又引用了a,a和b就构成了循环引用。

2.JS闭包：闭包，函数返回了内部函数还可以继续访问外部方法中定义的私有变量。

3.Dom泄露，当原有的DOM被移除时，子结点引用没有被移除则无法回收。

**7，JavaScript垃圾回收机制**

Javascript中，如果一个对象不再被引用，那么这个对象就会被GC(garbage collection)回收。如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。垃圾回收不是时时的，因为其开销比较大，所以垃圾回收器会按照固定的时间间隔周期性的执行。

函数a被b引用，b又被a外的c引用，这就是为什么函数a执行后不会被回收的原因。

**8，垃圾回收的两个方法：**

**标记清除法：**

1.垃圾回收机制给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包）。

2.操作1之后内存中仍存在标记的变量就是要删除的变量，垃圾回收机制将这些带有标记的变量回收。

**引用计数法：**

1.垃圾回收机制给一个变量一个引用次数，当声明了一个变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1。

2.当该变量的值变成了另外一个值，则这个值得引用次数减1。

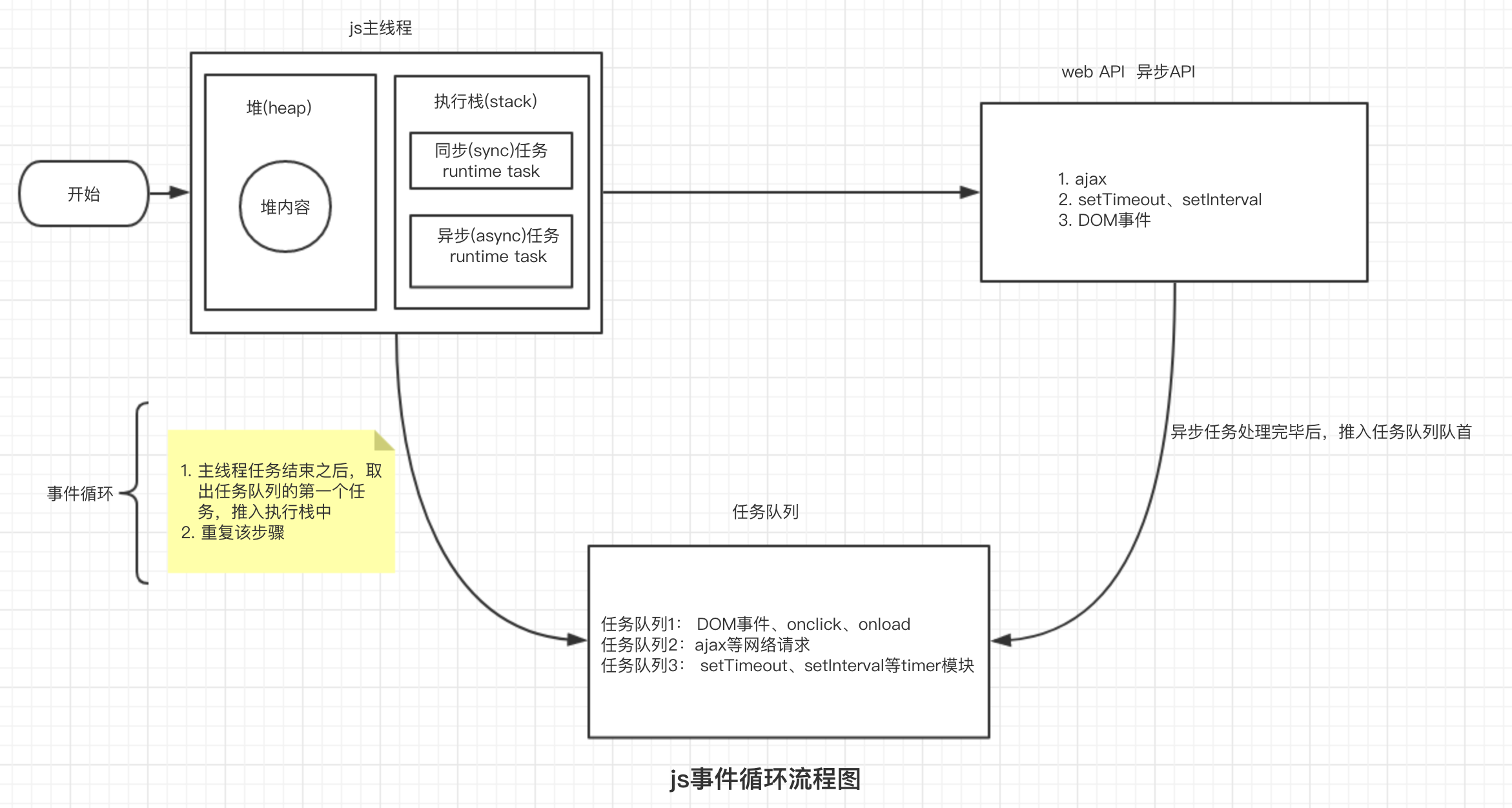
3.当这个值的引用次数变为0的时候，说明没有变量在使用，垃圾回收机制会在运行的时候清理掉引用次数为0的值占用的空间。

**JS运行机制**

JavaScript引擎是单线程运行的,浏览器无论在什么时候都只且只有一个线程在运行JavaScript程序.浏览器的内核是多线程的，它们在内核制控下相互配合以保持同步，一个浏览器至少实现三个常驻线程：javascript引擎线程，GUI渲染线程，浏览器事件触发线程。这些异步线程都会产生不同的异步的事件.

1. javascript引擎是基于事件驱动单线程执行的，JS引擎一直等待着任务队列中任务的到来，然后加以处理，浏览器无论什么时候都只有一个JS线程在运行JS程序。
2. GUI渲染线程负责渲染浏览器界面，当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时,该线程就会执行。但需要注意 GUI渲染线程与JS引擎是互斥的，当JS引擎执行时GUI线程会被挂起，GUI更新会被保存在一个队列中等到JS引擎空闲时立即被执行。
3. 事件触发线程，当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可来自JavaScript引擎当前执行的代码块如setTimeOut、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。（当线程中没有执行任何同步代码的前提下才会执行异步代码）

当程序启动时, 一个进程被创建，同时也运行一个线程, 即为主线程，js的运行机制为单线程



程序中跑两个线程，一个负责程序本身的运行，作为主线程； 另一个负责主线程与其他线程的的通信，被称为“Event Loop 线程" 。每当遇到异步任务，交给 EventLoop 线程，然后自己往后运行，等到主线程运行完后，再去 EventLoop 线程拿结果。

1）所有任务都在主线程上执行，形成一个执行栈（execution context stack）。

2）主线程之外，还存在一个"任务队列"（task queue）。系统把异步任务放到"任务队列"之中，然后继续执行后续的任务。

3）一旦"执行栈"中的所有任务执行完毕，系统就会读取"任务队列"。如果这个时候，异步任务已经结束了等待状态，就会从"任务队列"进入执行栈，恢复执行。

4）主线程不断重复上面的第三步。

"回调函数"（callback），就是那些会被主线程挂起来的代码。异步任务必须指定回调函数，当异步任务从"任务队列"回到执行栈，回调函数就会执行。"任务队列"是一个先进先出的数据结构，排在前面的事件，优先返回主线程。主线程的读取过程基本上是自动的，只要执行栈一清空，"任务队列"上第一位的事件就自动返回主线程。

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。**

**从主线程的角度看，一个异步过程包括下面两个要素：**

发起函数(或叫注册函数)A

回调函数callbackFn

它们都是在主线程上调用的，其中注册函数用来发起异步过程，回调函数用来处理结果。

**异步进程有:**

类似onclick等，由浏览器内核的DOM binding模块处理，事件触发时，回调函数添加到任务队列中；

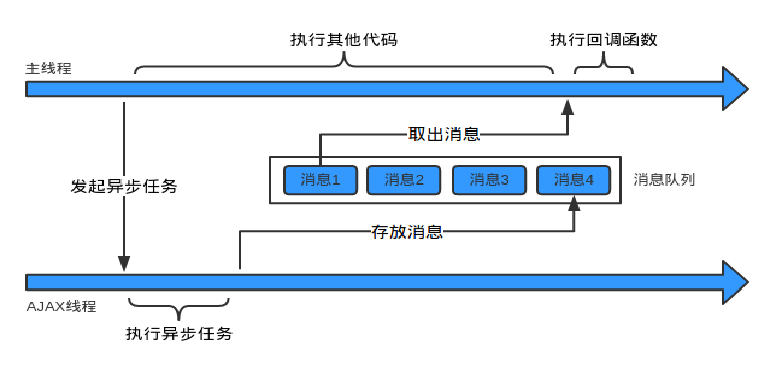
setTimeout等，由浏览器内核的Timer模块处理，时间到达时，回调函数添加到任务队列中；

Ajax，由浏览器内核的Network模块处理，网络请求返回后，添加到任务队列中。

例如setTimeout(fn, 1000)，其中的setTimeout就是异步过程的发起函数，fn是回调函数。用一句话概括：工作线程将消息放到消息队列，主线程通过事件循环过程去取消息。

消息队列：消息队列是一个先进先出的队列，它里面存放着各种消息。

事件循环：事件循环是指主线程重复从消息队列中取消息、执行的过程。



**流程如下:**

1. 主线程读取js代码, 形成相应的堆和执行栈, 执行同步任务
2. 当主线程遇到异步任务,，指定给异步进程处理, 同时继续执行同步任务
3. 当异步进程处理完毕后， 将相应的异步任务推入到任务队列首部
4. 主线程任务处理完毕后,，查询任务队列，则取出一个任务队列推入到主线程的执行栈
5. 重复执行第2、3、4步，这就称为事件循环

**JS-Web-API 知识点与高频考题解析**

**BOM**

BOM（浏览器对象模型）是浏览器本身的一些信息的设置和获取，例如获取浏览器的宽度、高度，设置让浏览器跳转到哪个地址。

navigator： 获取浏览器特性（即俗称的UA）然后识别客户端

location： 获取网址、协议、path、参数、hash 等

history： 操作浏览器的历史纪录，（前进，后退等功能）

**1，什么是window对象? 什么是document对象?**

window:它是一个顶层对象,而不是另一个对象的属性，即浏览器的窗口。

document:代表整个HTML 文档,可用来访问页面中的所有元素

Window 对象表示当前浏览器的窗口，是JavaScript的顶级对象。我们创建的所有对象、函数、变量都是 Window 对象的成员。

Window 对象的方法和属性是在全局范围内有效的。

Document 对象是 HTML 文档的根节点与所有其他节点（元素节点，文本节点，属性节点, 注释节点）

Document 对象使我们可以通过脚本对 HTML 页面中的所有元素进行访问

Document 对象是 Window 对象的一部分，可通过 window.document 属性对其进行访问

**2，事件是？IE与火狐的事件机制有什么区别？ 如何阻止冒泡？**

1. 我们在网页中的某个操作（有的操作对应多个事件）。例如：当我们点击一个按钮就会产生一个事件。是可以被 JavaScript 侦测到的行为。

2. 事件处理机制：IE是事件冒泡、Firefox同时支持两种事件模型，也就是：捕获型事件和冒泡型事件；

3. ev.stopPropagation();（旧ie的方法 ev.cancelBubble = true;）

**3，解释一下事件代理**

事件代理的原理其实就和作用域链的原理差不多，但是事件代理是利用事件的冒泡原理来实现的，事件代理就是通过给祖先元素添加事件，通过事件目标对象开始向上查找找到匹配的子节点为止，如果找不到则到绑定事件的那个祖先元素为止，找到了就触发事件，并且可以通过js中call和apply来改变触发事件函数中的this为当前绑定节点，也是通过一层一层逐层向上的方式进行匹配查找而触发对应事件，好处就是可以使后添加的dom元素也同样有之前存在元素的事件，jquery中可以使用on，delegate，live实现的，不过在jquery1.7版本以后吧live给废除了，原因就是live绑定事件的祖先元素是整个html页面的根节点，所以性能消耗比较大，在后边的版本中给删除了，使用on，delegate代替

**优点：**

使代码简洁

减少浏览器的内存占用

缺点： 使用不当会造成事件在不应该触发时触发

function bindEvent(elem, type, selector, fn) {

    // 这样处理，可接收两种调用方式 bindEvent(div1, 'click', 'a', function () {...}) 和 bindEvent(div1, 'click', function () {...}) 这两种

    if (fn == null) {

        fn = selector

        selector = null

    }

    // 绑定事件

    elem.addEventListener(type, function (e) {

        var target

        if (selector) {

            // 有 selector 说明需要做事件代理

            // 获取触发时间的元素，即 e.target

            target = e.target

            // 看是否符合 selector 这个条件

            if (target.matches(selector)) {

                fn.call(target, e)

            }

        } else {

            // 无 selector ，说明不需要事件代理

            fn(e)

        }

    })

}

// 使用代理，bindEvent 多一个 'a' 参数

var div1 = document.getElementById('div1')

bindEvent(div1, 'click', 'a', function (e) {

    console.log(this.innerHTML)

})

// 不使用代理

var a = document.getElementById('a1')

bindEvent(div1, 'click', function (e) {

    console.log(a.innerHTML)

})

**4，offsetWidth/offsetHeight,clientWidth/clientHeight与scrollWidth/scrollHeight的区别，**

offsetWidth/offsetHeight返回值包含content + padding + border，效果与e.getBoundingClientRect()相同

clientWidth/clientHeight返回值只包含content + padding，如果有滚动条，也不包含滚动条

scrollWidth/scrollHeight返回值包含content + padding + 溢出内容的尺寸

**5，focus/blur与focusin/focusout的区别与联系**

focus/blur不冒泡，focusin/focusout冒泡

focus/blur兼容性好，focusin/focusout在除FireFox外的浏览器下都保持良好兼容性，如需使用事件托管，可考虑在FireFox下使用事件捕获elem.addEventListener('focus', handler, true)

可获得焦点的元素：

window

链接被点击或键盘操作

表单空间被点击或键盘操作

设置tabindex属性的元素被点击或键盘操作

**6，mouseover/mouseout与mouseenter/mouseleave的区别与联系**

mouseover/mouseout是标准事件，所有浏览器都支持；mouseenter/mouseleave是IE5.5引入的特有事件后来被DOM3标准采纳，现代标准浏览器也支持

mouseover/mouseout是冒泡事件；mouseenter/mouseleave不冒泡。需要为多个元素监听鼠标移入/出事件时，推荐mouseover/mouseout托管，提高性能

标准事件模型中event.target表示发生移入/出的元素,vent.relatedTarget对应移出/如元素；在老IE中event.srcElement表示发生移入/出的元素，event.toElement表示移出的目标元素，event.fromElement表示移入时的来源元素

**7，介绍DOM0，DOM2，DOM3事件处理方式区别**

DOM0级事件处理方式：

   btn.onclick = func;

   btn.onclick = null;

DOM2级事件处理方式：

   btn.addEventListener('click', func, false);

   btn.removeEventListener('click', func, false);

   btn.attachEvent("onclick", func);

   btn.detachEvent("onclick", func);

DOM3级事件处理方式：

   eventUtil.addListener(input, "textInput", func);

   eventUtil 是自定义对象，textInput 是DOM3级事件

**8，事件的三个阶段**

捕获、目标、冒泡

js的冒泡(Bubbling Event)和捕获(Capture Event)的区别

冒泡型事件：事件按照从最特定的事件目标到最不特定的事件目标(document对象)的顺序触发。

捕获型事件(event capturing)：事件从最不精确的对象(document 对象)开始触发，然后到最精确(也可以在窗口级别捕获事件，不过必须由开发人员特别指定)。

DOM事件流：同时支持两种事件模型：捕获型事件和冒泡型事件，但是，捕获型事件先发生。两种事件流会触及DOM中的所有对象，从document对象开始，也在document对象结束。

事件捕获

当你使用事件捕获时，父级元素先触发，子级元素后触发，即div先触发，p后触发。

事件冒泡

当你使用事件冒泡时，子级元素先触发，父级元素后触发，即p先触发，div后触发。

阻止冒泡

• 在W3c中，使用stopPropagation（）方法

• 在IE下设置cancelBubble = true；

在捕获的过程中stopPropagation（）；后，后面的冒泡过程也不会发生了。

阻止捕获

阻止事件的默认行为，例如click <a>后的跳转

• 在W3c中，使用preventDefault（）方法；

• 在IE下设置window.event.returnValue = false;

**9，介绍事件“捕获”和“冒泡”执行顺序和事件的执行次数？**

按照W3C标准的事件：首是进入捕获阶段，直到达到目标元素，再进入冒泡阶段

事件执行次数（DOM2-addEventListener）：元素上绑定事件的个数

注意1：前提是事件被确实触发

注意2：事件绑定几次就算几个事件，即使类型和功能完全一样也不会“覆盖”

事件执行顺序：判断的关键是否目标元素

非目标元素：根据W3C的标准执行：捕获->目标元素->冒泡（不依据事件绑定顺序）

目标元素：依据事件绑定顺序：先绑定的事件先执行（不依据捕获冒泡标准）

最终顺序：父元素捕获->目标元素事件1->目标元素事件2->子元素捕获->子元素冒泡->父元素冒泡

注意：子元素事件执行前提 事件确实“落”到子元素布局区域上，而不是简单的具有嵌套关系

在一个DOM上同时绑定两个点击事件：一个用捕获，一个用冒泡。事件会执行几次，先执行冒泡还是捕获？

该DOM上的事件如果被触发，会执行两次（执行次数等于绑定次数）

如果该DOM是目标元素，则按事件绑定顺序执行，不区分冒泡/捕获

如果该DOM是处于事件流中的非目标元素，则先执行捕获，后执行冒泡

**10，window.onload 和 document.DOMContentLoaded  (注：$(document).ready())  的区别？**

一般情况下，DOMContentLoaded事件要在window.onload之前执行，当DOM树构建完成的时候就会执行DOMContentLoaded事件，而window.onload是在页面载入完成的时候，才执行，这其中包括图片等元素。大多数时候我们只是想在DOM树构建完成后，绑定事件到元素，我们并不需要图片元素，加上有时候加载外域图片的速度非常缓慢。

**DOM**

讲 DOM 先从 HTML 讲起，讲 HTML 先从 XML 讲起。XML 是一种可扩展的标记语言，所谓可扩展就是它可以描述任何结构化的数据，它是一棵树！

**1，documen.write和 innerHTML的区别**

document.write只能重绘整个页面

innerHTML可以重绘页面的一部分

**2，DOM操作——怎样添加、移除、移动、复制、创建和查找节点?**

1）创建新节点

createDocumentFragment()    //创建一个DOM片段

createElement()   //创建一个具体的元素

createTextNode()   //创建一个文本节点

2）添加、移除、替换、插入

appendChild()

removeChild()

replaceChild()

insertBefore() //在已有的子节点前插入一个新的子节点

3）查找

getElementsByTagName()    //通过标签名称

getElementsByName()    //通过元素的Name属性的值(IE容错能力较强，会得到一个数组，其中包括id等于name值的)

getElementById()    //通过元素Id，唯一性

**3，attribute和property的区别是什么？**

attribute是dom元素在文档中作为html标签拥有的属性；

property就是dom元素在js中作为对象拥有的属性。

所以：

对于html的标准属性来说，attribute和property是同步的，是会自动更新的，

但是对于自定义的属性来说，他们是不同步的，

**4，src和href的区别**

src用于替换当前元素，href用于在当前文档和引用资源之间确立联系。

src是source的缩写，指向外部资源的位置，指向的内容将会嵌入到文档中当前标签所在位置；在请求src资源时会将其指向的资源下载并应用到文档内，当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等元素也如此，类似于将所指向资源嵌入当前标签内。这也是为什么将js脚本放在底部而不是头部。

Src source,指向外部资源的位置,如果我们添加<script src ="js.js"></script>浏览器会暂停其他资源的下载和处理,直到该资源加载,编译,执行完毕(图片和框架也是如此),这也就是为什么js脚本要放在底部。

src用于替换当前元素,href用于在当前文档和引入资源之间建立联系。

**对象**

**1，JavaScript继承的几种实现方式？**

1，构造函数继承，使用call和apply两个方法的特性可以实现，改变方法中的this

2，原型链继承

3，组合式继承

**2，javascript创建对象的几种方式？**

javascript创建对象简单的说,无非就是使用内置对象或各种自定义对象，当然还可以用JSON；但写法有很多种，也能混合使用。

1、对象字面量的方式       person={firstname:"Mark",lastname:"Yun",age:25,eyecolor:"black"};

2、用function来模拟无参的构造函数

           function Person(){}

                       var person=new Person();//定义一个function，如果使用new"实例化",该function可以看作是一个Class

                       person.name=“Mark";

                       person.age="25";

                       person.work=function(){

                       alert(person.name+" hello...");

           }

           person.work();

3、用function来模拟参构造函数来实现（用this关键字定义构造的上下文属性）

           function Pet(name,age,hobby){

                       this.name=name;//this作用域：当前对象

                       this.age=age;

                       this.hobby=hobby;

                       this.eat=function(){

                                   alert("我叫"+this.name+",我喜欢"+this.hobby+",是个程序员");

                        }

            }

           var maidou =new Pet("麦兜",25,"coding");//实例化、创建对象

           maidou.eat();//调用eat方法

4、用工厂方式来创建（内置对象）

                      var wcDog =new Object();

                      wcDog.name="旺财";

                      wcDog.age=3;

                      wcDog.work=function(){

                                   alert("我是"+wcDog.name+",汪汪汪......");

                        }

           wcDog.work();

5、用原型方式来创建

           function Dog(){ }

           Dog.prototype.name="旺财";

           Dog.prototype.eat=function(){alert(this.name+"是个吃货");}

           var wangcai =new Dog();

           wangcai.eat();

5、用混合方式来创建

           function Car(name,price){

                       this.name=name;

                       this.price=price;

           }

           Car.prototype.sell=function(){

                       alert("我是"+this.name+"，我现在卖"+this.price+"万元");

            }

            var camry =new Car("凯美瑞",27);

           camry.sell();

**3，谈谈This对象的理解。**

this分为几个不同的使用场景，在function中this指的的是window，如果是实用new 调用的话this指的是当前的实例化对象，在事件调用函数中this指的调用事件的window特殊的是在IE中的attachEvent中的this总是指向全局对象Window；，在定时器中this指的是window，在es6中有一个箭头函数，在箭头函数中this永远指向的是父级对象，this也是可以改变的，在js中call, apply, bind都可以改变this的指向， call, apply都是执行一个函数并且改变this，区别就是参数传递不一样，而bind是返回一个绑定this之后的新函数

**4，javascript 代码中的"use strict";是什么意思 ? 使用它区别是什么？**

use strict是一种ECMAscript 5 添加的（严格）运行模式,这种模式使得 Javascript 在更严格的条件下运行,

使JS编码更加规范化的模式,消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为。

默认支持的糟糕特性都会被禁用，比如不能用with，也不能在意外的情况下给全局变量赋值;

全局变量的显示声明,函数必须声明在顶层，不允许在非函数代码块内声明函数,arguments.callee也不允许使用；

消除代码运行的一些不安全之处，保证代码运行的安全,限制函数中的arguments修改，严格模式下的eval函数的行为和非严格模式的也不相同;

提高编译器效率，增加运行速度；

为未来新版本的Javascript标准化做铺垫。

**5，JSON 的了解？**

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。

它是基于JavaScript的一个子集。数据格式简单, 易于读写, 占用带宽小

如：{"age":"12", "name":"back"}

JSON字符串转换为JSON对象:

var obj =eval('('+ str +')');

var obj = str.parseJSON();

var obj = JSON.parse(str);

JSON对象转换为JSON字符串：

var last=obj.toJSONString();

var last=JSON.stringify(obj);

**6，.call() 和 .apply() 的区别？**

**7，什么是函数节流？介绍一下应用场景和原理？**

函数节流(throttle)是指阻止一个函数在很短时间间隔内连续调用。 只有当上一次函数执行后达到规定的时间间隔，才能进行下一次调用。 但要保证一个累计最小调用间隔（否则拖拽类的节流都将无连续效果）

函数节流用于 onresize, onscroll 等短时间内会多次触发的事件

函数节流的原理：使用定时器做时间节流。 当触发一个事件时，先用 setTimout 让这个事件延迟一小段时间再执行如果在这个时间间隔内又触发了事件，就 clearTimeout 原来的定时器， 再 setTimeout 一个新的定时器重复以上流程。

。

函数节流简单实现：

function throttle(method, context) {

    clearTimeout(methor.tId);

    method.tId = setTimeout(function(){

        method.call(context);

    }， 100); // 两次调用至少间隔 100ms

}

// 调用

window.onresize = function(){

   throttle(myFunc, window);

}

**8，new 操作符具体干了什么？**

创建实例对象，this 变量引用该对象，同时还继承了构造函数的原型

属性和方法被加入到 this 引用的对象中

新创建的对象由 this 所引用，并且最后隐式的返回 this

new共经历了四个过程。

var fn = function () { };

var fnObj = new fn();

1、创建了一个空对象

var obj = new object();

2、设置原型链

obj.\_proto\_ = fn.prototype;

3、让fn的this指向obj，并执行fn的函数体

var result = fn.call(obj);

4、判断fn的返回值类型，如果是值类型，返回obj。如果是引用类型，就返回这个引用类型的对象。

if (typeof(result) == "object"){

fnObj = result;

} else {

fnObj = obj;}

**兼容与优化**

**1，页面重构怎么操作？**

网站重构：在不改变外部行为的前提下，简化结构、添加可读性，而在网站前端保持一致的行为。

也就是说是在不改变UI的情况下，对网站进行优化，在扩展的同时保持一致的UI。

对于传统的网站来说重构通常是：

表格(table)布局改为DIV+CSS

使网站前端兼容于现代浏览器(针对于不合规范的CSS、如对IE6有效的)

对于移动平台的优化

针对于SEO进行优化

深层次的网站重构应该考虑的方面

减少代码间的耦合

让代码保持弹性

严格按规范编写代码

设计可扩展的API

代替旧有的框架、语言(如VB)

增强用户体验

通常来说对于速度的优化也包含在重构中

压缩JS、CSS、image等前端资源(通常是由服务器来解决)

程序的性能优化(如数据读写)

采用CDN来加速资源加载

对于JS DOM的优化

HTTP服务器的文件缓存

**2，列举IE与其他浏览器不一样的特性？**

1、事件不同之处：

   1-1，触发事件的元素被认为是目标（target）。而在 IE 中，目标包含在 event 对象的 srcElement 属性；

   1-2，获取字符代码、如果按键代表一个字符（shift、ctrl、alt除外），IE 的 keyCode 会返回字符代码（Unicode），DOM 中按键的代码和字符是分离的，要获取字符代码，需要使用 charCode 属性；

   1-3，阻止某个事件的默认行为，IE 中阻止某个事件的默认行为，必须将 returnValue 属性设置为 false，Mozilla 中，需要调用 preventDefault() 方法；

   1-4，停止事件冒泡，IE 中阻止事件进一步冒泡，需要设置 cancelBubble 为 true，Mozzilla 中，需要调用 stopPropagation()；

**3，什么叫优雅降级和渐进增强？**

优雅降级：Web站点在所有新式浏览器中都能正常工作，如果用户使用的是老式浏览器，则代码会针对旧版本的IE进行降级处理了,使之在旧式浏览器上以某种形式降级体验却不至于完全不能用。

如：border-shadow

渐进增强：从被所有浏览器支持的基本功能开始，逐步地添加那些只有新版本浏览器才支持的功能,向页面增加不影响基础浏览器的额外样式和功能的。当浏览器支持时，它们会自动地呈现出来并发挥作用。

如：默认使用flash上传，但如果浏览器支持 HTML5 的文件上传功能，则使用HTML5实现更好的体验；

**4，说说严格模式的限制**

严格模式主要有以下限制：

变量必须声明后再使用

函数的参数不能有同名属性，否则报错

不能使用with语句

不能对只读属性赋值，否则报错

不能使用前缀0表示八进制数，否则报错

不能删除不可删除的属性，否则报错

不能删除变量delete prop，会报错，只能删除属性delete global[prop]

eval不会在它的外层作用域引入变量

eval和arguments不能被重新赋值

arguments不会自动反映函数参数的变化

不能使用arguments.callee

不能使用arguments.caller

禁止this指向全局对象

不能使用fn.caller和fn.arguments获取函数调用的堆栈

增加了保留字（比如protected、static和interface）

设立"严格模式"的目的，主要有以下几个：

消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;

消除代码运行的一些不安全之处，保证代码运行的安全；

提高编译器效率，增加运行速度；

为未来新版本的Javascript做好铺垫。

注：经过测试IE6,7,8,9均不支持严格模式。

**5，检测浏览器版本版本有哪些方式？**

根据 navigator.userAgent // UA.toLowerCase().indexOf('chrome')

根据 window 对象的成员 // 'ActiveXObject' in window

**6，总结前端性能优化的解决方案**

**优化原则和方向**

性能优化的原则是以更好的用户体验为标准，具体就是实现下面的目标：

多使用内存、缓存或者其他方法

减少 CPU 和GPU 计算，更快展现

**优化的方向有两个：**

减少页面体积，提升网络加载

优化页面渲染

**减少页面体积，提升网络加载**

静态资源的压缩合并（JS 代码压缩合并、CSS 代码压缩合并、雪碧图）

静态资源缓存（资源名称加 MD5 戳）

使用 CDN 让资源加载更快

**优化页面渲染**

CSS 放前面，JS 放后面

懒加载（图片懒加载、下拉加载更多）

减少DOM 查询，对 DOM 查询做缓存

减少DOM 操作，多个操作尽量合并在一起执行（DocumentFragment）

事件节流

尽早执行操作（DOMContentLoaded）

使用 SSR 后端渲染，数据直接输出到 HTML 中，减少浏览器使用 JS 模板渲染页面 HTML 的时间

**7，图片懒加载与预加载**

图片懒加载的原理就是暂时不设置图片的src属性，而是将图片的url隐藏起来，比如先写在data-src里面，等某些事件触发的时候(比如滚动到底部，点击加载图片)再将图片真实的url放进src属性里面，从而实现图片的延迟加载

图片预加载是指在一些需要展示大量图片的网站，实现图片的提前加载。从而提升用户体验。常用的方式有两种，一种是隐藏在css的background的url属性里面，一种是通过javascript的Image对象设置实例对象的src属性实现图片的预加载。相关代码如下：

CSS预加载图片方式：

#preload-01 { background: url(<http://domain.tld/image-01.png>) no-repeat -9999px -9999px; }

#preload-02 { background: url(<http://domain.tld/image-02.png>) no-repeat -9999px -9999px; }

#preload-03 { background: url(<http://domain.tld/image-03.png>) no-repeat -9999px -9999px; }

Javascript预加载图片的方式：

function preloadImg(url) {

   var img = new Image();

   img.src = url;

   if([img.complete](http://img.complete/)) {

       //接下来可以使用图片了

       //do something here

   } else {

       img.onload = function() {

           //接下来可以使用图片了

           //do something here

       };

   }

}

**5，描述浏览器的渲染过程，DOM树和渲染树的区别？**

浏览器的渲染过程：

解析HTML构建 DOM(DOM树)，并行请求 css/image/js

CSS 文件下载完成，开始构建 CSSOM(CSS树)

CSSOM 构建结束后，和 DOM 一起生成 Render Tree(渲染树)

布局(Layout)：计算出每个节点在屏幕中的位置

显示(Painting)：通过显卡把页面画到屏幕上

DOM树 和 渲染树 的区别：

DOM树与HTML标签一一对应，包括head和隐藏元素

渲染树不包括head和隐藏元素，大段文本的每一个行都是独立节点，每一个节点都有对应的css属性

**7，重绘和回流（重排）的区别和关系？**

重绘：当渲染树中的元素外观（如：颜色）发生改变，不影响布局时，产生重绘

回流：当渲染树中的元素的布局（如：尺寸、位置、隐藏/状态状态）发生改变时，产生重绘回流

注意：JS获取Layout属性值（如：offsetLeft、scrollTop、getComputedStyle等）也会引起回流。因为浏览器需要通过回流计算最新值

回流必将引起重绘，而重绘不一定会引起回流

**8，如何最小化重绘(repaint)和回流(reflow)？**

需要要对元素进行复杂的操作时，可以先隐藏(display:"none")，操作完成后再显示

需要创建多个DOM节点时，使用DocumentFragment创建完后一次性的加入document

缓存Layout属性值，如：var left = elem.offsetLeft; 这样，多次使用 left 只产生一次回流

尽量避免用table布局（table元素一旦触发回流就会导致table里所有的其它元素回流）

避免使用css表达式(expression)，因为每次调用都会重新计算值（包括加载页面）

尽量使用 css 属性简写，如：用 border 代替 border-width, border-style, border-color

批量修改元素样式：elem.className 和 elem.style.cssText 代替 elem.style.xxx

**9，script 的位置是否会影响首屏显示时间？**

在解析 HTML 生成 DOM 过程中，js 文件的下载是并行的，不需要 DOM 处理到 script 节点。因此，script的位置不影响首屏显示的开始时间。

浏览器解析 HTML 是自上而下的线性过程，script作为 HTML 的一部分同样遵循这个原则

因此，script 会延迟 DomContentLoad，只显示其上部分首屏内容，从而影响首屏显示的完成时间

**存储**

**cookie**

cookie 本身不是用来做服务器端存储的（计算机领域有很多这种“狗拿耗子”的例子，例如 CSS 中的 float），它是设计用来在服务器和客户端进行信息传递的，因此我们的每个 HTTP 请求都带着 cookie。但是 cookie 也具备浏览器端存储的能力（例如记住用户名和密码），因此就被开发者用上了。

使用起来也非常简单，document.cookie = ....即可。

但是 cookie 有它致命的缺点：

存储量太小，只有 4KB

所有 HTTP 请求都带着，会影响获取资源的效率

API 简单，需要封装才能用

**locationStorage 和 sessionStorage**

后来，HTML5 标准就带来了sessionStorage和localStorage，先拿localStorage来说，它是专门为了浏览器端缓存而设计的。其优点有：

存储量增大到 5MB

不会带到 HTTP 请求中

API 适用于数据存储 localStorage.setItem(key, value) localStorage.getItem(key)

sessionStorage的区别就在于它是根据 session 过去时间而实现，而localStorage会永久有效，应用场景不同。例如，一些需要及时失效的重要信息放在sessionStorage中，一些不重要但是不经常设置的信息，放在localStorage中。

**es6/7**

**1，说说对es6的理解（说一下es6，知道es6吗）**

语法糖(箭头函数，类的定义，继承)，以及一些新的扩展（数组，字符串，对象，方法等），对作用域的重新定义，以及异步编程的解决方案（promise，async，await）、解构赋值的出现

**2，ES6常用特性**

变量定义(let和const,可变与不可变，const定义对象的特殊情况)

解构赋值

模板字符串

数组新API(例：Array.from(),entries(),values(),keys())

箭头函数(rest参数，扩展运算符，::绑定this)

Set和Map数据结构(set实例成员值唯一存储key值，map实例存储键值对(key-value))

Promise对象(前端异步解决方案进化史，generator函数，async函数)

Class语法糖(super关键字)

**3，说说你对Promise的理解**

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件监听——更合理和更强大。Promise 有三种状态：pending（进行中）、fulfilled（已成功）和rejected（已失败）。但是无法获取到pending状态，在promise中接受两个内置参数分别是resolve（成功）和reject（失败），Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数。then方法可以传递两个回调函数第一个是成功，第二个是失败，失败回调也可以使用promise的catch方法回调，promise还有一个强大的功能那就是all方法可以组合多个promise实例，包装成一个新的 Promise 实例。

**4，介绍一下async和await；**

async 会将其后的函数（函数表达式或 Lambda）的返回值封装成一个 Promise 对象，而 await 会等待这个 Promise 完成，并将其 resolve 的结果返回出来。

async / await是ES7的重要特性之一，也是目前社区里公认的优秀异步解决方案。目前async / await 在 IE edge中已经可以直接使用了，但是chrome和Node.js还没有支持。幸运的是，babel已经支持async的transform了，所以我们使用的时候引入babel就行。在开始之前我们需要引入以下的package，preset-stage-3里就有我们需要的async/await的编译文件。

**5，es6中的Module**

ES6 中模块化语法更加简洁，使用export抛出，使用import from 接收，

如果只是输出一个唯一的对象，使用export default即可

// 创建 util1.js 文件，内容如

export default {

    a: 100

}

// 创建 index.js 文件，内容如

import obj from './util1.js’

如果想要输出许多个对象，就不能用default了，且import时候要加{...}，代码如下

// 创建 util2.js 文件，内容如

export function fn1() {

    alert('fn1')

}

export function fn2() {

    alert('fn2')

}

// 创建 index.js 文件，内容如

import { fn1, fn2 } from './util2.js’

**6，ES6 class 和普通构造函数的区别**

class 其实一直是 JS 的关键字（保留字），但是一直没有正式使用，直到 ES6 。 ES6 的 class 就是取代之前构造函数初始化对象的形式，从语法上更加符合面向对象的写法

1. class 是一种新的语法形式，是class Name {...}这种形式，和函数的写法完全不一样
2. 两者对比，构造函数函数体的内容要放在 class 中的constructor函数中，constructor即构造器，初始化实例时默认执行
3. class 中函数的写法是add() {...}这种形式，并没有function关键字

而且使用 class 来实现继承就更加简单了

在class中直接extends关键字就可以实现继承，而不像之前的继承实现有多种不同的实现方式，在es6中就只有一种

注意以下两点：

使用extends即可实现继承，更加符合经典面向对象语言的写法，如 Java

子类的constructor一定要执行super()，以调用父类的constructor

**7，ES6 中新增的数据类型有哪些？**

Set 和 Map 都是 ES6 中新增的数据结构，是对当前 JS 数组和对象这两种重要数据结构的扩展。由于是新增的数据结构

1. Set 类似于数组，但数组可以允许元素重复，Set 不允许元素重复
2. Map 类似于对象，但普通对象的 key 必须是字符串或者数字，而 Map 的 key 可以是任何数据类型

**8，箭头函数的作用域上下文和 普通函数作用域上下文 的区别**

箭头函数其实只是一个密名函数的语法糖，区别在于普通函数作用域中的this有特定的指向，一般指向window，而箭头函数中的this只有一个指向那就是指当前函数所在的对象，其实现原理其实就是类似于之前编程的时候在函数外围定义that一样，用了箭头函数就不用定义that了直接使用this

**9.es6如何转为es5？**

使用Babel 转码器，Babel 的配置文件是.babelrc，存放在项目的根目录下。使用 Babel 的第一步，就是配置这个文件。

**算法**

**1，浅拷贝vs深拷贝**

拷贝其实就是对象复制，为了解决对象复制是产生的引用类型问题

浅拷贝：利用迭代器，循环对象将对象中的所有可枚举属性复制到另一个对象上，但是浅拷贝的有一个问题就是只是拷贝了对象的一级，其他级还如果是引用类型的值的话依旧解决不了

深拷贝：深拷贝解决了浅拷贝的问题，利用递归的形势便利对象的每一级，实现起来较为复杂，得判断值是数组还是对象，简单的说就是，在内存中存在两个数据结构完全相同又相互独立的数据，将引用型类型进行复制，而不是只复制其引用关系。

**2，常见的几种数组排序算法JS实现**

1，快速排序

从给定的数据中，随机抽出一项，这项的左边放所有比它小的，右边放比它大的，然后再分别这两边执行上述操作，采用的是递归的思想，总结出来就是　实现一层，分别给两边递归，设置好出口

function fastSort(array,head,tail){

   //考虑到给每个分区操作的时候都是在原有的数组中进行操作的，所以这里head,tail来确定分片的位置

   /\*生成随机项\*/

   var randomnum = Math.floor(ranDom(head,tail));

   var random = array[randomnum];

   /\*将小于random的项放置在其左边  策略就是通过一个临时的数组来储存分好区的结果，再到原数组中替换\*/

   var arrayTemp = [];

   var unshiftHead = 0;

   for(var i = head;i <= tail;i++){

     if(array[i]<random){

       arrayTemp.unshift(array[i]);

       unshiftHead++;

     }else if(array[i]>random){

       arrayTemp.push(array[i]);

     }

     /\*当它等于的时候放哪，这里我想选择放到队列的前面，也就是从unshift后的第一个位置放置\*/

     if(array[i]===random){

       arrayTemp.splice(unshiftHead,0,array[i]);

     }

   }

   /\*将对应项覆盖原来的记录\*/

   for(var j = head , u=0;j <= tail;j++,u++){

     array.splice(j,1,arrayTemp[u]);

   }

   /\*寻找中间项所在的index\*/

   var nowIndex = array.indexOf(random);

   /\*设置出口，当要放进去的片段只有2项的时候就可以收工了\*/

   if(arrayTemp.length <= 2){

     return;

   }

   /\*递归，同时应用其左右两个区域\*/

   fastSort(array,head,nowIndex);

   fastSort(array,nowIndex+1,tail);

}

2，插入排序

思想就是在已经排好序的数组中插入到相应的位置，以从小到大排序为例，扫描已经排好序的片段的每一项，如大于，则继续往后，直到他小于一项时，将其插入到这项的前面

function insertSort(array){

   /\*start根据已排列好的项数决定\*/

   var start=1;

   /\*按顺序，每一项检查已排列好的序列\*/

   for(var i=start; i<array.length; start++,i++){

     /\*跟已排好序的序列做对比，并插入到合适的位置\*/

     for(var j=0; j<start; j++){

       /\*小于或者等于时（我们是升序）插入到该项前面\*/

       if(array[i]<=array[j]){

         console.log(array[i]+' '+array[j]);

         array.splice(j,0,array[i]);

         /\*删除原有项\*/

         array.splice(i+1,1);

         break;

       }

     }

   }

}

3，冒泡排序

故名思意 ，就是一个个冒泡到最前端或者最后端，主要是通过两两依次比较，以升序为例，如果前一项比后一项大则交换顺序，一直比到最后一对

function bubbleSort(array){

   /\*给每个未确定的位置做循环\*/

   for(var unfix=array.length-1; unfix>0; unfix--){

     /\*给进度做个记录，比到未确定位置\*/

     for(var i=0; i<unfix;i++){

       if(array[i]>array[i+1]){

         var temp = array[i];

         array.splice(i,1,array[i+1]);

         array.splice(i+1,1,temp);

       }

     }

   }

 }

4，选择排序

将当前未确定块的min或者max取出来插到最前面或者后面

function selectSort(array){

   /\*给每个插入后的未确定的范围循环，初始是从0开始\*/

   for(var unfixed=0; unfixed<array.length; unfixed++){

     /\*设置当前范围的最小值和其索引\*/

     var min = array[unfixed];

     var minIndex = unfixed;

     /\*在该范围内选出最小值\*/

     for(var j=unfixed+1; j<array.length; j++){

       if(min>array[j]){

         min = array[j];

         minIndex = j;

       }

     }

     /\*将最小值插入到unfixed，并且把它所在的原有项替换成\*/

     array.splice(unfixed,0,min);

     array.splice(minIndex+1,1);

   }

 }

**3，写一个数组去重的方法**

/\*\* 方法一：

\* 1.构建一个新的数组存放结果

\* 2.for循环中每次从原数组中取出一个元素，用这个元素循环与结果数组对比

\* 3.若结果数组中没有该元素，则存到结果数组中

\* 缺陷：不能去重数组中得引用类型的值和NaN

\*/

function unique(array){

  var result = [];

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

    if(result.indexOf(array[i]) == -1) {

      result.push(array[i]);

    }

  }

  return result;

}

// [1,2,1,2,'1','2',0,'1','你好','1','你好',NaN,NaN] => [1, 2, "1", "2", 0, "你好",NaN,NaN]

// [{id: '1'}, {id: '1'}] => [{id: '1'}, {id: '1’}]

//方法二：ES6

Array.from(new Set(array))

// [1,2,1,2,'1','2',0,'1','你好','1','你好',NaN,NaN] => [1, 2, "1", "2", 0, "你好", NaN]

**4，说一下js模板引擎**

模板引擎原理总结起来就是：先获取html中对应的id下得innerHTML，利用开始标签和关闭标签进行字符串切分，其实是将模板划分成两部份内容，一部分是html部分，一部分是逻辑部分，通过区别一些特殊符号比如each、if等来将字符串拼接成函数式的字符串,将两部分各自经过处理后，再次拼接到一起，最后将拼接好的字符串采用new Function()的方式转化成所需要的函数。

常用的模版引擎主要有，Template.js，handlebars.js

**5，是否了解公钥加密和私钥加密。**

一般情况下是指私钥用于对数据进行签名，公钥用于对签名进行验证;

HTTP网站在浏览器端用公钥加密敏感数据，然后在服务器端再用私钥解密。

**6， js深度复制的方式**

1.使用jq的$.extend(true, target, obj)

2.newobj = Object.create(sourceObj)，// 但是这个是有个问题就是 newobj的更改不会影响到 sourceobj但是 sourceobj的更改会影响到newObj

3.newobj = JSON.parse(JSON.stringify(sourceObj))

**7，js设计模式**

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模

详细：<http://www.alloyteam.com/2012/10/common-javascript-design-patterns/>

8，获取当前网址url并从中截取信息。

9，数组去重，并手写代码

**10，图片懒加载与预加载？**

1，图片懒加载的原理就是暂时不设置图片的src属性，而是将图片的url隐藏起来，比如先写在data-src里面，等某些事件触发的时候(比如滚动到底部，点击加载图片)再将图片真实的url放进src属性里面，从而实现图片的延迟加载

Javascript预加载图片的方式：

function preloadImg(url) {

    var img = new Image();

    img.src = url;

    if([img.complete](http://img.complete/)) {

        //接下来可以使用图片了

        //do something here

    } else {

        img.onload = function() {

            //接下来可以使用图片了

            //do something here

        };

    }

}

2，图片预加载，是指在一些需要展示大量图片的网站，实现图片的提前加载。从而提升用户体验。常用的方式有两种，一种是隐藏在css的background的url属性里面，一种是通过javascript的Image对象设置实例对象的src属性实现图片的预加载。

CSS预加载图片方式：

#preload-01 { background: url(<http://domain.tld/image-01.png>) no-repeat -9999px -9999px; }

#preload-02 { background: url(<http://domain.tld/image-02.png>) no-repeat -9999px -9999px; }

#preload-03 { background: url(<http://domain.tld/image-03.png>) no-repeat -9999px -9999px; }

11，实现一个弹窗插件，怎么实现

12，实现一个轮播效果

**13，统计字符串中次数最多字母**

function findMaxDuplicateChar(str) {

  if(str.length == 1) {

    return str;

  }

  var charObj = {};

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

    if(!charObj[str.charAt(i)]) {

      charObj[str.charAt(i)] = 1;

    } else {

      charObj[str.charAt(i)] += 1;

    }

  }

  var maxChar = '',

      maxValue = 1;

  for(var k in charObj) {

    if(charObj[k] >= maxValue) {

      maxChar = k;

      maxValue = charObj[k];

    }

  }

  return maxChar + '：' + maxValue;

}

**14，变态题目解析**

<https://juejin.im/entry/58adabd6da2f6054bd1f499f>

27，对Node的优点和缺点提出了自己的看法？

\*（优点）因为Node是基于事件驱动和无阻塞的，所以非常适合处理并发请求，因此构建在Node上的代理服务器相比其他技术实现（如Ruby）的服务器表现要好得多。此外，与Node代理服务器交互的客户端代码是由javascript语言编写的，因此客户端和服务器端都用同一种语言编写，这是非常美妙的事情。

\*（缺点）Node是一个相对新的开源项目，所以不太稳定，它总是一直在变，而且缺少足够多的第三方库支持。看起来，就像是Ruby/Rails当年的样子。

**模块化**

**1，commonjs?requirejs?AMD|CMD|UMD?**

1.CommonJS就是为JS的表现来制定规范，NodeJS是这种规范的实现，webpack 也是以CommonJS的形式来书写。因为js没有模块的功能，所以CommonJS应运而生。但它不能在浏览器中运行。 CommonJS定义的模块分为:{模块引用(require)} {模块定义(exports)} {模块标识(module)}

2.RequireJS 是一个JavaScript模块加载器。         RequireJS有两个主要方法(method): define()和require()。这两个方法基本上拥有相同的定义(declaration) 并且它们都知道如何加载的依赖关系，然后执行一个回调函数(callback function)。与require()不同的是， define()用来存储代码作为一个已命名的模块。 因此define()的回调函数需要有一个返回值作为这个模块定义。这些类似被定义的模块叫作AMD (Asynchronous Module Definition，异步模块定义)。

3.AMD 是 RequireJS 在推广过程中对模块定义的规范化产出 AMD异步加载模块。它的模块支持对象 函数 构造器 字符串 JSON等各种类型的模块。 适用AMD规范适用define方法定义模块。

4.CMD是SeaJS 在推广过程中对模块定义的规范化产出

AMD与CDM的区别：

（1）对于于依赖的模块，AMD 是提前执行(好像现在也可以延迟执行了)，CMD 是延迟执行。

（2）AMD 推崇依赖前置，CMD 推崇依赖就近。

（3）AMD 推崇复用接口，CMD 推崇单用接口。

（4）书写规范的差异。

5.umd是AMD和CommonJS的糅合。

AMD 浏览器第一的原则发展 异步加载模块。

CommonJS模块以服务器第一原则发展，选择同步加载，它的模块无需包装(unwrapped modules)。这迫使人们又想出另一个更通用的模式UMD ( Universal Module Definition ), 希望解决跨平台的解决方案。UMD先判断是否支持Node.js的模块( exports )是否存在，存在则使用Node.js模块模式。

**2，模块化的理解**

模块化的话其实主要就是对于js功能逻辑的划分，在js中我们一般都吧一个js文件定义成一个模块，模块主要的职责就是（封装实现，暴露接口，声明依赖）

**3，AMD和CMD的区别**

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。

CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible.

CMD 推崇依赖就近，AMD 推崇依赖前置。

AMD 的 API 默认是一个当多个用，CMD 的 API 严格区分，推崇职责单一。比如 AMD 里，require 分全局 require 和局部 require，都叫 require。CMD 里，没有全局 require，而是根据模块系统的完备性，提供 seajs.use 来实现模块系统的加载启动。CMD 里，每个 API 都简单纯粹。

**前端安全**

**1，XSS（Cross Site Scripting，跨站脚本攻击）**

这是前端最常见的攻击方式，很多大型网站（如 Facebook）都被 XSS 攻击过。

举一个例子，我在一个博客网站正常发表一篇文章，输入汉字、英文和图片，完全没有问题。但是如果我写的是恶意的 JS 脚本，例如获取到document.cookie然后传输到自己的服务器上，那我这篇博客的每一次浏览都会执行这个脚本，都会把访客 cookie 中的信息偷偷传递到我的服务器上来。

其实原理上就是黑客通过某种方式（发布文章、发布评论等）将一段特定的 JS 代码隐蔽地输入进去。然后别人再看这篇文章或者评论时，之前注入的这段 JS 代码就执行了。JS 代码一旦执行，那可就不受控制了，因为它跟网页原有的 JS 有同样的权限，例如可以获取 server 端数据、可以获取 cookie 等。于是，攻击就这样发生了。

XSS的危害

XSS 的危害相当大，如果页面可以随意执行别人不安全的 JS 代码，轻则会让页面错乱、功能缺失，重则会造成用户的信息泄露。

比如早些年社交网站经常爆出 XSS 蠕虫，通过发布的文章内插入 JS，用户访问了感染不安全 JS 注入的文章，会自动重新发布新的文章，这样的文章会通过推荐系统进入到每个用户的文章列表面前，很快就会造成大规模的感染。

还有利用获取 cookie 的方式，将 cookie 传入入侵者的服务器上，入侵者就可以模拟 cookie 登录网站，对用户的信息进行篡改。

XSS的预防

那么如何预防 XSS 攻击呢？—— 最根本的方式，就是对用户输入的内容进行验证和替换，需要替换的字符有：

& 替换为：&amp;

< 替换为：&lt;

> 替换为：&gt;

” 替换为：&quot;

‘ 替换为：&#x27;

/ 替换为：&#x2f;

替换了这些字符之后，黑客输入的攻击代码就会失效，XSS 攻击将不会轻易发生。

除此之外，还可以通过对 cookie 进行较强的控制，比如对敏感的 cookie 增加http-only限制，让 JS 获取不到 cookie 的内容。

**2，CSRF（Cross-site request forgery，跨站请求伪造）**

CSRF 是借用了当前操作者的权限来偷偷地完成某个操作，而不是拿到用户的信息。

例如，一个支付类网站，给他人转账的接口是<http://buy.com/pay?touid=999&money=100>，而这个接口在使用时没有任何密码或者 token 的验证，只要打开访问就直接给他人转账。一个用户已经登录了[http://buy.com](http://buy.com/)，在选择商品时，突然收到一封邮件，而这封邮件正文有这么一行代码<img src="<http://buy.com/pay?touid=999&money=100>"/>，他访问了邮件之后，其实就已经完成了购买。

CSRF 的发生其实是借助了一个 cookie 的特性。我们知道，登录了[http://buy.com](http://buy.com/)之后，cookie 就会有登录过的标记了，此时请求<http://buy.com/pay?touid=999&money=100>是会带着 cookie 的，因此 server 端就知道已经登录了。而如果在[http://buy.com](http://buy.com/)去请求其他域名的 API 例如<http://abc.com/api>时，是不会带 cookie 的，这是浏览器的同源策略的限制。但是 —— 此时在其他域名的页面中，请求<http://buy.com/pay?touid=999&money=100>，会带着[buy.com](http://buy.com/)的 cookie ，这是发生 CSRF 攻击的理论基础。

预防 CSRF 就是加入各个层级的权限验证，例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者指纹才行。除此之外，敏感的接口使用POST请求而不是GET也是很重要的。