### 1.document.getElementById()返回的是DOM对象，而$()返回的是jQuery对象.

### 2.jQuery对象与DOM对象也可以相互转换:

var $v =$("#v") ; //jQuery对象

var v=$v[0]; //DOM对象

var v=document.getElementById("v"); //DOM对象

var $v=$(v); //jQuery对象

### 3.Window 对象表示浏览器中打开的窗口。

如果文档包含框架（frame 或 iframe 标签），浏览器会为 HTML 文档创建一个 window 对象，并为每个框架创建一个额外的 window 对象。

### 4.[document 对象]

　该对象是window和frames对象的一个属性,是显示于窗口或框架内的一个文档。

### 5.js blob对象：

blob：二进制大对象。

file 对象只是 blob 对象的一个更具体的版本，blob 存储着大量的二进制数据，并且 blob 的 size 和 type 属性，都会被 file 对象继承。

大文件分割 (slice() 方法)

slice() 方法接受三个参数，起始偏移量，结束偏移量，还有可选的 mime 类型。如果 mime 类型，没有设置，那么新的 Blob 对象的 mime 类型和父级一样。

当要上传大文件的时候，此方法非常有用，可以将大文件分割分段，然后各自上传，因为分割之后的 Blob 对象和原始的是独立存在的。

不过目前浏览器实现此方法还没有统一，火狐使用的是 mozSlice() ，Chrome 使用的是 webkitSlice() ，其他浏览器则正常的方式 slice()。

### 6.::-webkit-scrollbar{/\* 1 \*/} /\*滚动条垂直方向的宽度与水平方向的高度\*/

::-webkit-scrollbar-button{/\* 2 \*/} /\*滚动条按钮\*/

::-webkit-scrollbar-track{/\* 3 \*/} /\*滚动条轨道\*/

::-webkit-scrollbar-track-piece{/\* 4 \*/} /\*滚动条垂直方向轨道件\*/

::-webkit-scrollbar-thumb{/\* 5 \*/} /\*滚动条轨道上的按钮\*/

::-webkit-scrollbar-corner{/\* 6 \*/} /\*滚动条轨道上的滚动角\*/

::-webkit-resizer 定义右下角拖动块的样式

### 7.$.proxy() 叫静态方法也叫工具方法 $.extend()(添加静态函数)

$().html（）叫对象方法 $.fn.extend() (添加“成员函数”)

jquery常用静态方法：

jQuery.each( object, callback )

返回值:Object

说明:

通用例遍方法，可用于例遍对象和数组。

jQuery.grep( array, callback, [invert] )

返回值: Array

说明:

使用过滤函数过滤数组元素。

jQuery.map( array, callback )

返回值:Array

说明:

将一个数组中的元素转换到另一个数组中。

jQuery.extend( [deep], target, object1, [objectN] )

jQuery.makeArray( obj )

jQuery.merge( first, second )

jQuery.inArray( value, array )

jQuery.unique( array )

jQuery.isArray( obj )

jQuery.isFunction( obj )

jQuery.trim( str )

jQuery.param( obj )

### 8.js extend:(继承)

1.原型链继承

概念：对象继承至原型对象.

核心： 将父类的实例作为子类的原型.

实现:动态添加原型对象. 直接替换原型对象. 利用**extend函数.**

function Cat(){

}

Cat.prototype = new Animal();

Cat.prototype.name = 'cat';

Prototype 对Object类进行的扩展主要通过一个静态函数Object.extend(destination, source)实现了JavaScript中的继承

2.构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

3.实例继承

核心：为父类实例添加新特性，作为子类实例返回

function Cat(name){

var instance = new Animal();

instance.name = name || 'Tom';

return instance;

}

4.拷贝继承

function Cat(name){

var animal = new Animal();

for(var p in animal){

Cat.prototype[p] = animal[p];

}

Cat.prototype.name = name || 'Tom';

}

5.组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

6.寄生组合继承

核心：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

(function(){

// 创建一个没有实例方法的类

var Super = function(){};

Super.prototype = Animal.prototype;

//将实例作为子类的原型

Cat.prototype = new Super();

})();

7.js中call和apply都可以实现继承

/\*定义一个人类\*/

function Person(name,age)

{

this.name=name;

this.age=age;

}

/\*定义一个学生类\*/

function Student(name,age,grade)

{

Person.apply(this,arguments);

this.grade=grade;

}

//创建一个学生类

var student=new Student("zhangsan",21,"一年级");

apply的一个巧妙的用处,可以将一个数组默认的转换为一个参数列表([param1,param2,param3] 转换为 param1,param2,param3)

使用： var max=Math.max.apply(null,array) （Math.max只支持Math.max(param1,param2,param3…)）

Array.prototype.push.apply(arr1,arr2)（Array.prototype.push只提供了push(param1,param,…paramN) ）

call传递多个参数，是任意形式。apply第二个参数必须是数组形式。

call, apply作用就是借用别人的方法来调用,就像调用自己的一样.

(call, apply可以获取别的对象和方法)

### 9.#jquery源码结构

(function(){

(21 , 94) 定义了一些变量和函数 jQuery = function(){};

(96 , 283) 给jQuery对象，添加一些方法和属性

(285 , 347) extend ：jQuery继承方法

(349 , 817) jQuery.extend() : 扩展一些工具(静态)方法

(877 , 2856) Sizzle : 选择器的实现（复杂）

(2880 , 3042) Callbacks : 回调对象 ：实现函数的统一管理

(3043 , 3183) Deferred : 延迟对象 ：实现异步的统一管理

(3184 , 3295) support : （浏览器）功能检测

(3308,3652):data()方法的功能：数据缓存

$("#div1").data('name',"hello");

并没有扩展到元素身上。如果扩展到对象上，会出现内存泄漏问题。避免大数据量的数据添加到元素身上，导致内存泄漏

(3653,3797):queue-队列管理(入队) dequeue(出队)

$("#div2").animate({left:100});

先把他们两个先存在队列中，执行完一个，在调用出队方法。

(3803,4299):attr(),prop(),val(),addClass()等：对元素属性的操作。

(4300,5128):on(),trigger():事件操作相关方法。

(5140,6057):DOM操作，添加，删除，获取，包装等等。DOM筛选……

(6058,6620):css()方法，专门针对样式的操作。

(6621,7854):提交的数据和ajax的操作-ajax,load,getJSON

(7855,8585):animate,运动方法，show,hide,fadeIn,fadeOut等等。

(8585,8792):offset,scrollTop,位置与尺寸的方法

(8804,8821):JQ支持模块化的一个模式，不仅支持amd的，还可以支持commonjs

(8826) windows.jQuery = windows.$ = jQuery 对外提供接口

})();

(function(){

[\_code]

})();//匿名函数，外部不能访问其中内容，减少冲突

windows.[\_function] = [\_function];//提供外部调用接口

### 10.python内置wsgi服务器为 wsgiref

### 11.实现深拷贝：

var cloneObj = JSON.parse(JSON.stringify(obj));

这会抛弃对象的constructor，也就是深复制之后，无论这个对象原本的构造函数是什么，在深复制之后都会变成Object。另外诸如RegExp对象是无法通过这种方式深复制的

slice和concat方法可以实现对数组的深拷贝： arr2=arr1.slice(0); arr4=arr3.concat();

对象的深拷贝（对象的值可能还是对象，所以要递归）

function deepCopy(p, c) {

　　　　var c = c || {};

　　　　for (var i in p) {

　　　　　　if (typeof p[i] === 'object') {

　　　　　　　　c[i] = (p[i].constructor === Array) ? [] : {};

　　　　　　　　deepCopy(p[i], c[i]);

　　　　　　} else {

　　　　　　　　　c[i] = p[i];

　　　　　　}

　　　　}

　　　　return c;

}

其它类型的深拷贝：

/\* Method of Boolean, Number, String\*/

Boolean.prototype.clone = function() { return this.valueOf(); };

Number.prototype.clone = function() { return this.valueOf(); };

String.prototype.clone = function() { return this.valueOf(); };

/\* Method of Date\*/

Date.prototype.clone = function() { return new Date(this.valueOf()); };

/\* Method of RegExp\*/

RegExp.prototype.clone = function() {

var pattern = this.valueOf();

var flags = '';

flags += pattern.global ? 'g' : '';

flags += pattern.ignoreCase ? 'i' : '';

flags += pattern.multiline ? 'm' : '';

return new RegExp(pattern.source, flags);

};

**基本的数据类型有**：`undefined，boolean，number，string，null.基本类型的访问是按值访问的，就是说你可以操作保存在变量中的实际的值

1.基本类型的值是不可变得 2.基本类型的比较是值的比较 3.基本类型的变量是存放在栈区的

**引用类型**: 对象

1.引用类型的值是可变的 2.引用类型的值是同时保存在栈内存和堆内存中的对象 3.引用类型的比较是引用的比较

typeof 进行类型检测

### 12.闭包：闭包是一个函数，这个函数有权访问另一个函数作用域中的变量

1. 在内部的匿名函数的作用域链中，实际上包含外部函数的变量对象

2. 这个内部的匿名函数被return到外部后，这个匿名函数它的作用域链被初始化了

3. 包含createCompareFunction函数的变量对象以及全局变量对象

4. 所以，匿名函数可以访问createCompareFunction函数中所有的变量

5. 更为重要的是，createCompareFunction函数执行完毕被销毁后，其变量对象还有一份保存在匿名函数中，所以仍可以访问其内部的变量.

优点：**保护函数内的变量安全,加强了封装性**

闭包的缺点就是常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。

内存泄露:用动态存储分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元

闭包是解决函数式语言的一个问题的一种技术，这个问题就是如何保证将函数当做值创造并传来传去的时候函数仍能正确运行。

**闭包作用：1.**匿名自执行函数 2.缓存(闭包维持变量不被释放) 3.实现封装

### 13.对象的constructor属性用于返回创建该对象的函数，也就是我们常说的构造函数。

创建function之后js自动创建constructor，可以被修改。‘-

在JavaScript中，每个具有原型的对象都会自动获得constructor属性。

除了arguments、Enumerator、Error、Global、Math、RegExp、Regular Expression等一些特殊对象之外，其他所有的JavaScript内置对象都具备constructor属性。

例如：Array、Boolean、Date、Function、Number、Object、String等。所有主流浏览器均支持该属性。

### 14.字面量(literal)：字面量（literal）是用于表达源代码中一个固定值的表示法。

数字（Number）字面量 可以是整数或者是小数，或者是科学计数(e)。

123e5

字符串（String）字面量 可以使用单引号或双引号 be written with double or single quotes:

"John Doe"

表达式字面量 用于计算：

5 \* 10

数组（Array）字面量 定义一个数组：

[40, 100, 1, 5, 25, 10]

对象（Object）字面量 定义一个对象：

{firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}

函数（Function）字面量 定义一个函数：

function zwj(){ return true }

与字面量相对的是：

用Array的构造函数。如：

var aValues = new Array(“string”, 24, true, null);

### 15.event,window.event:

Firefox的event只能在事件发生的现场使用

IE6/7/8支持通过window.event获取对象，通过attachEvent方式添加事件时也支持事件对象作为句柄第一个参数传入

Firefox只支持事件对象作为句柄第一个参数传入

IE9/Opera/Safari/Chrome两种方式都支持

**两个对象进行逻辑与：**

返回的是定义了的那个，如果两个都定义了，就返回第一个.

### 16.在JavaScript中,代码总是有一个上下文对象,代码处理该对象之内. 上下文对象是通过this变量来体现的, 这个this变量永远指向当前代码所处的对象中.

### 17.Deferred延迟对象：对异步统一管理

ajax返回一个Deferred对象，success、error、complete是ajax提供的语法糖，功能与Deferred对象的done、fail、always一致

$.when() 为多个操作指定回调函数

jquery的ajax返回一个受限的Deferred对象，还记得受限的Deferred对象吧，也就是没有resolve方法和reject方法，不能从外部改变状态

观察者模式是开发中经常使用的模式，这个模式由两个主要部分组成：主题和观察者。通过观察者模式，实现主题和观察者的解耦。

实现观察者模式的就是 Deferred 了.

// 注册观察者 $.when

观察者想一次性注册多个事件，那么，可以通过 then 来注册

$.get( "test.php" ).then(

function() {

alert( "$.get succeeded" );

}, function() {

alert( "$.get failed!" );

}

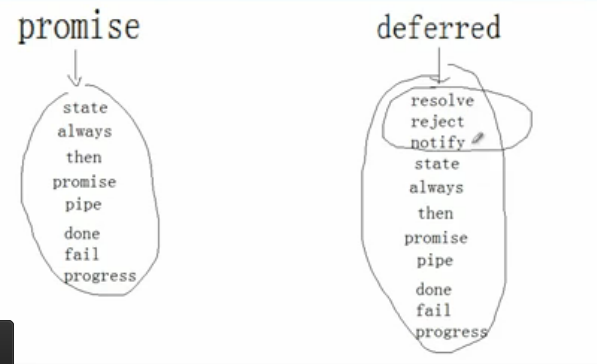
);

deferred.promise()方法。它的作用是，在原来的deferred对象上返回另一个deferred对象，后者只开放与改变执行状态无关的方法（比如done()方法和fail()方法），

屏蔽与改变执行状态有关的方法（比如resolve()方法和reject()方法），从而使得执行状态不能被改变。

deferred.resolve()手动改变deferred对象的运行状态为"已完成"，从而立即触发done()方法。

deferred.reject()这个方法与deferred.resolve()正好相反，调用后将deferred对象的运行状态变为"已失败"



Promise无法修改状态

### 18.li:

去掉前面的点：

list-style:none;

横向排列:

float:left

### 19.bootstrap:

button去边框：

.btn,.btn:focus{

border: 0;

}

**a去边框：**

a:focus,a.active:focus,a:active:focus,a.focus,a:active.focus,a.active.focus{

outline: none;

}

**边框线：**

border-bottom: 1px solid #ffefaa;

### 20.getElementById()不接受带数字的id

### 21.iframe去边框 属性frameborder="no"

### 22.<a href="#">点击a时会有bug，造成iframe网页宽度错乱，最好写成<a href="javascript:void(0)">

### 23.自调用函数：(function(){})()

### 24.rel作用规定当前文档与被连接文档的关系， stylesheet表示文档为外部样式。插件的样式一点要加这句话，不然插件样式不起作用

### 25.margin-top:是按照父元素with来计算的；

### 26.wrting-mode:vertical:vertical-lr(改变书写方向为纵向)

### 27.要使用top,left,buttom,right,要把父元素设为postion:relative。postion:absolute:将对象从文档流中拖出，其层叠通过z-index属性定义。

### 28.jquery.extend(object)扩展jquery的静态方法。$.zwj();

jquery.fn = jquery.prototype

jquery.fn.extend(object)拓展jquery对象（原型的）(实例方法)方法。

query.fn.extend({ zwj:function(){ $(this).click()} }) ==>$('#id').zwj();

### 29.具有相同class的dom，获取某一个$('.class').eq();

或者使用each()遍历$（'.class'）.each(function(index,element){})

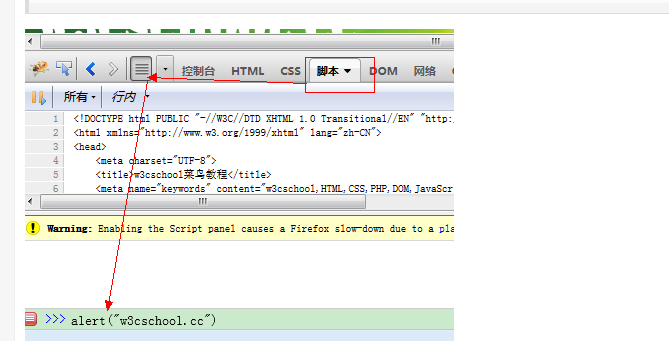
### 30.chrome consele可以直接输入js代码



Firefox的firebug也可以：



或者是：



### 31.new Date("2016.10.22 17:0:0")生成时间对象

### 32.冒泡行为：当点击内部 <span> 元素时，会触发自身及外层 <div> 和 <body> 的点击事件，这就是事件冒泡引起的.

**默认行为：比如：链接的跳转和表单的提交**

1）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。

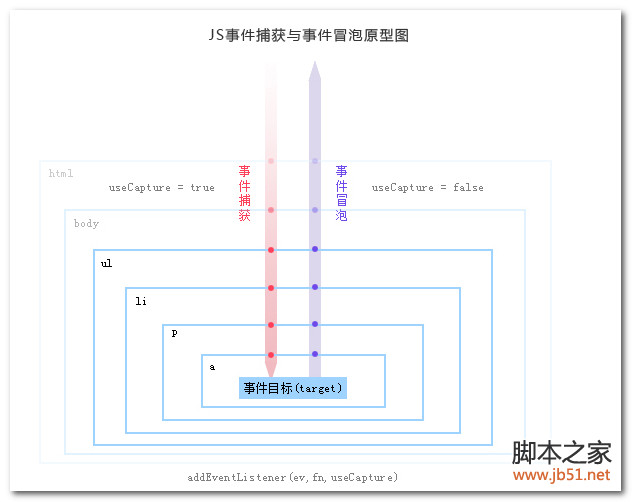
IE 5.5: div -> body -> document

IE 6.0: div -> body -> html -> document

Mozilla 1.0: div -> body -> html -> document -> window

2）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)。

3）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。



IE只支持事件冒泡，不支持事件捕获，它也不支持addEventListener函数，不会用第三个参数来表示是冒泡还是捕获，它提供了另一个函数attachEvent

阻止浏览器的默认行为preventDefault()

停止事件冒泡 stopPropagation()

$('a').click(function (e) {

e.preventDefault();

});

//禁止提交表单跳转

$('form').submit(function (e) {

e.preventDefault();

});

注意：如果想让上面的超链接同时阻止默认行为且禁止冒泡行为，可以把两个方法同时

写上：event.stopPropagation()和 event.preventDefault()。这两个方法如果需要同时启用的时候，

还有一种简写方案代替，就是直接 return false。

$('a').click(function (e) {

return false;

});

addEventListener(‘type’,function,true);第三个参数为true表示采用事件捕获。默认为false。

**先执行捕获**（从文档根节点开始，捕获事件对象）-》目标（目标事件位置）-》**冒泡**（从目标位置开始，向外发生。）

**先处理事件-》默认行为**

大多数情况事件处理函数先执行，再执行默认事件，例外：input的checkbox默认事件会先于事件处理函数。

### 33.AMD模块

**定义：define（id,dependencies,factory）**

**模块和依赖是可选的，factory是实际代码。**

id：指定义中模块的名字，可选；如果没有提供该参数，模块的名字应该默认为模块加载器请求的指定脚本的名字。如果提供了该参数，模块名必须是“顶级”的和绝对的（不允许相对名字）。

**模块名的格式**

模块名是用正斜杠分割的有意义单词的字符串，

**单词须为驼峰形式**，或者"."，".."，

模块名不允许文件扩展名的形式，如“.js”，

模块名可以为 "相对的" 或 "顶级的"。如果首字符为“.”或“..”则为相对的模块名，

顶级的模块名从根命名空间的概念模块解析，

相对的模块名从 "require" 书写和调用的模块解析，

模块名用来唯一标识定义中模块，它们同样在依赖性数组中使用。

define("alpha", ["require", "exports", "beta"], function (require, exports, beta) {

exports.verb=function() {

return beta.verb();

//Or:return require("beta").verb();

}

});

**AMD需要在声明模块时指定所有的依赖，通过形参传递依赖到模块内容中。**

**define(['dep1','dep2'],function(dep1,dep2){**

**return function(){};**

**});**

**模块引用:require(['math'], function (math) {**

**math.add(2, 3);**

**});**

[**CommonJS**](http://wiki.commonjs.org/wiki/Modules/1.1)**规范：**

**定义：模块引用，模块定义，模块****标识**

**模块引用：var math = require('math');**

**模块定义：exports.add = function(){ }**

**模块标识:**

**模块标识其实就是传递给require（）方法的参数,**id 通常是文件的完整路径**，由"\"分开的词组成，每个词必须使用驼峰式，或者是"."或".."。熟悉路径的同学应该知道"."表示当前路径，".."表示上一级路径。在这里，"."表示当前模块，".."表示上一级模块。**

[**module.export 和 export方法的区别**](http://blog.csdn.net/cike110120/article/details/12753253)**:**

**假设这是rocker.js文件**

**exports.name = function() {**

**console.log('My name is Lemmy');**

**};**

**var rocker = require('./rocker.js');**

**rocker.name();**

**module.exports = function(name, age) {**

**this.name = name;**

**this.age = age;**

**this.about = function() {**

**console.log(this.name +' is '+ this.age +' years old');**

**};**

**};**

**var Rocker = require('./rocker.js');**

**var r = new Rocker('zero', 62);**

**模块是一个特定的类型就用Module.exports。**

**exports其实是module.exports的引用。**

**模块兼容：**

**// 检测上下文环境是否为AMD或CMD**

**var hasDefine = typeof define === 'function',**

**// 检查上下文环境是否为Node**

**hasExports = typeof module !== 'undefined' && module.exports;**

**if (hasDefine) {**

**// AMD环境或CMD环境**

**define(definition);**

**} else if (hasExports) {**

**// 定义为普通Node模块**

**module.exports = definition();**

**} else {**

**// 将模块的执行结果挂在window变量中，在浏览器中this指向window对象**

**this[name] = definition();**

**}**

### 34.匿名函数：

**在前面加上一个布尔运算符（只多了一个感叹号），就是表达式了**

**!function(){}()**

**// 这么写会报错，因为这是一个函数定义：function() {}()**

**// 常见的（多了一对括号），调用匿名函数：**

**(function() {})()**

**;(function() {})()**

**在前面加分号可以有多种用途:**

**防止多文件集成成一个文件后，高**[**压缩**](http://www.cr173.com/k/yasuo/)**出现语法错误。**

### 35.Automatic Semicolon Insertion (ASI, 自动分号插入机制)

ASI机制不是说在解析过程中解析器自动把分号添加到代码中，而是说解析器除了分号还会以换行为基础按一定的规则作为断句的依据，从而保证解析的正确性。

### 36.noConflict: function( deep ){}

**//处理冲突，因为有些库可能会用到$甚至jQuery，这时就可以调用chaojidan = $.onConflict(true);这时chaoijdan就可以当做jQuery了,并且其他库可以使用$标识了**

### 37.$(function(){}) = $(document).ready(function(){})等dom加载完

**window.onload() = function（）｛｝是等页面资源加载完**

**DOMContentLoaded当所有DOM解析完以后会触发这个事件**

**DOMContentLoaded是如何在document对象中被触发的;**

**document.addeventListener('DOMContentLoaded',function(){...},false);**

### 38.js click() 方法可模拟在按钮上的一次鼠标单击

**setInterval(function(){ document.getElementById("su").click(); },1000)**

### 39.作用域链：

**在函数内部会创建一个[[Scope]]属性，这个属性指向一条作用域链。**

**with语句可以临时在作用域链的前端临时增加一个普通对象。**

**当代码在一个环境中执行时，会创建变量对象的的一个作用域链（scope chain）。作用域链的用途，是保证对执行环境有权访问的所有变量和函数的有序访问。**

**作用域链的前端，始终都是当前执行的代码所在环境的变量对象。如果这个环境是一个函数，则将其活动对象作为变量对象。**

**每一个函数都有自己的执行环境，当执行流进一个函数时，函数环境就会被推入一个环境栈中，而在函数执行之后，栈将其环境弹出，把控制权返回给之前的执行环境，这个栈也就是作用域链。**

**如果执行环境是函数，那么将其活动对象（activation object, AO）作为作用域链第一个对象，第二个对象是包含环境，下一个是包含环境的包含环境。**

**函数的[scope]属性：每个函数在定义的时候（生成函数实例的时候）都会分配一个[scope]属性，这个属性指向的当前的“作用域链”。这个属性开发人员是访问不到的，只有javascript能访问。**

**作用域链：当函数调用时，javascript引擎会维护一个这次调用的作用域链，这个作用域链条是函数的[scope]指向的作用域链加上函数调用时的活动对象，形式如[ 活动对象, 函数定义时的作用域链条]。**

### 40.Restful

**REST（英文：Representational State Transfer，简称REST）翻译是"表现层状态转化".**

**所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。**它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。

**"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。**

比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可

以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

**如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State**

**Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。**

客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方

式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。

**总结一下什么是RESTful架构：**

**（1）每一个URI代表一种资源；**

**（2）客户端和服务器之间，传递这种资源的某种表现层；**

**（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。**

### 41.event.stopPropagation() 在内层使用

### 42.base64为什么可以减少请求次数？

**使用base64，图片数据已经保存到页面，所以无需再发送请求**

### 43.oncontextmenu:在元素中用户右击鼠标时触发并打开上下文菜单

**通过document.getElementById(“div”).oncontextmenu = function(){ return false;}取消默认右键.**

### 44.Object.prototype.toString.call(obj) === '[object Array]'

**使用toString()判断类型是最直接有效正确的方式，没有bug.typeof，instanceof都有问题。**

### 45.object.hasOwnProperty(proName);

**判断proName的名称是不是object对象的一个属性或对象。不能检测原型链中的属性,使用object.prototype.hasOwnProperty（）**

### 46.$.proxy更改this指向：

**$.proxy(show,document)()**

**function show(){**

**alert(this);**

**}**

### 47.FileReader.readAsDataURL()读取图片，并且把读取的文件赋值给img的src:src = reader.result;可以在浏览器看到图片。

**FileReader.readAsBinaryString(file)将文件读取为二进制码**

**FileReader.readAsText(file, [encoding])将文件读取为文本**

### 48.jquery对属性的操作：

**.**

prop() 方法设置或返回被选元素的属性和值,prop方法是高版本的jquery引入的。

对于HTML元素本身就带有的固有属性，在处理时，使用prop方法。

对于HTML元素我们自己自定义的DOM属性，在处理时，使用attr方法。

<input id="chk1" type="checkbox" />是否可见  
<input id="chk2" type="checkbox" checked="checked" />是否可见

像checkbox，radio和select这样的元素，选中属性对应“checked”和“selected”，这些也属于固有属性，因此需要使用prop方法去操作才能获得正确的结果。

$("#chk1").prop("checked") == false

$("#chk2").prop("checked") == true

如果上面使用attr方法，则会出现：

$("#chk1").attr("checked") == undefined

$("#chk2").attr("checked") == "checked"

返回 Value 属性

**返回第一个匹配元素的 value 属性的值。**

**$(selectot).val()**

**设置 Value 属性的值**

**$(selector).val(value)**

### 49.web中全局环境被认为是window对象。

**代码在一个自己的执行环境时，会创建变量对象的一个作用域链。**

**延长作用域链：**

**with语句会将指定的对象添加到作用域链中。**

**js没有块级作用域：**

**if语句中的变量声明会将变量添加到当前执行环境，语句结束也依旧存在于循环外部的**

**执行环境中。**

### 50.js继承：

**原形链：**

**child.prototype = new parent();**

**构造函数：**

**function parent(){**

**this.colors = ["1","2","3"];**

**}**

**function child(){**

**parent.call(this);**

**}**

**组合继承：**

**function parent(name){**

**this.colors = ["1","2","3"];**

**this.name="zzz";**

**}**

**function child(name,age){**

**this.age = age ;**

**parent.call(this,name);**

**}**

**原型继承：**

**function object( o ){**

**function f(){};**

**f.prototype = o ;**

**return new f();**

**}**

**es5提供：object.create();**

**var person = {**

**name : "zwj"**

**};**

**var onePerson = Object.create(person);**

**寄生式继承：**

**function createF(original){**

**var clone = object(original);**

**clone.say = function(){**

**alert("say");**

**};**

**return clone;**

**}**

**寄生组合式继承：**

**function parent(name){**

**this.name="zzz";**

**}**

**parent.prototype.say = function(){**

**alert(this.name);**

**}**

**function child( name , age){**

**parent.call(this ,name);**

**this.age = age ;**

**}**

**function inheritprototype(parent,child){**

**var prototype = object(parent.prototype);**

**prototype.constructor = child;**

**child.protype = prototype ;**

**}**

**inheritprototype(parent,child);**

**child.prototype.say = function(){**

**alert(this.age);**

**}**

### 51.object.preventExtensions():阻止给对象添加属性

**var zwj = {};**

**object.preventExtensions(zwj);**

**object.seal();密封对象。属性不可添加，删除**

**object.freeze();冻结对象。属性不可添加，删除，修改**

### 52.cors:

**请求另一个域中的资源，xhr.open(),传入绝对路径。**

**可以访问status，statusText。**

**不能使用setRequestHeader()**

**不能发送cookie，**

**调用getAllResponeseHeader()返回空字符串**

### 53.SSE（Server-Sent Events，服务器发送事件）是围绕只读Comet 交互推出的API 或者

模式。

**SSE API允许网页获得来自服务器的更新（HTML5），用于创建到服务器的单向连接，服**

**务器通过这个连接可以发送任意数量的数据。服务器响应的MIME类型必须是text/event-stream，而且是浏览器中的JavaScript API 能解析格式输出。SSE 支持短轮询、长轮询和HTTP 流，而且能在断开连接时自动确定何时重新连接。**

**xhr和sse也能实现双向通信**

### 54.函数科里化：

**函数柯里化，是固定部分参数，返回一个接受剩余参数的函数，也称为部分计算函数，**

**目的是为了缩小适用范围，创建一个针对性更强的函数。**降低适用范围，提高适用性.

柯里化有3个常见作用：**1. 参数复用**；**2. 提前返回；3. 延迟计算/运行**

**例：**

**var adder = function(num) {**

**return function(y) {**

**return num + y;**

**}**

**反柯里化函数，从字面讲，意义和用法跟函数柯里化相比正好相反，扩大适用范围，创**

**建一个应用范围更广的函数。使本来只有特定对象才适用的方法，扩展到更多的对象。**

**Js科里化:**根据定义来说，柯里化就是将一个接收“多个”参数的函数拆分成一个或者许多个接收“单一”参数的函数。

**var currying = function(fn) {**

**// fn 指官员消化老婆的手段**

**var args = [].slice.call(arguments, 1);//获取第二个一个参数,’即例子中的’"合法老婆"**

**// args 指的是那个合法老婆**

**return function() {**

**// 已经有的老婆和新搞定的老婆们合成一体，方便控制**

**var newArgs = args.concat([].slice.call(arguments));**

**// 这些老婆们用 fn 这个手段消化利用，完成韦小宝前辈的壮举并返回**

**return fn.apply(null, newArgs);**

**};**

**};**

**// 下为官员如何搞定7个老婆的测试// 获得合法老婆**

**var getWife = currying(function() {**

**var allWife = [].slice.call(arguments);**

**// allwife 就是所有的老婆的，包括暗渡陈仓进来的老婆**

**console.log(allWife.join(";"));**

**}, "合法老婆");**

**// 获得其他6个老婆**

**getWife("大老婆","小老婆","俏老婆","刁蛮老婆","乖老婆","送上门老婆");**

**// 换一批老婆**

**getWife("超越韦小宝的老婆");**

### 55.自定义事件：

**使用EventTarget();**

**var target = new EventTarget();**

### 56.eval() 函数可计算某个字符串，并执行其中的的 JavaScript 代码。

**document.write(eval("2+2"))**

### 57.web计时:

**核心performance.navigation。提供了度量页面性能的一些属性。**

### 58.函数节流：

**某些代码不可以在没有间断的情况下连续重复执行.简单地讲，就是让一个函数无法在很短的时间间隔内连续调用，只有当上一次函数执行后过了你规定的时间间隔，才能进行下一次该函数的调用.函数节流的原理挺简单的，那就是定时器。当我触发一个时间时，先setTimout让这个事件延迟一会再执行，如果在这个时间间隔内又触发了事件，那我们就clear掉原来的定时器，再setTimeout一个新的定时器延迟一会执行.**

**var throttle = function(fn, delay){**

**var timer = null;**

**return function(){**

**var context = this, args = arguments;**

**clearTimeout(timer);**

**timer = setTimeout(function(){**

**fn.apply(context, args);**

**}, delay);**

**};**

**};**

**它使用闭包的方法形成一个私有的作用域来存放定时器变量timer。而调用方法为**

**window.onresize = throttle(myFunc, 100);**

### 59.鸭子类型：

**鸭子类型的通俗说法是：“如果它走起路来像鸭子，叫起来也是鸭子，那么它就是鸭子。”在鸭子类型中，关注的不是对象的类型本身，而是它是如何使用的。...在使用鸭子类型的语言中，这样的一个函数可以接受一个任意类型的对象，并调用它的走和叫方法....这是由于动态类型语言对变量类型的宽容给实际编码带来了很大的灵活性。由于无需进行类型检测，我们可以尝试调用任何对象的任意方法，而无需去考虑它原本是否被设计为拥有该方法。**

### 60.arguments 是JavaScript里的一个内置对象，在函数体内，标识符arguments是指向实参对象的引用，他像数组但不是数组，实参对象是一个类数组对象 arguments[]具体参数，arguments.length参数个数.

**转换成数组：var args = Array.prototype.slice.call(arguments); 那么现在这个变量args就含有一个含有函数所有参数的标准javascript数组对象.**

### 61.C语言如何读取zip文件内容？

**打开方式用二进制方式打开,如 FILE \*fp = fopen("a.zip","rb+");然后读用fread去读。**

**对于程序来说，不管后缀名如何，文件分为两种类型：文本文件和二进制文件。**

### 62.js中有5种基本数据类型：Undefined、Null、Boolean、Number和String，Symbol(es6)。（基本数据类型：内存地址存放的是变量值，按值访问）

**引用数据类型：Object（引用数据类型：内存存放的是引用）**

### 63.python：

**不可变对象：int,string,float,tuple(相同的值有相同的地址，给以不同值时会重新分配内存)**

**可变对象：dict,list（赋值是创建了一个对象的引用）**

**不同的数据类型可以通过字面量区分。**

### 64.在js中所有对象都基于Object.

**Object.prototype.toString返回对象字符串表示。此方法获取this对象的[[class]]属性的值，在es5中[[class]]有Arguments,Array,Boolean,Date,Error,Function,JSON,Math,Number,Object,RegExp,String.特殊的undefined，Null.**

**typeof arry,null,Date,正则都会认为是Object.**

**instanceof 会认为[]是Object.**

### 65.js清空数组：

**arry.splice(0,arry.length);**

**arry=[];//这不是严格意义的删除，赋空值，等待垃圾gc回收**

**arry.length = 0;**

### 66.bootsrap 布局：

网格：需要包含在.container中

.row可快速构建网格

<div class="container">

<div class="row">

<div class="col-lg-2"></div>

</div>

<div class="row"></div>

</div>

<div class="container"></div>

</div>

响应式导航：

<div class="navbar navbar-default">

<div class = "container">

<div class="navbar-header">

<button type="button" class="navigation" data-toggle="collapse" data-target=".navbar-collapse">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

</div>

<div class="navbar-collapse" role="navigation">

<ul class="nav navbar-nav">

<li class="hidden-sm hidden-md"><a href="javascript:void(0)" target="\_blank"></a></li>

<li><a href="javascript:void(0)" target="\_blank">1</a></li>

<li><a href="javascript:void(0)" target="\_blank">2</a></li>

<li><a href="javascript:void(0)" target="\_blank">3</a></li>

<li><a href="javascript:void(0)" target="\_blank">4</a></li>

<li><a href="javascript:void(0)" target="\_blank">5</a></li>

</ul>

<ul class="nav nav-bar navbar-right hidden-sm">

<li><a href="javascript:void(0)">登录</a></li>

</ul>

</div>

</div>

</div>

默认导航栏.navbar .navbar-default（白底黑字）.navbar-inverse(黑底白字) role="navigation"有助于增强访问性

折叠内容必须包裹在.collapse,.navbar-collapse的<div>中，data-toggle告诉js需要对按钮做什么，data-target指示要切换到哪个元素。.navbar-fixed-top让导航栏固定在顶部。（.nav-fixed-bottom固定在尾部）

### 67.promise对象

promise以同步的操作流程，操作异步事件，避免层层嵌套

Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供[统一接口](http://wiki.commonjs.org/wiki/Promises/A)。

简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。

Promise其实是状态机：

有三种状态：”pending”, “resolved”, “rejected”

相应的状态的事件处理函数,并注册回调处理函数：then， done， fail， always

更新对应状态，并且调用注册函数：resolve， reject

**Github实现的：https://github.com/zmrzwj/zwj/blob/master/%E7%BB%83%E4%B9%A0%E9%A2%98/%E9%9D%A2%E8%AF%95%E9%A2%98/9.12.01%20%E6%89%8B%E5%86%99%E4%B8%80%E6%AC%BE%20Promise.html**

### 68.<meta name="viewport" content="width=device-width, initial-scale=1">

viewport 手机把页面放在一个虚拟的窗口（viewport）中，这样就不会把每个页面挤到很小的窗口中。

**width=device-width:程序宽度与屏幕宽度一样**

**height=device-height：程序高度与屏幕高度一样**

**initial-scale=1：应用启动时缩放尺寸**

**minimum-scala=1.0:用户可以缩放的最小尺寸**

**maximum-scale=1.0:用户可以放大的最大尺寸**

**user-scalable = no:用户可以通过手势来缩放应用**

### 69.meta :

**属性：http-equiv name sheme**

**值：**（content-type,expires,refresh,set-cookie）,(author,description,keywords,generator,revised,others),(some\_text)

<meta http-equiv="content-type" content="text/html;charset=gb2313">

先设置属性为content-type或expires,再设置content=....

### 70.<script type="text/script">

**var divs = getElementsByTagName('div');**

**var ps = getElementsByTagName('p');**

**</script>**

### 71.隐式迭代:

**在jq中，所有的方法都被设计成自动操作对象集合，而不是单一的一个对象，这样避免了大量的没有意义的循环**

$('.className') 这其实是一个数组集合，并不是直接的DOM元素

所以像 $('.className').addClass('newClass')，执行的其实是类似这样的语句

$('.className').each(function() {  
$(this).addClass('newClass');  
});

### 72.var reg = /^(?:#([\w-]+)|.([\w-]+)|([\w]+)|(\\*))$/ //id,class,标签，\*的选择器的正则表达

### 73.var m=reg.exec(selector);

**//exec 返回的是数组，如果有匹配返回第一个匹配的字符串内容。，没有就返回null**

### 74.$('.dv','p');===$('p .dv');

**$('.dv','p')支持dom,jquery对象，$('.dv',document.body)===$('.dv',$('body'));**

### 75.沙箱：一个独立的运行环境,js沙箱处理模块依赖关系的闭包被称为沙箱。

### 76.能力检测：

能力检测：

var support={};

support.getElementsByClassName = !!document.getElementsByClassName;

//不仅要判断是否存在，还有判断能力是否符合要求。

support.getElementsByClass = (function(){

var isExist = !!document.getElementsByClassName();

if(isExist && typeof document.getElementsByClassName === 'function'){

var div = document.createElement('div'),

divWithClass = document.createElement('div');

divWithClass.className = 'c';

div.appendChild(divWithClass);

return div.getElementsByClassName('c')[0] === divWithClass;

}

})();

if(support.getElementsByClassName){

return node.getElementsByClassName(className)

}else{

自己的代码（className）;

}

### 77.str.replace(/^\s+|\s+$/g,'');//去空格

### 78.if(document.getNameClass){

**return document.getNameClass(class);**

**}else{**

**}**

### 79.document.documentElement:返回文档根节点，即html元素

### 80.HTML NodeList节点列表：

addEventListener("click",function(){})添加事件

appendChild()添加子元素

element.className = "zwj"

element.id = "zwj"

getAttribute("name")返回name属性值

setAttribute("")设置属性值

hasAttribute()元素中存在指定的属性

removeAttribute()删除指定属性

removeEventListener()删除事件 IE8使用detachEvent(),attachEvent()

replaceChild() 替代之元素

element.style 设置样式

removeChild()删除子元素

### 81.1.利用动态特性添加成员

**2.覆盖原型对象实现继承时，一点要给新对象添加constructor属性，以便模拟对象的类型，但是如果对对象要求不严格可以忽略。**

**3.利用组合式继承添加原型成员 对象.extend(对象)**

**原生对象污染**

### 82.以边框计算宽和高

**默认以内容计算宽度**

**以边框开始计算宽度：**

**box-sizing:border-box;**

**-webkit-box-sizing: border-box;**

### 83.width:100%;

**max-width:640px;**

**margin:0 auto;//标准容器会居中**

### 84.postion:fixed;是脱离标准流的，以window计算，left,top;等

1:1的显示在移动设备当中图标会失真。

压缩图标

### 85.实现自己的each方法：//函数没有返回值时，结果为undefined

var each = function(arr,fn){

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

//fn(arr[i]);

if(fn.call(arr[i],i,arr[i]) === false){//call改变this

break;

}

}

}

### 86.自己实现getElementsByClass:

var getClass = function(className,results){

var results = results || [];

if(document.getElementsByClass(className)){

results.push.apply(results,document.getElementsByClass(className));

}else{

tempArr =document.getElementsByTagName('\*');

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

if((' '+tempArr[i].className + ' ').indexOf(' '+className + ' ') !== -1){

results.push(tempArr[i])

}

}

}

}

### 87.实现push方法://try..catch..性能有损耗。

**把try ... catch..写在外面**

var getTag = function(tag,results){

results = results || [];

try{

results.push.apply(results,document.getElementsByTagName(tag));

}catch(e){

myPush(results,document.getElementsByTagName(tag));

}

return results;

}

var myPush = function(target,els){

var j = target.length,

i = 0;

while((target[i++] = els[i++])){}

target.length = j - 1;

};

### 88.作用域是变量和函数可以访问的范围。

变量的作用域有全局作用域和局部作用域。在代码任何地方都可以访问的对象有全局作用域。局部作用域只在固定的代码可以访问。

作用域链ECMA规定了一个内部属性[[scope]]，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数作用域链，它决定哪些数据可以被访问。

代码优化：全局变量总是位于运行上下文作用域链的最末端，因此在标识符解析的时候，查找全局变量是最慢的。

只有函数才会创建作用域

### 89.iframe父子页面通信

父页面调用子页面：

document.getElementById('id').contentWindow.func()

document.getElementById('id').contentWindow.document.getElementById("子元素id")

通过iframe的name获取窗口，iframeName.window.func();iframeName.window.document.getElementById("子元素id");

通过frames[]数组直接获取，window.frames['iframeName'].func();window.frames['iframeName'].document.getElementById("子元素id");

子页面调用父页面：

parent.window.func();parent.window.getElementById("父元素id")；

top.window.func();top.window.getElementById("父元素id")；

兄弟访问：

parent.window.document.getElementById("iframe").contentWindow.func();

parent.window.document.getElementById("iframe").contentWindow..document.getElementById("子元素id");

### 90.中文换行：white-space: pre-wrap;

### 91.IIFE立即执行函数

+，!,-等一元运算符可以引导解释器，指明运算符是一个表达式。任何把函数声明变成表达式的作法，都可以使解释器正确调用定义函数。

### 92.css transition:元素从一种样式变为另一种样式时添加效果。

transition-property:过度元素属性名。

transition-duration：过渡花费时间，默认0.

transition-timing-function：过渡的时间曲线，默认ease。

transition-delay：过渡效果从何时开始。

简写：transition：width 2s linear 2s;

### 93.css transform:

rotate():旋转 eg:transform:rotate(10deg)'deg'是度的意思。

skew():倾斜 eg:transform:skew(10deg)

scale():放大 eg:transform:scale(1.5)

translate():位移 eg:transform:translate(120px,0) 右移动120px

查看3D时，需要添加透视perspective。

transform-origin:设置旋转元素的基点位置。支持百分比

transform-style:flat(不保留) | preserve-3d(保留) 使被转换的子元素保留其3D位置。

rotate3d(x,y,z,deg),x,y,z表示x,y,z轴坐标方向矢量。deg为旋转角度。

### 94.伪类：

：first-child:第一个子类元素

：active：被激活的元素

：focus：拥有键盘输入焦点的元素

：link:未被访问的链接添加样式。

：visited:已经访问的链接。

：not(selector):除..以外的元素。

：nth-child(n)：第n个子元素

：nth-last-child(n):倒数第n个元素

：first-letter:选择元素的第一个字母

：before ：after要配合content使用。

**::（双冒号）表示伪元素**

::before ::after ::first-line:当前元素的第一行。

::first-letter选择第一个字母

::selection:当前元素中，选中的文字

**a:hover::before伪类和伪元素合用**

利用before做三角：

div::after{

width:0;

heigth:0;

border-top:20px solid red;

content:"";

}

**也可利用before，after做左右横线中间文字**

### 95.animation:

animation-name:keyframe名称。

animation-duration：动画完成时间。

animation-timing-function:linear线性 ease先低后快

animation-delay：动画开始前的延迟。

animation-iteration-conut:动画应该播放的次数 infinite:无限次播放 10：播放10次

animation-direcation:是否轮流反向播放动画。alternate:轮流反向播放

@keyframes mymove{

from{top:0px}

to{top:200px}

}

@keyframes mymove{

0%{top:0px}

25%{top:50px}

50%{top:100px}

75%{top:150px}

100%{top:200px}

}

### 96：！important:设置同名属性的优先级。

### 97：box-sizing:border-box;元素的内边距和边框不会增加元素宽度

**box-sizing:content-box;在宽高之外计算元素的内边距和边框。**

**box-sizing:inherit;从父元素继承box-sizing属性值**

### 98.table-layout:fixed;保持单元格的等宽。

### 99.calc()函数：元素设置动态值。

**width：calc(100%-100px);**

### 100.渐变：linear-gradient(起始点（top,left,left top）,起始色，结束色)

**radial-gradient:径向渐变**

**repeating-linear-gradient:重复渐变。**

**(ie9以下用滤镜解决filter)**

### 101.transparent：=rgba(0,0,0,0) 全透明黑色的速记法。

### 101.Object.defineProperty(object(待修改的对象),name(属性名),descriptor(修饰符，目标属性拥有的特性，要求为对象))

修饰符的属性设置：

｛

configurable:true;//可配置true可从父元素删除，false无法修改

enumerable:true;//可以通过for in来遍历，可用prototyIsEnumerable来判断属性是否可枚举。

writable:true;//值是否可以修改

value:xxx;

｝

### 103.Math.max.apply(null ,arry)求数组最大值

### 104.instanceof :array也是Object

**Array.prototype.concat = function(){**

**var arr = this.slice(0); //从以有的数组中选取元素，slice(start,end)左闭右开，end没有规定会选取从开始到结束的所有元素。**

**arguments.forEach(function(x){**

**if(x instanceof Array){**

**x.forEach(function(y){**

**arr.push(y);**

**})**

**}else{**

**arr.push(x);**

**}**

**}**

**)**

**return arr;**

**}**

### 105.声明对象的三种方式：let person = {} ; let cat = new Object(); let dog = Object.create(null);

### 106:'use strict' 严格模式 这句话不在首行则无效。

全局变量显式声明。

禁止with语句。

禁止this指向全局变量。

禁止在函数内部遍历调用栈。function f(){ f.arguments;//报错 }

禁止删除变量。除非configurable=true;

对象不能有重名属性。

函数不能有重名参数。

禁止8进制。

不允许对arguments赋值。

arguments不在追踪参数变化。

禁止使用argument.callee

函数必须声明在顶层

### 107:es6:

**（1），let:块级作用域有效，（会出现暂时性死区）**

var a=1;

if(true){

a=2;(报错)

let a = 3;

}

**let:不允许重复声明。**

**const:声明只读变量。**

**const pi =3.14 //不允许修改 ，实质保证内存地址不得改动**

**（2），解构赋值**

let [a,[b],d] = [1,[2,3],4];

let {foo,bar} = {foo:"aaa",bar:"bbb"};

解构失败值为undefined

**（3）字符串扩展**

**j**s内部字符以utf-16存储，每个字符固定2个字节。

字符串遍历器for of :

for(let code of 'foo'){

cosole.log(code);

}

es5:charAt(),es6:at()返回给定字符的位置

normalize():将字符的不同方法统一，称为unicode正规化。

includes(),starsWith(),endsWith()查找子串

repeat();将字符串重复n次，返回新字符串。‘hello’.repeat(3)

padStart(),padEnd()字符串补全。

模版字符串：``(反引号)可以在字符串中嵌入变量，以$(变量名)的形式书写.标签模版alert`123` === alert(123)

**(4)数值扩展：二进制（Ob）和八进制(Oo),isFinite(),isNaN(),NaN运算结果非数值。极小常量Number.EPSILON.安全数Number.isSafeInteger()-2^53-2^53.**

**（5）数组扩展：**

Array.from().将对象转化为数组

es5:[].slice.call(json); es6:Arry.from(json);

Arry.of(3,11,8)将一组数转为数组。

fill()方法：填充一个数组。

includes():是否包含。

entries()键值对遍历,keys()对键名的遍历,values()键值的遍历

indexof对NaN会误判。

Array(3) = [,,,]数组空位

**（6）函数扩展：**

length属性返回没有指定默认值的参数个数。

...扩展运算符，将数组变为参数序列。

console.log(1,...[2,3,4],5);

...取代apply方法Max.max.apply(null,[14,3,77]); Math.max(...[14,3,77]);

[...'hello']将字符串转为数组。

扩展运算符调用的是iterator接口，没有iterator接口会报错。

箭头函数：=>

var f = x=>x; === var f =function(x){ return x ;} 注意：不可使用arguments对象，不可使用yield命令。

::绑定运算符 foo::bar； === bar.bind(foo);

尾调用：某个函数最后一步是调用另一个函数；

function f(x){ return g(x); }

尾调用优化：只保留内层函数的调用帧。正常模式无效，严格模式有效。

参数尾逗号function a(x,c,){ }

**(7)对象的扩展**

允许在对象内直接写变量。

简写：｛

birth, // === birth:birth

hello(){} //=== hello:function(){}；

允许字面量键，[keya] : 'value'; var keya = {a:1};当键为对象时会被转换为[object object]的字符串。必须放到[]中。

Object.getOwnPropertyDescriptor方法获取属性的描述符。

**（8）symbol函数：**

let s = Symbol();s是独一无二的值。

魔术字符串：在代码中多次出现，与代码强耦合的具体字符串或数值。

重新使用同一个Symbol值：var let = Symbol.for('foo');

｝

**（9）set和map**

new Set([1,2,3]); add(),delete(),has(),clear(),keys(),values(),forEach()

new Map();set(),get(),has(),forEach(),keys(),values()

set,map,json,array可以互转

**（10）proxy**

修改某些默认操作，相当于在目标对象前架设一层拦截器，new Proxy();

**(11)以后把一些Object的方法放到reflect上。**

**（12）promise:解决异步的方案。**

三种状态：pending(进行),resolved（完成）,rejected（失败）.

resolve(),reject()。 promise.then(functon(){},function(){})一个回调为完成，一个回调为失败

Promise.all([p1,p2,p3])多个promise对象，当有一个reject时，返回第一个reject的实例的返回值。

cath()捕获失败，done()像then一样捕获Fulfilled和Rejected,finally()无论promise最后怎样都会运行。

**（13）iterator：**

遍历过程：创建一个指针，再调用next()方法访问下一个。当返回为done时遍历结束。

Symbol.iterator方法返回一个遍历对象。

**（14）generator生成器：**

语法：function\* xx(){ yield ;}

generator相当于是分段执行，到yield暂停，next继续执行。

执行到yield后面的语句不会马上执行。yield类似return语句。

function\* g(){

yield 1;

yield 2;

yield 3;

}

g.next();

异步编程方法4种：回调函数，事件监听，发布/订阅，promise对象

**（15）,async函数**

async和await一起使用。

async function f(){ return await 123; }

**(16).class**

class foo{

constructor(){}

}

一个类必须有constructor方法，没有显示定义，也会默认添加一个空constructor.

class不存在变量提升（hoisting,把变量提升到作用域开始部分）。

Object.setPrototypeOf(B.prototype,A,prototype);将指定的对象的原型设置为另一个对象。

Object.getPrototypeOf(child);用来从子类上获取父类。

super();代表父类构造函数。但内部this指向自己。

new.target,返回new命令作用的那个构造函数。

class内部调用new.target返回当前class.

**（17），decorator装饰器，对类的改变是编译时发生的不是运行时发生的。**

语法：@

类的修饰：function a(target){ target.isa = true; }

@a

class A{ }

//相当于 A = a(A) || A;

**类方法的修饰：**

装饰器函数介绍3个参数：目标对象，属性名，属性描述。

function defun(target,name,descriptor){ return descriptor; }

多个装饰器可以一起使用。（由于存在函数提升，所有无法把装饰器用在函数上）

Mixin:混入，在一个对象中混入另一个对象。

**（18），module**

import导入模块

export模块对外接口。

var name="zwj"; var age=2; export ｛name,age｝;

as重命名：function fun(){} fun as zwj;把fun重命名为zwj.

export default function(){} ;//mou.js 不给模块取名默认输出。

使用时 import zwj from './mou';名字zwj可以随便给。

<script src="" defer>脚本会异步加载，等到页面全部渲染结束

<script src="" async>一旦下载完就会停止渲染，执行脚本。

**（19），二进制数组**

var buf = new ArrayBuffer(32);一段32字节的内存。

fetch(url).then()取回数据api也是ArrayBuffer对象。

**（20），SIMD：单指令**

常用于矢量运算。

SIMD提供了12种数据类型。都是128个二进制。

Float32x4:4个32位浮点数。

### 108.csp(内容安全政策)

**eval() new Function() setTimeout(),setInterval()被禁止**

### 109:非置换元素：

img,input,select,textarea,button,label等这些元素有内在尺寸，即默认就有格式化外表范围的元素。

### 110.正则 /i忽略大小写 /m多行匹配

### 111.@media（ css3 ）

响应式 Web 设计

<meta name="viewport" content="width=device-width, initial-scale=1.0，maximum-scale=1.0,user-scalable=0">

initial-scale:初始化放缩，maximum-scale：最大放缩，user-scalable:是否运行用户放缩。

viewport，浏览器上用来显示页面的区域，一般移动设备的viewport比浏览器可视区域大。浏览器有一个默认layout viewport，可通过document.documentElement.clientWidth.不同设备有不同的理想viewport（理想viewport是值用户不放缩和横向滚动条，就可以查看网页全部内容）。

使用document.write('<meta name='viewport' content='width=device-width,initial-scale=1'>')可以动态修改

width=400,initial-scale=1同时设置会区较大值。

@media (min-width: 768px){ //>=768的设备 }

@media (min-width: 992px){ //>=992的设备 }

@media (min-width: 1200){ //>=1200的设备 }

注意下顺序，如果你把@media (min-width: 768px)写在了下面那么很悲剧，

@media (min-width: 1200){ //>=1200的设备 }

@media (min-width: 992px){ //>=992的设备 }

@media (min-width: 768px){ //>=768的设备 }

因为如果是1440,由于1440>768那么你的1200就会失效。

所以我们用min-width时，小的放上面大的在下面，同理如果是用max-width那么就是大的在上面，小的在下面

语法： @media sMedia { sRules }

说明：

sMedia : 　指定设备名称。请参阅附录：设备类型（ all | aural | braille | handheld | print | projection | screen | tty | tv | embossed）

sRules : 　样式表定义

eg: @media screen and (max-width:500px){body{background:green;}}/\*宽度小于500px时 绿色\*/

### 112.Object.is() es6:加强版严格等于

### 113.退出git log 按q.

git log -p查看更改文件里的具体内容。

git diff git1 git2比较两次提交详细区别

git log --stat 每次提交的修改信息，只列出了哪些文件修改了。

git show <hash> <file>详细修改.

.gitignore 写入要忽略的文件。

### 114.Object.assign(target,sources);

将可枚举对象的属性，拷贝到目标对象。

### 115.for in 的缺点：

（1）遍历的值不一定按顺序的。

（2）在原型上扩展方法会被for in 出来

（3）各个浏览器循环出来的顺序不同

（4）在跨浏览器一步不用for in而使用hasOwnProperty()来判断下.for(var i in array ){ if(array.hasOwnProperty(i)){} }

给js内置对象，扩展方法时不要用prototype，而是Object.defineProperty(Array.prototye,"fun1",{ })

### 116.web workers(工作线程):提供一种在后台进程运行的方法。

### 117,Canvas 与 SVG 的比较:

**Canvas**

依赖分辨率

不支持事件处理器

弱的文本渲染能力

能够以 .png 或 .jpg 格式保存结果图像

最适合图像密集型的游戏，其中的许多对象会被频繁重绘

**SVG**

不依赖分辨率

支持事件处理器

最适合带有大型渲染区域的应用程序（比如谷歌地图）

复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）

不适合游戏应用

### 118,zoom:

**语法：**

zoom：normal | <number> | <percentage>

默认值：normal

适用于：所有元素

继承性：有

normal：

使用对象的实际尺寸。

<number>：

用浮点数来定义缩放比例。不允许负值

<percentage>：

用百分比来定义缩放比例。不允许负值

### 119,SVG:

SVG 意为可缩放矢量图形（Scalable Vector Graphics）.

xmlns="<http://www.w3.org/2000/svg>"表明了整个svg元素默认的命名空间是svg

<rect>

css属性：

fill 属性定义矩形的填充颜色。

stroke-width 属性定义矩形边框的宽度。

stroke 属性定义矩形边框的颜色。

fill-opacity 属性定义填充颜色透明度

stroke-opacity 属性定义笔触颜色的透明度

html属性：rx 和 ry 属性可使矩形产生圆角。

圆形 - <circle>

cx和cy属性定义圆点的x和y坐标。如果省略cx和cy，圆的中心会被设置为(0, 0)。

r属性定义圆的半径。

椭圆 - <ellipse>

CX属性定义的椭圆中心的x坐标

CY属性定义的椭圆中心的y坐标

RX属性定义的水平半径

RY属性定义的垂直半径

直线 - <line>

x1 属性在 x 轴定义线条的开始

y1 属性在 y 轴定义线条的开始

x2 属性在 x 轴定义线条的结束

y2 属性在 y 轴定义线条的结束

多边形 - <polygon>

<polygon points="200,10 250,190 160,210" style="fill:lime;stroke:purple;stroke-width:1"/>

points 属性定义多边形每个角的 x 和 y 坐标

曲线 - <polyline>

<polyline points="20,20 40,25 60,40 80,120 120,140 200,180" style="fill:none;stroke:black;stroke-width:3" />

文本 - <text>

在路径上写字：

<defs>

<path id="path1" d="M75,20 a1,1 0 0,0 100,0" />

</defs>

<text x="10" y="100" style="fill:red;">

<textPath xlink:href="#path1">I love SVG I love SVG</textPath>

</text>

stroke 属性定义一条线，文本或元素轮廓颜色。

stroke-width 属性定义了一条线，文本或元素轮廓厚度

stroke-linecap 属性定义不同类型的开放路径的终结。

stroke-dasharray 属性用于创建虚线。

SVG滤镜定义在<defs>元素中。<defs>元素定义短并含有特殊元素（如滤镜）定义。

<filter>标签用来定义SVG滤镜。<filter>标签使用必需的id属性来定义向图形应用哪个滤镜？

SVG滤镜用来增加对SVG图形的特殊效果。

<feGaussianBlur> 元素是用于创建模糊效果。

线性渐变 - <linearGradient>

<linearGradient>标签必须嵌套在<defs>的内部。<defs>标签是definitions的缩写，它可对诸如渐变之类的特殊元素进行定义。

放射性渐变 - <radialGradient>

<radialGradient>标签必须嵌套在<defs>的内部。<defs>标签是definitions的缩写，它可对诸如渐变之类的特殊元素进行定义。

路径 - <path>

下面的命令可用于路径数据：

M = moveto

L = lineto

H = horizontal lineto

V = vertical lineto

C = curveto

S = smooth curveto

Q = quadratic Bézier curve

T = smooth quadratic Bézier curveto

A = elliptical Arc 椭圆弧

Z = closepath

注意：以上所有命令均允许小写字母。大写表示绝对定位，小写表示相对定位。

<svg xmlns="" version="1.1">

<path d="M150 0 L75 200 L225 200 Z" />

</svg>

### 120.objecturl = URL.createObjectURL(blob||file);

创建一个指向该参数对象的URL.

URL.revokeObjectURL(objecturl):释放一个createObjectURL创建的url对象。

a.download为下载文件名字。

window.atob(target)解密base64，window.btoa(target)加密为base64

window.btoa(encodeURIComponent(str)，window.decodeURIComponent(window.atob()))解决中文乱码问题

<img>转base64:

(1),通过canvas：

var canva = document.getElementById("myCanvas");

var ctx = canva.getContext("2d");

var img=document.getElementById("img");

var width =img.width;

var height = img.height;

canva.width = width;

canva.height= height;

ctx.drawImage(img,0,0,width,height);起始点 0，0 宽，高width,height

var dataURL = canvas.toDataURL("image/png");

canvas.toDataURL(type, encoderOptions);

type 可选

图片格式，默认为 image/png

encoderOptions 可选

在指定图片格式为 image/jpeg 或 image/webp的情况下，可以从 0 到 1 的区间内选择图片的质量。如果超出取值范围，将会使用默认值 0.92。其他参数会被忽略。

var save\_link = document.createElementNS("zwj", "a");//createElementNS() 方法与 createElement() 方法相似，只是它创建的 Element 节点除了具有指定的名称外，还具有指定的命名空间。只有使用命名空间的 XML 文档才会使用该方法。

save\_link.href = url;

save\_link.download ="zwj.doc";

click(save\_link);

var click = function(node) {

**var event = new MouseEvent("click");**

**node.dispatchEvent(event);**

};

**(2),canvas转blob:**

oid canvas.toBlob(callback, type, encoderOptions);

参数:

callback

回调函数，可获得一个单独的Blob对象参数。

type 可选

DOMString类型，指定图片格式，默认格式为image/png。

encoderOptions 可选

Number类型，值在0与1之间，当请求图片格式为image/jpeg或者image/webp时用来指定图片展示质量。如果这个参数的值不在指定类型与范围之内，则使用默认值，其余参数将被忽略。

**(3)Blob(blobParts[, options])**

返回一个新创建的 Blob 对象，其内容由参数中给定的数组串联组成。

Blob.isClosed

布尔值，指示 Blob.close() 是否在该对象上调用过。 关闭的 blob 对象不可读。

Blob.size

Blob 对象中所包含数据的大小（字节）。

Blob.type

一个字符串，表明该Blob对象所包含数据的MIME类型。如果类型未知，则该值为空字符串。

### 121.启用应用缓存

<html manifest="example.appcache">

</html>

### 122.IndexedDB

IndexedDB作为一个nosql的本地DB存储；

window.indexedDB = window.indexedDB || window.mozIndexedDB || window.webkitIndexedDB || window.msIndexedDB; // 判断支持

// 打开我们的数据库

var request = window.indexedDB.open("MyTestDatabase");

request.onerror = function(event) {

alert("Why didn't you allow my web app to use IndexedDB?!");

};//错误处理

request.onsuccess = function(event) {

db = request.result;

};//成功返回

onupgradeneeded 是我们唯一可以修改数据库结构的地方

request.onupgradeneeded = function(event) {

var db = event.target.result;

// 创建一个对象存储空间来持有有关我们客户的信息。

// 我们将使用 "ssn" 作为我们的 key path 因为它保证是唯一的。

var objectStore = db.createObjectStore("customers", { keyPath: "ssn" });

// 创建一个索引来通过 name 搜索客户。

// 可能会有重复的，因此我们不能使用 unique 索引。

objectStore.createIndex("name", "name", { unique: false });

// 创建一个索引来通过 email 搜索客户。

// 我们希望确保不会有两个客户使用相同的 email 地址，因此我们使用一个 unique 索引。

objectStore.createIndex("email", "email", { unique: true });

// 在新创建的对象存储空间中保存值

for (var i in customerData) {

objectStore.add(customerData[i]);

}

};

向数据库中增加数据,//使用事务

var transaction = db.transaction(["customers"], "readwrite");

从事务中获得相关的对象存储空间对象

var objStore = transaction.objectStore('students');

var request = objStore.add(data);

request.onsuccess = function(e) {

alert("成功插入数据，id=" + e.target.result);

};

var request = objStore.get("110"); //按照id查询数据

//更新数据

var request = objStore.get("110");\*

request.onsuccess=function(e){

var student=e.target.result;

student.name='wwww1';

objStore.put(student);\*

}

//游标遍历所有数据

var request=objStore.openCursor();//打开游标

request.onsuccess = function(e){

var cursor = e.target.result;

if(cursor){

alert(cursor.value.name);

cursor.continue();

}else {

alert('遍历完成');

}

}

### 123.css 属性选择器

[attribute] 用于选取带有指定属性的元素。

[attribute=value] 用于选取带有指定属性和值的元素。

[attribute~=value] 用于选取属性值中包含指定词汇的元素。

[attribute|=value] 用于选取带有以指定值开头的属性值的元素，该值必须是整个单词。

[attribute^=value] 匹配属性值以指定值开头的每个元素。

[attribute$=value] 匹配属性值以指定值结尾的每个元素。

[attribute\*=value] 匹配属性值中包含指定值的每个元素。

### 124.js 匹配中文/[\\u4e00-\\u9fa5]/g 注意：是双\.

### 125.字面量可以立即求值，而new object(),是方法，会在原型链中遍历找到该方法，所以字面量更高效。

**new出来的对象，可以用多份，而字面量是引用的拷贝，数据只有一份。**

### 126.argument.callee 在哪一个函数中运行，它就代表哪一个函数.可以用arguments.callee来代替匿名的函数。

### 127.同源策略：

同domain（或ip）,同端口，同协议视为同一个域，一个域内的脚本仅仅具有本域内的权限，可以理解为本域脚本只能读写本域内的资源，而无法访问其它域的资源。这种安全限制称为同源策略。

如果非同源，共有三种行为受到限制。

（1） Cookie、LocalStorage 和 IndexDB 无法读取。

（2） DOM 无法获得。

（3） AJAX 请求不能发送。

**规避上面三种限制：**

1）网页一级域名相同，只是二级域名不同，浏览器允许通过设置document.domain共享 Cookie。

2）iframe

3）window.name 无论是否同源，只要在同一个窗口里，前一个网页设置了这个属性，后一个网页可以读取它。

4）window.postMessage 这个API为window对象新增了一个window.postMessage方法，允许跨窗口通信，不论这两个窗口是否同源。

5）JSONP是服务器与客户端跨源通信的常用方法。

6）WebSocket是一种通信协议，使用ws://（非加密）和wss://（加密）作为协议前缀。该协议不实行同源政策，只要服务器支持，就可以通过它进行跨源通信。

7）CORS是跨源资源分享（Cross-Origin Resource Sharing）的缩写。它是W3C标准，是跨源AJAX请求的根本解决方法。相比JSONP只能发GET请求，CORS允许任何类型的请求。

### 128.NaN与自己不相等。

### 129.把共同的部分放到prototype上，节约内存。

原型对象从对象的角度说。 原型属性从构造函数的角度说。

原型链的尽头（root）是Object.prototype。所有对象均从Object.prototype继承属性。Object.prototype.\_proto\_ === null.

\_proto\_\_（隐式原型）与prototype（显式原型）.

每一个函数在创建之后都会拥有一个名为prototype的属性，这个属性指向函数的原型对象。prototype下包含constructor和\_\_proto\_\_.

隐式原型指向创建这个对象的函数(constructor)的prototype.

显式原型的作用：用来实现基于原型的继承与属性的共享。

隐式原型的作用：构成原型链，同样用于实现基于原型的继承。

先有Object.prototype（原型链顶端），Function.prototype继承Object.prototype而产生，最后，Function和Object和其它构造函数继承Function.prototype (Function.prototype.prototype === undefined)而产生.

console.log(Function.prototype.\_\_proto\_\_ === Object.prototype); //true

console.log(Function.\_\_proto\_\_ ); //function () {}

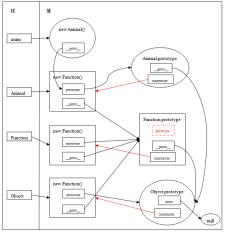
console.log(Object.\_\_proto\_\_ === Function.prototype); //true

console.log(Object.\_\_proto\_\_ === Function.\_\_proto\_\_); //true

console.log(Array.\_\_proto\_\_ === Function.\_\_proto\_\_); //true

Array.prototype.\_\_proto\_\_===Object.prototype //true

(Array.prototype.prototype === undefinded,Array.\_\_proto\_\_ === Function.prototype)



instanceof的运算规则：

左表达式的.\_\_proto\_\_ === 右表达式.prototype 如果不等 左表达式的.\_\_proto\_\_ = 左表达式的.\_\_proto\_\_.\_\_proto\_\_。

constructor 属性是专门为 function 而设计的，它存在于每一个 function 的prototype 属性中。。对象的constructor，属性用于返回创建该对象的函数，也就是我们常说的构造函数。每个具有原型的对象都会自动获得constructor属性。prototype.constructor是指向构建自己函数。(构造函数和构造器（constructor）不是一个东西)

Object.constructor === Function.constructor

Array.constructor === Object.constructor

Number.constructor === Boolean.constructor

都是函数对象，都是由function构造出来的。 constructor 的出现原本就是用来进行对象类型判断的

function Cfun(name){

this.name=name;

this.age =23;

}

Cfun.prototype.\_\_proto\_\_ === Object.prototype //true

var child = new Cfun('zwj'); //child.\_\_proto\_\_ = Cfun.prototype,所以child instanceof Cfun，也instanceof Object。

new关键字把child的隐式原型指向Cfun的prototype。

child instanceof Function //false 因为child.\_\_proto\_\_是Cfun.prototype child.\_\_proto\_\_.\_\_proto\_\_是Cfun.prototype.\_\_proto\_\_ === Object.prototype（顶端Object.prototype.\_\_proto\_\_ === null）。

原型的缺点：

无法保持独立；

信息共享

无法传参

两个属性相同改变

prototype是对象的原型，\_\_proto\_\_是对象实例化后的原型

### 130.call和apply的作用绑定函数，扩大作用域。

### 131.块级作用域：js任何一对花括号（｛和｝）中的语句集都属于一个块，在这之中定义的所有变量在代码块外都是不可见的，我们称之为块级作用域

大部分语言中块级作用域范围都是由函数块来界定的，即在函数内定义的变量在函数外无法访问。但在C语言中块级作用域是以成对的花括号来界定的，也就是说除了函数代码块外，if、for等结构也属于块级作用域。块级变量就是定义在一个块内部的变量，变量的生存周期就是这个块，出了这个块就无法访问。

### 132.Object.create(o)获取一个继承至对象o的对象。

**var 0child = Object.create( o ); //缺点：类型无关，和构造函数没关系。**

### 133.Object.create兼容所有浏览器：

1）用原生对象的方法。if(!Object.create){ Object.create = function(obj){ function F(){} F.prototype = obj; return new F(); } }

2）统一定义一个方法。function create(obj){ if( Object.create ){ return Object.create(obj) }else{ function F(){} F.prototype = obj //以传入参数为原型构造对象; return new F(); } }

### 134.逻辑中断（短路操作）

**|| 第一个为真，则返回表达式，为假，返回第二个表达式。**

**&& 第一个为假则返回，为真，返回第二个表达式。**

### 135.OFDMA：正交频分多址技术。

### 136.变量名编译后是一个内存空间地址，不占内存。引用类型的的变量会占用内存空间，大小是指针类型的大小。

### 137.Array.sort()他是根据unicode编码排序的。

**[1,12,3,10] -> [1,10,12,3]**

**他可以传递函数[1,12,3,10].sort(function(x,y){ return x-y })**

**内部实现：v8长度小与23用插入排序，大于23用快速排序。mozilla,sarfari用归并排序。**

### 138.Grisu 是把浮点数转换为字符串的算法。

### 139.稳妥模式：

在一些安全环境中，比如禁止使用this和new，这里的this是构造函数里不使用this，这里的new是在外部实例化构造函数时不适用new。没有公共属性，无法外部访问对象成员，只能通过方法访问。

function Box(name, age) {

var obj = new Object();

obj.name = name;

obj.age = age;

obj.run = function () {

return this.name + this.age + '运行中...'

};

return obj;

}

var box1 = Box('Lee', 100);

alert(box1.run());

动态原型模式： 将原型封装到构造函数里

function Box(name, age) {

this.name = name;

this.age = age;

if (typeof this.run != 'function') { //判断this.run是否存在

Box.prototype.run = function() {

return this.name + this.age + '运行中...'

};

}

}

//原型的初始化，只要第一次初始化，就可以了，没必要每次构造函数实例化的时候都初始化。

### 140.继承的类别

1）简单原型继承：

.核心

拿父类实例来充当子类原型对象

.优缺点

优点：

简单，易于实现

缺点：

修改sub1.arr后sub2.arr也变了，因为来自原型对象的引用属性是所有实例共享的。

可以这样理解：执行sub1.arr.push(2);先对sub1进行属性查找，找遍了实例属性（在本例中没有实例属性），没找到，就开始顺着原型链向上找，拿到了sub1的原型对象，一搜身，发现有arr属性。于是给arr末尾插入了2，所以sub2.arr也变了

创建子类实例时，无法向父类构造函数传参

2）借用构造函数继承（又叫：类式继承）：

function Super(val){

this.val = val;

this.arr = [1];

this.fun = function(){

// ...

}

}

function Sub(val){

Super.call(this, val); // 核心

// ...

}

.核心

等于是把父类的实例属性复制了一份给子类实例装上了（完全没有用到原型）

.优缺点

优点：

解决了子类实例共享父类引用属性的问题

创建子类实例时，可以向父类构造函数传参

缺点：

无法实现函数复用，每个子类实例都持有一个新的fun函数，太多了就会影响性能，内存爆炸。

3）混合方式：

function Super(){

// 只在此处声明基本属性和引用属性

this.val = 1;

this.arr = [1];

}

// 在此处声明函数

Super.prototype.fun = function(){};

function Sub(){

Super.call(this); // 核心

// ...

}

Sub.prototype = new Super(); // 核心

var sub1 = new Sub(1);

var sub2 = new Sub(2);

.核心

把实例函数都放在原型对象上，以实现函数复用。同时还要保留借用构造函数方式的优点，通过Super.call(this);继承父类的基本属性和引用属性并保留能传参的优点；通过Sub.prototype = new Super();继承父类函数，实现函数复用

.优缺点

优点：

不存在引用属性共享问题

可传参

函数可复用

缺点:

（一点小瑕疵）子类原型上有一份多余的父类实例属性，因为父类构造函数被调用了两次，生成了两份，而子类实例上的那一份屏蔽了子类原型上的。。。又是内存浪费，比刚才情况好点，不过确实是瑕疵

### 141.js定义接口的三种方式：( 接口优点：实现解耦，代码重用）

//注解描述的方式

/\*\*

interface f(){

function add(obj);

function sub(obj);

}

\*\*/

//属性检测法

这种方法更严谨一点。所有类都明确地声明自己实现了哪些接口，那些想与这些类打交道的对象可能针对这些声明进行检查。那些接口自身仍然只是注释，但现在你可以通过检查一个属性得知某个类自称实现了什么接口

/\*\*

Composite {

function add(child);

function remove(child);

function getChild(index);

}

interface FormItem {

function save();

}

\*\*/

var CompositeImplement = function(id, method, action) {

this.implementsInterfaces = ['Composite', 'FormItem'];

};

CompositeImplement.prototype.add = function(obj){ alert("add()"); };

CompositeImplement.prototype.remove = function(obj){};

CompositeImplement.prototype.save = function(obj){};

//检测CompositeImplement类的对象

function checkImplement( instance ){

//判断当前对象是否实现了所有接口

if(!isImplements(instance , 'Composite', 'FormItem')) {

　　　　 throw new Error("Object does not implement a required interface.");

　　 }

}

//公用的核心检测方法

function isImplements(object) {

　　for(var i = 1; i < arguments.length; i++) {

　　　　// Looping through all arguments

　　　　// 接受实现的每一个接口的名字.

　　　　var interfaceName = arguments[i];

　　　　var interfaceFound = false;

　　　　for(var j = 0; j < object.implementsInterfaces.length; j++) {

　　　　　　if(object.implementsInterfaces[j] == interfaceName) {

　　　　　　　　interfaceFound = true;

　　　　　　　　break;

　　　　　　}

　　　　}

　　　　if(!interfaceFound) {

　　　　　　return false;

　　　}

　　}

　　return true;

}

//鸭式辨型模仿接口

它把对象实现的方法集作作为判断它是不是某个类的实例的唯一标准。这种方法背后的观点很简单：如果对象具有与接口定义的方法同名的所有方法，那么就可以认为它实现了这个接口。

var Interface= function(name,methods){

//判断接口参数个数

if(arguments.length!=2){

throw newError("Interface constructor called width”+arguments.legnth+”arguments,butexpected exactly 2.");

}

this.name = name;

this.methods = []; //存放方法名字

for(var i=1,len=methods.length;i<len;i++){

if(typeof methods[i] !== "string"){

throw new Error("interface costructorexpects method names to be passed in as a string");

}

this.methods.push(methods[i]);

}

}

// 实例化Interface.

var Composite = new Interface('Composite', ['add', 'remove']);

var FormItem = new Interface('FormItem', ['save']);

//实现类

var CompositeImplement = function(){ }

CompositeImplement.prototype.add = function(obj){ alert("add()"); };

CompositeImplement.prototype.remove = function(obj){};

CompositeImplement.prototype.save = function(obj){};

//检验接口的方法

Interface.checkImplement = function (object){

//至少实现一个接口

if(arguments.length < 2) {

　　　　throw new Error("Function Interface.ensureImplements called with "

　　　　　　　　　　　　　　+arguments.length + "arguments, but expected at least 2.");

　　}

//获取接口实例对象

for(var i = 1, len = arguments.length; i < len; i++) {

　　　　var interface = arguments[i];

　　　　if(interface.constructor !== Interface) {

　　　　　　throw new Error("Function Interface.ensureImplements expects arguments"

　　　　　　　　　　　　　　+ "two and above to be instances of Interface.");

　　　　}

//循环实例对象的方法

　　　　for(var j = 0, methodsLen = interface.methods.length; j < methodsLen; j++) {

//接受每一个方法名

　　　　　　var method = interface.methods[j];

//

　　　　　　if(!object[method] || typeof object[method] !== 'function') {

　　　　　　　　throw new Error("Function Interface.ensureImplements: object "

　　　　　　　　　　　　　　　　+ "does not implement the " + interface.name + " interface. Method " + method + " was not found.");

　　　　　　}

　　　　}

　　}

}

var c = new CompositeImplement();

Interface.checkImplement(c,Composite,FormItem);

c.add();

### 142.链式编程

**·简单链式**

function dog(){

this.run = function(){

alert(123);

return this;//这是链式的关键return this

}

}

为了让对象的方法（function fun(){}）能够链式,则

Function.prototype.fun = function(name,fn){

this.prototype[name] = fn;

return this;

}

fun('event',function(){}).fun('event',function(){})

### 143.html5桌面提醒，由window.webkitNotifications提供

### 144.掺合模式（Mixin）：

mixin简单通俗的讲就是把一个对象的方法和属性拷贝到另一个对象上，注意这个继承还是有区别的。js是一种只支持单继承的语言，毕竟一个对象只有一个原型，如果想实现多继承，那就简单暴力的把需要继承的父类的所有属性都拷贝到子类上，就是使用mixin啦。也可以只从父类获取需要的一部分属性。

### 145.使用DocumentFragment（文档碎片(片段)）将一批子元素添加到任何类似node的父节点上。效率高是它真正的优势，试验表明，它比直接操作DOM快70%。（除IE8外都兼容）

var frag = document.createDocumentFragment();

for(var x = 0; x < 10; x++) {

var li = document.createElement("li");

li.innerHTML = "List item " + x;

frag.appendChild(li);

}

listNode.appendChild(frag);

### 146.js设计模式：

#### 1）单体模式（单体是在脚本加载时创建的，能将一系列有关联的变量和方法组织为一个逻辑单元，逻辑单元里面的内容通过单一的变量进行访问；）

·简单单体

var singleton = {

attr:1,

fun:function(){}

}

·闭包单体

var MyNamespace = {};

MyNamespace.Singleton= (function(){

return {

attr:1,

fun:function(){}

};

})();

·惰性单体（将构造器函数封装起来，通过一个外部接口函数延迟返回构造器）

var ext = {};

ext.base = (function(){

//利用变量控制返回的单体对象

var uniqInstance ;

//需要构造器init初始化单体对象方法

function init(){

var a =1;

var fn = function(){};

return {

attr:a,

method:function(){

return fn();

}

}

}

return {

getInstance: function(){

if(!uniqInstance){ //如果不存在，创建单体实例

uniqInstance = init();

}

return uniqInstance;

}

}

})()

·分支单体（判断程序的分支，用例：浏览器差异检测）

var def=false;

var more = (function(){

var standard = {

attr1:"1",

attr2:""

};

var standardNew = {

attr1:"1",

attr2:""

};

return (def)？ standard ： standardNew；

})();

alert(more.attr1);

#### 2)工厂模式( 消除两个类的依赖。使用一个方法来决定究竟要实例化哪个具体的类)

·工厂环境：

function carShop(){}

carShop.prototype = {

constructor:carShop,

sellCar:function( type ){

var car;

switch(type){

case 'banz':car = new benz();break;

case 'baoma':car = new baoma();break;

default:'not buy it';

}

//检验接口实现（必须对对象检测以确保实现了必要的方法）

Interface.checkImplement(car ,carInterface );

return car;

}

}

//接口对象的实例

var carInterface = new Interface("carInterface",['start','run']);

function benz(){}

benz.prototype.start = function(){}

benz.prototype.run = function(){}

function baoma(){}

baoma.prototype.start = function(){}

baoma.prototype.run = function(){}

var shop = new carShop();

var car = shop.sellCar('benz');

car.start();

car.run();

·简单工厂

//卖车商店

function carShop(){}

carShop.prototype = {

constructor:carShop,

sellCar:function( type ){

var car = carFactory.createCar( type );

return car;

}

}

//生产车的工厂

var carFactory = {

createCar:function(type){

var car;

switch(type){

case 'banz':car = new benz();break;

case 'baoma':car = new baoma();break;

default:'not buy it';

}

//检验接口实现（必须对对象检测以确保实现了必要的方法）

Interface.checkImplement(car ,carInterface );

return car;

}

}

·复杂工厂+动态工厂

BH.extend = function(subClass,superClass){

var F= function(){};

F.prototype = superClass.prototype;

subClass.prototype = new F();

subClass.prototype.constructor = subClass;

}

//super shop

function carShop(){};

carShop.prototype = {

constructor:carShop,

sellCar:function(){

this.abstractSellCar(type)

},

abstractSellCar:function(){

throw new Error('this method is abstract');

}

}

//benz shop

function benzShop(){};

BH.extend( benzShop,carShop );

benzShop.prototype = {

constructor:benzShop, //还原构造器（ 即使不修正也不会有什么影响，主要是防止显示调用构造函数出错 ）

sellCar:function(type){ //重新sellCar

var car;

var types = ['benz']; //所有benz的车，这里只例举一种做测试

types.map(function(v){

if(v === type){

car = carFactory.createCar( type );

}else{ alert('no car'); }

})

return car;

}

}

//baoma shop

function baomaShop(){};

BH.extend( baomaShop,carShop );

baomaShop.prototype = {

constructor:baomaShop,

sellCar:function(type){

var car;

var types = ['benz'];

types.map(function(v){

if(v === type){

car = carFactory.createCar( type );

}else{ alert('no car'); }

})

return car;

}

}

var shop1 = new benzShop();

var car1 = shop1.sellCar("benz");

car1.run();

完整代码：

var BH = {};

BH.extend = function(subClass,superClass){

var F= function(){};

F.prototype = superClass.prototype;

subClass.prototype = new F();

subClass.prototype.constructor = subClass;

}

var Interface= function(name,methods){

//判断接口参数个数

if(arguments.length!=2){

throw newError("Interface constructor called width”+arguments.legnth+”arguments,butexpected exactly 2.");

}

this.name = name;

this.methods = []; //存放方法名字

for(var i=0,len=methods.length;i<len;i++){

if(typeof methods[i] !== "string"){

throw new Error("interface costructorexpects method names to be passed in as a string");

}

this.methods.push(methods[i]);

}

}

//检验接口的方法

Interface.checkImplement = function (object){

//至少实现一个接口

if(arguments.length < 2) {

throw new Error("Function Interface.ensureImplements called with "

+arguments.length + "arguments, but expected at least 2.");

}

//获取接口实例对象

for(var i = 1, len = arguments.length; i < len; i++) {

var interface = arguments[i];

if(interface.constructor !== Interface) {

throw new Error("Function Interface.ensureImplements expects arguments"

+ "two and above to be instances of Interface.");

}

//循环实例对象的方法

for(var j = 0, methodsLen = interface.methods.length; j < methodsLen; j++) {

//接受每一个方法名

var method = interface.methods[j];

//

if(!object[method] || typeof object[method] !== 'function') {

throw new Error("Function Interface.ensureImplements: object "

+ "does not implement the " + interface.name + " interface. Method " + method + " was not found.");

}

}

}

}

function benz(){}

benz.prototype.start = function(){ alert('benz star'); }

benz.prototype.run = function(){ alert('benz run'); }

function baoma(){}

baoma.prototype.start = function(){}

baoma.prototype.run = function(){}

//生产车的工厂

var carFactory = {

createCar:function(type){

var car;

/\*switch(type){

case 'banz':car = new benz();break;

case 'baoma':car = new baoma();break;

default:'not buy it';

}\*/

var car = eval('new ' + type + '()');

//检验接口实现（必须对对象检测以确保实现了必要的方法）

Interface.checkImplement(car ,carInterface );

return car;

}

}

//接口对象的实例

var carInterface = new Interface("carInterface",['start','run']);

//super shop

function carShop(){}

carShop.prototype = {

constructor:carShop,

sellCar:function(){

this.abstractSellCar(type)

},

abstractSellCar:function(){

throw new Error('this method is abstract');

}

};

//benz shop

function benzShop(){}

BH.extend( benzShop,carShop );

benzShop.prototype = {

constructor:benzShop,

sellCar:function(type){

var car;

var types = ['benz']; //所有benz的车，这里只例举一种做测试

types.map(function(v){

if(v === type){

car = carFactory.createCar( type );

}else{ alert('no car'); }

});

return car;

}

};

//baoma shop

function baomaShop(){}

BH.extend( baomaShop,carShop );

baomaShop.prototype = {

constructor:baomaShop,

sellCar:function(type){

var car;

var types = ['benz'];

types.map(function(v){

if(v === type){

car = carFactory.createCar( type );

}else{ alert('no car'); }

});

return car;

}

}

var shop1 = new benzShop();

var car1 = shop1.sellCar("benz");

car1.run();

#### 3）桥接模式（既可以把两个对象连接，又能避免强耦合。主要是把抽象与现实分离，使他们独立化）

·事件监听

var event = {

addHandle:function(element,type,handler){

if(element.addEventListener){

element.addEventListener(type,handler,false);

}else if(element.attachEvent){

element.attachEvent('on'+type,handler)

}

},

removeHandler:function( element,type,handler ){

if(element.removeEventListener){

element.removeEventListener(type,handler,false);

}else if(element.detachEvent){

element.detachEvent('on'+type,handler)

}

}

}

//前台逻辑

var element = document.getElementByIdL('id');

event.addHandle(element,'click',bridgeHandle);

//后台逻辑

function sendReq(msg){

//$.post('url',{msg:this.value},function(result){});

alert('发送数据了！'+msg)

}

//桥接模式分离前后

function bridgeHandle(){

var msg = this.value;

senReq(msg);

}

·特权函数

var publicClass = function(){

var name = 'zwj'; //私有变量

this.getName = function(){ return name; } //可以访问私有变量的特权函数。

}

·用桥接模式联结多个类(使每个单元独立，有自己的变化)

var class1 = function(a,b){

}

var class2 = function(c){

}

var bridegeClass = function(a,b,c){

this.one = new class1(a,b);

this.two = new class2(c);

}

#### 4）组合模式（转为web动态页面量身定做的模式）

（1）你可以同样的方法处理对象的集合与其中的特定子对象。

（2）把一批子对象组织成树结构

--只有组合对象和叶子对象

var org = function(name){

this.name = name;

this.depts = [];

}

org.prototype = {

constructor:org,

addDepts:function(child){

this.depts.push(child);

return this;

},

getDepts:function(){

return this.depts;

}

}

var dept = function(name){

this.name = name;

this.persons = [];

}

dept.prototype = {

constructor:dept,

addPersons:function(child){

this.persons.push(child);

return this;

},

getPersons:function(){

return this.persons;

}

}

var person =function( name ){

this.name =name;

}

person.prototype = {

constructor:person,

sleep:function(){

document.write(this.name + '--睡觉--');

}

}

var p1 = new person('z1');

var p2 = new person('z2');

var p3 = new person('z3');

var dept1 = new dept('开发部门');

dept1.addPersons(p1).addPersons(p2);

var dept2 = new dept('销售部门');

dept2.addPersons(p3);

var org = new org("bjs");

org.addDepts(dept1).addDepts(dept2);

for(var i=0,depts = org.getDepts();i<depts.length;i++){

for(var j=0,persons =depts[i].getPersons();j<persons.length;j++){

if(persons[j].name === 'z1'){

persons[j].sleep();

}

}

}

//使用组合模式改写上面代码

var compositeInterface = new Interface('compositeInterface',['addChild','getChild']);

var leadInterface = new Interface('leafInterface',['working','sleep']);

var composite =function( name ){

this.name = name;

this.type = 'composite';

this.children = [];

};

composite.prototype = {

constructor:composite,

addChild:function(child){ this.children.push(child); return this; },

getChild:function(){

//接受叶子类型对象的数组

var elements = [];

//根据name让指定下所有类型为leaf的对象去执行

if(name && this.name !==name ){

}else{

this.children.forEach(function(v){

if(v.type === 'composite'){

v.children.forEach(arguments.callee);

}else if( v.type === 'leaf' ){ elements.push(v); }

})

}

return elements;

},

working:function(){

// 得到所有leaf类型的数组

var leafObject = this.getChild(name);

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

leafObject[i].working();

}

},

sleep:function(){

// 得到所有leaf类型的数组

var leafObject = this.getChild(name);

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

leafObject[i].sleep();

}

}

}

var leaf = function(name){

this.name = name;

this.type = 'leaf';

}

leaf.prototype = {

constructor:leaf,

addChild:function(){ throw new Error('this is disabled') },

getChild:function(){ if(this.name === name){ return this } return null; },

working:function(){ document.write(this.name + '--工作--'); },

sleep:function(){ document.write(this.name + '--睡觉--'); }

}

var p1 = new leaf('z1');

var p2 = new leaf('z2');

var p3 = new leaf('z3');

var dept1 = new composite('开发部门');

dept1.addChild(p1).addChild(p2);

var dept2 = new composite('销售部门');

dept2.addChild(p3);

var org = new composite('总部');

org.addChild(dept1).addChild(dept2);

org.sleep('开发部门');

#### 5）门面模式（简化类的接口，消除类与使用他的克服代码之间的耦合）

·简化API

function a(x){}

function b(y){}

function ab(x,y){ a(x); b(y);}

通俗解释：首先病人必须先挂号，然后门诊。如果医生要求化验，病人必须首先划价，然后缴款，才能到化验部门做化验。化验后，再回到门诊室。

解决这种不便的方法便是引进门面模式。可以设置一个接待员的位置，由接待员负责代为挂号、划价、缴费、取药等。这个接待员就是门面模式的体现，病人只接触接待员，由接待员负责与医院的各个部门打交道。

#### 6）适配器模式（适配器模式可用来在现有接口和不兼容的类之间进行适配。）

适配器则要把一个接口转换为另一个接口，它并不会滤除某些能力，也不会简化接口。如果客户系统期待的API不可用，那就需要用到适配器。

var clientObject ={

string1:'foo',

string2:'bar',

string3:'baz'

};

function adpter(o){

interfaceMethod(o.string1, o.string2, o.string3)

}

function interfaceMethod(str1,str2,str3){

}

clientToInterfaceAdapter函数的作用就在于对interfaceMethod函数进行包装，并把传递给它的参数转换为后者需要的形式。

#### 7）装饰者模式（为对象添加新特性，并不使用创建新子类的手段.从派生的子类添加方法）

BH.extend = function(subClass,superClass){

var F= function(){};

F.prototype = superClass.prototype;

subClass.prototype = new F();

subClass.prototype.constructor = subClass;

subClass.superClass = superClass.prototype;//定义子类的静态属性，保存父类原型

}

var carInterface = new Interface('car', ['getPrice','assemble']);

var Car = function(car){

this.car = car;

Interface.checkImplement(this,carInterface)

};

Car.prototype = {

constructor:Car,

getPrice:function(){ return 200; },

assemble:function(){ document.write("组装汽车--")}

}

var lighterDecorator = function(car){

Car.call(this,car);

};

BH.extend(lighterDecorator,Car);

lighterDecorator.prototype = {

constructor:lighterDecorator,

getPrice:function(){ return this.car.getPrice() + 1000; },

assemble:function(){ document.write('组装车灯---');}

}

var car = new Car();

alert(car.getPrice());

car.assemble();

car = new lighterDecorator(car);

alert(car.getPrice());

car.assemble();

函数装饰者:

创建了一个包装另外一个函数的装饰者，她的作用在于将被包装者的返回结果做处理

#### 8）享元模式（它最适合于解决因创建大量类似对象而累及性能的问题。是一种优化的设计模式）

优化：时间（代码运行时间），空间(内存)

享元模式用于减少应用程序所需要的对象的数量。这是通过将对象的内部状态划分为内存数据和外在数据两类而实现的。内在数据是指类的内部方法所需要的信息，没有这种数据的话类就不能正常运转。外在数据则是可以从类身上剥离并存储在其外部的信息。我们可以将内存状态相同的所有对象替换为同一个共享对象，用这种方法可以把对象数据减少到不同内存状态的数量。

var Car =function(make,model,year,owner,tag,renewDate){

//汽车的信息

this.make = make;

this.model = model;

this.year = year;

this.owner = owner;

this.tag = tag;

this.renewDate = renewDate;

}

Car.prototype = {

constructor:Car,

getMake ：function(){

return this.make;

},

getModel ： function(){

return this.model;

},

getYear ： function(){

return this.year;

},

transferOwership:function(newOwner,newTag,newRenewDate){

this.owner = newOwner;

this.tag = newTag;

this.renewDate = newRenewDate;

},

renewRegistration:function(newRenewDate){

this.renewDate = newRenewDate;

}

}

//享元模式改写

var Car =function(make,model,year,owner,tag,renewDate){

//汽车的信息

this.make = make;

this.model = model;

this.year = year;

}

Car.prototype = {

constructor:Car,

getMake:function(){

return this.make;

},

getModel : function(){

return this.model;

},

getYear : function(){

return this.year;

}

};

//使用工厂模式,把公共的部分提取出来

var CarFactory = (function(){

var createdCars = {};

return {

createCar:function(make,model,year) {

//检测Car是否创建.

if (createdCars[make + model + year]) {

return createdCars[make + model + year];

} else {

var car = new Car(make, model, year);

createdCars[make + model + year] = car;

return car;

}

}

}

})();

//使用单体将外在和内在数据结合到一起

var CarRecordManager = (function(){

var carRecordDatabase = {};

return {

//Add a new car record into thecity’s system.

addCarRecord:function(make,model,year,owner,tag,renewDate){

var car =CarFactory.createCar(make,model,year);

carRecordDatabase[tag] = {

owner:owner,

renewDate:renewDate,

car:car

}

}

}

})();

var arr = [];

var stime = new Date().getTime();

for(var i=0;i<50000;i++){

arr.push(CarRecordManager.addCarRecord('上海大众','迈腾','2012-02-03','bhx','bj0011','2013-04-01'));

}

var etime = new Date().getTime();

alert(etime -setime);

优点：降低资源负荷

缺点：妨碍调试。 使维护变困难。

#### 9）代理模式（代理是一个对象，它可以用来控制对另一个对象的访问。它还可以把本体的实例化推迟到真正需要的时候，对于实例化比较费时的本体，或者因尺寸较大以至于不用时不宜保存在内存中的本体，这特别有用）

var LibraryInterface = new Interface('library',['addbook','findbook']);

var Book =function(id,title,author){

this.id = id;

this.title = title;

this.author = author;

}

var library = function(books){

this.books = books;

}

library.prototype = {

constuctor:library,

addbook:function(book){

this.books[book.id] =book;

},

findbook:function(id){

if(this.books[id]){

return this.books[id];

}

}

}

//图书馆代理

var libraryProxy = function(books){

this.books = books;

this.library = null;

}

libraryProxy.prototype = {

initializaLibrary:function(){

if(this.library === null){

alert('真正本体');

this.library = new library(this.books);

}

},

addbook:function(book){

this.initializaLibrary();

this.library.addbook(book);

},

findbook:function(id){

this.initializaLibrary();

return this.library.findbook(id);

}

}

var proxy = new libraryProxy({

'01':new Book('01','java','zwj'),

'02':new Book('02','js','zwj2')

});

alert(proxy.findbook('02').author);

优点：

缺点：掩盖了大量复杂行为

#### 10）观察者模式（发布-订阅者模式）

对一个对象进行实时观察，当对象状态发生改变时进行通知。

//发布者

function Publish(name){

this.name = name;

this.subscribers = [];

}

//发布消息的方法

Publish.prototype.deliver= function(data){

var publish = this;

this.suscribers.forEach(function(fn){

fn(data,publish);

});

return this; //chain call链式编程

};

//以下两个方法是给观察者的

//订阅的方法

Function.prototype.subscribe = function(publisher){

var sub = this;

// some方法循环遍历数组并执行一个函数，如果有一个true，整体返回true

var alreadyExists = publisher.subscribers.some(function(item){return item === sub;}); //没有订阅过，添加到订阅的人的名单中

if(!alreadyExists){

publisher.subscribers.push(this);

}

return this;

};

//退订方法

//unsubscribe方法可供订阅者用来停止对指定发布者的观察：

Function.prototype.unsubscribe= function(publisher){

var that = this;

//filter循环遍历数组，不满足条件，删除元素

publisher.subscribers =publisher.subscribers.filter(function(item){return item!==that});

return this;

};

//实例化，发布者zwj，zhq

var publishZwj= new Publish('zwj');

var publishZhq= new Publish('zhq');

//2个观察者

var observer1= function(news){ alert("收到："+news) };

var observer2= function(news){ alert("收到："+news) };

observer1.subscribe(publishZwj).subscribe(publishZhq);

observer2.subscribe(publishZwj);

publishZwj.deliver('zwj发布消息');

优点：开发基于行为的大型应用程序的有力手段

缺点：创建可观察者的对象带来开销。（可以使用惰性加载解决）

#### 11）命令模式

客户创建命令；调用者执行该命令；接收者在命令执行时执行相应操作。用来消除调用者和接收者之间的耦合.

调用者只需要执行相应命令，不需要知道处理命令的内部实现

·简单命令模式

var commandInterface = new Interface('commandInterface',['execute']);

var command = function(obj){

this.ad = obj;

};

command.prototype.execute = function(){

this.ad.start();

}

var commander = new command(obj);

commander.execute();

·闭包命令模式

function makeStart(obj){

return function(){

obj.start();

}

}

var start = new makeStart(obj);

start();

3个参与者：客户，调用者，接收者

//有一个命令接口，一个执行命令，一个回退命令。

var commandInterface = new Interface('commandInterface',['execute','undo']);

var Moveup = function(cusor){

this.cusor = cusor;

}

Moveup.prototype = {

constructor:Moveup,

execute:function(){

this.cusor.move(0,-10);

},

undo:function(){

this.cusor.move(0,10);

}

}

//接收者：

var Cusor = function(){}

Cusor.prototype = {

constuctor:Cusor,

move:function(x,y){}

}

//客户：创建命令

var cusor = new Cusor();

var moveup = new Moveup(cusor);

#### 12）责任链模式

js内部使用这种模式处理事件捕获和冒泡事件。解决发出者与接受者的耦合

责任链模式流程：

-发送者知道链中的第一个接受者，他向这个接受者发出请求

-每一个接受者都对请求进行分析，要么处理，要么往下传递

-每一个接受者知道的其他对象只有一个，即他的下家

-如果没有任何接受者处理请求，那么请求将从链中离开，不同实现对此有不同反应。

var Assign = function(task){

this.task = task;

}

//接受任务的类

var Work = function(assign){

this.assign = assign;

}

Work.prototype ={

constructor:Work,

filterHandler:function(executor){

console.log(executor);

if(this.assign.task === executor.cando){

return executor.todo();

}else{

//传递给下一个对象

if(executor.successor){

arguments.callee.call(this,executor.successor);

}else{

document.write('没人会');

}

}

}

}

//处理者

var Executor =function(name,cando){

this.name =name;

this.cando = cando;

this.successor = null;

}

Executor.prototype ={

constructor:Executor,

todo:function(executor){

document.write(this.name+'开发'+this.cando);

},

//设置责任链的配置函数

setSuccessor:function(successor){

this.successor = successor;

}

}

//实例化处理对象

var e1=new Executor('a1','js');

var e2=new Executor('a2','java');

var e3=new Executor('a3','css');

//设置对象之间的责任链关系

e1.setSuccessor(e2);

e2.setSuccessor(e3);

//实例化任务对象

var assign = new Assign('ja');

//处理任务类的实例

var w = new Work(assign);

w.filterHandler(e1);

### 147）对象是人们要进行研究的任何事物，从最简单的整数到复杂的飞机等均可看作对象，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

**类是对某一类事物的抽象的，概念的定义。**

**对象是实际存在的该类事物的每个个体，也称实例。**

**面向过程：为了把大象装进冰箱，需要3个过程。**

**1) 把冰箱门打开（得到打开门的冰箱）**

**2) 把大象装进去（打开门后，得到里面装着大象的冰箱）**

**3) 把冰箱门关上（打开门、装好大象后，获得关好门的冰箱）每个过程有一个阶段性的目标，依次完成这些过程，就能把大象装进冰箱。**

**面向对象：**

**为了把大象装进冰箱，需要做三个动作（或者叫行为）。**

**每个动作有一个执行者，它就是对象。**

**1) 冰箱，你给我把门打开**

**2) 冰箱，你给我把大象装进去（或者说，大象，你给我钻到冰箱里去）**

**3) 冰箱，你给我把门关上**

**依次做这些动作，就能把大象装进冰箱。**

**面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤1步1步实现，使用之际1个1个依次调用即可了。**

**面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成1个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。**

### 148）1.#!/usr/bin/python

指定用来执行脚本的解释器，必须在第一行

2.+执行一次就会去申请一块新内存，join会首先计算申请的总空间，然后一次性申请所需内存，然后将字符串每一个元素复制到内存中去。

### 149）http协议（RFC2616）

（网际互联卷1）

Http特点：

应用层：应用层上，它基于tcp,它本身不提供可靠性和重传。

请求/响应：一旦建立会话，一端必须向令一端发送http请求，并由另一端响应。

无状态：服务器不保留以前的请求或会话历史。

双向传输：浏览器请求页面，服务器返回副本。浏览器也可以通过表单向服务器发送数据。

能力协商：浏览器和服务器可以协商一些细节。如字符集。

支持缓存：http允许浏览器询问服务器，确定自从缓存该页之后内容是否改变。

支持中介：运行机器作为代理服务器。

http请求信息组成：

**通用头(General header)：**

Request URL:

Request Method:

Status Code:

Remote Address:

**请求头(request-header)：**

1. Accept:text/plain, \*/\*; q=0.01
2. Accept-Encoding:gzip, deflate
3. Accept-Language:zh-CN,zh;q=0.8
4. Connection:keep-alive
5. Content-Type:application/json; charset=utf-8
6. Cookie:xxx
7. Host:www.cnblogs.com
8. Referer:http://www.cnblogs.com/CareySon/archive/2012/04/27/HTTP-Protocol.html
9. User-Agent:Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.104 Safari/537.36
10. X-Requested-With:XMLHttpRequest

**响应头(response-header)：**

1. Cache-Control:private
2. Accept-Ranges**:**bytes 表明服务器是否支持指定范围请求及哪种类型的分段请求
3. Connection:keep-alive
4. Content-Encoding:gzip
5. Content-Type:text/html; charset=utf-8
6. Date:Tue, 29 Aug 2017 06:10:44 GMT
7. Transfer-Encoding传输编码:chunked 分块传输（输出的内容长度不能确定）
8. Vary:Accept-Encoding
9. X-Frame-Options:SAMEORIGIN
10. X-UA-Compatible:IE=10
11. ETag**:**W/"137-1417599402000" 它根据实体内容生成的一段hash字符串

长度编码首部：（借用了邮箱的基本格式2822和MIME扩展）

Conten-type:

获知请求中的消息主体是用何种方式编码，再对主体进行解析。

Post表单提交数据 application/x-www-form-urlencoded。

使用表单上传文件时为multipart/form-data

提交 JSON 字符串application/json 方便的提交复杂的结构化数据，特别适合 RESTful 的接口。{'title':'test', 'sub' : [1,2,3]};以{"title":"test","sub":[1,2,3]}形式传输。

**text/xml** 它是一种使用 HTTP 作为传输协议，XML 作为编码方式的远程调用规范。

Content-length:

Content-Encoding:

Content-language:

协商：

服务驱动和代理驱动.

Accept-Encoding:编码

Accept-Charset:字符集

Accept-Language:语言

**HTTP1.1：**

引入了http持久连接。持久连接提供重用TCP连接。Keep-Alive功能使客户端到服务器端的连接持续有效。

头：Connection:keep-alive;Keep-Alive:Timout=20(秒).

**HTTP管道：**

持久HTTP让我们重用已有的连接来完成多次应用请求，但多次请求必须严格按照FIFO队列顺序。HTTP管道做了优化，HTTP请求放到一个TCP连接中一一发送，而在发送过程中不需要等待服务器对前一个请求的响应；只不过，客户端还是要按照发送请求的顺序来接收响应。Http管道可以让我们把FIFO队列从客户端（请求队列）迁移到服务端（响应队列）。

Http管道缺点：1.一个慢响应阻塞后续请求。2.并行处理请求服务器必须缓冲管道响应，占用服务器资源。3.响应失败可能终止tcp连接。4.可能有中间代理要检测管道兼容性，如果不支持它可能中断连接。（大多数浏览器还是禁用Http管道计算的）

浏览器允许我们并行打开多个TCP连接，大多数浏览器支持每个主机打开6个。每个http连接就需要一个tcp连接。

**域名分区：**

**我们不必通过一个主机提供所以资源，可以手工分散到多个子域名。**

**资源连接和合并**

**把多个js或css合并为一个，把多张图合并为精灵图。这些都是以内容为中心的优化。缺点：带来额外的处理，部署，编码也给缓存代理负担，影响页面执行速度。30-50kb是每个js文件大小的合适范围。**

**补充：**

|  |  |
| --- | --- |
| 浏览器 | 最大长度（字符数） |
| Internet Explorer | 2083 |
| Firefox | 65,536 |
| chrome | 8182 |
| Safari | 80,000 |
| Opera | 190,000 |
| curl（linux下指令） | 8167 |

**详细：**

**Requests部分**

|  |  |  |
| --- | --- | --- |
| Header | 解释 | 示例 |
| Accept | 指定客户端能够接收的内容类型 | Accept: text/plain, text/html |
| Accept-Charset | 浏览器可以接受的字符编码集。 | Accept-Charset: iso-8859-5 |
| Accept-Encoding | 指定浏览器可以支持的web服务器返回内容压缩编码类型。 | Accept-Encoding: compress, gzip |
| Accept-Language | 浏览器可接受的语言 | Accept-Language: en,zh |
| Accept-Ranges | 可以请求网页实体的一个或者多个子范围字段 | Accept-Ranges: bytes |
| Authorization | HTTP授权的授权证书 | Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| Cache-Control | 指定请求和响应遵循的缓存机制 | Cache-Control: no-cache |
| Connection | 表示是否需要持久连接。（HTTP 1.1默认进行持久连接） | Connection: close |
| Cookie | HTTP请求发送时，会把保存在该请求域名下的所有cookie值一起发送给web服务器。 | Cookie: $Version=1; Skin=new; |
| Content-Length | 请求的内容长度 | Content-Length: 348 |
| Content-Type | 请求的与实体对应的MIME信息 | Content-Type: application/x-www-form-urlencoded |
| Date | 请求发送的日期和时间 | Date: Tue, 15 Nov 2010 08:12:31 GMT |
| Expect | 请求的特定的服务器行为 | Expect: 100-continue |
| From | 发出请求的用户的Email | From: user@email.com |
| Host | 指定请求的服务器的域名和端口号 | Host: www.zcmhi.com |
| If-Match | 只有请求内容与实体相匹配才有效 | If-Match: “737060cd8c284d8af7ad3082f209582d” |
| If-Modified-Since | 如果请求的部分在指定时间之后被修改则请求成功，未被修改则返回304代码 | If-Modified-Since: Sat, 29 Oct 2010 19:43:31 GMT |
| If-None-Match | 如果内容未改变返回304代码，参数为服务器先前发送的Etag，与服务器回应的Etag比较判断是否改变 | If-None-Match: “737060cd8c284d8af7ad3082f209582d” |
| If-Range | 如果实体未改变，服务器发送客户端丢失的部分，否则发送整个实体。参数也为Etag | If-Range: “737060cd8c284d8af7ad3082f209582d” |
| If-Unmodified-Since | 只在实体在指定时间之后未被修改才请求成功 | If-Unmodified-Since: Sat, 29 Oct 2010 19:43:31 GMT |
| Max-Forwards | 限制信息通过代理和网关传送的时间 | Max-Forwards: 10 |
| Pragma | 用来包含实现特定的指令 | Pragma: no-cache |
| Proxy-Authorization | 连接到代理的授权证书 | Proxy-Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| Range | 只请求实体的一部分，指定范围 | Range: bytes=500-999 |
| Referer | 先前网页的地址，当前请求网页紧随其后,即来路 | Referer: http://www.zcmhi.com/archives/71.html |
| TE | 客户端愿意接受的传输编码，并通知服务器接受接受尾加头信息 | TE: trailers,deflate;q=0.5 |
| Upgrade | 向服务器指定某种传输协议以便服务器进行转换（如果支持） | Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11 |
| User-Agent | User-Agent的内容包含发出请求的用户信息 | User-Agent: Mozilla/5.0 (Linux; X11) |
| Via | 通知中间网关或代理服务器地址，通信协议 | Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1) |
| Warning | 关于消息实体的警告信息 | Warn: 199 Miscellaneous warning |

**Responses 部分**

|  |  |  |
| --- | --- | --- |
| Header | 解释 | 示例 |
| Accept-Ranges | 表明服务器是否支持指定范围请求及哪种类型的分段请求 | Accept-Ranges: bytes |
| Age | 从原始服务器到代理缓存形成的估算时间（以秒计，非负） | Age: 12 |
| Allow | 对某网络资源的有效的请求行为，不允许则返回405 | Allow: GET, HEAD |
| Cache-Control | 告诉所有的缓存机制是否可以缓存及哪种类型 | Cache-Control: no-cache |
| Content-Encoding | web服务器支持的返回内容压缩编码类型。 | Content-Encoding: gzip |
| Content-Language | 响应体的语言 | Content-Language: en,zh |
| Content-Length | 响应体的长度 | Content-Length: 348 |
| Content-Location | 请求资源可替代的备用的另一地址 | Content-Location: /index.htm |
| Content-MD5 | 返回资源的MD5校验值 | Content-MD5: Q2hlY2sgSW50ZWdyaXR5IQ== |
| Content-Range | 在整个返回体中本部分的字节位置 | Content-Range: bytes 21010-47021/47022 |
| Content-Type | 返回内容的MIME类型 | Content-Type: text/html; charset=utf-8 |
| Date | 原始服务器消息发出的时间 | Date: Tue, 15 Nov 2010 08:12:31 GMT |
| ETag | 请求变量的实体标签的当前值 | ETag: “737060cd8c284d8af7ad3082f209582d” |
| Expires | 响应过期的日期和时间 | Expires: Thu, 01 Dec 2010 16:00:00 GMT |
| Last-Modified | 请求资源的最后修改时间 | Last-Modified: Tue, 15 Nov 2010 12:45:26 GMT |
| Location | 用来重定向接收方到非请求URL的位置来完成请求或标识新的资源 | Location: http://www.zcmhi.com/archives/94.html |
| Pragma | 包括实现特定的指令，它可应用到响应链上的任何接收方 | Pragma: no-cache |
| Proxy-Authenticate | 它指出认证方案和可应用到代理的该URL上的参数 | Proxy-Authenticate: Basic |
| refresh | 应用于重定向或一个新的资源被创造，在5秒之后重定向（由网景提出，被大部分浏览器支持） | Refresh: 5; url=  http://www.zcmhi.com/archives/94.html |
| Retry-After | 如果实体暂时不可取，通知客户端在指定时间之后再次尝试 | Retry-After: 120 |
| Server | web服务器软件名称 | Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) |
| Set-Cookie | 设置Http Cookie | Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1 |
| Trailer | 指出头域在分块传输编码的尾部存在 | Trailer: Max-Forwards |
| Transfer-Encoding | 文件传输编码 | Transfer-Encoding:chunked |
| Vary | 告诉下游代理是使用缓存响应还是从原始服务器请求 | Vary: \* |
| Via | 告知代理客户端响应是通过哪里发送的 | Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1) |
| Warning | 警告实体可能存在的问题 | Warning: 199 Miscellaneous warning |
| WWW-Authenticate | 表明客户端请求实体应该使用的授权方案 | WWW-Authenticate: Basic |

**cookie firefox现在50个 ，大小4095或4096字节 （js设置cookie ,document.cookie = "zwj",typeof document.cookies // "string"）**

**localStorage:每个域为5M**

**http2:新特性**

**·二进制协议**

**·头部压缩（提取相同的部分）**

**·多路复用（每一个单独连接可以包含多个并发流）**

**·优先级和依赖性**

**·服务器推送**

**·重置（发生RST\_STREAM来实现）**

**·流量控制（每个流都有自己的流量窗口）**

**https:(默认端口443。RFC2246 TLS1.0，RFC5246 TLS2.0)**

**https = http + ssl或http+tls**

### 150）clickjacking:点击劫持

**将目标网站用iframe嵌入，opacity =0;这网页是透明的，但可以点击。**

**防止自己的网站被嵌入：**

**if(top !== self){**

**top.location = self.location;**

**}**

### 151）navigator.mediaDevices.getUserMedia({video:true},function(e){ console.log(e); }) 调用摄像头

### 152）对于f loat:left;当一行放不下时会分成两行。

### 153）把.rst转换为html

**pip install sphinx**

**sphinx-build -b html 源 目标 （目录下要有conf.py）**

**或 make html(目录下要有Makefile文件)**

**No handlers could be found for logger "sklearn.datasets.twenty\_newsgroups"**

**python会下载数据，时间可能有点久**

### 154）ajax工作原理：

**相当于在用户和服务器之间加了一层中间层（ajax引擎），使用户操作与服务器响应异步化。核心：js,xmlhttprequest,dom.使用xmlhttprequest向服务器发送异步请求获取数据。使用js操作dom更新页面。**

**优点：1.无需刷新页面（得益于xmlhttp组件对象xmlhttprequest）**

**2.异步与服务器通信（不需要打断用户操作）**

**3.前后端负载平衡（可以把服务器的工作转到客户端）**

**4.基于标准被广泛支持**

**5.数据与呈现分离**

**缺点：**

**1.没有回退和历史记录功能，浏览器只能记忆静态页面的内容。（可以使用iframe重现）**

**2.安全问题**

**3.对搜索引擎支持教弱。（使用不当会增加网络流量，降低整个系统的性能）**

**4.破坏程序的异常处理机制**

**5.太多客户端代码造成开发成本。**

### 155）js活动对象在执行函数时创建，包含函数的形参和实参以及函数内部定义的变量和函数。

### 156)background-position:定位图像

**1、两个值前面一个是横向的定位，我们称为x轴方向定位。后面一个值是纵向的定位，我们称为y轴方向定位。如果只有一个值，那默认的就是x轴方向，这时y轴方向就默认的是上下居中对齐，也就是center。**

**2、坐标轴的原点就是对应容器的左顶点。**

**3、这个坐标的y轴箭头朝下，也就是右下方（容器内部）x y的值才都为正。**

**4、x y值分别表示背景图片的左顶点相对于坐标原点（也就是容器的左顶点）的值。**

**5、x y的值可以用百分比或者px来表示。**

**6、x y也可以用“left、right、top、bottom、center”这五个对齐方式来表示，但注意：用“left、right、top、bottom、center”来表示的时候，应用的是对齐规则，而不是坐标规则。x为left是表示图片的左边和容器的左边对齐，为right的时候表示图片的右边和容器的右边对其，y为top的时候表示图片的顶部和容器的顶部对齐，为bottom时表示图片的底部和容器的底部对齐，x y等于center的时候表示居中对齐。**

**7、x y用百分比或者px表示的时候，其值可以为负数。我们应用坐标规则就很容易理解负数表示的意义，x为负数时候表示图片左顶点在容器左顶点的左侧，y为负数时表示图片的左顶点在容器的左定点的上方。也就是向左和向上超出容器的范围。**

### 157)布局的几种方式：

**（1）流体布局：**

**流布局与固定宽度布局基本不同点就在于对网站尺寸的测量单位不同。固定宽度布局使用的是像素，但是流布局使用的是百分比，看到百分比，你应该想到，这将提供了很强的可塑性和流动性。换句话说，通过设置了百分比，你将不需要考虑设备尺寸或者屏幕宽度大小了，结论就是，你可以为每种情形找到一种可行的方案，因为你的设计尺寸将适应所有的设备尺寸。流布局与媒体查询和优化样式技术的关系密切。**

**（2）固定布局**

**在固定布局中，网页的宽度是必须指定为一个像素值，一般设为960像素。在过去，开发人员发现960像素是最适合作为网格布局的宽度，因为960可以整除3，4，5，6，8，10，12和15。尽管到了今天，在web开发中还是比较普遍使用固定宽度布局的，原因是这种布局提供很强的稳定性与可控性。**

**（3）弹性布局：**

**弹性布局跟流布局很相像，主要不同是大小单位。弹性布局的大小单位不是像素或者百分比，而是em。一个em单位等于定义在CSS规则：font-size的值大小。例如，我们设置文本的font-size大小为20像素，那么1em就等于20px，2em就等于40px，以此类推。这种布局为开发人员提供了一种很强的排版控制。**

**（4）混合布局**

**最后介绍这个混合布局，是上面介绍的两种或者更多布局类型的组合。例如，开发人员可能需要设置某些指定元素（侧栏或者底部）一个固定宽度，剩下的就选择百分比或者em。**

### 158）em是CSS中的一个相对单位，它的单位长度是根据元素的文本垂直长度来决定的。可以作用在width、height、line-height、margin、padding、border等样式的设置上。子元素div的font-size是根据其父元素body的font-size确定的。

### 159）vue组件原理：

**组件生命周期：**

**创建vue对象，监听data对象数据变化，Vue内部初始化事件，编译模版把data数据和模版生成html.用上面编译好的html内容替换el属性指向的dom对象或选择权对应的html标签里的内容，实时监听数据变化更新DOM，调用vm.$destroy()销毁实例，destroyed.**

**仔细观察上面的js代码，我们发现vue实例化一个组件可以分成三步。**

**使用extend定义（构造）组件MyComponent**

**使用component注册组件**

**在初始化app实例的过程中，渲染组件**

**·1，定义组件**

**·2，注册组件**

**注册功能只需要完成组件与标签名的映射就可以了**

**·3渲染组件**

**细分为以下三个步骤。**

**识别组件**

**组件指令化**

**渲染、挂载组件**

**vue双向绑定的实现原理：**

**一、访问器属性**

**访问器属性是对象中的一种特殊属性，它不能直接在对象中设置，而必须通过 defineProperty() 方法单独定义。**

**var obj = { };**

**// 为obj定义一个名为 hello 的访问器属性**

**Object.defineProperty(obj, "hello", {**

**get: function () {return sth},**

**set: function (val) {/\* do sth \*/}**

**})**

**二,DocumentFragment:**

**Vue 进行编译时，就是将挂载目标的所有子节点劫持（真的是劫持，通过 append 方法，DOM 中的节点会被自动删除）到 DocumentFragment 中，经过一番处理后，再将 DocumentFragment 整体返回插入挂载目标。**

**三,观察者模式**

向子组件传递数据;

组件实例的作用域是**孤立的**。这意味着不能 (也不应该) 在子组件的模板内直接引用父组件的数据。要让子组件使用父组件的数据，我们需要通过子组件的 **props** 选项。

**[watch对对象进行深度观察](https://cn.vuejs.org/v2/api/" \l "watch)**

data() {

　　return {

　　　　bet: {  
　　　　　　pokerState: 53,  
　　　　　　pokerHistory: 'local'  
　　　　}

}

},

watch: {

　　bet: {  
　　　　handler(newValue, oldValue) {  
　　　　　　console.log(newValue)  
　　　　},  
　　　　deep: true  
　　}  
}

**Compile:**

**transclude**

transclude的意思是内嵌，这个步骤会把你template里给出的模板转换成一段dom，然后抽取出你el选项指定的dom里的内容

**compile**

compile的的过程具体就是\*\*遍历模板解析出模板里的指令\*\*。更精确的说是解析后生成了指令描述对象。

同时，compile函数是一个高阶函数，他执行完成之后的返回值是另一个函数：link，所以compile函数的第一个阶段是编译,返回出去的这个函数完成另一个阶段：link。

**link**

compile阶段将指令解析成为指令描述对象(descriptor)，闭包在了link函数里，link函数会把descriptor传入Directive构造函数，创建出真正的指令实例。此外link函数是作为参数传入linkAndCaptrue中的，后者负责执行link，同时取出这些新生成的指令，先按照指令的预置的优先级从高到低排好顺序，然后遍历指令执行指令的\_bind方法，这个方法会为指令创建watcher，并计算表达式的值，完成前面描述的依赖收集。并最后执行对应指令的bind和update方法，使指令生效、界面更新。

此外link函数最终的返回值是unlink函数，负责在vm卸载时取消对应的dom到数据的绑定。

生命周期主要分为4个过程：

create。new Vue时，会先进行create，创建出Vue对象。

mount。根据el, template, render方法等属性，会生成DOM，并添加到对应位置。

update。当数据发生变化后，会重新渲染DOM，并进行替换。

destory。销毁时运行。

**PROPS:**组件实例的作用域是**孤立的**。这意味着不能 (也不应该) 在子组件的模板内直接引用父组件的数据。要让子组件使用父组件的数据，我们需要通过子组件的 **props** 选项。

每次父组件更新时，子组件的所有 prop 都会更新为最新值。这意味着你**不应该**在子组件内部改变 prop。如果你这么做了，Vue 会在控制台给出警告。

为什么我们会有修改 prop 中数据的冲动呢？通常是这两种原因：

prop 作为初始值传入后，子组件想把它当作局部数据来用；

prop 作为初始值传入，由子组件处理成其它数据输出。

对这两种原因，正确的应对方式是：

定义一个局部变量，并用 prop 的值初始化它：

|  |
| --- |
| props: ['initialCounter'],  data: function () {  return { counter: this.initialCounter }  } |

定义一个计算属性，处理 prop 的值并返回。

|  |
| --- |
| props: ['size'],  computed: {  normalizedSize: function () {  return this.size.trim().toLowerCase()  }  } |

**[Prop 验证](https://cn.vuejs.org/v2/guide/components.html" \l "Prop-验证)**

propA:Number或propA:{type:Number}

type 可以是下面原生构造器：

String

Number

Boolean

Function

Object

Array

Symbol

其它验证：

required: true//指定它为必选段

default: 100//默认值，可以是函数

validator: function (value) {return value > 10}//验证条件

### 160）angular核心：

**·模块化和依赖注入**

**·双向数据绑定**

**·指令**

**HTML Parser&dirctive：**

**complie:变量DOM，指令优先级排序，执行compile函数，把每个compile函数返回的link函数打包到总的link函数中。**

**link:将scope绑定到DOM上，在元素上注册事件监听器，使用$watch监控数据模型。**

**Two way DataBinding:**

**核心：脏检查 难点：震荡问题或循环依赖**

**Dependeny Injection:**

**1.分析匹配参数名 2.fn.$injections=[] 3.使用fn.call或apply传递需要注入的对象**

**Module&controller&service**

**1·AngularJS 指令**

**ng-app 指令初始化一个 AngularJS 应用程序。**

**ng-init 指令初始化应用程序数据。**

**ng-model 指令把元素值（比如输入域的值）绑定到应用程序**

**自定义的指令：app.directive("runoobDirective", function() {｝）**

**2·ng-model**

**指令可以将输入域的值与 AngularJS 创建的变量绑定**

**3·$scope 作用域：**

**各作用域之间遵循 JS 的对象原型继承方式。**

**$scope 的使用贯穿整个 Angular App 应用,它与数据模型相关联,同时也是表达式执行的上下文.有了 $scope 就在视图和控制器之间建立了一个通道,基于作用域视图在修改数据时会立刻更新 $scope,同样的 $scope 发生改变时也会立刻重新渲染视图。$rootScope 是所有 $scope 的最上层对象,可以理解为一个 Angular 应用中得全局作用域对象,所以为它附加太多逻辑或者变量并不是一个好主意。**

**系统的划分它的作用和功能:**

**提供了观察者可以监听数据模型的变化**

**可以将数据模型的变化通知给整个 App**

**可以进行嵌套,隔离业务功能和数据**

**给表达式提供上下文执行环境**

**$scope 的生命周期有4个阶段:**

**1. 创建**

**控制器或者指令创建时, Angular 会使用 $injector 创建一个新的作用域,然后在控制器或指令运行时,将作用域传递进去.**

**2. 链接**

**Angular 启动后会将所有 $scope 对象附加或者说链接到视图上,所有创建 $scope 对象的函数也会被附加到视图上.这些作用域将会注册当 Angular 上下文发生变化时需要运行的函数.也就是 $watch 函数, Angular 通过这些函数或者何时开始事件循环.**

**3. 更新**

**一旦事件循环开始运行,就会开始执行自己的脏值检测.一旦检测到变化,就会触发 $scope 上指定的回调函数**

**4. 销毁**

**通常来讲如果一个 $scope 在视图中不再需要, Angular 会自己清理它.当然也可以通过 $destroy() 函数手动清理.**

**4·app.controller('name',function($scope){}) 控制器：**

**ng-controller 指令定义了应用程序控制器。控制器的 $scope （相当于作用域、控制范围）用来保存AngularJS Model(模型)的对象。**

**5·过滤器：**

**过滤器可以使用一个管道字符（|）添加到表达式和指令中。可以多个连用{{ expression | filter1 | filter2 | ... }}**

**ng的内置过滤器：**

**1. currency (货币处理)**

**使用currency可以将数字格式化为货币，默认是美元符号，你可以自己传入所需的符号，例如我传入人民币：**

**{{num | currency : '￥'}}**

**2. date (日期格式化)**

**原生的js对日期的格式化能力有限，ng提供的date过滤器基本可以满足一般的格式化要求。用法如下：**

**{{date | date : 'yyyy-MM-dd hh:mm:ss EEEE'}}**

**3. filter(匹配子串)**

**用来处理一个数组，然后可以过滤出含有某个子串的元素，作为一个子数组来返回。可以是字符串数组，也可以是对象数组。如果是对象数组，可以匹配属性的值。它接收一个参数，用来定义子串的匹配规则。**

**$scope.childrenArray = [**

**{name:'kimi',age:3},**

**{name:'cindy',age:4}**

**];**

**$scope.func = function(e){return e.age>4;}**

**{{ childrenArray | filter : 'a' }} //匹配属性值中含有a的**

**{{childrenArray | filter : func }} //参数是函数，指定返回age>4的**

**4. json(格式化json对象)**

**json过滤器可以把一个js对象格式化为json字符串，作用就和我们熟悉的JSON.stringify()一样：**

**{{ jsonTest | json}}**

**5. limitTo(限制数组长度或字符串长度)**

**limitTo过滤器用来截取数组或字符串，接收一个参数用来指定截取的长：**

**{{ childrenArray | limitTo : 2 }} //将会显示数组中的前两项**

**6. lowercase(小写)**

**把数据转化为全部小写。**

**7. uppercase(大写)**

**8. number(格式化数字)**

**number过滤器可以为一个数字加上千位分割，像这样，123,456,789。同时接收一个参数，可以指定小float类型保留几位小数：**

**{{ num | number : 2 }}**

**9. orderBy(排序)**

**orderBy过滤器可以将一个数组中的元素进行排序，接收一个参数来指定排序规则，参数可以是一个字符串，表示以该属性名称进行排序。**

**{{ childrenArray | orderBy : 'age' }}//按age属性值进行排序**

**自定义过滤器：**

**app.filter('odditems',function(){})**

**6·AngularJS 服务(Service)：**

**在 AngularJS 中，服务是一个函数或对象，可在你的 AngularJS 应用中使用。AngularJS 内建了30 多个服务。**

**$http 服务向服务器请求数据**

**app.controller('myCtrl', function($scope, $http) {**

**$http.get("welcome.htm").then(function (response) {**

**$scope.myWelcome = response.data;**

**});**

**});**

**$timeout 服务**

**$interval 服务**

**创建自定义服务：app.service('zwj', function() {｝）； app.controller('myCtrl', function($scope, zwj) {})**

**7·AngularJS 事件**

**ng-click ，ng-dblclick ，ng-blur ，ng-change ，ng-copy ......**

**<button ng-click="toggle()">隐藏/显示</button>**

**app.controller('personCtrl', function($scope) {**

**$scope.toggle = function() {**

**$scope.myVar = !$scope.myVar;**

**};**

**});**

**8·AngularJS 模块：**

**angular.module 函数来创建模块。模块定义了一个应用程序。模块是应用程序中不同部分的容器。模块是应用控制器的容器。控制器通常属于一个模块。angular.module（'name',[]）;[]内为模块依赖**

**9·AngularJS 输入验证**

**$dirty 表单有填写记录**

**$valid 字段内容合法的**

**$invalid 字段内容是非法的**

**$pristine 表单没有填写记录**

**$error 记录错误信息**

**AngularJS内置的验证器**

**验证器 描述**

**required 正如前面所讨论的那样，这确保字段是必需的，字段标记为无效，直到填写。**

**ng-required 与required不同，这标志着字段也是必需的，ng-required指令允许我们基于控制器布尔条件标记输入字段为必需。**

**ng-minlength 该指令可以设定输入文本值的最小长度**

**ng-maxlength 该指令可以设定文本输入值的最小长度**

**ng-pattern 作为这个指令的一部分，文本输入的有效性可以针对指定的正则表达式模式进行检查。**

**type=”email” 文本输入内置电子邮件验证。**

**type=”number” 带有数量验证的文本输入。也可以有最小和最大值的附加属性。**

**type=”date” 如果浏览器支持，显示一个HTML日期选择器。否则，默认为一个文本输入。ng-model**

**绑定到日期对象。预计日期格式为yyyy-mm-dd(例如2009-10-24)。**

**type=”url” 带有输入验证的URL文本输入。**

**10·AngularJS API：**

**AngularJS 全局 API 用于执行常见任务的 JavaScript 函数集合。**

**angular.bind**

**angular.bootstrap**

**angular.copy**

**angular.element**

**angular.equals**

**angular.extend**

**angular.forEach**

**angular.fromJson**

**angular.identity**

**angular.injector**

**angular.isArray**

**angular.isDate**

**angular.isDefined**

**angular.isElement**

**angular.isFunction**

**angular.isNumber**

**angular.isObject**

**angular.isString**

**angular.mock**

**angular.module**

**angular.noop**

**angular.toJson**

**angular.uppercase**

**angular.version**

**11·AngularJS 包含：**

**在 AngularJS 中，你可以在 HTML 中包含 HTML 文件。<div ng-include="'runoob.htm'"></div>**

**12·AngularJS 动画**

**AngularJS 提供了动画效果，可以配合 CSS 使用。AngularJS 使用动画需要引入 angular-animate.min.js 库。**

**13·AngularJS 依赖注入**

**$provide服务负责告诉Angular如何创造一个新的可注入的东西：即服务(service)，服务会被叫做provider的东西来定义，你可以使用$provide来创建一个provider，**

**app.config(function($provide) {**

**$provide.provider('greeting', function() {this.$get = function() {return function(name) {alert("Hello, " + name);};};})；});**

**factory，service以及value全部都是用来定义一个providr的简写 。**

**注入器（$injector）：注入器负责从我们通过$provide创建的服务中创建注入的实例。**

**配置provider：provider允许我们进行一些配置。**

**创建被调用者的实例的工作由IOC容器来完成，然后注入调用者。（IOC:英文全称：Inversion of Control，中文名称：控制反转，它还有个名字叫依赖注入（Dependency Injection））**

**14·AngularJS 路由**

**<li><a href="#/">首页</a></li>**

**<div ng-view></div>**

**angular.module('routingDemoApp',['ngRoute'])**

**.config(['$routeProvider', function($routeProvider){**

**$routeProvider**

**.when('/',{template:'这是首页页面'})**

**.when('/computers',{template:'这是电脑分类页面'})**

**.when('/printers',{template:'这是打印机页面'})**

**.otherwise({redirectTo:'/'});**

**}]);**

**template:**

**如果我们只需要在 ng-view 中插入简单的 HTML 内容，则使用该参数：**

**.when('/computers',{template:'这是电脑分类页面'})**

**templateUrl:**

**如果我们只需要在 ng-view 中插入 HTML 模板文件，则使用该参数：**

**$routeProvider.when('/computers', {**

**templateUrl: 'views/computers.html',**

**});**

**以上代码会从服务端获取 views/computers.html 文件内容插入到 ng-view 中。**

**controller:**

**function、string或数组类型，在当前模板上执行的controller函数，生成新的scope。**

**controllerAs:**

**string类型，为controller指定别名。**

**redirectTo:**

**重定向的地址。**

**resolve:**

**指定当前controller所依赖的其他模块。**

**路由有几个常用的事件：**

**$routeChangeStart：这个事件会在路由跳转前触发**

**$routeChangeSuccess：这个事件在路由跳转成功后触发**

**$routeChangeError：这个事件在路由跳转失败后触发**

**15·脏检查**

**脏检查，将原对象复制一份快照，在某个时间，比较现在对象与快照的值，如果不一样就表明发生了变化，这个策略要保留两份变量，而且要遍历对象，比较每个属性，这样会有一定的性能问题**

**$watch 对象，Angular 每一个绑定到UI的数据，就会有一个 $watch 对象。**

**$scope.$watch( watchExp, listener, objectEquality ); 参数存储在$$watchers数组中。$watch中会记录上旧的值。**

**为了监视一个变量的变化，你可以使用$scope.$watch函数。这个函数有三个参数，它指明了”要观察什么”(watchExp)，”在变化时要发生什么”(listener),以及你要监视的是一个变量还是一个对象。**

**$scope.$apply**

**当一个控制器/指令/等等东西在AngularJS中运行时，AngularJS内部会运行一个叫做$scope.$apply的函数。。**

**$apply方法。这个方法能够触发$digest方法。$digest方法的执行就标志着一轮Digest Cycle的开始。**

**angular所系统的方法中都会触发比较事件，比如：controller初始化的时候，所有以ng-开头的事件爱你执行后。**

**$scope.$apply(function(){**

**$scope.date = new Date();**

**//在这里去手动触发脏检查**

**})**

**angular并不是定时的。会触发脏检测的有以下：**

**controller 初始化**

**几乎所有ng-开头的事件(ng-click,ng-change...)**

**http请求**

**$timeout,$interval**

**手动调用$apply(), $digest()**

**16·angularjs的$on、$emit、$broadcast**

**$emit只能向parent controller传递event与data**

**$broadcast只能向child controller传递event与data**

**$on用于接收event与data**

**编译函数(compile)负责对模板DOM进行转换。**

**链接函数(link)负责将作用域和DOM进行链接。**

### 161)MVVM是Model-View-ViewModel

Model 是数据，view是UI，

ViewModel就是View与Model的连接器，部分代码可以写成公用的框架。

唯一的区别是，它采用双向绑定（data-binding）：View的变动，自动反映在 ViewModel



**MVP:**MVP 模式将 Controller 改名为 Presenter，同时改变了通信方向。



1. 各部分之间的通信，都是双向的。

2. View 与 Model 不发生联系，都通过 Presenter 传递。

3. View 非常薄，不部署任何业务逻辑，称为"被动视图"（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

### 162)去除浮动：（float会造成父页面高度坍塌）

**:after{**

**clear: both;**

**display: table;**

**content: " ";**

**}**

### 163)<meta http-equiv="refresh" content="6"> 6秒刷新一下

### 164)异步加载js的方式：

**1）.在html5中，script新增了async的属性，script添加了该属性之后，下载脚本时将可以与页面其他内容并行下载，但是该属性必须在ie9以上的浏览器中才可以使用，并且只能用于加载外部js脚本。<script async src="example.js"></script>**

**2）html4中也有一个defer属性。asyc与defer属性的不同点是，async会让脚本在加载完可用时立即执行，而defer脚本则会在dom加载完毕后执行,即使脚本已经下载。**

**3）利用XHR异步加载js内容并执行。**

**<script>**

**var xhr = new XMLHttpRequest();**

**xhr.open("get", "js/defer.js",true)**

**xhr.send();**

**xhr.onreadystatechange = function() {**

**if (xhr.readyState == 4 && xhr.status == 200) {**

**eval(xhr.responseText);**

**}**

**}**

**</script>**

**4）动态创建script标签**

**var DOMScript=document.createElement("script");**

**DOMScript.src="someting.js";**

**DOMScript.onloadDone=false;**

**DOMScript.onload=function(){**

**DOMScript.onloadDone=true;**

**init();**

**}**

**DOMScript.onreadystatechange=function(){**

**if(("loaded" === DOMScript.readyState || "complete" ===DOMScript.readyState) && ! DOMScript.onloadDone){**

**DOMScript.onloadDone=true;**

**init();**

**}**

**}**

**5）.iframe方式，利用iframe加载一个同源的子页面，让子页面内的js影响当前父页面的一种方式。**

### 165) http://yslow.org/ yslow下载地址，测试网页性能

**https://chrome.google.com/webstore/detail/yslow/ninejjcohidippngpapiilnmkgllmakh#**

### 166) CSS expressions:css表达式。（已淘汰）

**CSS属性的值等于Javas cript表达式计算的结果。left: expression(document.body.offsetWidth - 110 + "px");**

### 167)js动画:

**缺点：(1)JavaScript在浏览器的主线程中运行，而主线程中还有其它需要运行的JavaScript脚本、样式计算、布局、绘制任务等,对其干扰导致线程可能出现阻塞，从而造成丢帧的情况。**

**(2)代码的复杂度高于CSS动画.**

**优点：(1)JavaScript动画控制能力很强**

**(2)动画效果比css3动画丰富,有些动画效果**

**(3)CSS3有兼容性问题，而JS大多时候没有兼容性问题**

**CSS动画:**

**缺点：(1)运行过程控制较弱,无法附加事件绑定回调函数。**

**(2)代码冗长。想用 CSS 实现稍微复杂一点动画,最后CSS代码都会变得非常笨重。**

**优点： (1)浏览器可以对动画进行优化。**

**(2)强制使用硬件加速 （通过 GPU 来提高动画性能）**

**CSS animations, transforms 以及 transitions 不会自动开启GPU加速，而是由浏览器的缓慢的软件渲染引擎来执行。那我们怎样才可以切换到GPU模式呢，很多浏览器提供了某些触发的CSS规则。如：**translate3d(x,y,z).

### 167)p:nth-child(odd)

**{**

**background:#ff0000;**

**}**

**p:nth-child(even)**

**{**

**background:#0000ff;**

**}**

**p:nth-child(3n+0)**

**{**

**background:#ff0000;**

**}**

### 168)background-size:

**cover:把背景图像扩展至足够大，以使背景图像完全覆盖背景区域。背景图像的某些部分也许无法显示在背景定位区域中。**

**contain:把图像图像扩展至最大尺寸，以使其宽度和高度完全适应内容区域。**

### 169)div垂直居中：

**top:50%;margin-top:-100px;(先把位置居中，再根据div宽高调整margin-top,margin-left);**

**升级写法：{**

**width: 600px; height: 400px;**

**position: absolute; left: 50%; top: 50%;**

**transform: translate(-50%, -50%); /\* 50%为自身尺寸的一半 \*/**

**}**

**translate(-50%,-50%) 作用是，往上（x轴）,左（y轴）移动自身长宽的 50%，以使其居于中心位置**

**升级兼容：{**

**width: 600px; height: 400px;**

**position: absolute; left: 0; top: 0; right: 0; bottom: 0;**

**margin: auto; /\* 有了这个就自动居中了 \*/**

**}**

auto，其计算值取决于可用空间（剩余空间）。

**{**

display: table-cell;

    vertical-align: middle;

    text-align: center;

**}**

{

    display: flex;

    justify-content:center;/\*justify-content定义项目在主轴上的对齐（左对齐，右对齐，居中）\*/

    align-items:Center;/\*align-items属性定义多个项目在**一行**(交叉轴)上如何对齐。\*/

}

文字垂直居中：line-height设置为高大小。line-height：200%根据自己的font-size设置百分比，font-size默认16px，让子元素继承。line-height:1.5em，em根据自己字体计算，让子元素继承。line-height：1.5,也是根据字体计算。

### 170)null与undefined:

**null表示"没有对象"，即该处不应该有值.typeof null ==>Object**

**undefined表示"缺少值"，即此处应该有一个值，但是还没有定义。**

**typeof undefined==>undefined**

**false与null不相等。false与**undefined不相等。Undefined与null相等。

null与””不相等,undefined与””不相等。false与””相等

Typeof null === ‘object’

由于 null 代表的是空指针（大多数平台下值为 0x00），因此，null 的类型标签是 0，typeof null 也因此返回 "object"。

虽然 typeof null 会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象然而 null 表示为全零，所以将它错误的判断为 object。

类型标签存储在单元的低位中：

000：object        数据是对对象的引用。

1：int        数据是一个31位有符号整数。

010：double        数据是对双浮点数的引用。

100：string        数据是对字符串的引用。

110：boolean        数据是一个布尔值。

### 171)HTML5新增了哪些内容或API

Drag&Drop API (拖放)

History  API（历史记录）

Notification API (桌面通知)

Communication API （跨域通信）

WebSocket API

Web Worker

Storage API

Canvas和WebGL

Device API：Geolocation API（[地理位置](http://www.w3school.com.cn/html5/html_5_geolocation.asp)API），Calendar API

[FileReader](http://mdn.beonex.com/en/DOM/FileReader.html)

[querySelector()](http://mdn.beonex.com/en/DOM/Locating_DOM_elements_using_selectors.html)

### 172)[HTML中<input>和<textarea>的区别](http://www.cnblogs.com/abcd1234/p/4709486.html)

一个是<input>元素的单行文本框，不能自动换行，用来放置字数较少的单行文字内容；一种是<textarea>的多行文本框。

### 173)Div实现textarea

<div contenteditable="true"></div>（核心：contenteditable（html5）,再调整css即可）

### 174)阻止移动设备自动识别页面上的电话号码、email地址

<meta name="format-detection" content="telephone=no" /><meta name="format-detection" content="email=no" /><meta name="format-detection" content="address=no" /><meta name="format-detection" content="date=no" />

### 175)[左边定宽，右边自适应布局的几种方法](http://www.cnblogs.com/wx1993/p/6727653.html)

1.

.left{  
 float: left;  
 width: 200px;  
 background: darkred;  
}  
.right{  
 width: 100%;  
 background: green;  
}

2.

.main{  
 display: flex;  
}  
.left{  
 width: 100px;  
 background: darkred;  
}  
.right{  
 flex: 1;  
 background: green;  
}

3.

.left{  
 float: left;  
 width: 200px;  
 margin-right: -100%;  
 background: #be4f4f;  
}  
.main{  
 float: left;  
 width: 100%;  
}  
.right{  
 margin-left: 200px;  
 background: #5a6a94;  
}

<div class="left">left</div>  
<div class="main">  
 <div class="right">right</div>  
</div>

### 176)flex  Flex 布局以后，子元素的float、clear和vertical-align属性将失效。

flex-direction: row | row-reverse | column | column-reverse;

属性决定主轴的方向（即项目的排列方向）

* row（默认值）：主轴为水平方向，起点在左端。
* row-reverse：主轴为水平方向，起点在右端。
* column：主轴为垂直方向，起点在上沿。
* column-reverse：主轴为垂直方向，起点在下沿。

flex-wrap: nowrap | wrap | wrap-reverse;

默认情况下，项目都排在一条线（又称"轴线"）上。flex-wrap属性定义，如果一条轴线排不下，如何换行。

flex-flow: <flex-direction> || <flex-wrap>;

flex-flow属性是flex-direction属性和flex-wrap属性的简写形式，默认值为row nowrap。

justify-content: flex-start | flex-end | center | space-between | space-around;

* justify-content属性定义了项目在主轴上的对齐方式。
* flex-start（默认值）：左对齐
* flex-end：右对齐
* center： 居中
* space-between：两端对齐，项目之间的间隔都相等。
* space-around：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。

align-items: flex-start | flex-end | center | baseline | stretch;

* align-items属性定义项目在交叉轴上如何对齐。
* flex-start：交叉轴的起点对齐。
* flex-end：交叉轴的终点对齐。
* center：交叉轴的中点对齐。
* baseline: 项目的第一行文字的基线对齐。
* stretch（默认值）：如果项目未设置高度或设为auto，将占满整个容器的高度。

align-content: flex-start | flex-end | center | space-between | space-around | stretch;

align-content属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

* flex-start：与交叉轴的起点对齐。
* flex-end：与交叉轴的终点对齐。
* center：与交叉轴的中点对齐。
* space-between：与交叉轴两端对齐，轴线之间的间隔平均分布。
* space-around：每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。
* stretch（默认值）：轴线占满整个交叉轴。

Order

order属性定义项目的排列顺序。数值越小，排列越靠前，默认为0。

flex-grow

flex-grow属性定义项目的放大比例，默认为0，即如果存在剩余空间，也不放大。

flex-shrink

flex-shrink属性定义了项目的缩小比例，默认为1，即如果空间不足，该项目将缩小。

flex-basis

flex-basis属性定义了在分配多余空间之前，项目占据的主轴空间（main size）。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为auto，即项目的本来大小。

Flex

flex属性是flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选。

align-self

align-self属性允许单个项目有与其他项目不一样的对齐方式，可覆盖align-items属性。默认值为auto，表示继承父元素的align-items属性，如果没有父元素，则等同于stretch。

### 177)在普通流中的盒子会参与一种格式上下文,这个盒子可能是块盒也可能是行内盒,但不可能同时是块盒又是行内盒。块级盒参与块级格式上下文(BFC),行内级盒参与行级格式上下文(IFC)。

**BFC**

BFC（Block Formatting Context）叫做“块级格式化上下文”。

**BFC的布局规则如下：**

1.内部的盒子会在垂直方向，一个个地放置；

2.盒子垂直方向的距离由margin决定，**属于同一个BFC的两个相邻Box的上下margin会发生重叠**；

3.每个元素的左边，与包含的盒子的左边相接触，即使存在浮动也是如此；

4.BFC的区域不会与float重叠；

5.BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素，反之也如此；

6.计算BFC的高度时，浮动元素也参与计算。

**介绍过了BFC的布局规范，再来说说哪些元素会产生BFC。**

1.根元素；

2.float的属性不为none；

3.position为absolute或fixed；

4.display为inline-block，table-cell，table-caption，flex；

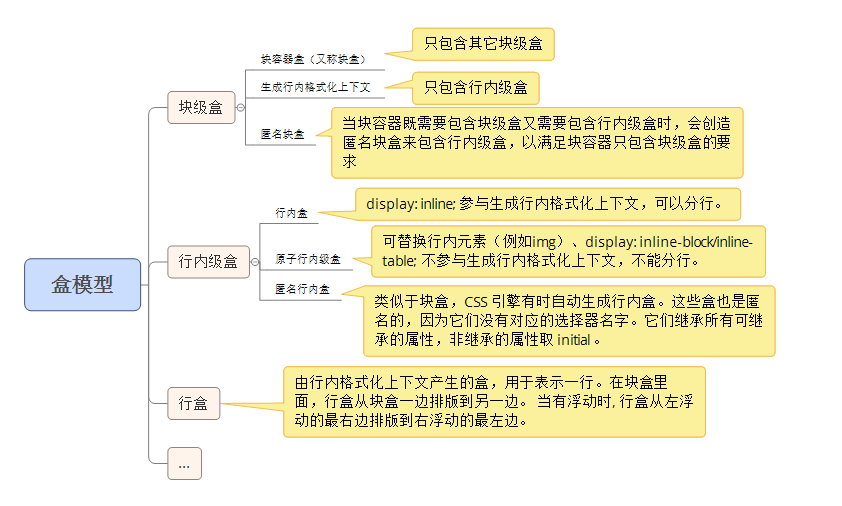
5.overflow不为visible

**IFC**

(Inline Formatting Contexts)直译为"内联格式化上下文"

在IFC中，盒子水平放置，一个接着一个，从包含块的顶部开始。水平margins,borders,和padding在这些盒子中被平分。这些盒子也许通过不同的方式进行对齐:他们的地步和顶部也许被对齐，或者通过文字的基线进行对齐。矩形区域包含着来自一行的盒子叫做line box。

**IFC布局规则：**

1. 框会从包含块的顶部开始，一个接一个地水平摆放。
2. 摆放这些框的时候，它们在水平方向上的外边距、边框、内边距所占用的空间都会被考虑在内。在垂直方向上，这些框可能会以不同形式来对齐：它们可能会把底部或顶部对齐，也可能把其内部的文本基线对齐。能把在一行上的框都完全包含进去的一个矩形区域，被称为该行的行框。水平的margin、padding、border有效，垂直无效。不能指定宽高。
3. 行框的宽度是由包含块和存在的浮动来决定。行框的高度由行高计算这一章所描述的规则来决定。

**GFC**

**GFC**(GridLayout Formatting Contexts)直译为"网格布局格式化上下文"

**FFC**  
FFC(Flex Formatting Contexts)直译为"自适应格式化上下文"

### 178）display

#### block

**【特征】**

　　[1]不设置宽度时，宽度为父元素宽度

　　[2]独占一行

　　[3]支持宽高

**【标签】**

<address><article><aside><blockquote><body><dd><details><div><dl><dt><fieldset><figcaption><figure><footer><form><h1><header><hgroup><hr><html><legend><menuitem><nav>**<ol><**optgroup><option><p><section><summary>**<ul><li>**

　　[注意]menuitem标签只有firefox支持

**【不支持的样式】**

　　[1]vertical-align

#### inline

**【特征】**

　　[1]内容撑开宽度

　　[2]非独占一行

**[3]不支持宽高,width,height失效。Padding,margin有效。**

　　[4]代码换行被解析成空格

**【标签】**

**<a><**abbr><area><b><bdi><bdo><br><cite><code><del><dfn><em>**<i>**<ins><kbd>**<label>**<map><mark><output><pre><q><rp><rt><ruby><s><smap><small>***<span><strong>***<sub><sup><time><u><var><wbr>

**【不支持的样式】**

　　[1]background-position

　　[2]clear

　　[3]clip

　　[4]height | max-height | min-height

　　[5]width | max-width | min-width

　　[6]overflow

　　[7]text-align

　　[8]text-indent

　　[9]text-overflow

#### inline-block

**【特征】**

　　[1]不设置宽度时，内容撑开宽度

　　[2]非独占一行

**[3]支持宽高**

　　[4]代码换行被解析成空格

**【标签】**

<audio>**<button><canvas>**<embed>**<iframe><img><input>**<keygen><meter><object><progress><***select>***<textarea><video>

**【不支持的样式】**

　　[1]clear

 【IE兼容】

　　IE7-浏览器不支持给块级元素设置inline-block样式，解决方法如下：首先将其变成行内元素，使用具有行内元素的特性，然后触发haslayout，使其具有块级元素的特性，如此就可以模拟出inline-block的效果

div{

display:inline-block;

\*display: inline;

zoom: 1;

　　[注意][关于inline-block元素底部空隙的问题移步到此](http://www.cnblogs.com/xiaohuochai/p/5271217.html#anchor2)

#### none

**【特征】**

　　隐藏元素并脱离文档流

**【标签】**

<base><link><meta><title><datalist><dialog><param>**<script><source><style>**

#### list-item

**【特征】**

　　[1]不设置宽度时，宽度撑满一行

　　[2]独占一行

　　[3]支持宽高

#### run-in

　　run-in是一个有意思的**块/行**内元素混合，可以使某些块级元素成为下一个元素的行内部分。如果一个元素生成run-in框，而且该框后面是一个块级框，那么该run-in元素将成为块级框开始处的一个行内框，run-in框格式化成另一个元素中的行内框，但它们仍从文档中的父元素继承属性

　　[注意]只有safari和IE8+支持

<h3 style="display:run-in">run-in test</h3><p>paragraph</p>

　　若run-in框后面不是块级框时，run-in框本身将成为块级框

<span style="display:run-in">run-in test</span><span>paragraph</span>

#### 表格类元素

table{display: table;}

thead{display: table-header-group;}

tbody{display: table-row-group;}

tfoot{display: table-footer-group;}

tr{display: table-row;}

td,th{display: table-cell;}

col{display: table-column;}

colgroup{display: table-column-group;}

caption{display: table-caption;}

　　表格类元素的display共有以上几种，<thead><tbody><tfoot><tr><col><colgroup>因为无法设置margin和padding用的较少，下面将着重介绍下<table>、<td>、<th>、<caption>这四个标签对应的display属性

##### table

**【特征】**

　　[1]不设置宽度时，宽度由内容撑开

　　[2]独占一行

　　[3]支持宽高

　　[4]默认具有表格特征，可设置table-layout、border-collapse、border-spacing等表格专有属性

　　[注意]对于display为table和inline-table，若处于分隔边框模型即border-collapse:separate;，margin和padding都可设置；若处于合并边框模型即border-collapse:collapse，只可设置margin

##### inline-table

**【特征】**

　　[1]不设置宽度时，宽度由内容撑开

　　[2]非独占一行

　　[3]支持宽高

　　[4]默认具有表格特征，可设置table-layout、border-collapse、border-spacing等表格专有属性

##### table-cell

**【特征】**

　　[1]不设置宽度时，宽度由内容撑开

　　[2]非独占一行

　　[3]支持宽高

　　[4]垂直对齐

　　[5]同级等高

　　[注意]display:table-cell的元素不可以设置margin，但可以设置padding

##### table-caption

**【特征】**

　　[1]不设置宽度时，宽度由内容撑开

　　[2]独占一行

　　[3]支持宽高

　　[注意]display:table-caption的元素margin和padding都可设置

### 179)清除浮动的方式：

父级div定义 伪类:after

.outer {zoom:1;} /\*==for IE6/7 Maxthon2==\*/

.outer :after {

clear:both;

content:' ';

display:block;

height: 0;

visibility:hidden;

}

原理同**clear：both。"display: block; clear: both;"是确保这个空白字符是非浮动的独立区块。**

**添加新的元素，应用clear：both**

<div class="clear"></div>

.clear{clear:both; height: 0; line-height: 0; font-size: 0}

**原理是父容器现在必须考虑非浮动子元素的位置，而后者肯定出现在浮动元素下方，所以显示出来，父容器就把所有子元素都包括进去了。这种方法比较简单，但是要在页面中增加冗余标签，违背了语义网的原则。**

**给父级元素定义overflow**

.over-flow{

overflow: auto; //也可以设置hidden

zoom: 1; //zoom: 1; 处理兼容性问题

}

**由于加上overflow:hidden之后要计算超出的大小来隐藏，所以父容器会自动撑开自己把所有的子元素放进去，同时会计算浮动的子元素。**

### 180)css隐藏元素

{ display: none; /\* 不占据空间，无法点击 \*/ }

{ visibility: hidden; /\* 占据空间，无法点击 \*/ }

{ position: absolute; clip:rect(1px 1px 1px 1px); /\* 不占据空间，无法点击 \*/ }

{ position: absolute; top: -999em; /\* 不占据空间，无法点击 \*/ }

{ position: relative; top: -999em; /\* 占据空间，无法点击 \*/ }

{ position: absolute; visibility: hidden; /\* 不占据空间，无法点击 \*/ }

{ height: 0; overflow: hidden; /\* 不占据空间，无法点击 \*/ }

{ opacity: 0; filter:Alpha(opacity=0); /\* 占据空间，可以点击 \*/ }

{ position: absolute; opacity: 0; filter:Alpha(opacity=0); /\* 不占据空间，可以点击 \*/ }

{

zoom: 0.001;

-moz-transform: scale(0);

-webkit-transform: scale(0);

-o-transform: scale(0);

transform: scale(0);

/\* IE6/IE7/IE9不占据空间，IE8/FireFox/Chrome/Opera占据空间。都无法点击 \*/

}

{

position: absolute;

zoom: 0.001;

-moz-transform: scale(0);

-webkit-transform: scale(0);

-o-transform: scale(0);

transform: scale(0);

/\* 不占据空间，无法点击 \*/

}

display:none隐藏产生reflow和repaint(回流与重绘)，而visibility:hidden没有这个影响前端性能的问题。

株连性：display:none就是“株连性”明显的声明：一旦父节点元素应用了display:none，父节点及其子孙节点元素全部不可见。visibility:hidden，则其子孙后代也都会全部不可见。如果子孙元素应用了visibility:visible。那么这个子孙元素又会刘谦般地显现出来。

### 回流（Reflow）是指布局引擎为frame计算图形的过程。

frame是一个矩形，拥有宽高和相对父容器的偏移。frame用来显示盒模型（content model）， 但一个content model可能会显示为多个frame，比如换行的文本每行都会显示为一个frame。

**重绘**（Repaint）发生在元素的可见性发生变化时，**比如背景色、前景色等**。 因此回流必然会引起重绘。

**回流何时发生：**

**当页面布局和几何属性改变时会回流**。下述情况会发生浏览器回流：

1、添加或者删除可见的DOM元素；

2、元素位置改变；

3、元素尺寸改变——边距、填充、边框、宽度和高度

4、内容改变——比如文本改变或者图片大小改变而引起的计算值宽度和高度改变,尤其是输入控件；

5、页面渲染初始化；

6、浏览器窗口尺寸改变——resize事件发生时；

7.CSS伪类激活

**如何减少回流、重绘**

减少回流、重绘其实就是需要减少对render tree的操作（合并多次多DOM和样式的修改），并减少对一些style信息的请求，尽量利用好浏览器的优化策略。具体方法有：

1. **直接改变className，如果动态改变样式，则使用cssText（考虑没有优化的浏览器）。**
2. **让要操作的元素进行”离线处理”，处理完后一起更新。**

a） 使用DocumentFragment进行缓存操作,引发一次回流和重绘；  
b) 使用display:none技术，只引发两次回流和重绘；  
c) 使用cloneNode(true or false) 和 replaceChild 技术，引发一次回流和重绘；

1. 不要经常访问会引起浏览器flush队列的属性，如果你确实要访问，利用缓存。
2. **// 别这样写，大哥**
3. **for(循环) {**
4. **el.style.left = el.offsetLeft + 5 + "px";**
5. **el.style.top = el.offsetTop + 5 + "px";**
6. **}**
8. **// 这样写好点**
9. **var left = el.offsetLeft,**
10. **top = el.offsetTop,**
11. **s = el.style;**
12. **for (循环) {**
13. **left += 10;**
14. **top += 10;**
15. **s.left = left + "px";**
16. **s.top = top + "px";**
17. **}**
18. **让元素脱离动画流，减少回流的Render Tree的规模**

### 182)clientHeight / scrollHeight / offsetHeight 等属性的区别图

网页(内容)可见区域宽：document.body.clientWidth

网页(内容)可见区域高：document.body.clientHeight 即页面浏览器中可以看到内容的这个区域的高度，一般是最后一个工具条以下到状态栏以上的这个区域，与页面内容无关。

网页可见区域宽：document.body.offsetWidth (包括边线的宽)

网页可见区域高：document.body.offsetHeight (包括边线的宽)

网页正文全文宽：document.body.scrollWidth

网页正文全文高：document.body.scrollHeight（总高）

IE、Opera 认为 scrollHeight 是网页内容实际高度，可以小于 clientHeight。

NS、 FF 认为 offsetHeight 和 scrollHeight 都是网页内容高度，只不过当网页内容高度小于等于 clientHeight 时，scrollHeight 的值是 clientHeight，而 offsetHeight 可以小于 clientHeight。

网页被卷去的高：document.body.scrollTop

网页被卷去的左：document.body.scrollLeft

网页正文部分上：window.screenTop

网页正文部分左：window.screenLeft

屏幕分辨率的高：window.screen.height

屏幕分辨率的宽：window.screen.width

屏幕可用工作区高度：window.screen.availHeight

屏幕可用工作区宽度：window.screen.availWidth

可见区域宽度： document.documentElement.clientWidth //这个作为可视高度更准确。  
  可见区域高度：document.documentElement.clientHeight

jQuery获取位置和尺寸**相关函数：  
$(document).height()** 获取整个页面的高度  
**$(window).height()** 获取当前也就是浏览器所能看到的页面的那部分的高度。这个大小在你缩放浏览器窗口大小时会改变，与document是不一样的  
**scrollTop()**    获取匹配元素相对滚动条顶部的偏移。  
**scrollLeft()**  获取匹配元素相对滚动条左侧的偏移。  
**scroll([[data],fn])** 当滚动条发生变化时触犯scroll事件

Event坐标：

chrome：

e.pageX——相对整个页面的坐标  
e.layerX——相对当前坐标系的border左上角开始的坐标  
e.offsetX——相对当前坐标系的border左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对可视区域的坐标

ff：

e.pageX——相对整个页面的坐标  
e.layerX——相对当前坐标系的border左上角开始的坐标  
e.offsetX——无  
e.clientX——相对可视区域的坐标  
e.x——无

opera：

e.pageX——相对整个页面的坐标  
e.layerX——无  
e.offsetX——相对当前坐标系的内容区域左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对可视区域的坐标

safari：（这个和chrome是一样的）

e.pageX——相对整个页面的坐标  
e.layerX——相对当前坐标系的border左上角开始的坐标  
e.offsetX——相对当前坐标系的border左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对可视区域的坐标

IE9：

e.pageX——相对整个页面的坐标  
e.layerX——相对当前坐标系的border左上角开始的坐标 + 滚动条滚过的距离（这个NB轰轰了····=。=）  
e.offsetX——相对当前坐标系的内容区域左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对当前坐标系的border左上角开始

IE8：

e.pageX——无  
e.layerX——无  
e.offsetX——相对当前坐标系的内容区域左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对当前坐标系的border左上角开始

IE7：

e.pageX——无  
e.layerX——无  
e.offsetX——相对当前坐标系的内容区域左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对当前坐标系的border左上角开始

IE6：

e.pageX——无  
e.layerX——无  
e.offsetX——相对当前坐标系的内容区域左上角开始的坐标  
e.clientX——相对可视区域的坐标  
e.x——相对当前坐标系的border左上角开始

### 183)获取body不需要，只须document.body。

### 184)ParseInt()实现：

如果 *string* 以 "0x" 开头，parseInt() 会把 *string* 的其余部分解析为十六进制的整数。如果 *string* 以 0 开头，那么 ECMAScript v3 允许 parseInt() 的一个实现把其后的字符解析为八进制或十六进制的数字。如果 *string* 以 1 ~ 9 的数字开头，parseInt() 将把它解析为十进制的整数。parseInt("a123a")==》NaN parseInt("123a3")==》123

### 185)!-[1,][全世界最短的IE判定](http://www.cnblogs.com/rubylouvre/archive/2010/01/28/1658006.html)(IE8及以下为true)

alert([1,2]);//相当于alert([1,2].toString()); --这在IE与非IE上都相同，都会弹出"1,2"

alert([1,]);//相当于alert([1,].toString());--在非IE的标准浏览器上，js引擎会自动删除最后的","，所以在IE上会弹出"1,"，而在非IE上会弹出"1"

alert(+[1,]);//根据上面的解释，这一句在IE上相当于alert(+"1,")，而在非IE上相当于alert(+"1")，正号的作用在于试图将字符串转换为数字，"1,"当然不是数字，而"1"可以转换为数字1，所以最终的结果：IE上会转换失败弹出"NaN"，而非IE浏览器上会弹出数字"1"

### 186)设备像素比devicePixelRatio

window.devicePixelRatio是设备上物理像素和设备独立像素(device-independent pixels (dips))的比例。  
公式表示就是：window.devicePixelRatio = 物理像素 / dips

**设备像素**：设备像素又称物理像素（physical pixel），设备能控制显示的最小单位，我们可以把这些像素看作成显示器上一个个的点。

**CSS像素**：CSS像素是Web编程的概念，独立于设备的用于逻辑上衡量像素的单位，也就是说我们在做网页时用到的CSS像素单位，是抽象的，而不是实际存在的。

**设备独立像素**：(也叫密度无关像素)，可以认为是计算机坐标系统中的一个点，这个点代表一个可以由程序使用并控制的虚拟像素(比如：CSS 像素,只是在android机中CSS 像素就不叫”CSS 像素”了而是叫”设备独立像素”)，然后由相关系统转换为物理像素

1像素边框：

使用viewport:

(function() {

var scale = 1.0;  
 if (window.devicePixelRatio === 2) {

scale \*= 0.5;

}

if (window.devicePixelRatio === 3) {

scale \*= 0.333333;

}

var text = '<meta name="viewport" content="initial-scale=' + scale + ', maximum-scale=' + scale +', minimum-scale=' + scale + ', width=device-width, user-scalable=no" />';

document.write(text);

})();

使用0.5px(ios8支持)：

**div**{

**border**:1px solid #000;

}

@**media** (-webkit-min-device-pixel-ratio: 2) {

**div**{

**border**:0.5px solid #000;

}

}

**transform:scale**

首先用伪类创建边框，然后通过媒体查询来适配

.border-bottom{

position: relative;

border-top: none !important;

}

.border-bottom::after {

content: " ";

position: absolute;

left: 0;

bottom: 0;

width: 100%;

height: 1px;

background-color: #e4e4e4;

-webkit-transform-origin: left bottom;

transform-origin: left bottom;

}

/\* 2倍屏 \*/

@media only screen and (-webkit-min-device-pixel-ratio: 2.0) {

.border-bottom::after {

-webkit-transform: scaleY(0.5);

transform: scaleY(0.5);

}

}

/\* 3倍屏 \*/

@media only screen and (-webkit-min-device-pixel-ratio: 3.0) {

.border-bottom::after {

-webkit-transform: scaleY(0.33);

transform: scaleY(0.33);

}

}

### ****187)图片懒加载****

一个页面有很多图片，而首屏出现的图片大概就一两张，那么我们还要一次性把所有图片都加载出来吗？显然这是愚蠢的，不仅影响页面渲染速度，还浪费带宽。

**思路：**

将页面里所有img属性src属性用data-xx代替，当页面滚动直至此图片出现在可视区域时，用js取到该图片的data-xx的值赋给src。

<script>

var num = document.getElementsByTagName('img').length;

var img = document.getElementsByTagName("img");

var n = 0; //存储图片加载到的位置，避免每次都从第一张图片开始遍历

lazyload(); //页面载入完毕加载可是区域内的图片

window.onscroll = lazyload;

function lazyload() { //监听页面滚动事件

var seeHeight = document.documentElement.clientHeight; //可见区域高度

var scrollTop = document.documentElement.scrollTop || document.body.scrollTop; //滚动条距离顶部高度

for (var i = n; i < num; i++) {

if (img[i].offsetTop < seeHeight + scrollTop) {

if (img[i].getAttribute("src") == "") {

img[i].src = img[i].getAttribute("data-src");

}

n = i + 1;

}

}

}

</script>

1. 事件代理在JS世界中一个非常有用也很有趣的功能。当我们需要对很多元素添加事件的时候，可以通过将事件添加到它们的父节点而将事件委托给父节点来触发处理函数。这主要得益于浏览器的事件冒泡机制。

当我们的鼠标移到Li上的时，为每个Li都添加一些类似onMouseOver或者onClick之类的事件监听。如果这个UL中的Li子元素会频繁地添加或者删除，我们就需要在每次添加Li的时候都调用这个addListeners Li方法来为每个Li节点添加事件处理函数。

**var** oBox = document.getElementById("box");  
oBox.onclick = **function** (ev) {  
 **var** ev = ev || window.event;//标准浏览器用ev.target，IE浏览器用event.srcElement  
 **var** target = ev.target || ev.srcElement;

//li还包含其它元素时，要把target变为li

**while**(target !== oBox ){  
 **if**(target.tagName.toLowerCase() == 'li'){  
 console.log(target);  
 **break**;  
 }  
 target = target.parentNode;  
 }  
 **if** (target.nodeName.toLocaleLowerCase() == 'input') {  
 **switch** (target.id) {  
 **case** 'add' :  
 alert('添加');  
 **break**;  
 **case** 'remove' :  
 alert('删除');  
 **break**;  
 **case** 'move' :  
 alert('移动');  
 **break**;  
 **case** 'select' :  
 alert('选择');  
 **break**;  
 }  
 }  
}

187)[**setTimeout延时0毫秒的作用**](http://www.cnblogs.com/facial/p/5797696.html)

实现javascript的异步；  
正常情况下javascript都是按照顺序执行的。但是我们可能让该语句后面的语句执行完再执行本身，这时就可以用到setTimeout延时0ms来实现了。

### pointer-events:none 元素永远不会成为鼠标事件的target。但是，当其后代元素的pointer-events属性指定其他值时，鼠标事件可以指向后代元素，在这种情况下，鼠标事件将在捕获或冒泡阶触发父元素的事件侦听器。

### ****189)reduce****函数：[x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)

### 190）一是HTML5推出的history API，由pushState()记录操作历史，监听popstate事件来进行视图切换；

二是使用hash值，通过监听hashchange事件来进行视图切换。

### 191）在 javascript 中数据类型可以分为两类：

* **原始数据类型值** primitive type，比如Undefined,Null,Boolean,Number,String。
* **引用类型值**，也就是对象类型 Object type,比如Object,Array,Function,Date等。

声明变量时不同的内存分配

* 原始值：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**。这是因为这些原始类型占据的空间是固定的，所以可将他们存储在较小的内存区域 – 栈中。这样存储便于迅速查寻变量的值。
* 引用值：存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存地址。这是因为：引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响。

### 192）1.const定义的变量不可以修改，而且必须初始化。

1. var定义的变量可以修改，如果不初始化会输出undefined，不会报错。

3.let是块级作用域，函数内部使用let定义后，对函数外部无影响。

### 193)js 数据类型判断

1. typeof 一元运算符

1. 'undefined' --- 这个值未定义；

2. 'boolean'    --- 这个值是布尔值；

3. 'string'        --- 这个值是字符串；

4. 'number'     --- 这个值是数值；

5. 'object'       --- 这个值是对象或null；

6. 'function'    --- 这个值是函数。

问题：arry,null都判断为object

1. instanceof

instanceof运算符用来判断一个构造函数的prototype属性所指向的对象是否存在另外一个要检测对象的原型链上

obj instanceof Object;

instanceof的运算规则：左表达式的.\_\_proto\_\_ === 右表达式.prototype 如果不等 左表达式的.\_\_proto\_\_ = 左表达式的.\_\_proto\_\_.\_\_proto\_\_

问题：Function instanceof Object; //true String instanceof String;//false

1. Object.prototype.toString

在[ES3](http://bclary.com/2004/11/07/)中,Object.prototype.toString方法的规范如下:

Object.prototype.toString()

在toString方法被调用时,会执行下面的操作步骤:

1. 获取this对象的[[Class]]属性的值.

2. 计算出三个字符串"[object ", 第一步的操作结果Result(1), 以及 "]"连接后的新字符串.

3. 返回第二步的操作结果Result(2).

[[Class]]是一个内部属性,所有的对象(原生对象和宿主对象)都拥有该属性.在规范中,

原生对象的[[class]]内部属性的值一共有10种.分别是:"Array", "Boolean", "Date", "Error", "Function", "Math", "Number", "Object", "RegExp", "String".

[ES6](http://people.mozilla.org/~jorendorff/es6-draft.html),**[[class]]内部属性没有了**,取而代之的是另外一个内部属性[[NativeBrand]].

1. **Object.prototype.toString ( )**

在toString方法被调用时,会执行下面的操作步骤:

1. 如果**this**的值为**undefined**,则返回"[object Undefined]".
2. 如果**this**的值为**null**,则返回"[object Null]".
3. 让O成为调用ToObject(**this)**的结果.
4. 如果O有[[NativeBrand]]内部属性,让tag成为表29中对应的值.
5. 否则
   1. 让hasTag成为调用O的[[HasProperty]]内部方法后的结果,参数为@@toStringTag.
   2. 如果hasTag为**false**,则让tag为"Object".
   3. 否则,
      1. 让tag成为调用O的[[Get]]内部方法后的结果,参数为@@toStringTag.
      2. 如果tag是一个abrupt completion,则让tag成为NormalCompletion("???").
      3. 让tag成为tag.[[value]].
      4. 如果Type(tag)不是字符串,则让tag成为"???".
      5. 如果tag的值为"Arguments", "Array", "Boolean", "Date", "Error", "Function", "JSON", "Math", "Number", "Object", "RegExp",或者"String"中的任一个,则让tag成为字符串"~"和tag当前的值连接后的结果.
6. 返回三个字符串"[object ", tag, and "]"连接后的新字符串.

新增类型：

Object.prototype.toString.call(Map()) //"[object Map]"

Object.prototype.toString.call(Set())) //"[object Set]"

### 194)实现继承的extend

function extend(subClass,superClass){

    var Func = function(){} ;

    Func.prototype = superClass.prototype ;

    subClass.prototype = new Func() ;

    subClass.prototype.constructor = subClass ;

} ;

实现对象拓展的extend:

**function** extend(deep,target,options){  
 **for**(**var** v **in** options){  
 **var** temp = options[v];  
 **if**(deep&&temp **instanceof** Array){  
 target[v] = extend(deep,[],temp);  
 }**else if**(deep&&temp **instanceof** Object){  
 target[v] = extend(deep,{},temp);  
 }**else**{  
 target[v] = options[v];  
 }  
 }  
 **return** target;  
}

几种深拷贝的实现方式：

JSON.stringify()将对象转为字符串，再赋值，再用JSON.parse()转回对象。

Var copy = [].slice(0);这只对第一层做了拷贝，当元素为对象时没有深拷贝。

jQuery.extend( [deep ], target, object1 [, objectN ] )

默认：浅拷贝，加true参数为深拷贝

### 195)HTML 自定义元素教程

[HTML5 标准](https://www.w3.org/TR/html5/infrastructure.html#extensibility-0)的

"User agents must treat elements and attributes that they do not understand as semantically neutral; leaving them in the DOM (for DOM processors), and styling them according to CSS (for CSS processors), but not inferring any meaning from them."

浏览器必须将自定义元素保留在 DOM 之中，但不会任何语义。除此之外，自定义元素与标准元素都一致。

HTML5 标准规定了自定义元素是合法的。然后，W3C 就为自定义元素制定了一个单独的 [Custom Elements 标准](https://w3c.github.io/webcomponents/spec/custom/#custom-elements)。

它与其他三个标准放在一起---- HTML Imports，HTML Template、Shadow DOM----统称为 [Web Components](https://www.w3.org/standards/techs/components#w3c_all) 规范。目前，这个规范只有 Chrome 浏览器[支持](http://caniuse.com/#feat=custom-elements)。

Custom Elements 标准对自定义元素的名字做了[限制](https://developers.google.com/web/fundamentals/getting-started/primers/customelements)。

"自定义**元素的名字必须包含一**个破折号（-）所以<x-tags>、<my-element>和<my-awesome-app>都是正确的名字，而<tabs>和<foo\_bar>是不正确的。这样的限制使得 HTML 解析器可以分辨那些是标准元素，哪些是自定义元素。"

### 196)HTML Imports（HTML导入）

<link rel="import" href="component.html" >

component.html：

<link rel="stylesheet" href="css/style.css">

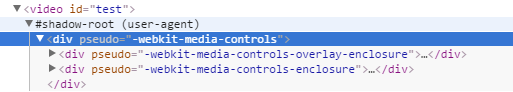
<script src="js/script.js"></script>

HTML Template：（html新标签 HTML模版）

Shadow DOM：

shadow-dom，直译的话就是 影子dom

<video id='test'></video>



虽然我们创建的是一个空标签，但是在这个空标签内部，存在一个 shadow-dom ，点开 shadow-dom 可以看到内有乾坤，大有内容。其实这内部的具体内容，就是 <video> 的具体实现。

1. shadow **host**

对于一个内部有 shadow-dom 的元素而言，它必然需要一个宿主元素，对于上面的例子而言， <video> 标签，就是 shadow-dom 的宿主元素。

1. shadow**-root**

通过 createShadowRoot(下文会提及) 返回的文档片段被称为 shadow-root 。它和它的后代元素，都将对用户隐藏，但是它们是实际存在的，在 chrome 中，我们可以通常审查元素去查看它们的具体 DOM 实现。

Shadow-dom 是游离在 DOM 树之外的节点树，但是他的创建基于普通 DOM 元素（非 document），并且创建后的 Shadow-dom 节点可以从界面上直观的看到。更重要的是，Shadow-dom 具有良好的密封性。

这是浏览器提供的一种“封装”功能，提供了一种强大的技术去隐藏一些实现细节。什么意思呢？以 [w3c](http://www.w3school.com.cn/tiy/t.asp?f=html5_video) 上的一个 <video> 例子为例，我们仅仅是填写了一个空白的标签，再加上 src 属性里填上视频地址，就可以播放视频了。

1. 可以**使用伪元素控制 shadow-dom 样式**

video{  
 display:block;  
 width:300px;  
 margin:20px auto;  
}  
  
**video::-webkit-media-controls-panel**{  
 display:flex!important;  
 background-color: deeppink;  
}

1. **使用 Javascript 创建一个 shadow-dom 元素**

HTMLElement.prototype.createShadowRoot =   
 HTMLElement.prototype.createShadowRoot ||  
 HTMLElement.prototype.webkitCreateShadowRoot ||  
 function() {};  
  
var tmpl = document.querySelector('#content');  
var host = document.querySelector('#shadow-dom-host');  
// Create a Shadow Root  
**var root = host.createShadowRoot();**// Append elements to the Shadow Root  
root.appendChild(document.importNode(tmpl.content, true));

### 197)dns协议

基于udp,当报文大于512字节使用tcp.端口53

本地使用递归查询，所谓递归查询就是：如果主机所询问的本地域名服务器不知道被查询的域名的IP地址，那么本地域名服务器就以DNS客户的身份，向其它根域名服务器继续发出查询请求报文(即替主机继续查询)，而不是让主机自己进行下一步查询。

DNS服务器间迭代查询，就是要么给出所要查询的IP地址，要么告诉本地服务器：“你下一步应当向哪一个域名服务器进行查询”。然后让本地服务器进行后续的查询。



DNS的记录类型：

常见的DNS记录类型如下。

（1） A：地址记录（Address），返回域名指向的IP地址。

（2） NS：域名服务器记录（Name Server），返回保存下一级域名信息的服务器地址。该记录只能设置为域名，不能设置为IP地址。

（3）MX：邮件记录（Mail eXchange），返回接收电子邮件的服务器地址。

（4）CNAME：规范名称记录（Canonical Name），返回另一个域名，即当前查询的域名是另一个域名的跳转，详见下文。

（5）PTR：逆向查询记录（Pointer Record），只用于从IP地址查询域名，详见下文。

Dns工具

dig命令：

dig [www.baidu.com](http://www.baidu.com) +trace参数可以显示DNS的整个分级查询过程。

**host 命令**

**nslookup 命令**

**whois 命令**

在宽带路由器上可以自动从[WAN接口](https://www.baidu.com/s?wd=WAN%E6%8E%A5%E5%8F%A3&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y1rHnvPH6YmHf4P1f1mHbk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWDkPW63PHnsPj0kPH0zn16vPs)获得运营商的DNS的IP地址,也可以配置静态DNS。在客户端设置自动获取dns,就会从路由获取dns,给黑客进行dns劫持的机会。

**注意：DNS解析比较耗时，要减少DNS解析次数。**

### ****198)页面解析****

1. 解析HTML生成DOM树，
2. 解析CSS文件生成CSSOM树，并解析Javascript代码
3. CSS和DOM组合形成渲染树，

根据CSS选择符的特指度来判定优先级；一条样式声明的特指度由以下四个部分决定：S-I-C-E

1. 进行布局，在浏览器中绘制渲染树中节点的属性(位置，宽度，大小等)

布局是一个递归过程，从跟呈现节点开始，递归遍历子节点，计算集合几何信息。

1. 绘制元素，绘制各个节点的可视属性(颜色背景等，此时可能会形成多个图层)
2. 合并图层，将页面呈现给用户面

### 199)[HTTP的长连接和短连接](http://www.cnblogs.com/cswuyg/p/3653263.html)

 HTTP1.1规定了默认保持长连接（HTTP persistent connection ，也有翻译为持久连接），数据传输完成了保持TCP连接不断开（不发RST包、不四次握手），等待在同域名下继续用这个通道传输数据；相反的就是短连接。

HTTP首部的Connection: Keep-alive，客户端的长连接不可能无限期的拿着，会有一个超时时间，如Keep-Alive: timeout=20。

长连接的数据传输完成识别

1是判断传输数据是否达到了Content-Length指示的大小；2动态生成的文件没有Content-Length，它是分块传输（chunked），这时候就要根据chunked编码（HTTP回应是无法确定消息大小的，比如大文件的下载，或者后台需要复杂的逻辑才能全部处理页面的请求，这时用需要实时生成消息长度，服务器一般使用chunked编码。编码使用若干个Chunk组成，由一个标明长度为0的chunk结束，每个Chunk有两部分组成，第一部分是该Chunk的长度和长度单位（一般不写），第二部分就是指定长度的内容，每个部分用CRLF（Carriage-Return Line-Feed 回车换行）隔开。在最后一个长度为0的Chunk中的内容是称为footer的内容，是一些没有写的头部内容。）来判断，chunked编码的数据在最后有一个空chunked块，表明本次传输数据结束。

HTTP 流水线技术，**在一个TCP连接内，多个HTTP请求可以并行，下一个HTTP请求在上一个HTTP请求的应答完成之前就发起。**这个技术必须要求客户端和服务器端都能支持，目前有部分浏览器完全支持，而服务端的支持仅需要：按HTTP请求顺序正确返回。

### 200)cookie free

很多建议将站点的静态文件（如图片、[**js**](http://lib.csdn.net/base/javascript)、css 等）放在一个专门的域名下访问，由于该域名与主站域名不同，所以浏览器就不会把主域名下的 Cookie 传给该域，减少了网络开销，一定程度提高了页面加载速度，特别是细碎静态文件特别多的情况下效果显著。

### 201)反向ajax

反向Ajax的基本概念是客户端不必从服务器获取信息，服务器会把相关信息直接推送到客户端。这样做的目的是解决Ajax传统Web模型所带来的一个限制：实时信息很难从技术上解决。

**1 轮询**

客户端定时轮询服务器，看是否存在更新，并且显示服务器传回的当前信息。在网页上，使用一些简单的JavaScript代码以持续地更新页面，就可以实现相同的事情。确实，能够从表面实现所谓的推送，这就是轮询技术。

2 **Comet（服务器推）**

这是一种基于HTTP长连接的服务器推动方式。客户端向服务器发送请求后，服务器将数据通过response发送给客户端，但并不会将此response关闭，而是一直通过response将最新的数据发送给客户端浏览器，直到客户端浏览器关闭。

1. **Piggyback**

### ****202)setTimeout何时执行****

在现有浏览器环境中，Javascript执行引擎是单线程的，主线程的语句和方法，会阻塞定时任务的运行，在Javascript执行引擎之外，存在一个任务队列，当在代码中调用setTimeout()方法时，注册的延时方法会挂到浏览器内核其他模块处理，当延时方法到达触发条件，即到达设置的延时时间时，该模块再将要执行的方法添加至该模块的任务队列中。这一过程与执行引擎主线程独立，执行引擎在主线程方法执行完毕，到达空闲状态时，才会从该模块的任务队列中顺序提取任务来执行，这期间的时间，可能大于注册任务时设置的延时时间；

浏览器在空闲状态下，会不断的尝试从模块的任务队列中提取任务，这称为事件循环模型；

### 203)requestAnimationFrame

浏览器可以优化并行的动画动作，更合理的重新排列动作序列，并把能够合并的动作放在一个渲染周期内完成，从而呈现出更流畅的动画效果。

requestAnimationFrame的原理与使用方法与setTimeout/setInterval类似   
它是以递归的形式来实现动画   
既然它是专门用来作Web动画的，它就一定有它自己的优势

使用setTimeout/setInterval制作动画有以下缺点

* 不能保证ms的准确性（JavaScript单线程，可能造成阻塞）
* 没有优化调用动画的循环机制
* 没有考虑到绘制动画的最佳时机（只是简单的按一定时间调用循环）

相比之下，requestAnimationFrame有以下优点

* 动画更加流畅，经由浏览器优化（页面刷新前执行一次）
* 窗口未激活时，动画暂停，有效节省CPU开销
* 省电，对移动端很友好

polyfill （Polyfill你可以理解为“腻子”，就是装修的时候，可以把缺损的地方填充抹平。即兼容写法）写法:

**if** (!Date.now)  
 Date.now = **function**() { **return new** Date().getTime(); };  
(**function**() {  
 'use strict';  
 **var** vendors = ['webkit', 'moz'];  
 **for** (**var** i = 0; i < vendors.length && !window.requestAnimationFrame ; ++i) {  
 **var** vp = vendors[i];  
 window.requestAnimationFrame = window[vp+'RequestAnimationFrame'];  
 window.cancelAnimationFrame = (window[vp+'CancelAnimationFrame']  
 || window[vp+'CancelRequestAnimationFrame']);  
 }  
 **if** (/iP(ad|hone|od).\*OS 6/.test(window.navigator.userAgent) // iOS6 is buggy  
 || !window.requestAnimationFrame || !window.cancelAnimationFrame) {  
 **var** lastTime = 0;  
 window.requestAnimationFrame = **function**(callback) {  
 **var** now = Date.now();  
 **var** nextTime = Math.max(lastTime + 16, now);  
 **return** setTimeout(**function**() { callback(lastTime = nextTime); },  
 nextTime - now);  
 };  
 window.cancelAnimationFrame = clearTimeout;  
 }  
}());

### 204)margin重叠

1. 水平边距永远不会重合

2、在规范文档中，2个或以上的块级盒模型相邻的垂直margin会重叠。最终的margin值计算方法如下：  
a、全部都为正值，取最大者；  
b、不全是正值，则都取绝对值，然后用正值减去最大值；  
c、没有正值，则都取绝对值，然后用0减去最大值。  
注意：相邻的盒模型可能由DOM元素动态产生并没有相邻或继承关系。  
3、相邻的盒模型中，如果其中的一个是浮动的（float），垂直margin不会重叠，并且浮动的盒模型和它的子元素之间也是这样。  
4、设置了overflow属性的元素和它的子元素之间的margin不被重叠（overflow取值为visible除外）。  
5、设置了绝对定位（position:absolute）的盒模型，垂直margin不会被重叠，并且和他们的子元素之间也是一样。  
6、设置了display:inline-block的元素，垂直margin不会重叠，甚至和他们的子元素之间也是一样。  
7、如果一个盒模型的上下margin相邻，这时它的margin可能重叠覆盖（collapse through）它。在这种情况下，元素的位置（position）取决于它的相邻元素的margin是否重叠。  
a、如果元素的margin和它的父元素的margin-top重叠在一起，盒模型border-top的边界定义和它的父元素相同。  
b、另外，任意元素的父元素不参与margin的重叠，或者说只有父元素的margin-bottom是参与计算的。如果元素的border-top非零，那么元素的border-top边界位置和原来一样。  
一个应用了清除操作的元素的margin-top绝不会和它的块级父元素的margin-bottom重叠。  
注意，那些已经被重叠覆盖的元素的位置对其他已经重叠的元素的位置没有任何影响；只有在对这些元素的子元素定位时，border-top边界位置才是必需的。

8、根元素的垂直margin不会被重叠。

外边距(margin)重叠示例  
外边距重叠是指两个垂直相邻的块级元素，当上下两个边距相遇时，起外边距会产生重叠现象，且重叠后的外边距，等于其中较大者。

另一个重叠现象是当一个元素包含在另一个元素之中时，子元素与父元素之间也会产生重叠现象，重叠后的外边距，等于其中最大者：  
两个元素下上叠加  
同理，如果一个无内容的空元素，其自身上下边距也会产生重叠。  
内外元素同向叠加

外边距的重叠只产生在普通流文档的上下外边距之间，这个看起来有点奇怪的规则，其实有其现实意义。设想，当我们上下排列一系列规则的块级元素（如段落P）时，那么块元素之间因为外边距重叠的存在，段落之间就不会产生双倍的距离。

防止外边距重叠解决方案：  
虽然外边距的重叠有其一定的意义，但有时候我们在设计上却不想让元素之间产生重叠，那么可以有如下几个建议可供参考：

1. 外层元素padding代替
2. 内层元素透明边框 border:1px solid transparent;
3. 内层元素绝对定位 postion:absolute:
4. 外层元素 overflow:hidden;
5. 内层元素 加float:left;或display:inline-block;
6. 内层元素padding:1px;

以上建议可根据实际情况来采取。

补充：1.span是行内元素，margin只有左右，不能设置上下的margin.

1. 行内元素的padding-top，padding-bottom和margin-top,margin-bottom是真的是无效；不过要注意一点，对于竖直方向的内边距该行内元素的内容范围是增大了，不过只是表象，对周围元素无任何影响。
2. Margin设置百分数是相对于父元素的 width 计算的。Pading:规定基于父元素的宽度的百分比的内边距。

4.margin折叠常规认知：

(1)margin折叠只发生在块级元素上；

(2)浮动元素的margin不与任何margin发生折叠；

(3)设置了属性[overflow](http://www.css88.com/book/css/properties/layout/overflow.htm)且值不为visible的块级元素，将不与它的子元素发生margin折叠；

(4)绝对定位元素的margin不与任何margin发生折叠；

(5)根元素的margin不与其它任何margin发生折叠；

### 205)数据结构及其选择

1.如果对数据集合的主要操作是遍历数据，就使用链表

2.如果代码符合生产者/消费者,则使用队列

3.如果需要映射一个uid到一个对象，就使用映射

4.如果需要存储大量数据，并快速检索，那么红黑树最好（Linux实现的红黑树为rbtree(lib/rbtree.c)）;

### 206)drag api

* 在拖动目标上触发事件 (源元素):
* [ondragstart](http://www.runoob.com/jsref/event-ondragstart.html) - 用户开始拖动元素时触发
* [ondrag](http://www.runoob.com/jsref/event-ondrag.html) - 元素正在拖动时触发
* ondragend - 用户完成元素拖动后触发
* 释放目标时触发的事件:
* [ondragenter](http://www.runoob.com/jsref/event-ondragenter.html) - 当被鼠标拖动的对象进入其容器范围内时触发此事件
* [ondragover](http://www.runoob.com/jsref/event-ondragover.html) - 当某被拖动的对象在另一对象容器范围内拖动时触发此事件
* [ondragleave](http://www.runoob.com/jsref/event-ondragleave.html) - 当被鼠标拖动的对象离开其容器范围内时触发此事件
* [ondrop](http://www.runoob.com/jsref/event-ondrop.html) - 在一个拖动过程中，释放鼠标键时触发此事件
* 注意：dragover要添加event.preventDefault();不然drop无效。

**var** dragged;  
  
document.addEventListener("dragstart",**function**(event){  
 console.log("start:",event);  
 dragged = event.target;  
 // 被拖拉节点的背景色变透明  
 event.target.style.opacity = 0.5;  
 // 兼容Firefox  
 event.dataTransfer.setData('text/plain', 'anything');  
});  
  
// 当拖完p元素输出一些文本元素和重置透明度  
document.addEventListener("dragend", **function**(event) {  
 console.log("end:",event);  
 //document.getElementById("demo").innerHTML = "完成 p 元素的拖动";  
 event.target.style.opacity = 1;  
 dragged.style.opacity = 1;  
});  
// 当p元素完成拖动进入droptarget,改变div的边框样式  
document.addEventListener("dragenter", **function**(event) {  
 console.log("dragenter:",event);  
 //event.target.style.border = "3px dotted red";  
});  
  
// 默认情况下,数据/元素不能在其他元素中被拖放。对于drop我们必须防止元素的默认处理  
document.addEventListener("dragover", **function**(event) {  
 //console.log("dragover:",event);  
 event.preventDefault();  
});  
  
// 当可拖放的p元素离开droptarget，重置div的边框样式  
document.addEventListener("dragleave", **function**(event) {  
 console.log("dragleave:",event);  
 event.target.style.border = "";  
});  
  
document.addEventListener('drop', **function**( event ) {  
 // 防止事件默认行为（比如某些Elment节点上可以打开链接）  
 event.preventDefault();  
 console.log("drop:",event);  
 //console.log("target.parentNode",event.target.parentNode);  
 event.target.style.opacity = 1;  
 dragged.style.opacity = 1;  
 **if**(event.target.parentNode.className == "drag"){  
 **var** temp = event.target.parentNode.innerHTML;  
 event.target.parentNode.innerHTML = dragged.parentNode.innerHTML;  
 dragged.parentNode.innerHTML =temp;  
 }  
  
 //event.target.style.opacity = "1";  
}, **false**);

<div draggable="false">  
 <div class="drag">  
 <div id="one" style="background: red;" draggable="true">1</div>  
 </div>  
 <div class="drag">  
 <div id="two" style="background: green;" draggable="true">2</div>  
 </div>  
 <div class="drag">  
 <div id="three" style="background: cornflowerblue" draggable="true">3</div>  
 </div>  
 <div class="drag">  
 <div id="four" style="background: yellow" draggable="true">4</div>  
 </div>  
 <div class="drag">  
 <div id="five" style="background: bisque" draggable="true">5</div>  
 </div>  
</div>

### 207)jQuery.clone()

clone()函数用于**克隆当前匹配元素集合的一个副本**，并以jQuery对象的形式返回。你也可以简单地理解为：克隆当前jQuery对象。

jQueryObject.clone( withDataAndEvents [, deepWithDataAndEvents ] )

|  |  |
| --- | --- |
| **参数** | **描述** |
| withDataAndEvents | 可选/Boolean类型是否同时复制元素的附加数据和绑定事件，默认为false。 |
| deepWithDataAndEvents | 可选/Boolean类型是否同时复制元素的所有子元素的附加数据和绑定事件，默认值即为参数withDataAndEvents的值。 |

### 208)insertBefore() 方法在您指定的已有子节点之前插入新的子节点。

parentElement.insertBefore.(newElement,targetElement)

parentElement为父元素，在调用insertBefore方法后，将新元素插入到目标元素之前。

newElement必需,需要插入的节点对象。targetElement,可选,在其之前插入新节点的子节点。如果未规定，则 insertBefore 方法会在结尾插入 newElement。

### 209)排序算法思想：



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 各种排序算法 | | | | | | |
| 类别 | 排序方法 | 时间复杂度 | | | 空间复制度 | 稳定性 |
| 平均 | 最好 | 最坏 | 辅助存储 |
| 插入排序 | 直接插入 | O() | O(n) | O() | O(1) | 稳定 |
| Shell排序 | O() | O(n) | O() | O(1) | 不稳定 |
| 选择排序 | 简单选择 | O() | O() | O() | O(1) | 不稳定 |
| 堆排序 | O() | O() | O() | O(1) | 不稳定 |
| 交换排序 | 冒泡排序 | O() | O(n) | O() | O(1) | 稳定 |
| 快速排序 | O() | O() | O() | O()-O(n) | 不稳定 |
| 归并排序 | | O() | O() | O() | O(n) | 稳定 |
| 基数排序 | | O(n) | 不一定 | O(n) | O(rd+n) | 稳定 |
| 注：r代表关键字的基数，d代表长度，n代表关键字个数。-左为最好情况，右为最差情况。 | | | | | | |

#### 插入排序

##### 直接插入排序

特点：稳定排序，时间复杂度O(N\*N)，空间复杂度O(1)

思想：每次将一个待排序的数据元素，插入到前面已经排好序的数列中的适当位置，使数列依然有序；直到待排序数据元素全部插入完。

场景：当数据已经基本有序时，采用插入排序可以明显减少数据交换和数据移动次数，进而提升排序效率。

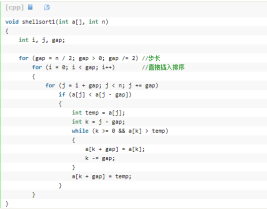


##### 希尔排序

特点：复杂度可以为O(n1.3)，空间复杂度：O(1)  ，稳定性：不稳定

思想：希尔排序的实质就是分组插入排序，目的让数据基本有序。首选增量为n/2,以此递推，每次增量为原先的1/2，直到增量为1；当增量为1，则变成直接插入排序。

场景：无法准确选择增量，所以此算法少用。



#### 交换排序

##### 冒泡排序

特点：时间复杂度为O(),空间复杂度为O(1),一种稳定的排序算法

思想：从第一个元素开始，对数组中两两相邻的元素比较，将值较小的元素放在前面，值较大的元素放在后面，一轮比较完毕，一个最大的数沉底成为数组中的最后一个元素，一些较小的数如同气泡一样上浮一个位置。

改进：设置记录发生过交换的位子，对于后面没有交换过的，表示已近有序不用参加排序。

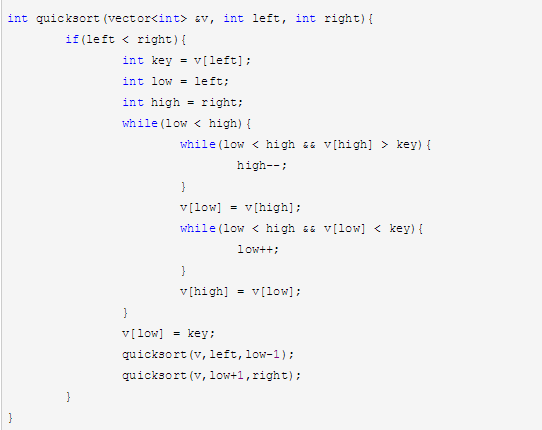
场景：时间复杂度高，一般不用。

##### 快速排序

特点：最坏情况是每次划分选取的基准都是当前无序区中关键字最小(或最大)的记录，划分的结果是基准左边的子区间为空(或右边的子区间为空)，而划分所得的另一个非空的子区间中记录数目，仅仅比划分前的无序区中记录个数减少一个。时间复杂度最坏为O(n\*n)，评价为：**O(nlogn)**空间复杂度**最优的情况为：O(logn)；最差的情况下为：O( n ) ；退化为冒泡排序的情况。为不稳定排序。**

思想：快速排序采用的思想是分治思想。通过一躺排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

场景：是基于关键字比较的内部排序算法中速度最快者，快速排序亦因此而得名。它的平均时间复杂度为O(nlgn)。如果序列基本有序则退化为冒泡排序。当n较大，且序列基本无序可以选择此算法。



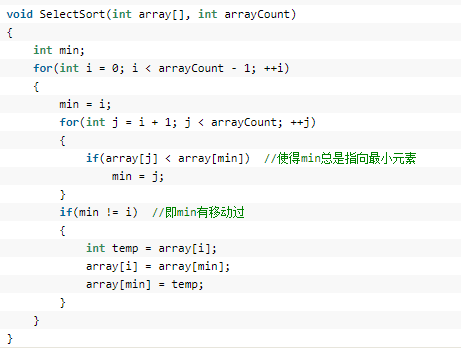
#### 选择排序

##### 简单选择排序

特点：时间复杂度,空间复杂度O(1),不稳定

思想：每趟从待排序的记录中选出关键字最小的记录，顺序放在已排序的记录序列末尾，直到全部排序结束为止。

场景：时间复杂度高，而且不稳定，没人用。



##### 堆排序

特点：最坏情况时间复杂度为O(N\*logN)，[堆排序中建堆过程的时间复杂度O(n)](http://blog.csdn.net/anonymalias/article/details/8807895)，空间复杂度O(1),不稳定

思想：堆是一种特殊的树形[**数据结构**](http://lib.csdn.net/base/datastructure)，其每个节点都有一个值，通常提到的堆都是指一颗完全二叉树，根结点的值小于（或大于）两个子节点的值，同时，根节点的两个子树也分别是一个堆。

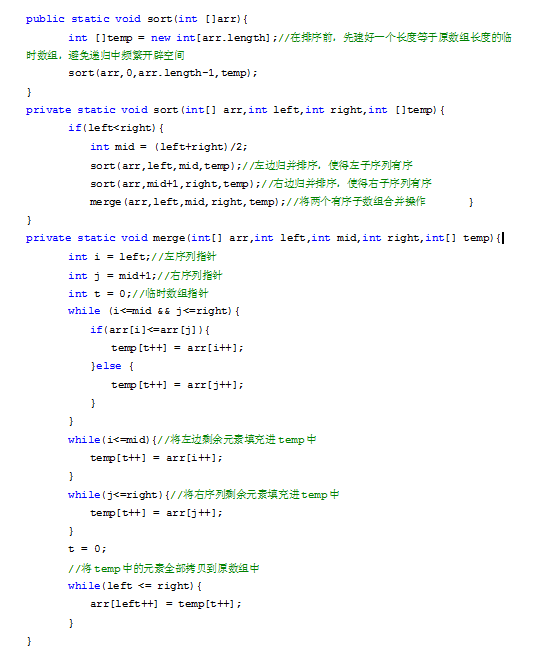
场景：堆排序适合，处理大数据（eg:找出一千万个数中最小的前一百个数,建立一百个节点的大顶堆，首先将前一百个数放入堆中，之后每放入一个数就删除一个堆顶元素，最后剩下的就是最小的一百个数。）

#### 归并排序（外部排序）

特点：每次合并操作的平均时间复杂度为O(n)，而完全二叉树的深度为|log2n|，归并排序的最好，最坏，平均时间复杂度均为O(nlogn)，**空间复杂度为: O(n)，是稳定的哟！**

思想：建立在归并操作上的一种有效的排序[**算法**](http://lib.csdn.net/base/datastructure)。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序将两个或多个已排序的表合并成一个表。

场景：归并排序虽然比较稳定，在时间上也是非常有效的（最差时间复杂度和最优时间复杂度都为 O(nlogn)  )，但是这种算法很消耗空间，一般来说在内部排序不会用这种方法，而是用快速排序；外部排序才会考虑到使用这种方法。java中Arrays.sort()采用了一种名为TimSort的排序算法，就是归并排序的优化版本。



#### 基数排序

特点：**平均情况**O(d\*(n+r))，**空间复杂度**O(n+r)，稳定

思想：利用多关键字

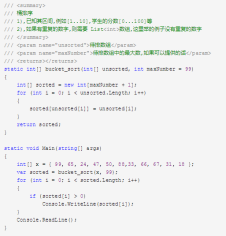
场景：对与关键字长度不太长的整死或字符排序，基数排序是比较优秀的。

1. 桶排序

特点：桶排序是稳定的，平均时间复杂度为线性的O(N+C)，空间复杂度 为O(N+M)

思想：桶排序是在已知数据的范围的条件下，创建若干个桶，根据相应的比较规则将待排数据落入各个对应的桶中，最后扫描桶来实现排序。

场景：使用条件较严格，但速度很快，比快排还快。



#### 计数排序

特点：它的优势在于在对一定范围内的整数排序时，它的复杂度为Ο(n+k)，这是一种牺牲空间换取时间的做法

思想：对于给定的输入序列中的每一个元素x，确定该序列中值小于x的元素的个数（此处并非比较各元素的大小，而是通过对元素值的计数和计数值的累加来确定）。一旦有了这个信息，就可以将x直接存放到最终的输出序列的正确位置上。

场景：数据范围小的时候。

### 210)查找算法的思想

#### 1)顺序表的查找

1. 顺序查找

特点：时间复杂度为O(n)，只需要一个辅助存储空间。

思想：按顺序比较每个元素，直到找到关键字为止。

场景：对表没有要求，简单适应面广

1. 二分查找

特点：时间复杂度为o(logN)

思想：**顺序表中的记录按关键字有序,取中间记录作为比较对象，若给定值与中间记录的关键字相等，则查找成功；若给定值小于中间记录的关键字，则在中间记录左半区继续查找；否则，在右半区查找。不断重复，知道查找成功或者查找失败为止。**

场景：**折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。**

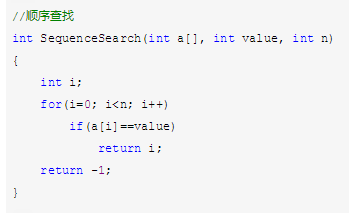
#### 2)索引表的查找

1. 顺序查找

特点：

思想：先用顺序查找找到待查数据在哪一块。再在已近确定的块中进行顺序查找。

场景：

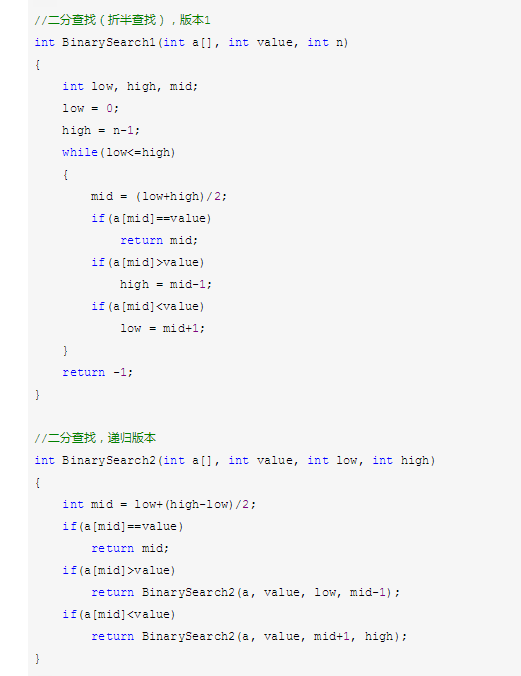


1. 二分查找

特点：最坏情况o(n/b),平局情况o(n/2b)

思想：讲表分为若干块，各块内部不一定有序。但块是有序的，抽取最大值及起始位置建立索引表。在块中用二分查找，在块内用顺序查找。

场景：



#### 3)散列表的查找

思想：根据键值方式(Key value)进行查找，通过散列函数，定位数据元素.

1. 散列函数

关键字经过散列函数得到一个随机地址。

①直接散列

取关键字本身或关键字的某个线性函数值作为散列地址，H(key) = key;

②数字分析法

如果事先知道关键字集合，并且每个关键字的位数比哈希表的地址码位数多时，可以从关键字中选出分布较均匀的若干位，构成哈希地址。例如，有80个记录，关键字为8位十进制整数d1d2d3…d7d8，如哈希表长取100，则哈希表的地址空间为：00~99。假设经过分析，各关键字中 d4和d7的取值分布较均匀，则哈希函数为：h(key)=h(d1d2d3…d7d8)=d4d7。例如，h(81346532)=43，h(81301367)=06。

③平均取中法

取关键字平方后的中间几位为哈希地址。

④折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为哈希地址，这方法称为折叠法。

⑤除留取余法

取关键字被某个不大于哈希表表长m的数p除后所得余数为哈希地址。

⑥随机数法

选择一个随机函数，取关键字的随机函数值为它的哈希地址，H(key)=random(key），其中random为随机函数。

1. 冲突处理

①开放地址法

当冲突发生时，使用某种探查(亦称探测)技术在散列表中形成一个探查(测)序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者碰到一个开放的地址 (即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。查找时探查到开放的地址则表明表中无待查的关键字，即查找失败。

又分为1.线性探测：顺序查看下一个单元 2.二次探测：再表的左右进行跳跃式探测 3.伪随机探测：实现一个伪随机数发生器。

②再散列法

同时构造多个不同的哈希函数。发生冲突时再用另一个哈希函数计算。

③链地址法

将互为同义词的结点链成一个单链表，而将此链表的头指针放在散列表[0..m-1]中。

④公共溢出区法

将哈希表分为基本表和溢出表，凡是发生冲突的元素，一律填入溢出表。

性能：填装因子=

#### 4)二叉树的查找

二叉查找树是先对待查找的数据进行生成树，确保树的左分支的值小于右分支的值，然后在就行和每个节点的父节点比较大小，查找最适合的范围。 这个算法的查找效率很高，但是如果使用这种查找方法要首先创建树。

二叉查找树（BinarySearch Tree，也叫二叉搜索树，或称二叉排序树Binary Sort Tree）或者是一棵空树，或者是具有下列性质的二叉树：

1）若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

2）若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

3）任意节点的左、右子树也分别为二叉查找树。

二叉查找树性质： 对二叉查找树进行中序遍历，即可得到有序的数列。

时间复杂度：O(logn)

### 211)树的种类：

#### [1. 二叉树](http://www.cnblogs.com/maybe2030/p/4732377.html#_label0)

二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)



**注：**完全二叉树是效率很高的数据结构，堆是一种完全二叉树或者近似完全二叉树，所以效率极高，像十分常用的排序算法、Dijkstra算法、Prim算法等都要用堆才能优化，二叉排序树的效率也要借助平衡性来提高，而平衡性基于完全二叉树。

#### [2. 二叉查找树](http://www.cnblogs.com/maybe2030/p/4732377.html#_label1)

又称为是二叉排序树（Binary Sort Tree）或二叉搜索树。二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

　　1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；

　　2) 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；

　　3) 左、右子树也分别为二叉排序树；

　　4) 没有键值相等的节点。

**二叉查找树的性质：对二叉查找树进行中序遍历，即可得到有序的数列。**

**二叉查找树的时间复杂度：它和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是平衡查找树设计的初衷。**

**二叉查找树的高度决定了二叉查找树的查找效率。**

#### 3.平衡二叉树

平衡二叉树（Balanced Binary Tree）又被称为AVL树（有别于AVL算法），且具有以下性质：它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用算法有红黑树、AVL树等。在平衡二叉搜索树中，我们可以看到，其高度一般都良好地维持在O(log2n)，大大降低了操作的时间复杂度。

　　最小二叉平衡树的节点的公式如下：

F(n)=F(n-1)+F(n-2)+1

##### 3.1 AVL树

AVL树是最先发明的自平衡二叉查找树。

AVL树最关键的也是最难的一步操作就是**旋转**。旋转主要是为了实现AVL树在实施了插入和删除操作以后，树重新回到平衡的方法。

**单旋转，双旋转**

##### 3.2 红黑树

红黑树是一种自平衡二叉查找树

**红黑树的性质：**

　　红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制的一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

　　性质1. 节点是红色或黑色。

　　性质2. 根是黑色。

　　性质3. 所有叶子都是黑色（叶子是NIL节点）。

　　性质4. 每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)

性质5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

**红黑树这种数据结构应用十分广泛，在多种编程语言中被用作符号表的实现**，如：

* Java中的java.util.TreeMap,java.util.TreeSet；
* C++ STL中的：map,multimap,multiset；
* .NET中的：SortedDictionary,SortedSet 等。
* 著名的linux进程调度[Completely Fair Scheduler](https://link.zhihu.com/?target=https://en.wikipedia.org/wiki/Completely_Fair_Scheduler),用红黑树管理进程控制块.
* epoll在内核中的实现，用红黑树管理事件块.
* nginx中，用红黑树管理timer等.
* Java的TreeMap实现.

#### 4.B树

B树是一种用于查找的平衡树,但是它不是二叉树.多路平衡排序树,(搜索树也为排序树)。

B-树为系统最优化大块数据的读和写操作。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。这种数据结构常被应用在数据库和文件系统的实作上。

关键字最多m-1个，当多于m-1个需要分裂子节点(即：子节点提升，剩下的关键字分裂为两个分支)。

#### 5.B+树

　　B+树是B树的变体，也是一种多路搜索树：

　　1) 其定义基本与B-树相同，除了：

　　2) 非叶子结点的子树指针与关键字个数相同；

　　3) 非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

　　4) 为所有叶子结点增加一个链指针；

5) 所有关键字都在叶子结点出现；

**B/B+树常用于 文件系统和数据库系统中**，它通过对每个节点存储个数的扩展，使得对连续的数据能够进行较快的定位和访问，能够有效减少查找时间，提高存储的空间局部性从而减少IO操作。它广泛用于文件系统及数据库中，如：

* Windows：HPFS文件系统；
* Mac：HFS，HFS+文件系统；
* Linux：ResiserFS，XFS，Ext3FS，JFS文件系统；
* 数据库：ORACLE，MYSQL，SQLSERVER等中。

#### 6.B\*树

B\*树是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针，将结点的最低利用率从1/2提高到2/3。

#### 7. Trie树

　　Tire树称为字典树，又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

　　Tire树的三个基本性质：

　　1) 根节点不包含字符，除根节点外每一个节点都只包含一个字符；

　　2) 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；

　　3) 每个节点的所有子节点包含的字符都不相同。

　　Tire树的应用：

　　1) 串的快速检索

　　给出N个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。

在这道题中，我们可以用数组枚举，用哈希，用字典树，先把熟词建一棵树，然后读入文章进行比较，这种方法效率是比较高的。

　　2) “串”排序

　　给定N个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出。用字典树进行排序，采用数组的方式创建字典树，这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行先序遍历即可。

　　3) 最长公共前缀

　　对所有串建立字典树，对于两个串的最长公共前缀的长度即他们所在的结点的公共祖先个数，于是，问题就转化为求公共祖先的问题。

### 212）尾递归

如果一个函数中所有递归形式的调用都出现在函数的末尾，我们称这个递归函数是尾递归的。

线性递归:

|  |  |
| --- | --- |
| 1  2  3 | **long** Rescuvie(**long** n) {  **return** (n == 1) ? 1 : n \* Rescuvie(n - 1);  } |

尾递归:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **long** TailRescuvie(**long** n, **long** a) {  **return** (n == 1) ? a : TailRescuvie(n - 1, a \* n);  }  **long** TailRescuvie(**long** n) {//封装用的    **return** (n == 0) ? 1 : TailRescuvie(n, 1);  } |

尾递归是用来优化递归算法空间复杂度的，其原理是当递归调用出现在函数的最后一步时，编译器就可以丢弃当前函数的调用帧，这样，整个递归过程中，就仅仅存在一个调用帧，这样就减小了内存的消耗。

对快速排序：尾递归与递归效率差不多，但是可以减少递归深度。

### 213）b标签和strong标签给我们的主观感受都是加粗，但对搜索引擎来说b标签和普通的文字并没有什么区别，而strong标签却是起强调作用的。也就是说如果你想让搜索引擎认为你的某句话很重要时那就用strong标签。如果只是想让用户看到加粗的效果，那就用b标签。

同理如em标签也是针对搜索引擎来起作用的，i标签只是让用户看到展示的是斜体。

title网站标题能直接告诉搜索引擎和用户这个页面是关于什么主题和内容的网站。从文章的角度上看，用户进到内容页里，想看到的当然就是文章的内容，H1文章标题则是最重要的、最应该突出的。

### 214)几个函数：

**Object.assign()（复制和合并对象）**

Object.assign() 方法用于把一个或多个源对象的可枚举属性值复制到目标对象中，返回值为目标对象。

**语法**

Object.assign(target, ...sources)

target: 目标对象  
sources: 源对象

**返回值** 目标对象

**描述**

Object.assign 方法只复制源对象中可枚举的属性和对象自身的属性。它在源对象上使用 [[Get]]， 在目标对象上使用 [[Set]]， 会调用 getter 和 setter。它不适合用于把一个包含 getter 的对象属性合并到一个原型中。如果要把属性定义连同可枚举性复制到一个原型中，应该使用 Object.getOwnPropertyDescriptor() 和 Object.defineProperty() 方法。

String 和 Symbol 类型的属性都会被复制。

当发生错误时，例如有一个属性是不可写的，将会抛出一个 TypeError 错误，目标对象保持不变。

注意 Object.assign() 源对象为 null 或 undefined 时不会报错。

**Object.create(proto [, propertiesObject ])**

是E5中提出的一种新的对象创建方式，第一个参数是要继承的原型，如果不是一个子函数，可以传一个null，第二个参数是对象的属性描述符，这个参数是可选的。

**object.defineProperties(object, descriptors)**

方法直接在一个对象上定义新的属性或修改现有属性，并返回该对象。

object

必需。  对其添加或修改属性的对象。  这可以是本机 JavaScript 对象或 DOM 对象。

descriptors

必需。  包含一个或多个描述符对象的 JavaScript 对象。  每个描述符对象描述一个数据属性或访问器属性。

1. process.memoryUsage() nodejs内存管理

rss为进程的常驻内存（node所占的内存），

heapTotal：总堆、

heapUsed：堆内存使用情况

External： V8 引擎内部的 C++ 对象占用的内存。

判断内存泄漏，以heapUsed字段为准。

node-memwatch：

stats: { num\_full\_gc: 4, // 第几次全堆垃圾回收

num\_inc\_gc: 23, // 第几次增量垃圾回收

heap\_compactions: 4, // 第几次对老生代整理

usage\_trend: 0, // 使用趋势

estimated\_base: 7152944, // 预估基数

current\_base: 7152944, // 当前基数

min: 6720776, // 最小

max: 7152944 //最大 }

### **215)**实现**异步编程的几种方式**

1. 回调函数

把f2写成f1的回调函数。

回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度[耦合](http://en.wikipedia.org/wiki/Coupling_(computer_programming))（Coupling），流程会很混乱，而且每个任务只能指定一个回调函数。

1. 事件监听

另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以["去耦合"](http://en.wikipedia.org/wiki/Decoupling)（Decoupling），有利于实现[模块化](http://www.ruanyifeng.com/blog/2012/10/javascript_module.html)。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。

1. 发布/订阅

存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做["发布/订阅模式"](http://en.wikipedia.org/wiki/Publish-subscribe_pattern)（publish-subscribe pattern），又称["观察者模式"](http://en.wikipedia.org/wiki/Observer_pattern)（observer pattern）。

1. Promises对象

Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供[统一接口](http://wiki.commonjs.org/wiki/Promises/A)。

简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。

Promise其实是状态机：

有三种状态：”pending”, “resolved”, “rejected”

相应的状态的事件处理函数：then， done， fail， always

更新对应状态，并且调用注册函数：resolve， reject

### ****216)****JS数据结构转化为JSON字符串 —— JSON.stringify

**基本使用 —— 仅需一个参数**

传入一个JSON格式的JS对象或者数组

**第二个参数是函数**

**var** friend={

"firstName": "Good",

"lastName": "Man",

"phone":"1234567",

"age":18

};

**var** friendAfter=JSON.stringify(friend,**function**(key,value){

**if**(key==="phone")

**return** "(000)"+value;

**else** **if**(**typeof** value === "number")

**return** value + 10;

**else**

**return** value; //如果你把这个else分句删除，那么结果会是undefined

});

如果制定了第二个参数是函数，那么这个函数必须对每一项都有返回，这个函数接受两个参数，一个键名，一个是属性值，函数必须针对每一个原来的属性值都要有新属性值的返回。

**第二个参数若是数组**

**var** friend={

"firstName": "Good",

"lastName": "Man",

"phone":"1234567",

"age":18

};

//注意下面的数组有一个值并不是上面对象的任何一个属性名**var** friendAfter=JSON.stringify(friend,["firstName","address","phone"]);

console.log(friendAfter);//{"firstName":"Good","phone":"1234567"}//指定的“address”由于没有在原来的对象中找到而被忽略

如果第二个参数是一个数组，那么只有在数组中出现的属性才会被序列化进结果字符串，只要在这个提供的数组中找不到的属性就不会被包含进去，而这个数组中存在但是源JS对象中不存在的属性会被忽略，不会报错。

**第三个参数用于美化输出 —— 不建议用**

指定缩进用的空白字符，可以取以下几个值：

是1-10的某个数字，代表用几个空白字符

是字符串的话，就用该字符串代替空格，最多取这个字符串的前10个字符

没有提供该参数 等于 设置成null 等于 设置一个小于1的数

**将JSON字符串解析为JS数据结构 —— JSON.parse**

如果第一个参数，即JSON字符串不是合法的字符串的话，那么这个函数会抛出错误，所以如果你在写一个后端返回JSON字符串的脚本，最好调用语言本身的JSON字符串相关序列化函数，而如果是自己去拼接实现的序列化字符串，那么就尤其要注意序列化后的字符串是否是合法的，**合法指这个JSON字符串完全符合JSON要求的严格格式**。

值得注意的是这里有一个可选的第二个参数，这个参数必须是一个函数，这个函数作用在属性已经被解析但是还没返回前，将属性处理后再返回。

JSON.parse(friendAfter,**function**(k,v){

console.log(k);

console.log(v);

console.log("----");

});

遍历的时候，从头到尾进行遍历，如果是简单属性值（数值、字符串、布尔值和null），那么直接遍历完成，如果是遇到属性值是对象或者数组形式的，那么暂停，先遍历这个**子JSON**，而遍历的原则也是一样的，等这个**复合属性遍历完成**，那么再完成对这个属性的遍历返回。

**本质上，这就是一个深度优先的遍历。**

**影响 JSON.stringify 的神奇函数 —— object.toJSON**

如果你在一个JS对象上实现了toJSON方法，那么调用JSON.stringify去序列化这个JS对象时，JSON.stringify会把这个对象的toJSON方法返回的值作为参数去进行序列化。

**var** info={

"msg":"I Love You",

"toJSON":**function**(){

**var** replaceMsg=**new** Object();

replaceMsg["msg"]="Go Die";

**return** replaceMsg;

}

};

JSON.stringify(info);//出si了，返回的是：'"{"msg":"Go Die"}"',说好的忽略函数呢

其实Date类型可以直接传给JSON.stringify做参数，其中的道理就是，Date类型内置了toJSON方法。

**以上所用的3个函数，不兼容IE7以及IE7之前的浏览器。有关兼容性的讨论，留待之后吧。如果想直接在应用上解决兼容性，那么可以套用JSON官方的js，可以解决。**

### 217)Preload与Prefetch

**DNS prefetch**

DNS prefetching通过指定具体的URL来告知客户端未来会用到相关的资源，这样浏览器可以尽早的解析DNS。比如我们需要一个在example.com的图片或者视频文件。在<head>就可以这么写:

<linkrel="dns-prefetch"href="//example.com">

​当请求这个域名下的文件时就不需要等待DNS查询了。项目中有用到第三方的代码时这么做尤其有益（其他的使用场景，比如当静态资源和HTML不在一个域上，而在CDN上；又比如在重定向前可以加上DNS prefetch）。

**Preconnect**

和DNS prefetch类似，preconnect不光会解析DNS，还会建立TCP握手连接和TLS协议（如果需要）。用法如下：

<linkrel="preconnect"href="http://css-tricks.com">

现代浏览器竭尽所能的尝试预测网站可能需要哪些链接。通过提前连接，浏览器可以提前建立必要的通信，消除了实际请求中DNS、TCP和TLS的耗时。不过，即使是只能的现代浏览器，也没办法为每个网站可靠的预测所有连接。

**Prefetch**

当能确定网页在未来一定会使用到某个资源时，开发者可以让浏览器提前请求并且缓存好以供后续使用。prefetch支持预拉取图片、脚本或者任何可以被浏览器缓存的资源。

<linkrel="prefetch"href="image.png">

不同于DNS prefetch，上面的写法可是会去请求、下载资源并且缓存起来。当然也是有一些发生条件的。

**Subresource**

subresource可以用来指定资源是最高优先级的。比如，在Chrome和Opera中我们可以加上下面的代码：

<linkrel="subresource"href="styles.css">

和 "Link rel=prefetch"的语义不同，"Link rel=subresource"是一种新的连接关系。rel=prefetch指定了下载后续页面用到资源的低优先级，而rel=subresource则是指定当前页面资源的提前加载。

所以，如果资源是在当前页面需要，或者马上就会用到，则推荐用subresource，否则还是用prefetch。

**Prerender**

prerender是一个重量级的选项，它可以让浏览器提前加载指定页面的所有资源。

<linkrel="prerender"  href="/thenextpage.html"/>

prerender就像是在后台打开了一个隐藏的tab，会下载所有的资源、创建DOM、渲染页面、执行JS等等。如果用户进入指定的链接，隐藏的这个页面就会进入马上进入用户的视线。Google Search多年前就利用了这个特性实现了Instant Pages功能。微软最近也宣布会让Bing在IE11上用类似prerender的技术。

但是要注意，一定要在十分确定用户回点某个链接时才用这个特性，否则客户端就会无端的下载很多资源和渲染这个页面。

**preload**

一个基本的用法是提前加载资源，尽管大多数基于标记语言的资源能被浏览器的预加载器（Preloader）尽早发现，但不是所有的资源都是基于标记语言的，比如一些隐藏在 CSS 和 Javascript 中的资源。当浏览器发现自己需要这些资源时已经为时已晚，所以大多数情况，这些资源的加载都会对页面渲染造成延迟。  
**1.**浏览器可以设置正确的资源加载优先级，这种方式可以确保资源根据其重要性依次加载， 所以，Preload既不会影响重要资源的加载，又不会让次要资源影响自身的加载。

1. 浏览器可以确保请求是符合内容安全策略的，比如，如果我们的安全策略是**3.**Content-Security-Policy: script-src 'self'，只允许浏览器执行自家服务器的脚本，as 值为 script 的外部服务器资源就不会被加载。

as 属性的作用是告诉浏览器被加载的是什么资源，可能的 as 值包括： **"script" "style" "image" "media" "document"**

**4.**浏览器能根据 as 的值发送适当的 Accept 头部信息 浏览器通过 as 值能得知资源类型，因此当获取的资源相同时，浏览器能够判断前面获取的资源是否能重用。

**5.**Preload 的与众不同还体现在 onload 事件上（至少在 Chrome 中，prefetch 和 subresource 是不支持的）。也就是说你可以定义资源加载完毕后的回调函数。

此外，preload 不会阻塞 windows 的 onload 事件，除非，preload资源的请求刚好来自于会阻塞 window 加载的资源。

### 218)3d-web技术

WebVR、A-Frame、argon.js、AR.js

### 219)Vue批量更新DOM

 Vue 的 DOM 更新是异步执行的。理解这一点非常重要。当侦测到数据变化时， Vue 会打开一个队列，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。假如一个 watcher 在一个事件循环中被触发了多次，它只会被推送到队列中一次。然后，在进入下一次的事件循环时， Vue 会清空队列并进行必要的 DOM 更新。

所以Vue利用js中的事件循环进行优化减少不必要的计算和DOM操作。

### 220)router是通过监听 hashChange 事件来实现。

* Vue 插件机制
* 响应式数据机制
* Vue 渲染机制
* 地址变更监听

一个 router，传入的 routes 中的每一项即为一条路由（route）配置，表示在匹配给定的地址时，应该使用什么组件渲染视图。

将 router 传入 new Vue() 用于创建根组件，这样根组件中对应的视图区域，可以基于 router 中的配置，根据页面地址显示不同的内容。当然，这还需要在组件模板中使用 <router-view> 来定义区域。

1.**路由配置**

所有路由配置都通过一个单独的对象传到 **Router** 的构造函数。

2.**导航的钩子函数**

**3.链接（Links）**

**v-link** 指令已经被 **<router-link>** 组件替代. 这个组件接受以下属性参数:

**to** : 一个路径字符串, 或者一个对象 [***location descriptor***](https://github.com/vuejs/vue-router/blob/43183911dedfbb30ebacccf2d76ced74d998448a/flow/declarations.js#L50-L57) .

**tag** : 渲染成的 html 元素类型，默认是 **<a>** .

**exact** : 用于控制当前激活项的行为.

**append** : 控制相对链接路径的追加方式

**replace** : 替代而不是作为历史条目压榨

**active-class** : 当链接项激活时增加的 CSS 样式

**4.命名视图 (Named Views)**

单个路由现在可以映射到多个命名组件。这些组件将会在渲染在对应命名的多个 **<router-view>**

**5.滚动行为（Scroll Behavior）**

**scrollBehavior** 选项接受一个函数，返回在路由导航时页面如何滚动的规则。你可以代码控制是否要滚动的页面顶部、书签或者在状态中保存的位置

router的实现方式：

H5 history Api：

history.pushState(data, title, url) *//向浏览器历史栈中增加一条记录。*

history.replaceState(data, title, url) *//替换历史栈中的当前记录。*

### 221)Js实现队列与栈

unshift() 方法可向数组的开头添加一个或更多元素，并返回新的长度。

shift() 方法用于把数组的第一个元素从其中删除，并返回第一个元素的值。

[].push()与[].pop();[].unshift()与[].shift()为栈数据结构的访问规则LIFO（后进先出）

[].push(),[].shift();[].unshift()与[].pop()为队列数据结构的访问规则FIFO（first-in-first-out，先进先出）

### 223）表格thead保持，tbody滚动

tbody{  
 display:block;  
 max-height:387px;  
 overflow-y: auto;  
}  
  
tbody tr,td{  
 display:block;  
}

### 224)[分布式一致性原理](http://www.cnblogs.com/markhe/p/5587923.html)

#### 1.BASE

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。BASE理论的核心思想是：**即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性**。

1. **基本可用**

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性.

1. **软状态**

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

1. **最终一致性**

最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

#### 2.CAP****理论****

**一个分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容错性（P：Partition tolerance）这三个基本需求，最多只能同时满足其中两项**。

**1、一致性**

在分布式环境下，一致性是指数据在多个副本之间能否保持一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一直的状态。

**2、可用性**

可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。这里的重点是"有限时间内"和"返回结果"。

**3、分区容错性**

分区容错性约束了一个分布式系统具有如下特性：**分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障**。

#### 3.分布式一致性算法：

Paxos:(发音：帕格索斯)

基于消息传递的算法

Raft:

使用分而治之的思想把算法流程分为三个子问题，选举，日志复制，安全性。

一致性hash算法：

通过hash环的数据结果实现。构造2^32的整数环，数据通过hash算法得到key，在环上顺时针查找距离这个key最近的缓存服务器节点。

### 225)POSIX

[POSIX](https://baike.baidu.com/item/POSIX)表示[可移植操作系统接口](https://baike.baidu.com/item/%E5%8F%AF%E7%A7%BB%E6%A4%8D%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E6%8E%A5%E5%8F%A3)（Portable Operating System Interface of UNIX，缩写为 POSIX ），POSIX标准定义了操作系统应该为应用程序提供的接口标准，是[IEEE](https://baike.baidu.com/item/IEEE)为要在各种UNIX操作系统上运行的软件而定义的一系列API标准的总称，其正式称呼为IEEE 1003，而国际标准名称为ISO/IEC 9945。

### 226)DOM和BOM

DOM （Document Object Model）是 W3C 的标准；[所有浏览器公共遵守的标准]  
BOM （Browser Object Model）是 各个浏览器厂商根据 DOM

window 是 BOM 对象，而非 js 对象；

### 227）css特指度和优先级

I - C - E

I代表ID，有一个id，就在I的位置上加1.

C代表Class,代表类别。有一个类，就在C的位置上加1.

E代表Element,代表元素的种类，有一个元素名，就在E的位置上加1.

**CSS优先级基础**

Important  > 内联(行内) > ID >类 |伪类 |属性选择器 >标签 >通配符> 继承

 （1）内联：<div style="color:yellow"></div>

 （2）id：#cls{color:yellow;}

1. 类:.cls{color:yellow;}

（4）伪类： a:hover{color:yellow;}

（5）属性选择：[type=button]{};

（6）标签选择器：div{};

（7）通配符;\*{};

（8）继承：子元素继承父元素的样式

（外部样式）External style sheet <（内部样式）Internal style sheet <（内联样式）Inline style

### 228）css选择器效率和命名

**CSS三大特性—— 继承、 优先级和层叠。**

继承：即子类元素继承父类的样式;

优先级：是指不同类别样式的权重比较;

层叠：是说当数量相同时，通过层叠(后者覆盖前者)的样式。

**选择器有一个固有的效率，我们来看**[**Steve Souders**](http://stevesouders.com/)**给排的一个顺序：**

id选择器（#myid）

类选择器（.myclassname）

标签选择器（div,h1,p）

相邻选择器（h1+p）

子选择器（ul < li）

后代选择器（li a）

通配符选择器（\*）

属性选择器（a[rel="external"]）

伪类选择器（a:hover,li:nth-child）

**来自Mozilla的几点建议:**

避免普遍规则

不要在ID选择器前加标签名或类名

不要在类名选择器前加标签名

尽可能使用具体的类别

避免使用后代选择器

标签分类规则中不应该包含一个子选择器

子选择器的问题

借助相关继承关系

使用范围内的样式表

**精简高效CSS命名之“三无原则”:**

没有id，没有标签，没有层级

### 229)vue-cli

# 全局安装 vue-cli

$ npm install --global vue-cli

# 创建一个基于 webpack 模板的新项目

$ vue init webpack my-project

# 安装依赖，走你

$ cd my-project

$ npm install

$ npm run dev

#### Build问题：

npm run build 后dist文件夹下面直接浏览器打开index.html，css和[**js**](http://lib.csdn.net/base/javascript)的路径都不正确。 放到跟目录下就正常了

在文件夹里面的 config - index.js

在 build 部分的 assetsPublicPath 里面加一个点即可

assetsPublicPath: '/',

改为

assetsPublicPath: './'

Vue-cli webpack配置详解：

##### package.json

"scripts": {

"dev": "node build/dev-server.js",

"build": "node build/build.js"

}

#### 开发环境

开发环境的入口文件是 [build/dev-server.js](https://github.com/chenBright/vue-cli-webpack-analysis/blob/master/build/dev-server.js)

##### [build/dev-server.js](https://github.com/chenBright/vue-cli-webpack-analysis/blob/master/build/dev-server.js)

使用express作为后端框架，结合一些关于webpack的中间件

// 配置文件

**var** config = require('../config')

// 如果 Node 的环境无法判断当前是 dev / product 环境

// 使用 config.dev.env.NODE\_ENV 作为当前的环境

**if** (!process.env.NODE\_ENV) {

process.env.NODE\_ENV = JSON.parse(config.dev.env.NODE\_ENV)

}

// 可以强制打开浏览器并跳转到指定 url 的插件

**var** opn = require('opn')

// node自带的文件路径工具**var** path = require('path')

// express框架

**var** express = require('express')

**var** webpack = require('webpack')

// 测试环境，使用的配置与生产环境的配置一样

// 非测试环境，即为开发环境，因为此文件只有测试环境和开发环境使用

**var** proxyMiddleware = require('http-proxy-middleware')

**var** webpackConfig = process.env.NODE\_ENV === 'testing'

// 生产环境配置文件

? require('./webpack.prod.conf')

// 开发环境配置文件

: require('./webpack.dev.conf')

// 端口号为命令行输入的PORT参数或者配置文件中的默认值(在windows set PORT=9000可以改变其值)

**var** port = process.env.PORT || config.dev.port

// 配置文件中 是否自动打开浏览器

**var** autoOpenBrowser = !!config.dev.autoOpenBrowser

// 配置文件中 http代理配置// https://github.com/chimurai/http-proxy-middleware**var** proxyTable = config.dev.proxyTable

// 启动 express 服务**var** app = express()

// 启动 webpack 编译**var** compiler = webpack(webpackConfig)

// 可以将编译后的文件暂存到内存中的插件

**var** devMiddleware = require('webpack-dev-middleware')(compiler, {

// 公共路径，与webpack的publicPath一样

publicPath: webpackConfig.output.publicPath,

// 不打印

quiet: true

})

// Hot-reload 热重载插件

**var** hotMiddleware = require('webpack-hot-middleware')(compiler, {

log: () => {}

})

##### webpack.dev.conf.js

dev-server.js中使用了webpack.dev.conf.js文件，该文件是开发环境中webpack的配置入口。

// 工具函数集合

**var** utils = require('./utils')

**var** webpack = require('webpack')

// 配置文件

**var** config = require('../config')

// webpack 配置合并插件

**var** merge = require('webpack-merge')

// webpac基本配置

**var** baseWebpackConfig = require('./webpack.base.conf')

// 自动生成 html 并且注入到 .html 文件中的插件

**var** HtmlWebpackPlugin = require('html-webpack-plugin')

// webpack错误信息提示插件

**var** FriendlyErrorsPlugin = require('friendly-errors-webpack-plugin')

// 将 Hol-reload 热重载的客户端代码添加到 webpack.base.conf 的 对应 entry 中，一起打包

Object.keys(baseWebpackConfig.entry).forEach(**function**(name) {

baseWebpackConfig.entry[name] = ['./build/dev-client'].concat(baseWebpackConfig.entry[name])

})

module.exports = merge(baseWebpackConfig, {

module: {

// styleLoaders

rules: utils.styleLoaders({ sourceMap: config.dev.cssSourceMap })

},

// 最新的配置为 cheap-module-eval-source-map，虽然 cheap-module-eval-source-map更快，但它的定位不准确

// 所以，换成 eval-source-map

devtool: '#eval-source-map',

plugins: [

// definePlugin 接收字符串插入到代码当中, 所以你需要的话可以写上 JS 的字符串

// 此处，插入适当的环境

// https://webpack.js.org/plugins/define-plugin/

**new** webpack.DefinePlugin({

'process.env': config.dev.env

}),

// HotModule 插件在页面进行变更的时候只会重绘对应的页面模块，不会重绘整个 html 文件

// https://github.com/glenjamin/webpack-hot-middleware#installation--usage

**new** webpack.HotModuleReplacementPlugin(),

**new** webpack.NoEmitOnErrorsPlugin(),

// 将 index.html 作为入口，注入 html 代码后生成 index.html文件

**new** HtmlWebpackPlugin({

filename: 'index.html',

template: 'index.html',

inject: true

}),

// webpack错误信息提示插件

**new** FriendlyErrorsPlugin()

]

})

##### webpack.base.conf.js

在webpack.dev.conf.js中出现webpack.base.conf.js，这个文件是开发环境和生产环境，甚至测试环境，这些环境的公共webpack配置。

// node自带的文件路径工具**var** path = require('path')// 工具函数集合**var** utils = require('./utils')

// 配置文件**var** config = require('../config')

// 工具函数集合**var** vueLoaderConfig = require('./vue-loader.conf')

/\*\*

\* 获得绝对路径

\* @method resolve

\* @param {String} dir 相对于本文件的路径

\* @return {String} 绝对路径

\*/**function** **resolve**(dir) {

**return** path.join(\_\_dirname, '..', dir)

}

module.exports = {

entry: {

app: './src/main.js'

},

output: {

// 编译输出的静态资源根路径

path: config.build.assetsRoot,

// 编译输出的文件名

filename: '[name].js',

// 正式发布环境下编译输出的上线路径的根路径

publicPath: process.env.NODE\_ENV === 'production' ?

config.build.assetsPublicPath : config.dev.assetsPublicPath

},

resolve: {

// 自动补全的扩展名

extensions: ['.js', '.vue', '.json'],

// 路径别名

alias: {

// 例如 import Vue from 'vue'，会自动到 'vue/dist/vue.common.js'中寻找

'vue$': 'vue/dist/vue.esm.js',

'@': resolve('src'),

}

},

module: {

rules: [{

// 审查 js 和 vue 文件

// https://github.com/MoOx/eslint-loader

test: /\.(js|vue)$/,

loader: 'eslint-loader',

// 表示预先处理

enforce: "pre",

include: [resolve('src'), resolve('test')],

options: {

formatter: require('eslint-friendly-formatter')

}

},

{

// 处理 vue文件

test: /\.vue$/,

},

{

// 编译 js

test: /\.js$/,

},

{

// 处理图片文件

test: /\.(png|jpe?g|gif|svg)(\?.\*)?$/,

loader: 'url-loader',

},

{

// 处理字体文件

test: /\.(woff2?|eot|ttf|otf)(\?.\*)?$/,

loader: 'url-loader'

}

]

}

}

##### config/index.js

该文件在很多文件中都用到，是主要的配置文件，包含静态文件的路径、是否开启sourceMap等。

module.exports = {

// production 生产环境

build: {

// 构建环境

env: require('./prod.env'),

// 构建输出的index.html文件

index: path.resolve(\_\_dirname, '../dist/index.html'),

// 构建输出的静态资源路径

assetsRoot: path.resolve(\_\_dirname, '../dist'),

// 构建输出的二级目录

assetsSubDirectory: 'static',

// 构建发布的根目录，可配置为资源服务器域名或 CDN 域名

assetsPublicPath: '/',

// 是否开启 cssSourceMap

productionSourceMap: true,

// Gzip off by default as many popular static hosts such as

// Surge or Netlify already gzip all static assets for you.

// Before setting to `true`, make sure to:

// npm install --save-dev compression-webpack-plugin

// 默认关闭 gzip，因为很多流行的静态资源主机，例如 Surge、Netlify，已经为所有静态资源开启gzip

productionGzip: false,

// 需要使用 gzip 压缩的文件扩展名

productionGzipExtensions: ['js', 'css'],

// Run the build command with an extra argument to

// View the bundle analyzer report after build finishes:

// `npm run build --report`

// Set to `true` or `false` to always turn it on or off

// 运行“build”命令行时，加上一个参数，可以在构建完成后参看包分析报告

// true为开启，false为关闭

bundleAnalyzerReport: process.env.npm\_config\_report

},

// dev 开发环境

dev: {

// 构建环境

env: require('./dev.env'),

// 端口号

port: 3333,

// 是否自动打开浏览器

autoOpenBrowser: true,

assetsSubDirectory: 'static',

// 编译发布的根目录，可配置为资源服务器域名或 CDN 域名

assetsPublicPath: '/',

// proxyTable 代理的接口（可跨域）

// 使用方法：https://vuejs-templates.github.io/webpack/proxy.html

proxyTable: {},

// CSS Sourcemaps off by default because relative paths are "buggy"

// with this option, according to the CSS-Loader README

// (https://github.com/webpack/css-loader#sourcemaps)

// In our experience, they generally work as expected,

// just be aware of this issue when enabling this option.

// 默认情况下，关闭 CSS Sourcemaps，因为使用相对路径会报错。

// CSS-Loader README：https://github.com/webpack/css-loader#sourcemaps

cssSourceMap: false

}

}

##### utils.js

utils.js也是一个被使用频率的文件，这个文件包含了三个工具函数：

生成静态资源的路径

生成 ExtractTextPlugin对象或loader字符串

生成 style-loader的配置

// node自带的文件路径工具**var** path = require('path')// 配置文件**var** config = require('../config')// 提取css的插件// https://github.com/webpack-contrib/extract-text-webpack-plugin**var** ExtractTextPlugin = require('extract-text-webpack-plugin')

/\*\*

\* 生成静态资源的路径

\* @method assertsPath

\* @param {String} \_path 相对于静态资源文件夹的文件路径

\* @return {String} 静态资源完整路径

\*/

exports.assetsPath = **function** (\_path) {

**var** assetsSubDirectory = process.env.NODE\_ENV === 'production'

? config.build.assetsSubDirectory

: config.dev.assetsSubDirectory

// path.posix.join与path.join一样，不过总是以 posix 兼容的方式交互

**return** path.posix.join(assetsSubDirectory, \_path)

}

/\*\*

\* 生成处理css的loaders配置

\* @method cssLoaders

\* @param {Object} options 生成配置

\* option = {

\* // 是否开启 sourceMap

\* sourceMap: true,

\* // 是否提取css

\* extract: true

\* }

\* @return {Object} 处理css的loaders配置对象

\*/

exports.cssLoaders = **function** (options) {

options = options || {}

/\*\*

\* 生成 ExtractTextPlugin对象或loader字符串

\* @method generateLoaders

\* @param {Array} loaders loader名称数组

\* @return {String|Object} ExtractTextPlugin对象或loader字符串

\*/

**function** **generateLoaders** (loader, loaderOptions) {

**var** loaders = [cssLoader]

**if** (loader) {

loaders.push({

// 例如，sass?indentedSyntax

// 在?号前加上“-loader”

loader: loader + '-loader',

options: Object.assign({}, loaderOptions, {

sourceMap: options.sourceMap

})

})

}

// extract为true时，提取css

// 生产环境中，默认为true

**if** (options.extract) {

**return** ExtractTextPlugin.extract({

use: loaders,

fallback: 'vue-style-loader'

})

} **else** {

**return** ['vue-style-loader'].concat(loaders)

}

}

**return** {}

}

/\*\*

\* 生成 style-loader的配置

\* style-loader文档：https://github.com/webpack/style-loader

\* @method styleLoaders

\* @param {Object} options 生成配置

\* option = {

\* // 是否开启 sourceMap

\* sourceMap: true,

\* // 是否提取css

\* extract: true

\* }

\* @return {Array} style-loader的配置

\*/

exports.styleLoaders = **function** (options) {

**var** output = []

**var** loaders = exports.cssLoaders(options)

**for** (**var** extension **in** loaders) {

**var** loader = loaders[extension]

output.push({

test: **new** RegExp('\\.' + extension + '$'),

use: loader

})

}

**return** output

}

#### 生产环境

开发环境的入口文件是[build/build.js](https://github.com/chenBright/vue-cli-webpack-analysis/blob/master/build/build.js)。

##### build.js

该文件，为构建打包文件，会将源码进行构建（编译、压缩等）后打包。

// 设置当前环境为生产环境

process.env.NODE\_ENV = 'production'

// loading 插件

**var** ora = require('ora')

// 可以在 node 中执行`rm -rf`的工具

**var** rm = require('rimraf')

// node自带的文件路径工具

**var** path = require('path')

// 在终端输出带颜色的文字

**var** chalk = require('chalk')

**var** webpack = require('webpack')// 配置文件

**var** config = require('../config')

**var** webpackConfig = require('./webpack.prod.conf')

// 在终端显示loading效果，并输出提示**var** spinner = ora('building for production...')

spinner.start()

// 删除这个文件夹 （递归删除）

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {

**if** (err) **throw** err

// 构建

webpack(webpackConfig, **function** (err, stats) {

// 构建成功

// 停止 loading动画

spinner.stop()

**if** (err) **throw** err

// 打印提示

console.log(chalk.cyan(' Build complete.\n'))

console.log(chalk.yellow(

' Tip: built files are meant to be served over an HTTP server.\n' +

' Opening index.html over file:// won\'t work.\n'

))

})

})

##### webpack.prod.conf

该文件，为生产环境中webpack的配置入口(压缩文件的配置在这)。同时，它也依赖于前面提到的webpack.base.conf.js、utils.js和config/index.js。

// node自带的文件路径工具

var path = require('path')

// 工具函数集合

var utils = require('./utils')

var webpack = require('webpack')

// 配置文件

var config = require('../config')

// webpack 配置合并插件

var merge = require('webpack-merge')

// webpack 基本配置

var baseWebpackConfig = require('./webpack.base.conf')

// webpack 复制文件和文件夹的插件

// https://github.com/kevlened/copy-webpack-plugin

var CopyWebpackPlugin = require('copy-webpack-plugin')

// 自动生成 html 并且注入到 .html 文件中的插件

var HtmlWebpackPlugin = require('html-webpack-plugin')

// 提取css的插件

var ExtractTextPlugin = require('extract-text-webpack-plugin')

// webpack 优化压缩和优化 css 的插件

var OptimizeCSSPlugin = require('optimize-css-assets-webpack-plugin')

// 如果当前环境为测试环境，则使用测试环境

// 否则，使用生产环境

var env = process.env.NODE\_ENV === 'testing'

? require('../config/test.env')

: config.build.env

var webpackConfig = merge(baseWebpackConfig, {

module: {

// styleLoaders

rules: utils.styleLoaders({

})

},

// 是否开启 sourceMap

devtool: config.build.productionSourceMap ? '#source-map' : false,

output: {

// 编译输出的静态资源根路径

path: config.build.assetsRoot,

// 编译输出的文件名

filename: utils.assetsPath('js/[name].[chunkhash].js'),

// 没有指定输出名的文件输出的文件名

chunkFilename: utils.assetsPath('js/[id].[chunkhash].js')

},

plugins: [

// definePlugin 接收字符串插入到代码当中, 所以你需要的话可以写上 JS 的字符串

// 此处，插入适当的环境

new webpack.DefinePlugin({

'process.env': env

}),

// 压缩 js

new webpack.optimize.UglifyJsPlugin({

}),

// 提取 css

new ExtractTextPlugin({

filename: utils.assetsPath('css/[name].[contenthash].css')

}),

// 压缩提取出来的 css

// 可以删除来自不同组件的冗余代码

// Compress extracted CSS. We are using this plugin so that possible

// duplicated CSS from different components can be deduped.

new OptimizeCSSPlugin(),

// 将 index.html 作为入口，注入 html 代码后生成 index.html文件

new HtmlWebpackPlugin({

},

// 必须通过 CommonsChunkPlugin一致地处理多个 chunks

chunksSortMode: 'dependency'

}),

// 分割公共 js 到独立的文件

new webpack.optimize.CommonsChunkPlugin({

name: 'vendor',

minChunks: function (module, count) {

// node\_modules中的任何所需模块都提取到vendor

return (

module.resource &&

/\.js$/.test(module.resource) &&

module.resource.indexOf(

path.join(\_\_dirname, '../node\_modules')

) === 0

)

}

}),

// 将webpack runtime 和模块清单 提取到独立的文件，以防止当 app包更新时导致公共 jsd hash也更新

// extract webpack runtime and module manifest to its own file in order to

// prevent vendor hash from being updated whenever app bundle is updated

new webpack.optimize.CommonsChunkPlugin({

name: 'manifest',

chunks: ['vendor']

}),

// 复制静态资源

// https://github.com/kevlened/copy-webpack-plugin

new CopyWebpackPlugin([

{ }

])

]

})

// 开启 gzip 的情况时，给 webpack plugins添加 compression-webpack-plugin 插件

if (config.build.productionGzip) {

// webpack 压缩插件

var CompressionWebpackPlugin = require('compression-webpack-plugin')

// 向webpackconfig.plugins中加入下方的插件

webpackConfig.plugins.push(

new CompressionWebpackPlugin({})

)

}

// 开启包分析的情况时， 给 webpack plugins添加 webpack-bundle-analyzer 插件

if (config.build.bundleAnalyzerReport) {

}

### 230） [Source Map](http://blog.csdn.net/fundebug/article/details/61913777)

部署前端之前，开发者通常会对代码进行打包压缩，这样可以减少代码大小，从而有效提高访问速度。然而，压缩代码的报错信息是很难Debug的，因为它的行号和列号已经失真。这时就需要Source Map来还原真实的出错位置了。

#### 什么是Source map

简单说，Source map就是一个信息文件，里面储存着位置信息。也就是说，转换后的代码的每一个位置，所对应的转换前的位置。

有了它，出错的时候，除错工具将直接显示原始代码，而不是转换后的代码。这无疑给开发者带来了很大方便。

#### 如何启用Source map

正如前文所提到的，只要在转换后的代码尾部，加上一行就可以了。

　　//@sourceMappingURL=/path/to/file.js.map

map文件可以放在网络上，也可以放在本地文件系统。

浏览器启用：打开控制台-》（点竖的3个小点）-》settings->Sources(勾选相应条件)

#### 如何生成Source map

最常用的方法是使用Google的[Closure编译器](https://developers.google.com/closure/compiler/)。

生成命令的格式如下：

　　java -jar compiler.jar \   
　　　　--js script.js \  
　　　　--create\_source\_map ./script-min.js.map \  
　　　　--source\_map\_format=V3 \  
　　　　--js\_output\_file script-min.js

各个参数的意义如下：

　　- js： 转换前的代码文件  
　　- create\_source\_map： 生成的source map文件  
　　- source\_map\_format：source map的版本，目前一律采用V3。  
　　- js\_output\_file： 转换后的代码文件。

#### Source map的格式

打开Source map文件，它大概是这个样子：

　　{  
　　　　version : 3,  
　　　　file: "out.js",  
　　　　sourceRoot : "",  
　　　　sources: ["foo.js", "bar.js"],  
　　　　names: ["src", "maps", "are", "fun"],  
　　　　mappings: "AAgBC,SAAQ,CAAEA"  
　　}

整个文件就是一个JavaScript对象，可以被解释器读取。它主要有以下几个属性：

　　- version：Source map的版本，目前为3。

　　- file：转换后的文件名。

　　- sourceRoot：转换前的文件所在的目录。如果与转换前的文件在同一目录，该项为空。

　　- sources：转换前的文件。该项是一个数组，表示可能存在多个文件合并。

　　- names：转换前的所有变量名和属性名。

　　- mappings：记录位置信息的字符串，下文详细介绍。

#### mappings属性

下面才是真正有趣的部分：两个文件的各个位置是如何一一对应的。

关键就是map文件的mappings属性。这是一个很长的字符串，它分成三层。

　　第一层是行对应，以分号（;）表示，每个分号对应转换后源码的一行。所以，第一个分号前的内容，就对应源码的第一行，以此类推。

　　第二层是位置对应，以逗号（,）表示，每个逗号对应转换后源码的一个位置。所以，第一个逗号前的内容，就对应该行源码的第一个位置，以此类推。

第三层是位置转换，以[VLQ编码](http://en.wikipedia.org/wiki/Variable-length_quantity)表示，代表该位置对应的转换前的源码位置。

#### 位置对应的原理

每个位置使用五位，表示五个字段。

从左边算起，

　　- 第一位，表示这个位置在（转换后的代码的）的第几列。

　　- 第二位，表示这个位置属于sources属性中的哪一个文件。

　　- 第三位，表示这个位置属于转换前代码的第几行。

　　- 第四位，表示这个位置属于转换前代码的第几列。

　　- 第五位，表示这个位置属于names属性中的哪一个变量。

**有几点需要说明**。首先，所有的值都是以0作为基数的。其次，第五位不是必需的，如果该位置没有对应names属性中的变量，可以省略第五位。再次，每一位都采用VLQ编码表示；由于VLQ编码是变长的，所以每一位可以由多个字符构成。

如果某个位置是AAAAA，由于A在VLQ编码中表示0，因此这个位置的五个位实际上都是0。它的意思是，该位置在转换后代码的第0列，对应sources属性中第0个文件，属于转换前代码的第0行第0列，对应names属性中的第0个变量。

### 231）[DOM level 1\2\3](http://www.cnblogs.com/Sandra-R/p/4781739.html)

#### DOM 级别 0

DOM 级别 0 不是 W3C 规范。而仅仅是对在 Netscape Navigator 3.0 和 Microsoft Internet Explorer 3.0 中的等价功能性的一种定义。

DOM 发展过程中的关键角色有：ArborText、IBM、Inso EPS、JavaSoft、Microsoft、Netscape、Novell、the Object Management Group、SoftQuad、Sun Microsystems 以及 Texcel。

W3C 的 DOM 级别 1 建立于此功能性之上。

DOM 级别 1

DOM 级别 1 专注于 HTML 和 XML 文档模型。它含有文档导航和处理功能。

DOM 级别 1 于 1998 年 10 月 1 日成为 W3C 推荐标准。

第二版的工作草案在 2000 年 9 月 29 日。

#### DOM 级别 2

DOM 级别 2 对 DOM 级别 1 添加了样式表对象模型，并定义了操作附于文档之上的样式信息的功能性。

DOM 级别 2 同时还定义了一个事件模型，并提供了对 XML 命名空间的支持。

作为一项 W3C 推荐标准，DOM 级别 2 规范发布于 2000 年 11 月 13 日：

DOM Level 2 核心

DOM Level 2 核心 规定了访问和更改文档内容及结构的一个 API，此 API 同时包含用于 XML 的接口。

DOM Level 2 HTML

DOM Level 2 HTML 规定了操作 HTML 文档结构和内容的 API。（这部分规范仍然是工作草案）

DOM Level 2 Views

DOM Level 2 规定了对文档视图进行访问和更改的 API。视图是与原文档相关联的表现形式或某种备用的表现形式。

DOM Level 2 Style

DOM Level 2 Style 规定了动态访问及更改内容样式表的 API。

DOM Level 2 Events

DOM Level 2 Events 规定了访问文档事件的 API。

DOM Level 2 Traversal-Range

DOM Level 2 Traversal-Range 规定了动态遍历和识别文档中内容范围的 API。

#### DOM 级别 3

DOM Level 3 规定了内容模型 (DTD 和 Schemas) 和文档验证。同时规定了文档加载和保存、文档查看、文档格式化和关键事件。DOM Level 3 建立于 DOM Core Level 2 之上。

DOM Level 3 Requirements

DOM Requirements 文档已经为 Level 3 requirements 进行了更新，并于 2000 年 4 月 12 日发布为工作草案。

下面的 DOM Level 3 工作草案发布于 2000 年 9 月 1 日：

DOM Level 3 Core

DOM Level 3 Core 规定了访问和更改文档内容、结构及样式的一个 API。

DOM Level 3 Events

通过增加新的接口和新的事件集，DOM Level 3 Events API 对 Level 2 Event API 的功能进行了扩展。

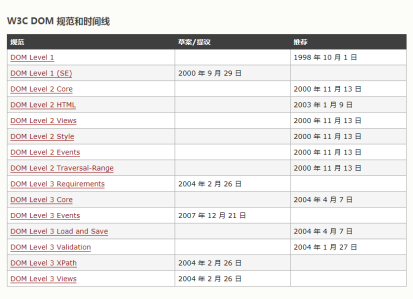
DOM Level 3 Load and Save

DOM Level 3 Content Model 规定了用于内容加载和保存、内容模型 (DTD and Schemas) 和文档验证支持的 API。

DOM Level 3 Views and Formatting

DOM Level 3 Views 规定了对文档视图进行访问和更改的 API。视图是与原文档相关联的表现形式或某种备用的表现形式。

#### W3C DOM 规范和时间线



### 232）HTML5 history 模式 和 hash 模式(router实现方式)

[**HTML5**](http://lib.csdn.net/base/html5)中的History API很好的支持了基于URL的页面无刷新操作

**history.pushState(stateObj, title, url);**

这个方法会往当前会话的历史栈中放入一条记录，stateObj是用户自定义的对象，用于记录一些有用的数据，title顾名思义就是标题信息，第三个参数是要放入的url信息。例如我们点击第二张图片时可以执行history.pushState({id: 2}, 'img: 2', '?preview=2');

**history.replaceState(stateObj, title, url);**

这个方法于pushState类似，唯一不同的是，执行这个操作后，浏览器会对会话历史栈内的记录进行替换而不是添加，这在特定场景下是比较适用的。

Hash模式利用window.loaction.hash获取url,onhashchange事件监听url的变化。

### 233）pc分为1440px,1680px,1920px.

1440px以1000px-1200px为主视觉区域，剩余的为背景。

长屏网页，全屏网页

网页设计4原则：

亲密性：

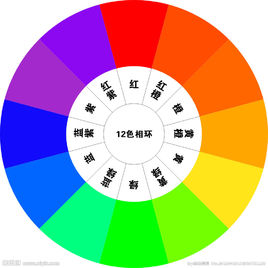
主要表达内容主次关系，逻辑关系，层次关系。

对齐：

让元素与元素间有个隐藏或显而易见的视觉关系。

重复和对比：

12色相环。



互补色可以创建很震撼的效果

相似色可以调配丰富的质感和层次。

三角色让作品颜色丰富。

千图网提供配色方案。

<http://www.58pic.com/peisebiao/>

### 234）float居中

包裹一层父元素，float: left;left: 50%;background: red;position: relative;

自己float: left;position: relative;left: -50%"

### 235）碰撞检测

**理想的碰撞**：完全按照多边形的外形和运行路径规划一个范围，在这个范围当中寻找会产生阻挡的物体，不管是什么物体，产生阻挡以后，我们运动的物体都必须在那个位置产生一个碰撞的事件。

**最简单的方法**，我们寻找物体的中心点，然后用这个中心点来画一个圆，如果是一个3D的物体，那么我们要画的就是一个球体。在检测物体碰撞的时候，我们只要检测两个物体的半径相加是否大于这两个物体圆心的实际距离。

**一种比较精密**的碰撞检测查询就是继续这种画圆的思路，然后把物体细分，对于物体的每个部件继续画圆，然后再继续进行碰撞检测，直到系统规定的，可以容忍的误差范围以后才触发碰撞事件，进行碰撞的一些操作。

#### 二维碰撞检测

##### 轴对齐边框

碰撞检测的一种较简单的形式是轴对齐的两个矩形之间，意味着没有旋转。该算法的工作原理是确保矩形的四边中任何一个之间没有间隙。任何间隙意味着不存在碰撞。

##### 圆碰撞

碰撞检测的另一简单形状在两个圆之间。该算法通过取两个圆的中心点，确保中心点之间的距离小于加在一起的两个半径。

##### 分离轴定理（SAT）

这是一种可以检测任何两个\*凸\*多边形之间的碰撞的碰撞算法。实现起来比上述方法更复杂，但更强大。这样的算法的复杂性意味着我们将需要考虑性能优化

#### 3D碰撞检测

##### 轴对齐边界框

与2D碰撞检测一样，**轴对齐边界框**（AABB）是确定两个游戏实体是否重叠的最快速算法。这包括将游戏实体包装在非旋转（因此轴对齐）框中，并且检查这些框在3D坐标空间中的位置，以查看它们是否重叠。

##### 点对AABB

检查一个点是否在AABB内部很简单 - 我们只需要检查点的坐标是否落在AABB内; 分别考虑每个轴。

##### AABB与AABB

检查AABB是否与另一个AABB相交是类似于点测试。我们只需要对每个轴做一个测试，使用框的边界。

##### 边界

使用边界球来检测碰撞比AABB复杂一点，但测试速度还是比较快的。球体的主要优点是它们不会旋转，所以如果包裹的实体旋转，则边界球体仍然是相同的。

**点对球**

要检查球体是否包含一个点，我们需要计算点与球体中心之间的距离。如果该距离小于或等于球体的半径，则该点在其内。

球体与球体

球体与球体测试类似于点对球体测试。我们需要测试的是球体中心之间的距离小于或等于其半径的总和。

球体与AABB

测试一个球体和一个AABB是否碰撞稍微复杂一些，但仍然简单快捷。一个逻辑方法是检查AABB的每个顶点，对每个顶点进行点对球体测试。然而，这是过分的 - 测试所有的顶点是不必要的，因为我们可以通过计算AABB的最近点（不一定是顶点）和球体的中心之间的距离来消除，看看它是否小于或等于球体的半径。我们可以通过将球体的中心夹在AABB的极限上来获得此值。

##### 使用物理引擎

**3D物理引擎**提供碰撞检测算法，其中大部分都是基于边界体积的。物理引擎的工作方式是通过创建一个**物理机体**，通常附加到它的视觉表示。该身体具有速度，位置，旋转，力矩等特性，也具有**物理形状**。这种形状是碰撞检测计算中考虑的形状。

### 236）让div具有blur事件

添加tabindex

<div id="blur" tabindex="1" ></div>

//去掉focus时的边框

.blur:focus{  
 outline: 0;  
}

### 237）点击某个div之外的部分

<div id="blur"></div>

方法1：

$(document).bind('click', **function**(e) {  
 **var** e = e || window.event; //浏览器兼容性  
 **var** elem = e.target || e.srcElement;  
 **while** (elem) { //循环判断至跟节点，防止点击的是div子元素  
 **if** (elem.id && elem.id == 'blur') {  
 **return**;  
 }  
 elem = elem.parentNode;  
 }  
 $('#blur').css('display', 'none'); //点击的不是div或其子元素  
});

方法2：对document的click事件绑定事件处理程序，使其隐藏该div 。对div的click事件绑定事件处理程序，阻止事件冒泡，防止其冒泡到document，而调用document的onclick方法隐藏了该div。

$(document).bind('click',**function**(){  
 $('#blur').css('display','none');  
});  
$('#blur').bind('click',**function**(e){  
 **if** (e.stopPropagation) {  
 e.stopPropagation();  
 }**else**{  
 e.cancelBubble = **true**;  
 }

});

### 238)ppi,dpi

Ppi:Pixels Per Inch也叫像素密度，所表示的是每英寸所拥有的像素数量。因此PPI数值越高，即代表显示屏能够以越高的密度显示图像。当然，显示的密度越高，拟真度就越高

Dpi:dots per inch ， 直接来说就是一英寸多少个像素点。常见取值 120，160，240。我一般称作像素密度，简称密度.

density ： 直接翻译的话貌似叫 密度。常见取值 1.5 ， 1.0 。和标准dpi的比例（160px/inc）

分辨率   ： 横纵2个方向的像素点的数量，常见取值 480X800 ，320X480

屏幕尺寸： 屏幕对角线的长度。电脑电视同理。

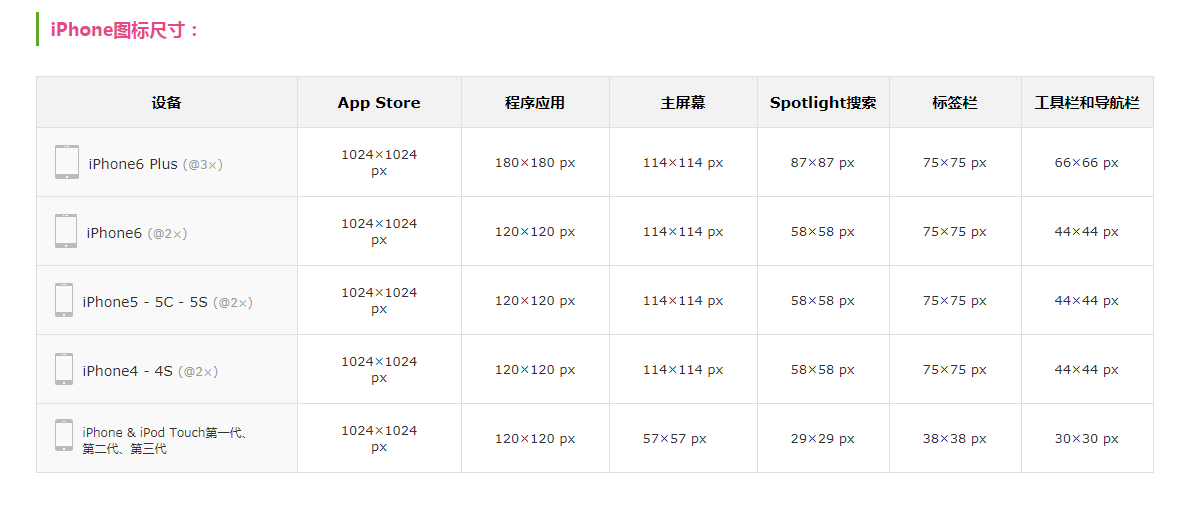
屏幕比例的问题。因为只确定了对角线长，2边长度还不一定。所以有了4：3、16：9这种，这样就可以算出屏幕边长了.

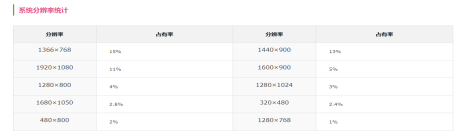
### 239）UI尺寸

未设置viewport，设备的默认宽度，基本是980px。

设置了viewport，width=device-width，弹出来的是设置好的宽度，375px、360px、320px







### 240) [canvas 的优化](https://developer.mozilla.org/zh-CN/docs/Web/API/Canvas_API/Tutorial/Optimizing_canvas)

#### canvas是内联元素，并且是属于内联置换元素.

#### 在离屏canvas上预渲染相似的图形或重复的对象

如果你发现你的在每一帧里有好多复杂的画图运算，请考虑创建一个离屏canvas，将图像在这个画布上画一次（或者每当图像改变的时候画一次），然后在每帧上画出视线以外的这个画布。

#### 避免浮点数的坐标点，用整数取而代之

#### 不要在用drawImage时缩放图像

#### 使用多层画布去画一个复杂的场景

#### 用CSS设置大的背景图

#### 用CSS transforms特性缩放画布

#### 将画布的函数调用集合到一起（例如，画一条折线，而不要画多条分开的直线）

#### 避免不必要的画布状态改变

#### 渲染画布中的不同点，而非整个新状态

#### 尽可能避免 [shadowBlur](https://developer.mozilla.org/zh-CN/docs/Web/API/CanvasRenderingContext2D/shadowBlur)特性

#### 尽可能避免[text rendering](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_text)

#### 使用不同的办法去清除画布([clearRect()](https://developer.mozilla.org/zh-CN/docs/Web/API/CanvasRenderingContext2D/clearRect) vs. [fillRect()](https://developer.mozilla.org/zh-CN/docs/Web/API/CanvasRenderingContext2D/fillRect) vs. 调整canvas大小)

#### 有动画，请使用[window.requestAnimationFrame()](https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestAnimationFrame) 而非[window.setInterval()](https://developer.mozilla.org/zh-CN/docs/Web/API/Window/setInterval)

#### 请谨慎使用大型物理库

## 241）内存屏障（Memory barrier）

内存屏障有两个能力：

1. 阻止屏障两边的指令重排序

2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效

内存屏障，在x86 上是”sfence”指令，强迫所有的、在屏障指令之前的 存储指令在屏障以前发生，并且让 store buffers 刷新到发布这个指令的 CPU cache。

## 242）协程

协程，又称微线程，纤程。英文名Coroutine。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

## 243）不让样式表渲染阻塞

@media screen and (orientation:landscape)和@media print

<link href=”a.css” rel=”stylesheet” media=”orientation:landscape”>

<link href=”a.css” rel=”stylesheet” media=”print”>

就不会出现资源加载而产生的渲染阻塞情况。

## 244)****关键渲染路径(CRP****  [Critical Rendering Path](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/)****)****

CRP总共有六步：

构建DOM树

构建CSSOM树

运行JavaScript

创建渲染树

生成布局

绘制页面

**1、构造DOM树**

DOM(Document Object Model)树是一个表示全解析过的HTML页面的对象。从根节点元素开始，会逐个创建页面中的每个元素/文本节点。元素包裹的其他元素会被作为子元素节点，并且每个节点会包含其全部的属性。

1. **构建CSSOM树**

CSSOM(CSS Object Model)是一个表示各个DOM相关样式的对象，它的表示方法与DOM相似，但是各个节点存在相关的样式值。不论其是显示、还是隐式声明这些样式。

CSS一直被认为是一种渲染阻塞资源。因此如果在首次加载时没有全部解析资源内容就无法进行渲染树的构建。与HTML不同，CSS具有层叠继承的特性，因此不能进行局部加载。定义在文档后面的样式属性会覆盖或更改写之前定义的同类属性。即，如果如果CSS可以进行局部加载的话会导致出现加载错样式的情况。因此表明，CSS必须全部解析之后才能进行下一步。

如果CSS文件适用于当前设备的话，仅仅只是会阻塞渲染。标签可以接受media属性来指定特定样式宽度的特定媒体查询。例如：如果我们有一个样式表具有orientation:landscape的媒体属性，并且我们查看该页面使用portrait模式，就不会出现资源加载而产生的渲染阻塞情况。

CSS要会导致脚本阻塞，这是由于JavaScript文件必须等待CSSOM构建结束之后才进行加载。

**3.运行JavaScript**

JavaScript被认为是解析阻塞资源，这表示当解析HTML文档自身时候会被JavaScript给阻塞掉。

当解析器解析到<script>标签时，无论该资源是内部还是外链的都会停止解析，并且等到资源被下载并运行结束后才继续进行解析。这也是为什么如果我们有一个引用了JavaScript文件的元素，它必须被放在可视文档元素之外的原因。

为避免JavaScript解析阻塞，它可以通过设定async属性来要求其异步加载。

**4、创建渲染树(Render Tree)**

渲染树是DOM树和CSSOM树的结合体，它代表最终会渲染在页面上的元素的结构对象。它表示只会关注可见内容，对于被隐藏或者CSS属性display:none的属性，不会被包含在结构内。

**5、生成布局**

布局决定了浏览器视窗大小，它提供了上下文依赖的CSS样式，如百分比或窗口的单位。视窗尺寸通常通过<head>标签中的<meta>中的viewport设定来决定。

**6、绘制页面**

最后，在绘制页面步骤。页面上的所有可见内容都会被转换为像素并呈现在屏幕上。

具体的绘制时间跟DOM数以及应用的样式有关。有些样式会话费更多的执行时间，比如复杂的渐变背景图片所需要的计算时间远超过简单固定背景色。

performance.timing这个内置的 API 可以帮助你记录下各个时间点的具体时间值

* **.navigationStart(**performance.timing.navigationStart**)** 浏览器完成卸载前一个文档的时间（也就是准备加载新页面的那个起始时间）。如果没有前一个文档，那么就返回timing.fetchStart的值。似乎只有Chrome非常严格遵守了此草案，即不把刷新页面以及一个标签页输入地址到指定页面视为发生文档的卸载。
* **.unloadEventStart** 如果前一个文档和当前文档同源，返回前一个文档发生unload事件前的时间。如果没有前一个文档或不同源则返回0。
* **.unloadEventEnd** 如果前一个文档和当前文档同源，返回前一个文档发生unload事件的时间。如果没有前一个文档或不同源则返回0。如果，发生了HTTP重定向或者类似的事情。并且从导航开始中间的每次重定向，并不都和当前文档同域的话则返回0。
* **.redirectStart** 如果发生了HTTP重定向或者类似的事情，并且从导航开始中间的每次重定向都和当前文档同域的话就返回开始重定向的timing.fetchStart的值。其他情况则返回0。
* **.redirectEnd** 如果发生了HTTP重定向或者类似的事情，并且从导航开始中间的每次重定向都和当前文档同域的话就返回最后一次重定向，接收到最后一个字节数据后的那个时间。其他情况则返回0。
* **.fetchStart** 如果一个新的资源（这里是指当前文档）获取被发起或类似的事情发生则fetchStart必须返回用户代理开始检查其相关缓存的那个时间，其他情况则返回开始获取该资源的时间。
* **.domainLookupStart**返回用户代理对当前文档所属域进行DNS查询开始的时间。如果此请求没有DNS查询过程，如长连接、资源cache、甚至是本地资源等那么就返回fetchStart的值。
* **.domainLookupEnd** 返回用户代理对结束对当前文档所属域进行DNS查询的时间。如果此请求没有DNS查询过程，如长连接、资源cache、甚至是本地资源等. 那么就返回fetchStart的值。
* **.connectStart** 返回用户代理向服务器服务器请求文档开始建立连接的那个时间，如果此连接是一个长连接又或者直接从缓存中获取资源（即没有与服务器建立连接）则返回domainLookupEnd的值。
* **.connectEnd** 返回用户代理向服务器服务器请求文档建立连接成功后（注意，不是断开连接的时间）的那个时间。如果此连接是一个长连接又或直接从缓存中获取资源 （即没有与服务器建立连接）,则返回domainLookupEnd的值。如果连接建立失败而用户代理进行重连则connectStart和connectEnd则应该是这次重连的相关的值。其中connectEnd必须包括建立连接的时间以及SSH握手协议和SOCKS认证等时间。
* **.secureConnectionStart** 可选特性。用户代理如果没有对应的东东就要把这个设置为undefined，如果有这个东东并且是HTTPS协议那么就要返回开始SSL握手的那个时间。如果不是HTTPS那么就返回0。
* **.requestStart** 返回从服务器、缓存、本地资源等开始请求文档的时间。如果请求中途连接断开了并且用户代理进行了重连并重新请求了资源，那么requestStart就必须为这个新请求所对应的时间。
* **.responseStart** 返回用户代理从服务器、缓存、本地资源中接收到第一个字节数据的时间。
* **.responseEnd** 返回用户代理接收到最后一个字符的时间，和当前连接被关闭的时间中更早的那个。同样文档可能来自服务器、缓存、或本地资源。
* **.domLoading** 返回用户代理把其文档的“current document readiness”设置为“loading”的时候。（current document readiness 其实就是document.readyState API对应的状态。）
* **.domInteractive** 返回用户代理把其文档的“current document readiness”设置为“interactive”的时候。从标准来说domReady的状态为“interactive”时意味着文档解析结束了，因为标准中描述DOM树创建结束后第一件事就是把“current document readiness”设置为“interactive”。
* **.domContentLoadedEventStart** 返回文档发生DOMContentLoaded事件的时间。DOMContentLoad和DOMInteractive之间差了两个步骤，其中之一是所有open elements出栈，然后去看看待运行的script list中是否有需要运行的脚本，如果有则执行，一直到这个列表为空了再触发DOMContentLoad.。需要主的是这个待运行脚本列表。有些可能在不同浏览器中被加入进去的行为可能不同。比如 document.write写入文档流的脚本，以及script deferr 的脚本.. 所以我们应该知道deferr的脚本也是要他推迟domContentLoaded的，也就是我们最常用的所谓domReady。
* **.domContentLoadedEventEnd** 文档的DOMContentLoaded事件的结束时间。所谓事件结束的时间是指如果DOMContentLoaded事件被开发者注册了回调事件，那么这个时间的End时间减去Start的时间就会是这个回调执行的大概事件。当然居于部分浏览器实现可能会有2-3ms的误差，但是这个时间基本可以忽略不计。类似的情况还有后面的.loadEventStart,End，即 window.onload 所有回调所消耗的时间。
* **.domComplete** 返回用户代理把其文档的“current document readiness”设置为“complete”的时候。如果current document readiness的某个状态被多次触发那么对应的domLoading, domInteractive, domContentLoadedEventStart, domContentLoadedEventEnd and domComplete这些对应的API返回的时间就应该是这个状态第一次触发的时间。
* **.loadEventStart** 文档触发load事件的时间。如果load事件没有触发那么该接口就返回0。
* **.loadEventEnd** 文档触发load事件结束后的时间。如果load事件没有触发,那么该接口就返回0。

## 245）js内建对象

## ****246）iframe****

**去边框**frameborder="no"

**iframe有那些缺点？**  
iframe会阻塞主页面的Onload事件；  
搜索引擎的检索程序无法解读这种页面，不利于SEO;  
iframe和主页面共享连接池，而浏览器对相同域的连接有限制，所以会影响页面的并行加载。  
使用iframe之前需要考虑这两个缺点。如果需要使用iframe，最好是通过javascript动态给iframe添加src属性值，这样可以绕开以上两个问题。

**iframe优点：**

1. 沙箱隔离。  
2. 引用第三方内容。  
3. 独立的带有交互的内容，比如幻灯片。  
4. 需要保持独立焦点和历史管理的子窗口，如复杂的Web应用。

## 247）跨域访问

jsonp、 iframe、window.name（即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的）、window.postMessage、服务器上设置代理页面。

**1.postmessage：**

html5实现不同域名窗口的通信。

语法：

otherWindow.postMessage(message,targetOrigin);

message:传递的数据；targetOrigin：目标域。

获取postMessage传来的数据需要通过onmessage.window.addEventListener('onmessage',function(e){ })

var worker = new Worker('a.js');

worker.onmessage = funtion(e){ }

a.js中：postMessage("hello");

跨域使用：

child.postMessage("hello","http://child.com:8080");

通过iframe嵌入页面。

在子页面：window.addEventListener('message',function(e){ e.data; e.origin })

**2.Proxy代理**

　　这种方式首先将请求发送给后台服务器，通过服务器来发送请求，然后将请求的结果传递给前端。

**3.后台进行处理**，如果确定接受请求则在返回结果中加入一个响应头：Access-Control-Allow-Origin:(允许请求的域);浏览器判断该相应头中是否包含Origin的值，如果有则浏览器会处理响应，我们就可以拿到响应数据，如果不包含浏览器直接驳回，这时我们无法拿到响应数据。

**Access-Control-Allow-Methods：（允许跨域的方式）**

access-control-allow-headers ：该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段

**跨域的判定流程**

**简单请求**

1浏览器先根据同源策略对前端页面和后台交互地址做匹配，若同源，则直接发送数据请求；若不同源，则发送跨域请求，增加一个Origin字段。

2服务器解析程序收到浏览器跨域请求后，根据自身配置返回对应文件头。若未配置过任何允许跨域，则文件头里不包含Access-Control-Allow-origin字段，若配置过域名，则返回Access-Control-Allow-origin+ 对应配置规则里的域名的方式。

3浏览器根据接受到的http文件头里的Access-Control-Allow-origin字段做匹配，**若无该字段**，说明不允许跨域；若有该字段，则对字段内容和当前域名做比对，如果同源，则说明可以跨域，浏览器发送该请求；若不同源，则说明该域名不可跨域，不发送请求

（但是不能仅仅根据服务器返回的文件头里是否包含Access-Control-Allow-origin来判断其是否允许跨域，因为服务器端配置多域名跨域的时候，也会出现不能跨域的域名返回包里没有Access-Control-Allow-origin字段的情况。下文配置说明里会讲。)

**非简单请求**

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

浏览器发现，这是一个非简单请求，就自动发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。

OPTIONS /cors HTTP/1.1

Origin: [http://api.bob.com](http://api.bob.com/)

Access-Control-Request-Method: PUT

Access-Control-Request-Headers: X-Custom-Header

Host: api.alice.comAccept-Language: en-USConnection: keep-alive

User-Agent: Mozilla/5.0..

**预检请求的回应**

服务器收到"预检"请求以后，检查了Origin、Access-Control-Request-Method和Access-Control-Request-Headers字段以后，确认允许跨源请求，就可以做出回应。

**浏览器的正常请求和回应**

一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样，会有一个Origin头信息字段。服务器的回应，也都会有一个Access-Control-Allow-Origin头信息字段。

## 248）一些****html常见兼容性问题****

1.IE6双边距BUG float引起的 使用display

 float: left; margin-left: 100px;结果margin-left变成了200px

2.3像素问题使用float引起的使用dislpay:inline -3px

3.超链接hover点击后失效 使用正确的书写顺序 link visited hover active

4.Ie z-index问题给父级添加position:relative

5.Png透明使用js代码改

6.Min-height最小高度！Important解决’

7.select在ie6下遮盖使用iframe嵌套

8.为什么没有办法定义1px左右的宽度容器（IE6默认的行高造成的，使用over:hidden,zoom:0.08 line-height:1px）

9.IE5-8不支持opacity，解决办法：

.opacity {

       opacity: 0.4

     filter: alpha(opacity=60); /\* for IE5-7 \*/

     -ms-filter:"progid:DXImageTransform.Microsoft.Alpha(Opacity=60)"; /\* for IE 8\*/

}

10. IE6不支持PNG透明背景，解决办法: IE6下使用gif图片

## 249）进程与线程

进程：处于执行期的程序。还包含其它资源。**资源分配的基本单位。**

线程：在进程中活动的对象。内核调度的对象是线程而不是进程。程序执行流的最小单元。一个标准的线程由线程ID，当前指令[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)(PC），[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8)集合和[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88)组成。线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源。1.**可并发执行。2.共享**[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)资源。

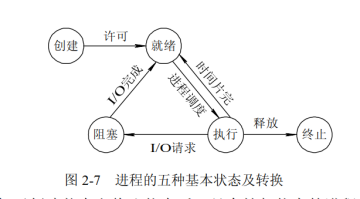
**进程状态：**

**三态模型**：

运行、就绪和阻塞

**五态模型**：

新建态，就绪态，运行态，阻塞态，终止态



内核把进程列表放在叫做任务队列的双向循环链表中。每一项为task\_struct也就是进程控制块（PCB）PCB是进程的唯一标识.包含了内核管理进程所有信息。

**进程描述符：**PID最大默认为32768，在<linux/threads.h>可以改。

**进程状态：**每个进程必属于五种状态之一。

TASK\_RUNNING（运行）,TASK\_INTERRUPTINLE（可中断，睡眠）,TASK\_UNINTERRUPTIBLE（不可中断）,\_\_TASK\_TRACED（被其它进程跟踪）,\_\_TASK\_STOPPED(停止)

**创建进程：**

Fork();linux fork(),使用写时拷贝页实现。fork创建子进程，把父进程数据空间、堆和栈**复制**一份；vfork创建子进程，与父进程内存数据**共享**；

**线程的实现：**

线程是现代编程的抽象概念。该机制提供同一程序内共享内存地址空间运行的一组线程。可以共享打开的文件资源。Linux把所有线程当进程实现，线程仅仅是一个与其它进程共享资源的进程。线程也有自己的stask\_struct.

**创建线程：**Clone();需要传递一些标志指明共享资源。

内核线程：内核线程没有独立地址，可以调度，可以抢占。

**孤儿进程：PID1是第一个进程，所以进程的父进程。孤儿进程会被托管到它下面。**

**进程调度：**

**进程有IO消耗型，处理器消耗型。Linux为了保证交互，倾向于调度IO消耗性。**

**进程优先级：linux使用nice(-20<+19),默认为0，越大优先级更低。第二种，实时优先级，0-99，值越大优先级越高。**

**时间片：表明进程被抢占前能持续运行时间。**

**调度类型：长程调度，短程调度，中程调度。**

**调度算法：**

**先来先服务FCFS，短进程优先，时间片轮转调度，基于优先级的调度，剩余时间最短者优先，响应比高者优先（缺点：很难估计预期执行时间），反馈调度算法（结合了优先级和时间片轮转）。**

**实时调度算法：最早截至时间优先调度算法EDF，最低松弛度优先LLF算法，速度单调调度算法RMS。**

**Linux调度算法：**

**Linux调度器以模块方式提供，允许不同类型进程可以正对性的选择调度算法。这些模块称为调度类。**

**CFS公平调度：**

**允许每个进程运行一段时间，循环轮转，选择运行最少的进程作为下一个运行进程，而不是采用分配每个进程时间片的做法。**

**调度实现：**

1. **时间记账 2.进程选择 3.调度器入口 4.睡眠和唤醒**

**CFS没有时间片概念，必须维护每个进程的时间记账。Vruntime变量存放进程的虚拟运行时间，即它运行了多旧还会运行多久。CFS使用红黑树组织进程队列，利用其迅速找到最小vruntime值的进程。Linux中红黑树为rbtree,他是自平衡二叉树。**

**进程间通信：**

**管道( pipe )：**管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

**有名管道 (named pipe)** ： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

**信号量( semophore )** ： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

**消息队列**( message queue ) ： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

**信号** ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

**共享内存**( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

**套接字( socket )** ： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

多线程与多进程对比：



## 250）web worker

它允许在 Web 程序中并发执行多个 JavaScript 脚本，每个脚本执行流都称为一个线程，彼此间互相独立，并且有浏览器中的 JavaScript 引擎负责管理。

HTML5 中的 Web Worker 可以分为两种不同线程类型，一个是专用线程 Dedicated Worker，一个是共享线程 Shared Worker。

#### 专用线程（dedicated worker）：

在创建专用线程的时候，需要给 Worker 的构造函数提供一个指向 JavaScript 文件资源的 URL，这也是创建专用线程时 Worker 构造函数所需要的唯一参数。当这个构造函数被调用之后，一个工作线程的实例便会被创建出来。下面是创建专用线程代码示例：

|  |  |
| --- | --- |
| 1 | var worker = new Worker('dedicated.js'); |

**与一个专用线程通信：**

专用线程在运行的过程中会在后台使用 MessagePort 对象，而 MessagePort 对象支持 HTML5 中多线程提供的所有功能，例如：可以发送和接受结构化数据（JSON 等），传输二进制数据，并且支持在不同端口中传输数据等。

为了在页面主程序接收从专用线程传递过来的消息，我们需要使用工作线程的 onmessage 事件处理器，定义 onmessage 的实例代码如下：

|  |  |
| --- | --- |
| 1 | worker.onmessage = function (event) { ... }; |

另外，开发人员也可以选择使用 addEventListener 方法，它最终的实现方式和作用和 onmessage 相同。

如果要想一个专用线程发送数据，那么我们需要使用线程中的 postMessage 方法。专用线程不仅仅支持传输二进制数据，也支持结构化的 JavaScript 数据格式。在这里有一点需要注意，为了高效地传输 ArrayBuffer 对象数据，需要在 postMessage 方法中的第二个参数中指定它。实例代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | worker.postMessage({   operation: 'list\_all\_users',   //ArrayBuffer object   input: buffer,   threshold: 0.8,  }, [buffer]); |

#### 共享线程 Shared Worker

共享线程可以由两种方式来定义：一是通过指向 JavaScript 脚本资源的 URL 来创建，而是通过显式的名称。当由显式的名称来定义的时候，由创建这个共享线程的第一个页面中使用 URL 会被用来作为这个共享线程的 JavaScript 脚本资源 URL。通过这样一种方式，它允许同域中的多个应用程序使用同一个提供公共服务的共享线程，从而不需要所有的应用程序都去与这个提供公共服务的 URL 保持联系。

无论在什么情况下，共享线程的作用域或者是生效范围都是由创建它的域来定义的。因此，两个不同的站点（即域）使用相同的共享线程名称也不会冲突。

共享线程的创建

创建共享线程可以通过使用 SharedWorker() 构造函数来实现，这个构造函数使用 URL 作为第一个参数，即是指向 JavaScript 资源文件的 URL，同时，如果开发人员提供了第二个构造参数，那么这个参数将被用于作为这个共享线程的名称。创建共享线程的代码示例如下：

var worker = new SharedWorker('sharedworker.js', ’ mysharedworker ’ );

与共享线程通信

共享线程的通信也是跟专用线程一样，是通过使用隐式的 MessagePort 对象实例来完成的。当使用 SharedWorker() 构造函数的时候，这个对象将通过一种引用的方式被返回回来。我们可以通过这个引用的 port 端口属性来与它进行通信。发送消息与接收消息的代码示例如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // 从端口接收数据 , 包括文本数据以及结构化数据  1. worker.port.onmessage = function (event) { define your logic here... };  // 向端口发送普通文本数据  2. worker.port.postMessage('put your message here … ');  // 向端口发送结构化数据  3. worker.port.postMessage({ username: 'usertext'; live\_city:  ['data-one', 'data-two', 'data-three','data-four']}); |

上面示例代码中，第一个我们使用 onmessage 事件处理器来接收消息，第二个使用 postMessage 来发送普通文本数据，第三个使用 postMessage 来发送结构化的数据，这里我们使用了 JSON 数据格式。

#### Web Worker典型应用场景

1）  使用专用线程进行数学运算

Web Worker最简单的应用就是用来做后台计算，而这种计算并不会中断前台用户的操作

2）  图像处理

通过使用从<canvas>或者<video>元素中获取的数据，可以把图像分割成几个不同的区域并且把它们推送给并行的不同Workers来做计算

3）  大量数据的检索

当需要在调用 ajax后处理大量的数据，如果处理这些数据所需的时间长短非常重要，可以在Web Worker中来做这些，避免冻结UI线程。

4）  背景数据分析

由于在使用Web Worker的时候，我们有更多潜在的CPU可用时间，我们现在可以考虑一下JavaScript中的新应用场景。例如，我们可以想像在不影响UI体验的情况下实时处理用户输入。利用这样一种可能，我们可以想像一个像Word（Office Web Apps 套装）一样的应用：当用户打字时后台在词典中进行查找，帮助用户自动纠错等等。

## 251)async和defer

如果 async="async"：脚本相对于页面的其余部分异步地执行（当页面继续进行解析时，脚本将被执行）

如果不使用 async 且 defer="**defer**"：脚本将在页面完成解析时执行，即使脚本已经下载完成。

如果既不使用 async 也不使用 defer：在浏览器继续解析页面之前，立即读取并执行脚本

一般的script标签（不带async等属性）加载时会阻塞浏览器，也就是说，浏览器在下载或执行该[js代码](https://www.baidu.com/s?wd=js%E4%BB%A3%E7%A0%81&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Ykryu9uHc3nHc3uyDvuWfd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHR1rHDdrHDdn1DvnH0vPHDYPs)块时，后面的标签不会被解析.

defer 属性标注的脚本是延迟脚本，使得浏览器延迟脚本的执行，也就是说，脚本会被异步下载但是不会被执行，直到文档的载入和解析完成，并可以操作，脚本才会被执行。

async 属性标注的脚本是异步脚本，即异步下载脚本时，不会阻塞文档解析，但是**一旦下载完成后，立即执行，阻塞文档解析**。

延迟脚本会按他们在文档里的出现顺序执行。

异步脚本在它们载入后执行，但是不能保证执行顺序。

使用async的意义就在于使得下载脚本时，不会阻塞文档的解析。因为async的脚本执行顺序是没有保证的，因此要确认脚本间没有依赖关系。

通常如果js不需要改变DOM结构时可以使用async进行异步加载（比如一些统计代码可以异步加载，因为此代码与页面执行逻辑无关，不会改变DOM结构）

## 252) Object.create()创建一个具有指定原型且可选择性地包含指定属性的对象

Object.create（proto [，propertiesObject ]）

**参数**

**proto**

应该是新创建的对象的原型的对象。

**propertiesObject**

可选的。如果没有指定，则可[undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined)枚举自己的属性（即其自身所定义的属性，沿其原型链定义的不可枚举的属性）的对象将指定要添加到新创建的对象的属性描述符以及相应的属性名称。这些属性对应于第二个参数[Object.defineProperties()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties)。

**实现继承：**

function Car (desc) {

    this.desc = desc;

    this.color = "red";

}

var car =  Object.create(Car.prototype);

**使用propertiesObject参数：**

var obj = {

            a:function(){

                console.log(100)

            },

            b:function(){

                console.log(200)

            },

            c:function(){

                console.log(300)

            }

        }

        var newObj = {};

        newObj = Object.create(obj,{

            t1:{

                value:'yupeng',

                writable:true

            },

            bar: {

                configurable: false,

                get: function() { return bar; },

                set: function(value) { bar=value }

            }

        })

## 253）渐变色

多色渐变

background: linear-gradient(to bottom right,yellow,blanchedalmond,green,blue) no-repeat;

**重复的线性渐变**

background: -webkit-repeating-linear-gradient(red, yellow 10%, green 20%) no-repeat;

**带占比的渐变色：**

background: linear-gradient(to bottom right,yellow 10%,blanchedalmond 20%,green 60%,blue 100%) no-repeat;

**径向渐变：**

background: radial-gradient(red, green, blue);

**带占比的径向渐变：**

background: radial-gradient(red 5%, green 15%, blue 60%);

**带形状的径向渐变：（**circle 表示圆形，ellipse 表示椭圆形**）**

background: radial-gradient(circle, red, yellow, green);

## 254）navigator.userAgent

***Chrome*** *:*"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.104 Safari/537.36"

***Firefox:***"Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:54.0) Gecko/20100101 Firefox/54.0"

***Edge:***"Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; rv:11.0) like Gecko"

***Ie11:***"Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)"

***Ie8:***"Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)"

**var** u = navigator.userAgent;

**mobile :** !!u.match(/AppleWebKit.\*Mobile.\*/),//是否为移动终端

**android :** u.indexOf('Android') > -1 || u.indexOf('Linux') > -1, //android终端或者uc浏览器

**iPhone :** u.indexOf('iPhone') > -1, //是否为iPhone或者QQHD浏览器

**iPad :** u.indexOf('iPad') > -1, //是否iPad

**webApp :** u.indexOf('Safari') == -1, //是否web应该程序，没有头部与底部

**weixin :** u.indexOf('MicroMessenger') > -1,//是否微信 （2015-01-22新增）

**qq :** u.match(/\sQQ/i) == " qq" //是否QQ

## 255）ECMA-262

**ECMA是European Computer Manufacturers Association的缩写，即**[欧洲计算机制造商协会](https://www.baidu.com/s?wd=%E6%AC%A7%E6%B4%B2%E8%AE%A1%E7%AE%97%E6%9C%BA%E5%88%B6%E9%80%A0%E5%95%86%E5%8D%8F%E4%BC%9A&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPjfLPvfYnHnkmynsmycd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW64njTLrHR3rjb4njfzPjfY)**。**[欧洲计算机制造商协会](https://www.baidu.com/s?wd=%E6%AC%A7%E6%B4%B2%E8%AE%A1%E7%AE%97%E6%9C%BA%E5%88%B6%E9%80%A0%E5%95%86%E5%8D%8F%E4%BC%9A&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPjfLPvfYnHnkmynsmycd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW64njTLrHR3rjb4njfzPjfY)**是制定**[信息传输](https://www.baidu.com/s?wd=%E4%BF%A1%E6%81%AF%E4%BC%A0%E8%BE%93&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPjfLPvfYnHnkmynsmycd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW64njTLrHR3rjb4njfzPjfY)**与通讯的国际化标准组织。**

**ECMAScript是ECMA制定的标准化**[脚本语言](https://www.baidu.com/s?wd=%E8%84%9A%E6%9C%AC%E8%AF%AD%E8%A8%80&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YkPjfLPvfYnHnkmynsmycd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPW64njTLrHR3rjb4njfzPjfY)**。**

ECMA-262 自然是ECMA第262号标准，这个标准就是ECMAScript。

TC39 是制定ECMA-262号标准的组织。

## 256）DOM createEvent() 方法（自定义事件）

语法：

createEvent(eventType)

eventType 的合法值和每个值创建的事件接口：

|  |  |  |
| --- | --- | --- |
| HTMLEvents | HTMLEvent | iniEvent() |
| MouseEvents | MouseEvent | iniMouseEvent() |
| UIEvents | UIEvent | iniUIEvent() |

实例：// Create the event.

var event = document.createEvent('Event');

// Define that the event name is 'build'.

event.initEvent('build', true, true);

// Listen for the event.

elem.addEventListener('build', function (e) {

// e.target matches elem}, false);

// target can be any Element or other EventTarget.

elem.dispatchEvent(event);

## 257）[require-ensure](http://webpack.github.io/docs/api-in-modules.html#require-ensure)

**require-ensure**

**说明:** require.ensure在需要的时候才下载依赖的模块，当参数指定的模块都下载下来了（下载下来的模块还没执行），便执行参数指定的回调函数。require.ensure会创建一个chunk，且可以指定该chunk的名称，如果这个chunk名已经存在了，则将本 次依赖的模块合并到已经存在的chunk中，最后这个chunk在webpack构建的时候会单 独生成一个文件。

**语法:**require.ensure(dependencies: String[], callback: function([require]), [chunkName: String])

dependencies: 依赖的模块数组

callback: 回调函数，该函数调用时会传一个require参数

chunkName: 模块名，用于构建时生成文件时命名使用

**注意点：**requi.ensure的模块只会被下载下来，不会被执行，只有在回调函数使用require(模块名)后，这个模块才会被执行。

## 258)this指向

**this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁**，**实际上this的最终指向的是那个调用它的对象**

1.全局作用域或者普通函数中this指向全局对象window,全局环境中，this默认绑定到window

2.方法调用中谁调用this指向谁

3.在构造函数或者构造函数原型对象中this指向构造函数的实例

4.setTimeout的执行环境跟调用它的函数的执行环境是分离的，因此setTimeout调用的函数中的this关键字指向window或global对象

This4种绑定机制：

### 1.默认绑定

全局环境中，this默认绑定到window

[IIFE](http://www.cnblogs.com/xiaohuochai/p/5731016.html)立即执行函数实际上是函数声明后直接调用执行

闭包函数是独立调用，而不是方法调用，所以this默认绑定到window

### 2.隐式绑定

一般地，被直接对象所包含的函数调用时，也称为方法调用，this隐式绑定到该直接对象

**隐式丢失**

隐式丢失是指被隐式绑定的函数丢失绑定对象，从而默认绑定到window。这种情况容易出错却又常见.

### 3.显式绑定

通过call()、apply()、bind()方法把对象绑定到this上，叫做显式绑定。对于被调用的函数来说，叫做间接调用

javascript中新增了许多内置函数，具有显式绑定的功能，如数组的5个[迭代方法](http://www.cnblogs.com/xiaohuochai/p/5682621.html#anchor10)：map()(不能遍历undefined)、forEach()不能遍历undefined)、filter()、some()、every()

### 4.new绑定

如果函数或者方法调用之前带有关键字new，它就构成构造函数调用。对于this绑定来说，称为new绑定。

【1】构造函数通常不使用return关键字，它们通常初始化新对象，当构造函数的函数体执行完毕时，它会显式返回。在这种情况下，构造函数调用表达式的计算结果就是这个新对象的值。

【2】如果构造函数使用return语句但没有指定返回值，或者返回一个原始值，那么这时将忽略返回值，同时使用这个新对象作为调用结果。

【3】如果构造函数显式地使用return语句返回一个对象，那么调用表达式的值就是这个对象。

## 259)css长度单位

#### 绝对长度单位

Px

像素或许被认为是最好的“设备像素”，而这种像素长度和你在显示器上看到的文字屏幕像素无关。

In（英寸）

英寸是一个物理度量单位，但是在CSS领域，英寸只不过被直接映射成像素罢了。

1in == 96px

Cm

对于世界上大多数的人来说，厘米是比较熟悉有用的物理度量单位。它也映射成像素

1cm == 37.8px

Mm

毫米是个小数量级的物理度量单位。

1mm == 0.1cm == 3.78px

#### 相对字体的长度

Em

Em 是一个相对单位。起初[排版度量](http://en.wikipedia.org/wiki/Em_(typography))时是基于当前字体大写字母”M”的尺寸的。当改变font-family时，它的尺寸不会发生改变，但在改变font-size的大小时，它的尺寸就会发生变化。

Rem

Rem和em一样也是一个相对单位，但是和em不同的是rem总是相对于根元素的字体大小的单位，而不像em一样使用级联的方式来计算尺寸。这种相对单位使用起来更简单。

Points

Point是一个物理度量单位，1pt=1/72 in.在CSS之外point是最常用的尺寸类型（可能这就是css支持point的原因）。它在语言里也很常见“当然他们把这个重要的信息设置成了小八号字体”。

Ex

ex是一个基于当前字体的x字母高度度量的。ex度量时有时候根据字体自身的信息，有时候由浏览器指明是通过一个小写字形来度量，最糟糕的情况是，它被设置成0.5em。

Pica

Pica和points一样，只不过1pc=12pt。

ch

ch的内涵和x高度相似，只是ch是基于字符0的宽度的而不是基于字符x高度的。当font-family改变的时候ch也会随着改变。

#### 可视区百分比长度单位

vw

vw是可视区宽度单位。1vw等于可视区宽度的百分之一。vw单位跟百分比很相似，不同的是vw的值对所有的元素都一样，与他们父元素或父元素的宽度无关。有点像rem单位那样总是相对于根元素。

vh

vh和vw（viewport widht）单位一样，不同的是vh是相对于可视区的高度。

vmin

vmin的值是当前vw和vh中较小的值。在标准尺寸类型的使用实例中，和由自己确定屏幕大小的vw、vh单位相比，vmin是一个更有用的度量标准。

vmax

Vmax的值是vw和vh中的较大的值。

#### 离奇的长度单位

百分比

以百分比为单位的长度值是基于具有相同属性的父元素的长度值。例如，如果一个元素呈现的宽度是450px，子元素的宽度设为50%，那么子元素呈现的宽度为225px。

Turn

 角度单位 turn(圈)

## 260)currentColor

currentColor表示“当前的标签所继承的文字颜色”。

**1.颜色图文合体变化**

**.icon{background-color: currentColor;}**

**.link:hover { color: #333; }**

## 261)background是以下复合属性值组合的简写

background-color 设置颜色作为对象背景颜色  
background-image 设置图片作为[背景图片](http://www.divcss5.com/jiqiao/j369.shtml)  
background-repeat 设置背景平铺重复方向  
background-attachment 设置或检索背景图像是随对象内容滚动还是固定的。  
background-position 设置或检索对象的背景图像位置。

background:#000 url(图片地址) no-repeat left top

## 262）background-clip

|  |  |
| --- | --- |
| border-box | 此值为默认值，背景从border区域向外裁剪，也就是超出部分将被裁剪掉 |
| padding-box | 背景从padding区域向外裁剪，超过padding区域的背景将被裁剪掉； |
| content-box | 背景从content区域向外裁剪，超过context区域的背景将被裁剪掉； |

## 263)skew()

**skew(<angle> [, <angle>]);**包含两个参数值，分别表示X轴和Y轴倾斜的角度，如果第二个参数为空，则默认为0，参数为负表示向相反方向倾斜。

**skewX(<angle>);**表示只在X轴(水平方向)倾斜。正为逆时针旋转。

**skewY(<angle>);**表示只在Y轴(垂直方向)倾斜。正为顺时针旋转。

Eg:transtorm:skew(-45deg)

## 264）css filter

filter: none | blur() | brightness() | contrast() | drop-shadow() | grayscale() | hue-rotate() | invert() | opacity() | saturate() | sepia() | url();

|  |  |
| --- | --- |
| **Filter** | **描述** |
| none | 默认值，没有效果。 |
| blur(px) | 给图像设置高斯模糊。"radius"一值设定高斯函数的标准差，或者是屏幕上以多少像素融在一起， 所以值越大越模糊； 如果没有设定值，则默认是0；这个参数可设置css长度值，但不接受百分比值。 |
| brightness(%) | 给图片应用一种线性乘法，使其看起来更亮或更暗。如果值是0%，图像会全黑。值是100%，则图像无变化。其他的值对应线性乘数效果。值超过100%也是可以的，图像会比原来更亮。如果没有设定值，默认是1。 |
| contrast(%) | 调整图像的对比度。值是0%的话，图像会全黑。值是100%，图像不变。值可以超过100%，意味着会运用更低的对比。若没有设置值，默认是1。 |
| drop-shadow(h-shadow v-shadow blur spread color) | 给图像设置一个阴影效果。阴影是合成在图像下面，可以有模糊度的，可以以特定颜色画出的遮罩图的偏移版本。 函数接受<shadow>(在CSS3背景中定义)类型的值，除了"inset"关键字是不允许的。该函数与已有的box-shadow box-shadow属性很相似；不同之处在于，通过滤镜，一些浏览器为了更好的性能会提供硬件加速。<shadow>参数如下：  **<offset-x>** **<offset-y>** (必须)  这是设置阴影偏移量的两个 <length>值. **<offset-x>** 设定水平方向距离. 负值会使阴影出现在元素左边. **<offset-y>**设定垂直距离.负值会使阴影出现在元素上方。查看**<length>**可能的单位. **如果两个值都是0**, 则阴影出现在元素正后面 (如果设置了 <blur-radius> and/or <spread-radius>，会有模糊效果).  **<blur-radius>** (可选)  这是第三个code><length>值. 值越大，越模糊，则阴影会变得更大更淡.不允许负值 若未设定，默认是0 (则阴影的边界很锐利).  **<spread-radius>** (可选)  这是第四个 <length>值. 正值会使阴影扩张和变大，负值会是阴影缩小.若未设定，默认是0 (阴影会与元素一样大小).  注意: Webkit, 以及一些其他浏览器 不支持第四个长度，如果加了也不会渲染。  **<color>** (可选)  查看 <color>该值可能的关键字和标记。若未设定，颜色值基于浏览器。在Gecko (Firefox), Presto (Opera)和Trident (Internet Explorer)中， 会应用color**color**属性的值。另外, 如果颜色值省略，WebKit中阴影是透明的。 |
| grayscale(%) | 将图像转换为灰度图像。值定义转换的比例。值为100%则完全转为灰度图像，值为0%图像无变化。值在0%到100%之间，则是效果的线性乘子。若未设置，值默认是0； |
| hue-rotate(deg) | **给图像应用色相旋转。**"angle"一值设定图像会被调整的色环角度值。值为0deg，则图像无变化。若值未设置，默认值是0deg。该值虽然没有最大值，超过360deg的值相当于又绕一圈。 |
| invert(%) | 反转输入图像。值定义转换的比例。100%的价值是完全反转。值为0%则图像无变化。值在0%和100%之间，则是效果的线性乘子。 若值未设置，值默认是0。 |
| opacity(%) | 转化图像的透明程度。值定义转换的比例。值为0%则是完全透明，值为100%则图像无变化。值在0%和100%之间，则是效果的线性乘子，也相当于图像样本乘以数量。 若值未设置，值默认是1。该函数与已有的opacity属性很相似，不同之处在于通过filter，一些浏览器为了提升性能会提供硬件加速。 |
| saturate(%) | **转换图像饱和度。**值定义转换的比例。值为0%则是完全不饱和，值为100%则图像无变化。其他值，则是效果的线性乘子。超过100%的值是允许的，则有更高的饱和度。 若值未设置，值默认是1。 |
| sepia(%) | **将图像转换为深褐色。**值定义转换的比例。值为100%则完全是深褐色的，值为0%图像无变化。值在0%到100%之间，则是效果的线性乘子。若未设置，值默认是0； |
| url() | URL函数接受一个XML文件，该文件设置了 一个SVG滤镜，且可以包含一个锚点来指定一个具体的滤镜元素。  filter: url(svg-url#element-id) |
| initial | 设置属性为默认值，可参阅： [CSS initial 关键字](http://www.runoob.com/cssref/css-initial.html) |
| inherit | 从父元素继承该属性，可参阅：[CSS inherit 关键字](http://www.runoob.com/cssref/css-inherit.html) |

## 265)dl dt dd

<dl> <dt> <dd>是一组合标签,都是块级元素，使用了dt dd最外层就必须使用dl包裹，此组合标签我们也又叫[表格标签](http://www.divcss5.com/html/h380.shtml)，与[table表格](http://www.divcss5.com/html/h380.shtml)类似组合标签，故名我们也叫dl表格

**<dl>  
<dt>列表标题</dt>  
<dd>列表内容</dd>  
<dd>列表内容</dd>  
</dl>**

**语法解释：**  
首先dt和dd是放于dl标签内，标签dt与dd处于dl下相同级。也就是dt不能放入dd内，dd不能放入dt内。在dl下，dt与dd处于同级标签。DD标签可以若干。同时不能不加dl地单独使用dt标签或dd标签。我们实践使用dl dt dd标签最多地方，通常是具有标题，而标题下对应有若干列表简单的（栏目标题+对应标题列表）和标题对应下面有内容。

## 266）波浪下滑线

a{  
 background: -webkit-linear-gradient(-45deg, transparent 40%, red 0, red 60%, transparent 0)  
 ,-webkit-linear-gradient(45deg, transparent 40%, red 0, red 60%, transparent 0);  
 background-size: .2em .1em;  
 background-repeat: repeat-x;  
 background-position:0 100%,.11em 100%;  
}

## 267）clip: <shape> | auto | inherit

<shape>：shape是一个函数功能，当使用仅使用rect()属性;rect()需要设置四个值：top, right, bottom和left。他们之间需要用逗号隔开。

rect()和<top>和<bottom>指定偏移量是从元素盒子顶部边缘算起；<left>和<right>指定的偏移量是从元素盒子左边边缘算起。

## 268)事务的ACID

1.原子性

对数据修改操作要么全部执行，要么完全不执行。保证事务失败，能整体回滚。

1. 一致性

加锁，保证执行顺序。保证事务单元执行成功后才可见，上一个成功下一个才会进来。但这会造成并发下降。

1. 3.隔离性**（Isolation)**

以性能为理由，对强一致性的破坏。四种隔离级别：

几个概念：

**读脏数据:**一事务对数据进行了增删改,但未提交,有可能[回滚](https://baike.baidu.com/item/%E5%9B%9E%E6%BB%9A),另一事务却读取了未提交的数据。

**幻像读:**一事务对数据进行了新增操作,另一事务两次查询的数据不一致。

**序列化读写**，利用排它锁,[事务](https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1)只能一个一个执行，避免了[脏读](https://baike.baidu.com/item/%E8%84%8F%E8%AF%BB)、[不可重复读](https://baike.baidu.com/item/%E4%B8%8D%E5%8F%AF%E9%87%8D%E5%A4%8D%E8%AF%BB)、[幻读](https://baike.baidu.com/item/%E5%B9%BB%E8%AF%BB/153502)。

隔离级别定义事务处理数据读取操作的隔离程度.

**可重复读**，读写锁。读锁可以对写锁升级。它保证事务内的任何数据在事务过程中都不会被修改或者移除。（MySQL默认的隔离级别）

**读以提交**，加读写锁，读锁可以升级为写锁。

**读未提交**，只加写锁不加读锁。--读读并行，--读写并行，--写读并行

其它级别：

快照读:当读取数据时，可以保证[读操作](https://baike.baidu.com/item/%E8%AF%BB%E6%93%8D%E4%BD%9C/12733869)读取的行是事务开始时可用的最后提交版本。这种隔离级别使用的不是[共享锁](https://baike.baidu.com/item/%E5%85%B1%E4%BA%AB%E9%94%81)，而是行[版本控制](https://baike.baidu.com/item/%E7%89%88%E6%9C%AC%E6%8E%A7%E5%88%B6).

4.持久性

持久性，意味着在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

     即使出现了任何事故比如断电等，事务一旦提交，则持久化保存在数据库中.

**典型异常应对策略：**

业务属性不匹配：

回滚

系统宕机：

重启进入recovery模式

**事务调优：**

减少锁的覆盖范围。

增加锁上可并行的线程数。

选择正确的锁类型：

悲观锁:适合并发争抢比较严重的场景。

悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。

**乐观锁：**适合并发争抢不太严重的场景。

乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。

实现方式：1.使用数据版本（Version）记录机制实现，即为数据增加一个版本标识。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

1. 乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

事务是什么？让很多操作顺序发生。

网络带来的：

去中心化：无限扩展，无限的数据安全（复制无穷多份），无限的服务可用性（复制无穷多个节点）

网络失去的：共享数据的困难（解决：消息传递）

基于锁的事务：

2phase lock:两阶段锁

2phase commit:两阶段提交

为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法。参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

**XA事务**：XA协议采用两阶段提交方式来管理[分布式事务](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E4%BA%8B%E5%8A%A1)。XA接口提供资源管理器与事务管理器之间进行通信的标准接口。XA 规范主要定义了[事务](https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1)管理器(Transaction Manager)和局部[资源管理器](https://baike.baidu.com/item/%E8%B5%84%E6%BA%90%E7%AE%A1%E7%90%86%E5%99%A8)(Local Resource Manager)之间的接口.

异常处理：第三者（协调者）负责跨机提交。

关键问题：延迟变大

## 269)HAProxy

HAProxy提供高可用性、负载均衡以及基于TCP和HTTP应用的代理，支持虚拟主机，它是免费、快速并且可靠的一种解决方案。HAProxy特别适用于那些负载特大的web站点，这些站点通常又需要会话保持或七层处理。

## 270）web安全问题

1.xss(跨站脚本攻击)**Cross Site Scripting**

Web应用程序在将数据输出到网页的时候存在问题，导致攻击者可以将构造的恶意数据显示在页面的漏洞。因为跨站脚本攻击都是向网页内容中写入一段恶意的脚本或者HTML代码，故跨站脚本漏洞也被叫做HTML注入漏洞（HTML Injection）。跨站脚本漏洞是在客户端发动造成攻击，也就是说，利用跨站脚本漏洞注入的恶意代码是在用户电脑上的浏览器中运行的。

**解决：不要在允许位置插入不可信数据**

**在向HTML元素内容插入不可信数据前对HTML解码**

2.csrf(跨站点请求伪造)**Cross Site Request Forgery**

CSRF攻击将使用受害者在网站上的身份和特权来模拟他们并执行恶意活动或交易。攻击者将尝试利用其浏览器中存储登录cookie的用户。

**解决：**1） 验证 HTTP Referer。在 HTTP 协议的请求头部含有一个字段叫 Referer，它记录了本次请求的来源地址。只需校验Referer是否以本域作为来源，则可以判断这个请求的真伪。

1. 加密cookie信息。
2. 使用令牌。添加一个隐藏表单域记录随机的令牌，在求的参数中包含该令牌。
3. 在 HTTP 头中自定义属性。为了解决上一个方法设置Token比较麻烦的问题，可以将令牌放到 HTTP 头中自定义的属性里。
4. 在每次敏感操作都弹出对话框需要用户进行二次确认。
5. 服务器端将用户会话的失效时间设置得较短一些。
6. 用户自己养成习惯，在一个站点操作完成后，马上退出登录以撤销认证会话。

3.clickjacking（点击劫持）

4.sql注入 SQL Injection

就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

SQL注入的产生原因通常表现在以下几方面：①不当的类型处理；②不安全的[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE/5947370)库配置；③不合理的查询集处理；④不当的错误处理；⑤转义字符处理不合适；⑥多个提交处理不当。

**解决：1.永远不要信任用户的输入。对用户的输入进行校验，可以通过**[**正则表达式**](https://baike.baidu.com/item/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F)**，或限制长度；对单引号和双"-"进行转换等。**

**2.永远不要使用动态拼装sql，可以使用参数化的sql或者直接使用**[**存储过程**](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E8%BF%87%E7%A8%8B)**进行数据查询存取。**

**3.永远不要使用**[**管理员**](https://baike.baidu.com/item/%E7%AE%A1%E7%90%86%E5%91%98)**权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。**

**4.不要把机密信息直接存放，加密或者hash掉密码和敏感的信息。**

**5.应用的异常信息应该给出尽可能少的提示，最好使用自定义的**[**错误信息**](https://baike.baidu.com/item/%E9%94%99%E8%AF%AF%E4%BF%A1%E6%81%AF)**对原始错误信息进行包装**

**6.sql注入的检测方法一般采取辅助**[**软件**](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6)**或网站平台来检测，软件一般采用sql注入检测工具jsky，网站平台就有亿思**[**网站安全**](https://baike.baidu.com/item/%E7%BD%91%E7%AB%99%E5%AE%89%E5%85%A8)**平台检测工具。MDCSOFT SCAN等。采用**[**MDCSOFT-IPS**](https://baike.baidu.com/item/MDCSOFT-IPS)**可以有效的防御SQL注入，XSS攻击等。**

盲注：就是在服务器没有错误回显时完成的错误。

1. 命令执行：

应用有时需要调用一些执行系统命令的函数，如PHP中的system、exec、shell\_exec、

passthru、popen、proc\_popen等，当用户能控制这些函数中的参数时，就可以将恶意系统命令拼接到正常命令中，从而造成命令执行攻击，这就是命令执行漏洞。

**解决：**

尽量少用执行命令的函数或者直接禁用

参数值尽量使用引号包括

在使用动态函数之前，确保使用的函数是指定的函数之一

在进入执行命令的函数/方法之前，对参数进行过滤，对敏感字符进行转义

6.服务端请求伪造攻击（Server-side Request Forgery）

漏洞排名：

弱口令，sql注入，信息泄漏，命令执行

## 271）长屏布局，整屏布局

## 272)JSON Web Token（JWT）是一个非常轻巧的规范.

## 273)[] == [] false;{} == {} false;[] == ![] true

![] 为false;当"=="的一边是字符串或数字，另一边是对象的时候（数组也是对象），先把对象值转换为原始值再判断相等。

JavaScript里除了false自身以外只有5个假值，分别是“”，undefined, null, 0， NaN。

除了这5个假值以外，其他所有值转布尔类型都是true。一切对象都是真值，包括var a=new Boolean(false)。!a === false为true.

当两个值都是对象 (引用值) 时, 比较的是两个引用值在内存中是否是同一个对象. 因为此 [] 非彼 [], 虽然同为空数组, 确是两个互不相关的空数组, 自然 == 为 false;

## 274）html5shiv.js兼容html5标签

## 275)解决网站爬虫

封ip，分析User-Agent。

图片防盗链：

1.referer

http 协议中，如果从一个网页跳到另一个网页，http 头字段里面会带个 Referer。图片服务器通过检测 Referer 是否来自规定域名，来进行防盗链。

但是：可以通过改变Referer来解决。

1. URL签名

开发者在URL中加入签名信息，把该URL转给第三方实现授权访问。第三方用户只需要使用HTTP的GET请求访问此URL即可下载Object。

基于OSS的防盗链最佳实践点如下：

使用三级域名URL，例如referer-test.oss-cn-hangzhou.aliyuncs.com/aliyun-logo.png，安全性比绑定二级域名更高。三级域名方式能够提供Bucket级别的清洗和隔离，能够应对被盗链后的流量暴涨的情况，也能避免不同Bucket间的互相影响，最终提高业务可用性；

禁止搜索引擎爬取可以修改robots.txt

## 276)jQuery.fn.init()

jQuery.fn.init.prototype = jQuery.fn = jQuery.prototype;

创建jQuery对象，new jQuery.fn.init();

源码：jQuery = function( selector, context ) {

// The jQuery object is actually just the init constructor 'enhanced'

return new jQuery.fn.init( selector, context, rootjQuery );

},

问:为什么不return new jQuery();

var jQuery = function(selector, context) {

return new jQuery();

}//很明显的问题，死循环了！

问：为什么不return jQuery.fn.init();

通过实例init函数，每次都构建新的init实例对象，来分隔this,避免交互混淆。就是使用new每个对象都有自己的this作用域。

## 277）正则

^ 匹配一个输入或一行的开头，/^a/匹配"an A"，而不匹配"An a"   
$ 匹配一个输入或一行的结尾，/a$/匹配"An a"，而不匹配"an A"   
\* 匹配前面元字符0次或多次，/ba\*/将匹配b,ba,baa,baaa

. **匹配除换行符 \n 之外的任何单字符**+ 匹配前面元字符1次或多次，/ba\*/将匹配ba,baa,baaa   
? 匹配前面元字符0次或1次，/ba\*/将匹配b,ba   
(x) 匹配x保存x在名为$1...$9的变量中   
x|y 匹配x或y   
{n} 精确匹配n次   
{n,} 匹配n次以上   
{n,m} 匹配n-m次   
[xyz] 字符集(character set)，匹配这个集合中的任一一个字符(或元字符)   
[^xyz] 不匹配这个集合中的任何一个字符   
[\b] 匹配一个退格符   
\b 匹配一个单词的边界   
\B 匹配一个单词的非边界   
\cX 这儿，X是一个控制符，/\cM/匹配Ctrl-M   
\d 匹配一个字数字符，/\d/ = /[0-9]/   
\D 匹配一个非字数字符，/\D/ = /[^0-9]/   
\n 匹配一个换行符   
\r 匹配一个回车符   
\s 匹配一个空白字符，包括\n,\r,\f,\t,\v等   
\S 匹配一个非空白字符  
\t 匹配一个制表符   
\v 匹配一个重直制表符   
\w 匹配一个可以组成单词的字符(alphanumeric，这是我的意译，含数字)，包括下划线

\W 匹配一个不可以组成单词的字符

\1表示重复正则第一个圆括号内匹配到的内容

**match 方法**   
使用正则表达式模式对字符串执行查找，并将包含查找的结果作为数组返回。

stringObj.match(rgExp)

**exec 方法**用正则表达式模式在字符串中查找，返回一个数组，其中存放匹配的结果，如果匹配失败，返回null。   
rgExp.exec(str)

**search 方法**  
返回与正则表达式查找内容匹配的第一个子字符串的位置（偏移位）。   
stringObj.search(rgExp)

去掉连续重复的字符：s.replace(/(.).\*(\1)/g,'$1');

## 278）无名天地之始，有名万物之母

## 279）验证码认证

服务端保存一份，客户端输入验证码返给服务端，服务端对比。

服务端不保存，全部发给客户端，客户端输入验证码，并把服务端刚发给他的一起返回，服务端对比。

## 280）负载均衡算法：

**1、轮询法**

　　将请求按顺序轮流地分配到后端服务器上，它均衡地对待后端的每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

**2、随机法**

     通过系统的随机算法，根据后端服务器的列表大小值来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，

其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。

**3、源地址哈希法**

     源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

**4、加权轮询法**

　　不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。

**5、加权随机法**

     与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

**6、最小连接数法**

     最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前

## 281）border渐变

可以通过border-image:linear-gradient().border-color不支持渐变色。

## 282）带预检(Preflighted)的跨域请求。当HTTP请求出现以下两种情况时，浏览器认为是带预检(Preflighted)的跨域请求： 1). 除GET、HEAD和POST(only with application/x-www-form-urlencoded, multipart/form-data, text/plain Content-Type)以外的其他HTTP方法。 2). 请求中出现自定义HTTP头部。 带预检(Preflighted)的跨域请求需要浏览器在发送真实HTTP请求之前先发送一个OPTIONS的预检请求，检测服务器端是否支持真实请求进行跨域资源访问，真实请求的信息在OPTIONS请求中通过Access-Control-Request-Method Header和Access-Control-Request-Headers Header描述。服务器端接到预检请求后，根据资源权限配置，在响应头中放入Access-Control-Allow-Origin Header、Access-Control-Allow-Methods和Access-Control-Allow-Headers Header，分别表示允许跨域资源请求的域、请求方法和请求头。此外，服务器端还可以加入Access-Control-Max-Age Header，允许浏览器在指定时间内，无需再发送预检请求进行协商，直接用本次协商结果即可。浏览器根据OPTIONS请求返回的结果来决定是否继续发送真实的请求进行跨域资源访问。这个过程对真实请求的调用者来说是透明的。

**以下是CORS协议规定的HTTP头，用来进行浏览器发起跨域资源请求时进行协商：**  
1. Origin。HTTP请求头，任何涉及CORS的请求都必需携带。  
2. Access-Control-Request-Method。HTTP请求头，在带预检(Preflighted)的跨域请求中用来表示真实请求的方法。  
3. Access-Control-Request-Headers。HTTP请求头，在带预检(Preflighted)的跨域请求中用来表示真实请求的自定义Header列表。  
4. Access-Control-Allow-Origin。HTTP响应头，指定服务器端允许进行跨域资源访问的来源域。可以用通配符\*表示允许任何域的JavaScript访问资源，但是在响应一个携带身份信息(Credential)的HTTP请求时，Access-Control-Allow-Origin必需指定具体的域，不能用通配符。  
5. Access-Control-Allow-Methods。HTTP响应头，指定服务器允许进行跨域资源访问的请求方法列表，一般用在响应预检请求上。  
6. Access-Control-Allow-Headers。HTTP响应头，指定服务器允许进行跨域资源访问的请求头列表，一般用在响应预检请求上。  
7. Access-Control-Max-Age。HTTP响应头，用在响应预检请求上，表示本次预检响应的有效时间。在此时间内，浏览器都可以根据此次协商结果决定是否有必要直接发送真实请求，而无需再次发送预检请求。

8. Access-Control-Allow-Credentials。HTTP响应头，凡是浏览器请求中携带了身份信息，而响应头中没有返回Access-Control-Allow-Credentials: true的，浏览器都会忽略此次响应。

## IMG_256283)两个栈实现一个队列

## 284)Js new的过程分为四个步骤：

创建一个新对象

把构造函数里的this替换为这个新对象

执行构造函数

返回这个新对象

C++

获得一块内存空间、调用构造函数、返回正确的指针。

## 285)编码函数

escape() 函数可对字符串进行编码，这样就可以在所有的计算机上读取该字符串。

escape("?!=()#%&")；

%3F%21%3D%28%29%23%25%26

**encodeURI()函数**

encodeURI() 函数可把字符串作为 URI 进行编码。

encodeURI(URIstring)返回值 为URIstring 的副本，其中的某些字符将被十六进制的转义序列进行替换。

**encodeURIComponent() 函数，**encodeURIComponent() 函数可把字符串作为 URI 组件进行编码。

## 286）http状态码

HTTP状态码（HTTP Status Code）是用以表示网页[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8)HTTP响应状态的3位数字代码。它由 RFC 2616 规范定义的，并得到RFC 2518、RFC 2817、RFC 2295、RFC 2774、RFC 4918等规范扩展。

[消息（1字头）](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#1)代表请求已被接受，需要继续处理。这类响应是临时响应

▪ [100 Continue](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#1_1)

▪ [101 Switching Protocols](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#1_2)

▪ [102 Processing](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#1_3)

[成功（2字头）](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2)代表请求已成功被服务器接收、理解、并接受

▪ [200 OK](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_1)

▪ [201 Created](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_2)

▪ [202 Accepted](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_3)

▪ [203 Non-Authoritative Information](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_4)

▪ [204 No Content](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_5)

▪ [205 Reset Content](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_6)

▪ [206 Partial Content](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_7)

▪ [207 Multi-Status](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#2_8)

[重定向（3字头）](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3)这类状态码代表需要客户端采取进一步的操作才能完成请求。

▪ [300 Multiple Choices](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_1)

▪ [301 Moved Permanently](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_2)

▪ [302 Move temporarily](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_3) 请求的资源**临时**从不同的 URI响应请求。就是地址还有可能改。

▪ [303 See Other](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_4)如果原来的请求是POST，Location头指定的重定向目标文档应该通过GET提取（HTTP 1.1新）

▪ [304 Not Modified](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_5)

▪ [305 Use Proxy](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_6)

▪ [306 Switch Proxy](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_7)

▪ [307 Temporary Redirect](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#3_8) 对于POST请求，表示请求还没有被处理，客户端应该向 Location里的URI重新发起POST请求。

4 [请求错误（4字头）](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4)状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理。

▪ [400 Bad Request](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_1)

▪ [401 Unauthorized](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_2) 当前请求需要用户验证。该响应必须包含一个适用于被请求资WWW-Authenticate 信息头用以询问用户信息。

▪ [402 Payment Required](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_3)

▪ [403 Forbidden](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_4) 服务器已经理解请求，但是拒绝执行它。

▪ [404 Not Found](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_5)

▪ [405 Method Not Allowed](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_6)

▪ [406 Not Acceptable](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_7)

▪ [407 Proxy Authentication Required](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_8)

▪ [408 Request Timeout](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_9)

▪ [409 Conflict](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_10)

▪ [410 Gone](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_11)

▪ [411 Length Required](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_12)

▪ [412 Precondition Failed](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_13)

▪ [413 Request Entity Too Large](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_14)

▪ [414 Request-URI Too Long](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_15)

▪ [415 Unsupported Media Type](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_16)

▪ [416 Requested Range Not Satisfiable](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_17)

▪ [417 Expectation Failed](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_18)

▪ [422 Unprocessable Entity](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_19)

▪ [423 Locked](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_20)

▪ [424 Failed Dependency](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_21)

▪ [425 Unordered Collection](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_22)

▪ [426 Upgrade Required](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_23)

▪ [449 Retry With](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_24)

▪ [451Unavailable For Legal Reasons](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#4_25)

5 [服务器错误（5、6字头）](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5)这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理

▪ [500 Internal Server Error](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_1)

▪ [501 Not Implemented](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_2) [未实现](http://blog.csdn.net/voipmaker/article/details/6333267)

▪ [502 Bad Gateway](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_3)

▪ [503 Service Unavailable](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_4)

▪ [504 Gateway Timeout](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_5)

▪ [505 HTTP Version Not Supported](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_6)

▪ [506 Variant Also Negotiates](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_7)

▪ [507 Insufficient Storage](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_8)

▪ [509 Bandwidth Limit Exceeded](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_9)

▪ [510 Not Extended](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_10)

▪ [600 Unparseable Response Headers](https://baike.baidu.com/item/HTTP%E7%8A%B6%E6%80%81%E7%A0%81/5053660?fr=aladdin#5_11)

302与301区别：

301表示A资源已经永远的被删除了。302表示A资源还在，重定向只是临时的从A跳到B。尽量使用301.302会有URL劫持。也就是内容是你的，网址确是别人的。

## 287)head请求方式：

只返回响应首部。HEAD方法跟GET方法相同，只不过服务器响应时不会返回消息体。一个HEAD请求的响应中，HTTP头中包含的元信息应该和一个GET请求的响应消息相同。这种方法可以用来获取请求中隐含的元信息，而不用传输实体本身。也经常用来测试超链接的有效性、可用性和最近的修改。

HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

|  |  |  |
| --- | --- | --- |
| 1 | GET | 请求指定的页面信息，并返回实体主体。 |
| 2 | HEAD | 类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| 5 | DELETE | 请求服务器删除指定的页面。 |
| 6 | CONNECT | HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。 |
| 7 | OPTIONS | 允许客户端查看服务器的性能。 |
| 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断 |

## 288）用递归和栈操作逆序一个栈

每次返回栈底元素，直到栈空，再把弹出的数据压栈。

**function** getLastElement(stack){  
 **var** result = stack.pop();  
 **if**(stack.length === 0){  
 **return** result;  
 }**else** {  
 **var** i = getLastElement(stack);  
 console.log(i);  
 stack.push(result);  
 **return** i;  
 }  
}  
**function** reverse(stack) {  
 **if**(stack.length === 0){  
 **return** ;  
 }  
 **var** i = getLastElement(stack);  
 reverse(stack);  
 stack.push(i);  
}

## 289）tcp(传输控制协议)与udp(用户数据报协议)区别

1.tcp首部20字节，udp首部8字节。

2.面向连接与无连接

3.tcp提供可靠的报文传输，udp不提供

4.TCP面向字节流，UDP是面向报文的

5.TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

6.TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

基于tcp的协议：ftp 20/21(20为保留端口),smtp 25,http 80,https 443,

基于udp的协议:SNMP(简单网络管理协议)，dns,NFS（网络文件系统,也有tcp版）

补充：以太网(Ethernet)数据帧的长度必须在46-1500字节之间，1500字节被称为链路层的MTU(最大传输单元)，ip包首部20字节。tcp socket为流套接字SOCK\_STREAM，udp为数据报套接字SOCK\_DGRAM，原始套接字SOCK\_RAW.ip也是无连接，不可靠的。面向无连接==无连接

## 290）浮点数存储

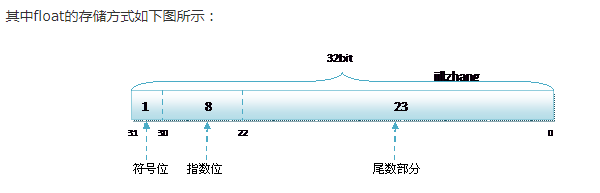
浮点类型的数据采用单精度类型（float）和双精度类型(double)来存储，float数据占用32bit,double数据占用64bit

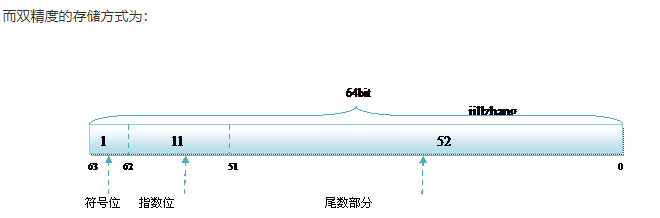
无论是单精度还是双精度在存储中都分为三个部分：

1.符号位(Sign) : 0代表正，1代表为负

2.指数位（Exponent又叫：阶码）:用于存储科学计数法中的指数数据，并且采用移位存储.

3.尾数部分（Mantissa）：尾数部分

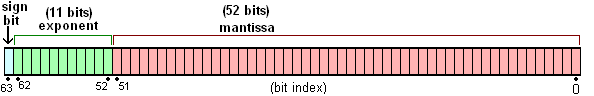




Javascript中所有的数字都是以IEEE-754标准格式表示。范围(安全数)+/-2^53次方（不含两个端点），超过这个范围，会有**两个或更多整数的双精度表示是相同的；**反过来说，超过这个范围，**有的整数是无法精确表示的**，只能**round到与它相近的浮点数**（说到底就是科学计数法）表示，这种情况下叫做**不安全整数**。

JS 中所有的数字类型，实际存储都是通过 8 字节 [double 浮点型](http://zh.wikipedia.org/wiki/%E9%9B%99%E7%B2%BE%E5%BA%A6%E6%B5%AE%E9%BB%9E%E6%95%B8) 表示的。

double 浮点数结构如下:

1 位符号位

11 位指数位

52 位尾数位

float的精度为2^23 = 8388608，一共七位,6~7位有效数字，double的精度为2^52 = 4503599627370496，一共16位,15~16位。

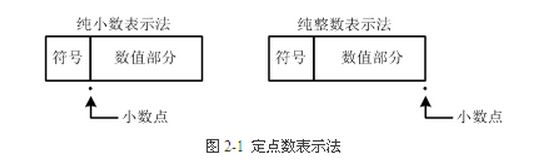
float的范围为-2^128 ~ +2^128，也即-3.40E+38 ~ +3.40E+38；double的范围为-2^1024 ~ +2^1024，也即-1.79E+308 ~ +1.79E+308。

E=255 并且尾数M全是0，则这个数的真值为±∞

E = 255 并且 M 不是0，则这不是一个数（NaN）

**定点数**

所谓定点格式，即约定机器中所有数据的小数点位置是固定不变的。通常将定点数据表示成纯小数或纯整数，为了将数表示成纯小数，通常把小数点固定在数值部分的最高位之前；而为了将数表示成纯整数，则把小数点固定在数值部分的最后面，如下图所示：



**浮点数**

这种表达方式利用科学计数法来表达实数，即用一个尾数(Mantissa，尾数有时也称为有效数字，它实际上是有效数字的非正式说法)，一个基数(Base)，一个指数(Exponent)以及一个表示正负的符号来表达实数

**IEEE浮点数**

两种基本的浮点格式：单精度和双精度。其中单精度格式具有24位有效数字(即尾数)精度，总共占用32位；双精度格式具有53位有效数字(即尾数)精度，总共占有64位。

## 291)前端缓存

为了加速访问减小服务器压力，浏览器在用户磁盘上对最近请求过的文档进行存储，当访问者再次请求这个页面时，浏览器就可以从[本地磁盘](https://baike.baidu.com/item/%E6%9C%AC%E5%9C%B0%E7%A3%81%E7%9B%98)显示文档，这样就可以加速页面的阅览。默认情况下，浏览器都会使用缓存数据。

**能使用缓存的原因：数据的局部性特征。**

Cache-Control

**其缓存指令对于前段常用的有如下no-cache、no-store、max-age这几个值；**

**public :**

**指定响应可以在代理缓存中被缓存，于是可以被多用户共享。**

**private :**

**响应只能在私有缓存中被缓存，不能放在代理缓存上。对一些用户信息敏感的资源，通常需要设置为private。**

**no-cache：**

表面意为“数据内容不被缓存”，而实际数据是被缓存到本地的，只是每次请求时候直接绕过缓存这一环节直接向服务器请求最新资源，由于浏览器解释不一样，

例如ie中我们设置了no-cache之后，请求虽然不会直接使用缓存，但是还会用缓存数据与服务器数据进行一致性检测(也就是说还是有几率会用到缓存的),

firefox中则完全无视no-cache存在，详细解释见no-store;

**no-store：**

指示缓存不存储此次请求的响应部分。

**max-age：**

例如Cache-control: max-age=3；表示此次请求成功后3秒之内发送同样请求不会去服务器重新请求，而是使用本地缓存；同样我们如果设置max-age=0表示立即抛弃缓存直接发送请求到服务器

一致性检测分为两种方式：1.检测日期是否过期，检测资源是否更新；

**Response:**

**Etag：**

它根据实体内容生成的一段hash字符串（类似于MD5或者SHA1之后的结果），可以标识资源的状态。 当资源发送改变时，ETag也随之发生变化。

引入**Etag的目的：**

Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间

如果某些文件会被定期生成，当有时内容并没有任何变化，但Last-Modified却改变了，导致文件没法使用缓存。

有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形

Expires

不推荐使用Expires，它指定的是具体的过期日期而不是秒数。因为很多服务器跟客户端存在时钟不一致的情况。Expires 是HTTP 1.0的东西，现在默认浏览器均默认使用HTTP 1.1，所以它的作用基本忽略。

禁用缓存：

方法一：

可以在meta标签标明<meta http-equiv="pragma" content="no-cache">

<meta http-equiv="cache-control" content="no-cache">

<meta http-equiv="expires" content="0">

方法二：

也可以动态去setRequestHeader，强制不用缓存设置组合如下：

cache-control='no-cache,no-store'

pragma='no-cache'

if-modified-since=0;

方法三：

请求端设置if-modified-since为已经过期的某个时间，可以是几年前或者几十年前。

方法四：

服务端设置Expires为过期某个时间，例如[**PHP**](http://lib.csdn.net/base/php)中header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");

实际开发中如果需要一致性检测则尽量去配合Etag以及last-Modified去进行比较然后返回使用缓存还是新数据;这个有点偏服务器端了，不再赘述。

**请求流程**

**Etag由服务器端生成**，客户端通过If-Match或者说If-None-Match这个条件判断请求来验证资源是否修改。常见的是使用If-None-Match.请求一个文件的流程可能如下：

====第一次请求===

1.客户端发起 HTTP GET 请求一个文件；

2.服务器处理请求，返回文件内容和一堆Header，当然包括Etag(例如"2e681a-6-5d044840")(假设服务器支持Etag生成和已经开启了Etag).状态码200

====第二次请求===

1.客户端发起 HTTP GET 请求一个文件，注意这个时候客户端同时发送一个If-None-Match头，这个头的内容就是第一次请求时服务器返回的Etag：2e681a-6-5d044840

2.服务器判断发送过来的Etag和计算出来的Etag匹配，因此If-None-Match为False，不返回200，**返回304，客户端继续使用**[**本地缓存**](https://baike.baidu.com/item/%E6%9C%AC%E5%9C%B0%E7%BC%93%E5%AD%98)；

流程很简单，问题是，如果服务器又设置了Cache-Control:max-age和Expires呢，怎么办？

答案是**同时使用**，也就是说在完全匹配If-Modified-Since和If-None-Match即检查完修改时间和Etag之后，服务器才能返回304.(不要陷入到底使用谁的问题怪圈)

**注意：不管怎么样的算法，在服务器端都要进行计算，计算就有开销，会带来性能损失。因此为了榨干这一点点性能，不少网站完全把Etag禁用了(比如Yahoo!)，这其实不符合HTTP/1.1的规定，因为HTTP/1.1总是鼓励服务器尽可能的开启Etag。**

## 292)死锁

是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

### ****产生死锁的四个必要条件：****

**（1） 互斥条件：一个资源每次只能被一个进程使用。  
（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。  
（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。  
（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。**

### 死锁预防

1.破坏互斥条件：资源不被一个进程独占使用，允许同时访问。

2.破坏占有并等待条件：

创建进程时，要求它申请所需的全部资源，系统或满足其所有要求，或什么也不给它。这是所谓的 “ 一次性分配”。

要求每个进程提出新的资源申请前，释放它所占有的资源。

3.破坏“不可抢占”条件

允许对资源实行抢夺。

4.破坏“循环等待”条件

破坏“循环等待”条件的一种方法，是将系统中的所有资源统一编号，进程可在任何时刻提出资源申请，但所有申请必须按照资源的编号顺序（升序）提出。这样做就能保证系统不出现死锁。

### 死锁避免：

死锁预防是设法至少破坏产生死锁的四个必要条件之一,严格的防止死锁的出现,死锁避免则不那么严格的限制产生死锁的必要条件的存在

#### ****银行家算法：****

        所谓银行家算法，是指在分配资源之前先看清楚，资源分配后是否会导致系统死锁。**如果会死锁，则不分配，否则就分配。**

#### ****安全序列****

是指系统中的所有进程能够按照某一种次序分配资源，并且依次地运行完毕，这种进程序列{P1，P2，...，Pn}就是安全序列。如果存在这样一个安全序列，则系统是安全的；如果系统不存在这样一个安全序列，则系统是不安全的。

### 死锁恢复：

1.重启 2.撤销进程 3.进程回退 4.按照某种原则逐个选择死锁进程进行撤销。

### 死锁检测：

通过检查锁的持有者与等待者是否会形成循环等待图来判定死锁

补充：发生的死锁后，JDK自带了两个工具(jstack和JConsole)，可以用来监测分析死锁的发生原因。

## 293)协议分层的优缺点

协议分层的原理（分层原理）：设计分层是为了目的站底n层收到正是源站第n层所发送的对象。

允许设计者只关注一层，而不必关注其它层。

缺点：可能使效率降低。分层使传输层无法知道互联网模块如何转发通信量，使传输层不能优化传输.

优点：

1. 各层之间是独立的。
2. 灵活性好。当任何一层发生变化时，只要层间接口关系保持不变，则在这层以上或以下各层均不受影响.
3. 结构上可分割开。各层都可以采用最合适的技术来实现。

4）易于实现和维护。

5）能促进标准化工作。因为每一层的功能及其所提供的服务都已有了精确的说明。

## 294)js string

字符串类型是所有有限的零个或多个16 位无符号整数值（“元素”）的有序序列。

中文以及全角的字符在任何地方都是占两个字节，英文和数字及半角的字符都占一个字节。

[**Python**](http://lib.csdn.net/base/python)中的字符串采用了intern机制，会自动intern.

也就是说，值相同的字符串对象只会保存一份，是共用的，这也决定了字符串必须是不可变对象。

## 295)height:100%无效

因为页面并没有缺省的高度值，所以，当你让一个元素的高度设定为百分比高度时，无法根据获取父元素的高度，也就无法计算自己的高度。换句话说，父元素的高度只是一个缺省值：height: auto;。当你要求浏览器根据这样一个缺省值来计算百分比高度时，只能得到undefined的结果。也就是一个null值，浏览器不会对这个值有任何的反应。

现在你给了这个div的高度为100%，它有两个父元素<body>和<html>。为了让你的div的百分比高度能起作用，你必须设定<body>和<html>的高度。

<html style="height: 100%;">

<body style="height: 100%;">

<div style="height: 100%;">

<p>

这样这个div的高度就会100%了

</p>

</div>

</body></html>

## 296)childNodes,children

childNodes：获取子元素的节点，包括文本节点、元素节点、注释节点；就是所有孩子节点。

children：获取子元素的元素节点（只包括元素节点）；接下来一层节点。

## 297)JS nodeType返回类型

元素节点的 nodeName 是标签名称  
属性节点的 nodeName 是属性名称  
文本节点的 nodeName 永远是 #text  
文档节点的 nodeName 永远是 #document

最重要的节点类型是：

|  |  |
| --- | --- |
| **元素类型** | **节点类型** |
| 元素 | 1 |
| 属性 | 2 |
| 文本 | 3 |
| 注释 | 8 |
| 文档 | 9 |

元素节点：

var d = document.getElementById("john");

属性节点：

<td id="john" name="myname">John</td>

var d = document.getElementById("john").getAttributeNode("name");

文本节点：

<td>Doe</td>

var d = document.getElementsByTagName("td")[0].firstChild

对文本节点可以通过node.textContent = “”替换文本内容。

## 298)localStorage,sessionStorage

方法：localStorage.getItem(key) ，localStorage.setItem(key,value) ,localStorage.removeItem(key),清空全部数据：localStorage.clear()

localStorage只能存储字符串。

读取速度与值大小无关。与读取次数有关。

**sessionStorage**一旦窗口或标签页被永久关闭了，那么所有通过sessionStorage存储的数据也就被清空了。

IE 9 > 4999995 + 5 = 5000000

firefox 22.0 > 5242875 + 5 = 5242880

chrome 28.0 > 2621435 + 5 = 2621440

Chrome >30.0 4.98M

safari 5.1 > 2621435 + 5 = 2621440

opera 12.15 > 5M （超出则会弹出允许请求更多空间的对话框）

Android 4.4-6.0 4.98M

Android 7.0 2.49M

cookie多达4097个字节。

Firefox每个域名cookie限制为50个。

Opera每个域名cookie限制为30个。

Chrome:每个域为53个

## 299)Array.isArray

Es5已支持，判断是否是数组

Array.isArray = function(arg) {

return Object.prototype.toString.call(arg) === '[object Array]';

};

## 300)diff算法

Web界面由DOM树来构成，当其中某一部分发生变化时，其实就是对应的某个DOM节点发生了变化。在React中，构建UI界面的思路是由当前状态决定界面。前后两个状态就对应两套界面，然后由React来比较两个界面的区别，这就需要对DOM树进行Diff算法分析。

**不同节点类型的比较**

在树中的同一位置前后输出了不同类型的节点，React直接删除前面的节点，然后创建并插入新的节点。

React在同一个位置遇到不同的组件时，也是简单的销毁第一个组件，而把新创建的组件加上去。

DOM Diff算法究竟是如何工作的，其复杂度控制在了O（n）；

## 301)ip协议

不可靠，无连接协议。分组可丢失，重复，延迟，不按次序。

头部长20字节。总长度16位。

路由查找下一站，查找目标是最长前缀匹配（LPM）。为了避免低效率搜索，无分类使用二叉线索的变形。

## 302)垃圾回收与算法

Java虚拟机在执行字节码时，把字节码解释成具体平台上的[机器指令](https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E6%8C%87%E4%BB%A4)执行。这就是Java的能够“一次编译，到处运行”的原因。

算法：

### 判断对象是否死亡：

1.引用计数

给每个对象添加引用计数器，有地方引用就加一，引用失效就减一。计数器为0，对象不可使用。

强引用：Object obj = new Object();只要强引用在永不会被回收。

软引用：描述有用但不是必须的对象。

弱引用：描述非必须对象，比软引用要弱。只能生存到下一次垃圾搜集之前。

虚引用：最弱的引用关系。设置虚引用的唯一目的对象被回收之前通知系统。

2.可达性分析算法

Gc root做为起始点，从这些节点向下搜索，搜索路径为引用链，当对象到gc root没有引用链相链。至少要经历两次标记，才会宣布死亡。

Java中可作为gc root的对象：

虚拟机中引用对象。

方法区中类静态属性引用的对象。

方法区常量引用对象。

本地方法栈中JNI引用对象。

### 垃圾回收算法：

1. 标记-清除 mark-sweep

两个阶段：标记要回收的对象，然后统一回收。

缺点：标记和清除两个过程效率不高。会产生大量碎片。

2.复制算法

将内存分为大小相等的两块，每次只使用其中一块。当一块用完，将存活对象复制到另一块。然后把使用过的空间清理掉。不用考虑内存碎片化，移动指针即可。

缺点：内存缩小为一半。对象存活率较高要进行很多复制操作。

3.标记-整理：

标记要回收的对象，同标记-清除。让所有存活对象向一端移动，清除端边界以外的内存。

1. 分代收集算法

根据对象存活的周期不同将内存划分为几块。Java分新生代大量对象死去用复制算法。老年代，用标记-清理或标记-整理

**Hotspot:**

主流Java虚拟机使用准确式GC,OopMap数据结构：存放对象引用。在OopMap协助下，准确完成GC Root枚举。安全点：gc只在安全点才暂停。安全区域：在代码片段之外，应用关系不会发生变化。

### 垃圾收集器：

Serial:单线程，新生代收集器。很老了，有停顿。

Parnew:Serial多线程版本。

Parallel scavenge收集器：新生代收集器。复制算法，并行的多线程收集器。无法与CMS配合工作。

Serial old:Serial老年代版本。也是单线程。使用标记-整理算法。

Parallel old:Parallel scavenge老年代版本。多线程，使用标记-整理算法。

CMS：以获取最短回收时间为目标的收集器。使用标记-清除算法。

步骤：初始标记，并发标记，重新标记，并发清除。

缺点：对cpu资源敏感。无法清除浮动垃圾。大量碎片空间。

G1：最新技术。是面向服务端应用的垃圾收集器。标记-整理算法。

特点：并发与并行。分代收集。空间整合。可预测停顿，可预测的停顿时间模型。

收集整个新生代与老年代。将整个内存分为多个大小相等的独立区域region.

步骤：初始标记，并发标记，最终标记，筛选回收。

### Java堆分配

**新生代，老年代**

**新生代：Eden区，From Survivor空间，To survivor空间，比例8：1：1**

**新生代的gc叫Minor GC,老年代 GC叫Major GC,对整个堆的GC叫Full GC.**

**Major GC比Minor GC慢10倍。一次Minor GC对象就长一岁，到达15（默认）就晋升老年代。**

eden区分配满的时候触发**Minor GC**，在进行MajorGC前一般都先进行了一次MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次MajorGC进行垃圾回收腾出空间。MajorGC采用标记—清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。当老年代也满了装不下的时候，就会抛出OOM（Out of Memory）异常。

每次使用Eden区，From Survivor空间，在垃圾回收时将存活对象拷贝到**To survivor空间**

新生代使用复制算法。

-Xmx    Java Heap最大值,默认值为物理内存的1/4,最佳设值应该视物理内存大小及计算机内其他内存开销而定;

-Xms    Java Heap初始值,Server端JVM最好将-Xms和-Xmx设为相同值,开发测试机JVM可以保留默认值;

-Xmn    Java Heap Young区大小,不熟悉最好保留默认值;

-Xss    每个线程的Stack大小,不熟悉最好保留默认值;

-XX:MaxNewSize=size

新生成对象能占用内存的最大值

-XX:MaxPermSize=64m

老生代对象能占用内存的最大值

### Java内存分配

**程序计数器：**当前线程执行的字节码的行号指示器。

**虚拟机栈：**线程私有，生命周期与线程相同。用于存储局部变量表，操作数，动态链接，方法出口。

**本地方法栈：**为虚拟机使用到的native方法服务。

**Java堆：**存放对象实例，

**方法区：**线程共享的内存区域，用于存储被虚拟机加载的类信息，常量，静态变量，即时编译后的代码等数据。

**运行时常量池：**是方法区的一部分。存放编译期生成的各种字面量和符号应用。

Java程序运行：源文件编译为字节码，字节码由java虚拟机解释执行。

### C++内存分配

**栈：**局部变量和函数参数

**堆：**使用new分配的内存块。

**自由存储区：**就是那些由 malloc 等分配的内存块，他和堆是十分相似的，不过它用 free 来 结束自己的生命的。

**全局/静态存储区：**全局变量和静态变量被分配到同一块内存中

**常量存储区：**存放的是常量，不允许修改

## 303)数据库面试题

**排序查询语句：order** **by** desc设定你想按哪个字段哪种方式来进行排序，再返回搜索结果

**union查询：**union 操作符用于连接两个以上的 SELECT 语句的结果组合到一个结果集合中。

**join查询：INNER JOIN（内连接,或等值连接）**：获取两个表中字段匹配关系的记录。

* **LEFT JOIN（左连接）：**获取左表所有记录，即使右表没有对应匹配的记录。
* **RIGHT JOIN（右连接）：** 与 LEFT JOIN 相反，用于获取右表所有记录，即使左表没有对应匹配的记录。

**分页查询：**limit  A,  B; A起点，B要查的行数；limit 20;每次查询只扫描20行.

es分页查询：\_search?scroll=5m&size=20;将会返回一个过期时间为5分钟scroll\_id.在请求GET /\_search/scroll?.request body为｛“scroll”:”5m”,”scroll\_id”=”xxxxxxc”｝

**数据库切分：**垂直拆分就是要把表按模块划分到不同**数据库**表中。水平拆分行，行数据拆分到不同表中，减小单张表体量。

**递归查询（父子查询）：**需要创建函数查询。

**数据库分片：**对数据通过一系列的切分规则将数据分布到不同的DB服务器上，通过路由规则路由访问特定的数据库，这样一来每次访问面对的就不是单台服务器了，而是N台服务器，这样就可以降低单台机器的负载压力。利用分库的规则，反向的路由到具体的DB，这个过程我们称之为“DB路由”。

**分片的替代方案**功能分区，复制，缓存和队列  
**mysql索引：**

MySQL中常见的索引

普通索引

唯一索引

主键索引

组合索引

全文索引

## 304)bind

Function.prototype.bind = **function**(){  
 **var** self = **this**;  
 **var** context = [].shift.call(arguments);// 需要绑定的this上下文  
 console.log(context);  
 **var** args = [].slice.call(arguments);// 剩余的参数转成数组  
 **return function** () {  
 // 并且组合两次分别传入的参数，作为新函数的参数  
 **return** self.apply(context,[].concat.call(args,[].slice.call(arguments)));  
 }  
}  
**var** altwrite = document.write;  
//altwrite("hello");  
/\*  
\* altwrite()函数改变this的指向global或window对象，导致执行时提示非法调用异常，  
\* \正确的方案就是使用bind()方法  
\* \*/  
altwrite.bind(document)("hello");

## 305)toFixed()

toFixed它是一个四舍六入五成双的诡异的方法，"**四舍六入五成双**"含义：对于位数很多的近似数，当有效位数确定后，其后面多余的数字应该舍去，只保留有效数字最末一位，这种修约（舍入）规则是“四舍六入五成双”，也即“4舍6入5凑偶”这里“四”是指≤4 时舍去，"六"是指≥6时进上，"五"指的是根据5后面的数字来定，当5后有数时，舍5入1；当5后无有效数字时，需要分两种情况来讲：①5前为奇数，舍5入1；②5前为偶数，舍5不进。（0是偶数）

**[Math.round()]是四舍五入，不保留为0的小数部分。如：1.10它得到1.1**

## 306)get和post区别

GET把参数包含在URL中，POST通过request body传递参数。(大多数)浏览器通常都会限制url长度在2K个字节

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

get是幂等的（idempotent），post不是，多次发出同样的POST请求后，其结果是创建出了若干的资源。

Post数据在payload里。

后台获取数据：get在req.query，post在req.body.(以express为例)。

GET产生一个TCP数据包;POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200(返回数据);而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。

但是1. GET与POST都有自己的语义，不能随便混用。

2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。

3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

补充：put与post区别

两者的区别主要在语义上。

在HTTP中，PUT被定义为idempotent（幂等）的方法，POST则不是，这是一个很重要的区别。

## 307)css属性默认值

Dispaly: inline

Position:static没有定位,元素出现在正常的流中,top,right,bottom,left 和 z-index 属性无效.

Flaot:none 不浮动

Background:not specified

background-color：transparent

Cursor:auto浏览器设置的光标。

## 308)tcp协议

**特点：**面向流，虚电路，带缓冲的传送，无结构流（没有特别照顾数据结构的数据流），全双工。

**可靠性：**数据不重复，不丢失。

**不丢包：**带重传的正面确认。发送方在发送下一个分组之前等待确认消息。发送方发送时发送分组会启用计时器，如果计时器超时则重传。

**自适应重传算法：**tcp监视每条连接的性能，由此推算超时时限，连接性能变化时，tcp修改时限值。Tcp计算传输过程样本往返时间。有新值就修改平均往返时间，一般计算的往返时间的加权平均值，作为往返时间估计值（RTT）。Timeout = β\*RTT

Tcp不应该为重传的报文更新往返时间估计，这就是**karn算法**。为了避免当延时增大后重传重复，karn要求对重传超时时限设置结合**计时器退避策略**。New\_time=r\*tiomout;通常r=2（以证明）,大多数时限都有上限。

高方差时延响应：协议要估计平均值还要估计方差。让方差代替β。

**滑动窗口：为了提高传输效率。（窗口大小占16位。）**

简单正面确认，会浪费大量网络带宽，应为在收到前一个分组确认之前会推迟下一个分组。

它允许发送方在等待一个确认到达时发送多个分组。当窗口第一个分组的确认收到后，它向后移动一格。因此窗口序号最小分组是没有收到确认的分组序列的第一个。由于tcp是双工的所以每一端维持两个窗口，一个连接四个窗口，一个接受数据滑动，一个接受数据滑动。

一个窗口定义三个指针，把已发送得到确认的分组和未确认的分组区分开。

窗口是可变大小：确认中包含一个窗口通知，表明接收方缓冲区当前可用空间大小。可变窗口提供流量控制。当窗口为0时，停止传输，应用清空缓存。两种例外：发送方发送紧急位为1的报文。为了避免非零窗口通知丢失可能造成的死锁，发送方定期发送非零窗口探寻报文。

当机器超载，这叫拥塞。

**拥塞控制：**

一个或多个交换点的数据报超载而导致延迟剧烈增加。路由器把数据报放在队列排队，路由器存储空间有限。饱和后丢弃报文。使用ICMP通知拥塞发生。

避免拥塞：使用慢开始和乘法减小。

拥塞窗口（cwnd）

数据接受方为滑动窗口，发送方为拥塞窗口。

**乘法减小：**发现报文丢失就拥塞窗口大小减半。留在窗口的报文，采用指数退避计算重传计时器。

**慢开始：**一条新连接开始通信时，或在一段时间拥塞后增加通信量，开始拥塞窗口只有一个报文，以后随着确认到达每次增加一个报文。（是由1，2，4，8来增加的。）

当拥塞窗口达到拥塞发生前窗口大小的一半（慢启动门限），tcp进入拥**塞避免阶段**，减小窗口增长速度。只有窗口所以报文段都被确认后，窗口大小才能增加1.这两种技术结合称加法增加乘法减小（AIMD）。

**快恢复：如果一个报文丢失，后续报文到达时，要求接收方产生一个ACK，该确认指明丢失的报文段在流中的位置。**

**快重传：**使用一连串的三个重复确认来激活一次重传，而不必等到计时器超时。

（网上的解释）如果tcp收到三个连续的ACK分组，此时不管是否过超时间隔则重传该分组，具体步骤如下:

1.将慢启动阈值ssthresh设为拥塞窗口(cwnd--可理解为发送数据包大小)的一半

2.对于连续三个重复的ACK报文，cwnd依次加1，重发分组

3.当新的ACK分组再次到来的时候，把拥塞窗口的大小设为慢启动阈值的大小

4.进入拥塞避免算法。

1，2,3个步骤为快速重传机制，为什么快，因为不需要等定时器时间到再重传

1,2,3,4则是快速恢复机制

**Reno算法**

Reno算法就是在慢启动和拥塞避免的基础上增加了快速重传和快速恢复算法，避免了在拥塞不严重的状况下，过大的减小拥塞窗口，降低TCP的传输效率

NewReno:针对有两个报文的情况进行了优化。

显式反馈机制：

**选择确认（SACK）：**允许接收方明确指出已收到哪些数据，还缺少哪些数据。

显式拥塞通知：当tcp报文通过互联网传播，路径上的路由使用IP首部中的一对比特位来记录拥塞。但需要了解拥塞的是发送方。因此接收方使用下一个ACK通知发送方拥塞发生。

**队首拥塞：分组必须按需传到接收端，如果中途一个分组未，后续分组要保存在接受端的tcp缓冲区等待丢失分组重传并到达接收端。**

分组到达的会存在无法预知的延迟变化，这个变化叫**抖动。**

**糊涂窗口综合征：**

当两端程序以不同速率工作，会产生严重性能问题。当发送端应用进程产生数据很慢、或接收端应用进程处理接收缓冲区数据很慢，或二者兼而有之；就会使应用进程间传送的报文段很小，特别是有效载荷很小； 极端情况下，有效载荷可能只有1个字节；传输开销有40字节(20字节的IP头+20字节的TCP头) 这种现象。

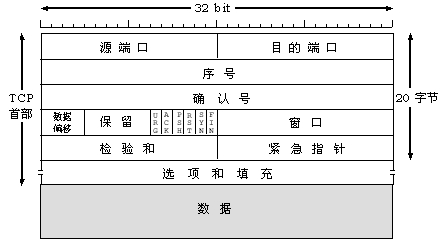
**避免糊涂窗口综合征**

**接收方：**窗口为0时，应用程序有收上去数据，但是并不立即会送窗口为1的通告，而是等待窗口大小满足一定的条件之后【**能够接收一个最大报文，或者缓冲区的一半**】，再来发送窗口通告，这样就不会产生小报文。

**推迟确认：**这表示当一个报文段到达时并不立即发送确认。接收端在确认收到的报文段之前一直等待，直到入缓存有足够的空间为止。延迟的确认防止了发送端的TCP滑动其窗口。当发送端的TCP发送完其数据后，它就停下来了。这样就防止了这种症状。迟延的确认还有另一个优点：它减少了通信量。接收端不需要确认每一个报文段。但它也有一个缺点，就是迟延的确认有可能迫使发送端重传其未被确认的报文段。可以用协议来平衡这个优点和缺点，tcp实现定义了确认的延迟不能超过500毫秒。

**发送方：**避免短报文，发送方采用的技术也具有自适应能力。发送方糊涂窗口避免技术称为自计时。它不是计算延时，而是利用确认到达来触发其余分组传输。等到数据足以填满一个最大长度报文段之后，在把缓冲区累计的数据发送出去。

**TCP报文格式：**

源端口和目的端口：各占2个字节，是运输层与应用层的服务接口。

**序号：**占4个字节。TCP连接传送的数据流中的每一个字节都被编上一个序号。首部中序号字段的值指的是**本报文段所发送的数据的第一个字节的序号**。

**确认号：**占4个字节，是**期望收到对方下一个报文段的数据的第一个字节的序号**。

**数据偏移：**占4 bit，它指出报文段的数据起始处距离TCP报文段的起始处有多远。实际上就是TCP报文段首部的长度。

保留：占6 bit，保留为今后使用。

紧急比特URG：当URG=1时，表明紧急指针有效。它告诉系统报文段中有紧急数据，应尽快传送。

**确认比特ACK**：ACK=1时确认号字段才有效，ACK=0时确认号字段无效。

推送比特PUSH：接收方接收到PUSH=1的报文段时会尽快的将其交付给接收应用进程，而不再等到整个接收缓存都填满后再向上交付。

**复位比特RST**：当RST=1时，表明TCP连接中出现严重差错，必须释放连接。复位比特还用来拒绝一个非法的报文段或拒绝打开一个连接。

**同步比特SYN：**在连接建立时用来同步序号。当SYN=1而ACK=0时，表明这是一个连接请求报文段。对方若同意建立连接，应在响应的报文段中使SYN=1和ACK=1。因此，SYN=1就表示这是一个连接请求或连接接收报文。

**终止比特FIN**：当FIN=1时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

窗口：占2个字节，用来控制对方发送的数据量，单位是字节，指明对方发送窗口的上限。

**校验和：**占2个字节，校验的范围包括首部和数据两个部分，计算校验和时需要在报文段前加上12字节的伪首部。

**紧急指针：**占2个字节，指出本报文段中紧急数据最后一

个字节的序号。只有当紧急比特URG=1时才有效。

**选项：**长度可变。**最大报文长度的选项：**即最大报文段长度MSS ，默认537。**窗口扩大比例选项：**包括类型，长度，位移，位移值表示窗口的二进制扩大因子。**时间戳选项：为了帮助计算延时，也处理序号超过2^32的情况，防止序号萦回。时间戳包含两个值：**一个时间戳，一个回送回答时间戳值。

**建立TCP连接：**

**三次握手：**

1）主机A向主机B发送TCP连接请求数据包，其中包含主机A的初始序列号seq(A)=x。(报文中同步标志位SYN=1，ACK=0，表示这是一个TCP连接请求数据报文；序号seq=x，表明传输数据时的第一个数据字节的序号是x)

2）主机B收到请求后，会发回连接确认数据包。（其中确认报文段中，标识位SYN=1，ACK=1，表示这是一个TCP连接响应数据报文，并含主机B的初始序列号seq(B)=y，以及主机B对主机A初始序列号的确认号ack(B)=seq(A)+1=x+1）

3）第三次，主机A收到主机B的确认报文后，还需作出确认，即发送一个序列号seq(A)=x+1；确认号为ack(A)=y+1的报文；

**三次握手的功能：确保双方做好数据传送准备，初始序号达成一致。**

**序号随机的原因：**

防止序列号猜测攻击：

对于一个需要授权用户才可以访问的服务器而言,当攻击者能够猜测出要攻击的系统**用于下一次连接时使用的初始序列号时,**就能够欺骗这台服务器,通过假冒对该服务器SYN/ACK包的应答来欺骗服务器连接已经建立。

危害：新型拒绝服务，话路劫持。

防止方法：tcp加密，改进ISN生成算法。

**当系统崩溃并重启，之后会使远程系统认为旧的连接依然是打开的。**

假设有这样一种情况,当系统崩溃并重启动后再次 与远程系统的某端口建立连接，如果系统随机选择的ISN恰好与崩溃前建立的某个TCP连接的ISN相同而且源IP地址 、源端口号、目的IP地址和目的端口号都与崩溃前的该 TCP连接相同 ，那么远程系统会误认 为旧的连接现在依然是打开的(此时是不同于前者的现有连接).

**握手为三次的原因：**

**三次握手是两端正确同步的充要条件。因为TCP建立在不可靠的交互服务上，报文可能丢失，延迟，重复和无交付的情况。**

**Flood泛洪攻击：**

**解决：监控半开连接，当连接达到一定数量，统一释放资源。**

**延迟TCP分配，当三次握手完成才分配资源。**

**三次握手的性能：**

创建一个新连接代价是巨大的，纽约到伦敦，光三次握手就需要56ms.办法：尽量重用连接。

但连接不是想重用就可以重用的。新方案：TFO，tcp快速打开。

**四次挥手：**

1） 关闭客户端到服务器的连接：首先客户端A发送一个FIN，用来关闭客户到服务器的数据传送，然后等待服务器的确认。其中**终止标志位FIN=1**，序列号seq=u

 2） **服务器收到这个FIN，它发回一个ACK**，确认号ack为收到的序号加1。  
 3） **服务器关闭到客户端的连接**：也是**发送一个FIN给客户端**。  
 4） **客户段收到FIN后，并发回一个ACK报文确认**，并将确认序号seq设置为收到序号加1，首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

**注意：中断连接端可以是Client端，也可以是Server端。**

为什么要先执行步骤2，而不是直接执行步骤3：因为通知应用程序并获得响应可能花费大量时间，为了防止对方重传第一个FIN。

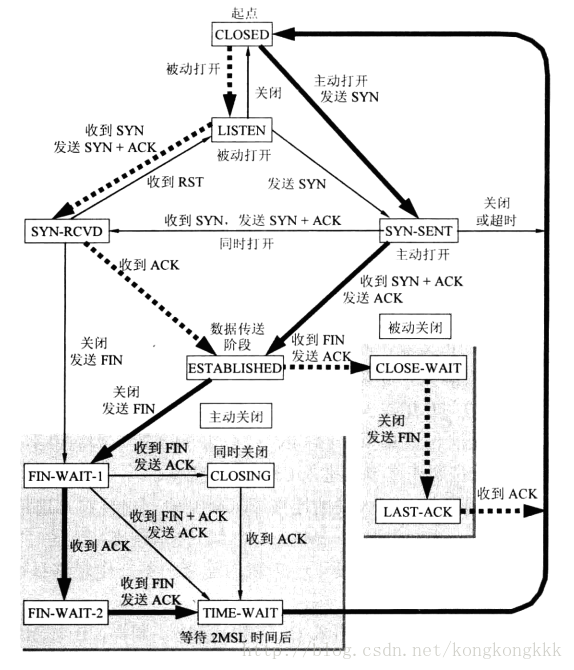
**为什么是4次：**

**TCP协议是全双工通信，这意味着客户端和服务器端都可以向彼此发送数据，所以关闭连接是双方都需要确认的共同行为。**

**等待2MSL时间主要目：**

怕最后一个ACK包对方没收到，那么对方在超时后将重发第三次握手的FIN包，主动关闭端接到重发的FIN包后可以再发一个ACK应答包。为了保证可靠，采用更加保守的等待时间2MSL。在TIME\_WAIT状态时两端的端口不能使用，要等到2MSL时间结束才可继续使用。当连接处于2MSL等待阶段时任何迟到的报文段都将被丢弃。

**TCP状态机：**



粗实线箭头表示**客户端的正常变迁**

粗虚线箭头表示**服务器的正常变迁**

细线箭头表示**异常变迁**

**CLOSED:**这个没什么好说的了，表示初始状态。

**LISTEN:**这个也是非常容易理解的一个状态，表示服务器端的某个SOCKET处于[监听状态](https://baike.baidu.com/item/%E7%9B%91%E5%90%AC%E7%8A%B6%E6%80%81)，可以接受连接了。

**SYN\_RCVD:**这个状态表示接受到了SYN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的[三次握手](https://baike.baidu.com/item/%E4%B8%89%E6%AC%A1%E6%8F%A1%E6%89%8B)会话过程中的一个中间状态，很短暂，基本上用netstat你是很难看到这种状态的，除非你特意写了一个客户端[测试程序](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E7%A8%8B%E5%BA%8F)，故意将三次TCP握手过程中最后一个ACK报文不予发送。因此这种状态时，当收到客户端的ACK报文后，它会进入到ESTABLISHED状态。

**SYN\_SENT:**这个状态与SYN\_RCVD遥想呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)，因此也随即它会进入到了SYN\_SENT状态，并等待服务端的发送[三次握手](https://baike.baidu.com/item/%E4%B8%89%E6%AC%A1%E6%8F%A1%E6%89%8B)中的第2个报文。SYN\_SENT状态表示客户端已发送SYN报文。

**ESTABLISHED：**这个容易理解了，表示连接已经建立了。

**FIN\_WAIT\_1:**这个状态要好好解释一下，其实FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)。而这两种状态的区别是：FIN\_WAIT\_1状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了FIN报文，此时该SOCKET即进入到FIN\_WAIT\_1状态。而当对方回应ACK报文后，则进入到FIN\_WAIT\_2状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应ACK报文，所以FIN\_WAIT\_1状态一般是比较难见到的，而FIN\_WAIT\_2状态还有时常常可以用netstat看到。

**FIN\_WAIT\_2：**上面已经详细解释了这种状态，实际上FIN\_WAIT\_2状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。

**TIME\_WAIT:**表示收到了对方的FIN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FIN\_WAIT\_1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。

**CLOSING:**这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送FIN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也就会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

**CLOSE\_WAIT:**这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方close一个SOCKET后发送FIN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)给自己，你系统毫无疑问地会回应一个ACK报文给对方，此时则进入到CLOSE\_WAIT状态。接下来呢，实际上你真正需要考虑的事情是查看你是否还有数据发送给对方，如果没有的话，那么你也就可以close这个SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE\_WAIT状态下，需要完成的事情是等待你去关闭连接。

**LAST\_ACK:**这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN[报文](https://baike.baidu.com/item/%E6%8A%A5%E6%96%87)后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。

例题：连续发送序列号70，100的报文，70丢失，100到达。第一个报文数据大小？30字节。第二个报文返回ACK序列号？70。如果70到达返回确认号？100。

## 309)vue使用闭包的地方

\_Set = (**function** () {  
 **function** Set () {  
 **this**.set = Object.create(**null**);  
 }  
 Set.prototype.has = **function** has (key) {  
 **return this**.set[key] === **true** };  
 Set.prototype.add = **function** add (key) {  
 **this**.set[key] = **true**;  
 };  
 Set.prototype.clear = **function** clear () {  
 **this**.set = Object.create(**null**);  
 };  
 **return** Set;  
}());

*/\*\*  
 \* Ensure a function is called only once.  
 \*/***function** once (fn) {  
 **var** called = **false**;  
 **return function** () {  
 **if** (!called) {  
 called = **true**;  
 fn.apply(**this**, arguments);  
 }  
 }  
}

*/\*\*  
 \* Parse simple path.  
 \*/***var** bailRE = /[^\w.$]/;  
**function** parsePath (path) {  
 **if** (bailRE.test(path)) {  
 **return** }  
 **var** segments = path.split('.');  
 **return function** (obj) {  
 **for** (**var** i = 0; i < segments.length; i++) {  
 **if** (!obj) { **return** }  
 obj = obj[segments[i]];  
 }  
 **return** obj  
 }  
}



## 310)udp协议

Udp头只有8字节。

 UDP协议是一种无连接的协议，该协议用来支撑那些需要在计算机之间传输数据的网络应用，包括网络视频会议系统在内的众多客户/服务器模式的网络应用。

1 .UDP是一个无连接的协议

 2 由于传输数据不建立连接，因此也就不需要维护连接状态。所以，一台服务器可同时向多个客户机传输相同的消息。

3吞吐量不受拥挤控制算法的调节，只受应用软件生成数据的速率、传输带宽、源端口和目标端口主机性能的限制。

4UDP使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态表。

 5.UDP是面向报文的。发送方的UDP对应用程序传输下来的报文，添加首部后就向下传送给IP层。既不拆分，也不合并，而是保留这些报文的边界。因此，应用程序需要选择合适的报文大小。

      虽然UDP是一个不可靠的协议，但它是分发信息的一个理想协议。例如，在屏幕上报告股.

**UDP会话劫持：**

如果是UDP协议，只需伪造IP地址，然后发送过去就可以了，因为UDP没有所谓的TCP三次握手，但基于UDP的应用协议有流控机制，所以也要做一些额外的工作。

## 311)SSL与TLS

TLS组成： TLS 记录协议和 TLS [握手协议](https://baike.baidu.com/item/%E6%8F%A1%E6%89%8B%E5%8D%8F%E8%AE%AE)

**TLS握手协议**

TLS交换数据前要协商建立加密信道。协商内容包括TLS版本，加密套件，可能还会验证证书。每一步都需要客户端与服务器往返一次。TLS建立信道很容易出错。服务器和浏览器会替我们做这些工作，我们需要提供和配置证书。

TLS [握手协议](https://baike.baidu.com/item/%E6%8F%A1%E6%89%8B%E5%8D%8F%E8%AE%AE)提供的连接安全具有三个基本属性：

1.可以使用非对称的，或公共密钥的[密码术](https://baike.baidu.com/item/%E5%AF%86%E7%A0%81%E6%9C%AF/2965467)来认证对等方的身份。该认证是可选的，但至少需要一个结点方。

2.共享加密密钥的协商是安全的。对[偷窃者](https://baike.baidu.com/item/%E5%81%B7%E7%AA%83%E8%80%85)来说协商加密是难以获得的。此外经过认证过的连接不能获得加密，即使是进入连接中间的攻击者也不能。

3.协商是可靠的。没有经过通信方成员的检测，任何攻击者都不能修改通信协商。

**TLS 记录协议**

TLS记录协议负责识别不同的消息类型，以及验证不同消息类型。

是一种[分层协议](https://baike.baidu.com/item/%E5%88%86%E5%B1%82%E5%8D%8F%E8%AE%AE/7703111)。每一层中的信息可能包含长度、描述和内容等字段。记录[协议支持](https://baike.baidu.com/item/%E5%8D%8F%E8%AE%AE%E6%94%AF%E6%8C%81)信息传输、将数据分段到可处理块、压缩数据、应用 MAC 、加密以及传输结果等。对接收到的数据进行解密、校验、[解压缩](https://baike.baidu.com/item/%E8%A7%A3%E5%8E%8B%E7%BC%A9)、重组等，然后将它们传送到高层客户机。

SSL(Secure Sockets Layer [安全套接层](https://baike.baidu.com/item/%E5%AE%89%E5%85%A8%E5%A5%97%E6%8E%A5%E5%B1%82)),及其继任者[传输层安全](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82%E5%AE%89%E5%85%A8)（Transport Layer Security，TLS）是为[网络通信](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1)提供安全及[数据完整性](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%AE%8C%E6%95%B4%E6%80%A7)的一种安全协议。最新3.0。

服务器认证阶段：1）[客户端](https://baike.baidu.com/item/%E5%AE%A2%E6%88%B7%E7%AB%AF)向服务器发送一个开始信息“Hello”以便开始一个新的会话连接；2）服务器根据客户的信息确定是否需要生成新的主[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5)，如需要则服务器在响应客户的“Hello”信息时将包含生成主密钥所需的信息；3）客户根据收到的服务器响应信息，产生一个主密钥，并用服务器的[公开密钥加密](https://baike.baidu.com/item/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%8A%A0%E5%AF%86)后传给服务器；4）服务器回复该主密钥，并返回给客户一个用主[密钥认证](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5%E8%AE%A4%E8%AF%81)的信息，以此让[客户认证](https://baike.baidu.com/item/%E5%AE%A2%E6%88%B7%E8%AE%A4%E8%AF%81)服务器。

[用户认证](https://baike.baidu.com/item/%E7%94%A8%E6%88%B7%E8%AE%A4%E8%AF%81)阶段：在此之前，服务器已经通过了[客户认证](https://baike.baidu.com/item/%E5%AE%A2%E6%88%B7%E8%AE%A4%E8%AF%81)，这一阶段主要完成对客户的认证。经认证的服务器发送一个提问给客户，客户则返回（数字）签名后的提问和其[公开密钥](https://baike.baidu.com/item/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5)，从而向服务器提供认证。

SSL握手协议层包括SSL握手协议（SSL HandShake Protocol）、SSL密码参数修改协议（SSL Change Cipher Spec Protocol）、应用数据协议（Application Data Protocol）和SSL告警协议（SSL Alert Protocol）。握手层的这些协议用于SSL管理信息的交换，允许应用协议传送数据之间相互验证，协商加密算法和生成密钥等。SSL握手协议的作用是协调客户和服务器的状态，使双方能够达到状态的同步。

SSL纪录协议（Record Protocol）为SSL连提供两种服务。

（1）保密性：利用握手协议所定义的共享密钥对SSL净荷（Payload）加密。

（2）完整性：利用握手协议所定义的共享的MAC密钥来生成报文的鉴别码（MAC）。

HTTPS和HTTP的区别主要为以下四点：

一、https协议需要到ca申请证书，一般免费证书很少，需要交费。

二、http是[超文本传输协议](https://baike.baidu.com/item/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)，信息是明文传输，https 则是具有[安全性](https://baike.baidu.com/item/%E5%AE%89%E5%85%A8%E6%80%A7)的[ssl](https://baike.baidu.com/item/ssl)加密传输协议。

三、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

四、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的[网络协议](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE)，比http协议安全。

https：是对服务端的认证。但也可以对客户端认证，如支付宝。https无法防御DNS劫持，可以防御http劫持。密钥协商是客户端生成随机字符串，用服务端公钥加密再返给服务端。

## 312)内存管理

**分页存储**

[**分页**](https://baike.baidu.com/item/%E5%88%86%E9%A1%B5)**存储管理是将一个进程的逻辑地址空间分成若干个大小相等的片，称为页面或页**，并为各页加以编号，从0开始，如第0页、第1页等。相应地，也把内存空间分成与页面相同大小的若干个存储块，称为(物理)块或页框(frame)，也同样为它们加以编号，如0#块、1#块等等。在为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于进程的最后一页经常装不满一块而形成了不可利用的碎片，称之为“页内碎片”。

获取Linux 内存页（基页）大小的命令：getconf PAGE\_SIZE ，一般的输出是4096，即 4KB。

**分段存储**

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等。每个段都有自己的名字。为了实现简单起见，通常可用一个段号来代替段名，每个段都从0开始编址，并采用一段连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而各段长度不等。整个作业的地址空间由于是分成多个段，因而是二维的，亦即，其逻辑地址由段号(段名)和段内地址所组成。

**段页存储**

段页式系统的基本原理，是基本分段存储管理方式和[基本分页存储管理方式](https://baike.baidu.com/item/%E5%9F%BA%E6%9C%AC%E5%88%86%E9%A1%B5%E5%AD%98%E5%82%A8%E7%AE%A1%E7%90%86%E6%96%B9%E5%BC%8F)原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。

虚拟内存：程序运行时，只把部分驻留内存，不需要的部分换出内存。虚拟内存需要硬件支持。

虚拟存储的特征：

1. 离散性：最基本特征。进程按照页或者段进行离散化放置。
2. 局部性：无需将整个进程驻留内存，只需载入必要的进程空间即可。
3. 对换性：程序运行有局部性，必然存在大部分并不是当前需要访问的空间，可以交换出去。
4. 虚拟性：对程序员来说，可以不用考虑物理存储系统的，可以认为内存无穷（内存与外部存储器大小总和）。

请求分页存储管理：

页面分配算法：

平均分配，按程序大小分配，按优先权分配。

置换算法：

最优页面置换算法，先进先出页面置换，最近最久未使用页面置换LRU（理想数据结果为双向链表），clock置换算法(最近未使用算法，NRC)。

抖动问题：系统缺页率很高，以致大量处理能力都用来处理缺页。预防：控制系统的多道程序。

注意：允许设置的虚拟[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98)最小值为2MB，最大值不能超过当前硬盘的剩余空间值，同时也不能超过32位[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F)的[内存寻址](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%AF%BB%E5%9D%80)范围——4GB。

## 313）设备管理与IO管理

移臂调度算法：

先来先服务，最短寻找时间优先调度算法，电梯调度算法，单向扫描调度算法。

## 314)前端垃圾回收与内存泄漏

引用计数和标记清除

DOM，js是两套垃圾回收机制。

**内存泄漏：**

**循环引用**（Circular References） - 当 IE 浏览器的 COM 组件与脚本引擎对象之间相互引用时，将导致内存泄露，这是最常见的形式。

**闭包（Closures）** - 闭包是循环引用的特殊形式，也是目前 Web 架构中使用最多的一种语言特性。闭包很容易被发现，因为它们依赖于特定的语言关键字，可以通过简单的搜索来查找。

**页面交叉泄露（Cross-Page Leaks）** - 页面交叉泄漏其实是一种较小的泄漏，它通常在你浏览过程中，由于内部对象 book-keeping 引起。我们将讨论 DOM 插入顺序问题，在示例中你将发现只需要微小的改动就可以避免 book-keeping 对象的产生。

**伪泄露（Pseudo-Leaks）** — 严格来说并不算真正的内存泄露，不过如果你不了解它，你将会在可用内存越来越少时非常懊恼。为了演示这个问题，我们将通过重写 script 元素中的内容来引发大量内存的“泄漏”。

当页面中元素被移除或替换时，若**元素绑定的事件仍没被移除**，在IE中不会作出恰当处理，此时要先手工移除事件，不然会存在内存泄露。解决：事件委托。

**在函数内定义事件，形成闭包**，定义事件回调时，由于是函数内定义函数，并且内部函数--事件回调的引用外暴了，形成了闭包解决之道，将事件处理函数定义在外部，解除闭包。

**自动类型装箱转换**

看网上资料，说下面的代码在ie系列中会导致内存泄露，先提个神，具体泄露与否先不管

var s=”lalala”;

alert(s.length);

s本身是一个string而非object，它没有length属性，所以当访问length时，[**js**](http://lib.csdn.net/base/javascript)引擎会自动创建一个临时String对象封装s，而这个对象一定会泄露。这个bug匪夷所思，所幸解决起来相当容易，记得所有值类型做.运算之前先显式转换一下：

var s="lalala";

alert(new String(s).length);

## 315)position:sticky 粘性定位

元素先按照普通文档流定位，然后相对于该元素在流中的 flow root（BFC）和 containing block（最近的块级祖先元素）定位。

而后，元素定位表现为在跨越特定阈值前为相对定位，之后为固定定位。

这个特定阈值指的是 top, right, bottom 或 left 之一，换言之，指定 top, right, bottom 或 left 四个阈值其中之一，才可使粘性定位生效。否则其行为与相对定位相同。

并且 top 和 bottom 同时设置时，top 生效的优先级高，left 和 right 同时设置时，left 的优先级高。

设定为 position:sticky 元素的任意父节点的 overflow 属性必须是 visible，否则 position:sticky 不会生效。这里需要解释一下：

如果 position:sticky 元素的任意父节点定位设置为 overflow:hidden，则父容器无法进行滚动，所以 position:sticky 元素也不会有滚动然后固定的情况。

如果 position:sticky 元素的任意父节点定位设置为 position:relative | absolute | fixed，则元素相对父元素进行定位，而不会相对 viewprot 定位。

.sticky {

position: sticky;

position: -webkit-sticky;

top: 0;

}

## 316)算法原理

贪心算法

背包问题

分治算法

动态规划

字符串匹配的算法：

Rabin-Karp 算法（字符串快速查找）

复杂度通常被认为是O(n+m)，优点：它能够处理多模式匹配.

KMP算法

BMH算法

## 317)事件流

事件发生时会在元素节点与根节点之间按照特定的顺序传播，路径所经过的所有节点都会收到该事件，这个传播过程即DOM事件流。

**冒泡型事件流**：事件的传播是从**最特定**的**事件目标**到最不特定的**事件目标**。即从DOM树的叶子到根。**捕获型事件流**：事件的传播是从**最不特定**的**事件目标**到最特定的**事件目标**。即从DOM树的根到叶子。

事件捕获的思想就是不太具体的节点应该更早接收到事件，而最具体的节点最后接收到事件。

addEventListener(event, function, useCapture)

* Event事件名，function回调函数，useCapture：true事件捕获，false事件冒泡。

[removeEventListener()](http://www.runoob.com/jsref/met-element-removeeventlistener.html) ：删除事件

attachEvent(type,handler),detachEvent(type,handler): type:事件类型， handler：处理程序函数.IE9之前的版本IE事件模型不支持事件捕获,只有事件冒泡。

**DOM3级新增事件stopImmediatePropagation()方法来阻止事件捕获。**该元素绑定的后序相同类型事件的监听函数的执行也将被阻止.

E.preventDefault()阻止默认行为。

E.stopPropagation()**方法既可以阻止事件冒泡，也可以阻止事件捕获，也可以阻止处于目标阶段。**

## 318）js类型转换

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **原始类型和对象类型** | **转换成String:** | **转换成Number** | **转换成Boolean** | **转换成Object** |
| undefined | "undefined" | NaN | false | throws TypeError |
| null | "null" | 0 | false | throws TypeError |
| true | "true" | 1 |  | new Boolean(true) |
| false | "false" | 0 |  | new Boolean(false) |
| "" (empty string) |  | 0 | false | new String("");Object("") |
| "1.2" (nonempty, numeric) |  | 1.2 | true | new String("1.2");Object("1.2") |
| "one" (nonempty, non-numeric) |  | NaN | true | new String("one");Object("one") |
| 0 | "0" |  | false | new Number(0);Object(0) |
| -0 | "0" |  | false | new Number(-0);Object(-0) |
| NaN | "undefined" |  | false | new Number(NaN);Object(NaN) |
| Infinity | "Infinity" |  | true | new Number(Infinity);Object(Infinity) |
| -Infinity | "-Infinity" |  | true | new Number(-Infinity);Object(-Infinity) |
| 1 (finite, non-zero) | "1" |  | true | new Number(1);Object(1) |
| {} (any object) |  | NaN | true |  |
| [] (empty array) | "" | 0 | true |  |
| [9] (1 numeric elt) | "9" | 9 | true |  |
| ['a'] (any other array) | use join() method | NaN | true |  |
| function(){} (any function | "undefined" | NaN | true |  |

严格相等指“===”，它**不允许比较双方进行强制类型转换**。

宽松相等指“==”，它**会对比较双方进行隐式强制类型转换**。

为对象就调用valueOf() 方法返回 对象的原始值。无值调用toString();

var fruits = ["Banana", "Orange"];var v=fruits.valueOf();返回"Banana", "Orange"。

Json或function返回本身。

1.对a,b强制按字符串string类型进行比较，可采用如下方法：  
"" + a == "" + b

2.对a,b强制按数字number类型进行比较，可采用如下方法：  
+a == +b

3.对a,b强制按布尔值boolean类型进行比较，可采用如下方法：  
!!a == !!b

"0" == null;   
//false：null转换为"null"  
"0" == undefined;   
//false：undefined转换为"undefined"  
"0" == false;   
//true：false转换为0；"0"转换为0  
"0" == NaN;   
//false：NaN转换为"NaN"  
"0" == 0;   
//true："0"转换为0  
"0" == "";   
//false：都是字符串，值不同  
false == null;   
//false：false转换为0；null转换为"null"，进而转换为数字，得到NaN  
false == undefined;   
//false：false转换为0；undefined转换为"undefined"，进而转换为数字，得到NaN  
false == NaN;   
//false：false转换为0；与NaN不同  
false == 0;   
//true：false转换为0  
false == "";   
//true：false转换为0；""转换为0  
false == [];   
//true：false转换为0；[]转换为0  
false == {};   
//false：false转换为0；{}转换为NaN  
"" == null;   
//false：""转换为0；null转换为"null"，进而转换为数字，得到NaN  
"" == undefined;   
//false：""转换为0；undefined转换为"undefined"，进而转换为数字，得到NaN  
"" == NaN;   
//false：""转换为0；与NaN不同  
"" == 0;   
//true：""转换为0  
"" == [];   
//true：""转换为0；[]转换为0  
"" == {};   
//false：""转换为0；{}转换为NaN  
0 == null;   
//false：null转换为"null"，进而转换为数字，得到NaN  
0 == undefined;   
//false：undefined转换为"undefined"，进而转换为数字，得到NaN  
0 == NaN;   
//false：0与NaN不同  
0 == [];   
//true：[]转换为0  
0 == {};   
//false：{}转换为NaN

**一些极端情况**

a. [] == ![]; //true  
　　[]被转换为0，![]被转换为false，进而被转换为0。  
b. 2 == [2]; //true  
　　[2]调用数组的valueOf()方法，返回”2”，进而被转换为2。  
c. “” == [null]; //true  
　　[null]调用数组的valueOf()方法，进而调用toString()方法，返回””。

转化规则

String，number转number

Boolean转number

undefined转“undefined”；

“”，undefined, null, 0， NaN。除了这5个假值以外，其他所有值转布尔类型都是true。

## 319)箭头函数

普通function函数和箭头函数的行为有一个微妙的区别，**箭头函数没有它自己的**this**值**，箭头函数内的this值继承自外围作用域。

箭头函数与非箭头函数间还有一个细微的区别，箭头函数不会获取它们自己的arguments对象。诚然，在ES6中，你可能更多地会使用不定参数和默认参数值这些新特性。

1. var obj = {
2. say: function () {
3. var f1 = function () {
4. console.log(this); // window, f1调用时,没有宿主对象,默认是window
5. setTimeout(() => {
6. console.log(this); // window
7. })
8. };
9. f1();
10. }
11. }
12. obj.say()
13. 严格模式下：指向undefeined
14. var obj = {
15. say: function () {
16. 'use strict';
17. var f1 = function () {
18. console.log(this); // undefined
19. setTimeout(() => {
20. console.log(this); // undefined
21. })
22. };
23. f1();
24. }
25. }
26. obj.say()

## 320)堆和栈的区别

堆和栈是一个统称，可以有很多的实现方式。计算机程序通常有一个栈叫做[调用栈](http://en.wikipedia.org/wiki/Call_stack)，用来存储当前函数调用相关的信息（比如：主调函数的地址，局部变量），因为函数调用之后需要返回给主调函数。栈通过扩展和收缩来承载信息。实际上，程序不是由运行时来控制的，它由编程语言、操作系统甚至是系统架构来决定。

堆是在任何内存中动态和随机分配的（内存的）统称；也就是无序的。内存通常由操作系统分配，通过应用程序调用 API 接口去实现分配。在管理动态分配内存上会有一些额外的开销，不过这由操作系统来处理。

如果你反汇编一些代码，你就会看到指针引用堆栈部分。就高级语言而言，语言有它自己的范围规则。一旦函数返回，函数中的局部变量会直接直接释放。你的编程语言就是依据这个工作的。

在堆中，也很难去定义。作用范围是由操作系统限定的，但是你的编程语言可能增加它自己的一些规则，去限定堆在应用程序中的范围。体系架构和操作系统是使用虚拟地址的，然后由处理器翻译到实际的物理地址中，还有页面错误等等。它们记录那个页面属于那个应用程序。不过你不用关心这些，因为你仅仅在你的编程语言中分配和释放内存，和一些错误检查

栈通常提前分配好了，因为栈必须是连续的内存块。语言的编译器或者操作系统决定它的大小。不要在栈上存储大块数据，这样可以保证有足够的空间不会溢出，除非出现了无限递归的情况（额，栈溢出了）或者其它不常见了编程决议。

堆是任何可以动态分配的内存的统称。这要看你怎么看待它了，它的大小是变动的。在现代处理器中和操作系统的工作方式是高度抽象的，因此你在正常情况下不需要担心它实际的大小，除非你必须要使用你还没有分配的内存或者已经释放了的内存。

栈更快因为所有的空闲内存都是连续的，因此不需要对空闲内存块通过列表来维护。只是一个简单的指向当前栈顶的指针。

## 321)flux架构

Flux将一个应用分成四个部分。

View： 视图层

Action（动作）：视图层发出的消息（比如mouseClick）

Dispatcher（派发器）：用来接收Actions、执行回调函数

Store（数据层）：用来存放应用的状态，一旦发生变动，就提醒Views要更新页面

## 322)elasticsearch：

{

"size": 1,

"query": {

"match\_all": {}

}

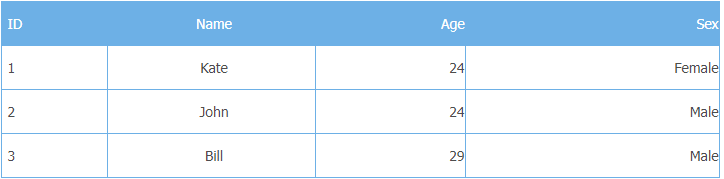
, "\_source": ["BUREAU\_CODE\_text","BUREAU\_CODE\_keyword"]

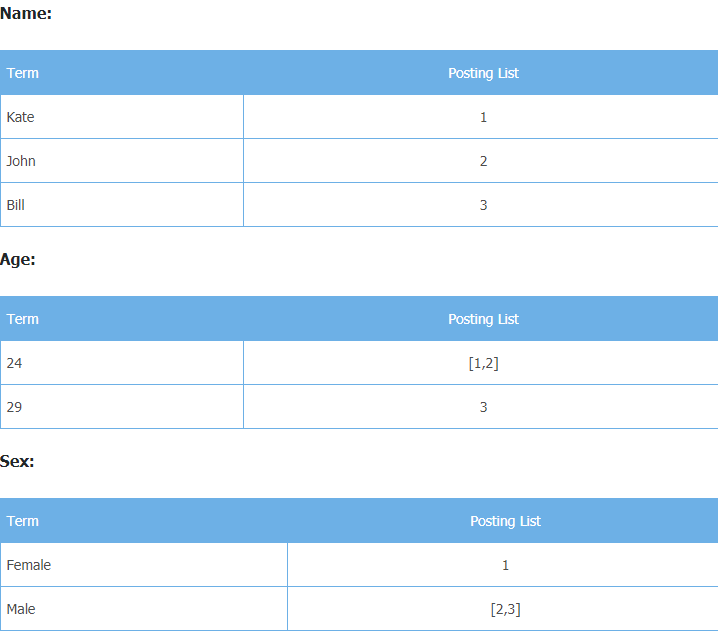
}

访问只返回确定的几个字段。

**es分页查询**：\_search?scroll=5m&size=20;将会返回一个过期时间为5分钟scroll\_id.在请求GET /\_search/scroll?。request body为｛“scroll”:”5m”,”scroll\_id”=”xxxxxxc”｝

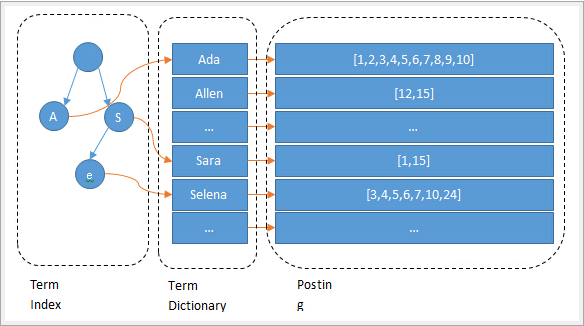
**倒排索引：**

Es建立索引的方式如下：

Posting list为倒排索引。

Elasticsearch为了能快速找到某个term，将所有的term排个序，二分法查找term，logN的查找效率，就像通过字典查找一样，这就是**Term Dictionary**。

B-Tree通过减少磁盘寻道次数来提高查询性能，Elasticsearch也是采用同样的思路，直接通过内存查找term，不读磁盘，但是如果term太多，term dictionary也会很大，放内存不现实，于是有了**Term Index**，就像字典里的索引页一样，A开头的有哪些term，分别在哪页，可以理解term index是一颗树，这棵树不会包含所有的term，**它包含的是term的一些前缀**。通过term index可以快速地定位到term dictionary的某个offset，然后从这个位置再往后顺序查找。term index不需要存下所有的term，而仅仅是他们的一些前缀与Term Dictionary的block之间的映射关系，再结合**FST(Finite State Transducers)**的压缩技术，可以使term index缓存到内存中。从term index查到对应的term dictionary的block位置之后，再去磁盘上找term，大大减少了磁盘随机读的次数。

**FST(Finite State Transducer,有限状态转换器)，又称Mealy Machine。是**Lucene中，很重要的一种数据结构。 **联合索引**

利用跳表(Skip list)的数据结构快速做“与”运算，或者

利用上面提到的bitset按位“与”

如果使用跳表，对最短的posting list中的每个id，逐个在另外两个posting list中查找看是否存在，最后得到交集的结果。

如果使用bitset，就很直观了，直接按位与，得到的结果就是最后的交集。

## 323)查看端口占用情况

Netstat: -t tcp -u udp -l 仅列出有在 Listen (监听)的服務状态 -p 显示建立相关链接的程序名 -n 拒绝显示别名,能显示数字的全部转化成数字

Lsof 是一个列出当前系统打开文件的工具.

-i 可以加端口，ip,如：lsof -i :22 查看22号端口占用情况。

ps -aux | grep tomcat -a/-A/-e all -u 查看特定用户进程 --sort命令来排序 -f显示程序间的关系 -x 显示没有控制终端的进程，同时显示各个命令的具体路径 -au   显示较详细的资讯 -aux   显示所有包含其他使用者的行程

ps -ef | grep Name

## 323)async和await:

async 函数返回的是一个 Promise 对象。

await 可以用于等待一个 async 函数的返回值.这也可以说是 await 在等 async 函数，但要清楚，它等的实际是一个返回值.await 不仅仅用于等 Promise 对象，它可以等任意表达式的结果，所以，await 后面实际是可以接普通函数调用或者直接量的。

1. **function** takeLongTime() {
2. **return** **new** Promise(resolve => {
3. setTimeout(() => resolve("long\_time\_value"), 1000);
4. });
5. }
6. //Async每一个步骤都需要之前每个步骤的结果。
7. Async function doIt() {
8. console.time("doIt");
9. **const** time1 = 300;
10. **const** time2 = awaitstep1(time1);
11. **const** time3 = awaitstep2(time1, time2);
12. **const** result = awaitstep3(time1, time2, time3);
13. console.log(`result is ${result}`);
14. console.timeEnd("doIt");
15. }
16. doIt();
17. //Promise()改写
18. **function** doIt() {
19. console.time("doIt");
20. **const** time1 = 300;
21. step1(time1)
22. .then(time2 => {
23. **return** step2(time1, time2)
24. .then(time3 => [time1, time2, time3]);
25. })
26. .then(times => {
27. **const** [time1, time2, time3] = times;
28. **return** step3(time1, time2, time3);
29. })
30. .then(result => {
31. console.log(`result is ${result}`);
32. console.timeEnd("doIt");
33. });
34. }
35. doIt();

## 324)vue路由

1.**v-link**有三种用法：<!-- 字面量路径 --><a v-link="'home'">Home</a>

<!-- 效果同上 --><a v-link="{ path: 'home' }">Home</a>

<!-- 具名路径 --><a v-link="{ name: 'detail', params: {id: '01'} }">Home</a>

**v-link**指令已经被一个新的[**<router-link> 组件**](http://router.vuejs.org/en/api/router-link.html)指令替代.

<router-link>不支持target="\_blank"，如果你想要在新标签页打开页面，应该使用<a>替代.

<router-link> 比起写死的 <a href="..."> 会好一些，理由如下：

无论是 HTML5 history 模式还是 hash 模式，它的表现行为一致，所以，当你要切换路由模式，或者在 IE9 降级使用 hash 模式，无须作任何变动。

在 HTML5 history 模式下，router-link 会拦截点击事件，让浏览器不在重新加载页面。

当你在 HTML5 history 模式下使用 base 选项之后，所有的 to 属性都不需要写（基路径）了。

2.**模式选择**：

· 默认hash: 使用URL hash值作为路由，支持所有浏览器  
· history: 依赖HTML5 History API和服务器配置  
· abstract：支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式。

更改模式：

var router = new VueRouter({

mode: 'abstract'

})

**3.路由定义**

由现在被定义为一个在 router 实例里的一个[**routes 选项**](http://router.vuejs.org/en/essentials/getting-started.html#javascript)数组。（不用router.map()）

var router = new VueRouter({

routes: [

{ path: '/foo', component: Foo },

{ path: '/bar', component: Bar }

]

})

嵌套路由：subRoutes更名为children.

{  
 path: '/home',  
 name: 'Home',  
 component: Home,  
 children:[  
 {path:'home01/:num',component:home01},  
 {path:'home02',component:home02}  
 ]  
}

**4.meta 属性**

现在任意的 route 属性必须在新 meta 属性的作用域内，以避免和以后的新特性发生冲突。

vue-router路由元信息说白了就是通过meta对象中的一些属性来判断当前路由是否需要进一步处理，如果需要处理，按照自己想要的效果进行处理即可！  
使用：meta: {requiresAuth: true}，record.meta.requiresAuth。

**5.$router**

$route.path 当前路由对象的路径，如'/view/a'

$rotue.params 关于动态片段（如/**user/:username**)的键值对信息,如{username: 'paolino'}

$route.query 请求参数，如/foo?user=1获取到query.user = 1

$route.router 所属路由器以及所属组件信息

$route.matched 数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。 $route.name 当前路径名字

**6.路由钩子**

**钩子**原名是“hook”，它是一种事件劫持机制，也就是说它会比你的事件更早进行执行处理。

vue里面提供了三大类钩子  
1、全局钩子

beforeEach  
2、某个路由独享的钩子

beforeEnter

beforeLeave: (to, from, next) => {

// ... }  
3、组件内钩子

beforeRouteEnter  
beforeRouteUpdate (2.2 新增)  
beforeRouteLeave

三种路由钩子中都涉及到了三个参数，这里直接上官方介绍吧

to: Route: 即将要进入的目标 路由对象  
from: Route: 当前导航正要离开的路由  
next: Function: 一定要调用该方法来 resolve 这个钩子。执行效果依赖 next 方法的调用参数。  
next(): 进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 confirmed （确认的）。  
next(false): 中断当前的导航。如果浏览器的 URL 改变了（可能是用户手动或者浏览器后退按钮），那么 URL 地址会重置到 from 路由对应的地址。  
next(‘/’) 或者 next({ path: ‘/’ }): 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。

**7.transition过渡动画**

<!-- 首先将要过渡的元素用transition包裹，并设置过渡的name，然后添加触发这个元素过渡的按钮（实际项目中不一定是按钮，任何能触发过渡组件的DOM操作的操作都可以） -->

<transition name="fade">

// 接着为过渡类名添加规则

&.fade-enter-active, &.fade-leave-active

transition: all 0.5s ease

&.fade-enter, &.fade-leave-active

opacity: 0

enter代表元素刚刚被插入 DOM 瞬间的状态。leave代表元素即将被从 DOM 中移除的瞬间的状态。

**过渡模式：**

<transition> 的默认行为 - 进入和离开同时发生

同时生效的进入和离开的过渡不能满足所有要求， Vue 提供了 过渡模式

in-out：新元素先进行过渡，完成之后当前元素过渡离开。

out-in：当前元素先进行过渡，完成之后新元素过渡进入。

## 325)vuex

export default {

Store,

install,

mapState,

mapMutations,

mapGetters,

mapActions

}

这里可以一目了然地看到 Vuex 对外暴露的 API。其中, Store 是 Vuex 提供的状态存储类，通常我们使用 Vuex 就是通过创建 Store 的实例

在使用 Vuex 的时候，通常会实例化 Store 类，然后传入一个对象，包括我们定义好的 actions、getters、mutations、state等.

 actions 和 mutations 的 key 允许重复，但 getters 的 key 是不允许重复的。官方推荐我们给这些全局的对象在定义的时候加一个名称空间来避免命名冲突。

## 326)502，504响应

502 Bad Gateway在高并发请求下，达到最大响应数，后续的请求就会出现502错误的。

**504 Gateway Time-out原因分析。**[nginx](http://www.ttlsa.com/nginx/).conf配置有关了。主要与以下几个参数有关：**fastcgi\_connect\_timeout、fastcgi\_send\_timeout、fastcgi\_read\_timeout、**fastcgi\_buffer\_size、fastcgi\_buffers、fastcgi\_busy\_buffers\_size、fastcgi\_temp\_file\_write\_size、fastcgi\_intercept\_errors。特别是前三个超时时间。如果fastcgi缓冲区太小会导致fastcgi进程被挂起从而演变为504错误。

## 327)nodejs高并发

并发请求，排队处理。并不是一定要处理完请求1才能去处理请求2：实际上请求的处理过程中，有很多的时间是耗在IO等其他地方，这时可以切换去处理其他请求，把等待的时间可以充分利用起来，达到更高的吞吐量。切换调度的策略是线程库，或者OS实现的，由于每个进程/线程需要占用不少资源（典型的是内存，一个线程通常需要2M的栈空间），更重要的是，线程/进程切换时的开销是非常大的。

同步阻塞IO（JAVA BIO）：   
    同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。   
同步非阻塞IO(Java NIO) ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。用户进程也需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问。   
异步阻塞IO（Java AIO）：    
   此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄（如果从UNP的角度看，select属于同步操作。因为select之后，进程还需要读写数据），从而提高系统的并发性！

**同步和异步是针对应用程序和内核的交互而言的:**

**同步**指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，

**异步**是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知

**Reactor模式采用同步IO，而Proactor采用异步IO。**

**IO模型**，select/poll/epoll/iocp.linux2.6后，默认使用epoll，windows为iocp.

node使用**异步IO和事件循环**，node.js使用的是单线程模型，对所有I/O都采用异步的请求方式，避免频繁的上下文切换，在node.js执行的时候维护着一个事件队列；程序在执行时进入事件循环等待下一个事件到来，每个异步I/O请求完成后都会被推送到事件队列中的等待执行。

**事件循环机制**

所谓事件循环是指node.js会把所有的异步操作使用事件机制解决，有个线程在不断地循环检测事件队列。

node.js中所有的逻辑都是事件的回调函数，所以node.js始终在事件循环中，程序入口就是事件循环第一个事件的回调函数。事件的回调函数中可能会发出I/O请求或直接发射（ emit）事件，执行完毕后返回事件循环。事件循环会检查事件队列中有没有未处理的事件，直到程序结束。node.js的事件循环对开发者不可见，由libev库实现，libev不断检查是否有活动的、可供检测的事件监听器，直到检查不到时才退出事件循环，程序结束。

**CPU密集型任务存在短板**

它会智能地将一些涉及到IO、网络通信等耗时比较长的操作，交由worker threads去执行，执行完了再回调，这就是所谓的异步IO非阻塞吧。但是，那些非IO操作，只用CPU计算的操作，它就自己扛了，比如算什么斐波那契数列之类。它是单线程，这些自己扛的任务要一个接着一个地完成，前面那个没完成，后面的只能干等。

## 328)nginx高并发

tomcat 不是一个请求产生一个线程，tomcat会维护一个线程池，每一个http请求，会从线程池中取出一个空闲线程。

**进程模型**

**nginx采用一个master进程，多个woker进程的模式。s**

master进程主要负责收集、分发请求。当一个请求过来时，master拉起一个worker进程负责处理这个请求。

master进程也要负责监控woker的状态，保证高可靠性

woker进程一般设置为跟cpu核心数一致。nginx的woker进程跟apache不一样。apche的进程在同一时间只能处理一个请求，所以它会开很多个进程，几百甚至几千个。而nginx的woker进程在同一时间可以处理额请求数只受内存限制，因此可以处理多个请求。

**事件模型**

**nginx是异步非阻塞的。**

每进来一个request，会有一个worker进程去处理。但不是全程的处理，处理到什么程度呢？处理到可能发生阻塞的地方，比如向上游（后端）服务器转发request，并等待请求返回。那么，这个处理的worker不会这么傻等着，他会在发送完请求后，注册一个事件：“如果upstream返回了，告诉我一声，我再接着干”。于是他就休息去了。此时，如果再有request 进来，他就可以很快再按这种方式处理。而一旦上游服务器返回了，就会触发这个事件，worker才会来接手，这个request才会接着往下走。

## 329)管道与文件描述符

如果两个不相关的进程打开同一个文件，一个只读方式开打，一个只写方式打开，不就相当于创建了一个管道了，进程A往里面写，进程B读，同样实现进程间通信。管道其实也一样。

管道实现的进程间通信是在父进程fork()出子进程时，子进程会继承父进程的文件描述符表，而这个文件描述符表里记录了所有父进程打开的文件，所以子进程也继承了父进程打开的文件，所以父子进程可以通过同一个文件描述符去访问同一个文件(但是不相干的进程文件描述符相同也不代表同一个文件)，而管道就是建立在文件描述符上的，建立一个管道必须和两个文件描述符相关，一个可以赋予可读权限，一个可以赋予可写权限，但是这两个文件描述符其实都和同一个文件关联，所以你通过不同的文件描述符去访问这个文件时，可以获得不同的访问权限，如果在父子进程中，在父进程将fd[0]为可读，fd[1]可写，而子进程fd[0]可读,fd[1]可写，那么这个管道就将父子进程连起来。

内核中，对应于**每个进程都有一个文件描述符表**，表示这个进程打开的所有文件。文件描述表中每一项都是一个指针，指向一个用于描述打开的文件的数据块———file对象，file对象中描述了文件的打开模式，读写位置等重要信息，当进程打开一个文件时，内核就会创建一个新的file对象。需要注意的是，file对象不是专属于某个进程的，**不同进程的文件描述符表中的指针可以指向相同的file对象**，从而共享这个打开的文件。file对象有引用计数，记录了引用这个对象的文件描述符个数，只有当引用计数为0时，内核才销毁file对象，因此某个进程关闭文件，不影响与之共享同一个file对象的进程.

file对象中包含一个指针，指向dentry对象。dentry对象代表一个独立的文件路径，如果一个文件路径被打开多次，那么会建立多个file对象，但它们都指向同一个dentry对象。

进程使用fork系统调用创建一个子进程后，子进程将继承父进程的文件描述符表，因此在父进程中打开的文件可以在子进程中用同一个描述符访问。

多个进程打开文件：（1）加文件锁 （2）多路复用

**write与fwrite的问题？**

fwrite：在写入的时候会出现交叉的情况，write是Linux操作系统的系统调用，fwrite是ANSIC标准的C语言库函数，fwrite在用户态是有缓冲区的。因此需要锁机制来保证并发环境下的安全访问。

**当多个进程写同一个文件时，则可能产生预期不到得结果。**

内核为所有的打开文件维持一张文件表。每个文件表项包含：

      （a）文件状态标志（读、写、添加、同步和非阻塞等）。

      （b）当前文件偏移量。

      （c）指向该文件v节点的指针。

每个打开文件（或设备）都有一个v节点（v-node）结构。v节点包含了文件类型和对此文件进行各种操作的函数的指针。对于大多数文件，v节点还包含了该文件的i节点（i-node,索引节点）。这 些信息是在打开文件时从磁盘上读入内存的，所以所有关于文件的信息都是快速可供使用的。

**进程最大打开文件描述符数**:ulimit -n

**临时性：**通过ulimit -Sn设置最大打开文件描述符数的soft limit.

**永久性：**上面的方法只是临时性的，注销重新登录就失效了，而且不能增大hard limit，只能在hard limit范围内修改soft limit。若要使修改永久有效，则需要在/etc/security/limits.conf中进行设置.

## 330)inode

文件储存在硬盘上，硬盘的最小存储单位叫做"扇区"（Sector）。每个扇区储存512字节。

操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个"块"（block）。这种由多个扇区组成的"块"，是文件存取的最小单位。"块"的大小，最常见的是4KB，即连续八个 sector组成一个 block。

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。**这种储存文件元信息的区域就叫做inode**，中文译名为"索引节点"。**索引节点仅当文件被访问色才创建**。注意：inode有两种，一种是VFS的inode，一种是具体文件系统的inode。前者在内存中，后者在磁盘中。

### inode包含文件的元信息:

具体来说有以下内容：

　　\* 文件的字节数

　　\* 文件拥有者的User ID

　　\* 文件的Group ID

　　\* 文件的读、写、执行权限

　　\* 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。

　　\* 链接数，即有多少文件名指向这个inode

\* 文件数据block的位置

inode也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一**个是数据区，存放文件数据**；**另一个是inode区**（inode table），存放inode所包含的信息。

每个inode节点的大小，**一般是128字节或256字节**。**inode节点的总数，在格式化时就给定，一般是每1KB或每2KB就设置一个inode**。假定在一块1GB的硬盘中，每个inode节点的大小为128字节，每1KB就设置一个inode，那么inode table的大小就会达到128MB，占整块硬盘的12.8%。

查看每个硬盘分区的inode总数和已经使用的数量，可以使用df命令。每个文件都只对应一个inode，（**每1KB或每2KB就设置一个inode只是文件系统预设inode个数使用，与实际文件大小无关。**）df -i 查看磁盘inode.

每个inode都有一个号码，操作系统**用inode号码来识别不同的文件**。（通过inode去查找文件）

\*记事本多个进程文件后面的会覆盖前面的，而且文件可删除。

### 超级块：

各文件系统必须实现超级块，该对象用于存储特定文件系统的信息，通常对应于存放在磁盘特定扇区的文件系统超级块或文件系统控制块。对于基于内存的文件系统，他们会现场创建超级块并保存在内存中。

超级块：包含文件系统信息的数据结构，如inode/block大小、inode/block总量、使用量、剩余量、文件系统格式等。一个文件系统仅有一个superblock。文件系统的控制信息、单独文件的信息这些都包含超级块中。超级块通常存放在特定的扇区中。**目录文件的结构非常简单**，就是一系列目录项(dirent)的列表。每个目录项，由两部分组成：所包含文件的文件名，以及该文件名对应的inode号码。

### Dentry：

dentry的中文名称是目录项，是Linux文件系统中某个[索引节点](https://baike.baidu.com/item/%E7%B4%A2%E5%BC%95%E8%8A%82%E7%82%B9)(inode)的链接。这个[索引节点](https://baike.baidu.com/item/%E7%B4%A2%E5%BC%95%E8%8A%82%E7%82%B9)可以是文件的，也可以是目录的。

### 硬链接

一般情况下，文件名和inode号码是"一一对应"关系，每个inode号码对应一个文件名。但是，Unix/Linux系统允许，多个文件名指向同一个inode号码。

这意味着，可以用不同的文件名访问同样的内容；对文件内容进行修改，会影响到所有文件名；但是，**删除一个文件名**，不影响另一个文件名的访问。这种情况就被称为"硬链接"（hard link）。

ln命令可以创建硬链接：

### 软链接

除了硬链接以外，还有一种特殊情况。

文件A和文件B的inode号码虽然不一样，但是文件A的内容是文件B的路径。读取文件A时，系统会自动将访问者导向文件B。因此，无论打开哪一个文件，最终读取的都是文件B。这时，文件A就称为文件B的"软链接"（soft link）或者"符号链接（symbolic link）。

这意味着，文件A依赖于文件B而存在，如果删除了文件B，打开文件A就会报错："No such file or directory"。这是软链接与硬链接最大的不同：文件A指向文件B的文件名，而不是文件B的inode号码，文件B的inode"链接数"不会因此发生变化。

ln -s命令可以创建软链接。

### Inode查找过程：

在读取一个文件时，总是从根目录开始读取，每一个目录或者文件，在VFS中，都是一个文件对象，每一个文件对象都有唯一的一个inode与之对应。根目录的inode号为0，在superblock里，可以很快根据inode号索引到具体的inode，因此读取到的第一个inode就是根目录的。读取到了该目录后，内核对象会为该文件对象建立一个dentry，并将其缓存起来，方便下一次读取时直接从内存中取。而目录本身也是一个文件，目录文件的内容即是该目录下的文件的名字与inode号，目录文件的内容就像一张表，记录的文件名与其inode。根据路径即可找到当前需要读取的下一级文件的名字和inode，同时继续为该文件建立dentry，dentry结构是一种含有指向父节点和子节点指针的双向结构，多个这样的双向结构构成一个内存里面的树状结构，也就是文件系统的目录结构在内存中的缓存了。有了这个缓存，我们在访问文件系统时，通常都非常快捷。

## 331)共享内存

共享内存是一种最为高效的进程间通信方式，进程可以直接读写内存，而不需要任何数据的拷贝

**为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间。进程就可以直接读写这一块内存而不需要进行数据的拷贝，从而大大提高效率。**

由于多个进程共享一段内存，因此也需要依靠某种同步机制。

步骤：

<1>创建共享内存,shmget函数.

<2>映射共享内存，即把指定的共享内存映射到进程的地址空间用于访问,shmat函数

<3>撤销共享内存映射,shmat函数

<4>删除共享内存对象

控制共享内存,shmctl(int shmid,int cmd,struct shmid\_ds \*buf)

shmid  共享内存标识ID

cmd      IPC\_STAT得到共享内存的状态

          IPC\_SET改变共享内存的状态

**IPC\_RMID删除共享内存**

## 332)absolute,relative,fixed.float

**static：**无特殊定位，对象遵循正常文档流。top，right，bottom，left等属性不会被应用。

**relative：**对象遵循正常文档流，但将依据top，right，bottom，left等属性在正常文档流中偏移位置。而其层叠通过z-index属性定义。

**absolute：**对象脱离正常文档流，使用top，right，bottom，left等属性进行绝对定位。而其层叠通过z-index属性定义。

使用absoult定位的元素脱离文档流后，就只能根据祖先类元素(父类以上)进行定位，而这个祖先类还必须是以postion**非static方式**定位的， 举个例子，a元素使用absoulte定位，它会从父类开始找起，寻找以position非static方式定位的祖先类元素(注意，一定要是直系祖先才算哦~），直到<html>标签为止，这里还需要注意的是，relative和static方式在最外层时是以<body>标签为定位原点的，而absoulte方式在无父级是position非static定位时是以<html>作为原点定位。

祖先类的margin会让子类的absoulte跟着偏移，而padding却不会让子类的absoulte发生偏移。

**fixed：**对象脱离正常文档流，使用top，right，bottom，left等属性以窗口为参考点进行定位，当出现滚动条时，对象不会随着滚动。而其层叠通过z-index属性定义。

它和absoult定位一样，都脱离了文档流，并且能够根据top、right、left、bottom属性进行定位，但不同的是fixed是**根据窗口为原点**进行偏移定位的，也就是说它不会根据滚动条的滚动而进行偏移。

**什么是文档流？**

      将窗体自上而下分成一行行, 并在每行中按从左至右的顺序排放元素,即为文档流。

      只有三种情况会使得元素脱离文档流，分别是：浮动、绝对定位和相对定位。

**如果top和bottom一同存在的话，那么只有top生效。**

**如果left和right一同存在的话，那么只有left生效。**

**Absolute，fiexd:使float失效。**

**Absolute，fiexd，relative:margin-bottom,margin-right失效。**

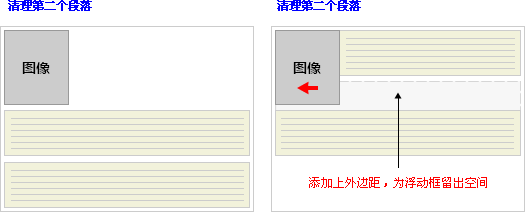
**Float:left,right,top,bottom无效。**

**Float right:margin-bottom无效，margin-left无效。**

**Relative:**使用的值始终为：left = -right。Top=-bottom.(测试结果只看left和top,设置right相当于left = -right)。

**Float元素:**可以向左或向右移动，直到它的外边缘碰到包含框或另一个浮动框的边框为止。

由于浮动框不在文档的普通流中，所以文档的普通流中的块框表现得就像浮动框不存在一样。**但它不会与内容重叠，也不会与其它浮动元素重叠**。浮动框旁边的行框被缩短，从而给浮动框留出空间，**行框围绕浮动框**。因此，**创建浮动框可以使文本围绕图像**。要想阻止行框围绕浮动框，需要对该框应用 [**clear 属性**](http://www.w3school.com.cn/cssref/pr_class_clear.asp)。clear 属性的值可以是 left、right、both 或 none，它表示框的哪些边不应该挨着浮动框。

为了实现这种效果，在被清理的元素的**上外边距**上添加足够的空间，使元素的顶边缘垂直下降到浮动框下面：

这是一个有用的工具，它让周围的元素为浮动元素留出空间。

## 333)express路由的实质

匹配http请求的url。

## 334)vertical-align

vertical-align用来设置垂直对齐方式。

vertical-align:baseline(元素的基线与父元素的基线对齐)

vertical-align:sub(降低元素的基线到父元素合适的下标位置)

vertical-align:super(升高元素的基线到父元素合适的上标位置)

vertical-align:bottom(把对齐的子元素的底端与行框底端对齐)

vertical-align:text-bottom(把元素的底端与父元素内容区域的底端对齐)

vertical-align:top(把对齐的子元素的顶端与行框顶端对齐)

vertical-align:text-top(把元素的顶端与父元素内容区域的顶端对齐)

vertical-align:middle(元素的中垂点与父元素的基线加1/2父元素中字母X的高度对齐)

vertical-align:(+-n)px(元素相对于基线上下偏移npx)

vertical-align:x%(相对于元素的line-height值)

vertical-align:inherit(从父元素继承vertical-align属性的值)

## 335)white-space

|  |  |
| --- | --- |
| normal | 默认。空白会被浏览器忽略。 |
| pre | 空白会被浏览器保留。其行为方式类似 HTML 中的 <pre> 标签。 |
| nowrap | 文本不会换行，文本会在在同一行上继续，直到遇到 <br> 标签为止。 |
| pre-wrap | 保留空白符序列，但是正常地进行换行。 |
| pre-line | 合并空白符序列，但是保留换行符。 |
| inherit | 规定应该从父元素继承 white-space 属性的值。 |

Nowrap:可以让div在一行不换行。子div是inline-block。

补充：word-break:break-all:单词允许换行，keep-all只能在半角或连字符处换行。

Css3 word-wrap：normal 只在允许的断字点换行,break-word:在长单词或URL内部进行换行。

## 336)cookie防篡改

为预防cookie篡改，网站在向用户的计算机发送cookie前应该实施一些保护措施（比如通过加密），Ingrian Networks公司的Active Application Security就是这样一种防止cookie篡改的平台。当cookie通过平台时，敏感的信息都将被加密。数字签名被创建用来验证发送者和接收者以后通信时的内容，如果内容被篡改，数字签名将和以前的不一样，服务器将拒绝登录。

1. 设置该 Cookie 不能被脚本读取。
2. 对 Cookie 内容进行加密，在加密前嵌入时间戳，保证每次加密后的密文都不一样（并且可以防止消息重放）。

3）客户端请求时，每次或定时更新 Cookie 内容

4）每次向 Cookie 写入时间戳，数据库需要记录最后一次时间戳

5）客户端提交 Cookie 时，先解密然后校验时间戳，时间戳若小于数据数据库中记录，即意味发生攻击。

## 337)两种新的数据结构：[WeakSet](http://es6.ruanyifeng.com/#docs/set-map#WeakSet) 和 [WeakMap](http://es6.ruanyifeng.com/#docs/set-map#WeakMap)

及时清除引用非常重要。但是，你不可能记得那么多，有时候一疏忽就忘了，所以才有那么多内存泄漏。

最好能有一种方法，在新建引用的时候就声明，哪些引用必须手动清除，哪些引用可以忽略不计，当其他引用消失以后，垃圾回收机制就可以释放内存。这样就能大大减轻程序员的负担，你只要清除主要引用就可以了。

ES6 考虑到了这一点，推出了两种新的数据结构：[WeakSet](http://es6.ruanyifeng.com/#docs/set-map#WeakSet) 和 [WeakMap](http://es6.ruanyifeng.com/#docs/set-map#WeakMap)。它们对于值的引用都是不计入垃圾回收机制的，所以名字里面才会有一个"Weak"，表示这是弱引用。

WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

const ws = new WeakSet();

ws.add(1)

const wm1 = new WeakMap();

const key = {foo: 1};

wm1.set(key, 2);

wm1.get(key) // 2

## 338)setImmediate()

  逻辑含义上讲, 与setTimeout(callback, 0)一致.在异步IO调用中, 如果同时使用setImmediate()和settimeout(), 可以保证, setImmediate()先于所有的setTimeout()执行.

## 339)Thread,Runnable

一种是继承Thread类，一种是实现Runnable接口.Thread类是在java.lang包中定义的。一个类只要继承了Thread类同时覆写了本类中的run()方法就可以实现多线程操作了，但是一个类只能继承一个父类，这是此方法的局限.

在实际开发中一个多线程的操作很少使用Thread类，而是通过Runnable接口完成,并实现run()。

实现Runnable接口相比继承Thread类有如下好处：

* 避免点继承的局限，一个类可以继承多个接口。
* 适合于资源的共享。

## 340)ES6中map遍历

forEach方法简短一点，但是它也有它的缺点，就是**不能够break来中途退出循环**，也**不能通过return来退出循环**。

for-of不仅仅可以用于**数组**，还可以用在**dom对象**上，同样作用于**Map和Set对象**。

介绍完了两种类型的新对象，让我们看看for-of如何作用于Set对象上的。

遍历Map对象则不同，因为Map是以键值对存储的，所以我们需要分开**两个单独的变量**来遍历。

for-of并**不是用来遍历普通的对象的属性的**，如果我们一定要这么做，可以使用for-in。

## 341)sctp协议

作为一个传输层协议，SCTP兼有TCP及UDP两者的特点。SCTP可以称为是TCP的改进协议，但它们之间仍然存在着较大的差别。

首先SCTP和TCP之间的最大区别是SCTP的连接可以是多宿主连接的，TCP则一般是单地址连接的。在进行SCTP建立连接时，双方均可声明若干IP地址（IPv4，IPv6或主机名）通知对方本端所有的地址。若当前连接失效，则可切换到另一个地址，而不需要重新建立连接。

其次SCTP是基于消息流，而TCP则是基于字节流。所谓基于消息流，是指发送数据和应答数据的最小单位是消息包（chunk）。一个SCTP连接（Association）同时可以支持多个流（stream），每个流包含一系列用户所需的消息数据（chunk）。而TCP则只能支持一个流。

在网络安全方面，SCTP增加了防止恶意攻击的措施。不同于TCP连接采用的三次握手机制，SCTP连接采用四次握手机制，有效的防止了类似于SYN Flooding的防范拒绝服务攻击。SCTP主要的贡献是对多重联外线路的支持，一个端点可以由多于一个IP地址组成，使得传输可在主机间或网卡间做到透明的网络容错备援。

所以http不一定使用tcp传输，也可以使用sctp.

## 342)jquery事件处理

Ele.on(“click”,”p”,function(){});

jQuery.fn.on(),jQuery.event.add()//给选中元素注册事件处理程序

jQuery.event.dispatch//分派事件处理函数

jQuery.event.fix修正event对象,jQuery.event.fix这个方法会对event做兼容处理.

jQuery.event.handlers//组装事件处理器队列

执行事件处理函数。

.on( events [, selector ] [, data ], handler(eventObject) )

events：事件名

selector : 一个选择器字符串，用于过滤出被选中的元素中能触发事件的后代元素

data :当一个事件被触发时，要传递给事件处理函数的

handler:事件被触发时，执行的函数

on方法实质只完成一些参数调整的工作，而实际负责事件绑定的是其内部jQuery.event.add方法。

add方法可以拆分2部分：一个事件预绑定期，一个事件执行期。

预绑定使用addEventListener，ie使用attachEvent();

eventHandle源码

回到内部绑定的事件句柄eventHandle ，可想而知eventHandle不仅仅只是只是充当一个回调函数的角色，而是一个实现了EventListener接口的对象

eventHandle中并没有直接处理回调函数，而是映射到jQuery.event.dispatch分派事件处理函数了,仅仅只是传入eventHandle.elem，arguments , 就是body元素 与事件对象.

on内部的实现机制

jQuery并没有将事件处理函数直接绑定到DOM元素上，而是通过$.data存储在缓存$.cahce上

第一步：获取数据缓存

//获取数据缓存

elemData = data\_priv.get( elem );

在$.cahce缓存中获取存储的事件句柄对象，如果没就新建elemData

第二步：创建编号

if ( !handler.guid ) {

handler.guid = jQuery.guid++;

}

为每一个事件的句柄给一个标示，添加ID的目的是 用来寻找或者删除handler，因为这个东东是缓存在缓存对象上的，没有直接跟元素节点发生关联

第三步：分解事件名与句柄

events，eventHandle 都是elemData缓存对象内部的，可见

在elemData中有两个重要的属性，

一个是events，是jQuery内部维护的事件列队

一个是handle，是实际绑定到elem中的事件处理函数

之后的代码无非就是对这2个对象的筛选，分组，填充了

第四步: 填充事件名与事件句柄

**涉及**

**多事件处理：**

如果是多事件分组的情况jQuery(...).bind("mouseover mouseout", fn);

事件可能是通过空格键分隔的字符串，所以将其变成字符串数组

**增加命名空间处理：**

事件名称可以添加指定的event namespaces（命名空间） 来简化删除或触发事件。

**引入jQuery的Special Event机制**

什么时候要用到自定义函数？有些浏览器并不兼容某类型的事件，如IE6～8不支持hashchange事件，你无法通过jQuery(window).bind('hashchange', callback)来绑定这个事件，这个时候你就可以通过jQuery自定义事件接口来模拟这个事件，做到跨浏览器兼容。

## 343)原型继承的缺点

继承了过多不必要的属性。

无法继承借用构造函数的原型。

共享原型：即将其他构造函数的原型直接赋值给本构造函数的原型

缺点：两个原型会想回影响，更改其中一个原型，更一个对应的原型也会被更改。

## 344)在 ES6中 改良的5个 JavaScript “缺陷”

1. 块级作用域

防止变量在作用域外被访问

防止重复声明变量

循环体中的闭包不再有问题

在 ES5 中，**如果循环体内有产生一个闭包，访问闭包外的变量**，会产生问题。在 ES6，**你可以使用 “let” 来避免问题**。

2. 词法作用域的 “this” (通过箭头函数)

在 ES5 中，“this” 会随着函数调用位置和调用方式改变，这种变化无常给很多开发者带来过痛苦的经历。 ES6 通过**箭头函数**带来的词法作用域的 “this” 消除了这个问题。

词法作用域的 “this” 特性让变量的 “this” **总是指向词法声明时的那个对象**。

ES5 中的 “this” 问题和两种解决方式

在 ES6 中的解决方法：

在 ES6 中简单使用箭头函数就可以自动获得词法作用域的 “this”

3. 处理 “arguments”

在 ES5 中，“arguments” 表现得像一个数组（例如：我们可以通过length来遍历它），但是它不是一个真正的数组。所以，一切数组方法，比如 sort、slice 等等都用不了，

在 ES6 中，我们可以使用一个新的特性叫做 **rest 参数**。它的形式为...参数名（比如：...args）。rest 参数是一个真正的数组，所以我们可以对它使用所有数组上可用的方法。

4. 类

从概念上讲，在 ES6 之前的 JS 中并没有和其他面向对象语言那样的“类”的概念。长时间里，人们把使用 new 关键字调用的函数（也叫构造器）当做“类”来使用。

由于 JS 不支持原生的类，而只是通过原型来模拟，各种模拟类的方式相对于传统的面向对象方式来说非常混乱，尤其是处理当子类继承父类、子类要调用父类的方法等等需求时。

ES6 带来了新的语法，与其他各种编程语言类似的语法，使得面向对象变得非常简单。

5. 严格模式

[严格模式](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)(use strict) 有助于防止问题用法，并且它也有助于[安全使用 JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#Securing_JavaScript)。在 ES5 中, 严格模式是可选项，但是在 ES6 中，[许多特性要求必须使用严格模式](http://www.ecma-international.org/ecma-262/6.0/#sec-strict-mode-code)。 因此大多数开发者和 babel 之类的工具默认添加 use strict 到 JS 文件的头部，以确保整个 JS 文件的代码都采用严格模式，这个习惯有助于我们写更好的 JavaScript。

## 345)闭包中的this

闭包中this指向window.

var myNumber = {

value: 1,

add: function(i){

var helper = function(i){

console.log(this); this.value += i;

}

helper(i);

}

}

myNumber.add(1);

解决：1.var that=this;//定义变量that用于保存上层函数的this对象

1. helper.apply(this,[i]);//使用apply改变helper的this对象指向，使其指向myNumber对象
2. ES6使用箭头函数解决

## 346)同步异步，阻塞非阻塞

**.**同步与异步  
同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)  
所谓同步，就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。

阻塞与非阻塞  
阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。  
非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

1 老张把水壶放到火上，立等水开。（同步阻塞）  
老张觉得自己有点傻  
2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）

老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。

3 老张把响水壶放到火上，立等水开。（异步阻塞）  
老张觉得这样傻等意义不大  
4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）

**所谓同步异步，只是对于水壶而言。所谓阻塞非阻塞，仅仅对于老张而言。**

## 347)for click解决方案

for(var i = 0;i<4;i++){

li.addEventListener(“click”,function(){

Alert(i);

})

}

方案1：将var改为let，块级作用域

方案2：立即执行函数，(function(i){

li.addEventListener(“click”,function(){

Alert(i);

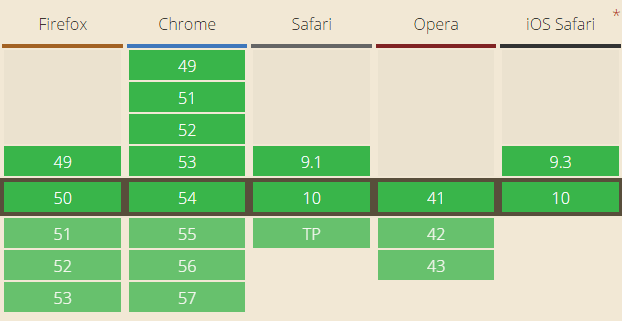
})

})(i);函数按值传递，传递了一份拷贝，所以外面i改变，不影响传递进去的值。

## 348)web字体图

## 349)css变量

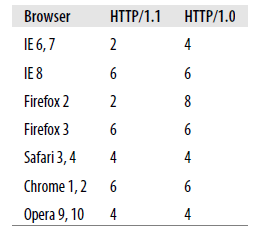
兼容：

变量声明：--v:#333(可以是数字，字母，下划线，横短线，**中文，日文**)

使用：color:var(--v)

属性名不可以走变量，不可以支持多个变量声明。变量不合法将使用缺省特性。变量的值可以传递。

## 350)浏览器最多并行下载多少个资源

**补充**：同一时间针对同一域名下的请求有一定数量限制，超过限制数目的请求会被阻塞。大多数浏览器的并发数量都控制在6以内。有些资源的请求时间很长，因而会阻塞其他资源的请求。因此，对于一些静态资源，如果放到不同的域名下面就能实现与其他资源的并发请求。

## 351)nginx配置反向代理

**错误： invalid number of arguments in "proxy\_pass" directive**

**改：语句后面加分号，注意中英文切换。**

**location /node/ {**

**proxy\_pass http://127.0.0.1:3333;**

**proxy\_redirect off;**

**proxy\_set\_header Host $http\_host;**

**proxy\_set\_header X-Real-IP $remote\_addr;**

**proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;**

**proxy\_set\_header X-NginX-Proxy true;**

**}**

**Node的路由地址为/node/mysql；/node/users；/node/**

## 352)fieldset标签

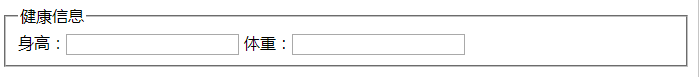
fieldset 元素可将表单内的相关元素分组。

<fieldset> 标签将表单内容的一部分打包，生成一组相关表单的字段。

当一组表单元素放到 <fieldset> 标签内时，浏览器会以特殊方式来显示它们，它们可能有特殊的边界、3D 效果，或者甚至可创建一个子表单来处理这些元素。

<fieldset> 标签没有必需的或唯一的属性。

[<legend> 标签](http://www.w3school.com.cn/tags/tag_legend.asp)为 fieldset 元素定义标题，text-center可调整位置。



<fieldset>  
 <legend>健康信息</legend>  
 身高：<input type="text" />  
 体重：<input type="text" />  
</fieldset>

## 353)box-reflect制作倒影

非标准请加前缀-webkit-box-reflect

语法：  
    box-reflect：none | <direction> <offset>? <mask-box-image>?  
    <direction> = above | below | left | right  
    <offset> = <length> | <percentage>  
    <mask-box-image> = none | <url> | <linear-gradient> | <radial-gradient> | <repeating-linear-gradient> | <repeating-radial-gradient>

## 354)js异步事件

  JS的异步是通过回调函数实现的，即通过任务队列，在主线程执行完当前的任务栈（所有的同步操作），主线程空闲后轮询任务队列，并将任务队列中的任务（回调函数）取出来执行。"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，当主线程开始执行异步任务，就是执行对应的回调函数。

    虽然JS是单线程的但是浏览器的内核是多线程的，在浏览器的内核中不同的异步操作由不同的浏览器内核模块调度执行，异步操作会将相关回调添加到任务队列中。而不同的异步操作添加到任务队列的时机也不同，如 onclick, setTimeout, ajax 处理的方式都不同，这些异步操作是由浏览器内核的 webcore 来执行的，webcore 包含上图中的3种 webAPI，分别是 DOM Binding、network、timer模块。

**onclick 由浏览器内核的 DOM Binding 模块来处理，当事件触发的时候，回调函数会立即添加到任务队列中。**

**setTimeout 会由浏览器内核的 timer 模块来进行延时处理，当时间到达的时候，才会将回调函数添加到任务队列中。**

**ajax 则会由浏览器内核的 network 模块来处理，在网络请求完成返回之后，才将回调添加到任务队列中。**

JS中的异步运行机制如下:

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

（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。

**JS种事件队列的优先级**

   在JS中ES6 中新增的任务队列（promise）是在事件循环之上的，事件循环每次 tick 后会查看 ES6 的任务队列中是否有任务要执行，也就是 ES6 的任务队列比事件循环中的任务（事件）队列优先级更高。

如 Promise 就使用了 ES6 的任务队列特性。也即在执行完任务栈后首先执行的是任务队列中的promise任务。其他的上面常见的异步操作加入队列的时间没有相应的优先级。

## 355)数组自带会改变数组的方法

**Pop().push(),shfit(),unshift(),splice(),sort(),reverse();**

**var A = [];**

**A[8] = 1;**

1. **length;//9长度会变成9**

## 356)日志文件系统

日志文件系统是一种特殊类型的文件系统，它记录一个被称为日志的追踪文件。日志可以让系统能够修复在异常终止时出现的不一致情况。它通过追踪之前提交到主文件系统的更改。在计算机非正常关机的情况下，任何数据丢失都可被重建。

手边的日志文件系统。

* [ext3](ftp://ftp.kernel.org/pub/linux/kernel/people/sct/ext3/) 许多流行的Linux发行版默认的文件系统
* [XFS](http://oss.sgi.com/projects/xfs/) 为高性能和大文件设计的文件系统
* [Btrfs](http://btrfs.wiki.kernel.org/) 校检copy-on-write(写入时复制)文件系统
* [ext4](https://ext4.wiki.kernel.org/index.php/Main_Page) 由ext3增加许多显著特性和扩展进化而来的文件系统

## 357)js浮点数计算问题

**0.1 + 0.2 结果0.30000000000000004**

**对于浮点数的四则运算,几乎所有的编程语言都会有类似精度误差的问题,只不过在C ++/C#/Java 这些语言中已经封装好了方法来避免精度的问题,而 JavaScript 是一门弱类**

**类型的语言,从设计思想上就没有对浮点数有个严格的数据类型,所以精度误差的问题就显得格外突出。**

**因为计算是转换为2进制的。**

**0.1=> 0.0001 1001 1001 1001…(无限循环)**

**0.2=> 0.0011 0011 0011 0011…(无限循环)**

**0.1和0.2转化为二进制之后,变成了一个无限循环的数字**

**对于无限循环的小数,计算机会进行舍入处理。进行双精度浮点数的小数部分最多支持52 位,所以两者相加之后得到这么一串0.0100110011001100110011001100110011001100110011001100因浮点数小数位的限制而截断的二进制数字,这时候,我们再把它转换为十进制就成了 0.30000000000000004。**

**方法一:指定要保留的小数位数(0.1 0.2).toFixed(1) = 0.3;**

**方法二:把需要计算的数字升级(乘以10的n次幂)成计算机能够精确识别的整数,等计算完毕再降级(除以10的n次幂),这是大部分编程语言处理精度差异的通用方法。**

**小数转二进制：**

**小数部分乘2。**

**如0.125 =>0.125\*2=>0.25\*2=>0.5\*2=》1**

**取0 取0 取1 结果：0.001**

## 358)python导入模块

**Python导入模块的方法有两种：import module 和 from module import，区别是前者所有导入的东西使用时需加上模块名的限定，而后者不要。**

正确的代码：  
>>> import Person  
>>> person = Person.Person('dnawo','man')  
>>> print person.Name  
或  
>>> from Person import \*  
>>> person = Person('dnawo','man')  
>>> print person.Name

导入文件为函数则使用第一种方法。

如果有目录：

目录下要有\_\_init\_\_.py（哪怕是个空文件夹）

引入方式变为from zwj.hello import hello（zwj为文件夹名）

## 359)大流量情况

**当突然出现大的访问量，考虑是否是DDOS攻击，再考虑负载均衡。**

## 360)百万级别点的绘制

**高德地图的做法：**

在低级别下并没有绘制出全部的点，而是选取了其中的一部分，选取的策略（可配置）简单来说分如下两步：

1. TopN选取：在高密度区域（上图的东南部，带背景色），排序选出高优先级的N个点，其余点抛弃。
2. 占位检测：绘制之前，如果发现某个点需要占用的矩形空间已经有其他点存在了，则不绘制该点，直接跳过。

通过以上两步，最终显示的点的规模可以被控制在浏览器允许的合理范围以内，效果上也保留了点的分布特征，能够直观的分辨密集区域和稀疏区域。

360)IE9下没有console.log()

某些版本的ie (360等)不打开控制台("F12")是没有console这个对象，在IE中使用了console.log导致的，ajax无法获取到data。删除console.log对象。请养成良好习惯，

用if (window.console) { … }包住console.log()动作。

361）background-size:cover

background-size：100% 100%;---按容器比例撑满，图片变形；

background-size：cover;---把背景图片放大到适合元素容器的尺寸，图片比例不变，但是要注意，超出容器的部分可能会裁掉。

background-position:center;

background-position属性支持 4 个参数值，前两个值用于横坐标，后两个值用于纵坐标

如果只提供一个值，则第二个值为center。如果提供两个值，第一个长度或百分比表示水平偏移，第二个长度或百分比表示垂直偏移。长度或百分比，表示背景图像的左上角相对于背景位置区域左上角的偏移。

## 361）application/json

告诉服务器消息的主题是序列化后的JSON字符串，

最终发送请求为{“tite”:”test”,”sub”:[1,2]}

## object-fit

CSS 属性指定可替换元素的内容应该如何适应到其使用的高度和宽度确定的框

**contain**

被替换的内容将被缩放，以在填充元素的内容框时保持其宽高比。 整个对象在填充盒子的同时保留其长宽比，因此如果宽高比与框的宽高比不匹配，该对象将被添加“[黑边](https://zh.wikipedia.org/wiki/%E9%BB%91%E9%82%8A)”。

**cover**

被替换的内容在保持其宽高比的同时填充元素的整个内容框。如果对象的宽高比与内容框不相匹配，该对象将被剪裁以适应内容框。

**fill**

被替换的内容正好填充元素的内容框。整个对象将完全填充此框。如果对象的宽高比与内容框不相匹配，那么该对象将被拉伸以适应内容框。

**none**

被替换的内容将保持其原有的尺寸。

**scale-down**

内容的尺寸与 none 或 contain 中的一个相同，取决于它们两个之间谁得到的对象尺寸会更小一些。

## 兄弟选择器用

设置表格线 ul>li+li{ border-top:1px solid #c2c2c2 }

## 左边div随右边div高度

.left {

Position: absolute;

Width: 50%;

Height: 100%;

}

.right: {

Margin-left: 50%;

}

## export default

每一个文件之中都只能有一个export default，但是可以有多个export,export default就是为模块指定默认的输出。

export default let a = 1;

使用import...from...可以给模块随便起名字

import aa from '';

export default相当于把对象添加到module的exports中，并且对象的key叫default。

所以export default {} 等于 exports[‘default’] = {}

## Vue keep-alive

在被keep-alive包含的组件/路由中，会多出两个生命周期的钩子:activated 与 deactivated。

在 2.2.0 及其更高版本中，activated 和 deactivated 将会在 <keep-alive> 树内的所有嵌套组件中触发。

**activated调用时机：**

**第一次进入**缓存路由/组件，在mounted后面，beforeRouteEnter守卫传给 next 的回调函数之前调用：

**再次进入**时，因为组件被缓存了，再次进入缓存路由/组件时，不会触发这些钩子：

// beforeCreate created beforeMount mounted 都不会触发

**deactivated：组件被停用(离开路由)时调用**

## Opacity

**子元素会继承opacity,子opacity由opacity(子)\*opacity(父)**

## 368）element-ui的form的button会引起表单请求触发造成请求失败。

## 369) 0/0 == NaN , 1/0 == Infinity

## void 0(为undefined,因为undefined是可以被覆盖的)

371)~~的妙用

~~它代表双非按位取反运算符，

如果你想使用比Math.floor()更快的方法，那就是它了。需要注意，对于正数，它向下取整；对于负数，向上取整；非数字取值为0，它具体的表现形式为：

~~false; // => 0

~~true; // => 1

~~1.9; // => 1

~~-1.9; // => -1

~null; // => 0

~~undefined; // => 0

~~Infinity; // => 0

--NaN; // => 0

~~0; // => 0

372)post可以缓存吗？

根据RFC 2616第9.5节：

“对POST方法的响应不可缓存，除非响应包含适当的Cache-Control或Expires头字段。”

基本上POST不是幂等操作。所以你不能用它来缓存。GET应该是幂等操作，因此它通常用于缓存。

PUT方法本身在语义上意味着放置或创建资源。它是一个幂等操作，但它不会用于缓存，因为在此期间可能发生了DELETE。DELETE方法本身在语义上意味着删除资源。它是幂等操作，但它不会用于缓存，因为PUT可能同时发生。