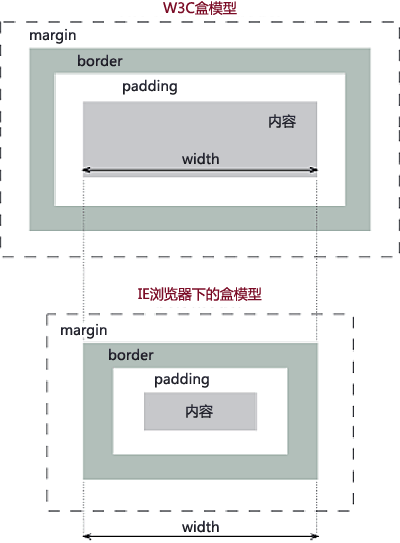
# HTML

1. 简述盒模型

盒模型分为两种：W3C盒模型和IE盒模型

IE盒模型的问题只会出现在IE5.5及其更早的版本，IE6+的版本在标准兼容模式下使用的是W3C的盒模型。

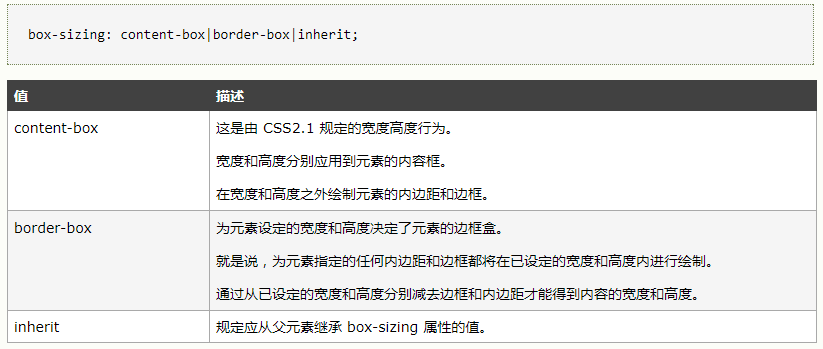
对于两种盒模型如图所示：



W3C标准盒模型：属性width、height只包含内容content，即width/height = content

IE盒模型：属性width、height包含content、border、padding，即width/height = content+border+padding

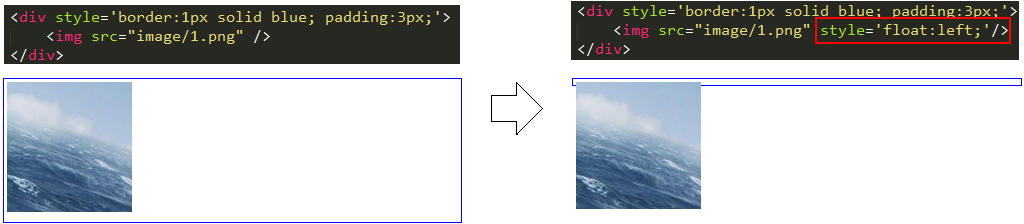
在IE8+、chrome、firefox等新增了属性box-sizing，该属性默认值为content-box，即标准盒模型；可将属性设置为border-box，即IE和模型。



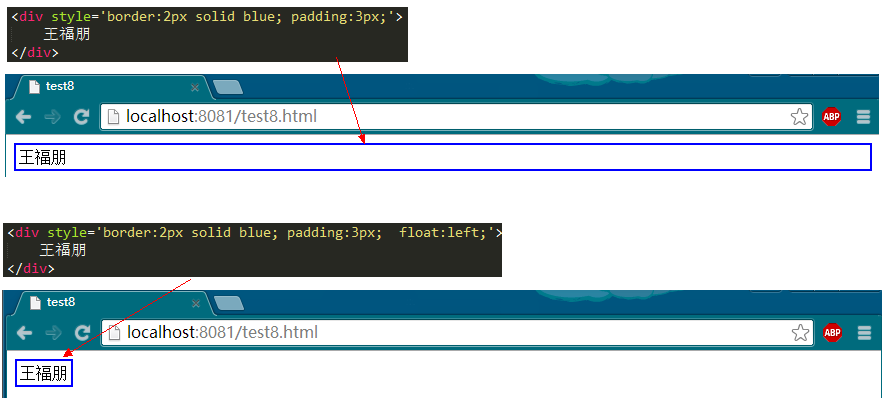
1. 对于浮动的理解，清除浮动的方法及原理？

float设置的初衷是用于——文字环绕效果

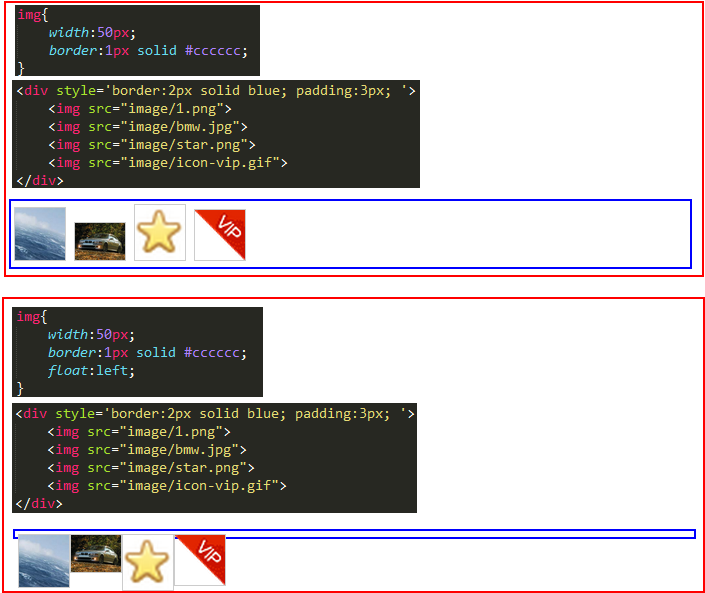
1. 破坏性。float破坏了父标签的原本结构，使得父标签出现了坍塌现象。导致这一现象的最根本原因在于：被设置了float的元素会脱离文档流。（其实float只所以被设置成具有破坏性，脱离文档流就是它的设置初衷——文字环绕效果）



1. 包裹性。div设置为float之后，其宽度会自动调整为包裹住内容宽度，而不是撑满整个父容器。



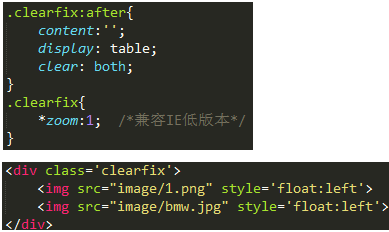
1. 清空格。



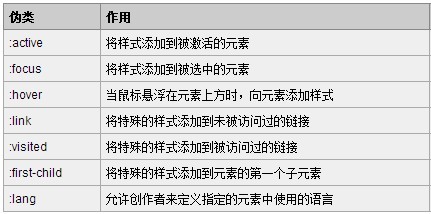
清除浮动与闭合浮动：

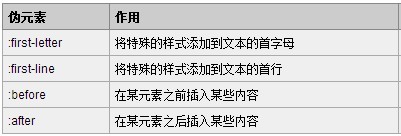


清除浮动方法：



1. 伪类和伪元素的区别





（1）伪类存在的意义是为了通过选择器找到那些不存在于DOM树中的信息以及不能被常规CSS选择器获取到的信息，伪类由一个冒号开头。

获取不存在于DOM树中的信息。如<a>标签的：link、visited等， 这些信息不存在于DOM树结构中，只能通过伪类选择器来获取，

获取不能被常规选择器获取的信息。比如伪类：target，它的作用是匹配文档的URI中某个标识符的目标元素。

（2）伪元素是在DOM树中创建了一些抽象元素，这些抽象元素是不存在于文档语言中的。比如document接口不提供访问元素内容的第一字或者第一行的机制，而伪元素可以使开发者提取到这些信息。并且，一些伪元素可以使开发者获取到不存在于源文档中的内容（::before，::after）

两者区别：

1. 伪类本质上是为了弥补常规css选择器的不足，以便获取到更多的信息
2. 伪元素本质上是创建了一个有内容的虚拟容器
3. CSS3中伪类和伪元素的语法不同
4. 可以同时使用多个伪类，而只能使用一个伪元素
5. HTML5标签语义化？

优点：

1. 可以提升可访问性与操作性
2. 改进搜索引擎的优化
3. 一般来说可以让HTML文件更小
4. 让代码更好的维护，与CSS3的关系更和谐
5. 了解过flex布局吗？说说它和传统布局的有何不同？

Flex是Flexible Box的缩写，意为“弹性布局”，用来为盒状模型提供最大的灵活性。

<http://www.ruanyifeng.com/blog/2015/07/flex-grammar.html>

1. 说出前端link和import的区别
2. link是标签，@import是CSS提供的一种方式。Link标签除了可以加载css外，还可以定义RSS、rel连接属性等，@import只能加载css
3. 加载顺序的差别。当一个页面加载的时候，link引用的css会同时被加载，而@import引用的css会等到页面全部被下载完再被加载。
4. 兼容性差别。[由于@import是css2.1](mailto:由于@import是css2.1)提出的所以老的浏览器不支持，只能IE5以上的才能识别，而link标签无此问题。
5. 使用dom样式时的差别。当使用javascript控制dom去改变样式的时候，只能使用link标签，因为@import不是dom可以控制的。
6. 你有经常使用CSS吧，水平垂直居中怎么实现？

# 架构模式

MVC —— MVP —— MVVM

<https://juejin.im/post/593021272f301e0058273468>

# Vue

# React

# JS

## 原型

什么是原型？什么是原型链？如何查看原型？

定义： JavaScript中有值类型和引用类型两种，所有的引用类型都是对象（对象是属性的集合），每个对象都有一个内置属性[[Prototype]]，它持有一个连接到另一个称为其原型对象的链接 。该原型对象又具有自己的原型，层层向上直到一个对象的原型为null，null没有原型，作为这个原型链中的最后一个环节。

（[[Prototype]]该属性es5之前没有标准访问，但大多数浏览器都支持通过\_\_proto\_\_来访问，es5添加了对这个内置属性的Get方法Object.getPrototypeOf()）

作用： 访问一个对象的属性时，引擎会先在其基本属性中查找，如果没有，就会在[[Prototype]]关联的对象中进行查找。如果也没有找到，同理，会继续查找后者的[[Prototype]]。

Prototype和\_\_proto\_\_: 每个函数都有一个属性叫做prototype，这个属性值是一个对象， 默认只有一个constructor属性，指向这个函数本身；每个对象都有一个\_\_proto\_\_属性，指向创建该对象的函数的prototype。

## 闭包

什么是闭包？闭包的三个特性？闭包的缺点？举个例子？

定义： 闭包是指有权访问另一个函数作用域中的变量的函数。

（创建闭包最常见的方式就是在一个函数内部创建另一个函数，通过另一个函数访问这个函数的局部变量。）

闭包特性：

1，函数当作第一级的值类型进行传递

2，函数在当前词法作用域之外执行

3，参数和变量不会被垃圾回收机制回收

（网上常见答案：1，函数嵌套函数 2，函数内部可以引用外部的参数和变量 3，同上）

闭包的缺点：

常驻内存，会增加内存使用量，使用不当会很容易造成内存泄漏

## 继承

实现继承的几种方式？

原型链继承、构造函数继承、组合继承、原型式继承、（寄生式继承、寄生组合式继承）

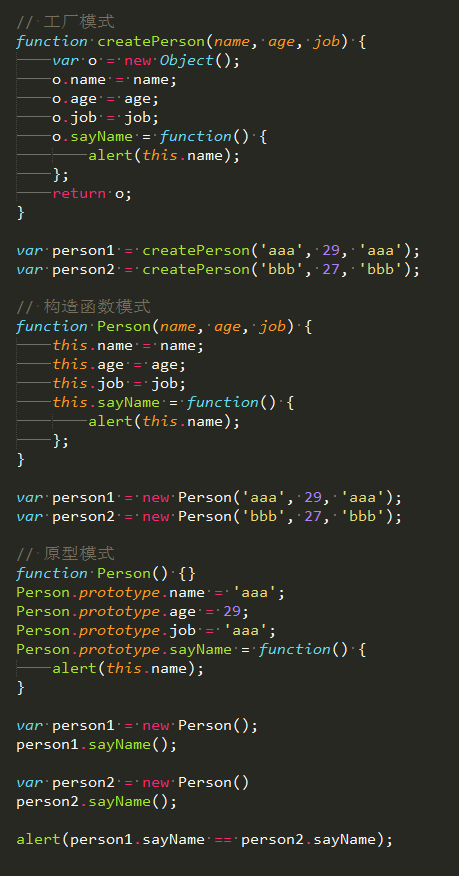




## 对象

创建对象的方法？

对象字面量方式、工厂模式、构造函数模式、原型模式、组合构造函数和原型模式





## New Foo()

new Foo()时发生了什么？

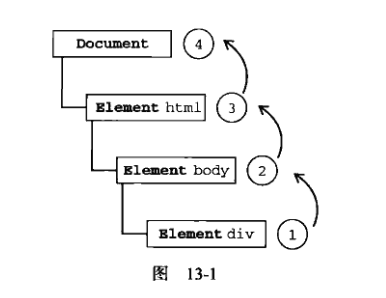
1. 创建一个新对象
2. 将构造函数的作用域给新对象（因此this就指向了这个新对象）
3. 执行构造函数中的代码（为这个新对象添加属性）
4. 返回新对象

## 事件

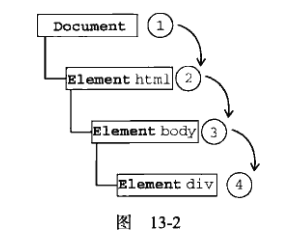
1. 事件流

事件流描述的是从页面中接收事件的顺序。一共有两种：事件冒泡流和事件捕获流。

IE的事件流叫做事件冒泡，即事件开始时由最具体的元素（文档中嵌套层次最深的那个节点）接收，然后逐级向上传播到较不具体的节点（文档）



Netscape Communicator团队提出的是另一种事件流叫做事件捕获流。事件捕获的思想是不太具体的节点应该最早接收到事件，而最具体的节点应该最后接收到事件。捕获的用意在于在事件到达预定目标之前捕获它。



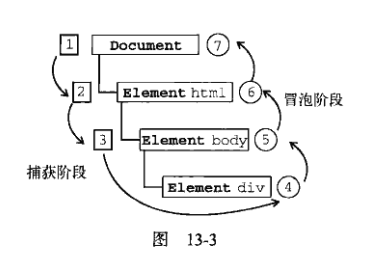
（由于老版本的浏览器不支持。因此很少有人用事件捕获。建议放心使用冒泡，有特殊需要时再使用事件捕获）

1. DOM事件



DOM级别一个可以分为4个级别：DOM0级，DOM1级，DOM2级，DOM3级。

DOM2级事件 规定的事件流包括三个阶段：事件捕获阶段、处于目标阶段、事件冒泡阶段。首先发生的是事件捕获，为截获事件提供机会。然后是实际的目标接收到事件。最后一个阶段是冒泡阶段，可以在这个阶段对事件做出响应。如下，单击div元素触发事件的顺序：



事件处理程序：

事件就是用户或浏览器自身执行的某种动作，如：click、load、mouseover。而响应某个事件的函数就叫做事件处理程序（或事件侦听器），事件处理程序的名字以“on”开头，如：onclick、onload。

## defer/async

Script标签的defer、async的区别？



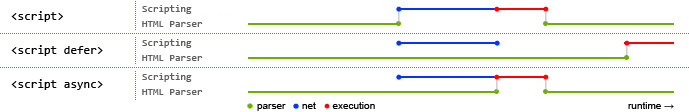
没有defer或async，浏览器会立即加载并执行指定的脚本，“立即”指的是在渲染该script标签之下的文档元素之前，也就是说不等待后续载入的文档元素，读到 加载并执行。



有async，加载和渲染后续文档元素的过程将和script.js的加载与执行并行进步（异步）



有defer，加载后续文档元素的过程将和script.js的加载并行进行（异步），但是script.js的执行要在所有元素解析完成之后，DOMContentLoaded事件触发之前完成



蓝色线代表读取，红色线代表执行时间，这俩都是针对脚本的；绿色线代表HTML解析

此图告诉我们以下几个要点：

1. defer 和 async 在网络读取（下载）这块儿是一样的，都是异步的（相较于 HTML 解析）
2. 它俩的差别在于脚本下载完之后何时执行，显然 defer 是最接近我们对于应用脚本加载和执行的要求的
3. 关于 defer，此图未尽之处在于它是按照加载顺序执行脚本的，这一点要善加利用
4. async 则是一个乱序执行的主，反正对它来说脚本的加载和执行是紧紧挨着的，所以不管你声明的顺序如何，只要它加载完了就会立刻执行
5. 仔细想想，async 对于应用脚本的用处不大，因为它完全不考虑依赖（哪怕是最低级的顺序执行），不过它对于那些可以不依赖任何脚本或不被任何脚本依赖的脚本来说却是非常合适的，最典型的例子：Google Analytics

## 前端路由原理

两种主流方式实现前端路由：

1. History API（使用html5提供的history API）

主要用到两个新增的API：history.pushState和history.replaceState



相同之处是，他们接收完全相同的参数，都是对浏览器的历史栈进行操作，将传递的url和相关数据压栈，并将浏览器地址栏的url替换成传入的url且不刷新页面；不同的地方是，pushState将指定的url直接压入历史记录栈顶，而replaceState是将当前历史记录栈顶替换成传入的数据。

1. Hash（利用url中的hash字段）

例子：<http://sandbox.runjs.cn/show/dxi5lgcx>

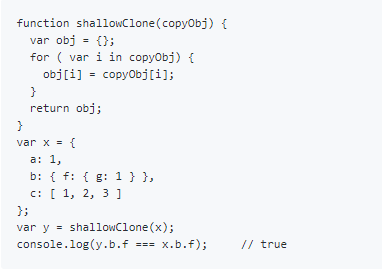
<https://segmentfault.com/a/1190000007238999>

<http://tomasran.space/archives/1bjDjwd2FgzJIsCyshbzag/>

## 深拷贝与浅拷贝

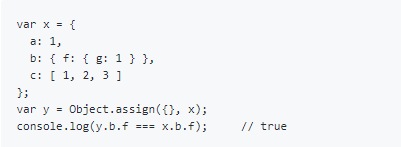
1，浅拷贝

* 1. 简单的引用复制



* 1. Object.assign()

Object.assign()方法可以把任意多个的源对象自身的可枚举属性拷贝给目标对象，然后返回目标对象。



2，深拷贝

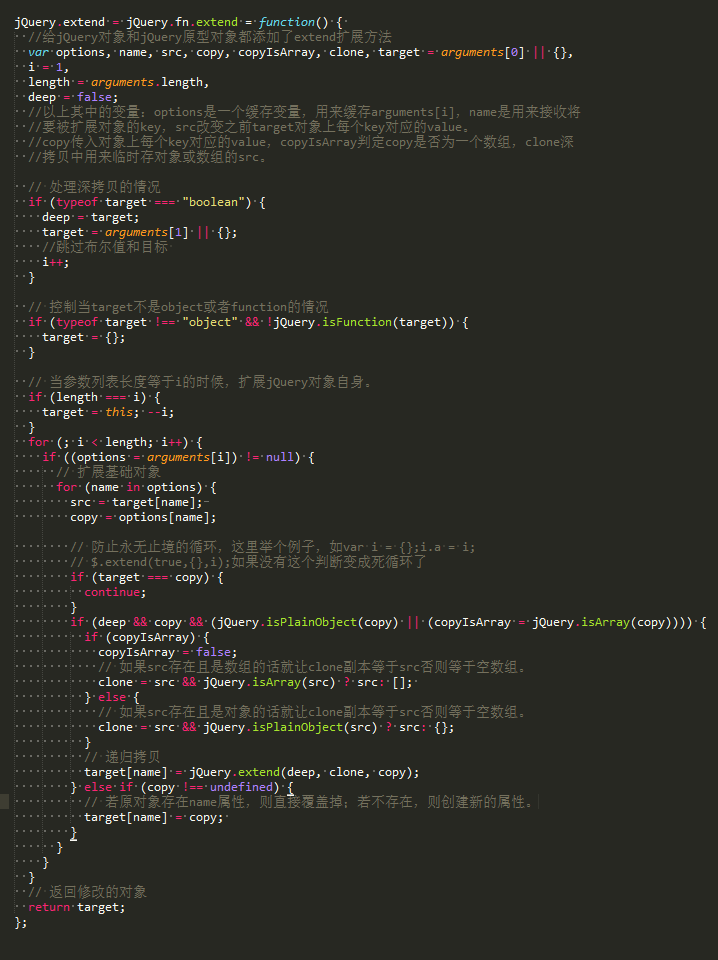
2.1 JSON对象的parse和stringfy

JSON对象是ES5中引入的新的类型，JSON对象parse方法可以将JSON字符串反序列化成JS对象，stringfy方法可以将JS对象序列化成JSON字符串，借助这两个方法，也可以实现对象的深拷贝。

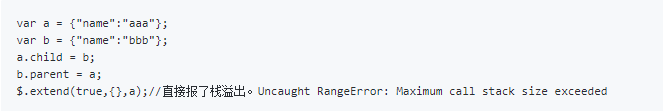


这种方法使用比较简单，可以满足基本的深拷贝需求，而且能够处理JSON格式能表示的所有数据类型。但是对于正则表达式类型、函数类型等无法进行深拷贝（而且会直接丢失相应的值），还有一点不好的地方是它会抛弃对象的constructor，也就是深拷贝之后，不管这个对象原来的构造函数是什么，在深拷贝之后都会变成Object。同时如果对象存在循环引用的情况也无法正确处理。

2.2 jQuery.extend()方法源码实现



jQuery的extend方法使用基本的递归思路实现了浅拷贝和深拷贝，但是这个方法也无法处理源对象内部循环引用，例如：



<https://github.com/wengjq/Blog/issues/3>

## ajax

1. 原生js发送ajax请求一般要经过哪些步骤
2. 实例化XMLHttpRequest对象
3. 连接服务器
4. 发送请求
5. 接收响应数据

具体实现如下：



1. ajax的readystate有5个状态 ，每个状态表示什么？

readyState是XMLHttpRequest对象的一个属性，用来标识当前XMLHttpRequest对象处于什么状态。

readyState总共有5个状态值，分别为0~4，每个值代表了不同的含义：

0：初始化，XMLHttpRequest对象还没有完成初始化

1：载入，XMLHttpRequest对象开始发送请求

2：载入完成，MLHttpRequest对象的请求发送完成

3：解析，XMLHttpRequest对象开始读取服务器的响应

4：完成，XMLHttpRequest对象读取服务器响应结束

1. Ajax的status有什么？

status是XMLHttpRequest对象的一个属性，表示响应的HTTP状态码

HTTP协议状态码总共可分为5大类：

1xx：信息响应类，表示接收到请求并且继续处理

2xx：处理成功响应类，表示动作被成功接收、理解和接收

3xx：重定向响应类，为了完成指定动作，必须接收进一步处理

4xx：客户端错误，客户请求包含语法错误或者是不能正确执行

5xx：服务器错误，服务器不能正确执行一个正确的请求

几个常用状态码：

400：服务器未能理解请求

404：服务器无法找到被请求的页面

502：请求未完成，服务器从上游服务器收到一个无效响应

503：请求未完成，服务器临时过载或当机

1. Promise封装ajax



1. GET和POST请求的区别
2. 同源与跨域？什么是同源策略？跨域通信的几种方式？JSONP原理

## 优化

代码优化基本方法，你做过什么性能优化？

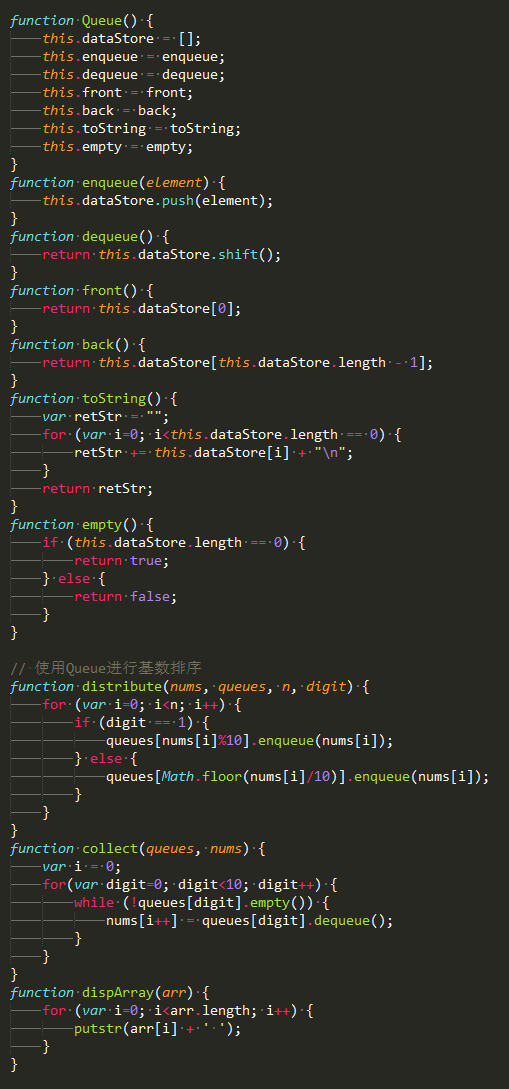
<https://github.com/creeperyang/blog/issues/1>

## JS实现数据结构：

### Stack(栈)



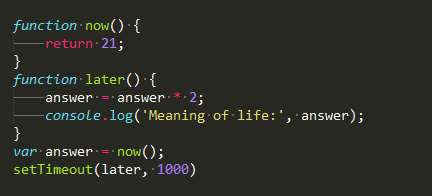
### Queue(队列)



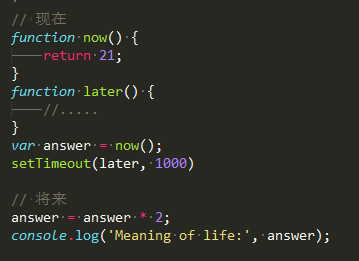
## 异步

### 异步编程

程序中现在运行的部分和将来运行的部分之间的关系就是异步编程的核心。



考虑以上代码，这个程序中有两个块：现在执行的部分，以及将来执行的部分。



“现在”这一块程序运行之后就会立即执行。但是setTimeout(…)还设置了一个事件（定时）在“将来”只能执行，所以later()的内容会在之后的某个时间执行。

任何时候，只要把一段代码包装成一个函数，并指定它在响应某个事件（定时器、鼠标点击、Ajax响应等）时执行，就是在代码中创建了一个“将来”执行的块，也由此在这个程序中引入了异步机制。

### 异步回调

前端如何解决异步回调地狱？

1. 减少代码嵌套
2. 模块化
3. 处理每一个错误

更高级的解决方案：

Promise、Generator、Async

<https://segmentfault.com/a/1190000009644973>

### Promise

#### 概念性

Promise是异步编程的一种解决方案。所谓Promise，简单说就是一个容器，里面保存某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise是一个对象，从语法上说，Promise是一个对象，从它可以获取异步操作的消息。

Promise对象的特点：

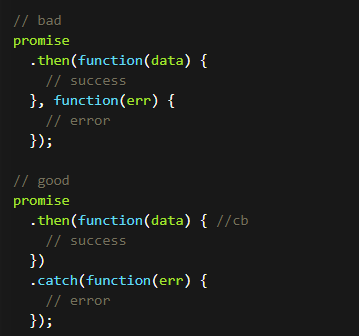
1. 对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：pending(进行中)、fulfilled(已成功)和rejected(已失败)。只有异步操作的结果，可以决定当前是哪种状态，任何其他操作都无法改变这个状态。
2. 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种情况：从pending到fulfilled和从pending变为rejected。只要这两种情况发生了，状态就凝固了，不会再变了。如果改变已经发生，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件(event)完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

Promise的缺点：

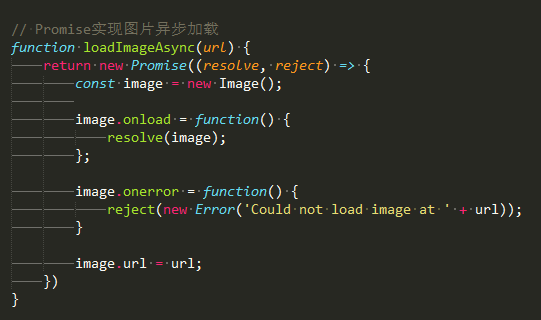
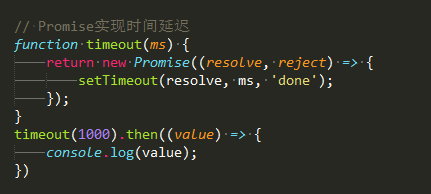
1. 无法取消Promise，一旦新建它就会立即执行，无法中途取消。
2. 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
3. 当处于pending状态时，无法得知目前进展到哪个阶段（刚刚开始还是即将结束）

错误捕获：

一般不建议在then方法里面定义Reject状态的回调函数（即then的第二个参数），总是使用catch方法



#### 实际应用



### Iterator(遍历器)

1，基本概念：

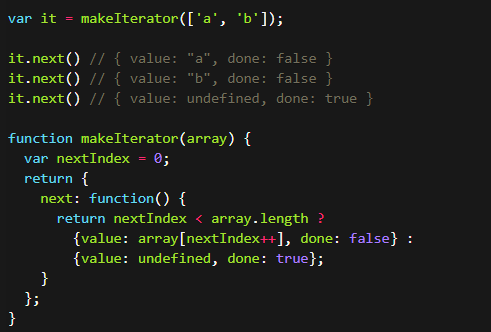
遍历器（Iterator）是一种接口，为各种不同的数据结构（Array/Object/Map/Set）提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）

2，作用：

1. 为各种数据结构，提供一个统一的、简便的访问接口；
2. 使得数据结构的成员能够按某种次序排列；
3. ES6创造了一种新的遍历命令for…of循环，Iterator接口主要供for…of消费
4. Iterator遍历过程
5. 创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器的本质上，就是一个指针对象
6. 第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。
7. 第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。
8. 不断调用指针对象的next方法，直到它指向数据结构的结束位置。

每一次调用next方法，都会返回数据结构当前成员的信息——返回一个包含value和done两个属性的对象。Value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

1. 一个模拟next方法返回值的例子



1. 默认Iterator接口

默认的Iterator接口部署在数据结构的Symbol.iterator属性，即一个数据结构只要具有Symbol.iterator接口，就可以认为是“可遍历的”。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数，执行这个函数就会返回一个遍历器。

具备Iterator接口的数据结构如下：

Array、Map、Set、String、TypedArray、函数的arguments对象、NodeList对象

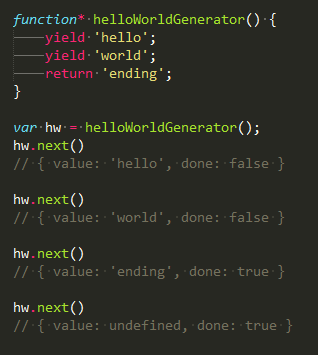
### Generator

1. 基本概念

语法上，Generator函数是一个状态机，封装了多个内部状态；执行Generator函数会返回一个遍历对象，即它也是一个遍历器对象生成函数，返回的遍历器对象，可以一次遍历Generator函数内部的每一个状态。

形式上，function关键字与函数名之间有一个星号；函数体内部使用yield表达式，定义不同的内部状态。

1. 执行步骤

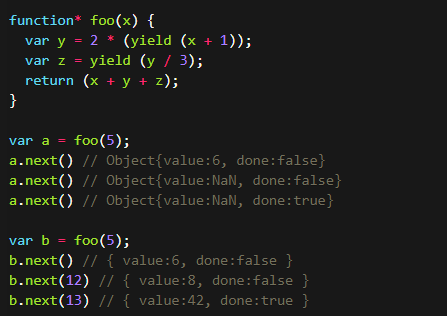


Generator函数的调用方法和普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，即遍历器对象。

然后，调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方执行，直到遇到下一个yield表达式（或return语句）为止。换言之，Generator函数是分段执行的，yield表达式是暂停执行的标记，而next方法可以恢复执行。

1. next方法的执行参数

yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式返回值。以下是个例子：



### Async

## 模块化

### 基本理解

模块时一个可重用的代码片段。无论是从代码语法结构上解耦，还是将代码拆分至不同的文件中，只要用某种方法将代码“分离”，就认为是一个模块，因此任何代码都可以处理成一个模块。

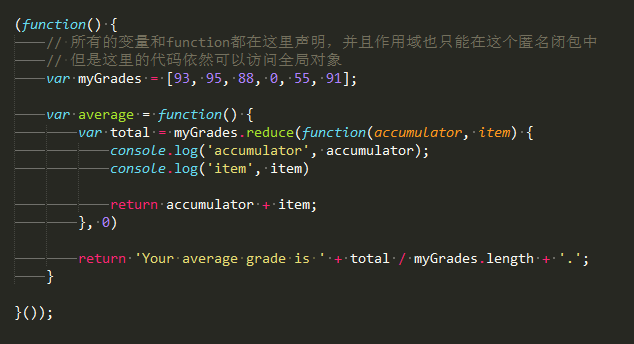
模块化的优点：

模块化可以使代码低耦合，功能模块直接不相互影响。

1. 可维护性。良好设计的模块会尽量与外部代码撇清关系，以便于独立对其进行改进和维护。维护一个独立的模块比起一团乱的代码来说要轻松很多。
2. 命名空间。再Javascript中，最高级别的函数外定义的变量都是全局变量（所有人都可访问）。也正因如此，当一些无关的代码碰巧使用到同名变量的时候，我们会遇到“命名空间“的问题。
3. 可复用性。

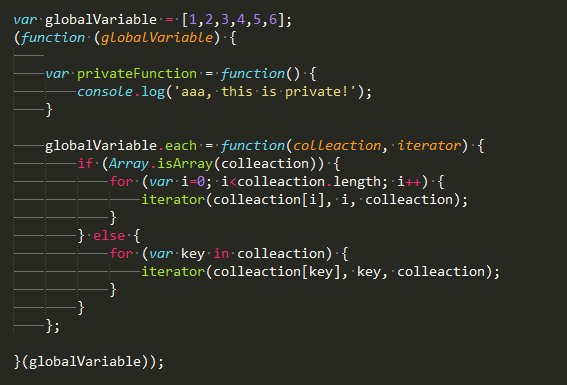
### 实现模块化的几种方式

1. 匿名闭包函数

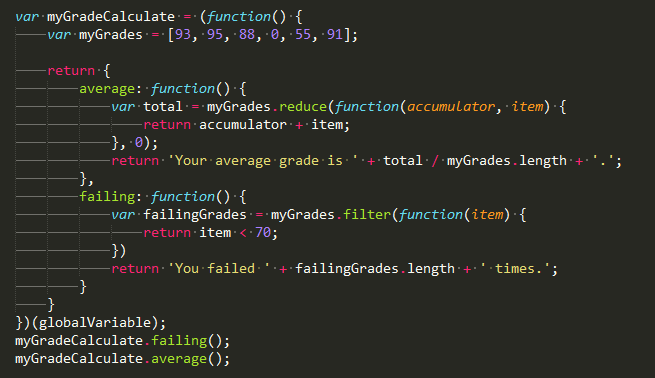


1. 引入全局变量

我们将全局变量当作一个参数传入到匿名函数然后使用，比隐式全局变量更清晰快速

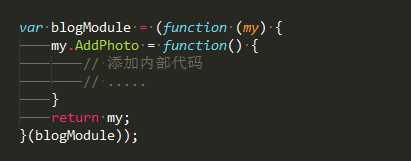


有时我们不仅需要传入一个全局变量，还要使用闭包中的方法，所以我们声明一个全局变量来保存它



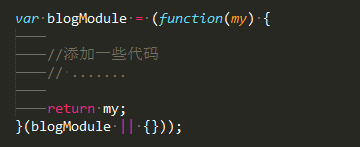
1. 模型扩展

Module模式的一个限制就是所有的代码都要写在一个文件中，但是在一些大型项目中，将一个功能分离成多个文件是非常重要的，因为可以多人合作易于开发。解决这个的方法就是，将声明的函数本身作为参数传入，给其添加一个函数属性，然后再返回。



松耦合扩展

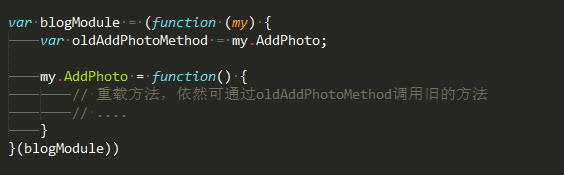
以上代码尽管可以执行，但是必须保证先声明blogModule，然后再执行代码。也就是说步骤不能乱。对于这个问题，可以用下面方式解决



通过这样的代码，每个分离的文件都保证这个结构，那么我们就可以实现任意顺序的加载，所以这个时候的var是必须要声明的，因为如果不声明，其他文件读取不到

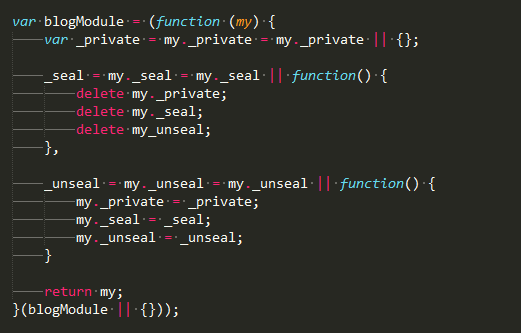
紧耦合扩展

松耦合扩展没办法重写一些属性和函数，也不能在初始化的时候就用Module属性，因为松耦合加载顺序不确定，紧耦合就通过限制加载顺序，提供了重载的机会



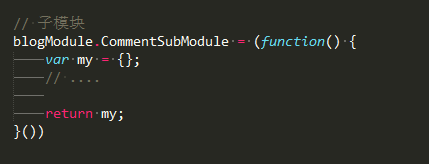
1. 跨文件共享私有对象

以上Module每个文件匿名函数里的私有对象都不能交叉访问，但是如果需要使用怎么解决呢



任何文件都可以对他们的局部变量\_private设置属性，并且设置对其他文件也立即生效。一旦这个模块加载结束，应用会调用blogModule.\_seal()“上锁”，这会阻止外部介入内部的\_private；如果这个模块需要再次增生，应用的声明周期内，任何文件都可以调用\_unseal()“开锁”，然后再加载新文件，加载后再次调用\_seal()“上锁”。

1. 子模块



### CommonJS、AMD和CMD

关于模块的编写方式Javascript在ES6出现之前还没有，目前，通行的模块规范有两种：CommonJS、AMD和CMD。

Node.js的产生将javascript语言用于服务器端编程，这标志着“Javascript模块化编程”正式诞生，因为在浏览器环境下，网页程序复杂性有限，没有模块也可以。但是在服务器端，与操作系统和其他应用程序互动，一定要有模块，否则没法编程。Node.js的模块系统就是参照CommonJS规范实现的。但是CommonJS不适用于浏览器，因为它是“同步加载”，而浏览器的页面只能采用“异步加载”，才能更好的创建用户体验，所以这也是AMD规范诞生的背景。AMD是"Asynchronous Module Definition"的缩写，意思就是“异步模块定义”。它采用异步加载模块的方式，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数(require([module], callback))中，等到加载完成之后，这个回调函数才会运行。目前实现AMD规范的Javascript库有：requirejs、curl.js等。

(舆论：AMD 规范在 CommonJS 社区争议很大，规范里太多 RequireJS 的影子。社区里不少人反感 RequireJS 打着 CommonJS 的口号，甚至建议 AMD 自立门户)

CMD是“Common Module Definition”通用模块定义，CMD规范是国内发展出来的（前端工程师玉伯提出，他是seajs的创作者），要解决的问题和AMD一样，也是异步加载模块，只不过在模块定义方式和模块执行时机上有所不同。

### RequireJS和Sea.js

（1）两者官方比较

相同点：两者要解决的问题相同，都是浏览器端的模块化开发，目标一致

不同点：

1，定位不同。RequireJS想成为浏览器端的模块加载器，同时也想成为Node等环境的模块加载器。Sea.js则专注于Web浏览器端，同时通过Node扩展的方式可以很方便的跑在Node环境中。

2，遵循的规范不同。RequrieJS遵循AMD规范（异步模块定义），Sea.js遵循CMD规范（通用模块规范）。规范的不同，导致了两者API不同。Sea.js更贴近CommonJS Module/1.1和Node Module规范。

3，推广理念不同。RequireJS在尝试让第三方类库修改自身来支持RequireJS，目前只有少数社区采纳。Sea.js不强推，采用自主封装的方式来“海纳百川”，目前已有较成熟的封装策略。

4，对开发调试的支持不同。Sea.js非常关注代码的开发调试，有nocache、debug等用于调试的插件。RequireJS无这方面的支持。

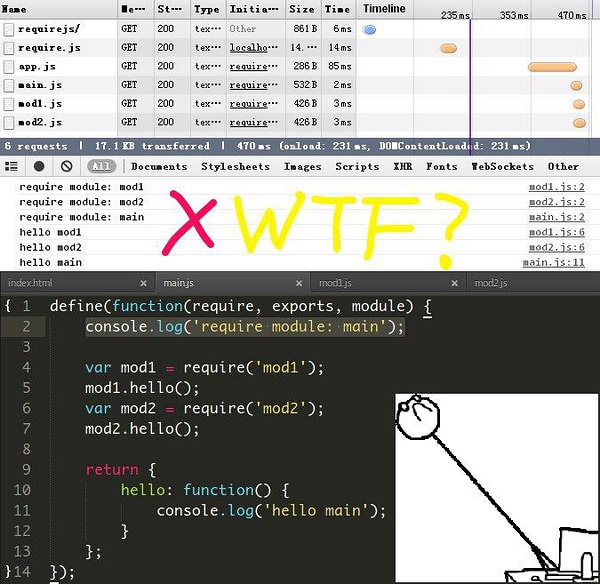
5，插件机制不同。RequireJS采取的是在源码中预留接口的形式，插件类型比较单一。Sea.js采取的是通用事件机制，插件类型更丰富。

<https://lifesinger.wordpress.com/2011/05/17/the-difference-between-seajs-and-requirejs/>

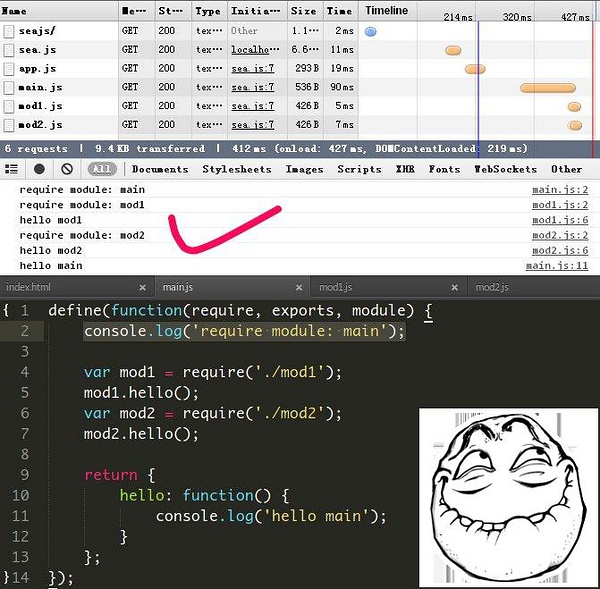
（2）执行机制不同详解

SeaJS对模块的态度是懒执行, 而RequireJS对模块的态度是预执行。

对同一段代码的执行，RequireJS结果如下：



SeaJS结果如下：

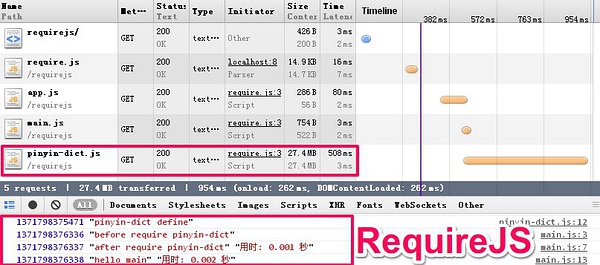


根据以上代码可以得出结论：

SeaJS只会在真正需要使用依赖模块的时候才执行该模块，RequireJS会尽早地执行依赖模块。

模块的加载都是并行的, 没有区别, 区别在于执行模块的时机, 或者说是解析

以下用图说明模块的执行时机：



<https://www.douban.com/note/283566440/>

### RequireJS

1. 概述

RequireJS的基本思想是，通过define方法将代码定义为模块；通过require方法实现代码的模块加载。

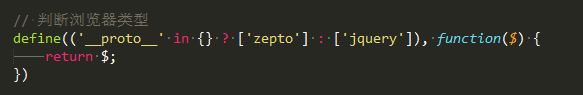
（1）首先将require.js嵌入网页，并指定主代码所在的脚本。如下：



（2）定义模块（独立模块和非独立模块）



一个实际例子：



（3）调用模块



（4）配置require.js：config方法



（5）使用插件（可查找官方文档）

<http://javascript.ruanyifeng.com/tool/requirejs.html>

### ES6模块化机制

ES6模块和CommonJS模块的差异：

1. CommonJS模块输出的是一个值得拷贝，ES6模块输出的是值得引用。
2. CommonJS模块是运行时加载，ES6模块是编译时输出接口。

<http://es6.ruanyifeng.com/#docs/module>

## 前端安全

1. XSS
2. 警惕iframe带来的风险
3. 别被点击劫持了
4. 错误内容的推断
5. 不安全的第三方依赖包
6. 用了HTTPS也可能掉坑里
7. 本地存储数据泄露
8. 缺失静态资源完整性校验

XSS攻击

<https://segmentfault.com/a/1190000006672214#articleHeader1>

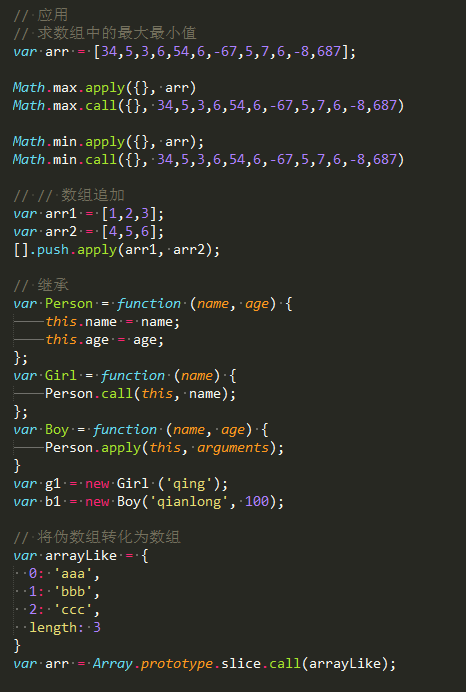
## apply()、call()和bind()

相同点：改变函数运行时的this指向

不同点：call和apply改变了函数的this上下文后便执行了该函数，而bind则是返回改变了上下文后的一个函数

call和apply的区别在于传递的参数，call和apply的第一个参数都是要改变上下文的对象，而call的从第二个参数开始以参数列表的形式展现，apply则是将其余参数放在一个数组里作为它的第二个参数。

应用：



## 存储

Session、 cookie、 sessionstorage、 localstorage四个比较



## 数组

1. 数组方法

一种分类：

1. ES6新增方法

Array.of、Array.form

1. 改变自身值的方法（9）

pop、push、shift、unshift、reverse、sort、splice、copyWithin（ES6）、fill（ES6）

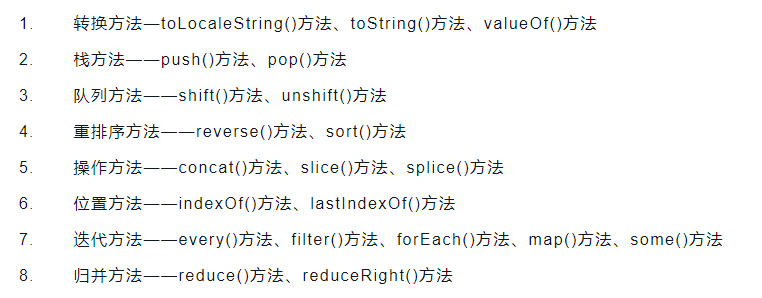
1. 不改变自身值的方法

concat、join、slice、toString、indexof、lastIndexOf、toLocalString

1. 遍历方法

foreach、every、some、filter、map、reduce、entries(ES6)、find&findIndex(ES6)、keys(ES6)、values(ES6)、Symbol.iterator(ES6)

另一种分类

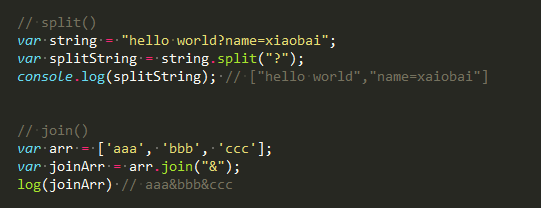


## 字符串

1. split()和join()的区别

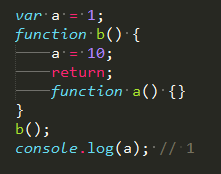
split()用于分割字符串，返回一个数组；join()用于连接多个字符或字符串，返回一个字符串。

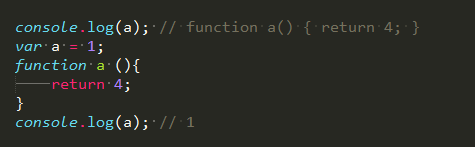
split()只有一个参数时，代表分割的字符串或正则表达式；两个参数时，第二个参数时指返回的数组中元素的个数。

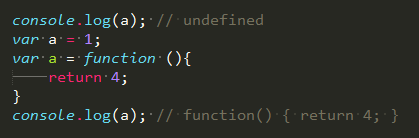


## 试题

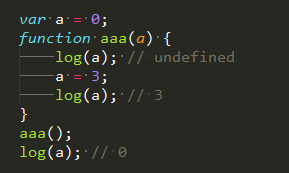
### 变量提升

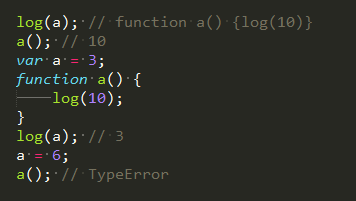
1. 

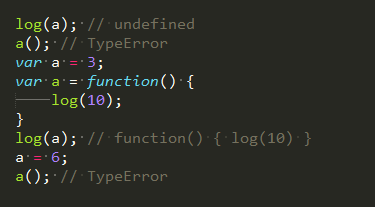
2， 



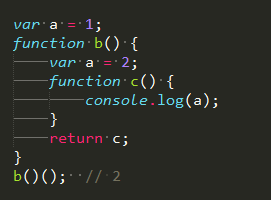
解析：同一个标识符的情况下，变量声明和函数声明都会提升；函数声明会覆盖变量声明，但不会覆盖变量赋值，即：如果声明变量的初始化或者赋值那么变量的优先级高于函数

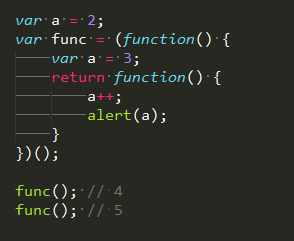
3， 

4， 

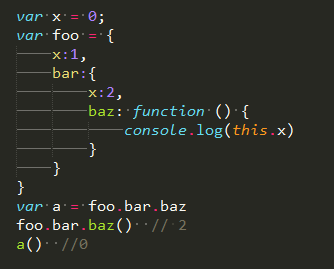


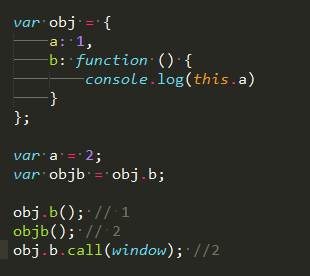
### 闭包

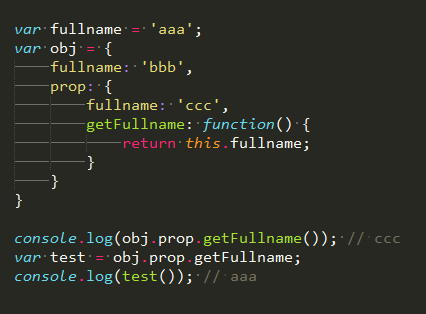
1， 

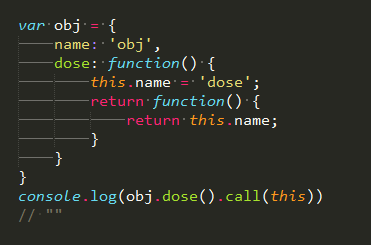
2， 

### This

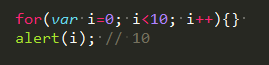
1. 

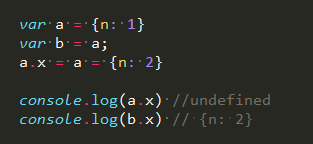
2， 

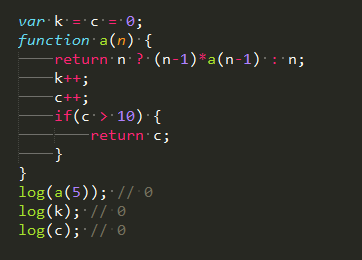
3， 

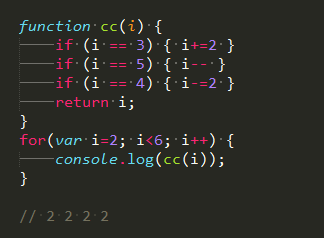
4， 

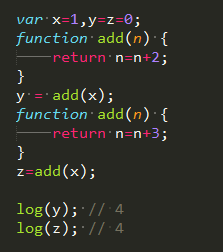
### 一般

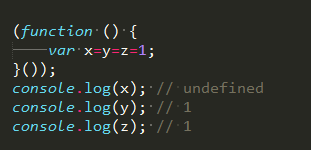
1， 

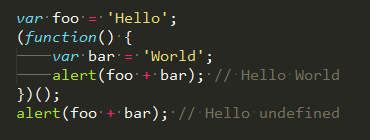
2， 

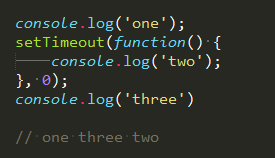
3， 

4， 

5， 

6， 

7， 

8， 

# 浏览器知识

## 一个页面从输入 URL 到页面加载完的过程中都发生了什么事情

url：协议、网络地址：端口号、资源路径、查询字符串？、片段标识符

1. 输入地址
2. 浏览器查找域名的IP地址
3. 浏览器向web服务器发送一个HTTP请求
4. 服务器的永久重定向响应
5. 浏览器跟踪重定向地址
6. 服务器处理请求
7. 服务器发那会一个HTTP响应
8. 浏览器显示HTML
9. 浏览器发送请求获取嵌入在HTML中的资源（如图片、音频、视频、CSS、JS等）

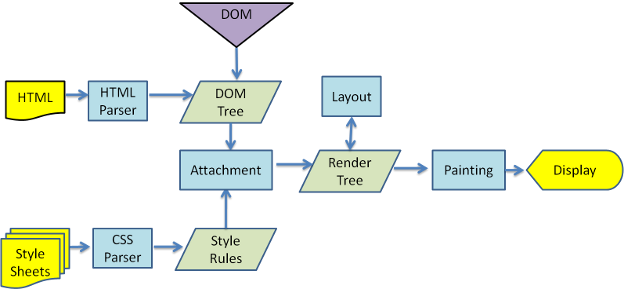
<http://web.jobbole.com/91239/>

<https://mp.weixin.qq.com/s/27zpNIGhVbx-on9FDs_6dw>

## 浏览器渲染原理

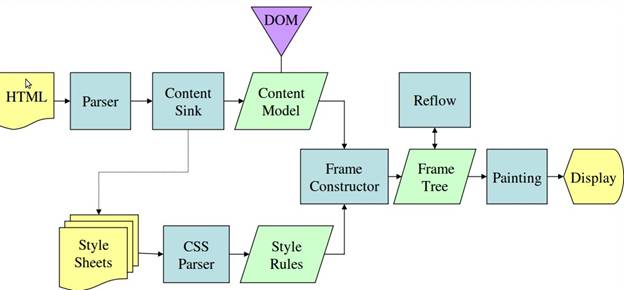
在浏览器还没有完全接收HTML文档的时，它就已经开始显示这个页面了。不同的浏览器解析过程不一样，如下是两种浏览器的解析过程：

Webkit主要流程：



构建DOM树和Style Rules ->构建Render Tree -> 布局Render Tree -> 绘制Render Tree

Geoko主要流程：



浏览器在解析html文件时，会“自上而下”加载，并在加载过程中进行解析渲染。在解析过程中，如果遇到请求外部资源时，如图片、外链CSS、JS等，请求过程都是异步的，并不影响html文档进行加载。浏览器首先会解析HTML文件构建DOM树，然后解析CSS文件构建渲染树，等到渲染树构建完成后，浏览器开始布局渲染树并将其绘制到屏幕上。这个过程比较复杂，设计到两个概念reflow（重排/回流）和repaint（重绘）。DOM节点中的各个元素都以盒模型的形式存在，这些都需要浏览器去计算其位置和大小等，这个过程称为reflow；当盒模型的位置，大小以及其他属性，如颜色，字体等确定下来之后，浏览器便开始绘制内容，这个过程称为repaint。页面在首次加载时必然会经历reflow和repaint。Reflow和repaint的过程是非常消耗性能的，尤其在移动设备上，它会破坏用户体验，有时会造成卡顿。所以我们尽可能少的减少reflow和repaint。

当文档加载过程中遇到JS文件时，HTML文档会挂起渲染过程，不仅要等到文档中JS文件加载完毕还要等待解析执行完毕，才会继续HTML的渲染过程。原因是JS有可能修改DOM结构，这就意味着JS执行完成前，后续所有资源的下载是没有必要的，这就是JS阻塞后续资源下载的根本原因。CSS文件的加载不影响JS文件的加载，但是影响JS文件的执行。JS代码执行前浏览器必须保证CSS文件已经下载并加载完毕。

## reflow（重排）和repaint（重绘）

（1）当DOM的变化影响了元素的几何属性（宽和高）——比如改变边框宽度或给段落增加文字，导致行数增加——浏览器需要重新计算元素的几何属性，同样其他元素的几何属性和位置也会因此受到影响。浏览器会使渲染树中受到影响的部分失效，并重新构造渲染树。这个过程称为“重排（reflow）”。完成重排后，浏览器会重新绘制受影响的部分到屏幕中，该过程称为“重绘（repaint）”“重排”必然导致“重绘”，“重绘”不一定需要“重排”

（2）何时发生重排：

添加或删除可见的DOM元素

元素位置改变

元素尺寸改变（包括外边距、内边距、边框厚度、宽度、高度等属性改变）

内容改变，例如文本改变或图片被另一个不同尺寸的图片替代

页面渲染器初始化

浏览器窗口尺寸改变

（3）如何降低浏览器重新渲染的频率的成本

批量修改DOM元素

使元素脱离文档流 -> 对其应用多重改变 -> 把元素带回到文档中

缓存布局信息

让元素脱离动画流

使用绝对定位页面上的动画元素，将其脱离文档流

让元素动起来。当它扩大时，会临时覆盖部分页面。但这只是页面一个小区域的重绘过程，不会产生重排并重绘页面的大部分内容

当动画结束时恢复定位，从而只会下移一次文档的其他元素

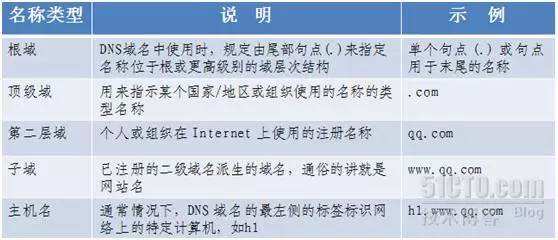
## 渲染引擎与js解释引擎

# 网络知识

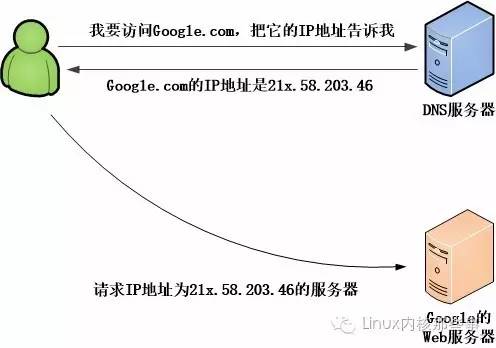
## DNS服务

DNS（Domain Name System，域名系统），因特网上作为域名和IP地址相互映射的一个分布式数据库，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。通过主机名，最终得到该主机名对应的IP地址的过程叫做域名解析（或主机名解析）。

DNS域名称空间的组织方式：



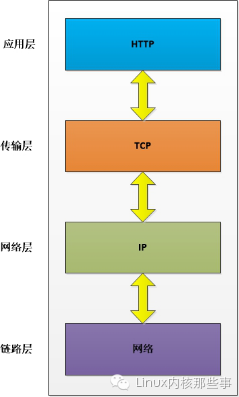
DNS服务把域名解析成IP地址的过程：



## HTTP协议

HTTP协议是构建在TCP/IP协议之上的，是TCP/IP的一个子集

TCP/IP协议是由一个四层协议组成的系统，这四层分别为：数据链路层、网络层、传输层和应用层。如图：



分层的好处是把各个相对独立的功能解耦，层与层之间通过规定好的接口来通信。如果以后需要修改或者重写某一个层的实现，只要接口保持不变也不会影响到其他层的功能。

以下为各层的主要作用：

1. 应用层

应用层一般是我们编写的应用程序，其决定了向用户提供的应用服务。应用层可以通过系统调用与传输层进行通信

处于应用层的协议很多，如：FTP（File Transfer Protocol，文件传输协议）、DNS（Domain Name System，域名系统）和HTTP（HyperText Transfer Protocol，超文本传输协议）等。

1. 传输层

传输层通过系统调用向应用层提供处于网络连接中的两台计算机之间的数据传输功能。

在传输层有两个性质不同的协议：TCP（Transmission Control Protocol，传输控制协议）和UDP（User Data Protocol，用户数据报协议）。

1. 网络层

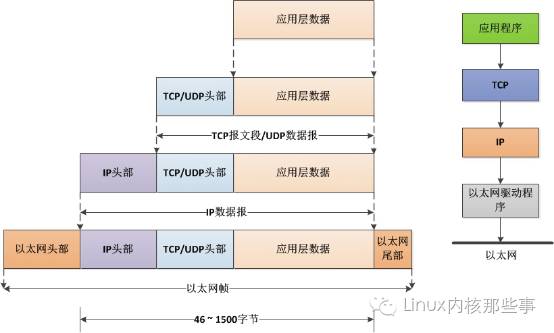
网络层用来处理在网络上流动的数据包，数据包是网络传输的最小数据单位。该层规定了通过怎样的路径（传输路线）到达对方计算机，并把数据包传输给对方。

1. 链路层

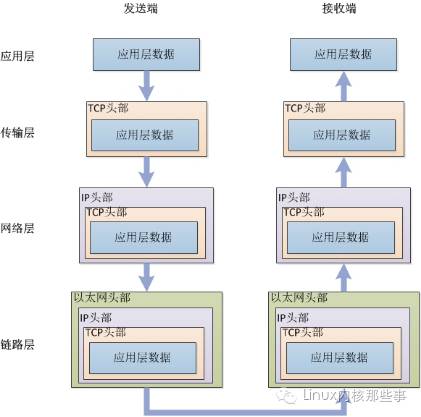
链路层用来处理连接网络的硬件部分，包括控制操作系统、硬件设备驱动、NIC（Network Interface Card，网络适配器）以及光纤等物理可见部分。硬件上的范畴均在链路层的作用范围之内。

数据包封装：

上层协议数据通过封装转变为下层协议数据。应用程序数据在发送到物理网络之前，会沿着协议栈从上往下传递，每层协议都将在上层协议数据的基础上加上自己的头部信息（链路层还会加上尾部信息），以为实现该层功能提供必要的信息。如图：



发送端发送数据时，数据会从上层传输到下层，且每经过一层都会被打上该层的头部信息。而接收端接收数据时，数据会从下层传输到上层，传输前会把下层的头部信息删除。如图：



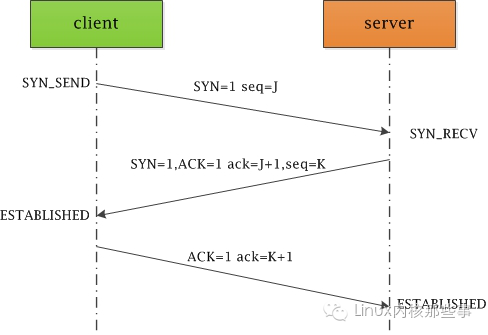
由于下层协议的头部信息对上层协议是没有实际的用途，所以在下层协议传输给上层协议的时候会把该层的头部信息去掉，这个封装过程对于上层协议来说是完全透明的。这样做的好处是，应用层只需要关心应用服务的实现，而不用管底层的实现。

## TCP/IP协议

## TCP三次握手

传输层协议主要有两个：TCP协议和UDP协议。TCP协议相对于UDP协议的特点是：TCP协议提供面向连接、字节流和可靠的传输

使用TCP协议进行通信的双方必须先建立连接，然后才能开始传输数据。TCP连接是全双工的，也就是说双方的数据读写可以通过一个连接进行。为了确保连接双方的可靠性，再双方建立连接时，TCP协议采用了三次握手策略。过程如图：



TCP协议三次握手的描述如下：

第一次握手：客户端将标志位SYN置为1，随机产生一个值为seq = J（J的取值范围为=1234567）的数据包发送到服务器，客户端进入SYVN\_SEND状态，等待服务器确认。

第二次握手：服务器收到数据包后由标志位SYN=1知道客户端请求建立连接，服务端将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给客户端以确认连接请求，服务端进入SYN\_RCVD状态。

第三次握手：客户端收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则jiang 标志位ACK置为1，ack=K+1，

并将该数据包发送给服务端。服务端检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，客户端和服务端进入ESTABLISHED状态，完成三次握手，随后客户端与服务端就可以开始传输数据了。

为什么需要三次握手？

三次握手的目的是，为了防止已失效的连接请求报文段突然又传送到服务端，因而产生错误。具体如下：

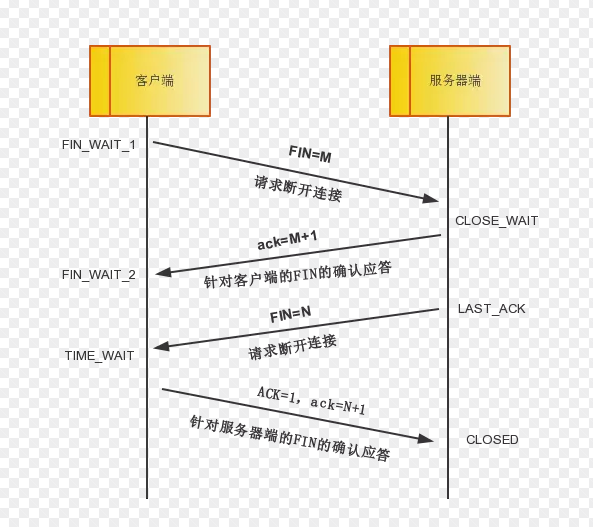
书中的例子是这样的，“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接

假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”。主要目的防止server端一直等待，浪费资源。

## TCP四次挥手

四次挥手即终止连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程邮客户端或服务端任一方执行close来触发。

由于TCP连接是全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上的数据流没了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。



中断连接端可以是客户端，也可以是服务器端。

第一次挥手：客户端发送一个FIN=M，用来关闭客户端到服务端的数据传送，客户端进入FIN\_WAIT\_1状态。意思是说“我没有数据要发给你了”，但是如果你服务器端还有数据没有发送完成，则不必着急关闭连接，可以继续发送数据。

第二次挥手：服务器端收到FIN后，发送一个ack=M+1给客户端，告诉客户端你的请求我已经收到了，但是我还没准备好，请你继续等我的消息。这时候客户端就进入FIN\_WAIT\_2状态，继续等待服务器端的FIN报文。

第三次挥手：当服务器端确定数据已经发送完成，则向客户端发送FIN=N报文，告诉客户端，好了，我这边数据发完了，准备好关闭连接了。服务器端进入LAST\_ACK状态。

第四次挥手：客户端收到FIN=N报文后，就知道可以关闭连接了，但是他还是不相信网络，怕服务器不知道关闭，所以发送ack=N+1后进入TIME\_WAIT状态。如果Server端没有收到ACK则可以chongchuan 。服务器端收到ACK后，就知道可以断开连接了。客户端等待了2MSL后依然没有收到回复，则证明服务器端已正常关闭，那好，我客户端也可以关闭连接了。最终完成四次挥手。

为什么建立连接是三次握手，而关闭连接确实四次挥手？

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN**放在一个报文里**发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

## HTTP与TCP/IP、DNS的关系



当客户端访问Web站点时，首先会通过DNS服务查询到域名的IP地址。然后浏览器生成HTTP请求，并通过TCP/IP协议发送给Web服务器。Web服务器接收到请求后会根据请求生成的响应内容，并通过TCP/IP协议返回给客户端

## HTTP缓存