[JS部分 4](#_Toc18007795)

[1、变量提升，var和let和const的区别（问过） 4](#_Toc18007796)

[2、call,apply,bind的区别以及各自的实现 4](#_Toc18007797)

[3、typeof操作符与instanceof操作符 5](#_Toc18007798)

[4、关于基本数据类型/Number的存储空间是多大，如果后台发送了一个超过最大字节的数字怎么办 6](#_Toc18007799)

[5、基本类型与引用类型在存储上的差别 7](#_Toc18007800)

[6、引用类型（object，array，function ,string） 8](#_Toc18007801)

[7、检测数组的方法 10](#_Toc18007802)

[8、数组降维 10](#_Toc18007803)

[九、原型和原型链 10](#_Toc18007804)

[九、确定原型与实例之间关系的方法 11](#_Toc18007805)

[十、对象的继承（组合继承和寄生组合继承） 11](#_Toc18007806)

[十一、实现对象的深拷贝 12](#_Toc18007807)

[十二、闭包 13](#_Toc18007808)

[十三、作用域安全的构造函数 15](#_Toc18007809)

[十四、任务队列和Event loop，浏览器中的event loop与node中有什么区别？ 15](#_Toc18007810)

[十五、setTimeOut和setTimeInterval 16](#_Toc18007811)

[十六、节流与防抖 16](#_Toc18007812)

[十七、事件流，事件发生的过程，事件代理的优势 18](#_Toc18007813)

[十八、this对象 19](#_Toc18007814)

[十九、new操作符都做了些什么？ 19](#_Toc18007815)

[十九、箭头函数与普通函数的区别 19](#_Toc18007816)

[二十、location对象 19](#_Toc18007817)

[二十一、history对象 20](#_Toc18007818)

[二十二、为元素添加事件有哪几种方法 20](#_Toc18007819)

[二十三、eval方法 20](#_Toc18007820)

[二十四、Math对象 20](#_Toc18007821)

[二十五、0.1+0.2 === 0.3返回的是真是假，为什么？0.05+0.25会等于0.3么？ 20](#_Toc18007822)

[二十六、js脚本的延迟执行 20](#_Toc18007823)

[二十七、promise对象 21](#_Toc18007824)

[二十八、ES6的一些新特性 22](#_Toc18007825)

[二十八、async函数 23](#_Toc18007826)

[二十九、for in循环与for of循环的区别 23](#_Toc18007827)

[三十、window.onload()与$(document).ready()与DOMContentLoaded的区别 24](#_Toc18007828)

[三十一、封装一个类型判断函数 24](#_Toc18007829)

[三十二、js的垃圾回收机制，GC算法 24](#_Toc18007830)

[三十三、CommonJS，AMD，ES6 模块化 25](#_Toc18007831)

[三十四、前端设计模式 25](#_Toc18007832)

[三十五、遍历对象的几种方法 26](#_Toc18007833)

[三十六、如何实现不可变对象 26](#_Toc18007834)

[HTML与CSS部分 27](#_Toc18007835)

[一、position的几种定位方式，都有什么作用？ 27](#_Toc18007836)

[二、盒子模型 28](#_Toc18007837)

[三、如何决定浏览器使用哪一种盒子模型 28](#_Toc18007838)

[四、CSS布局 28](#_Toc18007839)

[五、导航吸顶的效果怎么实现 29](#_Toc18007840)

[六、css的动画有了解过么？如何实现一个div从左移动到右？ 29](#_Toc18007841)

[七、浮动float 31](#_Toc18007842)

[八、BFC布局 31](#_Toc18007843)

[九、<img>标签的title与alt属性的区别 32](#_Toc18007844)

[十、各种居中方法的实现 32](#_Toc18007845)

[十一、display、visibility、opacity三者的区别 32](#_Toc18007846)

[十二、单行文本溢出与多行文本溢出 33](#_Toc18007847)

[十三、css实现一个三角形 33](#_Toc18007848)

[十四、前端性能优化 34](#_Toc18007849)

[十五、css选择器的优先级 34](#_Toc18007850)

[十六、H5C3的新特性 34](#_Toc18007851)

[十七、link和@import的区别 35](#_Toc18007852)

[十八、vw，vh，vmin，vmax，rem，em的区别是什么？ 35](#_Toc18007853)

[十九、给出子元素margin-top:5px, height:10px，问父元素高度是多少？ 36](#_Toc18007854)

[网络与浏览器部分 36](#_Toc18007855)

[一、发送ajax请求 36](#_Toc18007856)

[二、fetch和ajax的区别 37](#_Toc18007857)

[二、get请求与post请求的异同（百度问过） 37](#_Toc18007858)

[三、DNS解析时会发生什么？ 38](#_Toc18007859)

[3、在浏览器中输入一个url到页面全加载发生了什么？（百度、小米问过） 38](#_Toc18007860)

[四、什么是DOM树，什么是CSSOM树，什么是render树 39](#_Toc18007861)

[CSSOM树的节点是什么（小米问过） 39](#_Toc18007862)

[五、什么是cookie，cookie由哪些信息构成（百度大搜） 39](#_Toc18007863)

[六、cookie，localstorage，sessionstorage，session的异同 39](#_Toc18007864)

[七、前端缓存（http缓存） 40](#_Toc18007865)

[七、既然已经有了Etag，那Last-modified还有存在的必要么？ 40](#_Toc18007866)

[八、浏览器的渲染流程 41](#_Toc18007867)

[八、页面的重排重绘 41](#_Toc18007868)

[九、浏览器内的进程与浏览器内核 43](#_Toc18007869)

[十、图片加载优化 44](#_Toc18007870)

[十一、HTTPs如何保证数据安全传输，或者HTTPS如何进行数字证书验证 44](#_Toc18007871)

[十二、HTTP各版本的区别 46](#_Toc18007872)

[十三、TCP的流量控制 46](#_Toc18007873)

[十四、TCP的拥塞控制 46](#_Toc18007874)

[十五、TCP三次握手 47](#_Toc18007875)

[十六、TCP四次挥手 47](#_Toc18007876)

[十七、web开发中的会话跟踪有哪些 48](#_Toc18007877)

[十八、websoket与ajax的区别 48](#_Toc18007878)

[十九、HTTP的状态码 49](#_Toc18007879)

[二十、301和302的区别 53](#_Toc18007880)

[二十一、CDN的工作原理 53](#_Toc18007881)

[二十二、什么是跨域，为什么会有跨域，解决跨域的几种方法？ 54](#_Toc18007882)

[二十三、csrf攻击 56](#_Toc18007883)

[二十四、XSS攻击 57](#_Toc18007884)

[二十五、content-type都有哪些值？ 57](#_Toc18007885)

[react框架部分与打包工具 57](#_Toc18007886)

[一、react的生命周期 57](#_Toc18007887)

[二、异步setstate，setState到底是异步的还是同步的？ 60](#_Toc18007888)

[三、react和Vue的区别 60](#_Toc18007889)

[四、前端路由react-router的实现原理 61](#_Toc18007890)

[五、webpack打包原理 62](#_Toc18007891)

[六、如何实现组件按需加载 69](#_Toc18007892)

[七、react的ref属性 69](#_Toc18007893)

[八、react高级：React Context 71](#_Toc18007894)

[九、react的diff算法（简单实现） 72](#_Toc18007895)

[十、react组件间通信方法（包括父子组件通信、子组件之间的相互通信） 75](#_Toc18007896)

[十、react的key属性的用法，与真实应用场景结合可以解决什么问题 77](#_Toc18007897)

[综合性问题 77](#_Toc18007898)

[一些细碎的知识点 78](#_Toc18007899)

[一、数字字符串比较大小与数字比较大小有什么不同？ 78](#_Toc18007900)

[二、将字符串转化成数字时，parseInt与Number的区别？ 78](#_Toc18007901)

[三、关于element.style.width 78](#_Toc18007902)

[关于项目 78](#_Toc18007903)

[一、如何实现前后端鉴权？ 78](#_Toc18007904)

[二、页面加载海量数据，比如10w条记录一次性渲染到屏幕上，如何处理不冻结UI？ 80](#_Toc18007905)

[三、前端优化如何实现，或者说你在项目中做了哪些优化 81](#_Toc18007906)

[四、面试被问到项目难点该怎么说？ 81](#_Toc18007907)

# JS部分

1、变量提升，var和let和const的区别（问过）

(1) 当执行 JS 代码时，会生成执行环境，只要代码不是写在函数中的，就是在全局执行环境中，函数中的代码会产生函数执行环境，只此两种执行环境。使用var定义变量需要注意：var操作符定义的变量将成为定义该变量的作用域中的局部变量。也就是说如果在函数中用var定义了一个变量，在函数退出之后这个变量就会被销毁。

看一个面试常问的例子：

b() // 输出：call b

console.log(a) //输出： undefined

var a = 'Hello world'

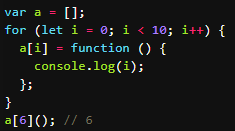
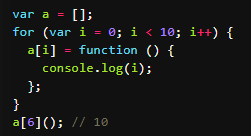
function b() {

console.log('call b')

}

造成这个现象的原因就是变量提升，具体点的解释就是：在生成执行环境时，会有两个阶段。第一个阶段是创建的阶段，JS 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 undefined，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用。

(2) 关于var和let在for循环中使用的区别：



左边：变量i是var命令声明的，在全局范围内都有效，所以全局只有一个变量i。循环结束后，i并没有消失，泄露成了全局变量，会存在于循环外部的执行环境中。

右边：变量i是let声明的，当前的i只在本轮循环有效，所以每一次循环的i其实都是一个新的变量，所以最后输出的是6。

(3) const常量 :不存在变量提升，与let一样只在声明所在的块级作用域有效。一旦声明必须赋值，不能使用null占位。

(4) 三者的区别：一、var声明的变量会挂载在window上，而let和const声明的变量不会，var声明变量存在变量提升，let和const不存在变量提升，let和const声明形成块作用域，同一作用域下let和const不能声明同名变量，而var可以，let存在暂时性死区。

2、call,apply,bind的区别以及各自的实现

call和apply的用途都是在特定的作用域中调用函数，即相当于设置函数体内的this对象。

两者的区别在于接收第二个参数的方式不同，call可以接收一个参数列表，apply只接受一个参数数组。

call方法的实现：

Function.prototype.myCall = function (context) {

var context = context || window

// 给 context 添加一个属性

// getValue.call(a, 'yck', '24') => a.fn = getValue

context.fn = this//这里的this是调用call的那个函数

// 将 context 后面的参数取出来

var args = [...arguments].slice(1)

// getValue.call(a, 'yck', '24') => a.fn('yck', '24')

var result = context.fn(...args)

delete context.fn// 删除 fn

return result

}

apply方法的实现：

Function.prototype.myApply = function (context) {

var context = context || window

context.fn = this

var result;

// 需要判断是否存储第二个参数，如果存在，就将第二个参数展开

if (arguments[1]) {

result = context.fn(...arguments[1])

} else {

result = context.fn()

}

delete context.fn

return result

}

bind方法的实现：

注意：是说当 bind 返回的函数作为构造函数的时候，bind 时指定的 this 值会失效，但传入的参数依然生效

Function.prototype.bind2=function (context) {

if (typeof this!=="function") {//判断调用bind的是不是函数

throw new Error("Function.prototype.bind - what is trying to be bound is not callable");

}

var self =this; //保存this对象

var args =Array.prototype.slice.call(arguments, 1);

var fNOP=function () {};

var fbound=function () {

self.apply(this instanceof self ?this: context,args.concat(Array.prototype.slice.call(arguments)));

}

// 当作为构造函数时，this 指向实例，self 指向绑定函数，因为下面一句 `fbound.prototype = this.prototype;`，已经修改了 fbound.prototype 为 绑定函数的 prototype，此时结果为 true，当结果为 true 的时候，this 指向实例。

// 当作为普通函数时，this 指向 window，self 指向绑定函数，此时结果为 false，当结果为 false 的时候，this 指向绑定的 context。

fNOP.prototype=this.prototype;

fbound.prototype=new fNOP();

return fbound;

}

3、typeof操作符与instanceof操作符

(1) typeof操作符的返回值是一个字符串。typeof对于基本类型，除了null以外都会显示正确的类型。typeof对于对象，除了函数都会显示object。

typeof null //object

typeof 1 // 'number'

typeof '1' // 'string'

typeof undefined // 'undefined'

typeof true // 'boolean'

typeof Symbol() // 'symbol'

typeof b // b 没有声明，但是还会显示 undefined

typeof [] // 'object'

typeof {} // 'object'

typeof console.log // 'function'

要注意：对未初始化的变量执行typeof操作会返回undefined，对于未声明的变量执行typeof操作同样会返回undefined。

(2) instanceof可以正确的判断对象的类型，因为其内部机制是通过判断对象的原型链中是不是能找到类型的prototype。用instanceof检测基本类型时，会始终返回false，因为基本类型不是对象。在检测一个引用类型值和Object构造函数时，instanceof操作符会始终返回true。

instanceof的实现：

function instanceof(left, right) {

let prototype = right.prototype // 获得类型的原型

left = left.\_\_proto\_\_； // 获得对象的原型

while (true) { // 判断对象的类型是否等于类型的原型

if (left === null)

return false

if (prototype === left) // 判断指针是否指向同一个对象

return true

left = left.\_\_proto\_\_

}

}

4、关于基本数据类型/Number的存储空间是多大，如果后台发送了一个超过最大字节的数字怎么办

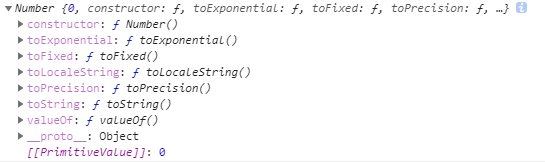
对于基本数据类型来说，虽然不管哪种方式创建出来的都是所属类的一个实例，但是字面量方式创建出来的是基本数据类型，实例方式创建出来的是对象数据类型。

问题：基本数据类型到底有没有原型？？？

当我let a = 777；

console.log(a.\_\_proto\_\_)

输出的是：



（1）null == undefined //true null === undefined //false

null === null //true

NaN不等于任何数，包括NaN自身

（2）其他数据类型转换为boolean的情况：

空字符串“”，数字0，NaN，null，undefined会转化为false

任何非空字符串，非零数字，任何对象（例如Boolean( {} ); 或者Boolean（[ ]））会转化为true

有个经典的选择题： ！[ ] == [ ] 的值是什么？ 答案是true

在进行逻辑非操作符的运算时，先把数据类型转为boolean，然后取反。例如: (!"" = true)

（3）数值转换：

Number()方法：Number(null) 结果为0，Number（undefined）结果为NaN。

Number("") 结果为0

parseInt()方法：该方法会忽略字符串前面的空格，并且会对字符串的每一个字符进行判断，如果第一个字符不是数值或者负号就会返回NaN。当解析字符遇到不合理字符时会返回前面解析的结果。该函数有第二个参数表示转换成几进制的数字。

例如：parseInt("") //NaN parseInt("124bb") //124 parseInt(" 23.3") //23

（4）字符串的特点：字符串是不可变的，意思是一旦创建，字符串的值就不能改变。

例如：var lang = "java"; lang = lang + "script";

这段代码的执行过程就是：先创建一个新字符串，然后将javascript填充进去，最后销毁java和script两个字符串。

（5）toString方法：

数值，布尔值，对象和字符串都有toString方法，null和undefined没有tostring方法，强行调用会报错。

String(null) = “null”，String(undefined) = "undefined"

（6）Object的每个实例都具有的方法：

（7）逻辑操作符

与操作&&是短路操作，在有一个操作数不是布尔值的情况下，与操作不一定会返回布尔值。

如果第一个操作数是对象，则返回第二个操作数；

如果第二个操作数是对象，则只有当第一个操作数的结果为 true时，才会返回该对象；

如果两个操作数都是对象，则返回第二个操作数；

如果有一个操作数是 null，则返回 null；

如果有一个操作数是 NaN，则返回 NaN；

如果有一个操作数是 undefined，则返回 undefined。

（8）使用 new 调用基本包装类型的构造函数，与直接调用同名的转型函数是不一样的。

例如：

var value = "25";

var number = Number(value); //转型函数

alert(typeof number); //"number"

var obj = new Number(value); //构造函数

alert(typeof obj); //"object"

（9）问题：Number()的存储空间是多大，如果后台发送了一个超过最大字节的数字怎们办？

JS中能够表示的最小数值保存在Number.MIN\_VALUE中，能够表示的最大数值保存在Number.MAX\_VALUE中，如果某次计算的结果超过了js的数值范围，那么这个数将会转为特殊的Infinity值，这个值将无法参加下一次的计算。判断一个数是不是Infinity可以用isFinite()函数。

5、基本类型与引用类型在存储上的差别

（1）基本类型储存在栈中，但是一旦被闭包引用则成为常住内存，会储存在内存堆中。引用类型值大小不固定，栈内存中存放地址指向堆内存中的对象。是按引用访问的。如下图所示：栈内存中存放的只是该对象的访问地址，在堆内存中为这个值分配空间。

（2）栈和堆有什么区别？（地平线）

6、引用类型（object，array，function ,string）

（1）Object类型

访问对象属性的两种方式：点表示法和方括号表示法。方括号表示法的优点就是可以通过变量来访问对象的属性。

（2）Array类型

数组的length属性很有特点：可以通过设置这个属性，从数组的末尾移除数据项或者向数组中添加新的数据项。

例如：var colors = ["red", "blue", "green"]; // 创建一个包含 3 个字符串的数组 colors.length = 4;

alert(colors[3]); //undefined

当把一个值放在超出当前数组大小的位置上时，数组就会重新计算其长度值，即长度值等于最后一项的索引加 1。除了数组原有的值和新放置数值，其他位置的数值都会为undefined。

Array类型的方法（调用方式为arr.function(...)）：

■ valueof：返回数组本身

■ toString：返回以逗号分割的、由数组本身的每个值的字符串形式拼接成的字符串

■ join(): 以传入的参数作为分隔符，拼接数组。默认参数是逗号。

※注意：如果数组中的某一项的值是 null 或者 undefined，那么该值在 join()、 toLocaleString()、toString()和 valueOf()方法返回的结果中以空字符串表示。

■ push(): 接收任意数量的参数，把它们逐个添加到数组末尾，并返回修改后数组的长度。

■ pop(): 从数组末尾移除最后一项,修改后数组的长度,然后返回移除的项。

■ shift():移除数组中的第一个项并返回该项，同时将数组长度减1。

■ unshift(): 数组前端添加任意个项并返回新数组的长度。

■ reverse(): 反转数组的顺序，返回一个反转后的数组。

■ sort(): 返回排序后的数组。（会把数组的每一项变成字符串再比较）。

■ concat(): 先创建当前数组一个副本，然后将接收到的参数 添加到这个副本的末尾，最后返回新构建的数组，不会改变当前数组的值。

■ slice(): 返回起始和结束位置之间的项（但不包括结束位置的项）。不会改变当前数组。

■ splice(): 删除—可以删除任意数量的项，只需指定 2 个参数：要删除的第一项的位置和要删除的项数。插入/替换—可以向指定位置插入任意数量的项，且同时删除任意数量的项，只需指3个参数：起始位置、要删除的项数（插入的话这个位置写0）和要插入的任意数量的项。

会改变调用它的数组，返回值为从原数组中删除的项。

■ indexOf():返回要查找的项在数组中的位置，或者在没找到的情况下返回-1。

■ map():对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。

■ forEach()：对数组中的每一项运行给定函数。这个方法没有返回值。

■ filter(): 对数组中的每一项运行给定函数，返回该函数会返回 true 的项组成的数组。

■ reduce(): 实现数字的归并，

reduce函数的实现：

Array.prototype.fakeReduce = function fakeReduce(fn, base) {

if (typeof fn !== "function") {

throw new TypeError("arguments[0] is not a function");

}

let initialArr = this;

let arr = initialArr.concat(); //创建数组的副本

if (base) {arr.unshift(base);} //如果有第二个参数，就把它添加到数组的首部

let index, newValue;

while (arr.length > 1) {

index = initialArr.length - arr.length + 1;

newValue = fn.call(null, arr[0], arr[1], index, initialArr);

arr.splice(0, 2, newValue); // 直接用 splice 实现替换

}

return newValue;

};

牛客网笔试题：["1", "2", "3"].map(parseInt) 的结果是什么？

答案：[1, NaN, NaN]

（3）Function类型

函数名实际上也是一个指针，指向函数对象。不带圆括号的函数名值访问函数指针。

函数的重载：js的函数没有重载，两个重名的函数都是后面的覆盖前面的。想要实现函数的重载：

方法一，可以用arguments.length属性

方法二，举个例子：我们希望people对象拥有一个find方法，当不传任何参数时，就会把people.values里面的所有元素返回来；当传一个参数时,就把first-name跟这个参数匹配的元素返回来；当传两个参数时，则把first-name和last-name都匹配的才返回来。因为find方法是根据参数的个数不同而执行不同的操作的，所以，我们希望有一个addMethod方法，能够如下的为people添加find的重载：

addMethod(people, "find", function() {}); /\*不传参\*/

addMethod(people, "find", function(a) {}); /\*传一个\*/

addMethod(people, "find", function(a, b) {}); /\*传两个\*/

实现方法如下：

function addMethod(object, name, fn) {

　　var old = object[name]; //把前一次添加的方法存在一个临时变量old里面

　　object[name] = function() { // 重写了object[name]的方法

　　　　// 如果调用object[name]方法时，传入的参数个数跟预期的一致，则直接调用

　　　　if(fn.length === arguments.length) {

　　　　　　return fn.apply(this, arguments);

　　　　// 否则，判断old是否是函数，如果是，就调用old

　　　　} elseif(typeof old === "function") {

　　　　　　return old.apply(this, arguments);

　　　　}

　　}

}

函数的声明与函数表达式的区别：

函数声明会有提升，可以后声明先使用

解析器会率先读取函数声明，并使其在执行任何代码之前可用（可以访问）；至于函数表达式，则必须等到解析器执行到它所在的代码行，才会真正被解释执行。

函数的内部属性：arguments和this，arguments.callee指向arguments对象对应的函数。

函数的属性和方法：length，prototype，length表示函数希望接受的命名参数的个数

（4）基本包装类型：Boolean，String，Number

※注意：每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象，该对象在本行语句执行完后会立即销毁。

例如：

var s1 = "some text";

var s2 = s1.substring(2);

当第二行代码访问 s1 时，访问过程处于一种读取模式，也就是要从内存中读取这个字符串的值。而在读取模式中访问字符串时，后台都会自动完成下列处理。(1) 创建 String 类型的一个实例； (2) 在实例上调用指定的方法； (3) 销毁这个实例。

不建议使用构造函数来创建基本数据类型，因为typeof操作符在检测时会返回不同的结果，在布尔表达式中使用时也会产生一些错误。

String类型的方法：

■ charAt():以单字符字符串的形式返回给定位置的那个字符

■ charCodeAt(): 以字符unicode编码的形式返回给定位置的那个字符

■ concat(): 会返回一个新的字符串，不修改原字符串

■ slice(): 返回截取的子字符串，不修改原字符串，第一个参数指定子字 符串的开始位置，第二个参数指定的是子字符串最后一个字符后面的位置

■ subString(): 同上

■ subStr(): 返回截取的子字符串，不修改原字符串，第一个参数指定子字 符串的开始位置，第二个参数指定的是返回的字符个数。

在传递给这些方法的参数是负值的情况下，它们的行为就不尽相同了。其中，slice()方法会将传 入的负值与字符串的长度相加，substr()方法将负的第一个参数加上字符串的长度，而将负的第二个 参数转换为 0。最后，substring()方法会把所有负值参数都转换为 0。

■ indexOf(): 从一个字符串中搜索给定的子字符串，然后返回子字符串的位置，第一个参数为要搜索的子字符串，第二个参数为开始搜索的位置。

■ trim(): 删除前置及后缀的所有空格，会创建一个字符串的副本并返回

■ toLowerCase(),toUpperCase()

■ splice():

7、检测数组的方法

（1）可以使用ES6新增的Array.isArray()方法。

（2）只有一个全局作用域的条件下可以用arr instanceof Array。

（3）如果用 arr instanceof Array 来检测arr是不是数组，如果只有一个全局作用域是没有问题的，但是如果有两个全局作用域，比如说网页包含多个框架，那么就会有多个不同版本的Array构造函数，在将一个框架内的数组传到另一个框架时候，数组就会因为创建自身的构造函数与当前框架的构造函数不是同一个版本而无法用instanceof判断arr是不是数组。

所以，用如下方式判断：  
 Object.prototype.toString.call(arr) == "[object Array]"

8、数组降维

实现一个多维数组的降维：

const flattenDeep = (arr) => Array.isArray(arr)

? arr.reduce( (a, b) => [...a, ...flattenDeep(b)] , [])

: [arr]

九、原型和原型链

**1、什么是原型**

原型对象：每个函数的prototype都会指向一个原型对象。

■ Object.getPrototypeOf(obj): 该方法会得到一个对象obj的原型对象

■ hasOwnProperty(): 由一个实例调用，判断该实例上的属性是在实例中还是在原型中，如果属性在实例中则返回true。

■ in操作符：in 操作符会在通过对象能够访问给定属性时返回 true，无论该属性存在于实例中还是原型中。for in循环返回的是所有能够通过对象访问的、可枚举的属性，像constructor和prototype就是不可枚举的。

■ Object.keys(obj): 接收一个对象作为参数，返回该对象的属性（不包括原型上的属性）组成的字符串数组。

函数都是通过new Function（）产生的，对于实例对象来说，都是通过 new 产生的。

所有函数的默认原型都是Object的实例。

function Foo() {} // function 就是个语法糖，内部等同于 new Function()

let a = { b: 1 } // 这个字面量内部也是使用了 new Object()

（1）首先，创建对象有几种方法：对象字面量，工厂模式和构造函数模式，而js中 的每个函数都会有个prototype属性，这个属性就指向函数的原型，我们就可以在原型上定义一个属性和方法，并且这些属性和方法会被构造函数的每一个实例共享。在通过构造函数创建了一个实例后，实例对象会有一个\_\_proto\_\_属性，该属性就指向构造函数的原型。当我们想访问对象上的属性时，会先从本身开始搜索，如果搜索不到则会去原型上搜索，就比如我们并没有定义数组的sort方法，但是却可以使用array.sort（）。

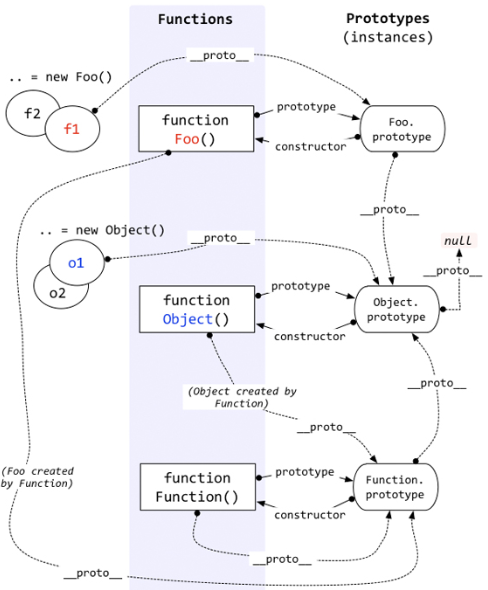
原型对象的缺点：如果原型上的属性friends不是基本数据类型而是引用类型的话，假设在一个实例上对friends进行修改，那么改动则会反应在每一个实例上，所以一般情况下使用自定义类型的时候都是组合使用构造函数模式和原型模式。

从原型扯到原型链，看下文C:\Users\Dudstin\AppData\Local\Temp\SGPicFaceTpBq\7252\0C64B126.png

**2、介绍一下原型链**

原型链是实现继承的主要方法，如果一个对象的原型对象是另外一个实例对象，这样层层递进就形成了原型链。

原型链的经典图片：



首先js引擎创建了 Object.prototype,然后创建了Function.prototype,并且通过 \_\_proto\_\_将两者联系了起来。这里也很好的解释了上面的一个问题，为什么let fun = Function.prototype.bind()没有 prototype属性。因为 Function.prototype是引擎创建出来的对象，引擎认为不需要给这个对象添加 prototype属性。有了Function.prototype以后才有了 function Function()，然后其他的构造函数都是 function Function()生成的。

从原型链扯到继承的几种方法，看第十条**C:\Users\Dudstin\AppData\Local\Temp\SGPicFaceTpBq\7252\0C64B126.png**

九、确定原型与实例之间关系的方法

方法一：使用instanceof操作符

方法二：使用isPrototypeOf（）方法：

例如：instance 为Subtype的一个实例

Subtype.prototype.isPrototypeOf(instance)； //返回true

十、对象的继承（组合继承和寄生组合继承）

（1）原型继承

（2）借用构造函数

（1）组合继承

function SuperType(name){

this.name = name;

this.colors = ["red", "blue", "green"];

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

function SubType(name, age){

//继承属性

SuperType.call(this, name);

this.age = age;

}

//继承方法

SubType.prototype = new SuperType();

SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){

alert(this.age);

};

（2）寄生组合继承

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。其背后的基本思路是：不必为了指定子类型的原型而调用超类型的构造函数，我们所需要的无非就是超类型原型的一个副本而已。

function SuperType(name){

this.name = name;

this.colors = ["red", "blue", "green"];

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

function SubType(name, age){

SuperType.call(this, name); //和组合式继承一样

this.age = age;

}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function(){

alert(this.age);

};

//采用寄生式继承来继承父类的原型

function inheritPrototype(subType, superType){

var prototype = object(superType.prototype); //创建对象副本

prototype.constructor = subType; //增强对象

subType.prototype = prototype; //指定对象

}

十一、实现对象的深拷贝

▲方法一：自己实现深拷贝函数

function deepClone(obj) {

if (Object.prototype.toString.call(obj) === '[object Function]') {

const str = obj.toString();

/^function\s\*\w\*\s\*\(\s\*\)\s\*\{(.\*)/.test(str);

const str1 = RegExp.$1.slice(0, -1);

return new Function(str1);

}

if (!obj || typeof obj !== 'object') return obj;

if (Object.prototype.toString.call(obj) === '[object Date]') return new Date(obj);

if (Object.prototype.toString.call(obj) === '[object RegExp]') return new RegExp(obj);

const cloneObj = Array.isArray(obj) ? [] : {};

for (const i in obj) {

if (obj.hasOwnProperty(i)) {

// 保证只遍历实例属性

cloneObj[i] = typeof obj[i] === 'object' ? deepClone(obj[i]) : obj[i];

}

}

return cloneObj;

}

let a=[1,2,3,4],

b=deepClone(a);

a[0]=2;

console.log(a,b);

▲方法二：我们还可以借助JSON的两个方法来实现深拷贝。

JSON.parse(JSON.stringify(object))

注意，该方法不会拷贝值为undefined的属性和symbol，不能序列化函数，不能解决循环引用的对象。

▲方法三：JQuery实现了一个深拷贝的方法extend。

*$*.extend( [deep ], target, object)

//deep表示是否深拷贝，为true为深拷贝，为false，则为浅拷贝

//target 是一个Object类型 目标对象，其他对象的成员属性将被附加到该对象上。

//object为要复制的对象。

十二、闭包

1、闭包是指有权访问另一个函数作用域中变量的函数。常见的就是在一个函数内部创建另一个函数。注意: 函数的形参也算函数作用域中的变量。

经典面试题，循环中使用闭包解决 var定义函数的问题：

for ( var i=0; i<5; i++) {

setTimeout( function timer() {

console.log( i );

}, 1000 );

}

上述代码的输出结果为5,5,5,5,5。而且是隔一秒中同时打印五个5。

解决的方法一：使用闭包

for (var i = 1; i <= 5; i++) {

(function(j) {

setTimeout(function timer() {

console.log(j);

}, 1000);

})(i);

}

解决方法二：使用let

for ( let i=1; i<=5; i++) {

setTimeout( function timer() {

console.log( i );

}, 1000 );

}

2、闭包的应用场景

（1）js设计模式中单例模式的主要思想就是利用了闭包，单例模式的核心就是保证全局只有一个对象可以访问。可以看下面这个例子：

class Singleton {

constructor() {}

}

Singleton.getInstance = (function() {

let instance

return function() {

if (!instance) {

instance = new Singleton()

}

return instance

}

})( );

let s1 = Singleton.getInstance()

let s2 = Singleton.getInstance()

console.log(s1 === s2) // true

（2）闭包可以将一些不希望暴露在全局的变量封装成“私有变量”。举个例子：

假如有一个计算乘积的函数mult，mult函数接收一些number类型的参数，并返回乘积结果。为了提高函数性能，我们增加缓存机制，将之前计算过的结果缓存起来，下次遇到同样的参数，就可以直接返回结果，而不需要参与运算。这里，存放缓存结果的变量不需要暴露给外界，并且需要在函数运行结束后，仍然保存，所以可以采用闭包。

var mult = (function(){

var cache = {};

var calculate = function() {

var a = 1;

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

a = a \* arguments[i];

}

return a;

}

return function() {

var args = Array.prototype.join.call(arguments, ',');

if(args in cache) {

return cache[args];

}

return cache[args] = calculate.apply(null, arguments);

}

}())

（3）在不使用let的情况下用for循环绑定事件。很经典的闭包问题。

3、闭包的缺点有哪些？

通常来说，函数的活动对象会随着执行期上下文一起销毁，但是，由于闭包引用另外一个函数的活动对象，因此这个活动对象无法被销毁，这意味着，闭包比一般的函数需要更多的内存消耗。在IE中闭包很可能导致内存泄漏。

十三、作用域安全的构造函数

在使用构造函数时，可能会遇到没有使用new操作符，直接调用构造函数的情况，这样会导致构造函数内的this对象映射到windows上。

解决方法：创建一个作用域安全的构造函数，如下：

function SafeFunc(name,job) {  
 if(this instanceof SafeFunc){  
 this.name = name;  
 this.job = job;  
 }  
 else{  
 return new SafeFunc(name.job);  
 }  
}

十四、任务队列和Event loop，浏览器中的event loop与node中有什么区别？

（1）解释一下概念：

"任务队列"是一个事件的队列（也可以理解成消息的队列）。任务队列有很多种。

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

**在js中，任务分成两种，同步执行任务和异步执行任务。**同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

异步执行的运行机制如下：

1、所有同步任务都在主线程上执行，形成一个[**执行栈**](http://www.ruanyifeng.com/blog/2013/11/stack.html)（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。

**（2）js中异步任务分为宏任务和微任务：**

微任务包括 process.nextTick ，promise ，Object.observe ，MutationObserver

宏任务包括 script ， setTimeout ，setInterval ，setImmediate ，I/O ，UI rendering

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask） 和 宏任务（macrotask）。在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task。

一个简单的例子：

console.log('script start');

setTimeout(function() {

console.log('setTimeout');

}, 0);

new Promise((resolve) => {

console.log('Promise')

resolve()

}).then(function() {

console.log('promise1');

})；

console.log('script end');

// 输出结果为：script start => Promise => script end => promise1 => setTimeout

上述例子的执行顺序：

● 第一轮事件循环开始，首先打印script start

●然后执行setTimeout，放入Event Table中，0秒后将回调函数放入宏任务的Event Queue中。

●new Promise 同步代码，立即执行console.log('promise'),然后看到微任务then，因此将其放入微任务的Event Queue中

●接下来执行同步代码console.log(‘script end’)

●主线程的宏任务，已经执行完毕，接下来要执行微任务，因此会执行Promise.then，到此，第一轮事件循环执行完毕

●第二轮事件循环开始，先执行宏任务，即setTimeout的回调函数，打印settimeout，然后查找是否有微任务，没有，时间循环结束

setTimeout(fn,0)的含义是，指定某个任务在主线程最早可得的空闲时间执行，也就是说，尽可能早得执行。它在"任务队列"的尾部添加一个事件，因此要等到同步任务和"任务队列"现有的事件都处理完，才会得到执行。

总结：到此做个总结，事件循环，先执行宏任务，其中同步任务立即执行，异步任务，加载到对应的的Event Queue中(setTimeout等加入宏任务的Event Queue，Promise.then加入微任务的Event Queue)，所有同步宏任务执行完毕后，如果发现微任务的Event Queue中有未执行的任务，会先执行其中的任务，这样算是完成了一次事件循环。接下来查看宏任务的Event Queue中是否有未执行的任务，有的话，就开始第二轮事件循环，依此类推。

（3）node中的event loop

Node 的 Event loop 分为6个阶段，它们会按照顺序反复运行

┌───────────────────────┐

┌─>│ timers │

│ └──────────┬────────────┘

│ ┌──────────┴────────────┐

│ │ I/O callbacks │

│ └──────────┬────────────┘

│ ┌──────────┴────────────┐

│ │ idle, prepare │

│ └──────────┬────────────┘ ┌───────────────┐

│ ┌──────────┴────────────┐ │ incoming: │

│ │ poll │<──connections─── │

│ └──────────┬────────────┘ │ data, etc. │

│ ┌──────────┴────────────┐ └───────────────┘

│ │ check │

│ └──────────┬────────────┘

│ ┌──────────┴────────────┐

└──┤ close callbacks │

└───────────────────────┘

十五、setTimeOut和setTimeInterval

setTimeout中所执行函数中的this，永远指向window！！注意是要**延迟执行的函数中的this**哦！！

setTimeOut的原理：setTimeOut的第一个参数是要实行的代码，第二个参数是一个毫秒数，但经过该时间后指定的代码不一定会执行。 JavaScript 是一个单线程序的解释器，因此一定时间内只能执行一段代码。为了控制要执行的代码，就有一个JavaScript任务队列。这些任务会按照将它们添加到队列的顺序执行。setTimeout()的第二个参数告诉 JavaScript 再过多长时间把当前任务添加到队列中。如果队列是空的，那么添加的代码会立即执行；如果队列不是空的，那么它就要等前面的代码执行完了以后再执行。

调用 setTimeout()之后，该方法会返回一个数值ID，表示超时调用。这个超时调用ID是计划执行代码的唯一标识符，可以通过它来取消超时调用。

十六、节流与防抖

（1）函数防抖：让一个函数在一定间隔内没有被调用时，才开始执行被调用方法。

函数防抖的简单实现：

const decounce = function(fn, delay) {

let timer = null;

return function() {

const context = this

let args = arguments

clearTimeout(timer) // 每次调用debounce函数都会将前一次的timer清空，确保只执行一次

timer = setTimeout(() => {

fn.apply(context, args)

}, delay)

}

}

函数防抖带有立即执行项的实现：

function now() { // 这个是用来获取当前时间戳的

return +new Date()

}

/\*\*

\* 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行

\*

\* @param {function} func 回调函数

\* @param {number} wait 表示时间窗口的间隔

\* @param {boolean} immediate 设置为ture时，是否立即调用函数

\* @return {function} 返回客户调用函数

\*/

function debounce (func, wait = 50, immediate = true) {

let timer, context, args;

// 延迟执行函数

const later = () => setTimeout(() => {

// 延迟函数执行完毕，清空缓存的定时器序号

timer = null

// 延迟执行的情况下，函数会在延迟函数中执行

// 使用到之前缓存的参数和上下文

if (!immediate) {

func.apply(context, args)

context = args = null

}

}, wait)

// 这里返回的函数是每次实际调用的函数

return function(...params) {

// 如果没有创建延迟执行函数（later），就创建一个

if (!timer) {

timer = later()

// 如果是立即执行，调用函数

// 否则缓存参数和调用上下文

if (immediate) {

func.apply(this, params)

} else {

context = this

args = params

}

// 如果已有延迟执行函数（later），调用的时候清除原来的并重新设定一个

// 这样做延迟函数会重新计时

} else {

clearTimeout(timer)

timer = later()

}

}

}

（2）函数节流：让一个函数无法在很短的时间间隔内连续调用，当上一次函数执行后过了规定的时间间隔，才能进行下一次该函数的调用。

函数节流的简单实现：

// 方法一：定时器实现

const throttle = function(fn,delay) {

let timer = null;

return function() {

const context = this;

let args = arguments;

if(!timer) {

timer = setTimeout(() => {

fn.apply(context,args);

clearTimeout(timer);

},delay)

}

}

}

// 方法二：时间戳

const throttle2 = function(fn, delay) {

let preTime = Date.now()

return function() {

const context = this

let args = arguments

let doTime = Date.now()

if (doTime - preTime >= delay) {

fn.apply(context, args)

preTime = Date.now()

}

}

}

十七、事件流，事件发生的过程，事件代理的优势

事件流描述的是从页面中接收事件的顺序。事件流分事件冒泡和事件捕获。

1、事件冒泡：事件由具体的节点接收，然后逐级向上传播到较为不具体的节点。

2、事件捕获：事件最先由不具体的节点接收，然后向下传递，最后传递到具体的节点。

3、事件代理：

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上。例如：

<ul id="ul">

<li>1</li>

<li>2</li>

<li>3</li>

<li>4</li>

<li>5</li>

</ul>

<script>

let ul = document.querySelector('#ul')

ul.addEventListener('click', (event) => {

console.log(event.target);

})

</script>

li元素是动态生成的，这时候采用事件代理将事件处理函数绑定在父元素ul上。

事件代理的方式相对于直接给目标注册事件来说，有以下优点：节省内存，不需要给子节点注销事件

十八、this对象

this对象的四种用法：

1、在一般函数方法中使用 this 指代全局对象

2、作为对象方法调用，this 指代上级对象

3、作为构造函数调用，this 指代new 出的对象，即构造函数的实例

4、可以通过call，apply，bind方法等改变this的指向

十九、new操作符都做了些什么？

function New(){

var obj={};//第一步 生成一个空对象

obj.\_\_proto\_\_=Cat.prototype;//第二步 将新对象的原型赋值为构造函数的原型

var res=Cat.apply(obj,arguments);//第三步 在新对象的执行环境调用构造函数，如果有参数会传进去

return typeof res==='object'?res:obj;//第四步 如果构造函数有返回值且返回值为对象则该对象作为new出来的新对象，如果构造函数没有返回值或返回的不是对象则将obj作为new出来的新对象

}

十九、箭头函数与普通函数的区别

1、箭头函数不可以当做构造函数，也就是说，不可以使用new命令，否则会报错。因为箭头函数没有原型。

2、箭头函数内部不可以使用arguments对象，该对象在函数体内部是不存在的，如果箭头函数不确定参数的个数，想实现和arguments一样的功能的话可以用rest对象。

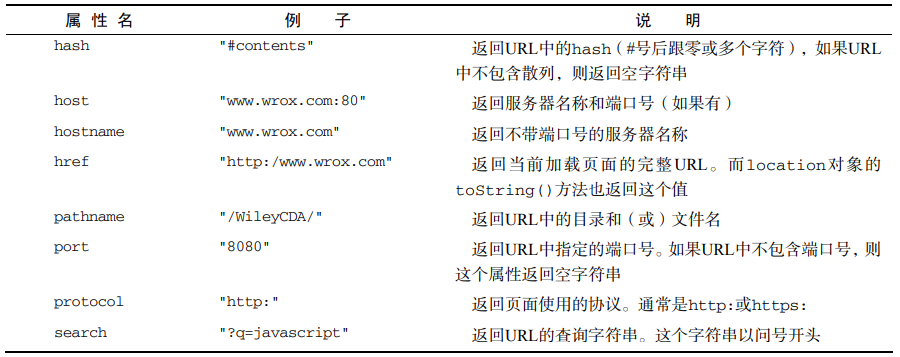
3、箭头函数内部的this对象就是定义是所在的对象，不是使用时所在的对象。出现这种情况的原因是箭头函数根本没有自己的this。

4、call，apply，bind函数不能改变箭头函数的this指向。

二十、location对象

location是一个很有用的BOM对象，

它的一些常用属性如下：



浏览器的位置操作：

assign()方法：可以立即打开新 URL 并在浏览器的历史记录中生成一条记录。

location.href：为这个属性赋值也可以达到和assign方法一样的效果。

每次修改 location 的上述属性（hash 除外），页面都会以新 URL 重新加载。浏览器的历史记录中也会生成一条新记录。

location.replace():只接受一个参数，即要导航到的 URL,结果虽然会导致浏览器位置改变，但不会在历史记录中生成新记录。在调用 replace()方法之后，用户不能回到前一个页面.

二十一、history对象

History 对象是 window 对象的一部分，可通过 window.history 属性对其进行访问，其包含着用户访问过的url。history对象的属性和方法：back()，forward()，go()；

H5新增的方法有：

1、pushState()：该方法可以接受三个参数，状态对象，新状态的标题，新的url地址（必须与当前网址处在同一个域中）。使用pushState会创建新的历史状态，也就是说会将新的url加入历史状态栈中，但是并不会想浏览器发请求，请求新的url。

2、replaceState(): 该方法也接受三个参数，同上，该方法是用于修改浏览历史中的当前记录。

例如：假定当前网页是example.com/example.html。

history.pushState({page: 1}, 'title 1', '?page=1');

history.pushState({page: 2}, 'title 2', '?page=2');

history.replaceState({page: 3}, 'title 3', '?page=3');

history.back()

// url显示为http://example.com/example.html?page=1

history.back()

// url显示为http://example.com/example.html

history.go(2)

// url显示为http://example.com/example.html?page=3

3、popstate事件：每当处于激活状态的历史记录条目发生变化时,popstate 事件就会在对应window对象上触发。 需要注意的是，调用 history.pushState() 或者 history.replaceState() 不会触发 popstate 事件。 opstate事件只会在浏览器某些行为下触发， 比如点击后退、前进按钮(或者在JavaScript中调用history.back()、history.forward()、history.go()方法)。

二十二、为元素添加事件有哪几种方法

addEventListener，IE7之前的浏览器不支持这个方法，所以添加事件要用**attachEvent**

二十三、eval方法

该方法只接受一个参数，即要执行的js字符串。js解释器会将eval中的字符串当成代码来解析并将结果插入到原位置。

eval中创建的任何变量或函数都不会被提升。

二十四、Math对象

Math.floor（）：向下舍入。

Math.ceil（）：向上舍入

Math.abs（）：取绝对值。

Math.pow（num，power）：返回num的power次幂。

Math.trunc（）： 该方法会将数字的小数部分去掉，只保留整数部分。例如Math.trunc(3/-2)，输出值为-1

产生一个范围内随机数的方法：Math.floor(Math.random()\*可能值的综述+第一个开始的值)。

二十五、0.1+0.2 === 0.3返回的是真是假，为什么？0.05+0.25会等于0.3么？

书上的P28有写到，0.1+0.2的值根本不是0.3，而是0.30000000000000004，造成这个现象的原因是因为IEEE754这种数值计算方法会将数值保存为二进制然后进行计算，由于浮点数用二进制表达时是无穷的，所以在进行算术计算时会产生**舍入误差**，但是0.05+0.25判断结果为真。

二十六、js脚本的延迟执行

<script>标签有**defer**属性，设置了该属性的脚本会延迟到整个页面都解析完毕后在运行，也就是告诉浏览器立即下载但不执行。

<script>标签还有**async**属性，与defer的作用相同但是标记为async的脚本不能保证按照它们指定的先后顺序执行。

如果两个属性都存在，那么async属性的优先级更高。



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

二十七、promise对象

**1、关于then方法返回的新promise对象的状态：**

新promise的状态取决于then方法的返回值，如果then()方法中返回了一个参数值，那么返回的Promise将会变成fulfilled状态。如果then()方法返回了一个Promise新实例，那么then方法返回的promise对象的状态就与这个新实例保持一致，如果then()方法没有明确指定的resolve(data)/reject(data)/return data时（即没有返回值），那么返回的新Promise就是fulfilled状态，可以一层一层地往下传递。

**2、promise对象的方法：**

then，catch，all，race，resolve，reject，finally。

**3、实现一个promise对象**

class myPromise{

constructor(fn){ // 在对象初始化时会传入一个函数

this.state = 'PENDING';

this.value;

this.chain = []; // 用于存储要调用的函数

const resolve = (result) => {

this.state = 'FULFILLED';

this.value = result;

for(let {handleResolve} of this.chain){

handleResolve(result)

}

}

const reject = (error) => {

this.state = 'REJECTED';

this.value = error;

for(let {handleReject} of this.chain){

handleReject(error);

}

}

try{

fn(resolve,reject);

}catch(err){

reject(err);

}

}

// then(handleResolve,handleReject){

// if(this.state == 'FULFILLED'){

// handleResolve(this.value)

// }else if(this.state == 'REJECTED'){

// handleReject(this.value)

// }else{

// this.chain.push({handleResolve,handleReject})

// }

// }

// 实现链式调用

then(handleResolve,handleReject){

return new myPromise((resolve,reject) => { // 由于这里使用的是箭头函数，所以this指向的是调用then方法的promise而不是new的promise

const \_onFulfilled = res => {

try {

//注意这里resolve有可能要处理的是一个promise

resolve(handleResolve(res));

} catch (err) {

reject(err);

}

};

const \_onRejected = err => {

try {

reject(handleReject(err));

} catch (\_err) {

reject(\_err);

}

};

if (this.state === 'FULFILLED') {

\_onFulfilled(this.value);

} else if (this.state === 'REJECTED') {

\_onRejected(this.value);

} else {

this.chain.push({ handleResolve: \_onFulfilled, handleReject: \_onRejected });

}

})

}

}

**4、实现一个promise.all方法**

Promise.all = (list) => {

    return new Promise((resolve,reject) => {

        const results = [];

        let count = 0;

        list.forEach((item, index) => {

            item.then((res) => {

                results[index] = res;

                if (++count === list.length) {

                    resolve(results);

                }

            }, (rej) => reject(rej));

        });

    });

}

二十八、ES6的一些新特性

**1、变量的声明方面**

1、用let声明变量

**2、字符串方面**

2、模板字符串

模板字符串的优势：在模板字符串的${}里可以放入表达式

**3、对象的方面**

**（1）**新增的对象的方法：

Object.is(val1,val2)   
//用于比较两个值是否值相等，类似于===，但是不同的是Object.is(NaN,NaN)为true，Object.is(+0，-0)为false  
Object.assign(target,source1,source2...)  
//将源对象的所有属性复制到目标对象上，同名属性会被覆盖  
//得到的target对象是源对象的一个浅拷贝，不会复制源对象继承的属性和不可枚举的属性，

（2）变量的解构赋值 ，可以利用解构赋值进行浅拷贝// 是使用了iterator

（3）对象字面量的简写

（4）for of循环 // 是使用了iterator

（5）展开运算符

* （6）Reflect对象，这个对象是为操作对象而提供的API，将Object对象的一些属于语言内部的方法放到Reflect对象上，从Reflect上能拿到语言内部的方法。如：Object.defineProperty
* 修改某些object方法返回的结果。如：Object.defineProperty(obj, name, desc)在无法定义属性的时候会报错，而Reflect.defineProperty(obj, name, desc)则会返回false
* 让Object的操作都变成函数行为。如object的命令式：name in obj和delete obj[name] 则与 Reflect.has(obj, name)、Reflect.deleteProperty(obj, name)相等

**4、函数方面**

7、箭头函数

8、iterator函数

**5、新的数据结构：set和map**

set与Array.from结合可以实现数组去重，set比较两个元素是否相等使用的是类似于“===”的算法，不同点在于NaN的比较。keys，values，entries方法均返回遍历器对象。

map：键值对的集合但是键可以是对象，但是只有同一个对象的引用才会被视为同一个键。**可以解决同名属性碰撞的问题**

扩展运算符、解构赋值、和Array.from( )方法内部调用的都是遍历器接口。

二十八、async函数

**1、async函数简介**

async函数会返回一个promise对象，async函数return语句返回的值会成为then的回调函数的参数。在函数执行的时候，如果遇到await则会先返回，等到await后面的异步操作完成后在接着执行函数体内后面的内容。

await命令后面是一个promise对象，如果不是promise其他数据类型，比如数字或字符串等等，会被转化为一个状态为fulfilled的promise对象。如果await后面的异步操作发生了错误，promise对象变成了reject状态且没有捕获该错误，那么async函数就会中断，await后面的代码不会被执行了。避免这样的情况出现，可以将await用try catch包裹，或者为await后面的promise对象添加一个catch方法。

简单的来说，如果await等到的不是一个 Promise 对象，那 await 表达式的运算结果就是它等到的东西。

如果它等到的是一个 Promise 对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果。

**2、acync await如何并行？**

第一种方法：

const foo = async () => {

const foo1 = new Promise((resolve) => setTimeout(resolve, 1000)).then(() => { console.log(1); });

const foo2 = new Promise((resolve) => setTimeout(resolve, 1000)).then(() => { console.log(2); });

await foo1;

await foo2;

**}**

第二种方法：使用promise.all

二十九、for in循环与for of循环的区别

for of循环的使用范围可以包括数组、set、map、结构，某些类数组对象（比如arguments对象和nodelist对象）、generator对象，以及字符串。

for in循环会以任意顺序遍历一个对象或者数组的可枚举属性，因此会遍历对象原型链上的属性。

for in循环遍历的是数组的键名，for of遍历的是键值。

三十、window.onload()与$(document).ready()与DOMContentLoaded的区别

● window.onload（）：必须在页面所有元素（包括图片，引用文件）加载完后执行。。

● $(document).ready() ：是页面中所有HTML DOM，CSS DOM结构加载完之后就会执行，其他资源可能没有加载完。

● DOMContentLoaded 事件触发代表初始的 HTML 被完全加载和解析，不需要等待 CSS，JS，图片加载。

一、DOM文档加载的步骤为：

1. 解析HTML结构。
2. 加载外部脚本和样式表文件。
3. 解析并执行脚本代码。
4. DOM树构建完成。//DOMContentLoaded
5. 加载图片等外部文件。
6. 页面加载完毕。//load

二、区别

1. 执行时间不同

* window.onload必须等到页面内包括图片的所有元素加载完毕后才能执行。
* $(document).ready()是DOM结构绘制完毕后就执行，不必等到加载完毕。
* 在第4步，会触发DOMContentLoaded事件。在第6步，触发load事件。  
  //原生js写法  
  document.addEventListener("DOMContentLoaded", function() {  
  // ...代码...  
  }, false);
* window.addEventListener("load", function() {
* // ...代码...
* }, false);

1. 编写个数不同

* window.onload不能同时编写多个，如果有多个window.onload方法，只会执行一个
* $(document).ready()可以同时编写多个，并且都可以得到执行

三、简化写法

* window.onload没有简化写法
* $(document).ready(function(){})可以简写成$(function(){});

三十一、封装一个类型判断函数

function getType(target) {

//先处理最特殊的Null

if(target === null) {

return 'null';

}

//判断是不是基础类型

const typeOfT = typeof target；

if(typeOfT !== 'object') {

return typeOfT;

}

//肯定是引用类型了

consttemplate= {

"[object Object]": "object",

"[object Array]" : "array",

"[object Function]": "function",

// 一些包装类型

"[object String]": "object - string",

"[object Number]": "object - number",

"[object Boolean]": "object - boolean"

};

const typeStr = Object.prototype.toString.call(target);

return template[typeStr];

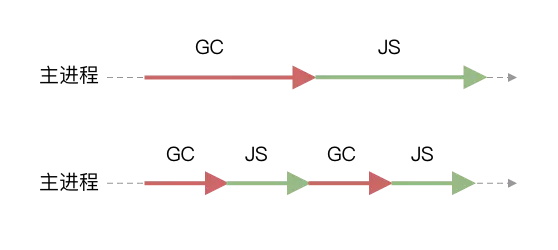
}

三十二、js的垃圾回收机制，GC算法

在讲垃圾回收之前先来说说**“全暂停”**，

垃圾回收一般发生时，代码很有可能正在执行一些逻辑，为避免应用逻辑与垃圾回收器看到的情况不一致，垃圾回收算法在执行时，需要停止应用逻辑。垃圾回收算法在执行前，需要将应用逻辑暂停，执行完垃圾回收后再执行应用逻辑，这种行为称为 「全停顿」（Stop The World）。例如，如果一次GC需要50ms，应用逻辑就会暂停50ms。

但是这种全暂停算法在新生代中使用的话，由于存活对象少，垃圾回收效率高，全停顿时间短，造成的影响小，但是老生代中存活对象多，垃圾回收时间长，全停顿造成的影响大，往往会导致糟糕的用户体验，所以2011年起，V8就把全暂停替换为**增量标记算法（incremental marking）**，将一次停顿进行的标记过程，分成了很多小步。每执行完一小步就让应用逻辑执行一会儿，这样交替多次后完成标记。



V8 实现了准确式 GC，GC 算法采用了分代式垃圾回收机制。因此，V8 将内存（堆）分为新生代和老生代两部分。

**1、新生代算法**

新生代中的对象一般存活时间较短，使用 Scavenge GC 算法。

在新生代空间中，内存空间分为两部分，分别为 From 空间和 To 空间。在这两个空间中，必定有一个空间是使用的，另一个空间是空闲的。新分配的对象会被放入 From 空间中，当 From 空间被占满时，新生代 GC 就会启动了。算法会检查 From 空间中存活的对象并复制到 To 空间中，如果有失活的对象就会销毁。当复制完成后将 From 空间和 To 空间互换，这样 GC 就结束了。如果一个对象被断定为，未被引用，就清除；有被引用，逃逸次数+1，如果此时逃逸次数为2，即对象经历过一次scavenge算法，就移入老生代，否则移入to-space。

**2、老生代算法**

老生代中的对象一般存活时间较长且数量也多，使用了两个算法，分别是标记清除算法（mark-sweep）和标记压缩算法（mark-compact）。

在老生代中，以下情况会先启动标记清除算法：

◆当内存无法为对象分配空间时，

◆空间中被对象超过一定限制

◆空间不能保证新生代中的对象移动到老生代中

标记，采用3色标记：黑、白、灰。步骤如下：

（1）GC开始，所以对象标记为白色。

（2）根对象标记为黑色，并开始遍历其子节点（引用的对象）。

（3）当前被遍历的节点，标记为灰色，被放入一个叫 marking bitmap 的栈。在栈中，把当前被遍历的节点，标记为黑色，并出栈，同时，把它的子节点（如果有的话）标记为灰色，并压入栈。（大对象比较特殊，这里不展开）

（4）当所有对象被遍历完后，就只剩下黑和白。通过压缩或清除的方式，清理掉白色，完成GC。

使用标记压缩算法是因为清除对象后会造成堆内存出现碎片的情况，当需要分配一个比较大的对象时，没有任何一个碎片内支持分配，所以需要进行标记压缩。当碎片超过一定限制后会启动压缩算法。在压缩过程中，将活的对象像一端移动，直到所有对象都移动完成然后清理掉不需要的内存。

三十三、CommonJS，AMD，ES6 模块化

1. **CommonJs模块化**

CommonJS定义的模块分为:{模块引用(require)} {模块定义(exports)} {模块标识(module)}。require()用来引入外部模块；exports对象用于导出当前模块的方法或变量，唯一的导出口；module对象就代表模块本身，exports就是module上的一个属性，即 exports = module.exports，所以我们不能对exports进行复制。

导出函数或者变量的两种方法：

module.exports = { //写上需要向外暴露的函数、变量，只是用exports时不可用此方法

add: add,

}

或者 module.exports.add = add;

或者 exports.add = add;

2、

三十四、前端设计模式

1、发布订阅模式（也叫观察者模式）

class MyEvent{

constructor(){

this.events = {};

}

// 订阅

on(eventName,callback){

if(this.events[eventName]){

this.events[eventName].push(callback)

}else{

this.events[eventName] = [callback];

}

}

// 发布

emit(eventName){

if(this.events[eventName]){

for(let item of this.events[eventName]){

item(arguments[1]);

}

}

}

}

const event = new MyEvent();

event.on('event-data',function(data){ // 订阅一个event\_data事件

console.log('event-data:' + data)

})

event.on('dance',function(data){

console.log('dance:'+data)

})

event.on('dance',function(data){

console.log('can Sing:'+data)

})

event.emit('event-data','liu');

event.emit('dance','grimmjoy')

2、单例模式

具体实现看闭包那一节

三十五、遍历对象的几种方法

1、for in循环：会遍历对象自身属性和继承的可枚举属性

2、Object.keys（obj）：会返回一个数组，数组中的值是对象自身的可枚举属性

3、Object.getOwnPropertyNames（obj）：会返回一个数组，数组中的值是对象自身的所有属性，包括可枚举的和不可枚举的，但是不包含Symbol

4、Object.getOwnPropertySymbols（obj）：会返回一个数组，数字中的值是对象所有的Symbol属性

5、Reflect.ownKeys（obj）：会返回一个数组，数组中的值是对象自身的所有属性（包括不可枚举的和symbol）

三十六、如何实现不可变对象

**1、不可扩展对象**

正常我们是可以通过object.xxx来为对象添加属性和方法的，如果想禁止这类行为的话可以通过Object.preventExtensions()方法，它可以使一个对象无法再增加新的属性

var obj = Object.preventExtensions({name: 'the shy'});

// 直接定义新的属性会报错

Object.defineProperty(obj, 'content', {

value: 'hello'

}); // TypeError: Cannot define property:p, object is not extensible.

// 非严格模式下通过点符号添加不会报错，但会静默失败，原对象仍然没有 content 属性

obj.content = 'hello';

obj.content; // undefined

**2、密封对象**

将一个对象变为不可扩展后，虽然不可以增加新的属性和方法，但是仍然可以修改以及删除对象上原有的属性，想禁止这类行为的话可以通过Object.seal()方法，它在实现对象不可扩展的基础上，将现有属性的configurable 特性设为 false。但是密封之后的对象还是可以改变现有属性上的值

Object.seal(obj);

Object.getOwnPropertyDescriptor(obj, 'content'); // seal 后 configurable 变为 false

// Object {

// value: "hello",

// writable: true,

// enumerable: true,

// configurable: false

// }

**3、冻结对象**

由于密封对象仍可以改变属性上的值，如果想禁止该行为的话，可以通过Object.freeze()方法，它可以使对象一个对象不能再添加新属性，也不可以删除旧属性，且不能修改属性的值。它将属性的writable 值变为了 false。

# HTML与CSS部分

一、position的几种定位方式，都有什么作用？

**1、position都有哪些取值？**

**绝对定位absolute**：使元素脱离文档流布局，元素原来在文档流中的空间由后面的元素填充。

特点：（1）如果这个元素没有父元素，那么将来top、left、right、bottom是相对于浏览器窗口来定位的

（2）如果绝对定位的元素有父元素，但父元素没有相对定位，那么这个时候top、left、right、bottom还是相对于浏览窗口来定位的

（3）如果绝对定位的元素有父元素，而且父元素有定位（非static），那么这个绝对定位的元素偏移是以自己的父元素为基础

（4）绝对定位之后的元素在页面上不会占据位置

**相对定位relative**：元素不会脱离文档流布局，只会改变自身的位置，原先在文档流中的位置留白，

**固定定位fixed**： 会使元素脱离标准文档流。相对于浏览器窗口定位。

**静态定位static**： 静态定位是默认的定位模式，即网页中所有的元素都默认是静态定位，在静态定位下无法通过偏移属性（top/left/right/bottom）来移动元素。静态定位唯一的作用就是：取消定位。

**新增的css3属性sticky：**这个属性是position:relative和position:fixed的结合体——当元素在屏幕内，表现为relative，就要滚出显示器屏幕的时候，表现为fixed。**sticky元素效果完全受制于父级元素们**，这和position:fixed定位有着根本性的不同，fixed元素直抵页面根元素，其他父元素对其left/top定位无法限制。

注意，在使用这个属性的时候有几个要强调的地方：（1）设置了position：sticky属性的元素一定要设置top等属性，不然不会生效；（2）sticky元素的外层元素（可以是父元素或者父元素的父元素等等以此类推）如果设置了overflow：visible以外的任何值，都会导致sticky失效；（3）同一个父容器中的sticky元素，如果定位值相等，则会重叠

**2、position, display, float一起设置，会怎么样？**

二、盒子模型

一个盒子中主要的属性就5个：width、height、padding、border、margin。在 CSS的标准盒子模型中，width 和 height 指的是内容区域的宽度和高度。增加内边距、边框和外边距不会影响内容区域的尺寸，但是会增加元素框的总尺寸。IE盒子模型中，width 和 height 指的是content+border+padding的宽度和高度。

三、如何决定浏览器使用哪一种盒子模型

css3中的box-sizing属性可以指定盒子模型种类，box-sizing：content-box指定盒子模型为W3C，box-sizing：border-box为IE盒子模型。

四、CSS布局

**1、三栏自适应布局**

（1）对左右块使用分别使用float:left和float:right，float使左右两个元素脱离文档流，中间元素在正常文档流中。  
**注意：**middle位置要求middle块一定要放在left和right之后，因为它占据文档流位置，所以一定要放在最后。

（2）这种方法就是利用了绝对定位absolute，让左右两块脱离文档流，此时left和right与文档流无关，不占据空间。定位的元素和其他元素重叠。此处的left是叠在middle上的。

（3）**圣杯布局**：使用BFC+float可以实现圣杯布局，也就是三栏布局：将middle的宽度设置为100%，然后将其float设置为left，其中的main块设置margin属性，然后左边栏设置float为left，之后设置margin为-100%，右栏也设置为float：left，之后margin-left为自身大小。

.wrapper{

overflow: hidden; //清除浮动

}

.middle{

width: 100%;

float: left;

}

.middle .main{

margin: 0 220px;

background: red;

}

.left{

width: 200px;

height: 300px;

float: left;

background: green;

margin-left: -100%;

}

.right{

width: 200px;

height: 300px;

float: left;

background: yellow;

margin-left: -200px;

}

（4）flex布局，这个布局的唯一缺点是可能会有兼容性的问题。

写一个<div>，在flex中类似于容器，并设div的display：flex。将left、right、middle全部包进去，middle块的样式中加上flex：1（或者auto）

五、导航吸顶的效果怎么实现

<script type="text/javascript">

var tit = document.getElementById("nav");//nav是导航条元素

//占位符的位置

var rect = tit.getBoundingClientRect();//获得页面中导航条相对于浏览器视窗的位置

var inser = document.createElement("div");

tit.parentNode.replaceChild(inser,tit);

inser.appendChild(tit);

inser.style.height = rect.height + "px";

//获取距离页面顶端的距离

var titleTop = tit.offsetTop;

//滚动事件

document.onscroll = function(){

//获取当前滚动的距离

var btop = document.body.scrollTop||document.documentElement.scrollTop;

//如果滚动距离大于导航条据顶部的距离

if(btop>titleTop){

//为导航条设置fix

tit.className = "clearfix fix";

}else{

//移除fixed

tit.className = "clearfix";

}

}

</script>

六、css的动画有了解过么？如何实现一个div从左移动到右？

**1、css3的2D转换**

（1）translate()方法：通过 translate() 方法，元素从其当前位置移动，根据给定的 left（x 坐标） 和 top（y 坐标） 位置参数，单个方向上的移动可以用translateX()和translateY()。

例如：transform：translate（50px,100px）

（2）rotate()方法：通过 rotate() 方法，元素顺时针旋转给定的角度。允许负值，元素将逆时针旋转。注意单位是deg，表示旋转多少角度

例如：transform：rotate（30deg）

（3）scale()方法：通过 scale() 方法，元素的尺寸会增加或减少，根据给定的宽度（X 轴）和高度（Y 轴）参数：第一个参数表示元素的宽度扩大的倍数，第二个参数表示元素缩小的倍数。单个方向上的放大或缩小可以用scaleX（）和scaleY（）。

例如：transform：scale（2,4）

（4）skew( )方法：通过 skew() 方法，元素翻转给定的角度，根据给定的水平线（X 轴）和垂直线（Y 轴）参数：

（5）matrix()方法：

**2、css3的动画**

**（1）@keyframs和animation属性**

@keyframs规则（关键帧）用于创建动画，在@keyframes中规定某项css样式，就能创建有当前样式逐渐改为新样式的动画效果。Chrome浏览器和safari浏览器需要前缀-webkit-。keyFrames内部可以用from和to或者百分比规定变化发生的时间。from相当于0%，to相当于100%。

创建动画后需要将其绑定到某个选择器的animaion属性并规定动画时间，否则不会产生动画的效果。

例如：@-webkit-keyframes myfirst{

from {background: red}

to{background: yellow}

}

div{

-webkit-animation: myfirst 5s;

}

**（2）requestAnimationFrame**

requestAnimationFrame会把每一帧中的所有DOM操作集中起来，在一次重绘或回流中就完成，并且重绘或回流的时间间隔使用显示器的刷新频率作为动画循环的时间间隔。**它能保证回调函数在屏幕每一次的刷新间隔中只被执行一次**，这样就不会引起丢帧现象，也不会导致动画出现卡顿的问题。

requestAnimationFrame的用法与settimeout很相似，只是不需要设置时间间隔而已。requestAnimationFrame使用一个回调函数作为参数，这个回调函数会在浏览器重绘之前调用。它返回一个整数，表示定时器的编号，这个值可以传递给cancelAnimationFrame用于取消这个函数的执行。

用requserAnimationFrame实现的一个进度条：

<div class="trans" id='trans'></div>

<div class="button" id='button'>显示进度条</div>

<script>

window.onload = function(){

var timer;

document.getElementById('button').onclick = function(){

cancelAnimationFrame(timer);

var myTrans = document.getElementById('trans');

myTrans.style.width = '0'

if(requestAnimationFrame){

requestAnimationFrame(function fn(){

var len = myTrans.style.width.length;

var oldLen = Number(myTrans.style.width.slice(0,len-2));

if(oldLen <= 500){

myTrans.style.width = oldLen + 5 + 'px';

timer = requestAnimationFrame(fn)

}else{

cancelAnimationFrame(timer);

}

})

}

}

}

</script>

**（3）settimeOut和settimeInterval实现动画效果，但是有个缺点是容易丢帧，因为settimeout本身的机制和电脑的显示屏刷新频率的限制**

**（4）animation属性与transition**

七、浮动float

1、若当前元素为浮动元素，它的上一个元素也为浮动时，当前元素会紧跟在上一个元素的后面，如果它的上一个元素不是浮动元素，那么当前元素会在上一个元素下面。

2、清除浮动的方法一般是两种：

（1）使用BFC布局，在外层元素上设置overflow：hidden

（2）使用伪元素：：after

八、BFC布局

BFC(Block formatting context)直译为"块级格式化上下文"。它是一个独立的渲染区域，只有Block-level box参与， 它规定了内部的Block-level Box如何布局，并且与这个区域外部毫不相干。

满足一下条件的元素会生成BFC元素：

* body元素
* float的值不为none(在不设置float属性的情况下，float的默认值为none)
* overflow的值不为visible(在不设置overflow属性的情况下，overflow的默认值visible）
* display的值为inline-block、table-cell、table-caption

**2、BFC的布局规则**

（1）内部的Box会在垂直方向上一个接一个的放置

（2）属于同一个BFC的两个相邻Box的margin会发生重叠，与方向无关

（3）BFC的区域不会与float的元素区域重叠

（4）计算BFC的高度时，浮动子元素也参与计算

（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元素，反之亦然

**3、利用BFC可以解决什么样的问题？**

（1）防止margin重叠

例如:<p>hahah<p>

<p>heiheihei<p>

如果对这两个元素都设置了margin：100px，那么实际上两个元素的间隔并不是200px而是100px。

解决方式就是在p元素的外面包上一层div，即设置div的overflow：hidden。这样两个p元素就不属于同一个BFC了。

（2）清除浮动

<div class = "wrap">

<div class = "in">child</div>

<div class = "in">child</div>

</div>

如果内部的div元素设置为了浮动，会导致父元素的高度塌陷。

解决方法就是将父元素设置为BFC元素，通过对class为wrap的元素设置overflow：hidden可以解决父元素塌陷的问题。

（3）两栏自适应布局：主要是借助布局规则的第三条

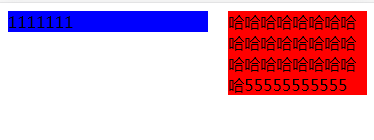
<div class="main">

<div class="left">1111111</div>

<div class="right"> 哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈55555555555</div>

</div>

给left元素设置float：left，并给right元素设置overflow：hidden

左图为设置了hidden，右图为没有设置，还是有区别的哈

九、<img>标签的title与alt属性的区别

**title**是[global attributes](http://www.w3.org/TR/html-markup/global-attributes.html#common.attrs.core)之一，用于为元素提供附加的信息。通常当鼠标滑动到元素上的时候显示。

**alt**是<img>的特有属性，是图片内容的等价描述，用于图片无法加载时显示、读屏器阅读图片。可提图片高可访问性，除了纯装饰图片外都必须设置有意义的值，搜索引擎会重点分析。

十、各种居中方法的实现

**1、水平居中**

（1）如果要居中的元素是行内元素

如果需要居中的元素为常规流中 inline / inline-block 元素，为父元素设置 text-align: center;

（2）如果要居中的元素是块元素

方法a、在要居中的元素上设置margin：0 auto

方法b、

.wrapper { //父元素

width: 80%;

height: 500px;

position: relative;

}

.center { //子元素

width: 500px;

position: absolute; //子元素设置绝对定位

left: 50%; //与父元素左边的距离为父元素宽度的一半

margin-left: -250px;//再向左移动元素宽度的一半

background-color: orange;

}

方法 c、可以只设置元素自身的position：relative，left: 50%;margin-left: -元素宽度的一半

方法d、设置父元素的display：flex，在设置justify-content: center使得子元素水平居中，

**3、水平垂直居中**

方法a、在要居中的元素的父元素上设置如下三个属性：

display：flex;

justify-content: center; //使子元素水平居中

align-items: center; //使子元素垂直居中

方法b、在要居中的元素的父元素上设置如下属性：display:table;然后在要居中的元素上设置如下属性：

display:table-cell;

text-align:center;

vertical-align:middle;

十一、display、visibility、opacity三者的区别

**1、display：none，visibility：hidden，opacity：0都可以隐藏元素，三者的区别是什么**

（1）display:none隐藏后不占据额外空间，它会产生回流和重绘

visibility:hidden和opacity:0元素虽然隐藏了，但它们仍然占据着空间，它们俩只会引起页面重绘。

（2）display:none不会被子元素继承，但是父元素都不在了，子元素自然也就不会显示了

visibility:hidden会被子元素继承，可以通过设置子元素visibility:visible使其显示出来

opacity: 0 也会被子元素继承，但是不能通过设置子元素opacity: 0使其重新显示

（3）display: none的元素都已经不再页面存在了，因此肯定也无法触发它上面绑定的事件； visibility: hidden元素上绑定的事件也无法触发；

opacity: 0元素上面绑定的事件是可以触发的。

**2、display属性有哪些取值？block与inline-block有什么区别？**

（1）display: none; ——让标签消失(隐藏元素并脱离文档流)

这个就不多说了，

（2）display: inline; ——元素内联元素(内联表签)

行内元素的display属性默认为inline，设置inline后元素就会变为行内元素，具有如下特点：

a、inline元素不会独占一行，多个相邻的行内元素会排列在同一行里，直到一行排列不下，才会新换一行

b、inline元素的宽度是自身内容的宽度(默认有多少占多少)

c、inline元素设置width和height属性无效

d、inline元素的margin和padding属性，水平方向的padding-left, padding-right, margin-left, margin-right都产生边距效果；但竖直方向的padding-top, padding-bottom, margin-top, margin-bottom不会产生边距效果。

（3）display: block; ——块级元素(块级标签)

block元素会独占一行，多个block元素会各自新起一行。默认情况下，block元素宽度自动填满其父元素宽度

block元素可以设置width和height属性。块级元素即使设置了宽度,仍然是独占一行

block元素也可以设置margin和padding属性

（4）display: inline-block; ——既有inline的属性也有block属性

说白了就是不占据一整行的块级元素，结合了inline元素与block元素的特点

注意：给inline-block元素设置margin：auto是不会生效的，所有不能用margin：0 auto对其进行居中

**这个属性有个缺陷：使用inline-block进行布局时会发现元素之间存在4px的间隙，这是由于换行引起的，解决方法是对父元素添加，{font-size:0}**

十二、单行文本溢出与多行文本溢出

1、单行文本溢出，在包裹文字的外层div设置如下三个属性：

white-space: nowrap;*/\*内容超宽后禁止换行显示\*/*overflow: hidden;*/\*超出部分隐藏\*/*text-overflow:ellipsis;*/\*文字超出部分以省略号显示\*/*

2、多行文本溢出，下面的写法适用于任何浏览器：

width: 100px;

overflow: hidden;

text-overflow: ellipsis;

display: -webkit-box;

-webkit-line-clamp: 3; //行数

-webkit-box-orient: vertical;

十三、css实现一个三角形

.triangle{//想实现哪个方向的三角形，就讲两边的border的颜色设置为transparent

这里实现一个向上的三角形  
 width: 0px;  
 height: 0px;

border-right:50px solid transparent;  
 border-left:50px solid transparent;  
 border-bottom:50px solid red;  
}这里实现的是一个正三角形，right和left的宽度变化会是三角形变长或边短

十四、前端性能优化

初级：

1、减少http请求的次数

尽可能的合并静态资源，如可以通过构建工具合并雪碧图，css和js文件

2、减少http请求的大小

可以使用gzip来压缩传输的内容，缩短网络时延。

3、将css和js文件放到外部文件中，避免使用<style>和<script>标签直接引入。

4、避免页面中有空的href和src属性

如果href和src属性为空，浏览器在渲染的过程中仍会将href属性和src属性中的空内容进行加载，直至加载失败，这样就阻塞了页面中其他资源的下载进程。

5、合理的使用浏览器的强缓存和协商缓存

6、使用静态CDN来存储文件，并且静态资源分域存放来增加下载的并行数。

cdn可以加快同一个地理区域内重复静态资源文件的响应下载，并且浏览器在同一时刻向同一个域名请求文件的并行下载数是有限的，因此可以利用多个域名的主机来存放不同的静态资源。通常是通过多个域名来分别存储js，css和图片文件。

可能会被问：现在带宽可以达到百兆，为什么还要用cdn呢？？

主要是距离问题，浏览器和服务器建立一次链接要tcp三次握手加四次挥手，如果不使用cdn，那么来回的距离非常远。

十五、css选择器的优先级

important >内联> ID >类>标签 | 伪类 | 属性选择>伪对象> 通配符>继承

注意：标签、伪类、属性选择，这三种选择器权重相同，是平级的。（完美世界的笔试考过）

十六、H5C3的新特性

**1、css新增了一些伪类选择器：**

 **:first-child**选择某个元素的第一个子元素；

 **:last-child**选择某个元素的最后一个子元素；

 **:nth-child()**选择某个元素的一个或多个特定的子元素；

 **:nth-last-child()**选择某个元素的一个或多个特定的子元素，从这个元素的最后一个子元素开始算；

 **:nth-of-type()**选择指定的元素；

 **:nth-last-of-type()**选择指定的元素，从元素的最后一个开始计算；

 **:first-of-type**选择一个上级元素下的第一个同类子元素；

 **:last-of-type**选择一个上级元素的最后一个同类子元素；

 **:only-child**选择的元素是它的父元素的唯一一个了元素；

 **:only-of-type**选择一个元素是它的上级元素的唯一一个相同类型的子元素；

 **:empty**选择的元素里面没有任何内容。

2、边框，可以创建圆角边框和阴影框

Border-radius属性和box-shadow属性

3、渐变：CSS3 定义了两种类型的渐变（gradients）：

（1）线性渐变（Linear Gradients）- 向下/向上/向左/向右/对角方向

例如：background: linear-gradient(direction, color-stop1, color-stop2,...);

（2）径向渐变（Radial Gradients）- 由它们的中心定义

例如：background: radial-gradient(center, shape size, start-color,...,last-color);

4、transform属性，上面介绍过了

5、transition属性：

**1.transition-property属性**

定义：设置对象中的参与过渡的属性

* 语法：transition-**property**：none | all | **property**

例子：transition-property：width  
说明：none： 没有属性改变； all : 默认值，所有属性都改变； property： 元素的属性名 color等

**2.transition-duration属性**

定义: 设置对象过渡的持续时间

* 语法：transition-duration：time

例子：transition-duration：1s  
说明：规定完成过渡效果需要花费的时间，以秒或者毫秒计，默认值0

**3.transition-timing-function属性**

定义：设置对象中过渡的动画类型

* 语法：transition-timing-**function**：linear 只能使用一个属性值

例子：transition-timing-function：linear  
说明：linear:线性过渡（匀速） cubic-bezier(0,0,1,1)； ease:平滑过渡（0--慢--快--慢），默认值 cubic-bezier(0.25,0.1,0.25,1)； ease-in:慢--快 cubic-bezier(0.42,0,1,1)  
ease-out:快--慢 cubic-bezier(0,0,0.58,1)； ease-in-out:慢--快--慢 cubic-bezier(0.42,0,0.58,1)；贝塞尔曲线

**4.transition-delay属性**

* 定义：设置对象延迟的过渡时间
* 语法：transition-delay：time

例子 : transition-delay：1s  
说明：指定秒或者毫秒数之前要等待切换效果的开始，默认是0

**5.transition复合属性**

* 定义：设置对象变换时的过渡
* 语法：transition ： **property** duration timing-function delay；

例子 : transition：width 1s ease 1s  
注意：时间顺序不能乱

5、新增动画效果，上文也介绍过了

6、HTML语义化标签的作用？

十七、link和@import的区别

* link属于XHTML标签，而@import是CSS提供的。
* 页面被加载时，link会同时被加载，而@import引用的CSS会等到页面被加载完再加载。
* import只在IE 5以上才能识别，而link是XHTML标签，无兼容问题。
* link方式的样式权重高于@import的权重。
* 使用dom控制样式时的差别。当使用javascript控制dom去改变样式的时候，只能使用link标签，因为@import不是dom可以控制的。

十八、vw，vh，vmin，vmax，rem，em的区别是什么？

这四个是css3新增的视窗单位，vm：视窗宽度的百分比 ，vh: 视窗高度的百分比

vm：当前较小的vw和vh ,vmax：当前较大的vw和vh

css中的body中先全局声明font-size=62.5%，这里的%的算法和rem一样。

**因为100%=16px，1px=6.25%，所以10px=62.5%，**

这是的1rem=10px，所以12px=1.2rem。

十九、给出子元素margin-top:5px, height:10px，问父元素高度是多少？

答案是父元素高度也是10px，因为会发生margin重叠，一个盒子如果没有上补白(padding-top)和上边框(border-top)，那么这个盒子的上边距会和其内部文档流中的第一个子元素的上边距重叠。

# 网络与浏览器部分

一、发送ajax请求

（1）要根据浏览器的版本生成XHR对象。IE7之前要用ActiveXObject对象，其他版本可以用XMLHttpRequest对象。

function createXHR(){

if (typeof XMLHttpRequest != "undefined"){

return new XMLHttpRequest();

}

else if (typeof ActiveXObject != "undefined"){

if (typeof arguments.callee.activeXString != "string"){

var versions = [ "MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0", "MSXML2.XMLHttp"], i, len;

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

try {

new ActiveXObject(versions[i]);

arguments.callee.activeXString = versions[i];

break;

} catch (ex){

//跳过

}

}

}

return new ActiveXObject(arguments.callee.activeXString);

}

else {

throw new Error("No XHR object available.");

}

}

（2）XHR对象有个readyState属性，表示请求或相应的活动阶段。readyState的取值在0到4之间。每当readyState的值改变时，就会触发readystatechange事件，对应一个onreadystatechange事件处理函数。

（3）发送ajax请求后，相应的数据会自动填充XHR对象的属性，其中包括responceText，responceXML，status，statusText。

（4）发送ajax请求的步骤：

//步骤一:创建异步对象

var xhr = new XMLHttpRequest();

//步骤二:设置请求的url参数,参数一是请求的类型,参数二是请求的url,可以带参数,动态的传递参数starName到服务端

xhr.open('get','getStar.php?starName='+name);

//步骤三:发送请求

xhr.send();

//步骤四:注册事件 onreadystatechange 状态改变就会调用

xhr.onreadystatechange = function () {  
 if (xhr.readyState==4) {

//步骤五 如果能够进到这个判断说明数据完美的回来了,并且请求的页面存在

if((xhr.status>=200&&xhr.status<300)||xhr.status==304){

alert(xhr.responceText)

}

else{alert("请求失败！")；}

}

}

（5）发送get请求时，需要把查询字符串追加到url的尾部，查询字符串的名和值可能会出现格式问题，所以记得要用encodeURIComponent()方法对名和值进行编码。

（6）如果是发送post请求，要想向服务器提交表单的话，需要在send方法之前设置请求头部信息。

xhr.open("post", "postexample.php", true);

xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

var form = document.getElementById("user-info");

xhr.send(serialize(form)); //对表单进行序列化

如果post请求想向服务器发送JSON数据，那么Content-Type的值应该为application/json。

牛客网笔试题：当一个 Ajax 请求由于跨域问题导致请求不能顺利完成时，是哪个环节导致的？

答案：浏览器的跨域是由同源策略导致，同源策略是指当浏览器执行一个脚本的时候会检查这个脚本是属于哪个页面的，即检查是否同源，只有同源的脚本才会被执行。

二、fetch和ajax的区别

1）fetch脱离了XHR使用了promise，是ES6规范里新的实现方式；Ajax是使用XMLHttpRequest对象的实例

2）fetch只对网络请求报错，对400，500都当做成功的请求，需要封装去处理，而ajax只有当状态为200或者304时才显示请求成功。

3）fetch默认不会带cookie，需要添加配置项。如果想要在同域中自动发送cookie,加上 credentials 的 same-origin 选项；对于跨域请求可以写成：fetch（URL，{credentials:’include’})。

4）fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了浪费

5）fetch没有办法原生监测请求的进度，而XHR可以

6）所有版本的IE都不支持fetch方法。

二、get请求与post请求的异同（百度问过）

1、GET请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割URL和传输数据，参数之间以&相连，而POST把提交的数据则放置在是requestbody中。

2、浏览器或者服务器对url的长度一般是有限制的，所以通过get请求传递参数会有限制，而post没有限制，可以传递大量的数据

3、get比post更不安全，因为参数直接暴露在url中，所以不能用来传递敏感信息。

get请求只能进行url编码，而post支持多种编码方式

get请求会浏览器主动cache，而post支持多种编码方式。

get请求参数会被完整保留在浏览历史记录里，而post中的参数不会被保留。

4、get和post本质上就是http发送请求的两种方法，底层都是TCP连接，不过对于GET方式的请求，浏览器会把http header和data一并发送出去，即只产生一个TCP包，服务器响应200（返回数据）；而对于POST，如果浏览器可以一次性将内容都发送完，则也是只产生一个TCP包，如果内容过多，浏览器会先发送header，服务器响应100 continue，浏览器再发送data，data会根据内容大小再划分成几个TCP包。最后服务器响应200 ok（返回数据）。

三、DNS解析时会发生什么？

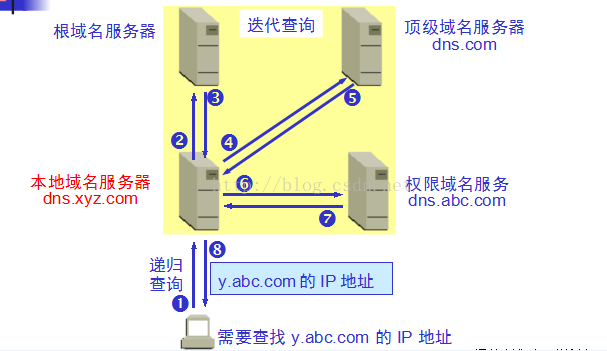
1、查询浏览器缓存（浏览器会缓存之前拿到的DNS 2-30分钟时间），如果没有找到，

2、检查系统缓存，检查hosts文件，这个文件保存了一些以前访问过的网站的域名和IP的数据。它就像是一个本地的数据库。如果找到就可以直接获取目标主机的IP地址了。没有找到的话，

3、需要检查路由器缓存，路由器有自己的DNS缓存，可能就包括了这在查询的内容；如果没有需要

4、查询ISP DNS 缓存：ISP服务商DNS缓存（本地服务器缓存）那里可能有相关的内容，如果还不行的话，需要，

5、递归查询：由本地域名服务器从根域名服务器，到顶级域名服务器，再到权限域名服务器，依次搜索对应目标域名的IP。



3、在浏览器中输入一个url到页面全加载发生了什么？（百度、小米问过）

（1）浏览器查看当前url是否存在**缓存**，如果请求资源在缓存中并且新鲜，跳转到转码步骤

（2）浏览器解析URL获取协议，主机，端口，path

（3）浏览器组装一个HTTP（GET）请求报文

（4）做 DNS 查询来获取Ip地址，获取过程如上一个问题。

（5） 接下来是 TCP 握手，应用层会下发数据给传输层，这里 TCP 协议会指明两端的端口号，然后下发给网络层。网络层中的 IP 协议会确定 IP 地址，并且指示了数据传输中如何跳转路由器。然后包会再被封装到数据链路层的数据帧结构中，最后就是物理层面的传输了 TCP 握手结束后会进行 TLS 握手，然后就开始正式的传输数据 。

（6）tcp链接建立完毕，开始发送HTTP请求。

（7）数据在进入服务端之前，可能还会先经过负责负载均衡的服务器，它的作用就是将请求合理的分发到多台服务器上，这时假设服务端会响应一个 HTML 文件 。

（8）服务器检查HTTP请求头是否包含缓存验证信息如果验证缓存新鲜，返回304等对应状态码。

（9）服务器将响应报文通过TCP连接发送回浏览器。浏览器接收HTTP响应，然后根据情况选择关闭TCP连接或者保留重用，关闭TCP连接的四次握手如下。

（10） 浏览器会判断状态码是什么，如果是 200 那就继续解析，如果 400 或 500 的话就会报错，如果 300 的话会进行重定向，这里会有个重定向计数器，避免过多次的重定向，超过次数也会报错 。

（11）浏览器开始解析文件，如果是 gzip 格式的话会先解压一下，然后通过文件的编码格式知道该如何去解码文件 。

（12）文件解码成功后会正式开始渲染流程，先会根据 HTML 构建 DOM 树，有 CSS 的话会去构建 CSSOM 树。如果遇到 script 标签的话，会判断是否存在 async 或者 defer ，前者会并行进行下载并执行 JS，后者会先下载文件，然后等待 HTML 解析完成后顺序执行，如果以上都没有，就会阻塞住渲染流程直到 JS 执行完毕。遇到文件下载的会去下载文件，这里如果使用a HTTP 2.0 协议的话会极大的提高多图的下载效率。

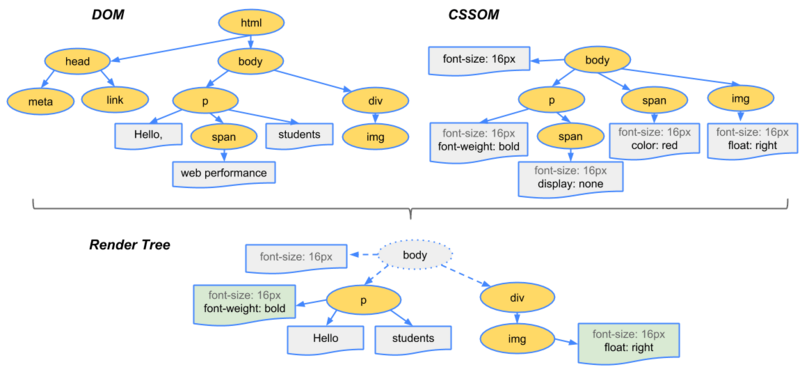
（13）初始的 HTML 被完全加载和解析后会触发 DOMContentLoaded 事件 。

（14） CSSOM 树和 DOM 树构建完成后会开始生成 Render 树，这一步就是确定页面元素的布局、样式等等诸多方面的东西 。

（15）在生成 Render 树的过程中，浏览器就开始调用 GPU 绘制，合成图层，将内容显示在屏幕上了。

四、什么是DOM树，什么是CSSOM树，什么是render树

CSSOM树的节点是什么（小米问过）



五、什么是cookie，cookie由哪些信息构成（百度大搜）

cookie是在客户端用于存储会话信息的，是网站为了辨别用户身份二存储在浏览器端的数据，大多数浏览器中cookie的大小为4KB。

（1）cookie的属性主要包括：名字，值，过期时间，路径和域。路径与域一起构成cookie的作用范围。

● Name 和 Value 属性由程序设定,默认值都是空引用。

● Domain属性的默认值为当前URL的域名部分，不管发出这个cookie的页面在哪个目录下的。

● Path属性的默认值是根目录，即 ”/” ，不管发出这个cookie的页面在哪个目录下的。可以由程序设置为一定的路径来进一步限制此cookie的作用范围。

● Expires 属性，这个属性设置此Cookie 的过期日期和时间。

注意，名字和值以外的属性是服务器给浏览器的指示，不会随着浏览器发送给服务器端

（2）获取方式，document.cookie，koa里ctx.cookie

六、cookie，localstorage，sessionstorage，session的异同

**1、cookie、localstorage、sessionstorage的异同**

相同点：都保存在浏览器端，同源的：

不同点：

（1）传递方式不同

cookie数据始终在同源的http请求中携带(即使不需要)，即cookie在浏览器和服务器间来回传递。 sessionStorage和loaclStorage不会自动把数据发给服务器，仅在本地保存。

fetch请求默认不携带cookie，cerdentials：same-origin表示可以同源发送，cerdentials：includes表示既可以同源也可以跨源

（2）数据大小不同

cookie数据还有路径(path)的概念，可以限制cookie只属于某个路径下。

存储大小限制也不同，cookie数据不能超过4k，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如会话标识。

sessionStorage和localStorage虽然也有存储大小的限制,但比cookie大得多，可以达到5M或者更大。

（3）数据有效期不同

sessionStorage:仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持；

localStorage:始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；

cookie只在设置cookie过期时间之前一直有效，即使窗口或浏览器关闭。

（4）作用域不同

sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；

localStorage在所有同源窗口中都是共享的；

cookie也是在所有同源窗口中都是共享的。

Web Storage支持事件通知机制，可以将数据更新的通知发送给监听者。

Web Storage的api接口使用更方便。

**2、cookie和session的区别**

七、前端缓存（http缓存）

前端缓存主要分成两大部分： http缓存和浏览器缓存，http缓存一般指强缓存协商缓存等，浏览器缓存是浏览器端用于在本地保存数据并进行快速读取以避免重复资源请求的传输机制的统称。浏览器有9种缓存机制：Http文件缓存，localstorage，sessionstorage，indexDB，cookie，web SQL，cacheStorage，application cache。

**当浏览器请求资源时，如何判断资源是否有缓存？**

在第一次请求页面成功返回资源后，会记录的每一个缓存文件与其url的对应关系，存储在本地，下一次再请求的时候，或者浏览器打开之后，通过请求资源的url去匹配是否有对应的资源；来确认这个文件是否有缓存（此处并不能确认文件是否已经过期）；

**1、HTTP 1.0: 基于Pragma&Expires的缓存实现**

当该pragma字段值为“no-cache”的时候（事实上现在RFC中也仅标明该可选值），会知会客户端不要对该资源读缓存，即每次都得向服务器发一次请求才行。Pragma属于通用首部字段，在客户端上使用时，常规要求我们往html上加上 <meta http-equiv="Pragma" content="no-cache">（仅对该页面有效，对页面上的资源无效）

pragma有很大的限制：仅有IE才能识别这段meta标签含义，其它主流浏览器仅能识别“Cache-Control: no-store”的meta标签。

选择合适的缓存策略：

对于大部分的场景都可以使用强缓存配合协商缓存解决，但是在一些特殊的地方可能需要选择特殊的缓存策略

* 对于某些不需要缓存的资源，可以使用 Cache-control: no-store ，表示该资源不需要缓存
* 对于频繁变动的资源，可以使用 Cache-Control: no-cache 并配合 ETag 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新。
* 对于代码文件来说，通常使用Cache-Control: max-age=31536000并配合策略缓存使用，然后对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件。

七、既然已经有了Etag，那Last-modified还有存在的必要么？

Etag和Last-modified是相辅相成的，会互相弥补对方的缺陷，

Etag可以弥补last-modified的一些不足：

例如：1、一些文件也许会周期性的更改，但是他的内容并不改变(仅仅改变的修改时间)，这个时候我们并不希望客户端认为这个文件被修改了，而重新GET

2、某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说1s内修改了N次)，If-Modified-Since能检查到的粒度是s级的，这种修改无法判断(或者说UNIX记录MTIME只能精确到秒)；

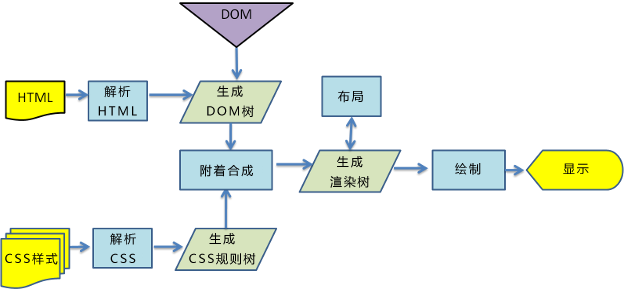
Last-modified也可以弥补etag的不足：

比如一些图片等静态文件的修改，如果每次扫描内容生成 ETag 来比较（ETag是通过资源内容来生成的），显然要比直接比较修改时间慢很多。所有说这两种判断是相辅相成的。

需要注意的是 如果同时有 etag 和 last-modified 存在，在发送请求的时候会一次性的发送给服务器，没有优先级，服务器会比较这两个信息（在具体实现上，大多数做法针对这种情况只会比对 etag）。服务器在输出上，如果输出了 etag 就没有必要再输出 last-modified（实际上大多数情况会都输出）。

八、浏览器的渲染流程

**1、浏览器渲染引擎的主要工作流程：**



（1）进程之间的任务传递  
 浏览器的主线程browser负责下载资源，即获取页面内容（HTML文档），然后将该页面通过rendererHost接口将任务传递给浏览器内核，或者叫做render进程。

（2）浏览器内核通过renderer接口接受任务，进行简单的解释后交给渲染线程。

（3）构建DOM树

渲染线程解析HTML文件生成DOM树。

（4）构建CSSOM规则树  
 渲染线程解析CSS文件并生成CSS规则树，每个CSS文件都被分析成一个StyleSheet对象，每个对象都包含CSS规则。CSS规则对象包含对应于CSS语法的选择器和声明对象以及其他对象。

（5）渲染阻塞  
 当渲染线程遇到一个 script 标记时，DOM 构建将暂停，直至脚本完成执行，然后继续构建DOM。每次去执行JavaScript脚本都会严重地阻塞DOM树的构建，如果JavaScript脚本还操作了CSSOM，而正好这个CSSOM还没有下载和构建，浏览器甚至会延迟脚本执行和构建DOM，直至完成其CSSOM的下载和构建。

（6）构建渲染树  
 通过DOM树和CSS规则树我们便可以构建render树。渲染线程会先从DOM树的根节点开始遍历每个可见节点。对每个可见节点，找到其适配的CSS样式规则并应用。

注意：渲染树每个节点都是可见节点（属性的dispaly：none的就不会出现在渲染树中）并且都含有其内容和对应规则的样式

（7）渲染树布局  
 布局阶段会从渲染树的根节点开始遍历，然后确定每个节点对象在页面上的确切大小与位置，布局阶段的输出是一个盒子模型，它会精确地捕获每个元素在屏幕内的确切位置与大小。

（6）渲染树绘制  
 在绘制阶段，遍历渲染树，调用[渲染器](https://www.baidu.com/s?wd=%E6%B8%B2%E6%9F%93%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的paint()方法在屏幕上显示其内容。渲染树的绘制工作是由浏览器的UI后端组件完成的。

（9）渲染线程将各层的信息发送给GPU进程，GPU将各个图层合并，将合并后的结果渲染到屏幕上。

八、页面的重排重绘

（1）什么是重排重绘？

重绘是当节点需要更改外观而不会影响布局的，比如改变 color 就叫称为重绘

重排是布局或者几何属性需要改变就称为重排。

设置元素的属性为display：none会触发重排，但是visibility：hidden不会。

（2）减少浏览器重排重绘的方法

● 改变节点样式的时候将所有改变合并。例如：

var el = document.querySelector('.el');

el.style.borderLeft = '1px';

el.style.borderRight = '2px';

el.style.padding = '5px';

上述的操作再最糟糕的情况下会触发浏览器的三次重排。优化的方法是合并所有改变一次处理。

var el = document.querySelector('.el');

el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px';

或者是定义一个新的类，直接切换类名

.active {

padding: 5px;

border-left: 1px;

border-right: 2px;

}

var el = document.querySelector('.el');

el.className = 'active';

● 批量修改DOM节点的情况下如何减少重排重绘的次数？主要思想：让元素脱离文档流，对其进行多重改变，让元素回到文档流中。

例如：我们要在ul节点中新增多个li节点。

<ul id="mylist">

<li><a href="https://www.mi.com">xiaomi</a></li>

<li><a href="https://www.miui.com">miui</a></li>

</ul>

将新数据更新到指定节点的方法为：

function appendNode($node, data) {

var a, li;

for(let i = 0, max = data.length; i < max; i++) {

a = document.createElement('a');

li = document.createElement('li');

a.href = data[i].url;

a.appendChild(document.createTextNode(data[i].name));

li.appendChild(a);

$node.appendChild(li);

}

}

如果是常规写法，不考虑重排重绘的性能问题的话，我们想更新元素会采用如下的写法：

let ul = document.querySelector('#mylist');

appendNode(ul, data);

但是这种写法在每次插入li节点时都会造成一次重排。

**优化的方法a：**使用**display**属性。

let ul = document.querySelector('#mylist');

ul.style.display = 'none';//将要操作的元素“消失”

appendNode(ul, data);

ul.style.display = 'block';//将要操作的元素“显示”

**优化的方法b：**使用**createDocumentFragment**，这个方法会创建一个虚拟的节点对象，它有一个很实用的特点，当请求把一个DocumentFragment节点插入文档树时，插入的不是DocumentFragment自身，而是它的所有子孙节点，即插入的是括号里的节点。

let fragment = document.createDocumentFragment();

appendNode(fragment, data);//在新创建的fragement上操作

ul.appendChild(fragment);

**优化的方法c**、拷贝节点，然后**操作节点的副本**，最后替换原节点。

let old = document.querySelector('#mylist');

let clone = old.cloneNode(true);//创建一个要操作的节点的副本

appendNode(clone, data);

old.parentNode.replaceChild(clone, old);

●尽可能的在DOM树的最末端改变class

● 避免设置多层内联样式，应将样式合并在一个外部类里

● 将动画效果应用到position属性为absolute或fixed的元素上，这样不会影响其他元素的布局，只会导致重绘。

九、浏览器内的进程与浏览器内核

**1、什么是进程和线程？**

进程是操作系统分配资源的最小单元，线程是操作系统调度的最小单元。一个程序至少一个进程，一个进程至少一个线程。

2、浏览器内的进程

首先，浏览器是多进程的，之所以浏览器能够运行，是因为系统给浏览器分配了资源，如cpu、内存，简单的说就是，浏览器每打开一个标签页，就相当于创建了一个独立的浏览器进程。例如我们查看chrome里面的任务管理器。

**浏览器到底包含了哪些进程呢？**

（1）Browser进程：浏览器的主进程（负责协调、主控），只有一个。作用有

● 负责浏览器界面显示，与用户交互。如前进，后退等

● 负责各个页面的管理，创建和销毁其他进程

● 将Renderer进程得到的内存中的Bitmap，绘制到用户界面上

● 网络资源的管理，下载等

（2）第三方插件进程：每种类型的插件对应一个进程，仅当使用该插件时才创建

（3）GPU进程：最多一个，用于3D绘制等

（4）浏览器渲染进程（浏览器内核）（Renderer进程，内部是多线程的）：默认每个Tab页面一个进程，互不影响。主要作用为页面渲染，脚本执行，事件处理等

**3、浏览器多进程的优势**

* 避免单个page 崩溃影响整个浏览器
* 避免第三方插件崩溃影响整个浏览器
* 多进程充分利用多核优势
* 方便使用沙盒模型隔离插件等进程，提高浏览器稳定性

**4、浏览器内核（重点！！！！！）**

**浏览器的渲染进程是多线程的，主要包含以下线程：**

**（1）GUI渲染线程**

* 负责渲染浏览器界面，解析HTML，CSS，构建DOM树和RenderObject树，布局和绘制等。
* 当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时，该线程就会执行
* 注意，**GUI渲染线程与JS引擎线程是互斥的**，当JS引擎执行时GUI线程会被挂起（相当于被冻结了），GUI更新会被保存在一个队列中**等到JS引擎空闲时**立即被执行。

**（2）js引擎线程**

* 也称为JS内核，负责处理Javascript脚本程序。（例如V8引擎）
* JS引擎线程负责解析Javascript脚本，运行代码。
* JS引擎一直等待着任务队列中任务的到来，然后加以处理，一个Tab页（renderer进程）中无论什么时候都只有一个JS线程在运行JS程序
* 同样注意，**GUI渲染线程与JS引擎线程是互斥的**，所以如果JS执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞。

（3）事件触发线程

* 归属于浏览器而不是JS引擎，用来控制事件循环（可以理解，JS引擎自己都忙不过来，需要浏览器另开线程协助）
* 当JS引擎执行代码块如setTimeOut时（也可来自浏览器内核的其他线程,如鼠标点击、AJAX异步请求等），会将对应任务添加到事件线程中
* 当对应的事件符合触发条件被触发时，该线程会把事件添加到js引擎任务队列的队尾，等待JS引擎的处理
* 注意，由于JS的单线程关系，所以这些待处理队列中的事件都得排队等待JS引擎处理（当JS引擎空闲时才会去执行）

（4）定时触发器线程

* 传说中的setInterval与setTimeout所在线程
* 浏览器定时计数器并不是由JavaScript引擎计数的,（因为JavaScript引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确）
* 因此通过单独线程来计时并触发定时（计时完毕后，添加到任务队列中，等待JS引擎空闲后执行）
* 注意，W3C在HTML标准中规定，规定要求setTimeout中低于4ms的时间间隔算为4ms。

（5）异步HTTP请求线程

* XMLHttpRequest在连接后是通过浏览器新开一个线程请求
* 将检测到状态变更时，如果设置有回调函数，异步线程就**产生状态变更事件**，将这个回调再放入事件队列中，再由JavaScript引擎执行。

**5、浏览器内核中线程之间的关系**

（1）**GUI渲染线程与JS引擎线程互斥**

由于JavaScript是可操纵DOM的，如果在修改这些元素属性同时渲染界面（即JS线程和UI线程同时运行），那么渲染线程前后获得的元素数据就可能不一致了。因此为了防止渲染出现不可预期的结果，浏览器设置GUI渲染线程与JS引擎为互斥的关系，当JS引擎执行时GUI线程会被挂起，GUI更新则会被保存在一个队列中等到JS引擎线程空闲时立即被执行。

十、图片加载优化

静态资源尽量使用 CDN 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名。对于 CDN 加载静态资源需要注意 CDN 域名要与主站不同，否则每次请求都会带上主站的 Cookie。

CDN主要功能是在不同的地点缓存内容，通过负载均衡技术，将用户的请求定向到最合适的缓存[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)上去获取内容，

十一、HTTPs如何保证数据安全传输，或者HTTPS如何进行数字证书验证

HTTPS是通过加入**SSL**层来加密HTTP数据进行安全传输的HTTP协议，SSL协议具体见 《现代前端技术解析》*P60。*

1、先了解一下数字证书，它有点像身份证，是由权威的CA机构颁发的，证书的主要内容有：公钥（Public Key）、ISSUER（证书的发布机构）、Subject（证书持有者）、证书有效期、签名算法、指纹及指纹算法。

指纹是一个证书的签名，是通过指纹算法sha1计算出来的一个hash值，这个很重要，是用来验证证书内容有没有被篡改的。签名就是在**信息**的后面再加上一段内容，可以证明**信息**没有被修改过，怎么样可以达到这个效果呢？一般是对**信息**做一个hash计算得到一个hash值，注意，这个过程是不可逆的，也就是说无法通过hash值得出原来的**信息**内容。在把**信息**发送出去时，把这个hash值加密后做为一个签名和**信息**一起发出去。 接收方在收到**信息**后，会重新计算**信息**的hash值，并和**信息**所附带的hash值(解密后)进行对比，如果一致，就说明信息的内容没有被修改过，因为这里hash计算可以保证不同的内容一定会得到不同的hash值，所以只要内容一被修改，根据**信息**内容计算的hash值就会变化。

1. HTTPS的通信过程：



1）客户端发起握手协商操作，它将发送一个ClientHello消息给服务器，消息中明确了其所支持的SSL/TLS版本、Cipher suite加密算法组合等，可以让服务器选择使用哪个加密算法，并提供了一个客户端随机数，用于以后生成会话密钥使用。

2）服务器将返回一个ServerHello消息，该消息包含了服务器选择的协议版本、加密算法，以及服务器随机数、会话ID等内容。其中，服务器选择的协议版本应小于等于客户端ClientHello中的协议版本。

3）服务器发送ServerHello消息，选择好协议版本和加密算法组合后，将发送Certificate（证书）消息，该消息包含了服务器的证书等信息，可通过证书链认证该证书的真实性。根据选择的加密算法组合的不同，服务器证书中的公钥也可被用于加密后面握手过程中生成的Premaster secret。

4）服务器发送ServerKeyExchange消息，消息中包含了服务器这边的EC Diffie-Hellman（这个算法是用来传输对称加密的秘钥的，简称DH算法）算法相关参数。

5）服务器发送ServerHelloDone消息，告知客户端服务器这边握手相关的消息发送完毕。

6）客户端发送ClientKeyExchange消息，消息中包含客户端这边的EC Diffie-Hellman算法相关参数，然后客户端都可根据接收到的服务器随机数和自身随机数运算出Premaster secret（它是客户端使用一些加密算法(例如：RSA, Diffie-Hellman)产生一个48个字节的Key，这个Key叫 PreMaster Secret，服务器端也会生成一个相同的Premaster secret），为生成会话密钥做准备。

7）客户端向服务器发送ChangeCipherSpec消息，通知服务器客户端已经切换到之前协商好的加密套件（Cipher Suite）的状态，此消息以后客户端会以加密方式发送数据。

8）在ChangecipherSpec传输完毕之后，客户端会使用之前协商好的加密套件和服务器整数中的公钥加密一段 preMaster secret 传送给服务端，此数据是为了在正式传输应用数据之前对刚刚握手建立起来的加解密通道进行验证。

9）服务端在接收到客户端传过来的 PreMaster 加密数据之后，使用私钥对这段加密数据进行解密，并对数据进行验证，也会使用客户端随机数，自己的随机数和Premaster Secret来生成session secret，客户端也使用同样的方式生成 Session Secret，一切准备好之后，会给客户端发送一个 ChangeCipherSpec，告知客户端已经切换到协商过的加密套件状态，准备使用加密套件和 Session Secret加密数据了。

10）服务端使用 Session Secret 加密一段 Finish 消息发送给客户端，以验证之前通过握手建立起来的加解密通道是否成功。

十二、HTTP各版本的区别

**1、HTTP1.0与HTTP1.1之间的区别**

**1) HTTP1.0**规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP（客户到服务器往返所花时间）连接，服务器不跟踪每个客户也不记录过去的请求。主要缺点就是每请求一个文档就要有两倍的RTT开销。

**2）HTTP1.1支持长连接**，或者叫做持久链接（persistent connection）。

在HTTP1.1协议中，请求消息中默认包含connection：keep-alive的头部信息，这样可以让客户端到服务器端之间的联系在一段时间内持续有效，在这段持续的时间内，客户端与服务器的其他文件请求可以复用这个已经建好的链接。

HTTP1.1 持久链接有两种方式，非流水线方式和流水线方式（with pipelining），非流水线是在客户收到前一个响应后才能发出下一个请求，流水线方式是客户在收到HTTP响应报文之前就能够接着发送新的请求报文。

**2、HTTP1.1与HTTP2.0的区别：**

现代前端技术解析*P52*

1）HTTP2.0采用完全二进制的格式来传输数据，而非1.1版本默认的文本格式，二进制在网络中传输的单位为帧，多个帧在网络中就形成了传输流。同时HTTP2.0对消息头部采用HPACK压缩传输，很大程度节省传输带宽。

2）HTTP2.0采用TCP多路复用的方式来降低网络请求连接建立和关闭的开销。这里的TCP多路复用是帧的多路复用，即不同的文件的传输帧可以在一个TCP连接中进行流式传输。这与http1.1中的不同，流水线方式的复用是串行的，一个文件传输完之后下一个文件才能复用这个连接。

3）HTTP2.0支持服务器端推送，服务器在指定的条件下能够把资源主动推送给客户端。

4）HTTP2.0支持传输流的优先级和流量控制机制，每个文件传输流都有自己的传输优先级，可以通过服务器来动态改变。

十三、TCP的流量控制

流量控制就是让发送方的发送速率不要太快，要让接收方来的及接收。

A向B发送数据时，

十四、TCP的拥塞控制

1、慢开始，

慢开始算法步骤具体如下

1. 连接初始设置拥塞窗口（Congestion Window） 为 1 MSS（一个分段的最大数据量）
2. 每过一个 RTT 就将窗口大小乘二
3. 指数级增长肯定不能没有限制的，所以有一个阈值限制，当窗口大小大于阈值时就会启动拥塞避免算法。

2、拥塞避免算法

在传输过程中可能定时器超时的情况，这时候 TCP 会认为网络拥塞了，会马上进行以下步骤：

将阈值设为当前拥塞窗口的一半

将拥塞窗口设为 1 MSS

启动拥塞避免算法

3、快重传与快启动

快速重传一般和快恢复一起出现。一旦接收端收到的报文出现失序的情况，接收端只会回复最后一个顺序正确的报文序号（没有 Sack 的情况下）。如果收到三个重复的 ACK，无需等待定时器超时再重发而是启动快速重传。具体算法分为两种：

TCP Taho 实现如下

将阈值设为当前拥塞窗口的一半

将拥塞窗口设为 1 MSS

重新开始慢开始算法

TCP Reno 实现如下

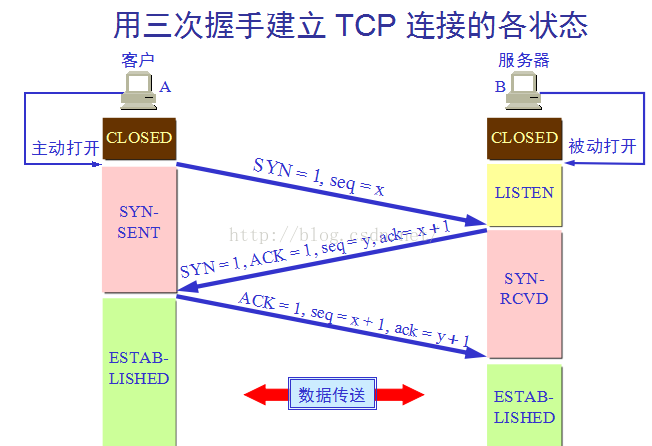
拥塞窗口减半

将阈值设为当前拥塞窗口

进入快恢复阶段（重发对端需要的包，一旦收到一个新的 ACK 答复就退出该阶段）

使用拥塞避免算法

十五、TCP三次握手

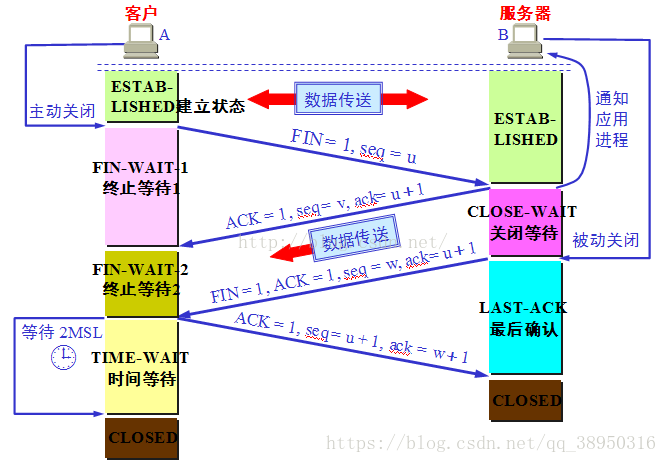


**2、问：为什么是三次握手而不是两次握手或者四次握手？？**

这主要是为了防止已失效的连接请求报文段突然又传回到了服务器端。我们考虑这样一种正常情况，客户端A发出链接请求，但是因为连接请求报文丢失而未收到确认，于是A再次重传链接请求，后来A收到了确认，建立了链接。数据传输完毕后就释放了链接。A总共发了两个请求报文段，其中一个丢失一个到达了服务器端。

但是假设现在出现了一种异常的情况，A发出的第一个连接请求报文段并没有丢失，而是在某个网络节点上滞留了，以至于延误到链接请求释放后的某个时间才到达服务端B。这时候就这个报文段就是“已失效的连接请求报文段”

十六、TCP四次挥手



十七、web开发中的会话跟踪有哪些

1、首先了解为什么要进行会话跟踪？

浏览器与服务器之间的通信是通过HTTP协议进行通信的，而HTTP协议是”无状态”的协议，它不能保存客户的信息，即一次响应完成之后连接就断开了，下一次的请求需要重新连接，这样就需要判断是否是同一个用户，所以才应会话跟踪技术来实现这种要求

2、会话跟踪的四种方法

**（1）URL重写：**

URL重写的技术就是在URL结尾添加一个附加数据以标识该会话,把会话ID通过URL的信息传递过去，以便在服务器端进行识别不同的用户 。

**（2）隐藏表单域**  
 将会话ID添加到HTML表单元素中提交到服务器,此表单元素并不在客户端显示 。

<input type="hidden">

**（3）使用cookie**

Cookie是Web服务器发送给客户端的一小段信息，客户端请求时可以读取该信息发送到服务器端，进而进行用户的识别。对于客户端的每次请求，服务器都会将Cookie发送到客户端,在客户端可以进行保存,以便下次使用。

**（4）使用 session**

在服务器端会创建一个session对象，产生一个sessionID来标识这个session对象，然后将这个sessionID放入到Cookie中发送到客户端，下一次访问时，sessionID会发送到服务器，在服务器端进行识别不同的用户。

十八、websoket与ajax的区别

先介绍一下websocket：

首先Websocket是基于HTTP协议的，或者说借用了HTTP的协议来完成一部分握手。为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，其中附加头信息"Upgrade: WebSocket，Connection: Upgrade"表明这是一个申请协议升级的 HTTP 请求，服务器端解析这些附加的头信息然后产生应答信息返回给客户端，客户端和服务器端的 WebSocket 连接就建立起来了，双方就可以通过这个连接通道自由的传递信息，并且这个连接会持续存在直到客户端或者服务器端的某一方主动的关闭连接。

1.本质不同

 Ajax,即异步JavaScript和XML，是一种创建交互式网页应用的网页开发技术，很多网站为了实现推送技术，所用的技术都是 Ajax 轮询。浏览器需要不断的向服务器发出HTTP请求，

 WebSocket 是 HTML5 开始提供的一种在单个TCP连接上进行全双工通讯的协议。

2.生命周期不同。

websocket建立的是长连接，在一个会话中一直保持连接；而ajax是短连接，数据发送和接受完成后就会断开连接。

3.适用范围不同

websocket一般用于前后端实时数据交互，而ajax前后端非实时数据交互。

4.发起人不同

Ajax技术需要客户端发起请求，而WebSocket服务器和客户端可以相互推送信息。

5、跨域的限制不同

ajax发送的请求会有跨域的问题，而websocket不会

十九、HTTP的状态码

1XX：表示消息

|  |  |  |  |
| --- | --- | --- | --- |
| **状态代码** | **状态信息** | **简述** | **含义** |
| 100 | Continue | 初始的请求已经接受，客户应当继续发送请求的其余部分。（HTTP 1.1新） | 客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。  客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应。 |
| 101 | Switching Protocols | 服务器将遵从客户的请求转换到另外一种协议（HTTP 1.1新） | 服务器已经理解了客户端的请求，并将通过Upgrade 消息头通知客户端采用不同的协议来完成这个请求。  在发送完这个响应最后的空行后，服务器将会切换到在Upgrade 消息头中定义的那些协议。 只有在切换新的协议更有好处的时候才应该采取类似措施。例如，切换到新的HTTP版本比旧版本更有优势，或者切换到一个实时且同步的协议以传送利用此类特性的资源。 |
| 102 | Processing |  | 由WebDAV（RFC 2518）扩展的状态码，代表处理将被继续执行。 |

2XX：成功

|  |  |  |  |
| --- | --- | --- | --- |
| **状态代码** | **状态信息** | **简述** | **含义** |
| 200 | OK | 一切正常，对GET和POST请求的应答文档跟在后面。 | 请求已成功，请求所希望的响应头或数据体将随此响应返回。 |
| 201 | Created | 服务器已经创建了文档，Location头给出了它的URL。 | 请求已经被实现，而且有一个新的资源已经依据请求的需要而建立，且其URI已经随Location头信息返回。假如需要的资源无法及时建立的话，应当返回’202 Accepted’。 |
| 202 | Accepted | 已经接受请求，但处理尚未完成。 | 服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行。在异步操作的场合下，没有比发送这个状态码更方便的做法了。 返回202状态码的响应的目的是允许服务器接受其他过程的请求（例如某个每天只执行一次的基于批处理的操作），而不必让客户端一直保持与服务器的连接直到批处理操作全部完成。在接受请求处理并返回202状态码的响应应当在返回的实体中包含一些指示处理当前状态的信息，以及指向处理状态监视器或状态预测的指针，以便用户能够估计操作是否已经完成。 |
| 203 | Non-Authoritative Information | 文档已经正常地返回，但一些应答头可能不正确，因为使用的是文档的拷贝（HTTP 1.1新）。 | 服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上有效的确定集合，而是来自本地或者第三方的拷贝。当前的信息可能是原始版本的子集或者超集。例如，包含资源的元数据可能导致原始服务器知道元信息的超集。使用此状态码不是必须的，而且只有在响应不使用此状态码便会返回200 OK的情况下才是合适的。 |
| 204 | No Content | 没有新文档，浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而Servlet可以确定用户文档足够新，这个状态代码是很有用的。 | 服务器成功处理了请求，但不需要返回任何实体内容，并且希望返回更新了的元信息。响应可能通过实体头部的形式，返回新的或更新后的元信息。如果存在这些头部信息，则应当与所请求的变量相呼应。 如果客户端是浏览器的话，那么用户浏览器应保留发送了该请求的页面，而不产生任何文档视图上的变化，即使按照规范新的或更新后的元信息应当被应用到用户浏览器活动视图中的文档。 由于204响应被禁止包含任何消息体，因此它始终以消息头后的第一个空行结尾。 |
| 205 | Reset Content | 没有新的内容，但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容（HTTP 1.1新）。 | 服务器成功处理了请求，且没有返回任何内容。但是与204响应不同，返回此状态码的响应要求请求者重置文档视图。该响应主要是被用于接受用户输入后，立即重置表单，以便用户能够轻松地开始另一次输入。 与204响应一样，该响应也被禁止包含任何消息体，且以消息头后的第一个空行结束。 |

3XX：重定向

|  |  |  |  |
| --- | --- | --- | --- |
| **状态代码** | **状态信息** | **简述** | **含义** |
| 300 | Multiple Choices | 客户请求的文档可以在多个位置找到，这些位置已经在返回的文档内列出。 | 被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。 除非这是一个 HEAD 请求，否则该响应应当包括一个资源特性及地址的列表的实体，以便用户或浏览器从中选择最合适的重定向地址。这个实体的格式由 Content-Type 定义的格式所决定。浏览器可能根据响应的格式以及浏览器自身能力，自动作出最合适的选择。当然，RFC 2616规范并没有规定这样的自动选择该如何进行。 |
| 301 | Moved Permanently | 客户请求的文档在其他地方，新的URL在Location头中给出，浏览器应该自动地访问新的URL。 | 被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一。 |
| 302 | Move temporarily | 类似于301，但新的URL应该被视为临时性的替代，而不是永久性的。 | 请求的资源临时从不同的URI响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。如果这不是一个GET或者HEAD请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。 |
| 303 | See Other | 类似于301/302，不同之处在于，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）。 | 对应当前请求的响应可以在另一个 URI 上被找到，而且客户端应当采用 GET 的方式访问那个资源。这个方法的存在主要是为了允许由脚本激活的POST请求输出重定向到一个新的资源。这个新的 URI 不是原始资源的替代引用。同时，303响应禁止被缓存。当然，第二个请求（重定向）可能被缓存。 |
| 304 | Not Modified | 客户端有缓冲的文档并发出了一个条件性的请求（一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。 | 如果客户端发送了一个带条件的 GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码。304响应禁止包含消息体，因此始终以消息头后的第一个空行结尾。 |

4XX：请求错误

|  |  |  |  |
| --- | --- | --- | --- |
| **状态代码** | **状态信息** | **简述** | **含义** |
| 400 | Bad Request | 请求出现语法错误。 | 1、语义有误，当前请求无法被服务器理解。2、请求参数有误。 |
| 401 | Unauthorized | 客户试图未经授权访问受密码保护的页面。应答中会包含一个 | 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源的 |
| 402 | Payment Required | 预留状态码 |  |
| 403 | Forbidden | 资源不可用。服务器理解客户的请求，但拒绝处理它。通常由于服务器上文件或目录的权限设置导致。 | 服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。 |
| 404 | Not Found | 无法找到指定位置的资源。这也是一个常用的应答。 | 请求失败，请求所希望得到的资源未被在服务器上发现。 |
| 405 | Method Not Allowed | 请求方法（GET、POST、HEAD、DELETE、PUT、TRACE等）对指定的资源不适用。（HTTP 1.1新） | 请求行中指定的请求方法不能被用于请求相应的资源。该响应必须返回一个Allow 头信息用以表示出当前资源能够接受的请求方法的列表。 PUT，DELETE 方法会对服务器上的资源进行写操作，因而绝大部分的网页服务器都不支持或者在默认配置下不允许上述请求方法，对于此类请求均会返回405错误。 |
| 406 | Not Acceptable | 指定的资源已经找到，但它的MIME类型和客户在Accpet头中所指定的不兼容（HTTP 1.1新）。 | 请求的资源的内容特性无法满足请求头中的条件，因而无法生成响应实体。 HEAD 请求，否则该响应就应当返回一个包含可以让用户或者浏览器从中选择最合适的实体特性以及地址列表的实体。实体的格式由 Content-Type 头中定义的媒体类型决定。浏览器可以根据格式及自身能力自行作出最佳选择。但是，规范中并没有定义任何作出此类自动选择的标准。 |
| 407 | Proxy Authentication Required | 类似于401，表示客户必须先经过代理服务器的授权。（HTTP 1.1新） | 与401响应类似，只不过客户端必须在代理服务器上进行身份验证。代理服务器必须返回一个 Proxy-Authenticate 用以进行身份询问。客户端可以返回一个 Proxy-Authorization 信息头用以验证。参见RFC 2617。 |
| 408 | Request Timeout | 在服务器许可的等待时间内，客户一直没有发出任何请求。客户可以在以后重复同一请求。（HTTP 1.1新） | 请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这一请求而无需进行任何更改。 |
| 409 | Conflict | 通常和PUT请求有关。由于请求和资源的当前状态相冲突，因此请求不能成功。（HTTP 1.1新） | 由于和被请求的资源的当前状态之间存在冲突，请求无法完成。这个代码只允许用在这样的情况下才能被使用：用户被认为能够解决冲突，并且会重新提交新的请求。该响应应当包含足够的信息以便用户发现冲突的源头。 冲突通常发生于对 PUT 请求的处理中。例如，在采用版本检查的环境下，某次 PUT 提交的对特定资源的修改请求所附带的版本信息与之前的某个（第三方）请求向冲突，那么此时服务器就应该返回一个409错误，告知用户请求无法完成。此时，响应实体中很可能会包含两个冲突版本之间的差异比较，以便用户重新提交归并以后的新版本。 |

二十、301和302的区别

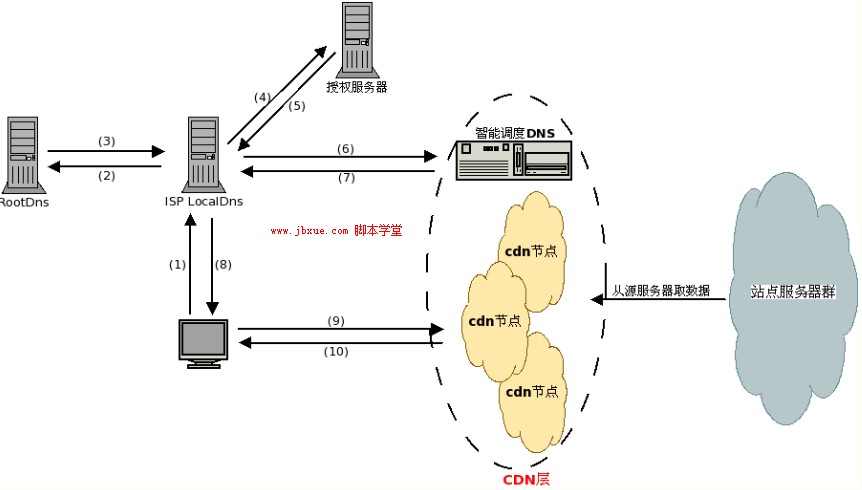
* 301表示旧地址A的资源已经被永久地移除了（这个资源不可访问了），**搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址**。301请求是可以缓存的，即通过查看statuscode可以发现后面写着fromcache。
* 302表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B，**搜索引擎会抓取新的内容而保存旧的网址。**

注意：尽量使用301进行跳转！！

302 重定向可能会有url劫持的问题。比如说，有的时候A 网址很短，但是它做了一个302重定向到B网址，而B网址是一个很长的乱七八糟的URL网址，甚至还有可能包含一些问号之类的参数。很自然的，A网址更加用户友好，而B网址既难看，又不用户友好。这时Google很有可能会仍然显示网址A。由于搜索引擎排名算法只是程序而不是人，在遇到302重定向的时候，并不能像人一样的去准确判定哪一个网址更适当，这就造成了网址URL劫持的可能性。也就是说，一个不道德的人在他自己的网址A做一个302重定向到你的网址B，出于某种原因， Google搜索结果所显示的仍然是网址A，但是所用的网页内容却是你的网址B上的内容，这种情况就叫做网址URL 劫持。你辛辛苦苦所写的内容就这样被别人偷走了。302重定向所造成的网址URL劫持现象，已经存在一段时间了。不过到目前为止，似乎也没有什么更好的解决方法。

在脉脉实习时候遇到过重定向的情况：脉脉主node运行在3000端口，而招聘node运行在3033端口，招聘node想要调用主node的接口，这时候就会遇到跨域的问题，一博在node层写了一个重定向，在3033端口找不到就去3000端口找

二十一、CDN的工作原理



（1）用户输入访问的域名,操作系统向 LocalDns 查询域名的ip地址.  
（2）LocalDns向 ROOT DNS 查询域名的授权服务器(这里假设LocalDns缓存过期)  
（3）ROOT DNS将域名授权dns记录回应给 LocalDns  
（4）LocalDns得到域名的授权dns记录后,继续向域名授权dns查询域名的ip地址  
    域名授权dns 查询域名记录后(一般是CNAME)，回应给 LocalDns  
    LocalDns 得到域名记录后,向智能调度DNS查询域名的ip地址  
    智能调度DNS 根据一定的算法和策略(比如静态拓扑，容量等),将最适合的CDN节点ip地址回应给 LocalDns  
    LocalDns 将得到的域名ip地址，回应给 用户端  
    用户得到域名ip地址后，访问站点服务器  
    CDN节点服务器应答请求，将内容返回给客户端.(缓存服务器一方面在本地进行保存，以备以后使用，二方面把获取的数据返回给客户端，完成数据服务过程)

二十二、什么是跨域，为什么会有跨域，解决跨域的几种方法？

**1、跨域是什么**

**2、为什么会有跨域？**

浏览器的同源策略，浏览器沙箱与沙箱，相当于进程与进程的隔离，沙箱之间通信sop，即同源策略

不同源之间的js代码如何相互通信，同源的话，不同网页之间的js代码可以相互操作对方的dom，随意访问cookie

不同源的限制行为：假设a,b不同源

**1、a不能访问b的dom**

首先，攻击者会在网页中内嵌一个iframe将自己伪造成真实的网站，iframe中是真的<www.yinhang.com>。

<iframe name="yinhang" src="www.yinhang.com"></iframe>

然后，因为没有同源策略限制，它就能拿到<www.yinhang.com>网站的DOM。

window.frames['yinhang'].document.getElementById('你输入账号密码的input').val();

通过DOM去获取输入的账号密码

**2、a不能读取b的cookie，localstorage等缓存**

**3、a请求b域内的接口**

举个例子：

受害者 Bob 在银行有一笔存款，通过对银行的网站发送请求 <http://bank.example/withdraw?account=bob&amount=1000000&for=bob2> 可以使 Bob 把 1000000 的存款转到 bob2 的账号下。通常情况下，该请求发送到网站后，服务器会先验证该请求是否来自一个合法的 session，并且该 session 的用户 Bob 已经成功登陆。

黑客 Mallory 自己在该银行也有账户，他知道上文中的 URL 可以把钱进行转帐操作。Mallory 可以自己发送一个请求给银行：<http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory>。但是这个请求来自 Mallory 而非 Bob，他不能通过安全认证，因此该请求不会起作用。

这时，Mallory 想到使用 CSRF 的攻击方式，他先自己做一个网站，在网站中放入如下代码： src=”<http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory> ”，并且通过广告等诱使 Bob 来访问他的网站。当 Bob 访问该网站时，上述 url 就会从 Bob 的浏览器发向银行，而这个请求会附带 Bob 浏览器中的 cookie 一起发向银行服务器。大多数情况下，该请求会失败，因为他要求 Bob 的认证信息。但是，如果 Bob 当时恰巧刚访问他的银行后不久，他的浏览器与银行网站之间的 session 尚未过期，浏览器的 cookie 之中含有 Bob 的认证信息。这时，悲剧发生了，这个 url 请求就会得到响应，钱将从 Bob 的账号转移到 Mallory 的账号，而 Bob 当时毫不知情。等以后 Bob 发现账户钱少了，即使他去银行查询日志，他也只能发现确实有一个来自于他本人的合法请求转移了资金，没有任何被攻击的痕迹。而 Mallory 则可以拿到钱后逍遥法外。

**对于Cookie，DOM和XMLHttpRequest（ajax）所有浏览器都会严格遵守同源策略。但是也有例外，如<img>标签，<script>标签，<iframe>标签等的链接会自动加载，更重要的是，表单提交也是可以跨域的。**所以黑客就可以通过这些标签进行CSRF攻击。

**3、解决跨域的几种方法**

（4）借助代理跨域

可以在node层借助http-proxy-middle实现跨域

假设现在在3000端口起一个服务，然后向3001端口发起请求，在3000端口的node文件里可以加上下面的代码：

var proxy = require('http-proxy-middleware');

var requestPort = 3000; // 请求页面跑在3000端口

var app = express();

app.use(express.static(\_\_dirname));

app.use('/api', proxy({target: 'http://localhost:3001/', changeOrigin: true}));

// changeOrigin设置为true，本地会虚拟一个服务端接收你的请求并代你发送该请求

// http://localhost:3000/api --> http://localhost:3001/api

如果不使用插件实现跨域的话

在3000端口发送请求时带上一个参数，如下所示：

btn.addEventListener('click', function () {

var xhr = new XMLHttpRequest();

var proxy\_url = 'http://localhost:3000/proxy?url=http://localhost:3001/';

// 携带参数{url: 'http://localhost:3001/'}向http://localhost:3000/proxy发出请求，获取数据

// porxy\_url是代理的url，不是最终请求的url，最终请求的url放在问号后面，在后台可以通过req.query.url获取

// 向porxy\_url发出请求，不会存在跨域问题，因为都是运行在3000端口

// url为实际请求的地址，向3000/proxy发出请求的同时携带这个包含url的对象，这个url在这里只是参数，不是请求路径

xhr.open('GET', proxy\_url);

xhr.send();

xhr.onreadystatechange = () => { // 请求状态改变后调用这个函数

if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) { // 如果请求成功

text.innerHTML = xhr.response;

}

}

})

然后在3000端口的node文件中获取这个参数中的url，向这个url发送请求

var requestPort = 3000; // 请求页面跑在3000端口

app.get('/proxy', function(request, response){

var url = request.query.url // http://localhost:3001/

// 向url发出请求

http.get(url, function(responseFromOtherDomain) {

// data事件会在数据接收过程中，每收到一段数据就触发一次，接收到的数据被传入回调函数。

responseFromOtherDomain.on("data", function(data) {

response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});

response.end(data);

});

});

});

（5）websocket实现跨域

在3000端口引入socket.io，并实例化一个io对象，通过io.connect链接3001端口，并且订阅一个data时间

var io = io.connect('http://127.0.0.1:3001');

io.on('data', function (data) {

alert('2s后改变数据👻');

p.innerHTML = data

});

再3001端口也实例化一个io对象，在3001端口订阅一个connection事件，当3000端口与之链接成功的时候触发回调函数，函数中发布一个data事件，第二个参数为要传给3000端口的数据。

var io = require('socket.io')(server);

io.on('connection', function (client) {

client.emit('data', 'Hello WebSocket from 3001.');

});

server.listen(3001, function () {

console.log('Responser is listening on port 3001');

});

//监听3001端口

二十三、csrf攻击

受害者 Bob 在银行有一笔存款，通过对银行的网站发送请求 <http://bank.example/withdraw?account=bob&amount=1000000&for=bob2> 可以使 Bob 把 1000000 的存款转到 bob2 的账号下。通常情况下，该请求发送到网站后，服务器会先验证该请求是否来自一个合法的 session，并且该 session 的用户 Bob 已经成功登陆。

黑客 Mallory 自己在该银行也有账户，他知道上文中的 URL 可以把钱进行转帐操作。Mallory 可以自己发送一个请求给银行：<http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory>。但是这个请求来自 Mallory 而非 Bob，他不能通过安全认证，因此该请求不会起作用。

这时，Mallory 想到使用 CSRF 的攻击方式，他先自己做一个网站，在网站中放入如下代码： src=”<http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory> ”，并且通过广告等诱使 Bob 来访问他的网站。当 Bob 访问该网站时，上述 url 就会从 Bob 的浏览器发向银行，而这个请求会附带 Bob 浏览器中的 cookie 一起发向银行服务器。大多数情况下，该请求会失败，因为他要求 Bob 的认证信息。但是，如果 Bob 当时恰巧刚访问他的银行后不久，他的浏览器与银行网站之间的 session 尚未过期，浏览器的 cookie 之中含有 Bob 的认证信息。这时，悲剧发生了，这个 url 请求就会得到响应，钱将从 Bob 的账号转移到 Mallory 的账号，而 Bob 当时毫不知情。等以后 Bob 发现账户钱少了，即使他去银行查询日志，他也只能发现确实有一个来自于他本人的合法请求转移了资金，没有任何被攻击的痕迹。而 Mallory 则可以拿到钱后逍遥法外。

**1） 检测http的referer**: http有一个referer字段，它记录了该 HTTP 请求的来源地址。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上来讲，这样并不安全。事实上，对于某些浏览器，比如 IE6 或 FF2，目前已经有一些方法可以篡改 Referer 值。

2）**使用csrf token**，CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。

使用access\_token，脉脉的前端发送请求时会讲uid和access\_token都在url中，例如：

var url = fconf.online\_url + 'contact/v4/pre\_init\_req?' + qs.stringify({  
         u2: u2,  
         acc: search\_value,  
         ...auth\_info,  /\*\* auth\_info里面就包含了uid和access\_tokens  
        from: from || 'nofr'  
});

脉脉的的后端会检验url中的uid和access\_token，而不是cookie中的access\_token和uid

二十四、XSS攻击

二十五、content-type都有哪些值？

**Content-Type 存在于请求和响应的头部，用于标识数据的类型。**

简介

Http 请求头里的Content-Type一般有这三种：

application/x-www-form-urlencoded ：数据被编码为名称/值对。这是标准的编码格式。

multipart/form-data ： 数据被编码为一条消息，页上的每个控件对应消息中的一个部分。

text/plain ： 数据以纯文本形式(text/json/xml/html)进行编码，其中不含任何控件或格式字符。

form的 enctype 属性为编码方式，常用有两种：application/x-www-form-urlencoded 和multipart/form-data，默认为application/x-www-form-urlencoded 。

当form表单发送的数据格式为键值对的形式的时候，（1）如果是get请求，浏览器用 x-www-form-urlencoded 的编码方式把form数据转换成一个字串（name1=value1&name2=value2...），然后把这个字串追加到url后面，用 ? 分割，加载这个新的url。此时context-type是x-www-form-urlencoded；（2）如果是post请求，浏览器把数据封装到http body中，然后发送到server。如果没有 type=file 的控件，用默认的application/x-www-form-urlencoded就可以了。

如果要上传文件的话就要用到multipart/form-data了。上传文件时或者form的enctype的值设置为multipart/form-data，那么context-type的值就变成了multipart/form-data。（enctype 属性规定在发送到服务器之前应该如何对表单数据进行编码。）

如果提交的数据是JSON字符串的话，context-type：application/json

常用的第三方请求库 Axios 、fetch默认就是用的 application/json。

Http 响应头里的Content-Type是用来告诉浏览器，服务端返回了什么格式的数据，常用的有如下几种：

text/html、text/xml、text/css，

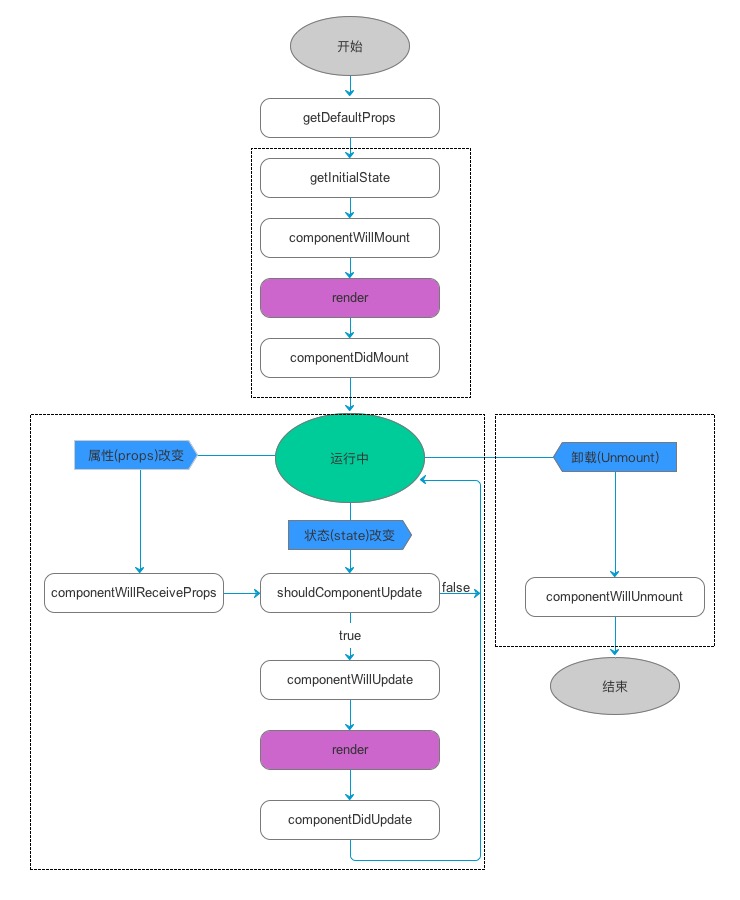
application/javascript、

img/png、img/gif

# react框架部分与打包工具

一、react的生命周期

**1、react16之前的生命周期：**



（1）创建阶段

该阶段主要发生在创建组件类的时候，即调用React.createClass的时候。这个阶段只会触发一个getDefaultProps方法，该方法返回一个对象，并且缓存下来。然后与父组件指定的props对象合并，最后赋值给this.props作为该组件的默认属性。对于那些没有被父辈组件指定的props属性的新建实例来说，这个方法返回的对象可用于为实例设置默认的props值。

（2）实例化阶段

* **getInitialState** 初始化组件的state的值，其返回值会赋值给组件的this.state属性。对于组件的每个实例来说，这个方法的调用次数有且只有一次。与getDefaultProps方法不同的是，每次实例创建时该方法都会被调用一次。
* **componentWillMount** 此方法会在完成首次渲染之前被调用。这也是在render方法调用前可以修改组件state的最后一次机会。在这个方法内修改state，render会知道state发生了改变。
* **render** 生成页面需要的虚拟DOM结构，用来表示组件的输出。render方法需要满足：（1）只能通过this.props和this.state访问数据；（2）可以返回null、false或者任何React组件；（3）只能出现一个顶级组件；（4）必需纯净，意味着不能改变组件的状态或者修改DOM的输出。
* **componentDidMount** 该方法发生在render方法成功调用并且真实的DOM已经被渲染之后，在该函数内部可以通过this.getDOMNode()来获取当前组件的节点。然后就可以像Web开发中的那样操作里面的DOM元素了。

（3）存在阶段

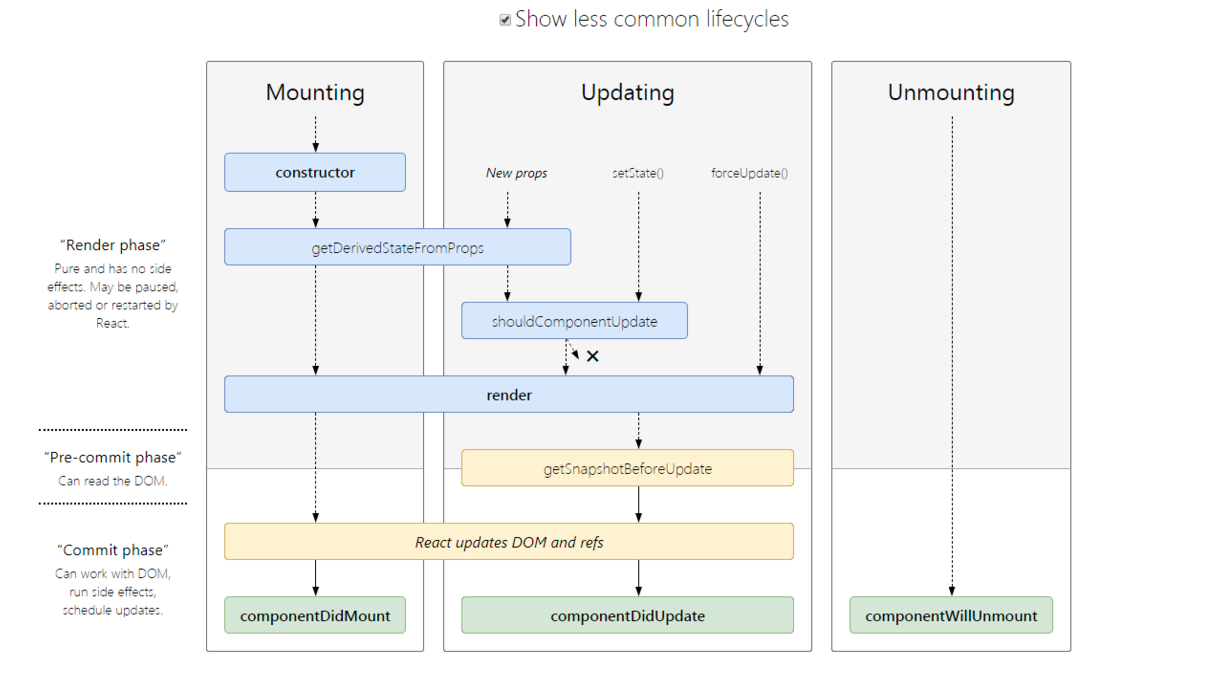
* **componentWillReceiveProps（nextProps）** 在任意时刻，组件的props都可以通过父辈组件来更改。当组件接收到新的props(这里不同于state）时，会触发该函数，我们同时也获得更改props对象及更新state的机会。
* **shouldComponentUpdate** 该方法用来拦截新的props和state，然后开发者可以根据自己设定逻辑，做出要不要更新render的决定，让它更快。
* **componentWillUpdate** 与componentWillMount方法类似，组件上会接收到新的props或者state渲染之前，调用该方法。但是不可以在该方法中更新state和props。
* **render** 生成页面需要的虚拟DOM结构，并返回该结构
* **componentDidUpdate** 与componentDidMount类似，更新已经渲染好的DOM。

（4）销毁阶段

**componentWillUnmount**

最后，随着一个组件从它的层级结构中移除，这个组件的生命也就走到了尽头。该方法会在组件被移出之前调被调用。在componentDidMount方法中添加的所有任务都需要在该方法中撤销，比如说创建的定时器或者添加的事件监听等。

2、react16之后的生命周期



挂载阶段:

* constructor: 构造函数，最先被执行,我们通常在构造函数里初始化state对象或者给自定义方法绑定this
* getDerivedStateFromProps: static getDerivedStateFromProps(nextProps, prevState),这是个静态方法,当我们接收到新的属性想去修改我们state，可以使用getDerivedStateFromProps，注意：getDerivedStateFromProps是一个静态函数，也就是这个函数不能通过this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及prevState来进行判断，根据新传入的props来映射到state，并且返回一个对象，这个对象将会作为新的state，**如果props传入的内容不需要影响到你的state，那么就需要返回一个null。**

**为什么要摒弃componentWillReceivedProps而使用getDerivedStateFromProps呢？**

**原因是性能问题：当外部多个属性在很短的时间间隔之内多次变化，就会导致componentWillReceiveProps被多次调用。这个调用并不会被合并，如果这次内容都会触发异步请求，那么可能会导致多个异步请求阻塞。**在使用getDerivedStateFromProps的时候，遇到了上面说的props在很短的时间内多次变化，也只会触发一次render，也就是只触发一次getDerivedStateFromProps。这样的优点不言而喻。

* render: render函数是纯函数，只返回需要渲染的东西，不应该包含其它的业务逻辑,可以返回原生的DOM、React组件、Fragment、Portals、字符串和数字、Boolean和null等内容
* componentDidMount: 组件装载之后调用，此时我们可以获取到DOM节点并操作，比如对canvas，svg的操作，服务器请求，订阅都可以写在这个里面，但是记得在componentWillUnmount中取消订阅

更新阶段:

* getDerivedStateFromProps: 此方法在更新个挂载阶段都可能会调用
* shouldComponentUpdate: shouldComponentUpdate(nextProps, nextState),有两个参数nextProps和nextState，表示新的属性和变化之后的state，返回一个布尔值，true表示会触发重新渲染，false表示不会触发重新渲染，默认返回true,我们通常利用此生命周期来优化React程序性能
* render: 更新阶段也会触发此生命周期
* getSnapshotBeforeUpdate: getSnapshotBeforeUpdate(prevProps, prevState),这个方法在render之后，componentDidUpdate之前调用，有两个参数prevProps和prevState，表示之前的属性和之前的state，这个函数有一个返回值，会作为第三个参数传给componentDidUpdate，如果你不想要返回值，可以返回null，此生命周期必须与componentDidUpdate搭配使用
* componentDidUpdate: componentDidUpdate(prevProps, prevState, snapshot),该方法在getSnapshotBeforeUpdate方法之后被调用，有三个参数prevProps，prevState，snapshot，表示之前的props，之前的state，和snapshot。第三个参数是getSnapshotBeforeUpdate返回的,如果触发某些回调函数时需要用到 DOM 元素的状态，则将对比或计算的过程迁移至 getSnapshotBeforeUpdate，然后在 componentDidUpdate 中统一触发回调或更新状态。

卸载阶段:

* componentWillUnmount: 当我们的组件被卸载或者销毁了就会调用，我们可以在这个函数里去清除一些定时器，取消网络请求，清理无效的DOM元素等垃圾清理工作

二、异步setstate，setState到底是异步的还是同步的？

setState函数的设计是用来更动state值，调用setstate后就默认触发组件重新渲染(re-render)，按照逻辑就是反正不管如何，只要开发者呼叫setState，React就去作整个视图的重新渲染。但是state的值并不是都与视图有关，很有可能是内部控制用的属性值，当你改变了这些与视图无关的state(状态)值，以现在的React设计来说，照样要触发重新渲染的执行过程，

解决的办法：

React提供了shouldComponentUpdate方法让开发者可以自行判断，自行提供对应的解决方式，

**2、setState的是异步还是同步的？**

（1）**setState 只在合成事件和钩子函数中是“异步”的，在原生事件和setTimeout 中都是同步的。**

（2）setState 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的callback拿到更新后的结果。

（3）setState 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和setTimeout 中不会批量更新，在“异步”中如果对同一个值进行多次setState，setState的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时setState多个不同的值，在更新时会对其进行合并批量更新。

三、react和Vue的区别

相似之处：都存在虚拟DOM的概念，将一个DOM映射成js对象，如果需要改变DOM节点要现在虚拟DOM上进行，当有变化产生时，一个新的Virtual DOM对象会被创建并计算新旧Virtual DOM之间的差别。之后这些差别会应用在真实的DOM上。

**不同之处：**

（1）在计算虚拟DOM的差异时，VUE在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。而在react中，每当应用的状态被改变时，全部子组件都会重新渲染。

（2）Vue的语法更贴近HTML的写法，只不过引入了一些属性，而React是采用JSX语法。

（3）在vue中数据存放在vue对象的data属性里，而在react中，组件的数据由state对象保管，不能直接更改，需要调用setstate方法。

（4）vue是双向数据绑定，而react是单向数据流，数据只能从父组件流向子组件

四、前端路由react-router的实现原理

**（1）什么是前端路由？**

路由就是指当浏览器的地址栏发生变化时页面显示不同的内容。

**（2）react-router实现路由的方式？**

1、BrowserHistory：

利用 HTML5 新增的history API：history.pushState()和history.repalceState()。

history.pushState({name: 'grimmjoy'}, '', 'http://127.0.0.1:5500/grimmjoy')

执行上面的操作后，浏览器的地址栏会变成第三个参数设置的值，而且会在浏览器的历史状态栈中增加一个新的状态信息。

history支持一个事件popState，但是坑爹的是当我们使用pushState或者replaceState去改变浏览器的地址栏时并不会触发popState事件，只有History.back()、History.forward()、History.go()事件会触发popstate事件的

想要监听pushState和replaceState，我们可以自己自定义一个事件

var \_wr = function (type) { // \_wr方法相当于新返回一个函数，

var orig = history[type]; // 获取history上type属性对应的值，这里就是拿到了pushState对应的函数

return function () { // 这个返回的函数会作为pushState的替代

var rv = orig.apply(this, arguments); // 函数执行时会先去执行原先的pushState函数

var e = new Event(type); // 创建一个event

e.arguments = arguments;

window.dispatchEvent(e); // 当触发这个event，然后就会被window监听到并调用回调函数

return rv;

};

};

history.pushState = \_wr('pushState'); // 自定义一个pushState事件

window.addEventListener('pushState', function (event) {

console.log('chufa')

})

或者可以用发布订阅模式实现自定义事件，在红宝书上404页

2、HashHistory：用于旧版Web浏览器

react-router可以通过location.hash来控制hash值的改变，通过下面的写法就可以在url的地址后面加上“#grimmjoy”。

location.hash = '#grimmjoy'

当hash值的改变时，都会在浏览器的访问历史中增加一个记录。因此我们能通过浏览器的回退、前进按钮控制hash的切换。而且URL中hash值只是客户端的一种状态，也就是说当向服务器端发出请求时，hash部分不会被发送。

注意：hash改变会触发window上的一个“hashchange”事件，通过对该事件进行监听我们就可以实现hash值不同页面显示的内容不同。

window.addEventListener('hashchange', function (event) {

let content = document.getElementById('content');

content.innerText = '从' + event.oldURL + '跳转到' + event.newURL

})

五、webpack打包原理

**1、Webpack 的运行流程是一个串行的过程**，从启动到结束会依次执行以下流程：

初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；

开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；

确定入口：根据配置中的 entry 找出所有的入口文件；

编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；

完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；

输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；

输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

**2、webpack打包原理**

原文的url**：**<https://www.jianshu.com/p/0e5247f9975f>

在webpack生成的bundle中，主要是一个立即执行函数，他的参数就是所有用到的模块，webpack会为每个模块分配一个值表示其在数组中的位置，立即执行函数里还会声明一个\_\_webpack\_require函数，主要是用来加载模块并提取模块中的export

**1.打包单一模块**

webpack.config.js

module.exports = {

entry:"./chunk1.js",

output: {

path: \_\_dirname + '/dist',

filename: '[name].js'

},

};

chunk1.js

var chunk1=1;

exports.chunk1=chunk1;

打包后，main.js(webpack生成的一些注释已经去掉)

(function(modules) { // webpackBootstrap

// The module cache

var installedModules = {};

// The require function

function \_\_webpack\_require\_\_(moduleId) {

// Check if module is in cache

if(installedModules[moduleId])

return installedModules[moduleId].exports;

// Create a new module (and put it into the cache)

var module = installedModules[moduleId] = {

exports: {},

id: moduleId,

loaded: false

};

// Execute the module function

modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);

// Flag the module as loaded

module.loaded = true;

// Return the exports of the module

return module.exports;

}

// expose the modules object (\_\_webpack\_modules\_\_)

\_\_webpack\_require\_\_.m = modules;

// expose the module cache

\_\_webpack\_require\_\_.c = installedModules;

// \_\_webpack\_public\_path\_\_

\_\_webpack\_require\_\_.p = "";

// Load entry module and return exports

return \_\_webpack\_require\_\_(0);

})([function(module, exports) {

var chunk1=1;

exports.chunk1=chunk1;

}]);

这其实就是一个立即执行函数，简化一下就是：

(function(module){})([function(){},function(){}]);

OK,看一下自运行的匿名函数里面干了什么：

function(modules) { // webpackBootstrap

// modules就是一个数组，元素就是一个个函数体，就是我们声明的模块

var installedModules = {};

// The require function

function \_\_webpack\_require\_\_(moduleId) {

...

}

// expose the modules object (\_\_webpack\_modules\_\_)

\_\_webpack\_require\_\_.m = modules;

// expose the module cache

\_\_webpack\_require\_\_.c = installedModules;

// \_\_webpack\_public\_path\_\_

\_\_webpack\_require\_\_.p = "";

// Load entry module and return exports

return \_\_webpack\_require\_\_(0);

}

整个函数里就声明了一个变量installedModules 和函数**webpack\_require**，并在函数上添加了一个m,c,p属性，m属性保存的是传入的模块数组，c属性保存的是installedModules变量，P是一个空字符串。最后执行**webpack\_require**函数，参数为零，并将其执行结果返回。下面看一下**webpack\_require**干了什么：

function \_\_webpack\_require\_\_(moduleId) {

//moduleId就是调用是传入的0

// installedModules[0]是undefined,继续往下

if(installedModules[moduleId])

return installedModules[moduleId].exports;

// module就是{exports: {},id: 0,loaded: false}

var module = installedModules[moduleId] = {

exports: {},

id: moduleId,

loaded: false

};

// 下面接着分析这个

modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);

// 表明模块已经载入

module.loaded = true;

// 返回module.exports(注意modules[moduleId].call的时候module.exports会被修改)

return module.exports;

}

接着看一下modules[moduleId].call(module.exports, module, module.exports, **webpack\_require**)，其实就是

modules[moduleId].call({}, module, module.exports, \_\_webpack\_require\_\_)

对call不了解当然也可以认为是这样(但是并不是等价，call能确保当模块中使用this的时候，this是指向module.exports的)：

function a(module, exports) {

var chunk1=1;

exports.chunk1=chunk1;

}

a(module, exports,\_\_webpack\_require\_\_);

传入的module就是{exports: {},id: 0,loaded: false}，exports就是{}，**webpack\_require**就是声明的**webpack\_require**函数(传入这个函数有什么用呢，第二节将会介绍)；  
运行后module.exports就是{chunk1:1}。所以当我们使用chunk1这个模块的时候（比如var chunk1=require("chunk1"),得到的就是一个对象{chunk1:1}）。如果模块里没有exports.chunk1=chunk1或者module.exports=chunk1得到的就是一个空对象{}

**2.使用模块**

上面我们已经分析了webpack是怎么打包一个模块的（入口文件就是一个模块），现在我们来看一下使用一个模块，然后使用模块的文件作为入口文件  
webpack.config.js

module.exports = {

entry:"./main.js",

output: {

path: \_\_dirname + '/dist',

filename: '[name].js'

}

};

main.js

var chunk1=require("./chunk1");

console.log(chunk1);

打包后

(function (modules) { // webpackBootstrap

// The module cache

var installedModules = {};

// The require function

function \_\_webpack\_require\_\_(moduleId) {

// Check if module is in cache

if (installedModules[moduleId])

return installedModules[moduleId].exports;

// Create a new module (and put it into the cache)

var module = installedModules[moduleId] = {

exports: {},

id: moduleId,

loaded: false

};

// Execute the module function

modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);

// Flag the module as loaded

module.loaded = true;

// Return the exports of the module

return module.exports;

}

// expose the modules object (\_\_webpack\_modules\_\_)

\_\_webpack\_require\_\_.m = modules;

// expose the module cache

\_\_webpack\_require\_\_.c = installedModules;

// \_\_webpack\_public\_path\_\_

\_\_webpack\_require\_\_.p = "";

// Load entry module and return exports

return \_\_webpack\_require\_\_(0);

})([function (module, exports, \_\_webpack\_require\_\_) {

var chunk1=\_\_webpack\_require\_\_(1);

console.log(chunk1);

}, function (module, exports) {

var chunk1 = 1;

exports.chunk1 = chunk1;

}]);

不一样的地方就是自执行函数的参数由

[function(module, exports) { var chunk1=1; exports.chunk1=chunk1;}]

变为

[function (module, exports, \_\_webpack\_require\_\_) {

var chunk1=\_\_webpack\_require\_\_(1);

console.log(chunk1);

}, function (module, exports) {

var chunk1 = 1;

exports.chunk1 = chunk1;

}]

其实就是多了一个main模块，不过这个模块没有导出项，而且这个模块依赖于chunk1模块。所以当运行**webpack\_require**(0)的时候，main模块缓存到installedModules[0]上，modules[0].call(也就是调用main模块)的时候，chunk1被缓存到installedModules[1]上，并且导出对象{chunk1：1}给模块main使用

**3.重复使用模块**

webpack.config.js

module.exports = {

entry:"./main.js",

output: {

path: \_\_dirname + '/dist',

filename: '[name].js'

}

};

main.js

var chunk1=require("./chunk1");

var chunk2=require(".chunlk2");

console.log(chunk1);

console.log(chunk2);

chunk1.js

var chunk2=require("./chunk2");

var chunk1=1;

exports.chunk1=chunk1;

chunk2.js

var chunk2=1;

exports.chunk2=chunk2;

打包后

(function (modules) { // webpackBootstrap

// The module cache

var installedModules = {};

// The require function

function \_\_webpack\_require\_\_(moduleId) {

// Check if module is in cache

if (installedModules[moduleId])

return installedModules[moduleId].exports;

// Create a new module (and put it into the cache)

var module = installedModules[moduleId] = {

exports: {},

id: moduleId,

loaded: false

};

// Execute the module function

modules[moduleId].call(module.exports, module, module.exports, \_\_webpack\_require\_\_);

// Flag the module as loaded

module.loaded = true;

// Return the exports of the module

return module.exports;

}

// expose the modules object (\_\_webpack\_modules\_\_)

\_\_webpack\_require\_\_.m = modules;

// expose the module cache

\_\_webpack\_require\_\_.c = installedModules;

// \_\_webpack\_public\_path\_\_

\_\_webpack\_require\_\_.p = "";

// Load entry module and return exports

return \_\_webpack\_require\_\_(0);

})([function (module, exports, \_\_webpack\_require\_\_) {

var chunk1 = \_\_webpack\_require\_\_(1);

var chunk2 = \_\_webpack\_require\_\_(2);

console.log(chunk1);

console.log(chunk2);

}, function (module, exports, \_\_webpack\_require\_\_) {

\_\_webpack\_require\_\_(2);

var chunk1 = 1;

exports.chunk1 = chunk1;

}, function (module, exports) {

var chunk2 = 1;

exports.chunk2 = chunk2;

}]);

不难发现，当需要重复使用模块的时候，缓存变量installedModules 就起作用了

**3、webpack都有哪些配置：**

webpack常用配置包括：

（2）entry

（3）output

（4）module：module对象里会定义一些选项，规定了webpack如何处理这些模块

module: {

noParse: [/front\_end\/dist\/base.js/],

rules: [

{

test: /\.css?$/,

loader: 'style-loader!css-loader'

},

{

test: /\.less?$/,

// loader: path.resolve(\_\_dirname, './node\_modules/happypack/loader?id=less')

loader: "style-loader!css-loader!less-loader"

},

{

test:/\.jade$/,

loader:"jade-loader"

},

// {

// test: /\.(jpg|png)$/,

// use: ['url?limit=8192']

// },

// {

// test: /\.(tpl|html)$/,

// use: ['string']

// },

// { test: /\.(eot|ttf|woff|woff2|svg)\??.\*$/,

// use: ['url-loader?name=fonts/[name].[md5:hash:hex:7].[ext]']

// },

{

test: /\.(js|jsx)$/,

exclude: /(node\_modules|bower\_components)/,

// loader: path.resolve(\_\_dirname, './node\_modules/happypack/loader?id=script')

loader: 'babel-loader?cacheDirectory=true',

options: {

babelrc: false,

presets: ["react", "env", "es2015", "stage-0" ],

plugins: ["transform-async-to-generator"]

}

}

]

},

noParse是防止 webpack 解析那些任何与给定正则表达式相匹配的文件，

rules是规定了一些规则，这些规则能够对模块(module)应用 loader，或者修改解析器(parser)。

（5）mode：分为development和production两种，区别在于不同模式下webpack启用的插件不同

（6）resolve：resolve可以修改webpack解析时候的一些细节

例如：当在 ES2015 中调用 import "react"，resolve 选项能够对 webpack 查找 "react" 的方式去做修改，

"react": path.resolve(\_\_dirname, './node\_modules/react'),

（7）plugins：

（8）performance：这些选项可以控制 webpack 如何通知「资源(asset)和入口起点超过指定文件限制」

const config = {

performance: {

// false | "error" | "warning" // 不显示性能提示 | 以错误形式提示 | 以警告...

hints: "warning",

// 开发环境设置较大防止警告

// 根据入口起点的最大体积，控制webpack何时生成性能提示,整数类型,以字节为单位

maxEntrypointSize: 5000000,

// 最大单个资源体积，默认250000 (bytes)

maxAssetSize: 3000000

}

}

hints：表示显示提示的方式

**2、webpack中loader和plugin的区别？**

* loader 用于对模块的源代码进行转换。Webpack将一切文件视为模块，但是webpack原生是只能解析js文件，如果想将其他文件也打包的话，就会用到loader。 所以Loader的作用是让webpack拥有了加载和解析\_非JavaScript文件\_的能力。loader 可以将文件从不同的语言（如 TypeScript）转换为 JavaScript，或将内联图像转换为 data URL。loader 甚至允许你直接在 JavaScript 模块中 import CSS文件！ 因为 webpack 本身只能处理 JavaScript，如果要处理其他类型的文件，就需要使用 loader 进行转换，loader 本身就是一个函数，接受源文件为参数，返回转换的结果。
* **Plugin**直译为"插件"。Plugin可以扩展webpack的功能，让webpack具有更多的灵活性。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

**3、项目中用过哪些plugin？**

（1）html-webpack-plugin可以根据模板自动生成html代码，并自动引用css和js文件

（2）happypack：通过多进程模型，来加速代码构建

（3）HotModuleReplacementPlugin 它允许在运行时更新所有类型的模块，而不需要完全刷新。

（4）extract-text-webpack-plugin ： 用于分离js与css，less文件。

（5）DllPlugin 和 DllReferencePlugin：用于分离第三方类库，在招聘那儿主要是将react，react-dom和axios分离出来。

**DLLPlugin** 它能把第三方库代码分离开，并且每次文件更改的时候，它只会打包该项目自身的代码，不会再去打包第三方库文件。所以打包速度会更快。另外需要在文件夹中配置一个webpack.config.dll.js文件，webpack.dll.config.js作用是把所有的第三方库依赖打包到一个bundle的dll文件里面，还会生成一个名为 manifest.json文件。

**DllReferencePlugin** 这个插件是在webpack.config.js中使用的，dllPlugin打包时会生成一个manifest.json文件，文件存放了第三方库文件的一个映射。当在使用webpack.config.js文件打包DllReferencePlugin插件的时候，会使用该DllReferencePlugin插件读取vendor-manifest.json文件，看看是否有该第三方库。

// 不是很懂

plugins: [

new webpack.DllReferencePlugin({

context: \_\_dirname,

manifest: require('./prd/script/dll-manifest.json'),

}),

（6）UglifyJsPlugin：打包后的文件很可能会含有一些空格、换行和注释，使得打包出来的文件过大，该插件可以对文件进行压缩，删除不需要的代码。

六、如何实现组件按需加载

**1、require.ensure()方法**

才发现脉脉里面在做落地页的时候就是用的按需加载，当匹配到对应路由的时候，执行回调函数也就是调用require.ensure。

ensure函数的用法如下

require.ensure(dependencies: String[], callback: function(require), chunkName: String)

* 依赖 dependencies

这是一个字符串数组，通过这个参数，在所有的回调函数的代码被执行前，我们可以将所有需要用到的模块进行声明。

* 回调 callback

当所有的依赖都加载完成后，webpack会执行这个回调函数。require 对象的一个实现会作为一个参数传递给这个回调函数。因此，我们可以进一步 require() 依赖和其它模块提供下一步的执行。

* chunk名称 chunkName

chunkName 是提供给这个特定的 require.ensure() 的 chunk 的名称。通过提供 require.ensure() 不同执行点相同的名称，我们可以保证所有的依赖都会一起放进相同的文件束(bundle)。这个新的 chunk 会被 webpack 通过 jsonp 来按需加载。

router.addRoute("/activity/birthday\_greetings", function (c) {   
   return require.ensure([], (require) => {   
       var m = require("./routes/activity/birthday\_greetings");   
       return (c ? c(m) : m) })   
},'birthday\_greetings');

**2、import()方法**

import("./module").then(module => { // module为模块  
 return module.default;  
}).catch(err => {  
 console.log("Chunk loading failed");  
});

**3、react-router+react-loadable实现按需加载**

引入Loadable函数，该函数接受一个对象作为参数，对象包含两个属性，loader表示按需加载的组件，一般都是通过import动态加载的，loading表示在组件加载期间显示的内容。

import Loadable from 'react-loadable';

const MyLoadingComponent = ( ) => <div />;  
<Route path="/bizmember/login/mobile\_code\_login" exact component={Loadable({  
     loader: () => import('./mobile-code-login'),  
     loading: MyLoadingComponent  
})} />

但是使用react-loadable需要引入babel-plugin-syntax-dynamic-import并在babelrc文件中增加一个"syntax-dynamic-import"。

**4、react-router的getComponents函数**

七、react的ref属性

ref是React提供的用来操纵React组件实例或者DOM元素的接口。它有两种使用方式：

// 作用于组件的实例

<CustomTextInput ref={ele => { this.textInput = ele}} />

// 作用于DOM元素

<input type="text" ref={(input) => {this.input = input}} />

**使用react来获取元素最好是使用回调函数的形式而不是通过字符串**

在react 16版本之前，获取DOM元素都是通过上述方式实现的，这种方式有一个缺点：每次组件重新渲染的时候，行内函数都会执行两次，第一次的ele的值为空，第二次才为真正的DOM对象。

因为在每次渲染中React都会创建一个新的函数实例。因此，React 需要清理旧的 ref 并且设置新的。  
通过将 ref 的回调函数定义成类的绑定函数的方式可以避免上述问题，

**1、新增的API：React.createRef（）**

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

// 第一步：调用API， create a ref to store the textInput DOM element

this.textInput = React.createRef();

this.focusTextInput = this.focusTextInput.bind(this);

}

focusTextInput() {

// Explicitly focus the text input using the raw DOM API

// 第三步：通过current属性获取真实的DOM节点

this.textInput.current.focus();

}

render() {

// tell React that we want to associate the <input> ref

// with the `textInput` that we created in the constructor

return (

<div>

<input

type="text"

ref={this.textInput} /> // 第二步：将创建的textInput与真实的DOM绑定起来

<input

type="button"

value="Focus the text input"

onClick={this.focusTextInput}

/>

</div>

);

}

}

**2、新增的API：React.forwardRef（）**

Forwarding refs中提供了一个React.forwardRef来创建组件，在React.forwardRef的方法中传递了参数ref，通过这个ref可以指向具体的某一个dom节点。具体的指向流程为：

父组件myRef——>React.forwardRef中的实参——>通过forwardRef方法创建的子组件中的ref——>指向子组件中的某一个dom节点。

const Child = React.forwardRef((props,ref)=>( // 第三步：使用forwardRef创建子组件，创建时会在构造方法中接受传递过来的ref作为参数

<input ref={ref} /> // 第四步：这个参数ref可以赋值给子组件中的DOM节点

));

class Father extends React.Component{

constructor(props){

super(props);

this.myRef=React.createRef();// 第一步：在父组件中使用createRef创建一个myRef

}

componentDidMount(){

console.log(this.myRef.current); // 第五步：在父组件中就可以通过current来获取到子组件的DOM节点

}

render(){

return <Child ref={this.myRef}/> // 第二步：通过子组件的ref属性将myRef传递给子组件内部

}

}

八、react高级：React Context

当你不想在组件树中通过逐层传递props或者state的方式来传递数据时，可以使用Context来实现**跨层级**的组件数据传递。

1、context的使用方式

如果要Context发挥作用，需要用到两种组件，一个是Context生产者(Provider)，通常是一个父节点，另外是一个Context的消费者(Consumer)，通常是一个或者多个子节点。所以Context的使用基于**生产者消费者模式**。

对于父组件，也就是Context生产者，需要通过一个静态属性childContextTypes声明提供给子组件的Context对象的属性，并实现一个实例getChildContext方法，返回一个代表Context的纯对象 (plain object) 。

var Provider = React.createClass({

childContextTypes: { // 声明要提供给子组件的对象

auth\_info: React.PropTypes.object,

req: React.PropTypes.object,

fconf: React.PropTypes.object,

helpers: React.PropTypes.object,

env: React.PropTypes.object,

wx\_sdk: React.PropTypes.object

},

getChildContext: function() { //实现一个方法

return context;

},

})

对于子组件需要通过一个静态属性contextTypes声明后，才能访问父组件Context对象的属性，否则，即使属性名没写错，拿到的对象也是undefined。

class TalentTab extends React.Component {

    static contextTypes = {

        auth\_info: PropTypes.object,

        req: PropTypes.object,

        fconf: PropTypes.object,

    };

createNewSid() {

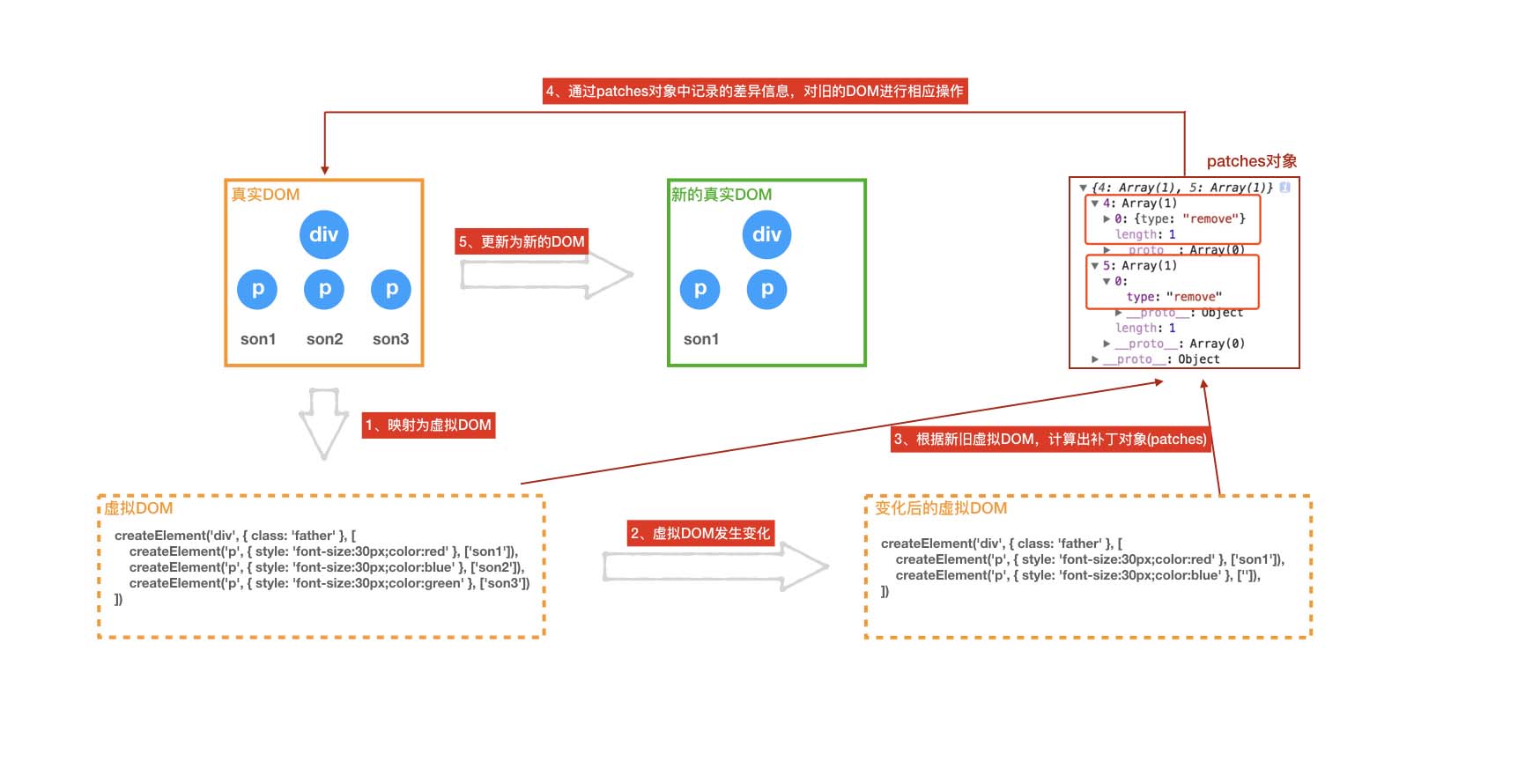
        let { auth\_info } = this.context // 子组件通过this.context访问

        return 's' + new Date().getTime() + auth\_info.u

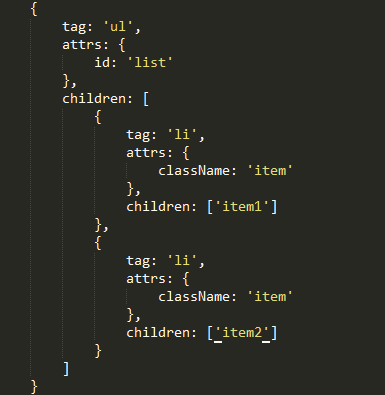
    }

}

九、react的diff算法（简单实现）



首先，需要了解一个虚拟dom树长什么样



嗯，大致就长这样。

在react中，当页面内容发生变化的时候，react不会直接去改变dom节点，而是先根据改变后的页面内容生成一个虚拟DOM树，将新旧两个DOM树进行diff算法，算法的主要思想就是**树的先序深度优先遍历**。在github上看到了一个diff算法的简单实现。在进行diff的过程中，会为每个节点生成一个补丁（patch），所有的补丁都有一个id，表示其在深度遍历中的位置，这样方便更新。补丁其实就是一个对象，对象里面有两个属性，type表示当前节点需要做什么操作，另一个属性的就会根据操作的不同而发生变化。

let patchs = {}

// 当前父节点下的子节点的索引

let globalIndex = 0

function diff(oldTree, newTree) {

walk(oldTree, newTree, globalIndex)

return patchs

}

function isString(node) {

return typeof node === 'string'

}

function walk(oldTree, newTree, index) {

// 每个元素都有一个补丁对象

let currentPatchs = []

// 新节点不存在的情况

if (!newTree) {

currentPatchs.push({

type: TYPES.REMOVE,

index

})

} else if (isString(oldTree)) {

// 新对象的值是文本了 若不相等则记录补丁

if (isString(newTree) && oldTree !== newTree) {

currentPatchs.push({

type: TYPES.TEXT,

text: newTree

})

}

if (!isString(newTree)) {

currentPatchs.push({

type: TYPES.REPLACE,

newNode: newTree

})

}

} else if (oldTree.type === newTree.type) {

let patchAttrs = diffProps(oldTree.props, newTree.props)

if (Object.keys(patchAttrs).length > 0) {

currentPatchs.push({

type: TYPES.ATTRS,

attr: patchAttrs

})

}

diffChildren(oldTree.children, newTree.children)

} else {

// type不一致

currentPatchs.push({

type: TYPES.REPLACE,

newNode: newTree

})

}

// 放到最外层的大补丁包中

if (currentPatchs.length > 0) {

patchs[index] = currentPatchs

}

}

function diffProps(oldProps, newProps) {

// 存放props补丁

let patchProps = {}

for (let key in oldProps) {

if (oldProps[key] !== newProps[key]) {

patchProps[key] = newProps[key]

}

}

for (let key in newProps) {

if (newProps[key] !== oldProps[key]) {

patchProps[key] = newProps[key]

}

}

return patchProps

}

function diffChildren(oldChildrens, newChildrens) {

oldChildrens.forEach((child, idx) => {

walk(child, newChildrens[idx], ++globalIndex)

})

}

export {

diff

}

在得到了两个树比较之后的补丁集合后，我们就可以根据补丁来改变真实的DOM节点。在下面的代码中，传给patch的是真实的DOM元素而不是虚拟DOM节点

let index = 0;// index表示当前处理到哪个patch了

let allPatchs = {};// 用于存放所有的补丁

function patch(node, patchs) {

allPatchs = patchs

walk(node)

}

function walk(node) {

// currentPatch为当前节点的补丁

let currentPatch = allPatchs[index++]

let childNodes = node.childNodes

childNodes.forEach(child => walk(child))

if (currentPatch) {

// 如果有补丁存在说明节点有变化，调用doPatch去更新

doPatch(node, currentPatch)

}

}

function doPatch(node, patchs) {

// patchs的格式为{type: TYPES.REPLACE,newNode: newTree}

patchs.forEach(patch => {

patch2dom(node, patch)

})

}

function patch2dom(node, patch) {

switch (patch.type) {

case TYPES.ATTRS:

for (let key in patch.attr) {

setAttr(node, key, patch.attr[key])

}

break

case TYPES.REMOVE:

node.parentNode.removeChild(node)

break

case TYPES.REPLACE:

let newNode = patch.newNode instanceof Element ?

render(patch.newNode) :

document.createTextNode(patch.newNode)

node.parentNode.replaceChild(newNode, node)

break

case TYPES.TEXT:

node.textContent = patch.text

break

default:

}

}

十、react组件间通信方法（包括父子组件通信、子组件之间的相互通信）

1、React组件间通信方式:

* **父组件向子组件通讯:** 父组件可以向子组件通过传 props 的方式，向子组件进行通讯
* **子组件向父组件通讯:** props+回调的方式,父组件向子组件传递props进行通讯，此props为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中
* **兄弟组件通信:** 找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信
* **跨层级通信:**Context设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言, 对于跨越多层的全局数据通过Context通信再适合不过
* **发布订阅模式:** 发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入event模块进行通信
* **全局状态管理工具:** 借助Redux或者Mobx等全局状态管理工具进行通信,这种工具会维护一个全局状态中心Store,并根据不同的事件产生新的状态

前三中通信方式都没什么好说的，第四种使用context通信在上面讲过了，主要看第五种方法

**2、利用node.js的event模块进行通信**

举个例子来说明：假如现在有个list组件和一个home组件，home组件有一个列表，我们需要实现点击home中的列表的某一项并将这项的内容显示在list中，

（1）首先我们需要引入events模块，并初始化一个eventEmitter对象。注意，在list和home中订阅和发布都要使用同一个eventEmitter对象，否则是不会生效的。

import { EventEmitter } from 'events';

export default new EventEmitter()

（2）然后我们在list组件中监听一个事件，当监听到时就触发回调函数去改变state中的值，一般监听事件的函数都写在componentDidmount中，并在组件即将卸载时取消监听。

class List extends Component {

constructor(props){

super(props)

this.state = {

name: "无"

}

}

componentDidMount(){

// 在list上监听一个事件

this.eventEmitter = emitter.addListener('changeMessage', (message) => {

this.setState({

name:message

});

});

}

componentWillUnmount(){

emitter.removeListener(this.eventEmitter);

}

render(){

return (

<div className='select'>

我选中的是：{this.state.name}

</div>

)

}

}

（3）再然后我们在home组件中为每一个列表项都绑定一个点击事件，事件的回调函数会使用emit方法去触发一个changeMessage事件。

class Home extends Component {

constructor(props){

super(props)

this.state = {

member:["the shy",'ning','rookie','jackeylove','baolan']

}

}

handleClick = (message) => {

emitter.emit('changeMessage', message);

};

render(){

return (

<div>

{

this.state.member.map((item,index)=>{

return <div key={index} className="ig" onClick={() =>{this.handleClick(item)}}>{item}</div>

})

}

</div>

)

}

}

十、react的key属性的用法，与真实应用场景结合可以解决什么问题

# 综合性问题

**一、下面的代码会输出什么？**

functionA(name) {

this.name= name ||'Tom'

this.msg="use 'this.' set in function"

}

functionB() {};

B.prototype=A;

var b =newB();

console.log(b.name);

console.log(b.msg);

答案是：

A

undefined

分析：

b.name返回 A，是因为b上面没有name属性，他就会沿着原型链向上查找，然而 b.\_\_proto\_\_ 为函数A，每一个函数都有一个属性为name，其值是函数的名字。

想要输出TOM和use 'this.' set in function 应该令B的原型为A 的实例。

二、请写出下面问题的答案

functionFoo() {

getName=function() {

console.log(1);

};

returnthis;

}

Foo.getName=function() {

console.log(2);

};

Foo.prototype.getName=function() {

console.log(3);

};

vargetName=function() {

console.log(4);

};

functiongetName() {

console.log(5);

}

//请写出以下输出结果：

Foo.getName(); //-> 2 Foo对象上的getName() ，这里不会是3，因为只有Foo的实例对象才会是3，Foo上面是没有3的

getName(); //-> 4 window上的getName，console.log(5)的那个函数提升后，在console.log(4)的那里被重新赋值

Foo().getName(); //-> 1 在Foo函数中，getName是全局的getName，覆盖后输出 1

getName(); //-> 1 window中getName();

newFoo.getName(); //-> 2 Foo后面不带括号而直接 '.'，那么点的优先级会比new的高，所以把 Foo.getName 作为构造函数

newFoo().getName();//-> 3 此时是Foo的实例，原型上会有输出3这个方法

# 一些细碎的知识点

一、数字字符串比较大小与数字比较大小有什么不同？

例如： "9">=="100" 返回的值为true 而9 >== 100返回的值为false

这是因为**数组字符串在比较大小时是按照字符对应位置一个个比较**

二、将字符串转化成数字时，parseInt与Number的区别？

当遇到不合理的字符时，将空字符转化成数字，这时parseInt（‘ ’）输出为NaN，而Number（‘ ’）输出为0，在实际情况中，很有可能遇到字符串是空的情况，如果使用parseint可能会导致判断错误

三、关于element.style.width

在做进度条的demo时用style.width想获取元素的初始宽度结果发现获取不到，发现其只能获取内联样式设置的宽度，不能获取class定义以及引入的css文件中定义的样式。

如果想获取外联样式设置的宽度可以使用element.offsetWidth。

# 关于项目

一、如何实现前后端鉴权？

**2、使用cookie-session鉴权**

session-cookie认证主要分四步：

1，服务器在接受客户端首次访问时在服务器端创建seesion，然后保存seesion(我们可以将seesion保存在内存中，也可以保存在redis中，推荐使用后者)，然后给这个session生成一个唯一的标识字符串,然后在响应头中种下这个唯一标识字符串。

2.签名。这一步只是对sid进行加密处理，服务端会根据这个secret密钥进行解密。（非必需步骤）

3.浏览器中收到请求响应的时候会解析响应头，然后将sid保存在本地cookie中，浏览器在下次http请求de 请求头中会带上该域名下的cookie信息。

4.服务器在接受客户端请求时会去解析请求头cookie中的sid，然后根据这个sid去找服务器端保存的该客户端的session，然后判断该请求是否合法。

**3、使用token鉴权**

使用基于 Token 的身份验证方法，大概的流程是这样的：

   1. 客户端使用用户名跟密码请求登录

   2. 服务端收到请求，去验证用户名与密码

   3. 验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

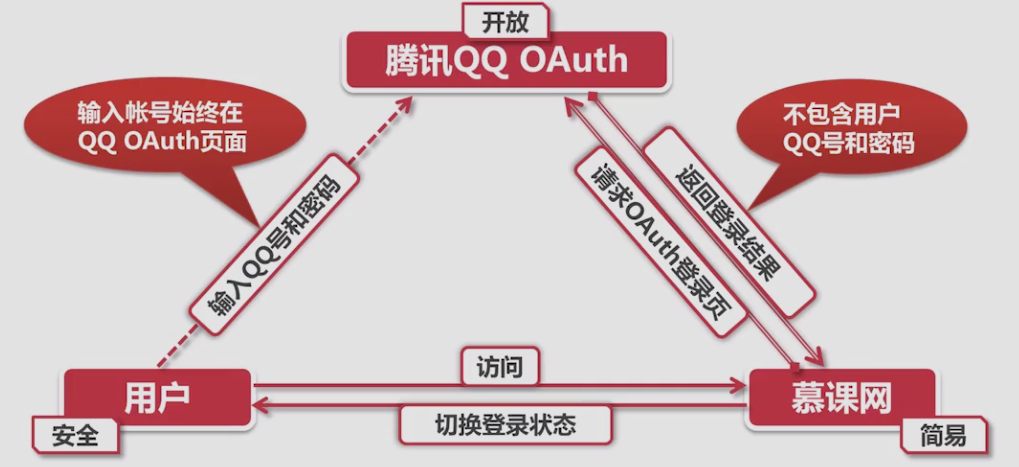
   4. 客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里

   5. 客户端每次向服务端请求资源的时候需要带着服务端签发的 Token

   6. 服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据

使用token的优势：（1）token本身就是一种登陆成功凭证，他是在登陆成功后根据某种规则生成的一种信息凭证，他里面本身就保存着用户的登陆状态。服务器端只需要根据定义的规则校验这个token是否合法就行，如果是使用session的话服务器还需要根据前端传过来的sessionid去查询用户的登录状态.（2）session-cookie是需要cookie配合的，只有浏览器才会去解析请求响应头里面的cookie,然后每次请求再默认带上该域名下的cookie。但是我们知道http代理客户端不只有浏览器，还有原生APP等等，这个时候cookie是不起作用的，token则可以放在http请求的头部，authorization字段里，这样可以避免csrf攻击。

**4、使用oauth鉴权（即第三方平台）**



步骤1：除了慕课网之外还有很多第三方站点在使用QQ登录，每个站点在揭露QQ登录之前都得到QQ互联的官网注册，提交一份申请的材料，说明揭露站点的描述、站点的域名、回调地址等信息，然后QQ互联的官网会给申请使用QQ登录的站点分配一组appid和appkey，类似账号和密码，appid是唯一的，每个申请QQ揭露的站点都有一个唯一的appid，这样，对于QQ登录的服务器来说，可以通过此id来判断当前是哪个站点在请求使用QQ登录，该站点是否在QQ互联平台提交过申请材料等；慕课网在提交申请材料的时候会把回调地址redirect\_uri预留给QQ互联的平台，服务端会对比相应的appid中预留的回调地址和提交过来的参数域名是否一样等

服务器的URL：[https://graph.qq.com/oauth/](https://link.jianshu.com?t=https:/graph.qq.com/oauth/)....?....

对应appid：&client\_id=100490398&....

回调地址：&redirect\_uri=[http://www.mukewang.com/](https://link.jianshu.com?t=http:/www.mukewang.com/).....

步骤2、在用户正确输入QQ号和密码，登录成功的情况下，redirect\_uri告诉QQ的OAuth服务器，当用户登录成功以后，要跳转到什么地方去，并且在跳转的过程中，URL后面会带上一个加密过的参数code（[http://www.mukewang.com/user/qqcallback?code=xxxxxxx](https://link.jianshu.com?t=http:/www.mukewang.com/user/qqcallback?code=xxxxxxx)...），当url跳转回慕课网指定的回调地址之后，在代码中可以获取到code。

步骤3、在步骤二中，慕课网拿到了一个加密过的参数code,此时，说明用户输入的QQ号和密码是匹配的，即登录成功了；但是出于安全考虑，单凭code还不能赋予第三方网站使用用户的QQ号去执行一些操作（因为在回调的过程中，code可能会被劫持），所以，为了确保code被合法的第三方服务器获取到，此时需要慕课网的服务器再次请求一个url地址：

User Authorization URL :用户授权的令牌请求服务地址

即用户在成功登陆qq后，慕课的服务器需要请求的一个带有特定参数的User Authorization URL：[https://xxx.qq.com/oauth/](https://link.jianshu.com?t=https:/xxx.qq.com/oauth/)...&client\_id=100490398&....&client\_secret=xxxxxxxxxx&......&code=xxxxxxxxxxx......

client\_id和client\_secret对应慕课网得到的一组appid和appkey，分别对应这两个参数，把他拼接到url中，最后还需要拼接上用户授权登录之后拿到的code参数（code是会过期的，有效时间都比较短，一般为10秒）  
code是一个有生命周期且只可使用一次的字符串，慕课网的服务器在很短的时间内就要完成这个一来一回的数据交互过程，一旦超过这个code的有效时间，OAuth的服务器就会认为登录的过程中可能出现了一些安全问题，然后拒绝第三方网站继续后面的操作。当慕课网的服务器请求了User Authorization URL之后，才会得到一个响应数据，响应数据中包含了一些信息：用户具体是谁，QQ昵称、头像等一些比较基本的账号信息，慕课网的服务器就可以把这些信息写入到用户数据表中，此时，code也失效了。

二、页面加载海量数据，比如10w条记录一次性渲染到屏幕上，如何处理不冻结UI？

<ul id="list-with-big-data">100000 数据</ul>

<script>

// 此处添加你的代码逻辑

</script>

// 解决方案如下:

主要涉及到的知识点有减少页面重排重绘，js的动画函数requestAnimationFrame，以及事件委托

<script>

const ulContainer = document.getElementById("list-with-big-data");

const total = 100000; // 插入数据的总数

const batchSize = 4; // 每次批量插入的节点个数，个数越多，界面越卡顿

const batchCount = total / batchSize; // 批处理的次数

let batchDone = 0; // 已完成的批处理个数

function appendItems() {

// 使用 DocumentFragment 减少 DOM 操作次数，对已有元素不进行回流

const fragment = document.createDocumentFragment();

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

const liItem = document.createElement("li");

liItem.innerText = batchDone \* batchSize + i + 1;

fragment.appendChild(liItem);

}

// 每次批处理只修改 1 次 DOM

ulContainer.appendChild(fragment);

batchDone++;

doAppendBatch();

}

function doAppendBatch() {

if (batchDone < batchCount) {

// 在重绘之前，分批插入新节点

window.requestAnimationFrame(appendItems);

}

}

// kickoff

doAppendBatch();

// 使用 事件委托 ，利用 JavaScript 的事件机制，实现对海量元素的监听，有效减少事件注册的数量

ulContainer.addEventListener("click", function(e) {

const target = e.target;

if (target.tagName === "LI") {

alert(target.innerText);

}

});

</script>

三、前端优化如何实现，或者说你在项目中做了哪些优化

**1、关于加载优化**

（1）在客服平台上，因为整个项目是个单页面引用，为了避免第一次进入系统时白屏的情况，可以使用webpack插件来在白屏期间插入loading图。

plugins: [

// Generates an `index.html` file with the <script> injected.

new HtmlWebpackPlugin({

inject: true,

template: paths.appHtml, // appHtml对应一个白屏期间显示的页面

}),

]

然后当页面完全加载了之后通过render方法的第三个参数可以将appHtml页面隐藏

ReactDOM.render(

<LocaleProvider locale={zn\_CN}>

<Router history={historyDriver}></Router>

</LocaleProvider>

, document.getElementById('root'), () => {

// At this point, page is fully loaded, remove loading screen

$('#app-loading-screen').animatedHide('fadeOut', 600);

}

);

**（2）使用服务器端渲染，即SSR**

脉脉app端用的就是服务器端渲染，使用jade模板加上Koa框架，当请求某个页面时匹配到页面对应的路由，在node层先请求一些页面需要的数据，然后将这些数据和html页面一起返回。数据会通过Jade模板渲染到页面的指定位置。

四、面试被问到项目难点该怎么说？

（1）首先第一点，在接手招聘脉脉的时候不知道路由配置的具体情况，直接输入IP:3033/search提示找不到页面，后来去看node的启动文件发现有一行代码是

require('./middleware/router')(app);

感觉很像配置路由的文件，然后就去看了一下这个router文件的内容，里面有一行这样的代码

require('./static')(app, path.join(baseDir, "/routes"), projectRootUrlPath);

再去static文件里看一下，

module.exports = (app, dir, projectRootUrlPath) => {

app.router = router;

glob(`${dir}/\*\*/\*.js`, (err, files) => {

if (!err) {

files.forEach(v => require(v)(app));

}

});

app.use(koaBody({ multipart: true }))

.use(mount(projectRootUrlPath, router.routes()))

.use(router.allowedMethods());

};

static导出了一个函数，这个函数接受了三个参数，第一个app就是node 启动文件job.js内创建的koa实例，第二参数是一个路径，这里这个路径的指向的是route文件夹，因为所有的路由都添加在route下，而第三个参数就是‘/bizjobs’，通过mount中间件在所有的路由前面都拼接上了/bizjobs

（2）在做大搜小搜改版的项目的时候，需求要求是实现一个页面，页面分成两个tab，其中tab2是需要实现的，tab1是一个现有的页面，但是这个页面在脉脉的主工程里，这就涉及到了跨工程引用页面，但是当时想的是重新实现一个老页面成本太高，而且内部逻辑也没人知道了，所以很困难，最后决定还是跨工程引用页面。

首先就遇到了一个问题，contacts页面以及其引用的页面包含了很多命名别名，而在本工程里并没有为这些别名配置对应的路径，所以我在webpack的resolve里配置了alias，这样打包才不会报错。

react版本也出现了问题，在打包时出现过下面这样的错误，



因为contacts原先所在的工程react的版本是15.6，而目前招聘工程的版本是16.8，contacts在原工程上运行没有问题，但是在现在的版本上就会报上面的错误，网上说的原因是因为react工程的版本和工程内引入的插件版本不同，而且contacts页面使用的语法已经不被react16版本所支持，比如在react16中已经没有了React.protoType，createClass属性并且contacts页面使用了很多这样的语法，如果找不到别的解决方案就只能一个个将这些写法都改过来，

我的第一反应是给招聘的工程使用的react降级，但是被我leader给否了。

解决方法也是在alias上增加下面的代码，然后把contact以及其引用的组件的语法全部替换成react16用到的语法。

resolve: {

alias: {

"react": path.resolve(\_\_dirname, './node\_modules/react'),

"react-dom": path.resolve(\_\_dirname, './node\_modules/react-dom'),

"radium": path.resolve(\_\_dirname, './node\_modules/radium'),

}

},

（3）继续踩坑，在项目能正常启动了之后调用接口以及鉴权出了问题，首先，contacts调用的接口都是部署在3000端口上的，而现在我们是在3033端口上运行这个页面并且页面发请求去调用接口都是去找当前域名下启动的接口，

主要是因为招聘node跑在3033端口，而主node跑在3000端口，从而将contacts引入到招聘项目之后contacts页面的接口全部不好用，报跨域，而且鉴权的时候也出现问题，这是因为我在3033端口下登录，所有的登录状态token什么的都记录在3033端口下的cookie里，当我访问contacts页面的时候，由于其运行在3000端口，而且contacts页面的一些接口调用时需要鉴权，所以node鉴权时候就去3000端口下找用户的登录状态结果找不到，所以页面就访问失败。解决的方案就是在脉脉主node的node启动文件app.js中去配置一下cors，如下所示

var options = {

origin: 'http://jobs.taou.com:3033',

credentials: true

}

app.use(cors(options))

并且在开发时，鉴权文件inapp.js里引用的config文件，config由引用了localconf文件，将localconf文件中的develop\_mode=true 和 default\_uid都写死，上线之前再都删掉。

我们请求path的写法： /aaa/bbb/ccc。因为开发环境两个工程的域名不同，bizjob 引 maimai\_node 组件里的跳转有问题，job.js: 对 biznode 找不到的路由重定向到3000端口，先去判断当前开发环境，并且得是在3033端口请求不到的数据，即返回404了才去重定向到3000端口。



（4）招聘node开发后上不了灰度，上灰度把灰度环境上挂了，在灰度环境服务启动失败，原因如下：

国新帮定位到是因为biz 的 node 的找不到@maimai 等符号链接

原因：

· 之前在 webpack 添加的 alias 只是使打包的页面能找到自定义别名，而不能使招聘node找到自定义别名

· 在本地能启动而上灰度后有问题：是因为 maimai\_node 在 gulp build 的时候会遍历 webpack 里的 alias 在maimai\_node/node\_modules里生成对应的符号链接。本地环境启动过maimai\_node，而 biznode 的灰度环境没有起过maimai\_node工程。

解决：887782168

· 参照 maimai\_node，在 bizjob 的gulp build 添加 create-maimai-link，每次打包时生成、检查符号链接

gulp.task('create-maimai-link', function(){

// path.resolve 作用：把一个路径或路径片段的序列解析为一个绝对路径

const node\_modules = path.resolve(\_\_dirname, '../../node\_modules');

const { resolve: { alias = {} } = {} } = webpackConfig || {};

Object.keys(alias).filter(key => key.indexOf('@maimai') == 0).forEach((key) => {

const destPath = alias[key];

// path.join() 作用：将路径片段使用特定的分隔符（"/"）连接起来形成路径

const maimaiSymlink = path.join(node\_modules, key);

// fs.existsSync 作用：判断路径是否存在，如果存在返回true

if (!fs.existsSync(destPath)) {

console.log('destPath: ' + destPath + ' nonexistent');

exitForJenkins(); // 执行process.exit(1);

return;

}

// 先删除老的符号链接， 防止webpack\_common\_config.resolve.alias 修改后symbol link没有指向新的地址

try {

fs.unlinkSync(maimaiSymlink);

} catch(e) {}

// fs.symlinkSync 作用：用于创建符号链接。第一个参数为为源目录或文件的路径，第二个参数为存放转换后的目录的路径，

fs.symlinkSync(path.relative(node\_modules, destPath), maimaiSymlink, 'dir'); // 无返回值

// 二次检查

if (!fs.existsSync(maimaiSymlink)) {

console.log('create: ' + key + ' failed');

exitForJenkins();

}

});

return true;

});

感觉可能会被问硬链接和符号链接（软连接）的区别：

首先要知道一下索引节点，在Linux系统中，内核为每一个新创建的文件分配一个Inode(索引结点)，每个文件都有一个惟一的inode号，我们可以将inode简单理解成一个指针，它永远指向本文件的具体存储位置。

**硬链接：**说白了就是一个指针，指向文件索引节点，系统不会为硬链接分配新的索引节点，相当于是两个指针指向同一个文件内容。在linux上创建硬链接的命令是ln 1.txt 2.txt

[root@centos7 home]# vi 1.txt

hello, this is 1.txt!

[root@centos7 home]# ln 1.txt 2.txt // 为1.txt建立硬链接2.txt，等同于ln 1.txt 2.txt

[root@centos7 home]# more 2.txt // 查看2.txt文件中的内容和1.txt文件内容一样

hello, this is 1.txt!

[root@centos7 home]# ls -li // 查看文件 可以看到1.txt和2.txt内容大小都是2B且索引号相同

总用量 69868

33845219 -rw-r--r--. 2 root root 44 1月 21 10:12 1.txt

33845219 -rw-r--r--. 2 root root 44 1月 21 10:12 2.txt

如果修改了创建的硬链接的内容，那么源文件的内容也会被修改

**软连接：为文件创建一个软连接的话，系统会为这个软连接重新分配索引节点。软连接这个文件包含了另一个文件的路径名，可以是任意文件或目录，可以链接不同文件系统的文件。**

[root@centos7 home]# ln -s 2.txt sLink //为2.txt文件建立符号链接sLink

[root@centos7 home]# ls –li //可以看到两个文件有不同的索引节点号

总用量 69864

33845219 -rw-r--r--. 1 root root 44 1月 21 10:12 2.txt

36830246 lrwxrwxrwx. 1 root root 5 1月 21 10:21 sLink -> 2.txt