一、HTML

1. Doctype

是一种标准通用标记语言的文档类型声明，目的是告诉语言解析器要通过什么样的文档类型规范解析文档。

1. WEB标准以及W3C标准

结构、表现和行为分离

标签闭合、标签小写、不随意嵌套、使用外链css和js

1. 严格模式和混杂模式

严格模式的排版和JS运作模式是以该浏览器支持的最高标准运行

混杂模式的页面以宽松的向后兼容的方式显示，模拟老式浏览器的行为以防止站点无法工作。

DOCTYPE不存在或格式不正确会导致混杂模式

1. HTML全局属性

<https://www.w3school.com.cn/tags/html_ref_standardattributes.asp>

1. 标签语义化：

让页面的内容结构化，便于浏览器、搜索引擎解析

在缺少css样式的情况下也可以以一种文档格式显示，并且容易阅读

搜索引擎的爬虫依赖于标记来确定上下文和各个关键字的权重，利于SEO

更容易将网站分块，便于维护理解

1. 常用标签的默认样式

块级元素：html,body,div,ol,ul,p,h1,h2,h3,h4,h5,h6,address,blockquote,form,dd,dl,dt,fieldset,frame,frameset,noframes,center,dir,hr,menu,pre {display:block};

列表元素：

li{display:list-item,list-style:disc}

ol{list-style-type:decimal}

ol ul,ul ol,ul ul,ol ol{margin-top:0;margin-bottom:0}

ol,ul{margin-left:40px}

预格式文本：

i,cite,em,var,address{font-style:italic}

big{font-size:1.17em}

small,sub,sup{font-size:.83em}

sub/sup{vertical-align:sub/super}

u,ins{text-decoration:underline}

标题

h1{font-size:2em;margin:.67em 0}

h2{font-size:1.5em;margin:.75em 0}

h3{font-size:1.17em;margin:.83em 0}

h4,p,blockquote,ul,fieldset,form,ol,dl,dir,menu{margin:1.12em 0}

h5{font-size:.83em;margin:1.5em 0}

h6{font-size:.75em;margin:1.67em 0}

h1,h2,h3,h4,h5,h6,b,strong{font-weight:bolder}

表格

table{display:table}

tr{display:table-row}

thead{display:table-header-group}

tbody{display:table-row-group}

tfoot{display:table-footer-group}

col{display:table-column}

其他

head{display:none}

body{margin:8px;line-height:1.12em}

button,textarea,input,object,select{display:inline-block}

blockquote{margin-left:40px;margin-right:40px;}

pre,tt,code,kbd,samp{font-family:monospace}

pre{write-space:pre}

hr{border:1px inset}

center{text-align:center}

abbr,acronym{font-variant:small-caps;letter-spacing:0.1em}

注：剩余的以后补充

1. 常用标签属性

<https://www.cnblogs.com/yt4561761/p/6538928.html>

1. Meta标签作用及用法

作用：

1. 搜索引擎优化
2. 定义页面使用语言
3. 自动刷新并指向新的页面
4. 实现网页转换时的动态效果
5. 控制页面缓冲
6. 控制网页显示的窗口

用法详见：<http://www.cnblogs.com/qiumohanyu/p/5431859.html>

1. SEO
2. 合理的title、description、keywords：

权重title>description>keywords

Title:强调重点

Description:高度概括页面内容

Keywords:列举出重要关键词

1. 语义化的HTML代码，符合W3C规范
2. 重要内容HTML代码放在最前
3. 重要内容不要用js输出
4. 少用iframe
5. 非装饰性图片必须加alt
6. 提高网站速度
7. 浏览器兼容

https://www.cnblogs.com/simba-lkj/p/6137494.html

1. HTML5离线缓存机制
2. 当浏览器访问一个包含manifest特性的文档时，如果应用缓存不存在，浏览器会加载文档，然后获取所有在清单文件中列出的文件，生成应用缓存的第一个版本。
3. 对该文档的后续访问会使浏览器直接从应用缓存中加载文档或其他在清单文件中列出的资源。此外，浏览器还会向window.applicationCache对象发送一个checking事件，在遵循合适的HTTP缓存规则前提下，获取清单文件。
4. 如果当前缓存的清单副本是最新的，浏览器将向applicationCache对象发送一个noupdate事件，到此，更新过程结束。（注：如果在服务器修改了任何缓存资源，同时也应该修改清单文件，这样浏览器才能知道它需要重新获取资源）。
5. 如果清单文件已经改变，文件中列出的所有文件包括通过调用applicationCache.add()方法添加到缓存中的那些文件都会被获取并放到一个临时缓存中，遵循适当的HTTP缓存规则。对于每个加入到临时缓存中的文件，浏览器会向applicationCache对象发送一个progress事件。如果出现任何错误，浏览器会发送一个error事件，并暂停更新。
6. 一旦所有文件都获取成功，它们会自动移送到真正的离线缓存中，并向applicationCache对象发送一个cached事件。由于文档已经被从缓存加载到浏览器中，所以更新后的文档不会重新渲染，直到页面重新加载。

详见http://www.ituring.com.cn/article/213932

1. CSS

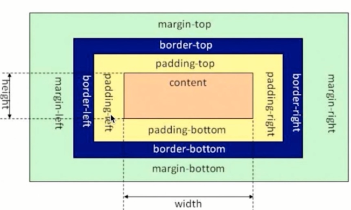
1.BFC(Block Formatting Context)：

BFC即块级格式化上下文：是一个独立的布局环境，其中的元素布局不受外界影响；内部的box(css布局的基本单位)会在垂直的方向一个接一个的放置；box垂直方向的距离由margin决定，属于同一个BFC的两个相邻box的margin会发生重叠；每个元素的margin box的左边，与包含块border box的左边相接触(对于从左往右的格式化，否则相反)，即使存在浮动也是如此；BFC的区域不会与浮动区域的box重叠；浮动元素参与BFC高度的计算。

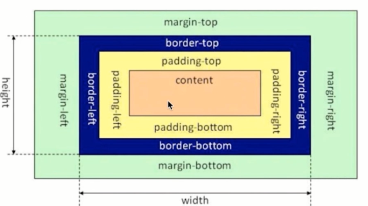
BFC的创建:

1. float属性不为none；
2. position为absolute或fixed
3. display为inline-block,table-cell,table-caption,flex,inline-flex;
4. Overflow不为visible
5. 根元素
6. 盒模型

标准盒模型(box-sizing:content-box)



IE盒模型(box-sizing:border-box)



1. css选择器
2. 标签选择器
3. 类选择器
4. ID选择器
5. 通配符
6. 组合选择器
7. 子元素选择器（>）
8. 群组选择器
9. 继承选择器
10. 伪类选择器
11. 兄弟选择器
12. 属性选择器

优先级：!important>行内样式>ID选择器>类选择器>标签选择器>通配符>继承选择器>浏览器默认属性

Css继承：https://www.cnblogs.com/super86/p/4121824.html

At规则：<https://blog.csdn.net/scorpio_h/article/details/86799790#>

1. css伪类、伪元素

伪类：用于向某些选择器设置特殊效果(弥补常规选择器的不足)

详见http://www.w3school.com.cn/css/css\_pseudo\_classes.asp和 <http://www.imooc.com/wenda/detail/460416>

伪元素：用于向某些选择器设置特殊的效果(创建一个有内容的虚拟容器，如清浮动，同时只能使用一个伪元素)

详见：<http://www.w3school.com.cn/css/css_pseudo_elements.asp>

1. HTML文档流

元素排版过程中，元素会自动从左往右，从上往下的流式排列。

定位：<http://www.w3school.com.cn/css/css_positioning.asp>

1. 雪碧图实现原理

将小图标和背景图像合并到一张图片上，利用CSS的背景定位来显示需要的图片部分

详见：<https://www.cnblogs.com/xuzewen/p/9310007.html>

1. 水平垂直居中方案

水平：

1. 父元素{text-align:center}，子元素{display:inline-block}
2. 子元素{display:table;margin:0 auto}
3. 父元素{position:relative}，子元素{display:absolute;left:50%;transform:translateX(-50%)}
4. ①父元素{display:flex;justify-content:center}

②父元素{display:flex}，子元素{margin:0 auto}

垂直：

1. 父元素{display:table-cell;vertical-align:middle}
2. 父元素{position:relative}，子元素{position:absolute;top:50%;transform:translateY(-50%)}
3. 父元素{display:flex;align-items:center}

详见：<https://segmentfault.com/a/1190000016389031>

1. sass和less

Sass(scss)：是一种动态语言，属于缩排语法

详见：<https://www.sass.hk/guide/>

Less：是一种动态语言...

详见：https://less.bootcss.com/

Sass和less的异同

不同：less基于JavaScript，是在客户端处理的，变量用”@”

Sass基于Ruby，是在服务器端处理的”$”

相同：

1.混入(Mixins)——class中的class；

2.参数混入——可以传递参数的class，就像函数一样；  
 3.嵌套规则——Class中嵌套class，从而减少重复的代码；

4.运算——CSS中用上数学；  
 5.颜色功能——可以编辑颜色；  
 6.名字空间(namespace)——分组样式，从而可以被调用；  
 7.作用域——局部修改样式；  
 8.JavaScript 赋值——在CSS中使用JavaScript表达式赋值。

详见：<https://www.cnblogs.com/hope666/p/6791790.html>

1. css模块化，按需加载，防止css阻塞渲染

Css模块化具体看团队

按需加载：详见Mixins

防止css阻塞渲染：浏览器大概渲染流程：浏览器自上而下解析目标HTMl文件=>HTML解析器将HTML结构转换为基础的DOM tree=>CSS解析器将CSS解析为CSSOM=>CSSOM和DOM合并构成渲染树=>计算渲染树中每个节点的信息绘制页面；因此当CSSOM没有构建完成时，页面不会渲染，即css阻塞渲染。

解决方法：精简CSS，尽早提供，利用媒体查询和类型

1. css动画

## @keyframes和animation

渐变、移动、旋转、缩放

详见：<https://www.runoob.com/css3/css3-tutorial.html>

1. css浏览器兼容写法及不同API在不同浏览器下的兼容情况

加前缀，剩下的以后总结补充

后期可以借助postcss插件辅助生成

1. 响应式布局方案
2. 媒体查询

使用@media媒体查询可以针对不同的媒体类型和不同的屏幕大小定义不同的样式，从而达到自适应的效果

缺点：多套样式代码较繁琐

1. 百分比

通过百分比单位使浏览器中组件的宽和高随着浏览器的变化而变化

1. Px

构成影响的小方点，具有相对大小

1. Em

相对尺寸，相对于当前对象内文本的font-size(如果当前对象内文本的font-size计量单位也是em，则当前对象内文本的font-size的参考对象为父元素文本的font-size)

1. Rem

rem单位是相对于根元素html的font-size来决定大小的，根据视图容器的大小动态的改变font-size。

1. Vw/vh

100vw为视窗的宽度，100vh为视窗的高度

(opera浏览器不兼容)

剩下的待补充

13.CSS常用效果

详见：<https://github.com/Bpimp/draft.git>

1. JavaScript
2. 数据类型

基本数据类型：null、string、number、undefined、boolen。

引用数据类型：对象数据类型（数组、正则、对象）、函数数据类型

注：ES6新增symbol

Symbol：表示独一无二的值，在为对象添加新的属性时，可以防止属性名的冲突。

Symbol实现：https://segmentfault.com/a/1190000015262174#articleHeader4

1. 数据结构

常用数据结构：Array、Object

不常用数据结构：Set、WeakSet、Map、WeakMap

结构型数据：JSON

特殊结构：Function

详见：<https://developer.mozilla.org/zh-CN/>

1. JavaScript中的变量在内存中的具体存储形式

基本数据类型(值类型)保存在栈内存，引用数据类型(引用类型)实际上是一个指针，这个指针保存在栈中，但是这个指针指向的对象则保存在堆内存中；访问时，基本数据类型访问的是它们实际保存的值。引用数据类型则是按引用访问，先从栈中读取内存地址，然后找到保存在堆内存中的值；复制时，基本数据类型会在栈中创建一个新值，然后把值复制到为新变量分配的空间中，引用类型的变量则是复制存储在栈中的指针，将指针复制到栈中为新变量分配的空间中，而这个指针副本和原指针指向的是同一个堆内存中的对象，两个变量实际上将引用同一个对象，因此改变其中一个将影响到另外一个；检测基本数据类型为typeof，引用数据类型为instanceof。

1. 基本数据类型对应的内置对象

字符串：String();

数组：Array();

数字：Number();

布尔值：Boolean();

...

1. null和undefined区别

Null：Null类型，代表”空值”，代表一个空对象指针，是一个特殊的对象值。

Undefined：当一个变量被声明但未初始化时，就是undefined，JavaScript独有。

1. JavaScript数据类型的判断方法
2. typeof()无法区分引用数据类型
3. Instanceof()无法区分原始数据类型
4. Constructor
5. Object.prototype.toString.call()
6. Jquery.type()

详见:https://www.cnblogs.com/dushao/p/5999563.html

1. JavaScript隐式类型转换

详见：<https://www.cnblogs.com/chenmeng0818/p/5954215.html>

1. JavaScript数字相关

Js的数字类型采用IEEE 754标准的64-bits的双精度存储，当数字到了计算机的最底层都会转换成二进制，但二进制无法准确表示这种包含小数点的数据，其本质是将浮点数转换成了用二进制表示的最接近的近似值，并且js也没有相应的封装类来处理浮点数运算。

精度丢失解决方法：整数不超过Math.pow(2,53)(最大数字)。注：最大安全数字为Math.pow(2,53)-1，（个人理解）。

小数先乘以倍数转换为整数，运算后在除以相同的倍数

1. **原型和原型链**

Js原型模式：js有一个根对象(Object.prototype)，通过拷贝这个根对象创建新的对象。详见：JavaScript设计模式与开发实践

Js原型规则：

1. 所有引用类型(数组、对象、函数)都具有对象特性，即可自由扩展属性
2. 所有的引用类型(数组、对象、函数)，都有一个\_proto\_属性(隐式原型)，属性值是一个普通的对象
3. 所有的函数都具有一个prototype(显示原型)，属性值也是一个普通对象
4. 所有的引用类型，其隐式原型指向其构造函数的显示原型：(obj.\_proto\_===Object.prototype)
5. 当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的\_proto\_(即它的构造函数的prototype)中去寻找
6. instanceof的底层实现原理

语法：Object instanceof constructor

Instanceof运行的过程中遍历左边对象的原型链，如果找到右边对象的prototype返回true，否则返回false。

手写实现instanceof：

function instanceOf(*left*,*right*) {  
 let rightproto=*right*.prototype;  
 *left*=*left*.\_\_proto\_\_;  
 while(true){  
 if(*left*===null){  
 return false;  
 }else if(*left*===rightproto){  
 return true;  
 }  
 *left*=*left*.\_\_proto\_\_;  
 }  
}

1. 继承的实现方法

详见：http://www.ruanyifeng.com/blog/2011/06/designing\_ideas\_of\_inheritance\_mechanism\_in\_javascript.html

1. 原型(类)继承：将父类的实例作为子类的原型

function Basepro(){  
 this.property=true;  
}  
Basepro.prototype.getValue=function(){  
 return this.property;  
};  
function Newpro(){  
}  
Newpro.prototype=new Basepro();

Newpro.prototype.constructor=Newpro;  
let *instance*=new Newpro();  
*console*.log(*instance*.getValue());

优点：简单，易于实现；父类新增原型方法或属性，子类都能访问到

缺点：来自原型对象的所有属性被所有实例共享，导致对一个实例的修改会影响到其它实例；创造子类实例时，无法向父类构造函数传参；等等

1. 构造继承：借用call或apply函数调用父类的构造函数

function Basepro(){  
 this.name ="Tom";  
}  
function Newpro(){  
 Basepro.call(this)  
}  
let *rat* = new Newpro();  
*rat*.name = "Jerry";  
let *cat* = new Newpro();  
*console*.log(*cat*.name);

优点：解决了子类实例共享父类引用类型属性的问题；创建子类实例时可以向父类传递参数；可以实现多继承

缺点：实例不是父类的实例，只是子类的实例，只能继承父类的实例属性和方法，不能继承原型属性和方法；方法都在构造函数中定义无法实现函数复用。

1. 组合继承：将原型链继承和构造继承相结合

function Basepro(*name*){  
 this.name=*name*;  
 this.type ="cat";  
}  
Basepro.prototype.getName=function () {  
 *console*.log(1,this.name);  
};  
function Newpro(*name*){  
 Basepro.call(this,*name*)  
}  
Newpro.prototype=new Basepro();  
Newpro.prototype.constructor=Newpro;  
  
let *rat* = new Newpro("Jerry");  
*rat*.type="rat";  
let *cat* = new Newpro("Tom");  
*console*.log(*rat*.type,*cat*.type);  
*rat*.getName();  
*cat*.getName();

优点：结合了一和二两种方法的优点，解决了不足之处

缺点：调用了两次父类的构造函数

1. 拷贝继承：将父类的所有属性拷贝进子类中

function Basepro(*name*,*type*){  
 this.name=*name*;  
 this.type=*type*;  
 this.eat=function () {  
 *console*.log(this.name+'正在吃');  
 }  
}  
Basepro.prototype.run=function () {  
 *console*.log(1);  
};  
function Newpro(*name*,*type*){  
 Basepro.call(this,*name*,*type*)  
}  
function extend(*child*,*parent*) {  
 let p=*parent*.prototype;  
 let c=*child*.prototype;  
 for (let i in p){  
 c[i]=p[i];  
 }  
}  
let *cat*=new Newpro('Tom','cat');  
let *rat*=new Newpro('Jerry','rat');  
extend(Newpro,Basepro);  
*console*.log(*rat*,*cat*);

1. ES6 class继承

使用extends constructor super等方法实现

详见react

注：继承方法较多，各种方法各有优缺点，根据环境自由选择，主要原则是子类可以继承父类的一些功能，子类不影响父类，实现代码复用

**案例：？？？**

1. New生成对象的过程
2. 创建一个新的对象
3. 新对象的\_\_proto\_\_属性指向构造函数的原型对象
4. 使this指向变更为新的对象
5. 执行构造函数内部的代码，将属性添加给新对象中的this属性
6. 返回新对象

手动实现方法：

function myNew() {  
 let obj=*Object*();  
 let Constructor=[].shift.call(*arguments*);  
 obj.\_\_proto\_\_=Constructor.prototype;  
 let result=Constructor.apply(obj,*arguments*);  
 return typeof result==='object'?result:obj;  
}

1. ES6 class相关

Class基本结构：

class className{  
 constructor(*name*){  
 this.name=*name*;  
  
 }  
 func(){  
 *console*.log('something');  
 }  
}  
let *p1*=new className('Tom');

1. 词法作用域和动态作用域

词法(静态)作用域：在函数定义的时候决定(JavaScript)

动态作用域：在函数调用的时候决定

let *val*=1;  
function foo() {  
 *console*.log(*val*);  
}  
function bar() {  
 let val=2;  
 foo();  
}  
bar();//静态作用域返回1，动态作用域返回2

参考：https://www.cnblogs.com/guaidianqiao/p/7762070.html

1. JavaScript作用域和作用域链

作用域：可访问变量，对象，函数的合集

详见：<https://www.runoob.com/js/js-scope.html>

作用域链：创建一个函数时，其所有局部作用域层层嵌套，包括window全局作用域组成的集合。

详见Js高级程序设计(第3版)

1. JavaScript执行上下文栈

当浏览器执行全局代码之前，JS引擎就会创建一个栈来储存管理所有的执行上下文对象，在全局执行上下文(window)确定后，将其添加到栈的最底层，当我们调用了一个函数，程序的执行流程会进入到该函数中，并创建一个新的执行上下文，并将这个上下文推入到执行栈顶。如果在该函数中又调用了一个函数，就会执行同样的操作，如果当前函数执行完成，就会将栈顶的对象移除，在所有的代码执行完后，栈中只剩下window。(后进先出)

1. This的原理及其应用场景

原理：<http://www.ruanyifeng.com/blog/2018/06/javascript-this.html>

应用场景: (1) 函数调用 指向全局对象window

1. 对象方法调用 指向该对象
2. 构造函数调用 指向新创建的对象
3. Call、apply或bind中调用 指向所传入的参数
4. 闭包

let foo=function () {  
 let a=10;  
 return function bar(*x*) {  
 if(*x*>a){  
 *console*.log(*x*);  
 }  
 }  
};  
let fn=foo();  
fn(15);

这个例子中，在函数foo中又定义了bar函数，并且内部函数bar可以引用外部函数foo的参数和变量，当foo执行时会返回bar函数，相关参数和变量都保存在返回的函数中，不会被销毁，这种结构被称为闭包。

闭包的作用见《JavaScript设计模式与开发实践》及https://blog.csdn.net/qq\_21132509/article/details/80694517及

http://www.cnblogs.com/TomXu/archive/2012/01/12/2308594.html#!comments

1. 堆栈溢出和内存泄漏

堆栈溢出：向堆栈中固定大小的局部数据块内写入了过多的数据，导致数据越界，覆盖了别的数据。常发生在递归中。

部分解决方法原理：http://www.ruanyifeng.com/blog/2015/04/tail-call.html

内存泄漏：用动态存储分配函数内存空间，在使用后未释放，导致内存单元一直被占用，造成内存浪费。

1. 模块化

把一个相对独立的功能封装成一个单独的文件，可输入制定的依赖，输出制定的函数，供外界调用，方便不同的项目进行重复调用。

作用:(1)解决项目中的全局变量污染问题。

1. 有利于多人协同开发
2. 方便代码复用和维护
3. 解决文件依赖问题，无需关注引用文件的顺序
4. Try...catch...finally执行机制

Try中为需要被执行的语句

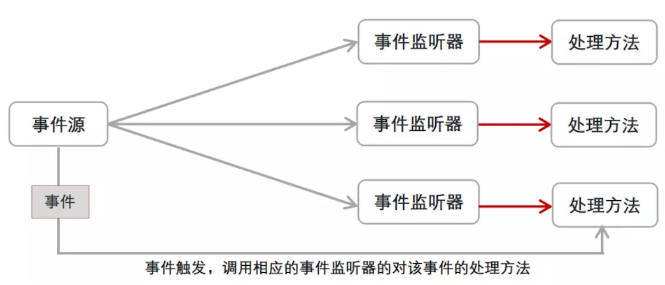
Catch当try中的语句发生错误时，才会执行这里的代码，其参数记录着try

中代码的错误信息

Finally在try语句块之后执行的语句块，无论是否有异常或捕获这里的语句都将执行

详见：<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/try...catch>

1. JavaScript异步编程
2. 回调函数：先执行程序的主要逻辑，将耗时的操作推迟进行。优点是简单、容易理解，但不利于代码的维护和阅读，各个部分之间高度耦合，流程较乱，且每个任务只能指定一个回调函数。
3. 事件监听：当事件源的某个事件被触发时，就会调用这个事件对应的事件监听器的处理事件的方法。优点是可以绑定多个事件，每个事件可以指定多个回调函数，而且可以”去耦合”，有利于实现模块化，缺点是整个程序都要变成事件驱动型，运行流程会变得不清晰。



1. 发布/订阅者模式：详见《JavaScript设计模式与开发实践》
2. Promise对象：

new *Promise*(function (*resolve*,*reject*) {  
 //异步任务  
 //resolve(内部可以传参，后同)成功or reject()失败  
}).then(()=>{  
 //成功后执行的任务  
 },()=>{  
 //失败后执行的任务  
 });

Promise Chain:多个then()的情况

Then函数执行后会返回一个新的Promise对象。如果then没有传入处理函数，就会返回一个继承了上一个处理状态的Promise对象；如果then传入处理函数，那么默认返回一个resolve状态的Promise对象；如果then传入了处理函数，通过处理函数显式的return了一个新的Promise，那么返回这个显式的Promise对象

注：无法中途终止后续任务的执行，可以使用catch方法可以捕获前面出现的任何一个错误，但会和then中的第二个参数冲突，而且catch之后的then还是会执行

Promise.all:实现串行

let p1=new *Promise*((*resolve*,*reject*)=>{  
  
});  
let p2=new *Promise*((*resolve*,*reject*)=>{  
  
});  
*Promise*.all([p1,p2]).then(*arr*=>{  
 //p1，p2执行完之后再执行  
 //arr为p1和p2中通过resolve或reject传入的参数的集合  
});

Promise.race([p1,p2]).then(arr=>{

//p1或p2之中一个执行完就执行

})

Promise.race:同上

Promise.resolve()/reject():返回一个相应状态的Promise对象

1. Async和await (ES7相当于Promise的语法糖)

Async作为关键字放在函数之前，表示此函数为异步函数，调用和平时一样，返回一个Promise对象，如果return未返回Promise语句，系统会自动将返回值包装成Promise的resolved值

async function fn(*mes*) {  
 if(*mes*){  
 return *mes*;  
 }else{  
 return 'failure';  
 }  
}  
fn(true).then(*res*=>{  
 *console*.log(*res*);  
});  
fn(false).then(*res*=>{  
 *console*.log(*res*);  
},*err*=>{  
 *console*.log(*err*)  
});  
/\*fn(false).catch(err=>{  
 console.log(err);  
})\*/

Await：只能在async函数中使用，后面接Promise对象或具有可调用then方法的对象，作用是让JavaScript等待直到所接的任务完成后再继续执行，但引擎可以同时执行其他任务

function double(*num*) {  
 return new *Promise*((*resolve*,*reject*)=>{  
 setTimeout(()=> {  
 *resolve*(*num* \* 2)  
 },2000)  
 })  
}  
async function testresult() {  
 let a=await double(10);  
 let b=await double(20);  
 let c=await double(30);  
 *console*.log(a,b,c);  
}  
*console*.log(1);  
testresult();s

EventLoop机制：即事件循环机制，是指浏览器或Node的一种解决JavaScript单线程运行时不会阻塞的一种机制，也就是异步的原理

详见：<https://juejin.im/post/5c3d8956e51d4511dc72c200#heading-15>

注：理解的马马虎虎，需要找时间深入理解

1. 宏任务和微任务

宏任务：包括script(整体代码),setTimeout,setInterval,setImmediate,I/O,UI rendering。

微任务：包括Promise,Object.observe(基本废除),MutationObserver,process.nextTick等

大概执行顺序：先执行同步代码，遇到异步宏任务则将异步宏任务放入到宏任务队列中，遇到异步微任务则将异步微任务放入微任务队列中，当所有同步代码执行完毕后，再将异步微任务从队列中调入主线程执行，微任务执行完毕后再将异步宏任务从队列中调入主线程执行，一直循环直至所有任务执行完毕。

1. Node和浏览器EventLoop的差异

Node的EventLoop详见：https://juejin.im/post/5baf37835188255c6c624d38

1. ECMAScript和JavaScript的关系

ECMAScript是JavaScript的标准或规格，JavaScript是ECMAScript的实现

1. Es5和es6的语法规范

略，平常开发时逐步了解

1. JavaScript全局对象、全局函数、全局属性

详见：<http://www.w3school.com.cn/jsref/jsref_obj_array.asp>

1. JavaScript高阶函数

高阶函数：一个接受函数作为参数或将函数作为输出返回的函数

如：map，reduce，filter等

1. setInterval

注意事项:(1) 不考虑任务执行本身所消耗的时间，所以两次执行间的时间间隔小于指定的时间

1. 具有累积效应，如果某个操作特别耗时，超过了setInterval的时间间隔，排在后面的操作会被累积起来，然后在很短的时间内连续触发，可能造成性能问题

使用setTimeout实现setInterval

function interval(*func*,*time*) {  
 let interv=function () {  
 *func.call*(null);  
 setTimeout(interv,*time*);  
 };  
 setTimeout(interv,*time*);  
}

1. 正则

详见：<http://www.w3school.com.cn/jsref/jsref_obj_regexp.asp和>

<https://www.cnblogs.com/zxin/archive/2013/01/26/2877765.html>

1. JavaScript异常处理的方式，统一的异常处理方案
2. try...catch:无法捕捉到语法错误和异步错误，只能捕捉到运行时非异步错误，且需要把代码块封装起来
3. Window.onerror:无法捕捉到语法错误，可以捕捉到运行时的异步和非异步错误
4. Promise错误：在最后写上catch，或使用全局异常捕获事件unhandledrejection
5. 数组去重、扁平化
6. 去重:

A:

function uniq(*arr*){  
 let newarr=[];  
 for(let i=0;i<*arr*.length,i++){  
 if(temp.indexOf(*arr*[i])===-1){  
 newarr.push(*arr*[i]);  
 }  
 }  
 return newarr;  
}

B:

function uniq(*arr*){  
 let obj={};  
 let newarr=[];  
 for(let i=0;i<*arr*.length;i++){  
 if(!obj[*arr*[i]]){  
 obj[*arr*[i]]=*arr*[i];  
 newarr.push(obj[*arr*[i]]);  
 }  
 }  
 return newarr;  
}

C:

function uniq(*arr*) {  
 *arr*.sort();  
 let newarr=[*arr*[0]];  
 for(let i=1;i<*arr*.length;i++){  
 if(*arr*[i]!==newarr[newarr.length-1]){  
 newarr.push(*arr*[i]);  
 }  
 }  
 return newarr;  
}

D:

function uniq(*arr*) {  
 let newarr=[];  
 for(let i=0;i<*arr*.length;i++){  
 for(let j=i+1;j<*arr*.length;j++){  
 if(*arr*[i]===*arr*[j]){  
 i++;  
 j=i;  
 }  
 }  
 newarr.push(*arr*[i]);  
 }  
 return newarr;  
}

E:

function uniq(*arr*) {  
 let newarr=*Array*.from(new *Set*(*arr*));  
 return newarr;  
}

F:

function uniq(*arr*) {  
 let newarr=[];  
 for(let i=0;i<*arr*.length;i++){  
 if(!newarr.includes(*arr*[i])){  
 newarr.push(*arr*[i])  
 }  
 }  
 return newarr;  
}

G:

function uniq(*arr*) {  
 let newArr = *arr*.reduce((*pre*,*cur*)=>{  
 if(!*pre*.includes(*cur*)){  
 return *pre*.concat(*cur*)  
 }else{  
 return *pre* }  
 },[]);  
 return newArr;  
 }

1. 扁平化

A:

function flat(*arr*) {  
 return *arr*.reduce((*result*,*item*)=>{  
 return *result*.concat(*Array*.isArray(*item*)?flat(*item*):*item*)  
 },[])  
}

B:

function flat(*arr*) {  
 return *arr*.toString().split(',').map((*item*)=>{  
 return *Number*(*item*);  
 })  
}

注：toString()可换成join(‘,’)

C:

function flat(*arr*) {  
 let newArr=[];  
 *arr*.map(*item*=>{  
 if(*Array*.isArray(*item*)){  
 newArr=newArr.concat(flat(*item*));  
 }else{  
 newArr.push(*item*);  
 }  
 });  
 return newArr;  
}

D:

[].concat(...arr);

E:

Flat方法，有兼容性问题，且会移除空值。

1. 深拷贝方法

A:

function deepCopy(*obj*) {  
 let newObj=*Array*.isArray(*obj*)?[]:{};  
 if(*obj* && typeof *obj*==='object'){  
 for(key in *obj*){  
 if(*obj*.hasOwnProperty(key)){  
 if(*obj*[key] && typeof *obj*[key]==='object'){  
 newObj[key]=deepCopy(*obj*[key]);  
 }else{  
 newObj[key]=*obj*[key];  
 }  
 }  
 }  
 }  
 return newObj;  
}

B:

function deepCopy(*obj*) {  
 let \_obj=*JSON*.stringify(*obj*);  
 newObj=*JSON*.parse(\_obj);  
 return newObj;  
}

C:

Jquery中的extend方法

D:

Lodash中的\_.cloneDeep()方法

1. 函数柯里化工具函数

const \_curry=(*fn*,*placeholder*='\_')=>{  
 \_curry.placehoder=*placeholder*;

//函数参数小于等于1，直接执行  
 if(*fn*.length<=1){  
 return *fn*;  
 }  
 let argsList=[];  
 const generator=(...*args*)=>{  
 let currentPlacehoderIndex=-1;  
 *args*.forEach(*arg*=>{  
 let placehoderIndex=argsList.findIndex(  
 *item*=>*item*===\_curry.placehoder  
 );  
 if(placehoderIndex<0){  
 currentPlacehoderIndex=argsList.push(*arg*)-1  
 }else if(placehoderIndex!==currentPlacehoderIndex){  
 argsList[placehoderIndex]=*arg*;  
 }else{  
 argsList.push(*arg*);  
 }  
 });  
 let realArgsList=argsList.filter(*arg*=>*arg*!==\_curry.placehoder);  
 if(realArgsList.length===*fn*.length){  
 return *fn*(...argsList)  
 }else if(realArgsList.length>*fn*.length){  
 throw new *Error*('超出初始函数参数最大值')  
 }else{  
 return generator  
 }  
 };  
 return generator;  
};

其他版本详见：[https://juejin.im/post/598d0b7ff265da3e1727c491#heading-0](https://juejin.im/post/598d0b7ff265da3e1727c491" \l "heading-0)

1. 防抖和节流工具函数
   1. 防抖工具函数：多次触发事件后，事件处理函数只执行一次，并且是在触发操作结束时执行

function throttling(*fn*,*delay*) {  
 let timer;  
 return function (*arg*) {  
 clearTimeout(timer);  
 timer=setTimeout(()=>{  
 *fn*(*arg*);  
 },*delay*)  
 }  
}

* 1. 节流工具函数：持续触发事件时，保证一定时间内只调用一次处理函数。

function dealWith(*fn*,*delay*,*mastExec*) {  
 let timer;  
 let lastTime=new *Date*();  
 return function (*arg*) {  
 let now=new *Date*();  
 clearTimeout(timer);  
 if(now-lastTime<*mastExec*){  
 timer=setTimeout(()=>{  
 *fn*(*arg*);  
 lastTime=now;  
 },*delay*);  
 }else{  
 *fn*(*arg*);  
 lastTime=now;  
 }  
 }

}

1. Sleep函数

让线程休眠，等到指定时间再重新唤起

function sleep(*pauseTime*) {  
 let start=new *Date*().getTime();  
 while(true){  
 if(new *Date*().getTime()-start>*pauseTime*){  
 break;  
 }  
 }  
}

1. 手动实现call、apply、bind函数
   1. Call：将目标函数的this指向传入的第一个参数，参数总数不定，且立即执行

*Function*.prototype.\_call=function (*obj*) {  
 let args=*Array*.prototype.slice.apply(*arguments*,[1]);  
 *obj*.fn=this;  
 *obj*.fn(...args);  
 delete *obj*.fn;  
}

* 1. Apply

*Function*.prototype.\_apply=function (*obj*,*arr*) {  
 *obj*.fn=this;  
 if(!*arr*){  
 *obj*.fn();  
 }else{  
 *obj*.fn(...*arr*);  
 }  
 delete *obj*.fn;  
};

* 1. Bind

*Function*.prototype.\_bind=function (*context*) {  
 let \_this=this;  
 let args=*Array*.prototype.slice.call(*arguments*,1);  
 let func=function () {};  
 let bound=function(){  
 \_this.apply(this instanceof \_this?this:*context*,args.concat(*Array*.prototype.slice.call(*arguments*)))  
 };  
 func.prototype=this.prototype;  
 bound.prototype=new func();  
 return bound;  
};

1. Promise和async await

Promise(符合Promise/A+规范)

详见：<https://juejin.im/post/5aa7868b6fb9a028dd4de672>

1. EventEmitter事件发布、订阅模式实现

class EventEmitter{  
 constructor(){  
 this.subs={}  
 }  
 on(*event*,*cb*){  
 (this.subs[*event*]||(this.subs[*event*]=[])).push(*cb*)  
 }  
 trigger(*event*,...*args*){  
 this.subs[*event*] && this.subs[*event*].forEach(*cb*=>{  
 *cb*(...*args*)  
 })  
 }  
 once(*event*,*onceCb*){  
 const cb=(...*args*)=>{  
 *onceCb*(...*args*);  
 this.off(*event*,*onceCb*)  
 };  
 this.on(*event*,cb)  
 }  
 off(*event*,*offCb*){  
 if(this.subs[*event*]){  
 let index=this.subs[*event*].findIndex(*cb*=>*cb*===*offCb*);  
 this.subs[*event*].splice(index,1);  
 if(!this.subs[*event*].length){  
 delete this.subs[*event*]  
 }  
 }  
 }  
}

详见：<https://juejin.im/post/5b987d92e51d450e51625080>

1. 双向绑定

界面的操作能实时反映到数据，数据的变更也能实时展现到界面

1. vue中的Object.defineProperty()

详见：<https://segmentfault.com/a/1190000011225943>

1. Angular的digest循环和dirty-checking，angular没有接触过
2. JSON.stringify和JSON.parse的实现

详见：<https://juejin.im/post/5c9c3989e51d454e3a3902b6>

或<https://juejin.im/post/5cef46226fb9a07eaf2b7516>

1. 模板引擎

详见：<https://juejin.im/post/5b3b93115188251afa62ad46>

1. 懒加载、下拉刷新、上拉加载、预加载
   1. 懒加载：在长网页中延迟加载图像，在用户滚动到可视区外之前，可视区域外的图片不会加载
   2. 预加载：将所有所需的资源提前请求加载到本地，在需要用的时候直接从缓存获取资源

详见：<https://juejin.im/post/5b0c3b53f265da09253cbed0>原文有点小问题，需要修改

* 1. 下拉刷新和上拉加载：晚上回去看视频

1. 常见数据结构
   1. 数组：略
   2. 栈：是一种遵从先进后出原则的有序集合，新添加和待删除的元素都保存在栈的同一端，称作栈顶，另一端叫做栈底，栈被广泛用在编程语言的编译器和内存中，用作保存变量、方法调用等。
   3. 队列：类似于栈，先进先出
   4. 链表：链表中的每一个元素由一个储存元素本身的节点和一个指向下一个元素的指针组成
   5. 集合：是由一组无序且唯一的项组成的，如ES6中的Set
   6. 字典和散列表：类似于Set，不过Set存储的是[值,值]对，而字典存储的是[键,值]对
   7. 树：由n个有限节点组成一个具有层次关系的集合，平常应用中用的大多是二叉树
   8. 图：由结点的有穷集合V和边的集合E组成，在图结构中常常将结点称为顶点，边是顶点的有序偶对，若两个顶点之间存在一条边，就表示这两个顶点具有相邻关系

详见：<https://juejin.im/post/5be6f4c66fb9a049e231857e>

1. 数据结构的知识内容暂时不作为掌握重点，有空时百度了解一下
2. 算法的时间复杂度和空间复杂度

详见：<https://juejin.im/post/5c2a1d9d6fb9a04a0f654581>

1. 排序算法

详见：<https://juejin.im/post/57dcd394a22b9d00610c5ec8>

1. 递归和循环

直接百度

1. 部分算法
   1. 回溯算法(八皇后)

let num = 8;

let iCount=0;

let n = num \* num;

let Ul = document.querySelector('#box');

for (let i = 0; i < n; i++) {

let aLi = document.createElement('li');

aLi.index = -1;

aLi.innerHTML = aLi.index;

Ul.appendChild(aLi);

}

let aLi = Ul.querySelectorAll('li');

for (let i = 0; i < num; i++) {

for (let j = 0; j < num; j++) {

aLi[i \* num + j].x = j;

aLi[i \* num + j].y = i;

*//aLi[i\*8+j].innerHTML='('+j+','+i+')';*

}

}

setQueen(0);

console.log(iCount);

function setQueen(iQueen) {

if(iQueen===num){

iCount++;

return ;

}

for (let i = 0; i < num; i++) {

if (aLi[iQueen \* num + i].index === -1) {

aLi[iQueen \* num + i].index = iQueen;

aLi[iQueen \* num + i].innerHTML = iQueen;

let x = aLi[iQueen \* num + i].x;

let y = aLi[iQueen \* num + i].y;

for (let j = 0; j < aLi.length; j++) {

if (aLi[j].index===-1&&(aLi[j].x === x || aLi[j].y === y || aLi[j].x - aLi[j].y === x - y || aLi[j].x + aLi[j].y === x + y)) {

aLi[j].index = iQueen;

aLi[j].innerHTML = iQueen;

}

}

setQueen(iQueen+1);

}

*//回溯*

for (let j = 0; j < aLi.length; j++) {

if (aLi[j].index===iQueen) {

aLi[j].index=-1;

}

}

}

}

* 1. 贪心算法等详见：<https://juejin.im/post/594dfe795188250d725a220a>

计算机基础

一、编译原理

1. 代码：程序员用开发工具所支持的语言写出来的源文件，是一组由字符、符号或信号码元以离散形式表示信息的明确的规则体系
2. 编译原理
3. 正则匹配原理和性能优化
4. AST：抽象语法树，可以将js代码解析成一个树状结构，接近计算机底层的原理。
5. Base64编码原理：基于64个可打印的字符来表示二进制的数据的一种方法
6. 各种进制的转换方法及表示方法(十进制转换成n进制的：toString(n)，n进制转换成十进制：parseInt(str,n))

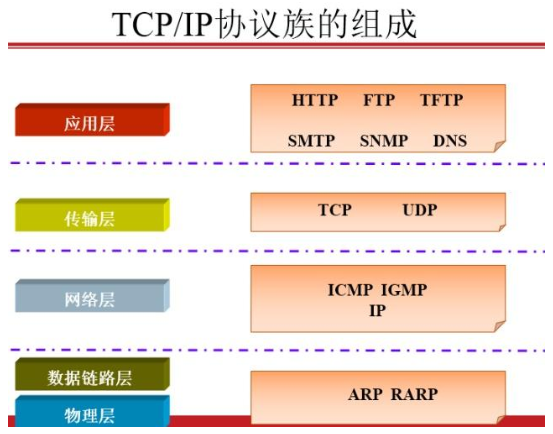
二、网络协议

1.协议基础

网络协议：是计算机网络中进行数据交换而建立的规则、标准或约定的集合，常用协议有IP、TCP、HTTP、POP3、SMTP等

协议栈：在网络中，为了完成通信，必须使用多层上的多种协议，这些协议按照层次顺序组合在一起，构成了协议栈，也称协议族。

TCP/IP协议族：包括IP协议、IMCP协议、TCP协议，工作标准，是一个开放式协议，Internet网络的标准。构成：



应用层：负责处理特定的应用程序细节

传输层：为两台主机上的应用程序提供端到端的通信

网络层：处理分组在网络中的活动

链路层：包括操作系统中的设备驱动程序和计算机中对应的网络接口卡，一起处理与其他传输媒介的物理接口细节。

1. 三次握手和四次挥手详细原理

三次握手：

第一次client发送一个SYN(J)包给server，然后等待server的ACK回复，进入SYN-SENT状态；

第二次server接受到SYN(seq=J)包后就返回一个ACK(J+1)包以及一个自己的\*\*SYN(K)\*\*包，然后等待client的ACK回复，server进入SYN-RECIVED状态； 第三次client接受到server发回的ACK(K+1)包后，进入ESTABLISHED状态。然后根据server发来的SYN(K)包，返回给等待中的server一个ACK(K+1)包。等待中的server收到ACK回复，也把自己的状态设置为ESTABLISHED。握手完成。

四次挥手：

第一次client发送了一个FIN(M)包，此时client进入FIN-WAIT-1状态，这表明client已经没有数据要发送了；

第二次server收到了client发来的FIN(M)包后，向client发回了一个ACK(M+1)包，此时server进入CLOSE-WAIT状态，client进入FIN-WAIT-2状态； 第三次server向client发送FIN(N)包，请求关闭连接，同时server进入LAST-ACK状态；

第四次client收到server发送的FIN(N)包，进入TIME-WAIT状态。向server发送\*\*ACK(N+1)\*\*包，server收到client的ACK(N+1)包以后，进入CLOSE状态，client等待一段时间还没有得到回复后判断server已正式关闭，进入CLOSE状态。

1. TCP保证可靠支付的方式
   1. 将数据截断为合理的长度
   2. 超时重发
   3. 对于收到的请求，给出确认响应
   4. 校验出包有错，丢弃报文段，不给出响应，TCP发送数据端，超时时会重发数据
   5. 对丢失数据进行重新排序，然后才交给应用层
   6. 对于重复数据进行筛选
   7. TCP还能提供流量控制
2. DNS(Domain Name System,域名系统)

一个记录ip地址的超级分布式数据库，作用是将域名解析成IP地址

解析过程：浏览器获取一个网站后，系统会先从hosts文件中寻找相应的IP地址，没有的话，DNS就会查看本地的DNS缓存，如果是第一次访问，本地没有缓存，就会到DNS服务器查找，还是没有的话则进行递归查找

DNS优化原理：？？？

1. CDN(Content Delivery Network,内容分发网络)

作用：尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定，通过在现有的互联网基础上构造一层智能虚拟网络，实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和相应时间等综合信息将用户的请求重新导向离用户最近的服务节点上，使用户可就近取得所需信息，解决Internet网络拥挤的状况，提高用户访问网站的相应速度。

CDN原理：详见<https://yq.aliyun.com/articles/104041>

1. HTTP
   1. 请求报文组成:A 请求行：请求方法、URL字段和HTTP协议版本

B 请求头部：浏览器类型、内容类型列表、主机名

C 空行：请求头结束

D 请求数据：部分请求方法没有

* 1. 响应报文组成:A 状态行：服务器HTTP协议的版本、状态码、状态信息

B 请求头：和请求报文的请求头类似

C 响应主体：服务器返回的具体数据

请求头含义、请求方法：详见<https://juejin.im/post/5a0ce1d95188253e24708454>

1. HTTP所有状态码的具体含义

详见：<https://blog.csdn.net/u014346301/article/details/53995333>

1. HTTP1.1和HTTP2.0

详见：<https://mp.weixin.qq.com/s/GICbiyJpINrHZ41u_4zT-A?>

1. HTTP加密原理

<https://juejin.im/post/5caab0bff265da24cf311d5b>

1. Websocket

详情：。。。

设计模式

详见：JavaScript设计模式与开发实践

浏览器

1. 操作DOM的API

详见：<https://juejin.im/post/5af43bd5f265da0b8336c6f7>

1. 操作BOM的API

详见：<https://juejin.im/post/5b16113f5188257d7a49ada7>

1. 性能优化

参考：<https://feclub.cn/post/content/dom>

以及react的diff算法

1. DOM事件流

事件流又称事件传播，包括三个阶段：事件捕获阶段，目标阶段，冒泡阶段，首先发生的是事件捕获，然后实际的目标接受到事件，最后是冒泡阶段，可以在这个阶段对事件作出响应

不同浏览器的差异: (1)执行的顺序不一样

(2)事件侦听的函数的区别

(3)事件参数不一样和this指向不一样

事件代理：利用冒泡原理把所点击的元素代理在他的父元素上

详见：<https://www.jianshu.com/p/a77d8928c5c9>

1. 网络请求

方式: (1)link标签的href属性

(2)script标签的src属性

(3)img标签的src属性

(4)ajax发送的请求

(5)表单提交发送请求

(6)a标签的href发送请求

(7)iframe的src属性发送请求

Ajax、fetch、axios：<https://www.jianshu.com/p/8bc48f8fde75>

1. 浏览器的同源策略

同源：如果两个页面的协议，端口和域名都相同，则两个页面具有相同的源。

同源策略限制了从同一个源加载的文档或脚本与来自另一个源的资源进行交互，用于隔离潜在恶意文件。

避免同源策略的方法：CORS、JSONP、document.domain

详见：<https://blog.csdn.net/unique_sirius/article/details/70275269>

<https://juejin.im/post/5c23993de51d457b8c1f4ee1>

1. 浏览器的存储机制

<https://segmentfault.com/a/1190000012578794>

1. 浏览器跨标签通信

<https://segmentfault.com/a/1190000018386033?utm_source=tag-newest>

1. JavaScript引擎

种类：(1)V8，是Google Chrome的一部分，也用于Node.js

(2)Rhino,用于HTMLUnit

* 1. JavaScriptCore,用于webkit型浏览器，08年升级为SquirrelFish
  2. SpiderMonkey，第一款JavaScript引擎，用于Mozilla Firefox
  3. Chakra（JScript引擎），用于IE11
  4. Chakra（JavaScript引擎），用于Microsoft Edge
  5. KJS，用于KDE项目的Konqueror网页浏览器

详细信息还需查询补充

1. 浏览器与服务器的交互过程

<https://blog.csdn.net/hyx709389494/article/details/83446751>

1. 浏览器从输入URL到页面展现的详细过程
2. 输入网址
3. DNS解析
4. 建立tcp连接
5. 客户端发送HTTP请求
6. 服务器处理请求
7. 服务器响应请求
8. 浏览器展示HTML
9. 浏览器发送请求获取其他在HTML中的资源

详见：<https://blog.csdn.net/wlk2064819994/article/details/79756669>

1. 浏览器解析HTML代码的原理，浏览器解析CSS及绘制DOM树

<https://www.cnblogs.com/wjlog/p/5744753.html>

<https://www.jianshu.com/p/ba0429f81dd3>

1. 浏览器运行机制

<https://www.cnblogs.com/caiyy/p/10406934.html>

1. 浏览器的重绘和回流

<https://juejin.im/post/5a9923e9518825558251c96a>

1. 浏览器的垃圾回收机制

标记清除：当变量进入环境时，将变量标记“进入环境”，当变量离开环境时，标记为“离开环境”。某一时刻，垃圾回收器会过滤掉环境中的变量，以及被环境变量引用的变量，剩下的就是被视为准备回收的变量

引用计数：当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另一个值，则这个值的引用次数就减1.当这个引用次数变成0时，则说明没有办法再访问这个值了，因而就可以将其所占内存空间给收回来。

1. 浏览器的缓存方案

参考：<https://juejin.im/post/5a59db5a518825732d7f8ce5>及

<https://www.cnblogs.com/skynet/archive/2012/11/28/2792503.html>

Node

1. 作用

是一个JavaScript运行环境，使JavaScript可以开发后端程序，实现几乎其他后端语言实现的所有功能

1. node基本使用：搭建运行环境，操作文件、数据库，常用API
2. 框架：express、koa

详见相关文档

1. Node的运行原理
2. Node事件驱动，非阻塞机制

<https://www.cnblogs.com/peiyu1988/p/8032982.html>

TypeScript

1. 泛型

详见：<https://www.tslang.cn/docs/handbook/generics.html>

1. 接口：一系列抽象方法的声明，一些方法特征的集合，第三方可以通过调用这组抽象方法，让具体的类执行具体的方法。（interface）
2. Typescript的优势

三大框架的新版本基本都含有typescript，有助于对框架的学习；

支持ES6；

强大的IDE支持：类型检测、语法提示；

等等。

1. typescript基础语法

详见：<https://zhongsp.gitbooks.io/typescript-handbook/content/>

1. 在框架中使用typescript

详见：<https://www.cntofu.com/book/2/doc/handbook/tutorials/React.md>和<https://www.jianshu.com/p/80140d2ade59>

框架

React

1. 架构

MVC->MVP->MVVM

<https://juejin.im/post/59fc625d51882529c0468dc9>

1. react和vue选型

详见<https://juejin.im/post/5cf09ebbf265da1bcb4f1234>

1. setState执行机制

详见：<https://juejin.im/post/5b45c57c51882519790c7441>

1. React的事件机制

React事件使用了事件委托的机制，所有的事件最后都是被委托到了document这个顶级DOM上，以一种先进先出的队列方式进行触发与回调。React拥有自己的合成事件，由对应的EventPlugin负责合成，不同的事件有不同的plugin，且使用EventPluginHub来负责合成事件对象的创建和销毁以提高事件的性能。

详见：<https://juejin.im/post/5bd32493f265da0ae472cc8e>

1. react虚拟DOM及diff算法

虚拟DOM：使用JavaScript对象来表示真实DOM，显示为一个树形结构，只保留了真实DOM节点的一些基本属性和节点之间的层次关系，相当于建立在JavaScript和DOM之间的一层缓存。

Diff算法：react同时维护两棵虚拟DOM树，一颗表示当前的DOM结构，另一棵在react状态变更将要重新渲染时生成，通过比较两棵树的差异，决定是否需要修改DOM结构。

1. React中的Fiber

Fiber就是通过对象记录组件上需要做或者已经完成的更新，一个组件可对应多个Fiber。

React采用自顶向下递归，从根组件或setState()后的组件开始，更新整个子树。当组件树越来越大，递归遍历的成本就越高，持续占用主线程，使主线程上的布局、动画等周期性任务以及交互相应就无法立即得到处理，造成卡顿的视觉效果。Fiber通过拆分要执行的JS，保证不会阻塞主线程。

大致原理：在render函数中创建的ReactElement树在第一次渲染的时候会创建一棵结构一模一样的Fiber节点树，不同的ReactElement类型对应不同的Fiber节点类型。不同的Fiber节点负责对应的ReactElement类型。

详见：[https://blog.csdn.net/qiqingjin/article/details/80118669#0-tsina-1-76044-397232819ff9a47a7b7e80a40613cfe1](https://blog.csdn.net/qiqingjin/article/details/80118669" \l "0-tsina-1-76044-397232819ff9a47a7b7e80a40613cfe1)

<https://juejin.im/post/5ab7b3a2f265da2378403e57>

1. React Router

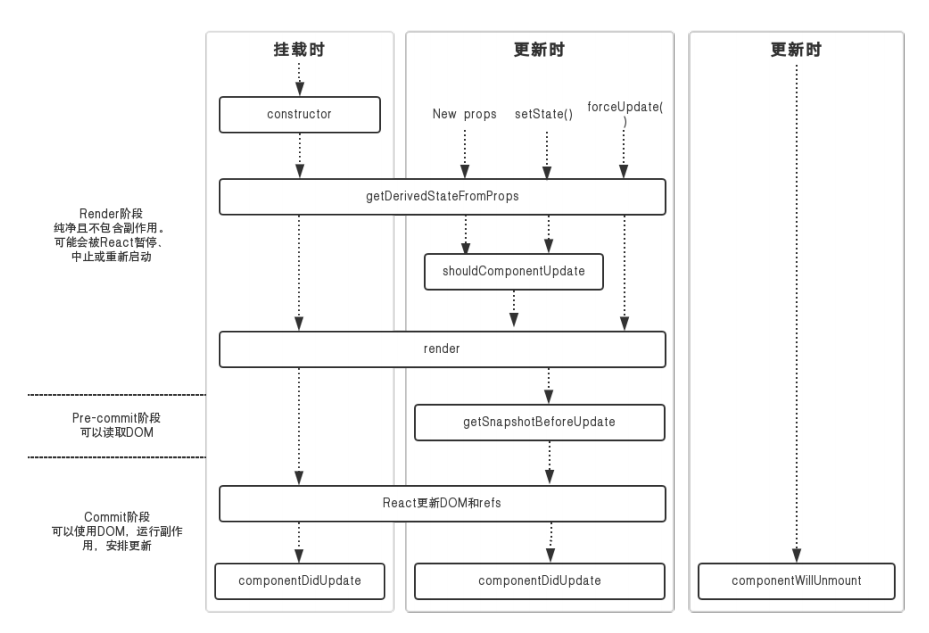
详见：<https://www.jianshu.com/p/53dc287a8020>

1. 熟悉React API

详见react文档，结合项目练习

1. react生命周期

React16以后



详见：<https://juejin.im/post/5ce9004d5188252a4f229425>

1. React HOC：高阶组件，render props，Hooks

详见react文档

1. React框架实现

先熟练运用，过一遍源码，再开始实现

vue

1. 内置指令

v-bind 动态的绑定数据，简写：

v-on 绑定事件监听器，简写@

v-text 更新数据，会覆盖已有结构

v-html 解析数据中的html结构

v-show 根据值的真假，切换元素的display属性

v-if 根据值的真假，切换元素，元素会被销毁，重建

v-else-if 多条件判断

v-else 之前所有条件都不符合渲染

v-for 基于数据集合多次渲染元素或模板块

v-model 在表单控件元素上创建双向数据绑定

v-pre 跳过元素和子元素的编译过程

v-once 只渲染一次，随后数据更新不重新渲染

v-cloak 隐藏未编译的Mustache语法，css中设置 [v-cloak]{display:none}

多端开发

1. SPA

单页应用：动态重写当前的页面来与用户交互，而不需要重新加载整个页面

优点：无刷新体验、前后端分离便于开发、减轻服务器压力、API共享：一套后端程序代码可以用于多种设备、完全的前端组件化。

缺点：不利于SEO优化、容易出错、对前端要求较高。

1. Viewport：用户网页的可视范围

Width：控制viewport的宽度，常用device-width，即设备宽度

Height：viewport的高度

Initial-scale：初始缩放比例，即每一次加载时页面缩放的比例

Maximum-scale：允许用户缩放的最大缩放比例

Minimum-scale：允许用户缩放的最小缩放比例

User-scaleable：是否允许用户缩放

1. 分辨率

单位英寸中所包含的像素点数

Ppi：像素每英寸

Dpi：点每英寸

Dp：设备独立像素

1. 移动端页面适配解决方案，不同机型适配方案

移动端:

1. 媒询：@media
2. Flex弹性布局
3. Rem+viewport缩放
4. Rem
5. 等
6. React native

有空系统学一下

1. 客户端开发技术

Electron等，暂时不作为重点

1. 小程序

微信小程序，有时间在练一练

1. 多端框架

暂时不作为重点

数据流管理

1. 跨组件通信(react)

父子组件通信：父->子：通过props传递信息

子->父:回调函数、利用自定义事件机制

跨级组件通信：层层传递props、context(详见react文档)

额外知识

1. 互联网人员术语

CEO:首席执行官类似总经理、总裁、是企业的法人代表。

CTO:首席技术官，类似总工程师

COO:首席运行官，类似常务总经理

CFO:财务总监

详见：https://zhidao.baidu.com/question/376453324.html

PM:项目主管或项目经理，再或者产品经理

QA:质量保证

UI:用户界面，指对软件的人机交互，操作逻辑，界面美观的整体设计

FE:前端开发  
 DEV:开发软件的代号

DBA:数据库管理员，从事管理和维护数据库管理系统(DBMS)

OPS:开源的基于J2EE平台且是以XML为中心的Web框架

1. 互联网行业术语

B2B:指企业与企业之间通过专用网络或Internet,进行数据信息的交换、传递，开展交易活动的商业模式

B2C:直接面向消费者销售产品和服务的商业零售模式

C2C:电子商务的专业用语，个人与个人之间的电子商务

O2O:将线下的商务机会与互联网结合，让互联网成为线下交易的平台

1. 互联网公司基本运作模式

BOSS：...

PM(产品经理)：产品需求文档(PRD)、时间安排

UE/UX(交互设计)：低保证原型、交互说明

UI(视觉设计)：视觉设计稿、切图标准、视觉规范

RD(程序开发)：前端开发、后端开发

QA(测试)：BUG测试、产品走查、可用性测试、视觉走查

OP(运维)：产品运营、流量统计、用户反馈