# 面试题

# 注册登录是怎么实现的   多用hoc。多用自定义hooks。

1.登陆注册要做成受控组件，组件定义state，和表单绑定  
   
2.redux-saga调用数据请求，发送action修改数据，  
 useEffect中dispatch发送数据请求，后端比对用户名是否重复，返回state  
  
3.前端根据返回的信息成功跳转登陆页  
  
4.登陆发送数据请求，数据库对比用户名密码是否正确, 根据后端返回的结果进入首页  
  
5.setCookie将用户登录名密码token存cookie中 通过JWT(Json web token)  
  
6.免密登陆 getCookie获取token 发给后端对比 根据返回结果是否自动登陆  
  
7.注册通过Ant Design ，validator中进行表单正则的验证  
  
8.用户体验 注册的时候跳转其他页面的时候给用户提示是否需要跳转，避免因为跳转后导致注册信息没有了   
 用组件内后置守卫做  
 如果输入框都没有填信息,不拦截跳转如果用户输入信息,弹窗提示,点确定,跳转,点取消,不跳转

# 后台管理系统和普通App面向用户的区别

toB和toC的项目  
  
面向企业内部和面向用户的项目的区别  
  
后台管理系统权限比较细 App高并发比较多 做性能优化

# 正则表达式

\d 匹配数字 \D 匹配非数字 \w 匹配数字字母下划线 \W 匹配非数字字母下划线   
\n 匹配一个换行符 \s 匹配任何不可见字符包括空格、制表符、换页符等等 \S 匹配任何可见字符  
^ 匹配输入字行首 $匹配输入行尾 \*(0到多次)匹配前面的子表达式任意次   
+(1到多) 匹配前面的子表达式一次或多次(大于等于1次）  
?(0或1)匹配前面的子表达式零次或一次   
{n}n是一个非负整数，匹配确定的n次  
{n,}n是一个非负整数。至少匹配n次

# Ajax

我对 ajax 的理解是，它是一种异步通信的方法，通过直接由 js 脚本向服务器发起 http 通信，然后根据服务器返回的数据，更新网页的相应部分，而不用刷新整个页面的一种方法。  
创建步骤：创建xhr对象->配置Ajax请求地址通过open方法->发送请求通过send方法->监听请求，接收响应  
//1：创建Ajax对象  
var xhr = window.XMLHttpRequest?new XMLHttpRequest():new ActiveXObject('Microsoft.XMLHTTP');// 兼容IE6及以下版本  
//2：配置 Ajax请求地址  
xhr.open('get','index.xml',true);  
//3：发送请求  
xhr.send(null); // 严谨写法  
//4:监听请求，接受响应  
xhr.onreadysatechange=function(){  
 if(xhr.readySate==4&&xhr.status==200 || xhr.status==304)  
 console.log(xhr.responseXML)  
}

# js 延迟加载的方式

js 的加载、解析和执行会阻塞页面的渲染过程，因此我们希望 js 脚本能够尽可能的延迟加载，提高页面的渲染速度。  
我了解到的几种方式是：  
  
1、将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。  
2、给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。  
3、给 js 脚本添加 async属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。  
4、动态创建 DOM 标签的方式，我们可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。

# 模块化开发的理解

我对模块的理解是，一个模块是实现一个特定功能的一组方法。在最开始的时候，js 只实现一些简单的功能，  
所以并没有模块的概念，但随着程序越来越复杂，代码的模块化开发变得越来越重要。  
  
由于函数具有独立作用域的特点，最原始的写法是使用函数来作为模块，几个函数作为一个模块，  
但是这种方式容易造成全局变量的污染，并且模块间没有联系。  
  
后面提出了对象写法，通过将函数作为一个对象的方法来实现，这样解决了直接使用函数作为模块的一些缺点，  
但是这种办法会暴露所有的所有的模块成员，外部代码可以修改内部属性的值。  
  
现在最常用的是立即执行函数的写法，通过利用闭包来实现模块私有作用域的建立，同时不会对全局作用域造成污染。

# js 的几种模块规范

js 中现在比较成熟的有四种模块加载方案：  
  
第一种是 CommonJS 方案，它通过 require 来引入模块，通过 module.exports 定义模块的输出接口。这种模块加载方案是服务器端的解决方案，它是以同步的方式来引入模块的，因为在服务端文件都存储在本地磁盘，所以读取非常快，所以以同步的方式加载没有问题。但如果是在浏览器端，由于模块的加载是使用网络请求，因此使用异步加载的方式更加合适。  
  
第二种是 AMD 方案，这种方案采用异步加载的方式来加载模块，模块的加载不影响后面语句的执行，所有依赖这个模块的语句都定义在一个回调函数里，等到加载完成后再执行回调函数。require.js 实现了 AMD 规范。  
  
第三种是 CMD 方案，这种方案和 AMD 方案都是为了解决异步模块加载的问题，sea.js 实现了 CMD 规范。它和require.js的区别在于模块定义时对依赖的处理不同和对依赖模块的执行时机的处理不同。  
  
第四种方案是 ES6 提出的方案，使用 import 和 export 的形式来导入导出模块。

# AMD和CMD规范的区别？

它们之间的主要区别有两个方面。  
第一个方面是在模块定义时对依赖的处理不同。AMD推崇依赖前置，  
在定义模块的时候就要声明其依赖的模块。而 CMD 推崇就近依赖，  
只有在用到某个模块的时候再去 require。  
  
第二个方面是对依赖模块的执行时机处理不同。  
首先 AMD 和 CMD 对于模块的加载方式都是异步加载，  
不过它们的区别在于模块的执行时机，  
AMD 在依赖模块加载完成后就直接执行依赖模块，  
依赖模块的执行顺序和我们书写的顺序不一定一致。  
而 CMD在依赖模块加载完成后并不执行，只是下载而已，  
等到所有的依赖模块都加载好后，进入回调函数逻辑，遇到 require 语句  
的时候才执行对应的模块，这样模块的执行顺序就和我们书写的顺序保持一致了。

# ES6模块与CommonJS 模块、AMD、CMD的差异

1.CommonJS 模块输出的是一个值的拷贝，  
ES6 模块输出的是值的引用。  
CommonJS 模块输出的是值的，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。  
ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令 import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。  
2.CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。CommonJS 模块就是对象，即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。  
而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

# requireJS的核心原理

require.js 的核心原理是通过动态创建 script 脚本来异步引入模块，然后对每个脚本的 load 事件进行监听，如果每个脚本都加载完成了，再调用回调函数

# js的原理(运行机制)

首先js是单线程运行的，在代码执行的时候，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。  
在执行同步代码的时候，如果遇到了异步事件，js引擎并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务  
  
所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。  
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，只有等主线程任务执行完毕，"任务队列"开始通知主线程，请求执行任务，该任务才会进入主线程执行。  
  
当同步事件执行完毕后，再将异步事件对应的回调加入到与当前执行栈中不同的另一个任务队列中等待执行。  
任务队列可以分为宏任务对列和微任务对列，当当前执行栈中的事件执行完毕后，js 引擎首先会判断微任务对列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。  
当微任务对列中的任务都执行完成后再去判断宏任务对列中的任务。  
  
异步运行机制如下：  
（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。  
（2）主线程之外，还存在一个"任务队列"（task queue）。  
 只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。  
（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。  
 那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。  
（4）主线程不断重复上面的第三步。

# arguments 的对象

arguments对象是函数中传递的参数值的集合。它是一个类似数组的对象，因为它有一个length属性，  
我们可以使用数组索引表示法arguments[1]来访问单个值，但它没有数组中的内置方法，  
如：forEach、reduce、filter和map。  
  
我们可以使用Array.prototype.slice将arguments对象转换成一个数组。  
  
Array.prototype.slice.call(arguments)

# V8 引擎的垃圾回收机制

v8的垃圾回收机制基于分代回收机制，这个机制又基于世代假说，这个假说有两个特点，一是新生的对象容易早死，另一个是不死的对象会活得更久。基于这个假说，v8 引擎将内存分为了新生代和老生代。  
新创建的对象或者只经历过一次的垃圾回收的对象被称为新生代。经历过多次垃圾回收的对象被称为老生代。  
新生代被分为 From 和 To 两个空间，To 一般是闲置的。当 From 空间满了的时候会执行 Scavenge(斯盖V橘) 算法进行垃圾回收。当我们执行垃圾回收算法的时候应用逻辑将会停止，等垃圾回收结束后再继续执行。这个算法分为三步：  
（1）首先检查 From 空间的存活对象，如果对象存活则判断对象是否满足晋升到老生代的条件，如果满足条件则晋升到老生代。如果不满足条件则移动 To 空间。  
（2）如果对象不存活，则释放对象的空间。  
（3）最后将 From 空间和 To 空间角色进行交换。  
新生代对象晋升到老生代有两个条件：  
（1）第一个是判断是对象否已经经过一次 Scavenge 回收。若经历过，则将对象从 From 空间复制到老生代中；若没有经历，则复制到 To 空间。  
（2）第二个是 To 空间的内存使用占比是否超过限制。当对象从 From 空间复制到 To 空间时，若 To 空间使用超过 25%，则对象直接晋升到老生代中。设置 25% 的原因主要是因为算法结束后，两个空间结束后会交换位置，如果 To 空间的内存太小，会影响后续的内存分配。  
  
老生代采用了标记清除法和标记压缩法。  
标记清除法首先会对内存中存活的对象进行标记，标记结束后清除掉那些没有标记的对象。由于标记清除后会造成很多的内存碎片，不便于后面的内存分配。所以了解决内存碎片的问题引入了标记压缩法。  
由于在进行垃圾回收的时候会暂停应用的逻辑，对于新生代方法由于内存小，每次停顿的时间不会太长，但对于老生代来说每次垃圾回收的时间长，停顿会造成很大的影响。 为了解决这个问题 V8 引入了增量标记的方法，将一次停顿进行的过程分为了多步，每次执行完一小步就让运行逻辑执行一会，就这样交替运行

# 垃圾回收机制的两种方法

现在各大浏览器通常用采用的垃圾回收有两种方法：标记清除、引用计数。  
  
1、标记清除  
　　这是javascript中最常用的垃圾回收方式。当变量进入执行环境是，就标记这个变量为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到他们。当变量离开环境时，则将其标记为“离开环境”。  
　　垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。  
  
2、引用计数  
　　另一种不太常见的垃圾回收策略是引用计数。引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变成0时，则说明没有办法再访问这个值了，因而就可以将其所占的内存空间给收回来。这样，垃圾收集器下次再运行时，它就会释放那些引用次数为0的值所占的内存。

# 会造成内存泄漏的操作

1.意外的全局变量  
2.被遗忘的计时器或回调函数  
3.脱离 DOM 的引用  
4.闭包  
  
第一种情况是我们由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。  
第二种情况是我们设置了setInterval定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。  
第三种情况是我们获取一个DOM元素的引用，而后面这个元素被删除，由于我们一直保留了对这个元素的引用，所以它也无法被回收。  
第四种情况是不合理的使用闭包，从而导致某些变量一直被留在内存当中。

# ECMAScript(ES)

ECMAScript 是编写脚本语言的标准，ECMA（European Computer Manufacturers Association）规定一下他的标准，因为当时有java语言了，又想强调这个东西是让ECMA这个人定的规则，所以就这样一个神奇的东西诞生了，这个东西的名称就叫做ECMAScript。  
javaScript = ECMAScript + DOM + BOM

# ES2015(ES6)新特性

块作用域  
类  
箭头函数  
模板字符串  
对象解构  
Promise  
模块  
Symbol  
代理（proxy）Set  
函数默认参数  
rest  
扩展运算符  
数组和对象的扩展

# ES2016(ES7)新特性

求幂运算符（\*\*）  
 (因颗录此)  
Array.prototype.includes()方法  
 数组原型的方法,查找一个数值是否在数组中，只能判断一些简单类型的数据，对于复杂类型的数据无法判断。  
 该方法接受两个参数，分别是查询的数据和初始的查询索引值。

# ES2017(ES8)新特性

async await  
  
函数参数列表结尾允许逗号  
 es2017允许函数对象的定义调用时参数可以加入尾逗号，以及json对象array对象都允许  
   
Object.values()  
 values: [obj]，返回obj自身可枚举属性的属性值的集合  
 (安吹斯)  
Object.entries()  
 entries：[obj], 与values类似，返回的一个2元素的数组  
   
Object.getOwnPropertyDescriptors()  
 getOwnpropertyDescriptors: [obj]，返回obj对象的属性描述符  
 (get 哦 泼破踢 迪斯亏不踢斯)  
String padding:   
 padStart()和padEnd()，填充字符串达到当前长度  
 在字符串首位开始添加string直到满足length为止并返回新的字符串  
   
ShareArrayBuffer和Atomics对象，用于从共享内存位置读取和写入  
(夏尔 啊锐 八法儿) (啊偷没此)

# ES2018(ES9)新特性

异步迭代  
Promise.finally()  
 (饭的嘞)  
Rest/Spread 属性  
(锐斯特)(斯破锐的)  
正则表达式命名捕获组（Regular Expression Named Capture Groups）  
正则表达式反向断言（lookbehind）  
正则表达式dotAll模式   
正则表达式 Unicode 转义   
 (右内扣的)  
非转义序列的模板字符串

# ES2019(ES10)新特性

行分隔符（U + 2028）和段分隔符（U + 2029）符号现在允许在字符串文字中，与JSON匹配  
更加友好的JSON.stringify  
新增了Array的flat()方法和flatMap()方法  
新增了String的trimStart()方法和trimEnd()方法  
Object.fromEntries()  
Symbol.prototype.description  
String.prototype.matchAll  
Function.prototype.toString()现在返回精确字符，包括空格和注释  
简化try{} catch{},修改catch绑定  
新的基本数据类型BigInt  
globalThis  
import()  
Legacy RegEx  
私有的实例方法和访问器

# ES2020(ES11)新特性

私有变量  
 ES11在类中新增私有变量控制符#，在内部变量或者函数前添加一个hash符号#，  
 可以将它们设置为私有属性，只能在类的内部可以使用。  
空值合并运算符  
 空值合并操作符就是 ?? :如果左侧的值为null或者undefined就返回左侧的值，如果没有就返回右侧的值  
可选链操作符  
 可选链操作符 (?.) :如果左侧表达式有值，就会继续访问右侧的字段  
BigInt  
 使用BigInt的方式有两种：  
 1.在数字后面加n  
 2.使用BigInt函数  
动态导入  
  
globalThis  
 提供一种标准化的方式去访问全局对象，可以在任意上下文中获取全局对象自身，并且不用担心环境的问题  
Promise.all缺陷与Promise.allSettled  
 promise.all可以并发执行异步任务,如果其中某个任务执行出现了异常，所有任务都会over，Promise会直接进入reject状态  
 使用Promise.allSettled，它会创建一个新的promise，在所有promise完成后返回一个包含每个promise结果的数组

# var,let和const的区别

1.var声明的变量会挂载在window上，而let和const声明的变量不会：  
2.var声明变量存在变量提升，let和const不存在变量提升  
3.let和const声明形成块作用域  
4.同一作用域下let和const不能声明同名变量，而var可以  
  
const   
 一旦声明必须赋值,不能使用null占位。  
 声明后不能再修改  
 如果声明的是复合类型数据，可以修改其属性  
  
暂存死区  
var a = 100;  
if(1){  
 a = 10;  
 //在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a，  
 // 而这时，还未到声明时候，所以控制台Error:a is not defined  
 let a = 1;  
}

# var和let在for循环的一些不同表现

在var中执行的时候：   
 因为var是没有块级作用域的，所以在for循环中声明的i会存在于test()函数作用域中。  
 每一次for循环就会声明一次i，但后面声明的变量会覆盖前面声明的变量。  
在let中执行的时候：  
 因为块级作用域的原因，let声明的i都会存在于for块级作用域中，每一次for循环都会生成一个块级作用域。

# 箭头函数

箭头函数表达式的语法比函数表达式更简洁  
  
箭头函数没有自己的this值。箭头函数里的this指向的是父级的this.   
  
箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。  
  
在箭头函数版本中，当只有一个表达式或值需要返回，我们只需要()括号，  
不需要 return 语句，箭头函数就会有一个隐式的返回。  
  
如果我们在一个箭头函数中有一个参数，则可以省略括号。  
  
箭头函数不能访问arguments对象。所以调用第一个getArgs函数会抛出一个错误。  
相反，我们可以使用rest参数来获得在箭头函数中传递的所有参数。

# 模板字符串

模板字符串是在 JS 中创建字符串的一种新方法。我们可以通过使用反引号使模板字符串化。  
  
在ES5版本中，我们需要添加\n以在字符串中添加新行。在模板字符串中，我们不需要这样做。  
你说呢  
在 ES5 版本中，如果需要在字符串中添加表达式或值，则需要使用`+`运算符。  
在模板字符串s中，我们可以使用${expr}嵌入一个表达式，这使其比 ES5 版本更整洁。

# 对象解构

对象解构是从对象或数组中获取或提取值的一种新的、更简洁的方法  
我们还可以为属性取别名  
let { firstName: fName, position } = employee;  
当然如果属性值为 undefined 时，我们还可以指定默认值  
let { firstName = "Mark" } = employee;

# Set及应用场景

一、创建Set对象实例  
Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用  
1.构造函数 (伊特波)  
 语法：new Set([iterable])  
 参数：iterable 如果传递一个可迭代对象，它的所有元素将被添加到新的Set中；  
 如果不指定此参数或其值为null，则新的 Set为空  
  
二、Set实例属性  
 size属性将会返回Set对象中元素的个数  
  
三、Set实例方法  
1.add() 方法用来向一个 Set 对象的末尾添加一个指定的值  
 语法：mySet.add(value)  
 参数：value 必需，需要添加到 Set 对象的元素的值  
2.delete() 方法可以从一个 Set 对象中删除指定的元素  
 语法：mySet.delete(value)  
 参数：value 将要删除的元素  
 返回值：成功删除返回 true，否则返回 false  
3.clear() 方法用来清空一个 Set 对象中的所有元素  
 语法：mySet.clear()  
4.has() 方法返回一个布尔值来指示对应的值value是否存在Set对象中  
 语法：mySet.has(value)  
 参数：value 必须，是否存在于Set的值  
 返回值：如果指定的值（value）存在于Set对象当中，返回true; 否则返回 false  
5.entries() (安吹斯)  
 语法：mySet.entries()  
 返回值：一个新的包含 [value, value] 形式的数组迭代器对象，value 是给定集合中的每个元素，  
 迭代器 对象元素的顺序即集合对象中元素插入的顺序  
6.values()  
 语法：mySet.values() 或者 mySet.keys()  
 返回值：返回一个 Iterator(因特瑞特) 对象，这个对象以插入Set对象的顺序包含了原 Set 对象里的每个元素  
7.forEach()  
 语法：mySet.forEach(callback[, thisArg]) (this傲歌)  
 参数：callback 每个元素都会执行的函数  
 thisArg 当执行callback函数时候，可以当作this来使用  
5、什么是WeakSet()？  
和Set结构类似，也是不重复的值的集合，但WeakSet的成员只能是对象(null 除外)。  
而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。  
  
应用场景:  
1、简单数组去重  
2、JSON数组去重  
 JSON数组是比较常见的一种数据结构，形如[{…},…{…}]  
 假如你需要统计某个属性中不同的值。  
 step1:先使用.map将JSON数组变成简单数组，然后用set执行去重  
 step2: 由于生成的Set属于可迭代对象，所以可以使用数组解构符进行解构  
3、二维数组去重  
 我们可以将Set用作存储每项的唯一值，结合reduce进行对比，得出无重复的项目。  
 当然，上面的代码缺点还是不少的。因为只是简单地将其转变成字符串作为对比的键，  
 所以不能区分[1,2]、[‘1’ ,‘2’]、[‘1,2’]等子数组。  
4、数组之间的对比  
 Set的特性不单单可以可以用于单数组，对于数组之间的比较也是十分在行。  
  
什么是WeakSet()？  
 和Set结构类似，也是不重复的值的集合，但WeakSet的成员只能是对象。  
  
WeakSet的API：  
 add() //增  
 delete() //删  
 has() //是否存在  
   
为什么WeakSet不可遍历？  
 因为WeakSet的成员都是弱引用，随时可能消失，成员是不稳定的。  
  
WeakSet的用处：  
 使用ws储存DOM节点，就不用担心节点从文档移除时，会引发内存泄漏(即在被移除的节点上绑定的click等事件)。

# Proxy

Proxy 可以理解成，在目标对象之前架设一层“拦截”，  
外界对该对象的访问，都必须先通过这层拦截，  
因此提供了一种机制，可以对外界的访问进行过滤和改写。  
Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”

# 函数式编程

函数式编程（通常缩写为FP）是通过编写纯函数，避免共享状态、可变数据、副作用 来构建软件的过程。数式编程是声明式 的而不是命令式 的，应用程序的状态是通过纯函数流动的。与面向对象编程形成对比，面向对象中应用程序的状态通常与对象中的方法共享和共处。  
函数式编程是一种编程范式 ，这意味着它是一种基于一些基本的定义原则（如上所列）思考软件构建的方式。当然，编程范式的其他示例也包括面向对象编程和过程编程。  
函数式的代码往往比命令式或面向对象的代码更简洁，更可预测，更容易测试

# 高阶函数

首先高阶函数肯定是函数，不同的是输入的参数和返回的值这两项中的一项必须是函数才能叫高阶函数。  
这个问题在回答的时候可以稍微拓展一下，介绍一下常用的的高阶函数，  
比如：map、flatMap、filter、reduce、fold。

# 为什么函数被称为一等公民

在JavaScript中，函数不仅拥有一切传统函数的使用方式（声明和调用），而且可以做到像简单值一样:  
  
赋值（var func = function(){}）、  
传参(function func(x,callback){callback();})、  
返回(function(){return function(){}})，  
  
这样的函数也称之为第一级函数（First-class Function）。不仅如此，JavaScript中的函数还充当了类的构造函数的作用，同时又是一个Function类的实例(instance)(因斯疼斯)。这样的多重身份让JavaScript的函数变得非常重要。

# . new操作符的实现

1、创建一个空对象 {}  
2、将构造函数中的this指向新创建的对象 (原型链)  
 obj.\_\_proto\_\_ = Dog.prototype // 设置原型链  
3、链接该对象到另一个对象 \_\_proto\_\_  
4、如果该函数没有返回对象，则返回this

# 回调函数？回调函数有什么缺点

回调函数是一个匿名函数，它作为一个参数传递给其他的代码，其作用是在需要的时候方便调用这段（回调函数）代码。可以让异步代码同步执行。  
回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell）

# instanceof的原理

instanceof 可以正确的判断对象的类型，因为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。  
实现 instanceof：  
  
首先获取类型的原型  
然后获得对象的原型  
然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为 null，因为原型链最终为 null  
  
function myInstanceof(left, right) {  
 let prototype = right.prototype  
 left = left.\_\_proto\_\_  
 while (true) {  
 if (left === null || left === undefined)  
 return false  
 if (prototype === left)  
 return true  
 left = left.\_\_proto\_\_  
 }  
}

# 设计模式

在[前端](https://www.fly63.com/)开发中，设计模式是一种被广泛应用的思想。设计模式可以帮助开发者解决常见的问题，并提供可重用的解决方案。本文将会介绍前端常见的设计模式，并通过[代码](https://www.fly63.com/tag/%E4%BB%A3%E7%A0%81)详解它们的实现。

**单例模式 观察者模式 工厂模式 装饰者模式 代理模式 适配器模式 命令模式 观察者模式**

#### 一、单例模式

单例模式是指一个类只能被实例化一次，并提供全局访问点。这个模式非常适合那些需要共享资源的场景。比如，在前端开发中，我们经常需要确保某些资源只被加载一次，而不是每次都重新加载。下面是一个使用单例模式的例子：

class Singleton {  
 constructor() {  
 if (typeof Singleton.instance === 'object') {  
 return Singleton.instance;  
 }  
 this.name = 'Singleton';  
 Singleton.instance = this;  
 return this;  
 }  
}  
   
const instance1 = new Singleton();  
const instance2 = new Singleton();  
   
console.log(instance1 === instance2); // true

上述代码中，我们创建了一个 Singleton 类，该类只能被实例化一次。如果多次尝试实例化该类，将会返回同一个实例。这样就保证了某些资源只会被加载一次，从而提高性能。

#### 二、观察者模式

观察者模式是指当一个对象状态改变时，它的所有依赖者都会收到通知并自动更新。这个模式非常适合那些需要实时更新用户界面的场景。下面是一个使用观察者模式的例子：

class ObserverList {  
 constructor() {  
 this.observerList = [];  
 }  
   
 add(observer) {  
 return this.observerList.push(observer);  
 }  
   
 remove(observer) {  
 this.observerList = this.observerList.filter((obs) => obs !== observer);  
 }  
   
 count() {  
 return this.observerList.length;  
 }  
   
 get(i) {  
 return this.observerList[i];  
 }  
}  
   
class Subject {  
 constructor() {  
 this.observers = new ObserverList();  
 }  
   
 addObserver(observer) {  
 this.observers.add(observer);  
 }  
   
 removeObserver(observer) {  
 this.observers.remove(observer);  
 }  
   
 notify(context) {  
 const observerCount = this.observers.count();  
 for (let i = 0; i < observerCount; i++) {  
 this.observers.get(i).update(context);  
 }  
 }  
}  
   
class Observer {  
 constructor() {  
 this.update = () => {};  
 }  
}  
   
const subject = new Subject();  
   
const observer1 = new Observer();  
observer1.update = function (context) {  
 console.log(`Observer 1: ${context}`);  
};  
   
const observer2 = new Observer();  
observer2.update = function (context) {  
 console.log(`Observer 2: ${context}`);  
};  
   
subject.addObserver(observer1);  
subject.addObserver(observer2);  
   
subject.notify('Hello World');

#### 三、工厂模式

工厂模式是指通过一个工厂类来创建其他类的实例。这个模式非常适合那些需要根据不同条件创建不同实例的场景。下面是一个使用工厂模式的例子

class ProductA {  
 constructor(name) {  
 this.name = name;  
 }  
 operation() {  
 console.log(`Product A (${this.name}) is working.`);  
 }  
}  
class ProductB {  
 constructor(name) {  
 this.name = name;  
 }  
 operation() {  
 console.log(`Product B (${this.name}) is working.`);  
 }  
}  
class Factory {  
 createProduct(type, name) {  
 switch (type) {  
 case 'A':  
 return new ProductA(name);  
 case 'B':  
 return new ProductB(name);  
 default:  
 throw new Error('Invalid product type.');  
 }  
 }  
}  
const factory = new Factory();  
const productA1 = factory.createProduct('A', 'productA1');  
const productA2 = factory.createProduct('A', 'productA2');  
const productB1 = factory.createProduct('B', 'productB1');  
const productB2 = factory.createProduct('B', 'productB2');  
productA1.operation(); // Product A (productA1) is working.  
productA2.operation(); // Product A (productA2) is working.  
productB1.operation(); // Product B (productB1) is working.  
productB2.operation(); // Product B (productB2) is working.

上述代码中，我们创建了两个产品类 `ProductA` 和 `ProductB`，以及一个工厂类 `Factory`。工厂类提供了一个创建产品实例的方法 `createProduct`，该方法根据传入的参数来决定创建哪种产品实例。在主程序中，我们通过工厂类创建了四个不同的产品实例，并分别执行它们的操作方法。

#### 装饰器模式

装饰者模式是指动态地给一个对象增加一些额外的功能。这个模式非常适合那些需要在运行时动态改变对象行为的场景

#### 五、代理模式

代理模式是指使用一个代理对象来控制对另一个对象的访问。这个模式非常适合那些需要控制对某些敏感资源的访问的场景

#### 六、适配器模式

适配器模式是指将不兼容接口的对象转化为兼容接口的对象。这个模式非常适合那些需要改变接口而不影响原有代码的场景。

#### 七、命令模式

命令模式是指将请求封装成一个对象，并提供与执行该请求相关的所有信息。这个模式非常适合那些需要执行多个不同操作的场景。

#### 八、观察者模式

观察者模式是指在对象之间定义一种一对多的依赖关系，使得每当一个对象改变状态时，所有依赖于它的对象都会得到通知并自动更新。这个模式非常适合那些需要在应用中实现事件处理的场景

# js的基本数据类型、值是如何存储的

JavaScript一共有8种数据类型  
7种基本数据类型：  
Undefined、Null、Boolean、Number、String、Symbol(es6新增，表示独一无二的值)和BigInt(es10新增)  
  
1种引用数据类型  
Object（Object本质上是由一组无序的名值对组成的）。  
里面包含 function、Array、Date等。  
  
JavaScript不支持任何创建自定义类型的机制，而所有值最终都将是上述 8 种数据类型之一。  
原始数据类型：直接存储在栈（stack）中，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储。  
引用数据类型：同时存储在栈（stack）和堆（heap）中，占据空间大、大小不固定。引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

# && 、||和!! 运算符

&& 叫逻辑与，在其操作数中找到第一个虚值表达式并返回它，如果没有找到任何虚值表达式，则返回最后一个真值表达式。它采用短路来防止不必要的工作。  
|| 叫逻辑或，在其操作数中找到第一个真值表达式并返回它。这也使用了短路来防止不必要的工作。在支持 ES6 默认函数参数之前，它用于初始化函数中的默认参数值。  
!! 运算符可以将右侧的值强制转换为布尔值，这也是将值转换为布尔值的一种简单方法

# js的数据类型的转换

在 JS 中类型转换只有三种情况，分别是：  
  
转换为布尔值（调用Boolean()方法）  
转换为数字（调用Number()、parseInt()和parseFloat()方法）  
转换为字符串（调用.toString()或者String()方法）

# js 内置对象

js 中的内置对象主要指的是  
在程序执行前存在全局作用域里的由 js定义的一些全局值属性、函数和用来实例化其他对象的构造函数对象。  
一般我们经常用到的如全局变量值 NaN、undefined，  
全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，  
还有提供数学计算的单体内置对象如 Math 对象。

# null 和 undefined 的区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。  
undefined 代表的含义是未定义，  
null 代表的含义是空对象（其实不是真的对象，请看下面的注意！）。一般变量声明了但还没有定义的时候会返回 undefined，null  
主要用于赋值给一些可能会返回对象的变量，作为初始化。  
当我们对两种类型使用 typeof 进行判断的时候，Null 类型化会返回 “object”，这是一个历史遗留的问题。当我们使用双等 号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

# {}和[]的valueOf和toString的结果

{} 的 valueOf 结果为 {} ，toString 的结果为 "[object Object]"  
[] 的 valueOf 结果为 [] ，toString 的结果为 ""

# js 的作用域和作用域链

作用域：   
 作用域是定义变量的区域，它有一套访问变量的规则，这套规则来管理浏览器引擎如何在当前作用域以及嵌套的作用域中根据变量（标识符）进行变量查找。  
  
作用域链：   
 作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，我们可以访问到外层环境的变量和函数。作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。  
 作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。  
当我们查找一个变量时，如果当前执行环境中没有找到，我们可以沿着作用域链向后查找。

# js 创建对象的方式

我们一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是我们可以使用函数来进行模拟，从而产生出可复用的对象创建方式，我了解到的方式有这么几种：  
  
（1）第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。  
  
（2）第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么我们就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此我们可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此我们可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次我们都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。  
  
（3）第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此我们可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。  
  
（4）第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此我们可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。  
  
（5）第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。  
  
（6）第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

# 对this、call、apply和bind的理解

1、在浏览器里，在全局范围内this 指向window对象；  
2、在函数中，this永远指向最后调用他的那个对象；  
3、构造函数中，this指向new出来的那个新的对象；  
4、call、apply、bind中的this被强绑定在指定的那个对象上；  
5、箭头函数中this比较特殊,箭头函数this为父作用域的this，  
 不是调用时的this.要知道前四种方式,都是调用时确定,  
 也就是动态的,而箭头函数的this指向是静态的,声明的时候就确定了下来；  
6、apply、call、bind都是js给函数内置的一些API，  
调用他们可以为函数指定this的执行,同时也可以传参。

# 什么是 DOM 和 BOM

DOM指的是文档对象模型，它指的是把文档当做一个对象来对待，这个对象主要定义了处理网页内容的方法和接口。  
BOM指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的法和接口。  
  
BOM的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，  
又是一个 Global（全局）(歌楼波)对象。  
这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。  
window 对象含有 location 对象、navigator(那V给特)对象、screen(斯歌锐)对象等子对象，  
并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

# 三种事件模型

事件 是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型。  
DOM0级模型：   
 这种模型不会传播，所以没有事件流的概念，同元素 绑定多个事件，只会绑定最后一次的事件，前面的会被覆盖。  
  
IE 事件模型：  
 在该事件模型中，一次事件共有两个过程，事件处理阶段，和事件冒泡阶段。  
 事件处理阶段会首先执行目标元素绑定的监听事件。  
 然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 document，  
 依次检查经过的节点是否绑定了事件监听函数，如果有则执行。  
 这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，  
 会按顺序依次执行。detachEvent删除事件  
   
DOM2 级事件模型：   
 在该事件模型中，一次事件共有三个过程，  
 第一个过程是事件捕获阶段事件处理阶段，和事件冒泡阶段。  
 捕获指的是事件从 document 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，  
 如果有则执行。。  
 这种事件模型，事件绑定的函数是

# 事件委托(代理)

本质上是利用了事件冒泡的机制。  
并且父节点可以通过事件对象获取到目标节点，因此可以把子节点的监听函数定义在父节点上，  
由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件委托  
支持为同一个DOM元素注册多个同类型事件，可将事件分成事件捕获和事件冒泡机制

# 事件传播

事件传播有三个阶段：  
捕获阶段–事件从 window 开始，然后向下到每个元素，直到到达目标元素事件或event.target。  
目标阶段–事件已达到目标元素。  
冒泡阶段–事件从目标元素冒泡，然后上升到每个元素，直到到达 window。  
  
什么是事件捕获  
当事件发生在 DOM 元素上时，该事件并不完全发生在那个元素上。在捕获阶段，事件从window开始，一直到触发事件的元素。  
什么是事件冒泡？  
事件冒泡刚好与事件捕获相反，当前元素---->body ----> html---->document ---->window。当事件发生在DOM元素上时，该事件并不完全发生在那个元素上。在冒泡阶段，事件冒泡，或者事件发生在它的父代，祖父母，祖父母的父代，直到到达window为止。

# DOM 操作

（1）创建新节点  
 createDocumentFragment() //创建一个DOM片段 (科瑞A特)(法歌们特)  
 createElement() //创建一个具体的元素  
 createTextNode() //创建一个文本节点  
  
（2）添加、移除、替换、插入  
 appendChild(node)   
 removeChild(node)  
 replaceChild(new,old) (锐普利斯)  
 insertBefore(new,old) (因色特比佛)  
  
（3）获取、查找  
 getElementById();  
 getElementsByName();  
 getElementsByTagName();  
 getElementsByClassName();  
 querySelector(); (斯来科特)  
 querySelectorAll();  
  
（4）属性操作  
 getAttribute(key); (去比又特)  
 setAttribute(key, value);  
 hasAttribute(key);  
 removeAttribute(key);

# 数据检测类型判断

检测方法4种  
  
 1、Object.prototype.toString.call()  
 作用: 可以检测所有数据类型  
 所有数据类型都可以检测,而且非常正确  
 语法: Object.prototype.toString.call( 'xxx'/11/[ ] )  
 返回值: [object Xxx], Xxx 就是对象的类型  
 2、constructor  
 作用: 可以检测基本数据类型和引用数据类型  
 弊端: 把类的原型进行重写, 很有可能把之前的constructor覆盖 检测出来的结果就会不准确  
 语法: ("xx")/([])/(function(){}).constructor === String/Array/Function   
 返回值: true/false  
3、instanceOf  
 原理: 判断对象类型，基于原型链去判断(obj instanceof Object)  
 左边对象的原型链proto上是否有右边构造函数的proptotype属性  
 作用: 判断左边的对象是否是右边构造函数的实例  
 弊端: 用于引用类型的检测, 对于基本数据类型不生效  
 语法: " "/[ ]/true instanceOf String/Array/Boolean  
 返回值: true/false  
4、typeOf  
 作用: 用于检测基本数据类型和函数  
 弊端: 引用数据类型(Arrary/function/object)只会返回Object, 不起作用  
 语法: typeOf " "/[ ]/xx  
 返回值: "string"/"boolean"/"object" (无法区分)

# 原型/原型链

原型：  
Javascript规定，每一个函数都有一个prototype对象属性，指向另一个对象  
prototype就是调用构造函数所创建的那个实例对象的原型  
  
原型链：  
实例对象与原型之间的连接，叫做原型链。  
JS在创建对象的时候，都有一个叫做\_\_proto\_\_的内置属性，用于指向创建它的函数对象的原型对象prototype。  
  
获取原型的方法  
p.proto  
p.constructor.prototype  
Object.getPrototypeOf(p)

# prototype、proto、constructor三者的关系

1、prototype:  
每一个函数都有一个prototype这个属性，而这个属性指向一个对象，这个对象我们叫做原型对象   
作用：节约内存扩展属性和方法可以实现类之间的继承   
2、\_\_proto\_\_:   
每一个对象都有一个\_\_proto\_\_属性,\_\_proto\_\_指向创建自己的那个构造函数的原型对象对象可以直接访问\_\_proto\_\_里面的属性和方法   
3、constructor：  
指向创建自己的那个构造函数 ，是原型上的方法  
  
总结：  
当我们创建一个构造函数的时候这个构造函数自带了一个prototype属性，而这个属性指向一个对象，也就是原型对象。 这个原型对象里面有一个constructor构造器，它的作用是指向创建自己的构造函数。  
除此之外 prototype还可以存放公共的属性和方法。   
当我们实例化一个对象的时候(被new调用的时候)，这个对象自带了一个 proto 属性，  
这个proto 指向创建自己的构造函数的原型对象。可以使用这个原型对象里面的属性和方法

# constructor与class的区别

传统的javascript中只有对象，没有类的概念。它是基于原型的面向对象语言。  
原型对象特点就是将自身的属性共享给新对象。   
  
ES6引入了Class（类）这个概念，通过class关键字可以定义类。  
该关键字的出现使得其在对象写法上更加清晰，更像是一种面向对象的语言  
  
constructor：   
constructor()方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。  
一个类必须有constructor()方法，如果没有显式定义，一个空的 constructor()方法会被默认添加。

# 构造函数与普通函数的区别

1. 返回值类型的区别：  
 构造函数是没有返回值类型的，  
 普通函数是有返回值类型的，即使函数没有返回值，返回值类型也要写上void。  
2. 函数名的区别：  
 构造函数的函数名必须要与类名一致，  
 普通函数的函数名只要符合标识符的命名规则即可。  
3. 调用方式的区别：  
 构造函数是在创建对象的时候由jvm调用的。  
 普通函数是由我们使用对象调用的，一个对象可以对象多次普通的函数，  
4. 作用上的区别：  
 构造函数的作用用于初始化一个对象。  
 普通函数是用于描述一类事物的公共行为的。

# 跨域出现的原因/解决方法

原因：由于浏览器的同源策略，即属于不同域的⻚面之间不能相互访问各自的⻚面内容。  
  
哪些情况下产生跨域  
1、域名不同  
2、端口号不同  
3、协议不同(http/https)  
4、域名和域名对应ip  
5、主域名相同(127.0.01 和 localhost) 多域名匹配一个ip地址  
6、子域名不同（一级和二级域名）  
  
解决方法  
  
1、后端代理   
后端不存在跨域（后端代码脱离浏览器，后端是服务器端）  
利用后端(自己公司的后端)去获取接口数据，将数据传给前端  
2、jsonp  
原理:  
 利用浏览器的"漏洞" src不受同源策略的影响，可以请求任何链接 。动态创建script标签，将事先写好的函数名传给服务器，供服务器使用  
（1）script标签src属性不存在跨域  
（2）get方式--传递函数名 --弊端  
（3）callback回调函数(传递函数名)  
3、反向代理   
proxy webpack配置  
"proxy": {  
 "/index.php": {  
 "target": "http://qinqin.net",  
 "changeOrigin": true  
 }  
 }  
4、CORS解决跨域(xhr2)(后端)  
CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。它允许浏览器向跨源(协议 + 域名 + 端口)服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。  
需要服务器(提供接口的源码里面)添加下面两句话。  
header('Access-Control-Allow-Origin:\*');  
header('Access-Control-Allow-Method:POST,GET');  
  
jsonp是一种非正式传输协议，用于解决跨域问题流程：   
1、创建一个全局函数   
2、创建一个script标签   
3、给script添加src   
4、给src添加回调函数test(callback=test) callback是传给后端的一个参数  
5、将script放到⻚面上   
6、script请求完成，将自己从⻚面上删除

# 闭包原理/优点/缺点/使用场景

闭包原理：定义在一个函数内部的函数(函数嵌套函数)，闭包就是将函数内部和函数外部连接起来的一座桥梁。  
打破了作用域链的规则 闭包就是能够读取其他函数内部变量的函数   
  
用途:  
第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，我们可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。  
另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。  
  
优点:  
1、使用闭包是不会污染全局环境，  
2、方便进行模块化开发，  
3、减少形参个数，延长了形参的生命周期，  
  
坏处:  
1、就是不恰当使用会造成内存泄漏   
  
闭包的不适用于返回闭包的函数是个特别大的函数,很多高级应用都要依靠闭包实现.  
  
使用场景  
1、通过循环给页面上多个dom节点绑定事件  
2、封装私有变量(计数器)   
3、延续局部变量的寿命  
4、高阶组件   
5、函数防抖  
  
模块化的就是以闭包为基础构建的;

# 内存泄漏

1、意外的全局变量   
2、被遗忘的定时器   
3、没有清除的dom应用 ，故要及时清除，  
4、滥用闭包  
5 深递归  
清除内存泄漏方法有两种，一是标记清除，二便是引用计数清除。

# promise/async&await

Promise是es6新增的，异步编程的一种解决方案，用来取代回调函数和事件，比传统的解决方案——回调函数和事件——更合理和更强大。  
  
Promise 对象用于延迟(deferred) 计算和异步(asynchronous)计算。一个Promise对象代表着一个还未完成，但预期将来会完成的操作。Promise 对象是一个返回值的代理，这个返回值在promise对象创建时未必已知。它允许你为异步操作的成功或失败指定处理方法。这使得异步方法可以像同步方法那样返回值:异步方法会返回一个包含了原返回值的 promise 对象来替代原返回值。  
  
有三种状态：pending（进行中）、fulfilled（resolve已成功）和rejected(锐杰克腾德)（已失败）  
  
promise的特点：  
（1）对象的状态不受外界影响。Promise对象代表一个异步操作。  
（2）一旦状态设定，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从pending变为resolve和从pending变为rejected。只要这两种情况发生，状态就凝固了  
   
promise的方法：  
promise实例方法：  
 Promise.prototype.then   
 Promise.prototype.catch  
   
 一：resolve函数的作用是，将Promise对象的状态从“未完成”变为“成功”（即从 pending 变为 resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；  
 二：reject函数的作用是，将Promise对象的状态从“未完成”变为“失败”（即从 pending 变为 rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。  
 三：Promise.prototype.finally  
 finally方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。  
  
promise的静态方法   
 Promise.all():用于将多个 Promise 实例，包装成一个新的 Promise 实例，接受一个数组作为参数，只有数组里面的每个状态都变成resolve，则新的 Promise 实例状态才会变成resolve.\*\*可以用于优化数据请求\*\*  
 Promise.race():将Promise对象数组中最先执行完成的内容通过后面then传出  
   
promise的基本使用：  
通过new promise创建一个promise对象，里面有一个参数，参数是一个回调函数，  
回调函数中有2个参数：resolve、reject   
 resolve()当异步执行成功的时候调用的方法，reject()当异步失败的时候调用的方法。  
除此之外promise有一个then方法，当成功的时候执行第一个回调函数，当失败的时候执行第二个回调函数。  
第二个回调函数也可以通过promise对象.catch调用   
  
Promise.all():当所有的异步代码都执行完毕以后才会执行.then中的操作  
Promise.race():只要有一个promise执行完毕后就会执行.then操作  
  
promise的三种状态：   
1.pending - 进行中   
2.fulfilled - 成功   
3.rejected - 失败  
  
链式调用:  
promise俗称链式调用，它是es6中最重要的特性之一  
简单的说可以不停的then调用嵌套在调用（异步之后，链式调用方式执行回调），这种操作方式称为promise  
  
async异步能干什么？  
就是用来修饰函数，使该函数异步执行，不阻碍后续函数的执行   
同时我们注意到，async修饰的函数也带有then catch方法，  
因此，经async修饰的函数也 是一个promise await只能放在async中，且只能修饰promise对象   
 1. promise的诞生是为了简化函数嵌套调用流程，也便于后续维护   
 2. async/await定义了异步函数，并在其内部可通过await等待promise对象，阻塞后 续的执行   
  
await关键字必须在async函数里面   
await会阻塞当前直到完成 async返回reject的方法,当抛出异常等同于reject   
  
async / await与 Promise的主要区别是：  
Promise代码完全都是Promise的API（then、catch等等），操作本身的语义反而不容易看出来,   
async / await函数的实现最简洁，最符合语义，几乎没有语义不相关的代码   
async / await 函数就是 Generator 函数的语法糖  
  
async/await函数的优势   
1. 使用async函数可以让代码简洁很多，不需要像Promise一样需要些then，不需要写匿名函数处理Promise的resolve值，也不需要定义多余的data变量，还避免了嵌套代码   
2. 使用aync/await的话，catch能处理JSON.parse错误 promise中不能处理   
3. 条件语句也和错误捕获是一样的，在 Async 中也可以像平时一般使用条件语句  
  
promise的状态处理的原理:

# promise.all和promise.race

Pomise.all的使用  
 Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。  
 Promise.all获得的成功结果的数组里面的数据顺序和Promise.all接收到的数组顺序是一致的，在前端开发请求数据的过程中，偶尔会遇到发送多个请求并根据请求顺序获取和使用数据的场景，使用Promise.all毫无疑问可以解决这个问题。  
归并   
Promise.race的使用  
 Promise.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。

# Generator

Generator 的中文名称是生成器，它是ECMAScript6中提供的新特性。  
在过去，封装一段运算逻辑的单元是函数。函数只存在“没有被调用”或者“被调用”的情况，  
不存在一个函数被执行之后还能暂停的情况，而Generator的出现让这种情况成为可能。  
  
通过 function\* 来定义的函数称之为“生成器函数”（generator function），它的特点是可以中断函数的执行，  
每次执行yield语句之后，函数即暂停执行，直到调用返回的生成器对象的next()函数它才会继续执行。  
  
也就是说 Generator 函数是一个状态机，封装了多个内部状态。  
执行 Generator 函数返回一个遍历器对象（一个指向内部状态的指针对象），  
调用遍历器对象的next方法，使得指针移向下一个状态。每次调用next方法，  
内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield表达式（或return语句）为止。  
  
yield关键字  
真正让Generator具有价值的是yield关键字，这个yield关键字让 Generator内部的逻辑能够切割成多个部分。  
发现函数执行到第一个yield关键字的时候就停止了。要让业务逻辑继续执行完，需要反复调用.next()  
可以简单地理解为yield关键字将程序逻辑划分成几部分，每次.next()执行时执行一部分。  
这使得程序的执行单元再也不是函数，复杂的逻辑可以通过yield来暂停。  
  
.next()调用时，返回一个对象，这个对象具备两个属性。  
其中一个属性是布尔型的done。它表示这个Generator对象的逻辑块是否执行完成。  
另一个属性是value，它来自于yield语句后的表达式的结果。  
  
通过.next()传递参数，可以赋值给yield关键字前面的变量声明。

# 堆/栈

1、堆——引用类型地址传递  
 堆：动态分配的内存，大小不定，不会自动释放。存放引用类型的值 先进后出FILO  
 引用类型: Object(Arrary, Date, Math,Function, Object)  
 访问方法: JS不能直接访问内存中的值, 只能操作对象的地址, 所以产生深/浅拷贝问题  
 栈--基本类型值传递  
2、栈——自动分配内存空间，系统自动释放，存放基本类型的值和引用类型的地址先进先出FIFO  
 基本类型: Undefined、Null、Boolean、Number 和 String, Symbol(ES6新增)  
 访问方法: 按值访问, 直接操作内存中的值

# 深/浅拷贝及方法

深/浅拷贝针对的是引用类型  
  
浅拷贝  
浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。  
新旧互相影响,改变的是地址  
新值===原值,只能拷贝一层  
  
数组方法: slice截取, concat拼接, filter过滤, map  
对象方法: Object.assign({},obj), Object.create(obj)  
   
展开运算符: {...obj},[...arr]   
  
深拷贝  
深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象  
新旧互不影响,改变的是值  
新值=/=原值,可以拷贝多层  
  
1、JSON序列化  
 JSON.parse(JSON.stringify(obj))对象-->字符串-->对象  
 这个方式的弊端：  
 1、如果obj里面有时间对象，则JSON.stringify后再JSON.parse的结果，时间将只是字符串的形式，而不是对 象的形式  
 2、如果obj里有RegExp(正则表达式的缩写)、Error对象，则序列化的结果将只得到空对象；  
 3、如果obj里有函数，undefined，则序列化的结果会把函数或 undefined丢失；  
 4、如果obj里有NaN、Infinity和-Infinity，则序列化的结果会变成null  
 5、JSON.stringify()只能序列化对象的可枚举的自有属性，不可枚举的不能被复制  
   
 可枚举:可枚举性（enumerable）用来控制所描述的属性，是否将被包括在for...in循环之中。具体来说，如果一个属性的enumerable为false，下面三个操作不会取到该属性。  
 for..in循环、Object.keys方法、JSON.stringify方法  
2、原生实现  
 递归+浅拷贝  
  
3、工具实现【第三方封装库】  
loadsh \_.cloneDeep(obj)  
 .DeepClone()  
 用于 Model / Entity / ... ... 等引用类型对象的深度克隆  
 特性说明  
 1.不需要对对象做任何特殊处理，直接 .DeepClone() 即可得到该对象的深度克隆  
 2.不受对象层次深度限制，均可实现深度克隆（下面会给出几个简单示例 ... ...）  
Immutable  
Object.create()

# set和map

Set:它类似于数组，但是成员的值都是唯一的，没有重复的值。  
Set本身是一个构造函数，用来生成 Set 数据结构，数组作为参数。  
  
不说set 数据结构Set new set()存储数据 set.size得到存储的数据长度   
  
has()判断某个值是否存在set中 foreach遍历set  
  
不说map : new map map.set map.get map.delete  
  
都是用来存储数据用的，但是存储的数据格式不同  
set 直接存储 任意类型数据  
map 存储数据的时候，必须以key，value的形式，  
set 使用forEach 遍历的时候，key和value值是一样的  
而map 遍历的时候，key就是存进去的对象的key，value就是存在的值  
  
for循环这种写法比较麻烦，因此数组提供内置的forEach方法。  
  
forEach没有返回值，无法中途跳出forEach循环，break命令或return命令都不能奏效。  
  
for...in循环主要是为遍历对象而设计的，不适用于遍历数组\*\*  
  
for...of循环相比上面几种做法，有一些显著的优点。  
有着同for...in一样的简洁语法，但是没有for...in那些缺点。  
不同于forEach方法，它可以与break、continue和return配合使用。  
提供了遍历所有数据结构的统一操作接口。

# 宏任务和微任务

js 是单线程执行的，js中的任务按顺序一个一个的执行，但是一个任务耗时太长，那么后面的任务就需要等待，为了解决这种情况，将任务分为了同步任务和异步任务，而异步任务又可以分为微任务和宏任务  
  
概念   
1.宏任务：当前调用栈中执行的代码成为宏任务。（主代码快，定时器等等）。   
2.微任务：当前（此次事件循环中）宏任务执行完，在下一个宏任务开始之前需要执行的任务,可以理解为回调事件。（promise.then，proness.nextTick等等）。   
3.宏任务中的事件放在callback queue中，由事件触发线程维护；微任务的事件放在微任务队列中，由js引擎线程维护。  
  
运行机制  
1. 在执行栈中执行一个宏任务。   
2. 执行过程中遇到微任务，将微任务添加到微任务队列中。  
3. 当前宏任务执行完毕，立即执行微任务队列中的任务。   
4. 当前微任务队列中的任务执行完毕，检查渲染，GUI线程接管渲染。   
5. 渲染完毕后，js线程接管，开启下一次事件循环，执行下一次宏任务（事件队列中取）。  
  
微任务：process.nextTick、MutationObserver、Promise.then catch finally  
宏任务：I/O、setTimeout、setInterval、setImmediate、requestAnimationFrame  
  
js执行顺序，（先执行宏任务列，微任务队列）  
先同步再异步，在此基础上先宏任务再微任务  
  
流程：  
 同步和异步任务分别进入不同的执行“场所”，同步进入主线程，异步进入Event Table并注册函数。当指定的事情完成时，Event Table会将这个函数移入Event Queue。主线程内的任务执行完毕为空，回去了Event Queue读取对应的函数，进入主线程。  
 上述过程会不断重复，也就是常说的Event Loop（事件循环）。  
 但是，JS异步还有一个机制，就是遇到宏任务，先执行宏任务，将宏任务放入event queue，然后再执行微任务，将微任务放入eventqueue，但是，这两个queue不是一个queue。当你往外拿的时候先从微任务里拿这个回调函数，然后再从宏任务的queue拿宏任务的回调函数，

# js异步操作

1、定时器都是异步操作  
2、事件绑定都是异步操作  
3、AJAX中一般我们都采取异步操作（也可以同步）  
4、回调函数可以理解为异步（不是严谨的异步操作）  
5、promise  
6、generator（ES6）   
 通过yield关键字可以让任务在需要的地方暂停,每一步的值可以通过next获取  
7、async/await(ES8)   
 await得到的就是async异步返回值,底层原理还是promise中的resolve方法  
8、设计模式-发布订阅模式  
9、事件监听

# 表单只能输入数字

正则 type number

# Proxy和Reflect

Proxy用于修改某些操作的默认行为，即对编程语言层面进行修改，属于“元编程”，Proxy意思为“代理”，即在访问对象之前建立一道“拦截”，任何访问该对象的操作之前都会通过这道“拦截”，即执行Proxy里面定义的方法  
  
let pro = new Proxy(target,handler)  
  
其中 new Proxy相当于创建了一个Proxy实例，target为所要拦截的目标对象，handler也是一个对象，里面定义的是对拦截对象所要进行的拦截方法  
  
Proxy也可以作为其他对象的原型对象使用  
  
上述实例将pro作为obj的原型对象使用，虽然obj本身没有name这个属性，但是根据原型链，会在pro上读取到name属性，之后会执行相对应的拦截操作。  
  
let pro = new Proxy(target,handler);  
let obj = Object.create(pro);

# Proxy常用的拦截方法

get(target,name,property)方法  
用于拦截某个读取属性的操作,第一个参数为目标对象，第二个参数为属性名称，第三个属性为操作所针对的对象（可选参数）  
  
set(target,name,value,property)  
用于拦截某个属性的赋值操作，第一个参数为目标对象，第二个参数为属性名，第三个参数为属性值，第四个参数为操作行为所针对的对象（可选参数）  
  
has(target,key)  
用来拦截对象是否具有某个属性值的操作，第一个参数为目标对象，第二个参数为属性名  
  
Reflect(锐付莱克特)对象:  
Reflect设计的目的是为了优化Object的一些操作方法以及合理的返回Object操作返回的结果，对于一些命令式的Object行为，Reflect对象可以将其变为函数式的行为  
  
Reflect(target,name,property) Reflect.has(obj,"name") Reflect.get(target,name,property)

# 浏览器优化

减少请求数量  
  
图片处理  
 雪碧图  
 gulp   
 Base64   
 使用字体图标来代替图片 - 自定义字体 @font-face{}   
 在安卓下可以使用webp格式的图片  
  
减小资源大小 - grunt gulp webpack  
 HTML压缩  
 CSS压缩  
 JS压缩与混乱  
 图片压缩  
  
优化网络连接  
 cdnCDN即内容分发网络，它能够实时地根据网络流量和各节点的连接、负载状  
 用户可就近取得所需内容，解决 Internet网络拥挤的状况，提高用户访问网站的响应速度  
  
优化资源加载  
 资源加载位置  
 1、CSS文件放在head中，先外链，后本页  
 2、JS文件放在body底部，先外链，后本页  
 3、body中间尽量不写style标签和script标签  
 资源加载时机  
 1、异步script标签  
 2、模块按需加载需要根据路由来加载当前页面需要的业务模块  
 3、资源懒加载与资源预加载  
  
减少重绘回流  
 当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流  
 当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如 background-color。则就叫称为重绘。  
 回流必将引起重绘，而重绘不一定会引起回流。  
减少回流和重绘  
 css3硬件加速（GPU加速）  
  
六.【DOM优化】  
1、缓存DOM  
2、减少DOM深度及DOM数量  
3、批量操作DOM  
4、批量操作CSS样式  
5、在内存中操作DOM  
6、DOM元素离线更新  
7、DOM读写分离  
8、事件代理  
9、防抖和节流  
10、及时清理环境

# 请求头

Accept(艾可塞特)  
告诉WEB服务器自己接受什么介质类型，\*/\* 表示任何类型，type/\* 表示该类型下的所有子类型，type/sub-type。  
　　  
Accept-Charset(恰斯特)  
浏览器告诉服务器自己能接收的字符集。  
  
Accept-Encoding(因扣定)  
浏览器申明自己接收的编码方法，通常指定压缩方法，是否支持压缩，支持什么压缩方法（gzip，deflate）  
  
Accept-Language  
浏览器申明自己接收的语言。语言跟字符集的区别：中文是语言，中文有多种字符集，比如big5，gb2312，gbk等等。  
　　  
Authorization(哦瑟惹C逊)  
当客户端接收到来自WEB服务器的 WWW-Authenticate 响应时，用该头部来回应自己的身份验证信息给WEB服务器。  
　　  
If-Match   
如果对象的 ETag 没有改变，其实也就意味著对象没有改变，才执行请求的动作，获取文档。  
  
If-None-Match  
如果对象的 ETag 改变了，其实也就意味著对象也改变了，才执行请求的动作，获取文档。  
 (莫得反得)  
If-Modified-Since(森恩斯)  
如果请求的对象在该头部指定的时间之后修改了，才执行请求的动作（比如返回对象），否则返回代码304，告诉浏览器该对象没有修改。例如：If-Modified-Since：Thu, 10 Apr 2008 09:14:42 GMT  
  
If-Unmodified-Since  
如果请求的对象在该头部指定的时间之后没修改过，才执行请求的动作（比如返回对象）。  
  
If-Range /reinge/  
浏览器告诉 WEB 服务器，如果我请求的对象没有改变，就把我缺少的部分给我，如果对象改变了，就把整个对象给我。浏览器通过发送请求对象的ETag 或者自己所知道的最后修改时间给 WEB 服务器，让其判断对象是否改变了。总是跟 Range 头部一起使用。  
  
Range  
浏览器（比如 Flashget 多线程下载时）告诉 WEB 服务器自己想取对象的哪部分。例如：Range: bytes=1173546  
  
Proxy-Authenticate(噢三特K特)  
代理服务器响应浏览器，要求其提供代理身份验证信息。  
  
Proxy-Authorization  
浏览器响应代理服务器的身份验证请求，提供自己的身份信息。  
  
Host  
客户端指定自己想访问的WEB服务器的域名/IP 地址和端口号。如Host：rss.sina.com.cn  
  
Referer(锐佛尔)  
浏览器向WEB 服务器表明自己是从哪个网页URL获得点击当前请求中的网址/URL，  
例如：Referer：http://www.ecdoer.com/  
  
User-Agent(A就特)  
浏览器表明自己的身份（是哪种浏览器）。  
例如：User-Agent：Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN;rv:1.8.1.14) Gecko/20080404 Firefox/2.0.0.14

# static有什么特性

new出来一个实例对象是否带有static属性 static用ES5怎么写  
  
没有创建对象，也能使用属性和调用方法  
  
用来形成静态代码块以优化程序性能。因为只会在类加载的时候执行一次。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。  
  
static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次  
  
被static修饰的变量或者方法是独立于该类的任何对象，也就是说，这些变量和方法不属于任何一个实例对象，而是被类的实例对象所共享。  
  
类第一次加载初始化的时候就去加载static部分,后面可以重新赋值  
  
static用ES5怎么写: 静态是通过类名直接调用 class A staticB   
  
直接用A.B 将B绑定在A上

# async    await

同步还是异步   
 await同步   
 async异步  
  
async和await有两个关键字,一个写在函数外面,一个写在函数里面,函数外面是异步的，函数里面是同步的，调用函数的那一行其实是异步的,下一行  
函数里面转成阻塞的  
  
async await => promise 改写  
  
async函数中  
let a=await promise 的a函数  
let b=await promise 的b函数  
  
promise.all改写   
Promise.allSettled Promise.any  
  
promise实现promise.all的方法  
  
async使用的时候报错,如何捕获try...catch  
  
var test3 = async function () {  
 try {  
 await p1();  
 await p2();  
 p3();  
 } catch (e) {  
 console.log('p1失败了', e)  
 }  
 }  
   
await后面有个接口   
接口要2S才能完成 接口2S才会执行

# $.ajax中async:false的阻塞和await这种阻塞有什么区别

没区别

# 怎么携带cookie发送给后端

设置请求头 请求头中携带cookie

# 对象浅拷贝在react中用到哪些地方,为什么

为什么在react中要使用浅拷贝  
  
redux中要求:状态是只读的,唯一且不可修改的,reducer必须是一个纯函数  
  
因为redux中数据不可更改,所以redux中的数据应该要拷贝 返回一个新值

# ajax、axios、fetch之间的详细区别以及优缺点

1.jQuery ajax   
 $.ajax({  
 type: 'POST',  
 url: url,  
 data: data,  
 dataType: dataType,  
 success: function () {},  
 error: function () {}  
 });  
优缺点：  
 是对原生XHR的封装，除此以外还增添了对JSONP的支持。  
 本身是针对MVC的编程,不符合现在前端MVVM的浪潮  
 基于原生的XHR开发，XHR本身的架构不清晰，已经有了fetch的替代方案  
 JQuery整个项目太大，单纯使用ajax却要引入整个JQuery非常的不合理  
 （采取个性化打包的方案又不能享受CDN服务）  
  
2.axios  
 axios({  
 method: 'post',  
 url: '/user/12345',  
 data: {  
 firstName: 'Fred',  
 lastName: 'Flintstone'  
 }  
 })  
 .then(function (response) {  
 console.log(response);  
 })  
 .catch(function (error) {  
 console.log(error);  
 });  
优缺点：  
 Axios本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范，  
 从它的官网上可以看到它有以下几条特性：  
 从 node.js 创建 http 请求  
 支持 Promise API  
 客户端支持防止CSRF  
 提供了一些并发请求的接口（重要，方便了很多的操作）  
  
3.fetch  
 try {  
 let response = await fetch(url);  
 let data = response.json();  
 console.log(data);  
 } catch(e) {  
 console.log("Oops, error", e);  
 }  
优缺点：  
 符合关注分离，没有将输入、输出和用事件来跟踪的状态混杂在一个对象里  
 更好更方便的写法  
 更加底层，提供的API丰富（request, response）  
 脱离了XHR，是ES规范里新的实现方式  
  
fetch是一个低层次的API，你可以把它考虑成原生的XHR，所以使用起来并不是那么舒服，  
需要进行封装，例如：  
1）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  
2）fetch默认不会带cookie，需要添加配置项  
3）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，造成了量的浪费  
4）fetch没有办法原生监测请求的进度，而XHR可以  
  
为什么要用axios?  
axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，它本身具有以下特征：  
 从浏览器中创建 XMLHttpRequest  
 从 node.js 发出 http 请求  
 支持 Promise API  
 拦截请求和响应  
 转换请求和响应数据  
 取消请求  
 自动转换JSON数据  
 客户端支持防止CSRF/XSRF

# 回流/重绘 | 防抖/节流

回流:  
当渲染树中的一部分或者全部因为元素的尺寸、布局、隐藏等改变而需要重新构建的时候，这时 候就会发生回流。 每个⻚面都至少发生一次回流，也就是⻚面第一次加载的时候。 在回流的时候，浏览器会使渲染树中受到影响的元素部分失效，并重新绘制这个部分的渲染树， 完成回流以后，  
  
重绘:  
当渲染树中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为重绘。  
  
防抖：  
短时间内多次触发同一事件，只执行最后一次，或者只执行最开始的一次，中间的不执行。  
在事件触发时，开始计时，在规定的时间（delay）内，若再次触发事件，将上一次计时（timer）清空，然后重新开始计时。保证只有在规定时间内没有再次触发事件之后，再去执行这个事件。   
  
节流：  
指定时间间隔内，若事件被多次触发，只会执行一次  
在事件触发之后，开始计时，在规定的时间（delay）内，若再次触发事件，不对此事件做任何处理。保证在规定时间内只执行一次事件.

# 减少重绘和回流的方式

CSS  
 使用transform替代top  
 使用visibility替换display:none  
 避免使用table布局  
 尽可能在DOM树的最末端改变class  
 避免设置多层内联样式  
 将动画效果应用到position属性为absoulte或fixed的元素上  
 避免使用css表达式  
 将频繁重绘或者回流的节点设置为图层  
 css硬件加速  
  
JavaScript  
 避免频繁操作样式  
 避免频繁操作DOM  
 避免频繁读取会引发重绘/回流的属性  
 对具有复杂动画的元素使用绝对定位

# 事件循环Event Loop

Event Loop 即事件循环  
是指浏览器或者Node的一种解决JavaScript单线程运行时不阻塞的一种机制，单线程的是所有任务都在主线程上完成,任务太多的时候,页面卡死,eventLoop可以解决单线程阻塞问题,程序中会有两个线程,一个主线程,一个eventLoop线程,负责主线程和其他进程之间的通信,遇到I/O的时候,主线程会让eventLoop线程通知对应的程序,主线程的任务会继续往后执行,等I/O程序执行完了,eventLoop线程会把结果返回给主线程,主线程利用回调函数调用结果,完成任务  
（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。  
（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。  
（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。  
（4）主线程不断重复上面的第三步。  
  
宏任务和微任务的执行顺序  
 一次事件循环中，先执行宏任务队列里的一个任务，再把微任务队列里的所有任务执行完毕，再去宏任务队列取下一个宏任务执行。

# 同步异步的区别

在js中，所有任务都分为同步任务和异步任务两大类。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。  
  
（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。  
（2）主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。  
（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。  
（4）主线程不断重复上面的第三步。  
  
有时候 setTimeout明明写的延时3秒，实际却5，6秒才执行函数，这又是因为什么？  
  
答：setTimeout 并不能保证执行的时间，是否及时执行取决于 JavaScript 线程是拥挤还是空闲。  
浏览器的JS引擎遇到setTimeout，拿走之后不会立即放入异步队列，同步任务执行之后，timer模块会到设置时间之后放到异步队列中。js引擎发现同步队列中没有要执行的东西了，即运行栈空了就从异步队列中读取，然后放到运行栈中执行。所以setTimeout可能会多了等待线程的时间。  
这时setTimeout函数体就变成了运行栈中的执行任务，运行栈空了，再监听异步队列中有没有要执行的任务，如果有就继续执行，如此循环，就叫Event Loop。

# 数据请求设置请求数据格式Content-Type

1.浏览器默认的 application/x-www-form-urlencoded(url因扣得的)  
这应该是最常见的 POST 提交数据的方式了。浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。  
  
2.multipart(莫踢怕特)/form-data  
这也是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，就要让 form 的 enctype 等于这个值  
  
3.application/json  
除了低版本 IE 之外的各大浏览器都原生支持 JSON.stringify   
  
4.text/xml  
相比于JSON，不能更好的适用于数据交换，它包含了太多的包装, 而且它跟大多数编程语言的数据模型不匹配，让大多数程序员感到诧异，XML是面向数据的，JSON是面向对象和结构的，JSON会给程序员一种更加亲切的感觉。

# 数组扁平化

1. reduce  
  
遍历数组每一项，若值为数组则递归遍历，否则concat。  
  
function flatten(arr) {   
 return arr.reduce((result, item)=> {  
 return result.concat(Array.isArray(item) ? flatten(item) : item);  
 }, []);  
}  
  
reduce是数组的一种方法，它接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。  
reduce包含两个参数：回调函数，传给total的初始值  
  
// 求数组的各项值相加的和：   
arr.reduce((total, item)=> { // total为之前的计算结果，item为数组的各项值  
 return total + item;  
}, 0);  
  
  
2. toString & split  
调用数组的toString方法，将数组变为字符串然后再用split分割还原为数组  
  
function flatten(arr) {  
 return arr.toString().split(',').map(function(item) {  
 return Number(item);  
 })  
}   
  
因为split分割后形成的数组的每一项值为字符串，所以需要用一个map方法遍历数组将其每一项转换为数值型  
  
3. join & split  
和上面的toString一样，join也可以将数组转换为字符串  
  
function flatten(arr) {  
 return arr.join(',').split(',').map(function(item) {  
 return parseInt(item);  
 })  
}  
  
4. 递归  
  
递归的遍历每一项，若为数组则继续遍历，否则concat  
  
function flatten(arr) {  
 var res = [];  
 arr.map(item => {  
 if(Array.isArray(item)) {  
 res = res.concat(flatten(item));  
 } else {  
 res.push(item);  
 }  
 });  
 return res;  
}  
  
  
5. 扩展运算符  
es6的扩展运算符能将二维数组变为一维  
  
[].concat(...[1, 2, 3, [4, 5]]); // [1, 2, 3, 4, 5]  
  
根据这个结果我们可以做一个遍历，若arr中含有数组则使用一次扩展运算符，直至没有为止。  
  
function flatten(arr) {  
 while(arr.some(item=>Array.isArray(item))) {  
 arr = [].concat(...arr);  
 }  
 return arr;  
}

# 如何扩展数组,让他拥有新的方法

Array.prototype.push.apply(a,b)

# class中super作用是什么

es5 的继承是先创建子类的this，然后将父类的方法添加到子类的this上去；   
es6 的继承是创建父类的this对象，然后再对this对象添加方法/属性。   
而super方法就是用来创建父类this对象的。  
  
实际上执行的是 super.sport.call(this);

# xss攻击和csrf攻击是什么

1、CSRF（Cross-site request forgery）：跨站请求伪造。  
（1）登录受信任网站A，并在本地生成Cookie。  
 （如果用户没有登录网站A，那么网站B在诱导的时候，请求网站A的api 接口时，会提示你登录）  
（2）在不登出A的情况下，访问危险网站B（其实是利用了网站A的漏洞）  
  
CSRF如何防御  
方法一: Token 验证：（用的最多）  
 （1）服务器发送给客户端一个token；  
 （2）客户端提交的表单中带着这个token。  
 （3）如果这个 token 不合法，那么服务器拒绝这个请求。  
方法二: 隐藏令牌：  
 把 token 隐藏在 http 的 head头中。  
 方法二和方法一有点像，本质上没有太大区别，只是使用方式上有区别。  
方法三: Referer(锐服尔) 验证：  
 Referer 指的是页面请求来源。意思是，只接受本站的请求，服务器才做响应；如果不是，就拦截。  
   
2、XSS（Cross Site Scripting）：跨域脚本攻击。  
XSS攻击的核心原理是：  
 不需要你做任何的登录认证，它会通过合法的操作（比如在url中输入、在评论框中输入），  
 向你的页面注入脚本（可能是js、hmtl代码块等）。  
最后导致的结果可能是：  
 盗用Cookie破坏页面的正常结构，插入广告等恶意内容D-doss攻击  
XSS的攻击方式  
1、反射型  
 发出请求时，XSS代码出现在url中，作为输入提交到服务器端，服务器端解析后响应，  
 XSS代码随响应内容一起传回给浏览器，最后浏览器解析执行XSS代码。这个过程像一次反射，所以叫反射型XSS。  
2、存储型存  
 储型XSS和反射型XSS的差别在于，提交的代码会存储在服务器端（数据库、内存、文件系统等），  
 下次请求时目标页面时不用再提交XSS代码。  
  
XSS的防范措施（encode + 过滤）主要有三个：  
1、编码：  
 对用户输入的数据进行HTML Entity(安特踢) 编码。  
2、过滤：  
 移除用户输入的和事件相关的属性。如onerror可以自动触发攻击，还有onclick等。  
 （总而言是，过滤掉一些不安全的内容）移除用户输入的Style节点、Script节点、Iframe节点。  
 （尤其是Script节点，它可是支持跨域的呀，一定要移除）。  
3、校正  
 避免直接对HTML Entity进行解码。  
 使用DOM Parse转换，校正不配对的DOM标签。  
 这个概念，它的作用是把文本解析成DOM结构。  
 比较常用的做法是，通过第一步的编码转成文本，然后第三步转成DOM对象，然后经过第二步的过滤。  
 还有一种简洁的答案：  
 首先是encode，如果是富文本，就白名单。  
  
3、CSRF 和 XSS 的区别:  
区别一：  
 CSRF：需要用户先登录网站A，获取 cookie。  
 XSS：不需要登录。  
区别二：（原理的区别）  
 CSRF：是利用网站A本身的漏洞，去请求网站A的api。  
 XSS：是向网站 A 注入 JS代码，然后执行 JS 里的代码，篡改网站A的内容。

# webapp和小程序有什么区别

首先，微信小程序已经提供了一套 view, data, model, router 层的开发工具，  
对于开发简单应用，小程序是可以比 webapp 更加快速的。  
  
但是实际上微信小程序提供的这一套开发框架，要开发一些复杂应用，是很困难的，  
因为：小程序不支持 npm 等 package manager(麦呢橘)无法复用社区中已经很成熟的 web 框架和工具组件只能封装 view 和 style，无法封装行为（handler），行为只能定义在 page 上小程序有 1mb 的限制，所以我们只能将图片之类的静态资源事先放在服务器上  
  
其次，微信小程序是由微信自己来 host，开发者只需要上传就好，  
而微信 webapp 需要开发者自己 host，还需要注册域名甚至备案才可以调用微信接口以及跟公众号集成。  
所以微信小程序降低了开发者的门槛。  
  
综上，对于简单的工具型应用，微信小程序可以让开发者加快开发速度，降低发布门槛，  
这种类型的应用比较适合微信小程序。对于复杂的应用，webapp 是更适合的形式。

# react优缺点

优点：   
1、采用虚拟 DOM，它并不直接对DOM进行操作，安插在javascript逻辑和实际的DOM之间，性能好   
2、跨s容：虚拟DOM帮助我们解决了跨浏览器问题，它为我们提供了标准化的API，甚至在IE8中都是没问题的。   
3、一切都是组件：代码更加 模块化 ，重用代码更容易，可维护性高。   
4、单向数据流：redux 实现是一个用于在JavaScript应用中创建单向数据层的架构，它随着React视图库的开发而被Facebook概念化。   
5、同构、纯粹的javascript：因为搜索引擎的爬虫程序依赖的是服务端响应而不是JavaScript的执行，预渲染你的应用有助于搜索引擎优化。   
6、兼容性好：比如使用RequireJS来加载和打包，而Browserify(不弱瑟)和Webpack适用于构建大型应用。它们使得那些艰难的任务不再让人望而生畏。   
7、JSX语法：为了更加便利的模拟DOM结构，我们使用了JSX语法，可以让我们在JS中编译DOM结构 8、函数式编程：JS的最大特点就是函数式编程，在React中，函数式编程可谓式无处不见  
  
缺点：  
不适合单独做一个完整的框架：react是视图层框架，大型项目想要一套完整的框架的话，也许还需要引入Flux和route相关的东西。

# react生命周期

16版本：  
初始阶段:  
constructor  
 定义state  
componentWillMount(这个钩子函数在将来会被弃用)  
 为挂载做准备工作  
render   
 解析 this.state 和 this.props   
 将jsx型的虚拟dom渲染为对象类型的虚拟dom  
 render 中不允许使用 this.setState() 如何你使用了，就会栈溢出  
componentDidMount  
 表示组件挂载结束  
 虚拟dom -> Real dom  
 可以获取真实dom  
 数据请求 -> 赋值给state  
 第三方库实例化/DOM操作  
   
更新阶段:  
componentWillReceiveProps(在17版本中被弃用)  
 这个钩子实际作用是判断组件身上的props是否发生改变  
shouldComponentUpdate  
 可以决组件是否要更新渲染 return true/false  
 接收新旧状态，用于作对比（浅对比）  
 一般需要我们手动作深对比  
 这个钩子函数是React组件性能优化的一种方式  
componentWillUpdate  
 表示组件更新前的准备  
 这个钩子函数在未来版本被弃用  
render  
 和初始化阶段的作用一致  
componentDidUpdate  
 对Real dom 作操作  
 注意：父组件数据更新，子组件会重新运行render，反之不行  
 每次都要运行子组件render，会造成react性能浪费   
 粗略解决方案：  
 在shouldComponentUpdate中，可判断哪些数据改变从而来控制return true/false   
 继而决定是否需要更新render渲染  
 更好的解决方案：  
 1.引用PureComponent，让类组件继承（extends）PureComponent   
 如：class App extends PureComponent{}  
 2.引用memo，让函数组件写在memo（）里 如：const memoApp =memo（function App() {}）  
 3.除了memo之外，更细致的是函数组件中还可以用useCallback（方法）和useMemo（对象和数组）两个钩子阻止组件render  
  
销毁阶段:  
componentWillUnmount  
 销毁组件  
 清除无用实例和事件  
错误捕获:  
 componentDidCatch  
 用户捕获子组件throw的错误，然后显示回退UI  
  
15版本：  
相对于16版本，初始化阶段（constructor钩子变成下列两个）：  
  
定义于初始化props的 getDefalutProps 钩子函数  
  
定义于初始化state的 getInitialState 钩子函数  
没有错误捕获阶段  
  
17版本：  
相对于16版本 少3多2  
  
少了componentWillMount，componentWillReceiveProps，componentWillUpdate 3个钩子函数  
 (斯塔得可) (迪外的)  
多了static getDerivedStateFromProps钩子函数 (破外服)  
 一个静态方法，所以不能在这个函数里面使用this，这个函数有两个参数nextProps和prevState，这个函数会返回一个对象用来更新当前的state对象，如果不需要更新可以返回null。  
 简单说就是，可以增加一次state状态  
 (斯那坡秀的)  
多了getSnapshotBeforeUpdate钩子函数  
 快照  
 这个函数有一个返回值，会作为第三个参数传递给componentDidUpdate钩子

# redux流程  redux的原理

view用actionCreator(酷睿A特)创建一个action，里面可能包含一些数据  
使用store的dispatch方法将action传入store  
store将action与旧的state转发给reducer  
reducer深拷贝state,并返回一个新的state给store  
store接收并更新state  
使用store.subscribe(萨布斯快不)订阅更新,重新render组件  
  
redux组成  
 state :用来存储数据和数据管理的、更新视图  
 reducer:是一个纯函数，接收旧 state 和 action，根据不同的 Action 做出不同的操作并返回新的 state  
 actions:发送动作给reducer,reducer接收动作，判断动作类型修改数据，修改事件后，组件重新做redux事件的订阅  
   
Redux三大原则  
单一数据源:  
 整个应用的 state 被存储在一个 Object tree 中，且只存在于唯一的Store中  
   
state 是只读的:  
 唯一改变 state 的方法就是触发 action，action 是一个用于描述发生事件的普通对象，视图部分只需要表达想要修改的意图，所有修改都会被集中化处理。  
   
状态的改变通过纯函数来完成:  
 Redux使用纯函数方式来执行状态的修改，Action表明了修改状态值的意图，而真正执行状态修改的则是Reducer。且Reducer必须是一个纯函数，当Reducer接收到Action时，Action并不能直接修改State的值，而是通过创建一个新的状态对象来返回修改的状态。

# redux 中间件

在action 和 store 之间执行中间件  
  
Redux中间件机制  
 Redux本身就提供了非常强大的数据流管理功能，但这并不是它唯一的强大之处，它还提供了利用中间件来扩展自身功能，以满足用户的开发需求  
  
 (米的为尔)   
applyMiddlewares():它是 Redux 的原生方法，作用是将所有中间件组成一个数组，依次执行。  
applyMiddleware顾名思义，用于调用各种中间件；  
applyMiddleware执行后，将所有入参中间件存入一个数组，并且返回一个闭包（闭包的概念不做累述）  
闭包接受一个createStore作为入参并且执行后返回下一个闭包。

# redux-thunk|异步action

首先检查参数 action 的类型，如果是函数的话，就执行这个 action 函数，  
并把 dispatch, getState, extraArgument(哎克斯拽埃歌门特) 作为参数传递进去，  
否则就调用 next 让下一个中间件继续处理 action   
  
好处  
 可以进行前后端数据交互  
缺点  
 将带有数据请求的action和没有带有数据请求的action混在一起了  
 缺点解决： 弃用redux-thunk,使用redux-saga  
 redux-saga可以将异步action和普通action区别开来

# redux-saga|集中处理异步action

redux-saga可以将异步action和普通action区别开来，控制器与更优雅的异步处理  
redux-saga就是用Generator(杰呢瑞特)来处理异步。  
  
redux-saga文档并没有说自己是处理异步的工具，而是说用来处理边际效应（side effects），这里的边际效应你可以理解为程序对外部的操作，比如请求后端，比如操作文件。  
  
redux-saga相当于在Redux原有数据流中多了一层，通过对Action进行监听，从而捕获到监听的Action，然后可以派生一个新的任务对state进行维护（这个看项目本身的需求），通过更改的state驱动View的变更。  
  
redux-saga同样是一个redux中间件，它的定位就是通过集中控制action，起到一个类似于MVC中控制器的效果。  
同时它的语法使得复杂异步操作不会像promise那样出现很多then的情况，更容易进行各类测试。

# redux与react-redux的关系

redux是独立的应用状态管理工具。它是可以独立于react之外的。如果我们需要在react当中运用它，那么我们需要手动订阅store的状态变化，来对我们的react组件进行更新。react-reudx这个工具，就帮我们实现了这个功能，我们只需对store进行处理，react组件就会有相应的变化。  
  
Redux的核心由三部分组成：Store, Action, Reducer。  
Store: 是个对象，贯穿你整个应用的数据都应该存储在这里。  
Action: 是个对象，必须包含type这个属性，reducer将根据这个属性值来对store进行相应的处理。除此之外的属性，就是进行这个操作需要的数据。  
Reducer: 是个函数。接受两个参数：要修改的数据(state) 和 action对象。根据action.type来决定采用的操作，对state进行修改，最后返回新的state。  
  
总结  
Redux: store, action, reducer  
store: getState, dispatch, subscribe  
combineReducers (克木拜恩)  
createStore  
store ️ dispatch ️ action ️ reducer  
  
react-redux:  
connect : 将store作为props注入  
provider(破外的): 使store在子孙组件的connect中能够获取到

# react-reudx

1. UI组件|显示页面?? (破森特逊弄)  
 React-Redux 将所有组件分成两大类：UI 组件（presentational component）和容器组件  
2. 容器组件|负责管理数据/复杂逻辑?  
 UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。  
3. Provider组件|容器组件获取state??  
 所有的 UI 组件都由用户提供，容器组件则是由 React-Redux 自动生成。也就是说，用户负责视觉层，状态管理则是全部交给它。  
4. connect()|UI组件+容器组件  
 React-Redux 提供connect方法，用于从 UI 组件生成容器组件。connect的意思，就是将这两种组件连起来。

# DVA与CRA相比的优点

dva:  
 dva 首先是一个基于 redux 和 redux-saga 的数据流方案，然后为了简化开发体验，dva 还额外内置了 react-router 和 fetch，所以也可以理解为一个轻量级的应用框架。--- 来自官方。  
 相比于cra只是多了内置的redux和redux-saga，帮我们处理了数据流这方面的需求而已。如果只是想要达到这个效果的话，直接在cra中增加dva-core的依赖也是可以做到的。  
  
umi:  
 是一个可插拔的企业级 react 应用框架。umi和cra都是应用框架，可能相比cra来说umi的功能点更多一些，只能说是功能性的话umi要相对来说更胜一筹

# flux

flux 是 react 中的类似于 vuex 的公共状态管理方案，它是 Facebook 官方给出的应用架构，利用数据的单向流动的形式对公共状态进行管理。现已不推荐使用。  
  
flux的组成  
 View：视图层  
 Action：视图发出的消息  
 Dispatcher：派发者，用来接收Action，执行回调函数  
 Store：数据层，存放状态，一旦发生改动  
   
flux 在进行数据更新时，会经历以下几步：  
 用户与 View 层交互，触发 Action  
 Action 使用 dispatcher.dispatch 将Action自己的状态发送给dispatcher  
 dispatcher 通过register注册事件，再通过Action传入的类型来触发对应的 Store 回调进行更新  
 Store 里进行相应的数据更新，并触发 View 层事件使试图也同步更新  
 View层 收到信号进行更新  
  
redux和flux的区别  
　　1）redux是flux中的一个实现  
　　2）在redux中我们只能定义一个store，在flux中我们可以定义多个  
　　3）在redux中，store和dispatch都放到了store，结构更加清晰  
　　4）在redux中本身就内置State对象，对仓库的管理更加明确

# React 中 keys 的作用

key是react用于追踪哪些列表被修改、被添加或者被移出的辅助标识。  
  
在开发过程中，我们需要保证某些元素在同级的元素中key是具有唯一性的特性，在react Diff算法中React会借助元素的key值来判断该元素是新创建的还是移动而来的元素，从而减少元素的不必要的重复渲染。此外，我们还需要借助key值来判断元素与本地状态的关联关系，因此我们绝不可忽视转换函数中 Key 的重要性。  
  
react根据key来决定是销毁重新创建组件还是更新组件，原则是：  
  
key相同，组件有所变化，react会只更新组件对应变化的属性。  
key不同，组件会销毁之前的组件，将整个组件重新渲染。

# react新特性和diff算法

render  
支持返回这五类:  
React elements, 数组, Fragments, Portal, String/numbers, boolean/null, 基础数据类型  
  
Fiber  
React Fiber的方法其实很简单——分片。把一个耗时长的任务分成很多小片，每一个小片的运行时间很短，虽然总时间依然很长，但是在每个小片执行完之后，都给其他任务一个执行的机会，这样唯一的线程就不会被独占，其他任务依然有运行的机会。  
  
新的生命周期函数  
由于异步渲染的改动，componentWillMount, componentWillReceiveProps,componentWillUpdate 三个函数将被废弃。  
由于这是一个很大的改变会影响很多现有的组件，所以需要慢慢的去改。  
目前react 16 只是会报warning，在react 17就只能在前面加UNSAFE\_的前缀来使用  
  
diff算法  
作用: 计算出Virtual DOM中真 正变化的部分，并只针对该部分进行原生DOM操作，而非重新渲染整个页面  
  
getDerivedStateFromProps  
static getDerivedStateFromProps(props, state)在调用render方法之前调用，  
无论是在初始安装还是后续更新。它应返回一个对象来更新状态，或者返回null以不更新任何内容。  
同城比较   
根据props更新state  
这个生命周期可用于替代componentWillReceiveProps  
  
getSnapshotBeforeUpdate()  
getSnapshotBeforeUpdate(prevProps, prevState)在最近呈现的输出被提交到例如DOM之前调用。它使组件可以在可能更改之前从DOM捕获一些信息（例如滚动位置）。此生命周期返回的任何值都将作为参数传递给componentDidUpdate()。  
  
hooks  
  
lazy、suspense  
lazy需要跟Suspence配合使用。  
lazy实际上是帮助我们实现代码分割的功能。  
  
由于有些内容，并不一定要在首屏展示，所以这些资源没有必要一开始就要去获取，那么这些资源就可以动态获取。  
这样的话，相当于把不需要首屏展示的代码分割出来，减少首屏代码的体积，提升性能。  
  
Suspence 很像Error Boundary，不同的是Error Boundary是用来捕获错误，显示相应的callback组件。而Suspence是用来捕获还没有加载好的组件，并暂停渲染，显示相应的callback。

# setState概述,同步异步

1. 两个参数及用法  
 第一个参数可以是对象或者函数，是更新state   
 第二个参数获取最新的state,副作用操作，dom操作事件触发声明，数据获取，第三方库实例化  
  
  
2. 同步/异步原理  
 setState在合成事件和钩子函数中是异步的   
 在原生事件和setTimeout中是同步的  
  
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，形式了所谓的“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。  
  
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。

# react合成事件

1. 合成事件原理  
 如果DOM上绑定了过多的事件处理函数，整个页面响应以及内存占用可能都会受到影响。React为了避免这类DOM事件滥用，同时屏蔽底层不同浏览器之间的事件系统差异，实现了一个中间层——SyntheticEvent。  
  
当用户在为onClick添加函数时，React并没有将Click时间绑定在DOM上面。  
而是在document处监听所有支持的事件，当事件发生并冒泡至document处时，React将事件内容封装交给中间层SyntheticEvent（负责所有事件合成）  
所以当事件触发的时候，对使用统一的分发函数dispatchEvent将指定函数执行。  
  
2. 与原生事件的区别  
 React合成事件一套机制：React并不是将click事件直接绑定在dom上面，而是采用事件冒泡的形式冒泡到document上面，然后React将事件封装给正式的函数处理运行和处理。

# react路由传参和读参

1.params  
<Route path='/path/:name' component={Path}/>  
<link to="/path/2">xxx</Link>  
this.props.history.push({pathname:"/path/" + name});  
读取参数用:this.props.match.params.name  
优势:刷新地址栏，参数依然存在  
缺点:只能传字符串，并且，如果传的值太多的话，url会变得长而丑陋。  
  
2.query  
<Route path='/query' component={Query}/>  
<Link to={{ path : ' /query' , query : { name : 'sunny' }}}>  
this.props.history.push({pathname:"/query",query: { name : 'sunny' }});  
读取参数用: this.props.location.query.name  
优势: 传参优雅，传递参数可传对象；  
缺点: 刷新地址栏，参数丢失  
  
3.state  
<Route path='/sort ' component={Sort}/>  
<Link to={{ path : ' /sort ' , state : { name : 'sunny' }}}>   
this.props.history.push({pathname:"/sort ",state : { name : 'sunny' }});  
读取参数用: this.props.location.query.state   
优缺点同query  
  
4.search  
<Route path='/web/departManange ' component={DepartManange}/>  
<link to="web/departManange?tenantId=12121212">xxx</Link>  
this.props.history.push({pathname:"/web/departManange?tenantId" + row.tenantId});  
读取参数用: this.props.location.search  
优缺点同params

# redux和mobx(类似双向数据绑定)区别

1. 组成部分  
 actions->state->computed values->Reactions  
2. 工作流  
 在mobx中， 数据是通过加 @observable 作为可监测的被观察者， 在view层中， 你可以通过添加@observer 将view作为观察者，对数据进行监测， 如果要触发改变数据，则使用@action, 事实上，你可以直接在view层改变数据， 但这种方式不便监控数据，因此不推荐直接改变数据。 而@computed可以用来计算数据， 也可以是计算多个数据之后返回新的数据， 如果其中数据改变， @computed也会触发改变  
3. 优点  
 不同于redux的单一数据流， mobx中，你可以同时在各个地方使用同一份state, 也可以在一个页面中使用多个store文件

# react组件通信5种

1.父组件向子组件通信  
React数据流动是单向的,父组件向子组件通信也是最常见的;父组件通过props向子组件传递需要的信息  
  
2.子组件向父组件通信  
利用回调函数，可以实现子组件向父组件通信：父组件将一个函数作为 props 传递给子组件，子组件调用该回调函数，便可以向父组件通信。  
  
3.非嵌套组件间通信  
非嵌套组件，就是没有任何包含关系的组件，包括兄弟组件以及不在同一个父级中的非兄弟组件。对于非嵌套组件，可以采用下面两种方式：  
 利用二者共同父组件的 context 对象进行通信  
 使用自定义事件的方式  
如果采用组件间共同的父级来进行中转，会增加子组件和父组件之间的耦合度，如果组件层次较深的话，找到二者公共的父组件不是一件容易的事，当然还是那句话，也不是不可以。  
  
4.跨组件通信  
所谓跨级组件通信，就是父组件向子组件的子组件通信，向更深层的子组件通信。跨级组件通信可以采用下面两种方式：  
 中间组件层层传递 props  
 使用 context 对象  
中间组件层层传递 props:  
 如果父组件结构较深，那么中间的每一层组件都要去传递 props，增加了复杂度，并且这些 props 并不是这些中间组件自己所需要的。不过这种方式也是可行的，当组件层次在三层以内可以采用这种方式，当组件嵌套过深时，采用这种方式就需要斟酌了。  
   
使用 context 对象:  
 context 相当于一个全局变量，是一个大容器，我们可以把要通信的内容放在这个容器中，这样一来，不管嵌套有多深，都可以随意取用。  
使用 context 也很简单，需要满足两个条件：  
 上级组件要声明自己支持 context，并提供一个函数来返回相应的 context 对象  
 子组件要声明自己需要使用 context  
  
5.redux  
首先由view dispatch拦截action，然后执行对应reducer并更新到store中，最终views会根据store数据的改变执行界面的刷新渲染操作。  
 1)首先先把redux相关工具安装好   
 2)通过创建一个store实例createStore,接收一个rootReducer和中间件执行函数  
 3)创建分块的数据rootReducer,通过combineReducers打造rootReducer,里面放分块的数据  
 4)在组件中通过高阶组件connect函数,接收store里的数据,把ActionCreators里的方法绑定到组件身上,  
 并且可以发送动作action给reducer  
 5)在reductor中根据action中的type动作类型,判断动作修改数据

# 组件按需加载

（实现方式，需要的依赖）懒加载  
  
1、vue异步组件技术  
 vue-router配置路由，使用vue的异步组件技术，可以实现按需加载。  
 但是，这种情况下一个组件生成一个js文件。  
  
2.import()  
  
3.webpack提供的require.ensure()  
  
4.第三方库比如react-loadable   
  
5.lazyload-loader

# 组件销毁

componentWillUnmount  
  
定义state flag:false 作为开关 定义方法if判断是否为1,改变state flag:true  
  
将方法绑定组件上 可以用三目或者短路原则&&控制组件的显示隐藏

# Ant D中input输入框三个属性怎么实现

onChange输入框内容改变时候回调 value输入的内容 defaultValue输入框默认值  
  
input默认值在input中绑定value 定义一个state 和input双向数据绑定 做成受控组件 定义一个事件 改变的时候获取e.target.value

# Hooks。不知道就闭麦

为什么使用hook  
 不必写class组件就可以用state和其他的React特性；  
 自定义hook 实现组件的复用 用useEffects代替生命周期方法 代码更加简洁  
  
出现原因3点  
 为了让函数组件拥有类组件的功能。  
 原因  
 比如useState、useReducer 状态定义  
 比如useEffect、useLayoutEffect 生命周期功能  
 比如useRef useImperativeHandle 替代了类组件的 Ref  
 为了优化函数组件，比如useMemo、useCallback  
 函数组件添加新的方法,如  
 useDebugValue显示自定义hook，自己添加hook类名  
 自定义hooks来复用状态  
相当于按需加载类逐渐的功能呢 就是hooks是结构出来的。 你不用你就xiang  
优点4点  
1.让函数组件可以定义状态   
 让函数组件可以使用生命周期、监听数据  
 让函数组件可以有Ref的功能（父组件获取子组件）  
 优化函数组件 useMemo、useCallback  
 自定义hooks来复用状态,  
 代码量比类组件更少，更清爽。  
2.使用规则2点  
 不要在循环，条件或嵌套函数中调用 Hook  
 只在函数组件中使用 Hooks  
3.常用Hook  
 useState:  
 定义状态 修改状态  
 返回一个数组，其中第一项是状态值，第二项是一个更新状态的函数。  
 状态一旦改变，React 就会重新渲染组件，变量获取新的状态值。  
 useEffect:  
 相当于componentDidMount,componentDidUpdate,componentWillUnmount三个钩子的组合  
 参数一：执行的回调函数；  
 参数二：该useEffect在哪些state发生变化时，才重新执行；  
 第二个参数是空数组的时候执行一次,相当于componentDidMount,不加的时候执行多次  
 第二个参数的作用就是 仅在更改时更新，实现性能的优化  
 DOM操作 第三方实例化可以做 清除无用实例和事件  
 useEffect传入的回调函数本身可以有一个返回值，  
   
 这个返回值是另外一个回调函数，来模拟componentWillUnmount  
 useLayoutEffect:  
 布局副作用  
 useEffect 在浏览器渲染完成后执行  
 useLayoutEffect 在浏览器渲染前执行  
 useLayoutEffect 里的任务最好影响了 Layout  
 为了用户体验，优先使用 useEffect (优先渲染)  
 useContext:  
 跨组件通信 createContext创建一个组件 <numContext.Provider value={num}>  
 useDebugValue:  
 自定义 hook 的标签 方便调试台查看  
 useMemo:  
 记忆组件 动态缓存 新值和旧值一样,不重新渲染页面,优化作用,类似于shouldComponentUpdate useCallBack:  
 作用和 useMemo 一样  
 useMemo和useCallback都会在组件第一次渲染的时候执行，之后会在其依赖的变量发生改变时再次执行  
 并且这两个hooks都返回缓存的值，useMemo返回缓存的变量，useCallback返回缓存的函数。  
 useCallback(x => console.log(x), [m]) 等价于  
 useMemo( () => x => console.log(x), [m])  
 useRef:  
 返回一个可变的ref对象  
 useImperativeHandle:  
 将组件中的方法放到外面使用  
 搭配React.forwardRef  
  
5.自定义hook  
 类似于高阶组件   
 高阶组件返回的一个类组件，而自定义Hook可以返回任何东西  
 高阶组件必须传递一个组件作为参数，而自定义Hook不需要  
 function useFriendStatus(friendID) {  
 const [isOnline, setIsOnline] = useState(null);  
 // 在开发者工具中的这个 Hook 旁边显示标签  
 // e.g. "FriendStatus: Online"  
 useDebugValue(isOnline ? 'Online' : 'Offline');  
 return isOnline;  
 }  
可以理解为数据的操作都在hook里进行，而外部只关心自己想要的。我只要数据列表，获取产品钩子（可能并不需要，可通过参数变更从而触发重新获取数据）、删除产品钩子  
为了封装方法：节流；

# react useState() Hook

1.使用 useState() 进行状态管理  
 调用useState() Hook 来启用函数组件中的状态。  
 useState(initialValue)的第一个参数initialValue是状态的初始值。  
 [state, setState] = useState(initialValue)返回一个包含2个元素的数组:状态值和状态更新函数。  
 使用新值调用状态更新器函数setState(newState)更新状态。或者，可以使用一个回调setState(prev => next)来调用状态更新器，该回调将返回基于先前状态的新状态。  
 调用状态更新器后，React 确保重新渲染组件，以使新状态变为当前状态。  
2.多种状态  
 通过多次调用useState()，一个函数组件可以拥有多个状态。  
 需要注意的，要确保对useState()的多次调用在渲染之间始终保持相同的顺序。  
3.状态的延迟初始化  
 每当 React 重新渲染组件时，都会执行useState(initialState)。如果初始状态是原始值（数字，布尔值等），则不会有性能问题。  
 当初始状态需要昂贵的性能方面的操作时，可以通过为useState(computeInitialState)提供一个函数来使用状态的延迟初始化，该函数仅在初始渲染时执行一次，以获得初始状态。在以后的组件渲染中，不会再调用该函数，从而跳过昂贵的操作。  
4.调用 useState()  
在使用useState() Hook 时，必须遵循 Hook 的规则：  
1.仅顶层调用Hook:不能在循环，条件，嵌套函数等中调用useState()。在多个useState()调用中，渲染之间的调用顺序必须相同。  
2.仅从React 函数调用 Hook:必须仅在函数组件或自定义钩子内部调用useState()。

# 高阶组件HOC

1.名词解释/作用  
 使函数复用，可以通过给组件传递方法来复用高阶组件中的函数方法。  
 高阶组件特点  
 高阶组件是一个函数  
 高阶组件接收一个组件作为参数进行使用，且需要在render函数中return返回这个组件  
 高阶组件的目的是为了： 复用组件,将多个组件都要使用的类似逻辑放在同一个地方进行处理，类似于在Vue中封装cookie以供重复使用  
   
2.常用高阶组件4个  
 React.memo()  
 connect()  
 provider()   
 withRouter() // 可以使用  
3.有自己封装过吗   
 拖拽封装，给组件里面的标签添加方法就可以实现拖拽，并返回标签的x与y坐标。  
 深对比，深复制封装、正则封装、页面路由跳转、路由数据接收解析。

# 拖拽封装思路

重点：  
1、一定要绝对定位，脱离文档流才可以移动。  
2、绑定拖拽的元素，移动和鼠标松开后是对document的绑定，因为移动的是整个div。  
3、点击：a= 获取当前鼠标坐标、b =div距浏览器距离、c = 鼠标在div内部距离=a-b。  
 移动：通过 a - c 建立鼠标与div的关系，防止鼠标超出div。  
  
拖拽状态 = 0鼠标在元素上按下的时候{   
 拖拽状态 = 1   
 记录下鼠标的x和y坐标   
 记录下元素的x和y坐标   
}   
  
鼠标在元素上移动的时候{   
 如果拖拽状态是0就什么也不做。   
 如果拖拽状态是1，那么   
 元素y = 现在鼠标y - 原来鼠标y + 原来元素y   
 元素x = 现在鼠标x - 原来鼠标x + 原来元素x   
}   
  
鼠标在任何时候放开的时候{   
 拖拽状态 = 0   
}

# 数据请求封装思路

# 自定义hook封装过组件吗

1.将Hello组件和App组件中共用的逻辑放在统一的自定义hook中写  
2.自定义hook,hook名以use开头  
3.其他组件通过import引入自定义hook,就可以使用了

# react和vue的区别

相同点:  
1.都使用了虚拟dom,如果需要改变任何元素的状态,先改虚拟dom,当有变化产生时,会创建新的虚拟dom,通过diff算法对比新旧虚拟dom的差异,只需要渲染差异部分就行  
2.都使用组件化  
  
不同点:  
1.react中有新语法jsx,vue用普通的html   
2.vue中父组件和子组件有通信的时候,父组件数据改变引起子组件改变,子组件会重新渲染，如果父子组件没有通信,父组件改变子组件不会渲染，react中不管是否有数据通信,父组件改变子组件都会渲染  
3.react:create-react-app  
 vue :vue-cli  
4.react中用redux管理状态,state通过setState更新  
 vue中数据由vuex管理

# react的class组件,没有class的时候怎么写

React.createClass   
我们最早使用这个方法来构建一个组件“类”，它接受一个对象为参数，  
对象中必须声明一个render方法，render返回一个组件实例

# 生命周期  钩子函数用es5怎么定义

protoType.componentDidMount

# 虚拟dom和diff算法

虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法 避免了没有必要的 dom 操作，从而提高性能  
具体实现步骤如下 用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树， 插到文档当中 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两 棵树差异把2所记录的差异应用到步骤1所构建的真正的 DOM 树上，视图就更新  
  
1.虚拟dom是什么  
 所谓的虚拟 dom，也就是虚拟节点。它通过JS的Object对象模拟DOM中的节点，然后再通过特定的render方法将其渲染成真实的DOM节点  
 用js模拟一颗dom树,放在浏览器内存中。当你要变更时,虚拟dom使用diff算法进行新旧虚拟dom的比较,将变更放到变更队列中,反应到实际的dom树,减少了dom操作。  
  
2.虚拟dom的使用基本流程（前四步骤）  
 1.获取数据  
 2.创建虚拟dom  
 3.通过render函数解析jsx，将其转换成虚拟dom结构  
 4.将虚拟dom渲染成真实dom  
 5.数据更改了  
 6.使用diff算法比对两次虚拟dom,生成patch对象  
 7.根据key将patch对象渲染到页面中改变的结构上，而其他没有改变的地方是不做任何修改的(虚拟dom的惰性原则)  
  
优点：  
保证性能下限：   
 框架的虚拟 DOM 需要适配任何上层 API 可能产生的操作，它的一些 DOM 操作的实现必须是普适的，所以它的性能并不是最优的；但是比起粗暴的 DOM 操作性能要好很多，因此框架的虚拟 DOM 至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限；  
无需手动操作 DOM：   
 我们不再需要手动去操作 DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟 DOM 和 数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率；  
跨平台：   
 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关，相比之下虚拟 DOM 可以进行更方便地跨平台操作，例如服务器渲染、weex 开发等等。  
  
缺点:  
无法进行极致优化：   
虽然虚拟 DOM + 合理的优化，足以应对绝大部分应用的性能需求，但在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化。首次渲染大量DOM时，由于多了一层虚拟DOM的计算，会比innerHTML插入慢。  
  
1.diff算法是什么  
 Diff算法是用于比较两个新旧虚拟dom树的差异的，比较完之后会得到一个差异对象，我们称之为patch补丁对象  
 比较后会出现四种情况：  
 1、此节点是否被移除 -> 添加新的节点  
 2、属性是否被改变 -> 旧属性改为新属性  
 3、文本内容被改变 -> 旧内容改为新内容  
 4、节点要被整个替换 -> 结构完全不相同 移除整个替换  
  
2.diff算法运行结束后，返回是什么  
 返回一个key

# react Native

React Native(内特服)能在手机上创建原生应用，React在这方面处于领先位置。使用JavaScript, CSS和HTML创建原生移动应用，这是一个重要的革新。Vue社区与阿里合作开发Vue版的React Native——Weex也很不错，但仍处于开发状态且并没经过实际项目的验证。  
  
既拥有Native的用户体验、又保留React的开发效率  
  
React Native与React.js的主要区别还是JSX,它使用XML标记的方式去直接声明界面，将HTML直接嵌入到JavaScript代码中

# react-router(路由)原理

react-router依赖基础 - history   
  
history是一个独立的第三方js库，可以用来兼容在不同浏览器、不同环境下对历史记录的管理  
  
老浏览器的history:   
 主要通过hash来实现，对应createHashHistory  
高版本浏览器:   
 通过html5里面的history，对应createBrowserHistory   
node环境下: (不弱惹)  
 主要存储在历史记录memeory里面，对应createMemoryHistory  
抽象了一个公共的文件createHistory:  
 此时的location跟浏览器原生的location是不相同的，最大的区别就在于里面多了key字段，  
 history内部通过key来进行location的操作  
  
原理:  
1.执行URL前进  
 createBrowserHistory: pushState、replaceState  
 createHashHistory: location.hash=\*\*\* location.replace()  
 createMemoryHistory: 在内存中进行历史记录的存储  
1.检测URL回退  
 createBrowserHistory: popstate  
 createHashHistory: hashchange  
 createMemoryHistory: 因为是在内存中操作，跟浏览器没有关系，不涉及UI层面的事情，所以可以直接进行历史信息的回退  
1.state的存储  
 为了维护state的状态，将其存储在sessionStorage里面:  
 基本原理:实现URL与UI可视化界面的同步。其中在react-router中，URL对应Location对象，而UI是由react components来决定的，这样就转变成location与components之间的同步问题。  
  
在react-router中最主要的component是Router、RouterContext、Link，history库起到了中间桥梁的作用  
  
安装react-router-dom   
 Link组件用于点击链接跳转其他页面，没有路由激活  
 NavLink 用于有路由激活效果的

# react-router-dom和react-router的区别

写法上的区别：  
import {Swtich, Route, Router, HashHistory, Link} from 'react-router-dom';  
  
import {Switch, Route, Router} from 'react-router';  
import {HashHistory, Link} from 'react-router-dom';  
  
react-router-dom:   
加入了在浏览器运行环境下的一些功能:  
 BrowserRouter和HashRouter组件，  
 前者使用pushState和popState事件构建路由，  
 后者使用window.location.hash和hashchange事件构建路由。  
  
react-router-dom是依赖于react-router的，其中Switch、Route、Router、Redirect等组件是直接引入react-router中的  
  
react-router-dom还另外新增了Link、BrowserRouter、HashRouter组件。  
  
在引入react-router-dom后不需要显性引入react-router，  
react-router-dom依赖react-router，npm都会将他们安装。  
  
react-router3.x与react-router-dom区别  
 react-router3.x版本下路由采用集中式配置，UI组件和路由是分开的。  
 react-router4.x版本下路由路由采用分散式配置，路由嵌套在UI组件当中，  
 更加契合组件化思想（组件中的路由也应该包含在组件之中）。

# react路由模式

我们一直在使用的路由方式是BrowserRouter，也就是浏览器的路由方式，其实React还有几种路由方式：  
 (不弱惹)  
1、BrowserRouter：浏览器的路由方式，也就是在开发中最常使用的路由方式  
2、HashRouter：在路径前加入#号成为一个哈希值，Hash模式的好处是，再也不会因为我们刷新而找不到我们的对应路径  
3、MemoryRouter：不存储history，所有路由过程保存在内存里，不能进行前进后退，因为地址栏没有发生任何变化  
4、NativeRouter：经常配合ReactNative使用，多用于移动端  
5、StaticRouter：设置静态路由，需要和后台服务器配合设置，比如设置服务端渲染时使用  
 (斯大推克)

# react路由配置

1、引入路由包  
 npm install --save react-router  
 npm install --save react-router-dom  
 react-router：是基本的router包，里边函的内容较多，  
 但是在网页开发中有很多用不到，现在的市面上的课程讲的基本都是这个包的教程。  
 react-router-dom：随着react生态环境的壮大，后出现的包，  
 这个包比react-router包轻巧了很多。  
2、设置路由配置文件  
 在src目录下新建一个Router/index.js文件用于管理路由，这里需要引入一些对应的组件和路由包文件。  
 Router的history是必需的props  
 Switch表示只渲染第一个与当前地址匹配的<Route>  
 Route的props path为路径，component为路径对应的页面  
 exact属性表示精确匹配,比如我们有多层路由进行嵌套时，exact可以帮助我们精确匹配到你想跳转的路由。 exact的值为bool型，为true是表示严格匹配，为false时为正常匹配  
  
3、在入口文件引入路由配置文件  
 import RouterConfig from './router/index.js';  
 ReactDOM.render(<RouterConfig/>, document.getElementById('root'));  
   
4、在各组件中使用路由  
 <ul className="menu">  
 <li><NavLink to='/Page1'>第一个页面</NavLink></li>  
 <li><NavLink to='/Page2'>第二个页面</NavLink></li>  
 </ul>  
   
1.Switch:表示一次只渲染一个组件  
2.Route:路由组件,用于展示一个组件 同router-view  
3.Redirect:重定向(锐得埃克特)  
4.lazy + Suspense(色斯盘丝):实现路由懒加载  
5.exact:路径完全匹配  
6.fallback:组件切换时候的转场组件

# react路由(router)实现异步加载

react-router，官网文档给出的是用webpack的bundle-loader  
  
require.ensure。这是webpack的旧式写法，现在已不推荐  
  
import()  
 符合ECMAScript提议的import()语法，该提案与普通 import 语句或 require 函数的类似，  
 但返回一个 Promise 对象。这意味着模块时异步加载的

# react原理

1. setState   
 setState在合成事件和钩子函数中是异步的   
 在原生事件和setTimeout中是同步的  
  
 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，  
 形式了所谓的“异步”，  
  
 第一个参数可以是对象或者函数 是更新state   
 第二个参数获取最新的state,副作用操作 dom操作事件触发声明 数据获取  
  
2. JSX语法的转化  
 JSX 仅仅是 createElement() 方法的语法糖（简化语法）  
 JSX 语法被 @babel/preset-react 插件编译为 createElement() 方法  
 react.createElement()  
 React 元素：是一个对象，用来描述你希望在屏幕上看到的内容  
  
3. 组件更新机制  
 setState() 的两个作用： 1. 修改 state 2. 更新组件（UI）  
 过程：父组件重新渲染时，也会重新渲染子组件。但只会渲染当前组件子树（当前组件及其所有子组件）   
  
4. 组件性能优化  
 减轻 state：只存储跟组件渲染相关的数据  
 避免不必要的重新渲染 : shouldComponentUpdate(nextProps, nextState)   
 通过返回值决定该组件是否重新渲染，返回 true 表示重新渲染，false 表示不重新渲染 起到优化作用  
  
5. 纯组件 PureComponent  
 PureComponent 内部自动实现了 shouldComponentUpdate 钩子，不需要手动比较   
 纯组件内部通过分别 对比 前后两次 props 和 state 的值，来决定是否重新渲染组件   
 纯组件内部的对比是 shallow compare（浅层对比）  
  
6. 虚拟 DOM 和 Diff 算法  
 数据改变视图更新  
 初次渲染时，React 会根据初始state（Model），创建一个虚拟 DOM 对象（树）。  
 根据虚拟 DOM 生成真正的 DOM，渲染到页面中。  
 当数据变化后（setState()，重新根据新的数据，创建新的虚拟DOM对象（树）。  
 与上一次得到的虚拟 DOM 对象，使用 Diff 算法 对比（找不同），生成patch补丁对象,得到需要更新的内容。  
 最终，React 只将变化的内容更新（patch）到 DOM 中，重新渲染到页面。

# react的connect高阶组件实现, 如何在全局中取得store

连接React组件与 Redux store。  
connect:connect函数的返回值是一个高阶组件,通过高阶组件来获取store中的数据   
connect底层原理:是闭包  
  
mapStateFromProps:从countReducer中解构出num数据,用来获取数据  
mapDispatchFromProps:将ActionCreators中的方法绑定到组件上,并且发送action  
  
connect调用的结果是返回一个高阶组件  
connect方法利用了合并分发的原理来帮助我们完成store内容的获取  
  
合并： 将store中的所有数据拿到手  
分发： 将我们UI需要的数据派发出去  
  
原理   
合并分发 ：合并的意思是：(我们项目中)redux的数据是集中在一处的  
分发的意思是：给的是所有数据中的分块的数据

# react的connect实现原理

首先connect之所以会成功，是因为Provider组件：  
在原应用组件上包裹一层，使原来整个应用成为Provider的子组件  
接收Redux的store作为props，通过context对象传递给子孙组件上的connect  
  
那connect做了些什么呢？  
它真正连接 Redux 和 React，它包在我们的容器组件的外一层，  
它接收上面 Provider 提供的 store 里面的 state 和 dispatch，  
传给一个构造函数，返回一个对象，以属性形式传给我们的容器组件。  
  
关于它的源码  
connect是一个高阶函数，首先传入mapStateFromProps、mapDispatchFromProps，  
然后返回一个生产Component的函数(wrapWithConnect)，  
然后再将真正的Component作为参数传入wrapWithConnect，  
这样就生产出一个经过包裹的Connect组件，该组件具有如下特点:  
  
通过props.store获取祖先Component的store  
props包括stateProps、dispatchProps、parentProps,合并在一起得到nextState，作为props传给真正的Component  
componentDidMount时，添加事件this.store.subscribe(this.handleChange)，实现页面交互  
shouldComponentUpdate时判断是否有避免进行渲染，提升页面性能，并得到nextState  
componentWillUnmount时移除注册的事件this.handleChange

# react-router3和react-router4 的区别

一、V3或者说V早期版本是把router 和 layout components 分开  
　 在V4中是：  
 集中式 router  
 通过 <Route> 嵌套，实现 Layout 和 page 嵌套  
 Layout 和 page 组件 是作为 router 的一部分  
二、在V3中，我们是将整个庞大的router直接丢给Dom  
　　在V4中，除了BrowserRouter，我们丢给DOM的我们的程序本身  
  
　　另外，V4 中，我们不再使用 {props.children} 来嵌套组件了，  
　　替代的 <Route>，当 route 匹配时，子组件会被渲染到 <Route> 书写的地方  
  
三、在V3 中的 routing 规则是 exclusive，意思就是最终只获取一个 route  
　　而 V4 中的 routes 默认是 inclusive 的，这就意味着多个 <Route>可以同时匹配和呈现  
  
　　如果只想匹配一个路由，可以使用Switch，在 <Switch> 中只有一个 <Route> 会被渲染，  
　　同时可以再在每个路由添加exact，做到精准匹配  
  
　　Redirect，浏览器重定向，当多有都不匹配的时候，进行匹配

# react中获取真实的dom节点

使用ref属性获取Dom元素后，再使用原生javascript获取内容

# react是什么层面上的框架

React框架本身和我们常用的JavaScript MVC框架，如：AngularJS，Backbone，Ember等，没有直接的可比性。  
React的官方博客中明确阐述了React不是一个MVC框架，  
而是一个用于构建组件化UI的库，是一个前端界面开发工具。所以顶多算是MVC中的V（view）。

# react数据流

React遵循从上到下的数据流向，即单向数据流。  
  
单向数据流并非‘单向绑定’，甚至单向数据流与绑定没有‘任何关系’。  
对于React来说，单向数据流（从上到下）与单一数据源这两个原则，  
  
限定了React中要想在一个组件中更新另一个组件的状态（类似于Vue的平行组件传参，或者是子组件向父组件传递参数）  
，需要进行状态提升。即将状态提升到他们最近的祖先组件中。  
  
子组件中Change了状态，触发父组件状态的变更，父组件状态的变更，  
影响到了另一个组件的显示（因为传递给另一个组件的状态变化了，这一点与Vue子组件的$emit()方法很相似）。

# react数据一定要在DidMount里获取更新的原因

componentDidMount方法中的代码，是在组件已经完全挂载到网页上才会调用被执行，所以可以保证数据的加载。  
在这方法中调用setState方法，会触发重渲染。这个方法就是用来加载外部数据用的，或处理其他的副作用代码。 静态  
  
constructor()  
 constructor()中获取数据的话，如果时间太长，或者出错，组件就渲染不出来，整个页面都没法渲染了。  
 constructor是作组件state初绐化工作，并不是设计来作加载数据的。  
  
componentWillMount()  
 如果使用SSR（服务端渲染）,componentWillMount会执行2次，一次在服务端，一次在客户端。  
 而componentDidMount不会。  
   
 constructor可以完成state初始化，componentWillMount使用的很少，目前16版本加入了UNSAFE来标识 componentWillMount，新的生命周期static getDerivedStateFromProps() 也会替代这个。  
  
 React16之后采用了Fiber架构，只有componentDidMount声明周期函数是确定被执行一次的，  
 类似ComponentWillMount的生命周期钩子都有可能执行多次，  
 所以不加以在这些生命周期中做有副作用的操作，比如请求数据之类。  
  
render()  
 无限render  
  
componentDidMount()  
 确保已经render过一次。提醒我们正确地设置初始状态，这样就不会得到导致错误的"undefined"状态。

# react中的受控组件和非受控组件

受控组件  
 HTML中的表单元素是可输入的，也就是有自己的可变状态  
 而React中可变状态通常保存在state中，并且只能通过setState()方法来修改  
 React讲state与表单元素值value绑定在一起，有state的值来控制表单元素的值  
 简单来说，值受到react控制的表单元素  
   
非受控组件  
 调用 React.createRef() 方法创建ref对象  
 将创建好的 ref 对象添加到文本框中  
 通过ref对象获取到文本框的值  
 简单来说，表单组件没有value prop就可以称为非受控组件

# react中的ref的3种方式

方式1: string类型绑定  
类似于vue中的ref绑定方式，可以通过this.refs.绑定的ref的名字获取到节点dom  
注意的是 这种方式已经不被最新版的react推荐使用，有可能会在未来版本中遗弃  
   
方式2: react.CreateRef()  
通过在class中使用React.createRef()方法创建一些变量，可以将这些变量绑定到标签的ref中  
那么该变量的current则指向绑定的标签dom  
   
方式3: 函数形式  
在class中声明函数，在函数中绑定ref  
使用这种方法可以将子组件暴露给父组件以使得父组件能够调用子组件的方法  
通过函数的方法绑定ref可以将整个子组件暴露给父组件  
  
当在子组件中调用onRef函数时，正在调用从父组件传递的函数。this.props.onRef（this）这里的参数指向子组件本身，父组件接收该引用作为第一个参数：onRef = {ref =>（this.child = ref）}然后它使用this.child保存引用。之后，可以在父组件内访问整个子组件实例，并且可以调用子组件函数。

# react中的static(静态方法)

静态方法和 React 没有直接关系，React 的组件都是继承自 React.Component 这个类，静态方法属于类本身。  
  
static并不是react定义的，而加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用。  
  
这里涉及到了ES6的class，我们定义一个组件的时候通常是定义了一个类，  
而static则是创建了一个属于这个类的属性或者方法。  
组件则是这个类的一个实例，component的props和state是属于这个实例的，

# react的props与state的区别

props:  
一般用于父组件向子组件通信，在组件之间通信使用。  
state:   
一般用于组件内部的状态维护，更新组建内部的数据，状态，更新子组件的props等。

# 混合开发

1. 混合开发是介于webapp和原生app之间的一种应用，它同时具有webapp可以跨平台的特性，也具备原生app可以进行安装使用的特性  
2. webApp   
 1. 优点： 跨平台 、 维护更新、项目迭代快  
 2. 缺点： 交互体验不好 、进入应用的方式麻烦  
3. 原生app  
 1.案例： 美团、饿了吗、微信、QQ   
 2.安装在手机中使用的   
 3.优点： 手机安装、交互体验好   
 4.缺点： 维护更新、项目迭代很慢、成本太高了  
4. 混合开发： 择中方案  
 跨平台  
 维护更新快 成本低  
 手机安装，交互体验好  
  
为什么混合开发会兴起呢？  
1. 混合开发的兴起是偶然的  
  
混合开发方式  
1. h5 主导  
 - 开发思维： H5【 vue/ react/ angular 】 + 第三方可以访问原生设备的库  
 - 微信公众号： h5网页 + 微信JSSDK   
 - 历史  
 1. PhoneGap + cordva.js 淘汰  
 2. vue/react/angular + ioinc.js   
 3. vue - uni-app   
 4. h5 + h5plus.js h5 + h5+  
 5. 微信公众号： webapp + js-jdk   
 6. vue/react + weex.js[ 阿里内部 ]  
2. React Native  
 1. facebook 团队项目 16 - 18年很热火，18年后半年开始热度下降，  
 facebook觉得这个框架对于开发者而言开发难度太大，维护成本也高，Facebook决定不再更新它了、  
 2. 典型应用： 饿了吗  
 3. React Native开发出来项目 - 原生app   
 4. React Native = React + 原生js  
 5. 构建React Native项目 - 脚手架   
 1. 构建项目环境: create-react-native-app   
 2. 手机调试：expo expo-cli  
 3. 目录解释  
 1. \_\_test\_\_ expo手机调试的测试文件夹，不用管  
 2. .expo 临时文件，运行项目  
 3. .expo-shared 分享  
 4. assets 静态资源  
 5. components 公共组件  
 6. constants 项目公用的常量  
 7. navigation 底部tabbar栏组件  
 8. node\_modules 依赖包  
 9. screens 页面

# 小程序

对于路由的触发方式以及页面生命周期函数如下:

|  |  |  |  |
| --- | --- | --- | --- |
| 路由方式 | 触发时机 | 路由前页面 | 路由后页面 |
| 初始化 | 小程序打开的第一个页面 |  | onLoad, onSHow |
| 打开新页面 | 调用 API wx.navigateTo 或使用组件 | onHide | onLoad, onShow |
| 页面重定向 | 调用 API wx.redirectTo 或使用组件 | onUnload | onLoad, onShow |
| 页面返回 | 调用 API wx.navigateBack 或使用组件或用户按左上角返回按钮 | onUnload | onShow |
| Tab 切换 | 调用 API wx.navigateBack 或使用组件或用户按调用 API wx.switchTab 或使用组件  或用户切换 Tab |  | 各种情况请参考下表 |
| 重启动 | 调用 API wx.reLaunch 或使用组件 | onUnload | onLoad, onShow |

页面跳转触发的生命周期，其实还是存在问题的，并非官方所说的那样。

SwitchTab的跳转BUG  
首页跳转到子页面后，在子页面上使用：  
<navigator type='switchTab' url="/pages/index/index" >  
 <view>跳转首页</button>  
</navigator>  
这种方式有问题，解决的办法是通过JS来实现跳转，代码如下：  
<view class="weui-btn-area">  
 <button class="weui-btn" bindtap="backIndex" type="default">返回主页</button>   
 </view>  
跳转成功后，重新调用onload方法，JS代码如下：  
backIndex:function(){  
 wx.switchTab({  
 url: '/pages/index/index',  
 success: function (e) {  
 var page = getCurrentPages().pop();  
 if (page == undefined || page == null) return;  
 page.onLoad();  
 }  
 })  
 }

# get与post的区别

1.提交方式  
get: get会将接收到的数据拼接到url地址中,以"?"问号划分,问号后面是接收到的数据,多个数据之间用&连接。用户可以很直观的看见。  
post: post会将接收到的数据放置在html header中一起发送到指定的url地址内。用户看不到这个过程。  
  
2.传递数据大小  
get: get传递数据的大小因为受到浏览器地址栏的限制,所以一般在2k-8k,这要据浏览器而定,比如谷歌浏览器就是8k。  
post: post传递数据的大小最小是2M,但理论上是无上限的。  
  
3.应用范围  
get: get一般用于获取/查询资源信息.多用于a标签的href属性中,也常用于location.href属性中。  
post: post一般是用于更新数据信息.多用于表单提交。  
  
4.安全性  
get的安全性比post较差。

# react父组件通过ref获取不到子组件的解决方案

在子组件中  
this.customfunction = this.customfunction.bind(this);  
即可

# React PureComponent 和 Component 区别

React.PureComponent 与 React.Component 几乎完全相同，但 React.PureComponent 通过prop和state的浅对比来实现 shouldComponentUpate()。  
　　如果React组件的 render() 函数在给定相同的props和state下渲染为相同的结果，在某些场景下你可以使用 React.PureComponent 来提升性能。  
　　React.PureComponent 的 shouldComponentUpdate() 只会对对象进行浅对比。如果对象包含复杂的数据结构，它可能会因深层的数据不一致而产生错误的否定判断(表现为对象深层的数据已改变视图却没有更新, 原文：false-negatives)。当你期望只拥有简单的props和state时，才去继承 PureComponent ，或者在你知道深层的数据结构已经发生改变时使用 forceUpate() 。或者，考虑使用 不可变对象 来促进嵌套数据的快速比较。  
　　此外,React.PureComponent 的 shouldComponentUpate() 会忽略整个组件的子级。请确保所有的子级组件也是”Pure”的。  
　　  
PureComponent的作用：  
 PureComponent 其实是在内部帮我们简单实现了一下shouldComponentUpdate的功能，以便提供组件的性能；这里的简单指是：对prop和state做浅比较，若浅比较结果相同，则该组件以及其子组件不做render；否则，render。  
  
使用PureComponent注意事项：  
 PureComponent主要针对prop和state为基本数据类型，如bool、string、number；  
 对于数组和对象等引用类型，则要引用不同，才会渲染；如果引用相同，则PureComponent浅比较返回结果相同，不做render；  
 PureComponent 中不建议再另外重写shouldComponentUpdate方法，否则会报warning信息：  
 PureComponent的最好作为展示组件，如果prop和state每次都会变，PureComponent做浅比较也会影响性能，可以考虑直接用Component；  
 对于prop和state数据结构比较复杂的情况，可以考虑自己重写shouldComponentUpdate方法来做优化；

# react中调用setState之后发生

React会将当前传入的参数对象与组件当前的状态合并,然后触发调和过程,在调和的过程中,React会以相对高效的方式根据新的状态构建React元素树并且重新渲染整个UI界面.  
  
React得到的元素树之后,React会自动计算出新的树与老的树的节点的差异,然后根据差异对界面进行最小化的渲染,在React的差异算法中,React能够精确的知道在哪些位置发生看改变以及应该如何去改变,这样就保证了UI是按需更新的而不是重新渲染整个界面

# SPA和MPA的异同

单页应用（SinglePage Application，SPA）  
 指只有一个主页面的应用，一开始只需加载一次 js,css 等相关资源。所有的内容都包含在主页面，对每一个功能模块组件化。单页应用跳转，就是切换相关组件，仅刷新局部资源。  
  
多页应用（MultiPage Application，MPA）  
 指有多个独立的页面的应用，每个页面必须重复加载 js,css 等相关资源。多页应用跳转，需要整页资源刷新。  
   
两者对比表格：  
 SPA ||| MPA  
结构:一个主页面 + 许多模块的组件 ||| 许多完整的页面  
体验:页面切换快，体验佳；当初次加载文件过多时，需要做相关的调优。 ||| 页面切换慢，网速慢的时候，体验尤其不好  
资源文件:组件公用的资源只需要加载一次 ||| 每个页面都要自己加载公用的资源  
适用场景:对体验度和流畅度有较高要求的应用，不利于 SEO（可借助 SSR 优化 SEO） ||| 适用于对 SEO 要求较高的应用  
过渡动画:Vue 提供了 transition 的封装组件，容易实现 ||| 很难实现  
内容更新:相关组件的切换，即局部更新 ||| 整体 HTML 的切换，费钱（重复 HTTP 请求）  
路由模式:可以使用 hash ，也可以使用 history ||| 普通链接跳转  
数据传递:因为单页面，使用全局变量就好（Vuex） ||| cookie 、localStorage 等缓存方案，URL 参数，调用接口保存等  
相关成本:前期开发成本较高，后期维护较为容易 ||| 前期开发成本低，后期维护就比较麻烦，因为可能一个功能需要改很多地方  
  
单页应用实现核心：前端路由  
前端路由的核心：改变视图的同时不会向后端发出请求。

# TS

TS好处：  
1、强类型   
2、不需要去浏览器中浏览效果，就能知道编译错误  
 静态类型检查可以做到early fail，即你编写的代码即使没有被执行到，一旦你编写代码时发生类型不匹配，  
 语言在编译阶段（解释执行也一样，可以在运行前）即可发现  
4、类型就是最好的注释，看类型我们就知道这个是什么  
3、即使ts中有编译报错，tsc依旧可以将其编译成js  
  
学习TS的基本数据类型  
定类型是为了安全，规矩多就安全  
1、基础数据类型 ：number \string\boolean\null\undefined  
 any 表示任意类型 void 表示空类型，空类型是针对函数的，表示函数没有返回值。返回空  
2、内置对象类型 ： Array \ Boolean \ HTMLElement  
3、自定义类型 : 接口 类 泛型 枚举类型

# TS 中 interface 和 type 的区别

在ts中，定义类型由两种方式：接口（interface）和类型别名（type alias）  
interface只能定义对象类型，  
type声明的方式可以定义组合类型，交叉类型和原始类型  
   
如果用type alias 声明的方式，会导致一些功能的缺失  
 1.interface方式可以实现接口的extends/implements，而type 不行  
 2.interface可以实现接口的merge，但是type不行

# Webpack

webpack打包react vue,与自己的源代码分离使用splitchunks  
webpack热更新:不用刷新浏览器而将新变更的模块替换掉旧的模块。  
  
webpack会分析每个入口文件，解析包依赖关系的各个文件，每个模块都打包到bundle.js。webpack给每个模块分配一个唯一的ID并通过这个ID索引和访问模块。页面运行时，先启动entry.js，其他模块会在运行require时候执行。  
  
1.Webpack Loader Plugin  
  
（loader ，plugin分别什么作用，哪个配置可以把依赖包抽离出来，不打包进去）  
【Loader】：用于对模块源码的转换，loader描述了webpack如何处理非javascript模块，并且在buld中引入这些依赖。loader可以将文件从不同的语言（如TypeScript）转换为JavaScript，或者将内联图像转换为data URL。比如说：CSS-Loader，Style-Loader等。babel-loader优雅降级配置ES高版本转成低版本  
  
【Plugin】：是用于在webpack打包编译过程里，在对应的事件节点里执行自定义操作，比如资源管理、bundle文件优化等操作,  
  
依赖包抽离 const ExtractTextWebapckPlugin= require("extract-text-webpack-plugin") module exclude node\_modules  
  
排除excloude排除node\_modules  
  
loader的使用很简单：  
  
在webpack.config.js中指定loader。module.rules可以指定多个loader，对项目中的各个loader有个全局概览。  
  
loader是运行在NodeJS中，可以用options对象进行配置。plugin可以为loader带来更多特性。loader可以进行压缩，打包，语言翻译等等。  
  
loader从模板路径解析，npm install node\_modules。也可以自定义loader，命名XXX-loader。  
  
语言类的处理器loader：CoffeeScript，TypeScript，ESNext（Bable）,Sass,Less,Stylus。任何开发技术栈都可以使用webpack。  
  
webpack常用的loader  
  
样式：style-loader、css-loader、less-loader、sass-loader等  
文件：raw-loader、file-loader 、url-loader等  
编译：babel-loader、coffee-loader 、ts-loader等  
校验测试：mocha-loader、jshint-loader 、eslint-loader等  
  
目的在于解决loader无法实现的其他事，从打包优化和压缩，到重新定义环境变量，功能强大到可以用来处理各种各样的任务。webpack提供了很多开箱即用的插件：CommonChunkPlugin主要用于提取第三方库和公共模块，避免首屏加载的bundle文件，或者按需加载的bundle文件体积过大，导致加载时间过长，是一把优化的利器。而在多页面应用中，更是能够为每个页面间的应用程序共享代码创建bundle。  
  
webpack功能强大，难点在于它的配置文件，webpack4默认不需要配置文件，可以通过mode选项为webpack指定了一些默认的配置，mode分为：development/production，默认是production。  
  
插件可以携带参数，所以在plugins属性传入new实例。  
  
webpack常用的plugin  
webpack内置UglifyJsPlugin，压缩和混淆代码。  
webpack内置CommonsChunkPlugin，提高打包效率，将第三方库和业务代码分开打包。  
ProvidePlugin：自动加载模块，代替require和import  
  
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extract-text-webpack-plugin 将js文件中引用的样式单独抽离成css文件  
  
DefinePlugin 编译时配置全局变量，这对开发模式和发布模式的构建允许不同的行为非常有用。  
  
【Mode】可以在config文件里面配置，也可以在CLI参数中配置：webpack--mode=production（一般会选择在CLI，也就是npm scripts里面进行配置）。  
  
在webpack4以下版本，webpack3.XX，通过plugins进行环境变量的配置。  
  
【resolve】模块，resolver是个库，帮助webpack找到bundle需要引入的模块代码，打包时，webpack使用enhanced-resolve来解析路径。   
  
2.we 优化  
多进程打包 安装插件thread-loader parallel-webpack HappyPack  
  
多进程压缩 parallel-uglify-plugin terser-webpack-plugin  
资源CDN 公用代码提取，使用 CDN 加载  
动态polyfill 动态 polyfill 指的是根据不同的浏览器，动态载入需要的 polyfill

# flex布局

flex是由 flex-grow:1父容器在主轴上还有多少剩余空间，  
  
flex-shrink当父元素的宽度大于所有子元素的宽度的和时（即父元素会有剩余空间），  
子元素如何分配父元素的剩余空间  
  
flex-basis基准值组成  
  
flex 怎么实现1个盒子垂直居中 如何实现四个盒子水平均匀分布  
  
父容器 display:flex; justify-content: center; align-items: center;  
  
display: flex; justify-content:space-evenly; align-items: center;

# 实现导航的吸顶效果

设置css:fixed固定定位   
  
判断滚动条滚动的距离大于导航条距顶部的距离,来判断是否实现吸顶,然后addClass添加样式

# less的hover简写

a{  
 &:hover {} //这里&代表它的上一级就是a  
}

# CSS伪类 ::after,before的应用场景

::before和::after必须配合content属性来使用，  
content用来定义插入的内容，content必须有值，至少是空。  
默认情况下，伪类元素的display是默认值inline，可以通过设置display:block来改变其显示。  
  
1.清除浮动:  
在浮动元素后面添加一个空的Div标签，然后在设置它的清除浮动要是，使用after伪元素  
  
2.常见消息框 :   
伪类content:' ' 伪类4条边必须宽度相同，而且其他三条边为transparent   
可以通过设置定位元素left,top值为50%,translate(-50%,-50%) 来使任意宽高的元素居中。  
  
div::before{  
 content:' ';  
  
3.阴影 : 通过设置before，after不同位置，不同旋转角度,要保证伪类的颜色及z-index  
  
div.outer::before,div.outer::after{content:'';  
 z-index:1;  
 width:50%;  
 height:3px;  
 position:absolute;  
 left:10px;  
 bottom:7px;  
 background-color:transparent;  
 box-shadow:5px 5px 10px rgba(0,0,0,0.5);  
 -webkit-transform:rotate(-3deg);  
  
4.做出各种图形效果  
  
#star-five:before {  
 border-bottom: 80px solid red;  
 border-left: 30px solid transparent;  
 border-right: 30px solid transparent;  
 position: absolute;  
 height: 0;  
 width: 0;  
 top: -45px;  
 left: -65px;  
 content: '';  
 transform: rotate(-35deg);  
 }  
 #star-five:after {  
 width: 0;  
 height: 0;  
 border-left: 100px solid transparent;  
 border-right: 100px solid transparent;  
 border-bottom: 70px solid yellow;  
 top: 7px;  
 left: -110px;  
 position: absolute;  
 display: block;  
 content: '';  
 transform: rotate(-70deg);  
 }

# CSS权重的计算

权重叠加 0,0,0,5 + 0,0,0,5 =0,0,0,10 而不是 0,0,1,0，所以不会存在10个div能赶上一个类选择器的情况  
  
继承的权重是0   
1)如果选中了，那么以上面的公式来计权重。谁大听谁的。  
2)如果没有选中，那么权重是0，因为继承的权重为0  
  
当选择器冲突时，权重高的生效；当权重相同时，写在后头的会把前面的覆盖。

# http

http2.0，https连接，  
  
http概述:超文本传输协议,是互联网上应用最为广泛的一种网络协议  
  
http的缺点  
1.通信使用明文可能会被窃听。  
2.不验证通信方的身份可能遭遇伪装。  
3.无法证明报文的完整性，可能已遭篡改。  
  
http的端口号是80；  
https的端口号是443；  
  
https就是在安全的传输层上发送的http。它在将http报文发送给TCP之前，  
先将其发送给了一个安全层 ，  
对其进行加密。http安全层是通过ssl及其现代替代协议TSL来实现的。  
  
https的优点  
（1）使用HTTPS协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；  
（2）HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，  
要比http协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。  
  
https的缺点  
但是https因为加了层ssl，所以在效率方面比较低，会使页面加载的时长延长近50%,也会增加10-20%的耗电。  
需要安装证书，在一定基础上增加部署费用，并且报文加密解密对数据传递有一点的效率影响。

# HTTP1.0和HTTP2.0的区别

1. http/2.0的目标是改善用户加载页面的时候更快

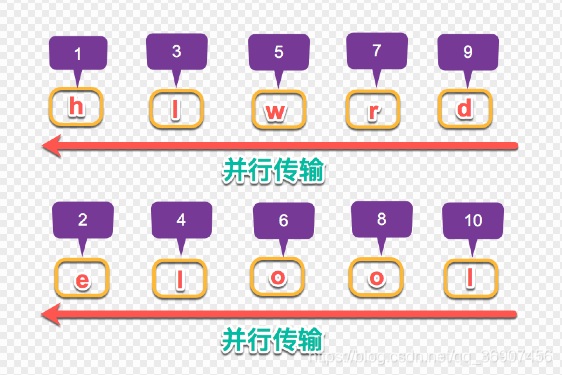
1. HTTP/2采用二进制格式而非文本格式

1. HTTP/2是完全多路复用的，而非有序并阻塞的——只需一个连接即可实现并行

http1.1在传输消息的时候，1.0通常都要传输几次消息，一次无法给浏览器进行渲染，但是每一次传输的时候客户端都需要重新请求建立链接, 常说的3次握手4次挥手 http1.1通过引入长连接和流水线技术处理了这个问题，就是让tcp连接一直开着 直到客户端通知关闭了在关闭，这样再次传输数据的时候就不用了重新建立连接了，但是他还是需要按照顺序请求和响应的，也就是如果第一个请求没有得到响应就会造成后面的请求堵塞，引发队头堵塞问题，虽然可以建立多个tcp连接 ，但是会浪费很多的资源。比如需要传输：hello world，只能从h到d一个一个的传输，不能并行传输，因为接收端并不知道这些字符的顺序，所以并行传输在HTTP1.1是不能实现的



HTTP/2引入二进制数据帧和流的概念，其中帧对数据进行顺序标识，如下图所示，这样浏览器收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况。同样是因为有了序列，服务器就可以并行的传输数据，这就是流所做的事情。



————————————————

版权声明：本文为CSDN博主「工程师-小白」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/qq_36907456/article/details/117135897>

# http对称加密非对称加密

1.对称密钥加密是指加密和解密使用同一个密钥的方式,一方通过密钥将信息加密后，  
把密文传给另一方，另一方通过这个相同的密钥将密文解密，转换成可以理解的明文  
  
2.非对称加密是加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。  
指使用一对非对称密钥，即公钥和私钥，公钥可以随意发布，但私钥只有自己知道。  
发送密文的一方使用对方的公钥进行加密处理，对方接收到加密信息后，使用自己的私钥进行解密。

# TCP和UDP是什么

### TCP：

传输控制协议（TCP，Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC 793 定义。

### UDP：

Internet 协议集支持一个无连接的传输协议，该协议称为用户数据报协议（UDP，User Datagram Protocol）。UDP 为应用程序提供了一种无需建立连接就可以发送封装的 IP 数据包的方法。RFC 768 描述了 UDP。

### TCP和UDP有什么区别?

1.TCP和UDP区别总结

TCP面向连接，通过三次握手建立连接，四次挥手接除连接;UDP是无连接的，即发送数据之前不需要建立连接，这种方式为UDP带来了高效的传输效率，但也导致无法确保数据的发送成功。

TCP是可靠的通信方式。通过TCP连接传送的数据，TCP通过超时重传、 数据校验等方式来确保数据无差错，不丢失，不重复，且按序到达；而UDP由于无需连接的原因，将会以最大速度进行传输，但不保证可靠交付，也就是会出现丢失、重复等等问题。

TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流，由于连接的问题，当网络出现波动时，连接可能出现响应问题；UDP是面向报文的，UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低。

每一条TCP连接只能是点到点的；而UDP不建立连接，所以可以支持一对一，一对多，多对一和多对多的交互通信，也就是可以同时接受多个人的包。

TCP需要建立连接，首部开销20字节相比8个字节的UDP显得比较大。

TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道。

# WebSocket和http有什么区别

socket是传输控制层协议，webSocket是应用层协议  
  
WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。  
一开始的握手需要借助HTTP请求完成。  
  
HTTP请求缺点：  
会导致过多不必要的请求，浪费流量和服务器资源，每一次请求、应答，都浪费了一定流量在相同的头部信息上  
  
然而WebSocket的出现可以弥补这一缺点。  
在WebSocket中，只需要服务器和浏览器通过HTTP协议进行一个握手的动作，  
然后单独建立一条TCP的通信通道进行数据的传送。  
  
原理:(webSocket)  
WebSocket同HTTP一样也是应用层的协议，但是它是一种双向通信协议，是建立在TCP之上的。  
  
1. 浏览器、服务器建立TCP连接，三次握手。这是通信的基础，传输控制层，若失败后续都不执行。  
2. TCP连接成功后，浏览器通过HTTP协议向服务器传送WebSocket支持的版本号等信息。（开始前的HTTP握手）  
3. 服务器收到客户端的握手请求后，同样采用HTTP协议回馈数据。  
4. 当收到了连接成功的消息后，通过TCP通道进行传输通信。  
  
WebSocket与HTTP的关系  
相同点:  
都是一样基于TCP的，都是可靠性传输协议。  
都是应用层协议  
  
不同点:  
WebSocket是双向通信协议，模拟Socket协议，可以双向发送或接受信息。HTTP是单向的。  
WebSocket是需要握手进行建立连接的。

# 错误调试工具

F12 断点 错误附近输出打印 火狐中的firebug IE开发者工具 Emmet

# http状态码

200 OK 请求成功。一般用于GET与POST请求  
3开头 重定向  
300 多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择  
301 永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替  
302 临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI  
303 查看其它地址。与301类似。使用GET和POST请求查看  
304 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源  
305 使用代理。所请求的资源必须通过代理访问  
306 已经被废弃的HTTP状态码  
307 临时重定向。与302类似。使用GET请求重定向  
400 客户端请求的语法错误，服务器无法理解  
401 请求要求用户的身份认证  
402 保留，将来使用  
403 服务器理解请求客户端的请求，但是拒绝执行此请求  
404 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面  
405 客户端请求中的方法被禁止  
406 N服务器无法根据客户端请求的内容特性完成请求  
407 请求要求代理的身份认证，与401类似，但请求者应当使用代理进行授权  
408 服务器等待客户端发送的请求时间过长，超时  
409 服务器完成客户端的 PUT 请求时可能返回此代码，服务器处理请求时发生了冲突  
410 客户端请求的资源已经不存在。410不同于404，如果资源以前有现在被永久删除了可使用410代码，网站设计人员可通过301代码指定资源的新位置  
411 服务器无法处理客户端发送的不带Content-Length的请求信息  
412 客户端请求信息的先决条件错误  
413 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个Retry-After的响应信息  
414 请求的URI过长（URI通常为网址），服务器无法处理  
415 服务器无法处理请求附带的媒体格式  
416 客户端请求的范围无效  
417 服务器无法满足Expect的请求头信息  
500 服务器内部错误，无法完成请求  
501 服务器不支持请求的功能，无法完成请求  
502 作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应  
503 由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中  
504 充当网关或代理的服务器，未及时从远端服务器获取请求  
505 服务器不支持请求的HTTP协议的版本，无法完成处理

# HTTP和HTTPS的区别

HTTP是超文本传输协议，信息是明文传输的，HTTPS是具有ssl/tls加密传输协议。  
  
默认端口不同，前者是80，后者是443。  
  
HTTPS比HTTP安全  
  
HTTPS协议需要到CA申请证书，需要一定费用

# 浏览器的缓存机制

什么是浏览器缓存   
 Web缓存是指一个Web资源（如html页面，图片，js，数据等）存在于Web服务器和客户端（浏览器）之间的副本。缓存会根据进来的请求保存输出内容的副本；当下一个请求来到的时候，如果是相同的URL，缓存会根据缓存机制决定是直接使用副本响应访问请求，还是向源服务器再次发送请求。比较常见的就是浏览器会缓存访问过网站的网页，当再次访问这个URL地址的时候，如果网页没有更新，就不会再次下载网页，而是直接使用本地缓存的网页。只有当网站明确标识资源已经更新，浏览器才会再次下载网页。浏览器和网站服务器是根据缓存机制进行缓存的  
  
非HTTP协议定义的缓存机制  
 浏览器缓存机制，其实主要就是HTTP协议定义的缓存机制（如： Expires； Cache-control等）。但是也有非HTTP协议定义的缓存机制，如使用HTML Meta 标签，Web开发者可以在HTML页面的<head>节点中加入<meta>标签  
   
 <meta http-equiv="Pragma" content="no-cache">  
 上述代码的作用是告诉浏览器当前页面不被缓存，每次访问都需要去服务器拉取。使用上很简单，但只有部分浏览器可以支持，而且所有缓存代理服务器都不支持，因为代理不解析HTML内容本身。  
   
浏览器在第一次请求发生后，再次请求时：  
 浏览器会先获取该资源缓存的header信息，根据其中的expires和cache-control判断是否命中强缓存），若命中则直接从缓存中获取资源，包括缓存的header信息，本次请求不会与服务器进行通信；   
 如果没有命中强缓存，浏览器会发送请求到服务器，该请求会携带第一次请求返回的有关缓存的header字段信息（Last-Modified/IF-Modified-Since、Etag/IF-None-Match）,由服务器根据请求中的相关header信息来对比结果是否命中协商缓存，若命中，则服务器返回新的响应header信息更新缓存中的对应header信息，但是并不返回资源内容，它会告知浏览器可以直接从缓存获取；否则返回最新的资源内容

# 浏览器的强缓存和协商缓存

这里说的缓存是指浏览器（客户端）在本地磁盘中对访问过的资源保存的副本文件。  
  
浏览器缓存主要有以下几个优点：  
1. 减少重复数据请求，避免通过网络再次加载资源，节省流量。  
2. 降低服务器的压力，提升网站性能。  
3. 加快客户端加载网页的速度，提升用户体验。  
  
浏览器缓存分为强缓存和协商缓存，两者有两个比较明显的区别：  
1. 如果浏览器命中强缓存，则不需要给服务器发请求；而协商缓存最终由服务器来决定是否使用缓存，即客户端与服务器之间存在一次通信。  
2. 在chrome中强缓存（虽然没有发出真实的http请求）的请求状态码返回是200(from cache)；而协商缓存如果命中走缓存的话，请求的状态码是304(not modified)。不同浏览器的策略不同，在Fire Fox中，from cache状态码是304.  
  
强缓存  
强缓存是利用http的返回头中的Expires或者Cache-Control两个字段来控制的，用来表示资源的缓存时间。  
Expires:  
 该字段是http1.0时的规范，它的值为一个绝对时间的GMT格式的时间字符串，比如Expires:Mon,18 Oct 2066 23:59:59 GMT。这个时间代表着这个资源的失效时间，在此时间之前，即命中缓存。这种方式有一个明显的缺点，由于失效时间是一个绝对时间，所以当服务器与客户端时间偏差较大时，就会导致缓存混乱。  
Cache-Control:  
 Cache-Control是http1.1时出现的header信息，主要是利用该字段的max-age值来进行判断，它是一个相对时间，例如Cache-Control:max-age=3600，代表着资源的有效期是3600秒。cache-control除了该字段外，还有下面几个比较常用的设置值：  
   
no-cache：不使用本地缓存。需要使用缓存协商，先与服务器确认返回的响应是否被更改，  
 如果之前的响应中存ETag，那么请求的时候会与服务端验证，如果资源未被更改，则可以避免重新下载。  
no-store：直接禁止游览器缓存数据，每次用户请求该资源，都会向服务器发送一个请求，每次都会下载完整的资源。  
public：可以被所有的用户缓存，包括终端用户和CDN等中间代理服务器。  
private：只能被终端用户的浏览器缓存，不允许CDN等中继缓存服务器对其缓存。  
  
Cache-Control与Expires可以在服务端配置同时启用，同时启用的时候Cache-Control优先级高。  
  
为什么要有Etag  
HTTP1.1中Etag的出现主要是为了解决几个Last-Modified比较难解决的问题：  
   
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)；  
3.某些服务器不能精确的得到文件的最后修改时间。  
   
Last-Modified与ETag是可以一起使用的，服务器会优先验证ETag，一致的情况下，才会继续比对Last-Modified，最后才决定是否返回304。  
  
200 OK（from cache）与304 Not Modified的区别  
 200 OK( from cache )不向服务器发送请求，直接使用本地缓存文件。  
 304 Not Modified则向服务器询问，若服务器认为浏览器的缓存版本还可用，那么便会返回304。  
  
缓存类型 获取资源形式 状态码 发送请求到服务器  
强缓存 从缓存取 200（from cache） 否，直接从缓存取  
协商缓存 从缓存取 304（Not Modified） 否，通过服务器来告知缓存是否可用  
  
用户行为对缓存的影响  
用户操作 Expires/Cache-Control Last-Modied/Etag  
地址栏回车 有效 有效  
页面链接跳转 有效 有效  
新开窗口 有效 有效  
前进回退 有效 有效  
F5刷新 无效 有效  
Ctrl+F5强制刷新 无效 无效

# 不能被缓存的请求

1. HTTP信息头中包含Cache-Control:no-cache，pragma:no-cache（HTTP1.0），  
 或Cache-Control:max-age=0等告诉浏览器不用缓存的请求  
2. 需要根据Cookie，认证信息等决定输入内容的动态请求是不能被缓存的  
3. 经过HTTPS安全加密的请求  
4. POST请求无法被缓存  
5. HTTP响应头中不包含Last-Modified/Etag，也不包含Cache-Control/Expires的请求无法被缓存

# 用户行为对缓存的影响

qq、fire fox 、safari 、chrome 这几个浏览器的访问同一个页面，不同的浏览器在 F5 刷新的时候 ，同一个文件 qq 、fire fox 浏览器会返回 `304 Not Nodified`，在请求头中不携带 `Expires/Cache-Control`； 而 chrome 和 safari 刷新的时候，会返回 `200 from cache`， 没有真正发起请求，走强缓存。可见不同的浏览器反馈是不一致的，所以下面表格中"F5刷新"时 `Expires/Cache-Control` 会无效我认为是存在一定争议的。而 Ctrl + F5 强制刷新的时候，会暂时禁用强缓存和协商缓存。

# eslint规则

Eslint 是一个JavaScript验证工具,有了它可以让你的编辑器像ide一样进行一些静态的错误提示功能.  
  
npm install eslint -g  
  
某些文件关闭eslint检测   
 在文件的最顶端加上注释 /\*eslint-disable\*/  
  
关闭某一行代码的eslint检查   
 // eslint-disable-next-line  
  
.eslintrc.json配置rules选项

# git

（版本回退是什么命令，哪个命令查看已删除的提交commitId）  
  
git-reset 版本回退  
  
git reset --hard xxx 回到上一个版本  
  
git reset --soft xxx 该命令将最近一次提交节点的提交记录回退到暂存区  
  
git reset --mixed xxx 是将最近一次提交节点记录回退到工作区  
  
git log 与 git reflog 查看历史记录（被删除的历史commit ID）  
  
git场景问题  
提交暂存区git add 出错 git reset HEAD <文件名> 回退  
  
提交本地仓库   
git commit出错:   
1.更改 commit 信息:git commit --amend -m“新提交消息”   
2.漏提交:git add missed-file // missed-file 为遗漏提交文件  
 git commit --amend --no-edit //--no-edit提交消息不会更改  
3.git reset --hard commit\_id git log查看提交的版本  
  
git revert 是提交一个新的版本  
git fetch 将远程主机的更新全部放到本地中  
  
git revert 和 git reset 的区别:  
  
(1)git revert是用一次新的commit来回滚之前的commit，git reset是直接删除指定的commit。  
(2)git revert是用一次逆向的commit“中和”之前的提交 ,合并的时候回滚的变化不会出现  
  
git reset是之间把某些commit在某个branch上删除,合并时候回滚的commit会被引入

# node.js

1.Node不是一个后端语言，但是它可以做类似后端语言的功能  
2.Node是使用谷歌V8引擎  
3.Node是js的一个运行环境  
4.Node具有非阻塞I/O 特点  
5.Node采用了Common.js规范   
  
node+koa2 node+express  
  
用于快速构建Node.js项目  
  
node.js 是一个基于 Chrome V8 引擎的 JavaScirpt 运行环境。  
  
Node.js使用了一个事件驱动、非阻塞式I/O的模型,使其轻量又高效\*\*  
  
Node.js基于commonjs规范  
  
事件驱动： 任务执行，发布者，订阅者，事件驱动 .  
异步（非阻塞）： 执行某一个任务的同时也可以执行其他任务  
同步（阻塞）： 执行某一个任务，这个任务如果没有执行完成，其他任务必须等待  
I/O: 输入/输出（ 数据库操作，文件系统操作等 ） - 服务器的环境  
非阻塞I/O模型： 当使用Node.js来实现数据库操作、文件系统等操作时，要进行的异步操作，异步操作的核心传统实现方式就是回调函数和事件。  
  
Node.js的包管理工具npm  
  
优点：Node.js 使用了一个事件驱动、非阻塞式 I/O 的模型，异步编程，使其轻量又高效。  
缺点：单进程，单线程，只支持单核cpu，不能充分的利用多核cpu服务器。一旦这个进程崩掉，那么整个web服务就崩掉了。  
  
内置模块 http 是用于创建一个能够处理和响应 http 响应的服务  
  
fs 用于对系统文件及目录进行读写操作。  
path 提供了一些用于处理文件路径的小工具  
Url：帮助我们对提交上来的url进行解析处理  
querystring 提供用于解析和格式化 URL 查询字符串的工具。qs.parse() qs.stringify()

# TS语法

TypeScript具有类型系统，且是JavaScript的超集。它可以编译成普通的JavaScript代码。   
TypeScript支持任意浏览器，任意环境，任意系统并且是开源的。  
  
npm install -g typescript  
  
tsc --init 生成tsconfig.json配置文件  
  
基础数据类型  
number、string、boolean、null 、undefined  
any 表示任意类型  
void 表示空类型  
内置对象类型  
 Array Boolean  
 HTMLElement  
 HTMLDivElement  
自定义类型  
接口 interface  
类  
泛型 未来的类型定义的时候不知道是什么类型，调用的时候才知道  
  
元组   
1、数组中的数据类型必须和规定的类型顺序对应起来  
2、当使用越界索引给数组赋值的时候，会使用联合类型（只要值是规定类型的某一种即可）。  
  
枚举 enum类型是对JavaScript标准数据类型的一个补充  
  
never类型表示的是那些永不存在的值的类型。  
  
readonly:只读属性,不可修改  
  
sex? :表示sex是一个可传属性,可以有也可以没有  
  
[propName: string]: any;表示新增的属性可以是任意类型  
  
arr3: Array<number> 数组类型定义  
  
arr2: (number | string)[]  
  
fn (a: number, b: number) : number 函数类型定义

# linux命令

pwd:输入pwd命令，Linux会输出当前目录。  
  
ls命令用来查看目录的内容。  
  
cd命令用来改变所在目录。  
  
cat命令可以用来合并文件，也可以用来在屏幕上显示整个文件的内容。  
  
grep命令的最大功能是在一堆文件中查找一个特定的字符串。  
  
touch命令用来创建新文件  
  
cp命令用来拷贝文件  
  
mv命令用来移动文件  
  
rm命令用来删除文件。   
  
mkdir 创建文件夹创建目录

# CSS3新特性

1 圆角边框：border-radius   
2 多背景图：background   
3 颜色和透明度：background: rgba(0,0,0,.5)   
4 多列布局和弹性盒：display: flex  
5 盒子的变幻（2D、3D）  
 transform: translate(50px,100px);//移动   
 transform: rotate(); //旋转   
 transform: scale(); //缩放  
 transform: skew(); //倾斜   
6 过渡 transition: width 1s linear 2s;   
动画：animation: myfirst 5s;  
@keyframes myfirst { 0% {background: block;} 25% {background: red;} 50% {background: yellow;} 100% {background: green;} }   
7 引入web字体（在服务器端存储）   
8 媒体查询   
9 阴影 h  
10 文字阴影 text-shadow div   
11 盒子阴影 box-shadow

# HTML5  h5新特性

1 语义化标签 section aside header nav  
2 表单新特性：autofocus 自动获取焦点  
 placeholder 占位  
 multiple  
 autocomplete 自动补全，是否自动记录之前提交的数据，以用于下一次输入建议  
 required 在表单提交时会验证是否有输入，没有则弹出提示消息。  
 min：限定输入数字的最小值。  
 max：限定输入数字的最大值  
3 video（视频）和audio（音频）   
4 canvas画布   
5 webworker 就是为 JavaScript 创造多线程环境，允许主线程创建 Worker 线程，将一些任务分配给后者运行。  
6 webscoket 服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种。  
7 拖拽api  
 dragstart：源对象开始拖放。   
 drag：源对象拖放过程中。   
 dragend：源对象拖放结束。   
 过程对象：   
 dragenter：源对象开始进入过程对象范围内。   
 dragover：源对象在过程对象范围内移动。   
 dragleave：源对象离开过程对象

# 标准盒模型与怪异盒模型区别

标准盒模型一个块的总宽度= width + margin(左右) + padding(左右) + border(左右)   
怪异盒模型一个块的总宽度= width + margin(左右)（即width已经包含了padding和 border值）

# CSS3弹性盒

弹性盒 display:flex（父元素添加）   
弹性盒：控制子元素按主轴方向排列   
弹性盒可以设置单独内容水平垂直居中  
父元素：display:flex 子元素：margin:auto   
灵活元素：灵活元素即使是内联元素也能设置宽高  
  
加父元素身上  
1.flex-direction 属性指定了弹性子元素在父容器中的位置。   
 row：横向从左到右排列（左对齐），默认的排列方式。  
 row-reverse：反转横向排列（右对齐，从后往前排，最后一项排在最前面。  
 column：纵向排列。  
 olumn-reverse：反转纵向排列，从后往前排，最后一项排在最上面。  
2.justify-content:flex-start/flex-end/space-between/space-around  
3.align-items: 设置或检索弹性盒子元素在侧轴（纵轴）方向上的对齐方式。   
 center/flex-start/flex-end/baseline/stretch  
4.flex-wrap:属性用于指定弹性盒子的子元素换行方式。nowrap/wrap/wrap-reverse  
5.align-content:设置各个行的对齐:   
 stretch - 默认。各行将会伸展以占用剩余的空间。  
 flex-start - 各行向弹性盒容器的起始位置堆叠。  
 flex-end - 各行向弹性盒容器的结束位置堆叠。  
 center -各行向弹性盒容器的中间位置堆叠。  
 space-between -各行在弹性盒容器中平均分布。  
 space-around   
   
加子元素身  
1、align-self：属性用于设置弹性元素自身在侧轴（纵轴）方向上的对齐方式。  
 align-self: auto | flex-start | flex-end | center | baseline | stretch  
 flex 属性用于指定弹性子元素如何分配空间。  
 1: 计算值为 1 1 auto  
 initial: 计算值为 0 1 auto  
 none：计算值为 0 0 auto  
inherit：从父元素继承  
2、第一个参数表示: flex-grow 定义项目的放大比例，默认为0，即如果存在剩余空间，也不放大  
3、第二个参数表示: flex-shrink 定义了项目的缩小比例，默认为1，即如果空间不足，该项目将缩小  
4、第三个参数表示: flex-basis给上面两个属性分配多余空间之前, 计算项目是否有多余空间, 默认值为 auto, 即项目本身的大小

# CSS元素居中

1.使用margin进行固定长度的偏移   
  
  
2.使用绝对定位并进行偏移(已知宽高)  
#father{ position:relative; }   
#son{ position: absolute; left:50%; top:50%;   
 margin-left: -子元素的宽/2; margin-top: -子元素的高/2; }   
  
3.使用绝对定位并margin自适应进行居中   
#father{ position:relative;}   
#son{position: absolute; left: 0; top: 0; right: 0; bottom: 0; margin:auto; }  
 (啊不斯陆特)  
4.使用弹性盒子来实现居中   
#father{ display: flex; justify-content: center; align-items: center; }  
 (加斯特佛) (额来) (哎特木)  
5.使用定位 + transform  
#father{ position:relative;}   
#son{ position: absolute; left: 50%; top: 50%; transform:translate(-50% -50%) }  
  
6.table-cell布局  
因为table-cell相当与表格的td，td为行内元素，无法设置宽和高，所以嵌套一层，嵌套一层必须设置display: inline-block;

# CSS几种定位的区别 relative, absolute, fixed

1、相对定位position:relative;参考物是自己,不脱离文档流(初始位置仍然占据空 间),top:100px; 给正值是向该容器的中心点移动;  
  
2、绝对定位position:absolute; 参考物是外层具有position属性的元素, 如果向外 都么有找到最后会参考body/html做定位  
  
3、固定定位position:fixed; 参考物是可视窗口   
  
4、粘性定位(了解)position:sticky; 是绝对定位+固定定位

# div外边距重叠的原因及解决办法

情况一:  
 两个div垂直边界相邻，margin会等于二者中margin较大的值   
解决方案:  
1.position：absolute   
2.float:left   
  
情况二:  
 子元素在父元素内，子元素的margin-top会与父元素的margin-top重叠，值等于二者中较大的，如果只有子元素设置了margin-top，则显示为父元素的margin-top   
解决方案:  
1.给父元素设置border（给子元素设置边框没有用）   
2.给父元素设置padding值   
3.给父元素或子元素添加float：left   
4.给父元素或子元素添加position：absolute   
5.给父元素添加overflow：hidden   
6.给子元素添加display：inline-block   
  
情况三:  
 一个空白元素自身的margin-top和margin-bottom会重叠，值为而这种较大的   
解决方案:  
1.设置透明border

# px，em，rem，pt的区别

1、px实际上就是像素，用px设置字体大小时，比较稳定和精确   
2、em就是根据基准来缩放字体的大小   
3、rem是相对于根元素字体大小来显示的   
4、pt的大小等于1英寸的1/72 浏览器的兼容性除了IE6-IE8，其他的浏览器都支持em和rem，px所有浏览器都支持

# CSS的BFC

1、什么是BFC BFC（Block formatting context）直译为“块级格式化上下文”。  
他是一个独立的渲染区域，也可以理解成一个独立的容器，并且这个容器里box的布局，与这个容器外的毫不相干。   
  
2、BFC渲染规则  
a.内部的box会在垂直方向，一个接一个的放置   
b.box垂直方向的距离由margin决定。属于同一个BFC的两个相邻box的margin会发生重 叠  
c.每个元素的margin box的左边，与包含块border box的左边相接触（对于从左往右的格 式化，否则相反）。  
即使存在浮动也是如此   
d.BFC的区域不会与float box重叠   
e.BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之 也如此   
f.计算BFC的高度时，浮动元素也参与计算   
  
3、如何产生BFC   
a.根元素  
b.float属性不为none   
c.position为absolute或fixed   
d.display为inline-block， table-cell，table-caption，flex，inline-flex   
e.overflow不为visible   
  
4、BFC的作用   
a.自适应两栏布局  
b.清除内部浮动   
c.防止垂直margin重叠（放在两个BFC里）

# 兼容问题

1）css浏览器兼容问题（比如设置元素透明度，ie浏览器使用滤镜实现，filter   
：progid：DXlmage Transform.Microsoft.Alpha(style=0,opacity=50);非IE浏览器   
opacity：0.5）   
解决方法：现在前端开发已经出现了非常多的框架和类库用于浏览器的兼容问题，比如jq类 库，解决获取元素中包含的所有文本内容兼容性问题，$(“element”).text(“element text”)   
  
2）css3浏览器前缀问题   
 -ms-：ie浏览器   
 -o-：opera欧朋浏览器   
 -webkit-：谷歌浏览器   
 -moz-：火狐浏览器  
  
工作中遇到的困难   
 1）IE6中高度不对问题   
 在div中给定了高度为1px,其它浏览器显示正常，可是ie6中显示的高度就不对了，这时我   
 给样式表中加 了个font-size:0px;line-height:0px;就好了   
 2)把border设为“0”像素虽然在页面上看不见，但按border默认值理解，  
 浏览器依然对 border- width/border-color进行了渲染，  
 即已经占用了内存值把border设为“none”即没有，浏览器解析“none”时将不作出渲染动作，即不会消耗内存

# 移动端适配

1.Media Queries 通过查询设备的宽度来执行不同的 css 代码，最终达到界面 的配置   
2.Flex弹性布局   
3.rem + viewport 缩放 实现原理 根据rem将页面放大dpr倍, 然后viewport设置为1/dpr.   
4、rem实现 移动端适配方案：   
 1）viewport（scale=1/dpr） dpr:备像素比   
 2）rem 3）flex 4）vm/vh  
 以iphone8为例，iphone8的CSS像素为375px\*677px，DPR是2，  
 所以其设备像素为750px\*1354px  
 750(px) / 375(px) = 2

# 什么是CSS预处理器？优点是什么

css预处理器用一种专门的编程语言，进行web页面样式设计，然后在编译成正常的css文件，  
以供项目使用 在css中使用变量、简单的逻辑程序、函数。可以让你的css更加简洁、适应性更强、可读性更佳、更易于代码的维护

# 0.5px线怎么实现（单边框0.5px）

方式一：border + border - img + 线性渐变linear-gradient  
方式二：定位 + 伪元素 + background + 线性渐变linear-gradient  
方式三：定位 + 伪元素 + transform缩放（scale）

# CSS布局 - 左侧宽度固定，右侧自适应

方法一:   
 设置绝对定位  
 宽度固定区域设置 position：absolute+ left/right+ top + width  
 自适应区域设置 margin-left/margin-right: 固定宽度区域的宽度。  
  
注意:  
 （1）若左侧固定的高度大于右侧自适应区域高度  
 页面布局就变成这样了：（测试布局区域紧挨着自适应区域的下方，盖住了部分固定区域）  
 简单粗暴的就是测试布局区域定位的top值直接设置为两个区域最高的高度值  
 （2）若左侧固定的高度大于右侧自适应区域高度，且测试区域不进行定位  
方法二:  
 浮动布局  
 左侧固定区域浮动+宽度，右侧自适应区域 设置margin-left :左侧宽度值。  
方法三:  
 BFC规则  
 左侧固定区域浮动+宽度，右侧自适应区域（非浮动元素）设置overflow：hidden。

# CSS布局 - 左右侧宽度固定，中间自适应

1、绝对定位布局：position + margin  
缺点: 如果中间栏含有最小宽度限制，或是含有宽度的内部元素，当浏览器宽度小到一定程度，会发生层重叠的情况。  
2、浮动布局： float + margin  
3、flex布局  
 高度由内容决定。  
4、table布局  
 高度由内容决定。  
5、Grid网格布局  
6、圣杯布局  
7、双飞翼布局  
8、对比圣杯布局和双飞翼布局  
（1）都是左右栏定宽，中间栏自适应的三栏布局，中间栏都放到文档流前面，保证先行渲染。  
（2）解决方案基本相似：都是三栏全部设置左浮动float：left,然后分别结局中间栏内容被覆盖的问题。  
（3）解决中间栏内容被覆盖问题时，圣杯布局设置父元素的padding,双飞翼布局在中间栏嵌套一个div,  
 内容放到新的 div中，并设置margin，实际上，双飞翼布局就是圣杯布局的改进方案。

# webpack 打包优化  怎么减少打包时间

多进程打包 - 速度分析  
多进程压缩 - 体积分析  
资源CDN  
动态polyfill - 根据不同浏览器，动态载入需要的polyfill,大幅度减少构建体积  
  
split-thunk  
前端构建项目中，为了提高打包效率，往往将第三库与业务逻辑代码分开打包，因为第三方库往往不需要经常打包更新。webpack建议使用CommonsChunk 来单独打包第三方库  
  
webpack loader(转换器)和plugin(插件)有什么区别:  
 loader它是一个转换器，将A文件进行编译形成B文件，这里操作的是文件，比如将A.scss转换为A.css，单纯的文件转换过程  
 plugin是一个扩展器，它丰富了webpack本身，针对是loader结束后，webpack打包的整个过程，它并不直接操作文件，而是基于事件机制工作，会监听webpack打包过程中的某些节点，执行广泛的任务  
  
loader:  
 优雅降级，  
 图片打包  
plugin:  
 html产出：把public下面的html文件打包到build里面的index.html并自动引入app.js  
 css抽离  
 静态资源拷贝

# 继承的6种方式

1.原型链继承  
 重点：让新实例的原型等于父类的实例。  
 特点：1、实例可继承的属性有：实例的构造函数的属性，父类构造函数属性，父类原型的属性。  
　　　 （新实例不会继承父类实例的属性！）  
 缺点：1、新实例无法向父类构造函数传参。  
 2、继承单一。  
 3、所有新实例都会共享父类实例的属性。  
　　　　　　（原型上的属性是共享的，一个实例修改了原型属性，另一个实例的原型属性也会被修改！）  
2.借用构造函数继承  
 重点：用.call()和.apply()将父类构造函数引入子类函数（在子类函数中做了父类函数的自执行（复制））  
 特点：1、只继承了父类构造函数的属性，没有继承父类原型的属性。  
 2、解决了原型链继承缺点1、2、3。  
 3、可以继承多个构造函数属性（call多个）。  
 4、在子实例中可向父实例传参。  
 缺点：1、只能继承父类构造函数的属性。  
 2、无法实现构造函数的复用。（每次用每次都要重新调用）  
 3、每个新实例都有父类构造函数的副本，臃肿。  
3.组合继承（组合原型链继承和借用构造函数继承）   
 重点：结合了两种模式的优点，传参和复用  
 特点：1、可以继承父类原型上的属性，可以传参，可复用。  
 2、每个新实例引入的构造函数属性是私有的。  
 缺点：调用了两次父类构造函数（耗内存），子类的构造函数会代替原型上的那个父类构造函数。  
4.原型式继承  
 重点：用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象。 object.create()就是这个原理。  
 特点：类似于复制一个对象，用函数来包装。  
 缺点：1、所有实例都会继承原型上的属性。  
 2、无法实现复用。（新实例属性都是后面添加的）  
5.寄生式继承  
 重点：就是给原型式继承外面套了个壳子。  
 优点：没有创建自定义类型，因为只是套了个壳子返回对象（这个），这个函数顺理成章就成了创建的新对象。  
 缺点：没用到原型，无法复用。  
6.寄生组合式继承（常用）  
 重点：修复了组合继承的问题  
 寄生：在函数内返回对象然后调用  
 组合：1、函数的原型等于另一个实例。  
 2、在函数中用apply或者call引入另一个构造函数，可传参

# 阻止冒泡和取消默认事件(默认行为)

防止事件捕获和冒泡:  
w3c的方法是e.stopPropagation()  
IE则是使用e.cancelBubble = true  
  
取消默认事件  
w3c的方法是e.preventDefault()  
IE则是使用e.returnValue = false  
  
jQuery用法  
阻止默认事件 return false (不停止冒泡)

# 作用域和作用域链

变量的作用域无非就是两种：  
全局变量和局部变量。javascript的作用域是相对函数而言的，可以称为函数作用域  
  
全局作用域：   
最外层函数定义的变量拥有全局作用域，即对任何内部函数来说，都是可以访问的  
  
局部作用域：   
局部作用域一般只在固定的代码片段内可访问到，而对于函数外部是无法访问的  
  
作用域链：  
根据在内部函数可以访问外部函数变量的这种机制，用链式查找决定哪些数据能被内部函数访问。  
 作用域链的前端，始终都是当前执行的代码所在环境的变量对象  
 作用域链中的下一个对象来自于外部环境，而在下一个变量对象则来自下一个外部环境，一直到全局执行环境  
 全局执行环境的变量对象始终都是作用域链上的最后一个对象

# 遍历数组的方式

1.for循环  
使用临时变量，将长度缓存起来，避免重复获取数组长度，当数组较大时优化效果才会比较明显。  
  
2.foreach循环  
遍历数组中的每一项，没有返回值，对原数组没有影响，不支持IE。  
有一些局限，不能continue跳过或者break终止循环  
  
3.map循环  
有返回值，可以return出来  
map的回调函数中支持return返回值  
并不影响原来的数组，return的是新数组  
  
4.for of遍历  
可以正确响应break、continue和return语句  
  
5.filter遍历  
不会改变原始数组,返回新数组  
  
6.every遍历  
every()是对数组中的每一项运行给定函数，如果该函数对每一项返回true,则返回true。  
  
7.some遍历  
some()是对数组中每一项运行指定函数，如果该函数对任一项返回true，则返回true。  
  
8.reduce  
reduce()方法接收一个函数作为累加器（accumulator），数组中的每个值（从左到右）开始缩减，最终为一个值。  
  
9.reduceRight  
reduceRight()方法的功能和reduce()功能是一样的，  
不同的是reduceRight()从数组的末尾向前将数组中的数组项做累加。  
  
10.find  
find()方法返回数组中符合测试函数条件的第一个元素。否则返回undefined   
  
11.findIndex  
对于数组中的每个元素，findIndex方法都会调用一次回调函数（采用升序索引顺序），直到有元素返回 true。  
只要有一个元素返回 true，findIndex立即返回该返回 true 的元素的索引值。  
如果数组中没有任何元素返回 true，则 findIndex 返回 -1。  
findIndex 不会改变数组对象。  
  
12.keys，values，entries  
ES6 提供三个新的方法 —— entries()，keys()和values() —— 用于遍历数组。  
它们都返回一个遍历器对象，可以用for...of循环进行遍历，  
唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历

# 数据分页

前端的话，定义一个新的数组for循环判断，有个公式，计算起始页和终止页的。  
符合条件push仅数组中，后端就是用limt判断

# 数组排序

1. 普通数组排序　　  
　　js中用方法sort()为数组排序。sort()方法有一个可选参数，是用来确定元素顺序的函数。如果这个参数被省略，那么数组中的元素将按照ASCII字符顺序进行排序。  
2. 冒泡排序  
（1）比较相邻的元素。如果第一个比第二个大，就交换他们两个位置。  
（2）对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。  
（3）针对所有的元素重复以上的步骤，除了最后一个。  
（4）持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。  
3. 快速排序：递归思想，两边快速的排序，冒泡排序的改进  
（1）选择数组中间数作为基数，并从数组中取出此基数；  
（2）准备两个数组容器，遍历数组，逐个与基数比对，较小的放左边容器，较大的放右边容器；  
（3）进行相同的操作，直到数组中只有一个元素时，返回该数组。  
4. 插入排序  
（1）从第一个元素开始，该元素可以认为已经被排序  
（2）取出下一个元素，在已经排序的元素序列中扫描  
（3）如果该元素（已排序）大于新元素，将该元素移到下一位置  
（4）重复步骤3，直到找到已排序的元素小于或者等于新元素的位置  
（5）将新元素插入到下一位置中  
（6）重复步骤2  
5. 选择排序  
（1）在未排序序列中找到最小（大）元素  
（2）并存放到排序序列的起始位置  
（3）然后，再从剩余未排序元素中继续寻找最小（大）元素  
（4）然后放到已排序序列的末尾。  
（5）以此类推

# 数组合并

1、concat  
 js的Array对象提供了一个叫concat()方法，连接两个或更多的数组，并返回结果。  
 var c = a.concat(b); //c=[1,2,3,4,5,6];  
  
 这里有一个问题，concat方法连接a、b两个数组后，a、b两个数组的数据不变，同时会返回一个新的数组。这样当我们需要进行多次的数组合并时，会造成很大的内存浪费，如果是数据量比较小的时候，还可以勉强用，如果数据量大的时候，这个就不妥了，所以这个方法肯定不是最好的。  
  
2、for循环  
大概的思路是：遍历其中一个数组，把该数组中的所有元素依次添加到另外一个数组中。直接上代码：  
 for(var i in b) { a.push ( b[i] );}  
   
3、apply  
 函数的apply方法有一个特性，那就是func.apply(obj,argv)，argv是一个数组。  
 a.push.apply(a,b);  
 调用a.push这个函数实例的apply方法，同时把，b当作参数传入，  
 这样a.push这个方法就会遍历b数组的所有元素，达到合并的效果。  
 这里可能有点绕，我们可以把b看成[4,5,6]，变成这样：  
 a.push.apply(a,[4,5,6]);  
 然后上面的操作就等同于：  
 a.push(4,5,6);  
 这样就很清楚了！

# 数组方法

push，在数组末尾添加一位或多位元素  
pop，删除数组最后一位元素  
unshift，在数组的开头添加一位或多位元素  
shift，删除数组的第一位元素  
join，将数组转换为字符串  
reserve，反转数组元素的顺序  
sort，对数组进行排序  
concat，连接两个或多个数组  
splice，添加或删除数组中的元素  
slice，从已有的数组中返回选定的元素  
indexOf、lastIndexOf，查找数组中的元素  
forEach，对数组进行遍历循环，对数组中每一项运行指定的函数  
map，迭代数组  
filter，对数组中的元素进行指定的检查返回符合条件的元素放入一个新数组中  
every，测试所有元素是否都符合指定条件  
some，测试某些元素是否符合指定条件  
reduce，接收一个函数作为累加器，数组中的每个值开始缩减，最终计算为一个值  
toString，将数组转换为字符串

# 数组去重

使用ES6 Set  
 var arr = [1, 1, 4, 2, 2, 3, 3, 3, 6, 6, 6];  
 arr = Array.from(new Set(arr));  
 console.log(arr);//[1, 4, 2, 3, 6]  
  
使用indexOf  
 var arr = [1, 1, 4, 2, 2, 3, 3, 3, 6, 6, 6];  
 var newArr = [];  
 arr.forEach((item) => {  
 newArr.indexOf(item) === -1 ? newArr.push(item) : "";  
 });  
 console.log(newArr);//[1, 4, 2, 3, 6]  
  
使用lastIndexOf  
 var arr = [1, 1, 4, 2, 2, 3, 3, 3, 6, 6, 6];  
 var newArr = [];  
 arr.forEach((item) => {  
 newArr.lastIndexOf(item) === -1 ? newArr.push(item) : "";  
 });  
 console.log(newArr);//[1, 4, 2, 3, 6]  
  
使用双重for循环加splice方法  
 var arr = [1, 1, 4, 2, 2, 3, 3, 3, 6, 6, 6];  
 for (var i = 0; i < arr.length; i++) {  
 for (var j = i + 1; j < arr.length; j++) {  
 if (arr[i] == arr[j]) {  
 arr.splice(j,1);  
 j--;  
 }  
 }  
 }  
 console.log(arr);//[1, 4, 2, 3, 6]  
  
使用forEach和includes方法  
 var arr = [1, 1, 4, 2, 2, 3, 3, 3, 6, 6, 6];  
 var newArr = [];  
 arr.forEach((item) => {  
 newArr.includes(item) ? "" : newArr.push(item);  
 });  
 console.log(newArr);//[1, 4, 2, 3, 6]  
  
使用fliter和includes方法  
 var arr = [1, 1, 4, 2, 2, 3, 3, 3, 6, 6, 6];  
 var newArr = [];  
 arr.filter((item) => {  
 newArr.includes(item) ? "" : newArr.push(item);  
 });  
 console.log(newArr);//[1, 4, 2, 3, 6]

# for...of 原理

for...of 是ES6引入用来遍历所有数据结构的统一方法。  
  
这里的所有数据结构只指具有iterator接口的数据。  
一个数据只要部署了 Symbol.iterator，就具有了 iterator接口，就可以使用 for...of 循环遍历它的成员。  
也就是说，for...of循环内部调用的数据结构为Symbol.iterator方法。  
部署在 Symbol.iterator 属性，或者说，一个数据结构只要具有 Symbol.iterator 属性，就认为是"可遍历的"。  
  
Iterator(伊特瑞特):  
 遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。  
 任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。  
 通俗点理解就是为了解决不同数据结构遍历的问题，引入了Iterator.  
  
Iterator的特点:  
 各种数据结构，提供一个统一的、简便的访问接口  
 使得数据结构的成员能够按某种次序排列  
 ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费  
   
原生具备 Iterator 接口的数据结构如下。  
 Array  
 Map  
 Set  
 String：字符串是一个类似数组的对象，也原生具有 Iterator 接口。  
 TypedArray：  
 通俗理解：ArrayBuffer是一片内存空间，不能直接引用里面的数据，可以通过TypedArray类型引用，  
 用户只能通过TypedArray使用这片内存，不能直接通过ArrayBuffer使用这片内存  
 函数的 arguments 对象  
 NodeList 对象  
   
除了原生具备Iterator 接口的数据之外，其他数据结构（主要是对象）的 Iterator 接口，  
都需要自己在Symbol.iterator属性上面部署，这样才会被for...of循环遍历。  
  
对象（Object）之所以没有默认部署 Iterator 接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作 Map 结构使用，ES5 没有 Map 结构，而 ES6 原生提供了。  
  
一个对象如果要具备可被for...of循环调用的 Iterator 接口，就必须在Symbol.iterator的属性上部署遍历器生成方法（原型链上的对象具有该方法也可）。

# 遍历对象的方式

第一种：   
for...in  
  
第二种：  
1）Object.keys（obj）  
2）Object.values（obj）  
参数：  
 obj：要返回其枚举自身属性的对象  
返回值：  
 一个表示给定对象的所有可枚举属性的字符串数组。  
  
第三种：  
使用Object.getOwnPropertyNames(obj)  
返回一个数组，包含对象自身的所有属性（包含不可枚举属性）  
遍历可以获取key和value

# map与filter和forEach的区别

map 与 filter 区别:  
 相同点：filter 和 map 都是对数组的操作，均返回一个新的数组  
 不同点：filter是满足条件的留下，是对原数组的过滤；map则是对原数组的加工，映射成一对一映射的新数组  
  
map 与 forEach 区别:  
 map()会分配内存空间存储新数组并返回，forEach()不会返回数据(undefined)。  
 forEach()允许callback更改原始数组的元素。map()返回新的数组。

# 面向对象编程

面向对象编程:  
将所需要做的功能抽象成一个“对象”，然后一遍遍地调用这个对象来完成你想要的功能。  
  
面向对象的三大特征:  
1.封装  
我们平时所用的方法和类都是一种封装，当我们在项目开发中，遇到一段功能的代码在好多地方重复使用的时候，我们可以把他单独封装成一个功能的方法，这样在我们需要使用的地方直接调用就可以了。  
  
2.继承  
继承在我们的项目开发中主要使用为子类继承父类，继承会继承父类的实例属性和实例方法，并不会继承静态属性和静态方法，并且静态方法只能通过类名去调用。  
  
3.多态  
多态的具体表现为方法重载和方法重写：  
方法重载：  
重载是指不同的函数使用相同的函数名，但是函数的参数个数或类型不同。调用的时候根据函数的参数来区别不同的函数  
方法重写：  
重写（也叫覆盖）是指在派生类中重新对基类中的虚函数（注意是虚函数）重新实现。即函数名和参数都一样，只是函数的实现体不一样  
  
三大特征的优点：  
封装：  
封装的优势在于定义只可以在类内部进行对属性的操作，外部无法对这些属性指手画脚，要想修改，也只能通过你定义的封装方法；  
  
继承：  
继承减少了代码的冗余，省略了很多重复代码，开发者可以从父类底层定义所有子类必须有的属性和方法，以达到耦合的目的；  
  
多态：  
多态实现了方法的个性化，不同的子类根据具体状况可以实现不同的方法，光有父类定义的方法不够灵活，遇见特殊状况就捉襟见肘了

# window.onload与document.ready的区别

1.执行时间  
 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。  
  
2.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  
  
3.简化写法  
 window.onload没有简化写法   
 $(document).ready(function(){})可以简写成$(function(){});

# this

this代表函数运行时，自动生成的一个内部对象，只能在函数内部使用，  
随着函数使用场合的不同，this的值会发生变化。  
  
指向:  
1.在方法中，this 表示该方法所属的对象。  
2.如果单独使用，this 表示全局对象。  
3.函数中，this 表示全局对象。  
4.严格模式下，this 是未定义的(undefined)。  
5.在事件中，this 表示接收事件的元素。  
6.类似 call() 和 apply() 方法可以将 this 引用到任何对象。  
7.箭头函数本身是没有this和arguments的，引用的this实际上是调用的是定义时的上一层作用域的this。

# <!DOCTYPE>

<!DOCTYPE> 声明位于文档中的最前面的位置，处于 <html> 标签之前。  
  
<!DOCTYPE> 声明不是一个 HTML 标签；它是用来告知 Web 浏览器页面使用了哪种 HTML 版本。  
  
在 HTML 4.01 中，<!DOCTYPE> 声明需引用 DTD （文档类型声明），  
因为 HTML 4.01 是基于 SGML （Standard Generalized Markup Language 标准通用标记语言）。  
DTD 指定了标记语言的规则，确保了浏览器能够正确的渲染内容。  
  
HTML5 不是基于 SGML，因此不要求引用 DTD。  
  
给您 HTML 文档添加 <!DOCTYPE> 声明，确保浏览器能够预先知道文档类型。

# CSS position属性

absolute:生成绝对定位的元素，相对于 static 定位以外的第一个父元素进行定位。  
fixed :生成固定定位的元素，相对于浏览器窗口进行定位。  
relative:生成相对定位的元素，相对于其正常位置进行定位。  
static :默认值。没有定位。  
sticky :粘性定位，该定位基于用户滚动的位置。  
inherit :规定应该从父元素继承 position 属性的值。

# js的解构

解构定义：允许按照一定模式，从数组和对象中提取值，对变量进行赋值。  
解构必须满足的条件：（模式匹配），只要等号两边的模式相同，左边的变量就会被赋予对应的值。  
主要介绍2种解构常用的类型：数组解构和对象解构。  
  
数组解构：  
 1.数组中的变量个数比赋值的数组中个数少的解构：  
 2.数组中的变量个数比赋值的数组中个数多的解构：  
 3.数组中的变量中有空字符的解构：上面的3种数组解构，赋值的数组的数值按位赋值给另外一方数组中的变量  
 4.数组中的变量中有不定参数(有些时候也叫扩展运算符)的解构：需要注意的是不定参数必须放最后。  
 5.数组中的变量中有默认值的解构：如果赋值的数组个数没有另外一方数组变量个数多，并且变量数组有默认，  
 没有进行赋值用默认，进行赋值用赋值的  
  
对象的解构：两方也需要相同类型的括号，对象结构使用大括号{}  
普通对象解构：  
 对象跟数值解构不同的一点，对象解构不是按位去赋值，是根据键名(属性名) 来进行赋值  
有默认值的对象解构  
 跟数组的默认值差不多，有赋值就用赋值的，没有就用默认值  
有不定参数的对象解构  
 跟数组的不定参数差不多，不定参数放在最后，不定参数创建一个对象接受的剩下的键名键值(属性名属性值)  
有嵌套的对象解构  
 主要注意嵌套层次是不是赋值一方对象的嵌套层次相同。

# js实现轮播图思路

1.图片移动实现原理：  
 利用浮动将所有所有照片依次排成一行，给这一长串图片添加一个父级的遮罩，每次只显示一张图，其余的都隐藏起来。对图片添加绝对定位，通过控制left属性，实现照片的移动。  
  
2.图片移动动画原理：  
从a位置移动到b位置，需要先计算两点之间的差值，通过差值和时间间隔，计算出每次移动的步长，通过添加定时器，每次移动相同的步长，实现动画效果。  
  
3.图片定位停止原理：  
每一张照片都有相同的宽度，每张照片都有一个绝对的定位数值，通过检测定每次移动后，照片当前位置和需要到达位置之间的距离是否小于步长，如果小于，说明已经移动到位，可以将定时器清除，来停止动画。  
  
4图片切换原理：  
在全局设置一个变量，记录当前图片的位置，每次切换或跳转时，只需要将数值修改，并调用图片页数转像素位置函数，再调用像素运动函数即可。  
  
5.自动轮播原理：  
设置定时器，一定时间间隔后，将照片标记加1，然后开始切换。  
  
6.左右点击切换原理：  
修改当前位置标记，开始切换。这里需要注意与自动轮播之间的冲突。当点击事件触发之后，停止自动轮播计时器，开始切换。当动画结束后再次添加自动轮播计时器。  
  
7.无缝衔接原理：  
需要无缝衔接，难度在于最后一页向后翻到第一页，和第一页向前翻到最后一页。由于图片的基本移动原理。要想实现无缝衔接，两张图片就必须紧贴在一起。所以在第一张的前面需要添加最后一张，最后一张的后面需要添加第一张。  
  
首先判断图片的位置，是否移动到位。当满足当前位置，与预定位置之间的距离小于一步时，认定为移动到位，并把图片直接定位到预定位置。然后判断，图片的位置是否需要跳转。  
ps：这里一定要在图片运动函数结束后，在进行跳转。  
  
8.预防鬼畜原理：  
始终保证轮播图的运动动画只有一个，从底层杜绝鬼畜。需要在每次动画开始之前，尝试停止动画定时器，然后开始为新的动画添加定时器。  
  
轮播图鬼畜的本质原因就是，同一时间多个定时器添加在图片上，这些定时器直接相互冲突，造成图片的抖动。  
解决方法：每次执行运动函数之前，先尝试清除一下，上一个轮播图的定时器。确保同时触发运动函数时，只有一个定时器在工作。  
  
9.预防暴力点击原理：  
如果用户快速点击触发事件，会在短时间内多次调用切换函数，虽然动画函数可以保证，不会发生鬼畜，但在照片从最后一张到第一张的切换过程，不会按照正常的轮播，而是实现了跳转。所以需要通过添加口令的方式来，限制用户的点击。当用户点击完成后，口令销毁，动画结束后恢复口令。  
  
10.小圆点的位置显示原理：  
每次触发动画时，通过全局变量标记，获取当前页数，操作清除所有小圆点，然后指定一页添加样式。  
  
11.点击触发跳转的原理：  
类似于左右点击触发，只是这是将全局页面标记，直接修改，后执行动画。需要避免与自动轮播定时器的冲突。

# js数组和对象的扩展

1.Array.from()方法用于将对象转为真正的数组(类数组转数组)  
2.Array.of()方法用于将一组值，转换为数组。  
 console.log(Array.of(1,2,3,4,4,50));//[1, 2, 3, 4, 4, 50]  
3.Object.assign(目标对象,对象1,对象2)用于对象的合并,将源对象的所有可枚举属性，复制到目标对象。(浅拷贝)

# js提升

一、提升（Hosting）  
简单说就是在js代码执行前引擎会先进行预编译，预编译期间会将变量声明与函数声明提升至其对应作用域的最顶端。  
  
二、变量提升  
在ES6之前，JavaScript没有块级作用域(一对花括号{}即为一个块级作用域)，只有全局作用域和函数作用域。  
变量提升即将变量声明提升到它所在作用域的最开始的部分。  
变量声明的提升是以变量所处的第一层词法作用域为“单位”的，即全局作用域中声明的变量会提升至全局最顶层，函数内声明的  
  
三、函数提升  
即函数提升只会提升函数声明，而不会提升函数表达式。  
  
四、为什么会有提升？  
函数提升就是为了解决相互递归的问题，大体上可以解决像ML语言这样自下而上的顺序问题。  
大概是说，变量提升是人为实现的问题，而函数提升在当初设计时是有目的的。

# 比较redux和vuex的区别

redux和vuex的区别  
 1）vuex是redux的基础上进行改变，对仓库的管理更加明确  
 2）使用mutation来替换redux中的reducer  
 3)vuex有自动渲染的功能，所以不需要更新  
   
redux和flux的区别  
　　1）redux是flux中的一个实现  
　　2））在redux中我们只能定义一个store，在flux中我们可以定义多个  
　　3）在redux中，store和dispatch都放到了store，结构更加清晰  
　　4）在redux中本身就内置State对象，对仓库的管理更加明确

# CSS display 属性

|  |  |  |
| --- | --- | --- |
| none | 此元素不会被显示。 |  |
| block | 此元素将显示为块级元素，此元素前后会带有换行符。 |  |
| inline | 默认。此元素会被显示为内联元素，元素前后没有换行符。 |  |
| inline-block | 行内块元素。（CSS2.1 新增的值） |  |
| list-item | 此元素会作为列表显示。 |  |
| run-in | 此元素会根据上下文作为块级元素或内联元素显示。 |  |
| compact | CSS 中有值 compact，不过由于缺乏广泛支持，已经从 CSS2.1 中删除。 |  |
| marker | CSS 中有值 marker，不过由于缺乏广泛支持，已经从 CSS2.1 中删除。 |  |
| table | 此元素会作为块级表格来显示（类似 |  |
| inline-table | 此元素会作为内联表格来显示（类似 |  |
| table-row-group | 此元素会作为一个或多个行的分组来显示（类似 |  |
| table-header-group | 此元素会作为一个或多个行的分组来显示（类似 |  |
| table-footer-group | 此元素会作为一个或多个行的分组来显示（类似 ）。 |  |
| table-row | 此元素会作为一个表格行显示（类似 |  |
| table-column-group | 此元素会作为一个或多个列的分组来显示（类似 |  |
| table-column | 此元素会作为一个单元格列显示（类似 ） |  |
| table-cell | 此元素会作为一个表格单元格显示（类似 | 和 ） |
| table-caption | 此元素会作为一个表格标题显示（类似 |  |
| inherit | 规定应该从父元素继承 display 属性的值。 |  |

# CSS选择符

CSS选择符  
 id选择器（#myid）  
 类选择器（.myclassname）  
 标签选择器（div）  
 相邻选择器（h1+p）  
 子选择器（ul>li）  
 后代选择器（li a）  
 群组选择器（div，p{}）  
 通配符选择器（\*）  
 属性选择器（a[title=""]）  
 伪类选择器（a:hover{}或者li:nth-child{}）

# CSS中link和@import的区别

link属于HTML标签，@import是CSS提供的  
  
页面被加载时，link引用的css文件会同时被加载，而@import引用的css文件要等到页面被加载完毕再加载  
  
import只在IE5以上才能识别，link是HTML标签，无兼容问题  
  
link 引入样式的权重大于@import 的引用

# CSS中display:none、visibility:hidden和opacity:0;的区别

display:none;隐藏对应的元素，在文档布局中不再给它分配空间，它各边的元素会合拢。  
  
visibility:hidden;隐藏对应的元素，但在文档布局中仍保留原来的空间。  
  
opacity:0;内容不可见，占据空间。

# CSS中rgba()和opacity的透明效果区别

rgba()和 opacity 都能实现透明效果，但最大的不同是 opacity 作用于元素，以及元素内的所有内容的透明度；  
  
rgba()只作用于元素的颜色或其背景色（设置 rgba 透明的元素的子元素不会继承透明效果）。

# CSS的外边距重叠

在 CSS 当中，相邻的两个盒子（可能是兄弟关系也可能是祖先关系）的外边距可以结合成一个单独的外边距。这种合并外边距的方式被称为折叠，并且因而所结合成的外边距称为折叠外边距。  
  
折叠结果遵循下列计算规则如下  
  
两个相邻的外边距都是正数时，折叠结果是它们两者之间较大的值。  
  
两个相邻的外边距都是负数时，折叠结果是两者绝对值的较大值。  
  
两个外边距一正一负时，折叠结果是两者之和。

# CSS清除浮动

清除浮动主要是为了解决父元素因为子元素浮动引起的内部高度为0的问题。  
常见清除浮动的方法  
 额外标签法。在最后浮动元素后加一个标签，设置clear:both；  
 给父级元素设置高度  
 给父级元素添加浮动  
 给父级元素设置overflow属性为hidden或者auto  
 使用after伪元素清除  
 .clearfix:after {  
 /\*伪元素是行内元素 正常浏览器清除浮动方法\*/  
 content: "";  
 display: block;  
 height: 0;  
 clear: both;  
 visibility: hidden;  
 }  
 /\*ie6清除浮动的方式 \*号只有IE6-IE7执行，其他浏览器不执行\*/  
 .clearfix {  
 \*zoom: 1;  
 }  
  
 使用after和before双伪元素清除  
 .clearfix:before,.clearfix:after {  
 content: "";  
 display: block;  
 clear: both;  
 }  
 .clearfix {  
 zoom: 1;  
 }

# CSS3transition过渡和animation动画的区别

animation属性类似于transition，他们都是随着时间改变元素的属性值，  
   
其主要区别在于：  
transition需要触发一个事件才会随着时间改变其CSS属性；  
animation在不需要触发任何事件的情况下，也可以显式的随时间变化来改变元素CSS属性，达到一种动画的效果  
   
1）动画不需要事件触发，过渡需要。  
2）过渡只有一组（两个：开始-结束） 关键帧，动画可以设置多个。

# 前后端开发中数据是怎么交互的

web后端和前端是怎么连接的？  
  
网站数据处理主要分为三层。  
第一层，表示层，这部分可以用HTML代码，CSS/Javascript代码来实现等。  
 通过前端代码可以实现网页的布局和设计。这层又可以称为显示层。也就是你用浏览器打开能看到的网页。  
第二层，是业务层，这层是负责处理数据的。常用的代码语言有PHP,JSP，Java等。  
 通过这些后台处理语言的算法来处理前台传回的数据。必要的时候进行操作数据库，然后把结果返回给前端网页。  
第三层，是数据层，这个就是数据库，用来存储数据的。  
 通过业务层的操作可以实现增删改数据库的操作。  
   
在网页上填一个表格然后提交会有以下几种数据传输经过：  
①你接触到的是这个网页是属于表示层，这个网页一般由HTML标签结合CSS/JAVASCRIPT来实现的。这时候你要先填入数据。  
②然后你按提交触发后台处理机制，这时候数据会传到后台的代码进行处理。这部分代码根据不同网站可以使PHP,JSP，JAVA等。代码根据程序员预设的算法将收到的数据进行处理之后会相应的对数据库进行操作，存储数据等。  
③成功操作完数据库之后，业务层的代码会再向表示层也就是显示器端传回一个指令通知你表格填写成功  
  
从前端向后台传递参数方法  
  
一、通过表单传递参数  
1.前端部分，在前端jsp页面设置form表单，确定需要传递的参数name让用户输入，  
 通过点击按钮后submit()提交到后台  
2.后台对前端请求的反应，接收数据，处理数据以及返回数据。  
  
二．通过ajax传递参数（有post和get写法）  
1.ajax是如何将前端数据传到后台的  
 type: "POST",//type是ajax的方法  
 url : "<%=path%>/resource/usermenus",//参数url,要把参数传到什么地方  
 data : {parentid:parentid,parentpath:parentpath},//传递什么数据  
 success : function(data){//sucess表示，当数据返回成功后要怎么做，返回的数据存储在data  
2.后台对前端请求的反应，接收数据  
3.前端接收到后端返回的数据进行处理的

# cookie的理解

cookie又叫会话跟踪技术，是由web服务器保存在用户浏览器上的小文本文件，它可以记录用户ID、密码、浏览过的网页、停留的时间等信息。当用户再次来到该网站时，网站通过读取cookie，得知用户相关信息，就可以做出相应的动作，如在页面显示欢迎用户标语，或者让用户不用输入ID、密码就直接登录等等。如果用户清理了cookie，那么用户曾登录过的网站信息就没有了。  
  
优点  
 极高的扩展性和可用性  
 通过良好的编程，可以控制保存在cookie中的session对象的大小  
 通过加密技术和安全传输技术，减少cookie被破解的可能  
 只在cookie中存放不敏感数据，即使被盗也不会有重大的损失  
 控制cookie的生命周期，使之不会永久有效。偷盗者可能拿到一个过期的cookie  
  
缺点  
 cookie有数量和长度的限制。每个domain最多有20条cookie，长度不能超过4KB，否则会被裁掉。  
 安全性问题。如果cookie被人拦截了，那个人就可以取得所有的session信息。即使加密也于事无补，因为拦截者不需要知道cookie的意义，他只需要原样转发就能达到目的。  
 有些状态不可能保存在客户端。

# 输入网址到呈现网页发生的过程

a.域名解析  
 DNS解析，每一台连上网计算机都有一个唯一标识即它的IP地址，DNS解析就将输入的网址解析成IP地址。  
DNS解析是一个递归查询的过程，例如要解析“www.baidu.com”时，过程如下：  
 在本地域名服务器中查询IP地址，未找到域名；  
 本地域名服务器会向根域名服务器发送请求，未找到域名；  
 本地域名服务器向.com顶级域名服务器发送请求，未找到域名；  
 本地域名服务器向.baidu.com域名服务器发送请求，找到该域名，将相应的IP返回给本地域名服务器；  
  
b.发起TCP连接的三次握手  
 HTTP协议是使用TCP协议作为其传输层协议的，在拿到服务器的IP地址后，客户端浏览器会与服务器建立TCP连接，该过程包括三次握手：  
 第一次握手：建立连接时，客户端向服务端发送请求报文（SYN）  
 第二次握手：服务器收到请求报文后，如同意连接，则向客户端发送确认报文（SYN/ACK）  
 第三次握手：客户端收到服务器的确认后，再次向服务器发送确认报文，完成连接（ACK）  
三次握手主要是为了防止已经失效的请求报文字段发送给服务器，浪费资源。  
  
c.建立TCP连接后浏览器发起HTTP请求  
 浏览器构建HTTP请求报文，并通过TCP协议传送到服务器的指定端口。HTTP请求报文一共有三个部分：  
 报文首部（请求行+各种首部字段+其他）  
 空行（它的作用是通过一个空行，告诉服务器请求头部到此为止。）  
 报文主体（应被发送的数据）通常并不一定要有报文主体  
   
d.服务端响应http请求，返回响应报文  
HTTP响应报文由四部分组成：响应行、响应头、空行、响应体  
响应行  
 响应行一般由协议版本、状态码及其描述组成  
  
响应头  
 响应头用于描述服务器的基本信息，以及数据的描述，服务器通过这些数据的描述信息，可以通知客户端如何处理等一会儿它回送的数据。  
常见的响应头字段含义：  
 Allow：服务器支持哪些请求方法(如GET、POST等)。  
 Content-Encoding：文档的编码(Encode)方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩文档能够显著地减少HTML文档的下载时间。Java的GZIPOutputStream可以很方便地进行gzip压缩，但只有Unix上的Netscape和Windows上的IE4、IE5才支持它。因此，Servlet应该通过查看Accept-Encoding头(即request.getHeader(“Accept- Encoding”))检查浏览器是否支持gzip，为支持gzip的浏览器返回经gzip压缩的HTML页面，为其他浏览器返回普通页面。  
 Content-Length：表示内容长度。只有当浏览器使用持久HTTP连接时才需要这个数据。如果你想要利用持久连接的优势，可以把输出文档写入 ByteArrayOutputStram，完成后查看其大小，然后把该值放入Content-Length头，最后通过byteArrayStream.writeTo(response.getOutputStream()发送内容。  
 Content- Type：表示后面的文档属于什么MIME类型。Servlet默认为text/plain，但通常需要显式地指定为text/html。由于经常要设置 Content-Type，因此HttpServletResponse提供了一个专用的方法setContentType。  
 Date：当前的GMT时间，例如，Date:Mon,31Dec200104:25:57GMT。Date描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。你可以用setDateHeader来设置这个头以避免转换时间格式的麻烦。  
 Expires：告诉浏览器把回送的资源缓存多长时间，-1或0则是不缓存。  
 Last-Modified：文档的最后改动时间。客户可以通过If-Modified-Since请求头提供一个日期，该请求将被视为一个条件GET，只有改动时间迟于指定时间的文档才会返回，否则返回一个304(Not Modified)状态。Last-Modified也可用setDateHeader方法来设置。  
 Location：这个头配合302状态码使用，用于重定向接收者到一个新URI地址。表示客户应当到哪里去提取文档。Location通常不是直接设置的，而是通过HttpServletResponse的sendRedirect方法，该方法同时设置状态代码为302。  
 Refresh：告诉浏览器隔多久刷新一次，以秒计。  
 Server：服务器通过这个头告诉浏览器服务器的类型。Server响应头包含处理请求的原始服务器的软件信息。此域能包含多个产品标识和注释，产品标识一般按照重要性排序。Servlet一般不设置这个值，而是由Web服务器自己设置。  
 Set-Cookie：设置和页面关联的Cookie。Servlet不应使用response.setHeader(“Set-Cookie”, …)，而是应使用HttpServletResponse提供的专用方法addCookie。  
 Transfer-Encoding：告诉浏览器数据的传送格式。  
 WWW-Authenticate：客户应该在Authorization头中提供什么类型的授权信息?在包含401(Unauthorized)状态行的应答中这个头是必需的。例如，response.setHeader(“WWW-Authenticate”, “BASIC realm=\”executives\”“)。注意Servlet一般不进行这方面的处理，而是让Web服务器的专门机制来控制受密码保护页面的访问。  
 注：设置应答头最常用的方法是HttpServletResponse的setHeader，该方法有两个参数，分别表示应答头的名字和值。和设置状态代码相似，设置应答头应该在发送任何文档内容之前进行。  
 setDateHeader方法和setIntHeader方法专门用来设置包含日期和整数值的应答头，前者避免了把Java时间转换为GMT时间字符串的麻烦，后者则避免了把整数转换为字符串的麻烦。  
 HttpServletResponse还提供了许多设置  
 setContentType：设置Content-Type头。大多数Servlet都要用到这个方法。  
 setContentLength：设置Content-Length头。对于支持持久HTTP连接的浏览器来说，这个函数是很有用的。  
 addCookie：设置一个Cookie(Servlet API中没有setCookie方法，因为应答往往包含多个Set-Cookie头)。  
  
响应体  
响应体就是响应的消息体，如果是纯数据就是返回纯数据，如果请求的是HTML页面，那么返回的就是HTML代码，如果是JS就是JS代码，如此之类。  
  
e.浏览器页面渲染  
　　解析文档构建DOM树  
　　构建渲染树  
　　布局和绘制渲染树  
　　  
f.断开TCP连接  
 第一次挥手：客户端想分手，发送消息（FIN）给服务器  
 第二次挥手：服务器通知客户端已经接受的挥手请求，返回确认消息（ACK），但还没做好分手准备  
 第三次挥手：服务器已经做好分手准备，通知客户端（FIN）  
 第四次挥手：客户端发送消息给服务器（ACK），确认分手，服务器关闭连接。

# 浏览器渲染原理及流程

1. 浏览器会将HTML解析成一个DOM树，DOM 树的构建过程是一个深度遍历过程：  
 当前节点的所有子节点都构建好后才会去构建当前节点的下一个兄弟节点。  
  
2. 将CSS解析成 CSS Rule Tree 。  
  
3. 根据DOM树和CSSOM来构造 Rendering Tree。渲染树  
 注意：Rendering Tree 渲染树并不等同于 DOM 树，  
 因为一些像Header或display:none的东西就没必要放在渲染树中了。  
  
4. 有了Render Tree，浏览器已经能知道网页中有哪些节点、各个节点的CSS定义以及他们的从属关系。  
 下一步操作称之为layout，顾名思义就是计算出每个节点在屏幕中的位置。  
  
5. 再下一步就是绘制，即遍历render树，并使用UI后端层绘制每个节点。

# 浏览器本地存储

HTML5中的Web Storage包含了两种存储方式，sessionStorage和localStorage。  
  
sessionStorage用于本地存储一个会话中的数据，这些数据只有在同一个会话中的页面才能访问并且当会话结束后数据也随之销毁。因此，sessionStorage不是一种持久化的本地存储，仅仅是会话级别的存储。  
  
localStorage用于持久化的本地存储，除非主动删除数据，否则数据永远不会过期  
  
首先总的来说，三者都是用于持久化数据存储的手段，都是存储在浏览器端，且同源.  
  
localStorage和sessionStorage都是Web存储，大小5M左右，完全存储在客户端，它们是因为本地存储数据而存在.  
  
cookies也是存储在浏览器端的，大小不超过4k，由服务器端存储在客户端。  
  
localStorage属于永久性存储,数据存储量大,而sessionStorage属于当会话结束的时候，存储的值会被清空，而cookie是通过设置过期时间来存储的。

# cookie,localStorage,sessionStorage区别

介绍  
1.cookie又叫会话跟踪技术，是由web服务器保存在用户浏览器上的小文本文件，它可以记录用户ID、密码、浏览过的网页、停留的时间等信息。当用户再次来到该网站时，网站通过读取cookie，得知用户相关信息,如果用户清理了cookie，那么用户曾登录过的网站信息就没有了。  
2.sessionStorage和localStorage是H5新增的本地存储  
  
区别  
1.存储大小的不同  
 localStorage的大小一般为5M  
 sessionStorage的大小一般为5M  
 cookies的大小一般为4K  
2.有效期不同：  
 localStorage的有效期为永久有效，除非你进行手动删除。  
 sessionStorage在当前会话下有效，关闭页面或者浏览器时会被清空。  
 cookies在设置的有效之前有效，当超过有效期便会失效。  
3.与服务器端的通信  
 localStorage不参与服务器端的通信。  
 sessionStorage不参与服务器端的通信。  
 cookies参与服务器端通信，每次都会携带http的头信息中。（如果使用cookie保存过多数据会带来性能问题）  
  
应用场景  
1.因为考虑到每个 HTTP 请求都会带着 Cookie 的信息，比较常用的一个应用场景就是判断用户是否登录。针对登录过的用户，服务器端会在他登录时往 Cookie 中插入一段加密过的唯一辨识单一用户的辨识码，下次只要读取这个值就可以判断当前用户是否登录啦。曾经还使用 Cookie 来保存用户在电商网站的购物车信息，如今有了 localStorage，现在基本更加方便  
2.sessionStorage:  
 可以用来统计当前页面元素的点击次数。  
3.localStoragese：  
 常用于长期登录(判断用户是否已登录),适合长期保存在本地的数据。sessionStorage：敏感账号一次性登录；  
  
登陆信息用cookie好还是localStorage好？  
 1.建议登陆信息用 cookie。即设置过期时间的cookie,看法是 cookie 默认会发送回到后端，  
 这样方便后端读取,而使用localStorage，你需要在每次请求的时候，都手动带上这个信息，  
 这大大增加了开发过程中带来的困难，非常麻烦，而且还要手动维护过期时间。  
 2.cookie有时效，而localStorage如果不手动清理则永久保存。  
 3.如果要设置关闭网页/标签就失效,请用SessionStorage。 这个类似你设置“不带时间的cookie”。

# hash值获取方式

可以通过window.location.hash获取hash值

# document.write和innerHTML的区别

document.write会重绘整个页面  
  
innerHTML重绘页面的一部分

# 进程和线程的区别

一个程序至少有一个进程，一个进程至少有一个线程  
  
线程的划分尺度小于进程，使得多线程程序的并发性高  
  
进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大提高了程序的运行效率  
  
每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制   
  
从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别

# babel原理

Babel 是一个通用的多功能 JavaScript 编译器，但与一般编译器不同的是它只是把同种语言的高版本规则转换为低版本规则，而不是输出另一种低级机器可识别的代码，并且在依赖不同的拓展插件下可用于不同形式的静态分析。（静态分析：指在不需要执行代码的前提下对代码进行分析以及相应处理的一个过程，主要应用于语法检查、编译、代码高亮、代码转换、优化、压缩等等）  
  
babel 做了什么  
和编译器类似，babel 的转译过程也分为三个阶段，这三步具体是：  
  
1.解析 Parse  
将代码解析生成抽象语法树( 即AST )，也就是计算机理解我们代码的方式(扩展：一般来说每个 js 引擎都有自己的 AST，比如熟知的 v8，chrome 浏览器会把 js 源码转换为抽象语法树，再进一步转换为字节码或机器代码)，而 babel 则是通过 babylon 实现的 。简单来说就是一个对于 JS 代码的一个编译过程，进行了词法分析与语法分析的过程。  
2.转换 Transform  
对于 AST 进行变换一系列的操作，babel 接受得到 AST 并通过 babel-traverse 对其进行遍历，在此过程中进行添加、更新及移除等操作。  
3.生成 Generate  
将变换后的 AST 再转换为 JS 代码, 使用到的模块是 babel-generator。  
  
而 babel-core 模块则是将三者结合使得对外提供的API做了一个简化。  
  
此外需要注意的是，babel 只是转译新标准引入的语法，比如ES6箭头函数：而新标准引入的新的原生对象，部分原生对象新增的原型方法，新增的 API 等（Proxy、Set 等）, 这些事不会转译的，需要引入对应的 polyfill 来解决。

# DVA工作流程