1.script标签中的defer属性是下载完成就执行，而async是渲染完再执行，多个脚本情况下，defer可以保证加载顺序，这样的说法对吗

1.对

2..call、bind方法是否可以改变箭头函数的this指向？

没有它自己的this值，箭头函数内的this值继承自外围作用域。在箭头函数中调用 this 时，仅仅是简单的沿着作用域链向上寻找，找到最近的一个 this 拿来使用。

箭头函数在定义之后，this 就不会发生改变了，无论用什么样的方式调用它，this 都不会改变；

3.let、var和const的区别？

let 是块级作用域，作用域范围更小，一个for 循环也是一个作用域，

var 是函数作用区，可以重复声明，存在变量的提升

const 是块级作用域， 是声明常量的，如果是引用类型的常量，是可以改变的，

如果是基本类型的常量是不可以改变的

4.webpack打包最基本的实现方式 是将所有的模块代码放到一个数组里 通过数组ID来引用不同的模块、对吗 对

5.react中如何阻止js原生事件的冒泡？

React 为提高性能，有自己的一套事件处理机制，相当于将事件代理到全局进行处理，也就是说监听函数并未绑定到DOM元素上。因此，如果你禁止react事件冒泡e.stopPropagation()，你就无法阻止原生事件冒泡；你禁用原生事件冒泡e.nativeEvent.stopPropagation()，React的监听函数就调用不到了。

正确的做法是应该是判断event.target对象，是否是目标对象、或包含的对象、或被包含的对象，来决定是否触发事件

6.commonJs的模块可以理解为值的引用，es6的模块可以理解为值的拷贝，是吗？

不对 说反了

es6模块的特点：

静态化，必须在顶部，不能使用条件语句，自动采用严格模式

treeshaking和编译优化，以及webpack3中的作用域提升

外部可以拿到实时值，而非缓存值(是引用而不是copy)

es6模块和commonjs模块的区别：

可以对commonjs模块重新赋值，对es6模块重新赋值会编译报错

commonjs是对模块的拷贝（浅拷贝），es6是对模块的引用（也就是说， es6模块只存只读，不能改变其值，具体点就是指针指向不能变，类似 const）

es6模块和commonjs模块的相同点：

两者都可以对对象内部属性的值进行改变

7..react组件传值。

React 中组件的几种通信方式，分别是：

1. 通过props属性实现组件间通信

父组件向子组件通信：使用 props -->不要超过三层

子组件向父组件通信：使用 props 回调-->不要超过三层

2.通过prop-types的context实现跨级组件间通信

跨级组件间通信双向：使用 context 对象，APP组件和其他所有子孙通信，不太适合组件间通信（可以实现，不好维护）

3.使用事件订阅实现非嵌套组件间通信，也可以实现跨级组件间通信

(1)安装 cnpm i events -S

(2)新建一个文件 ev.js

(3)A组件 绑定自定义事件，在B触发，回调带回数据

4.reudx解决复杂应用中组件通信问题

8.了解dva吗？

dva开发框架:

---react-router、redux、redux-saga二次封装

--代码量变少

--开发和维护更加方便

---webpack脚手架--roadhog

---内置redux-devtools-extension

1、webpack性能优化：

\* webpack代码分割：他可以把代码分割到不同的文件中，按需或者并行的加载这些文件。分割后会得到更小的文件，控制文件加载的优先级，优化加载时间.

分割的方法的话，可以在配置文件里配置（entry points）

\* 删除没用的依赖

\* 在多页面应用中，提取公共代码块，减少不同入口处重叠的代码.

\* happypack可以让loader多进程的处理文件.

2、redux：

只能通过dispatch，action来修改state。

异步处理的时候没有好的方法，只能通过中间件来做，像thunk.

3、setstate（异步原理）：

先观察一下setstate接受的参数，如果第一个是对象，那么他就是一个异步操作，这个时候我们只能通过第二个参数，也就是一个回调函数，才能得到一个实时的状态.如果第一个参数是函数的话，他接收的是新的props和没改的state，这个用得比较少.

首先写成setstate异步原因是：

如果setstate是同步的话，在react中不可能只有一个state，这样的话多个会造成state阻塞.

在state重新渲染的时候，props不会同步更新到dom上，在dom上是没有办法操作props的.

4、像组件间的通信层级比较大，使用redux来做.

5、url到页面显示发生了什么

首先我们输入url，浏览器会先查看浏览器缓存，系统缓存，和路由缓存，有的话直接显示，没有就继续进行，然后浏览器会利用dns解析地址，并发起tcp连接，建立三次握手，然后发送http请求，请求数据包，服务器收到请求，处理并且响应，浏览器收到响应，渲染到页面上，解析html代码，生成dom树，像dom树就是以HTMLdocument为根节点，其余节点为子节点组成树形的数据结构.另外在这期间可能会遇到需要解析的js代码，那么会停止等待js解析结束后，在进行其他的渲染，也是一个很常见的问题，所以我们要把js放到整个html渲染结束后再去执行.

6、强缓存和协商缓存

我们在第一次获取资源后，会根据返回的信息来决定是以什么形式进行缓存。而当我们在下次请求的时候，会先获取该资源缓存的header信息，判断是否命中强缓存，若命中直接从缓存中获取资源，如果没有命中强缓存，浏览器会请求到服务器并返回第一次的header信息，然后进行对比从而观察是否命中了协商缓存，如果命中会更新新的header里面的信息，并从缓存中获取内容.

7、tcp三个接口：首先用户向服务器发送一个syn（同步序列编号），等待服务器响应，确认用户的syn，同时响应会一个syn+ack包，后者用户收到syn+ack（确认字符）后，服务器连接成功，这就是三次握手.

8、dns是否能缓存：能的，在我们多次访问某个相同的地址的时候，dns会在第一次访问后，把这个ip访问的结果临时缓存起来，而后面的多次调用也都是调用这个里面的.

9、kpi问题:最初的时候，会进行简单的一些考核，像工作态度，项目完成度，主动性啥的，后期的话也就没有.

10、redux：redux用来处理数据，内部分为三个大的部分进行配合，action reducer store ，在心的state返回到store中以后，我们需要用subscript去使render方法执行，然后渲染到dom上去.但是这种方法很麻烦，在处理多个状态数据的时候.

11、react-redux：而react-redux就是解决这个问题产生的功能，它主要分为两个主要的部分，provider和connect方法：

privider是一个组件，我们一般都会将顶层的组件写在里面，这样我们的所有组件也就可以受react-redux的控制了，但是store要作为参数传入其中，目的就是让所有组件都能访问redux中的数据.

connect方法：connect是一个高阶函数，首先传入mapStateToProps、mapDispatchToProps，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

12、中间件（thunk，saga）：thunk和saga都是在react中用来解决异步操作的，

thunk：出现的原因的在于我们在正常dispath打reducer中的时候，是同步执行的，而这就导致了，我在页面中触发事件请求某个结果，会造成必须等到结果响应回来才能更新视图层，那么解决这个问题我们推出了thunk，对action进行扩展，

我们可以在actionCreactor中编写逻辑，扩展action的功能，处理请求结果，等到处理结束后，传入store中，再去通过订阅进行更新.

saga：和thunk基本作用相同，而saga主要把逻辑代码放到了saga.js中，和thunk比较的话，具有更多的API，处理大型项目的时候优于thunk.

13、js中异步方法：

回调函数 事件监听 发布订阅 （sunscript） promise async+await

14、react与vue区别

\* react整体是函数式的思想，把组件设计成纯组件，状态和逻辑通过参数传入，所以是单向数据流，而vue的思想是响应式的，他是基于数据是可变的，当属性发生改变时响应式的更新对应的虚拟dom.

\* react的内部追求的是所有的都是js，用 jsx语法来写，用js操作css等，vue则是把他们分开，自己独立处理，

\* react api很少，vue声明式的写法，api很多.

\* react可以通过高阶组件来扩展，而vue则是通过mixins来扩展.

15、跨域：jsonp cros iframe+window.name, proxy

jsonp的缺点只能使用get方法。不确定他是否失败，因为没有相关的函数，容易受到攻击.‘

cros是web服务通过不同域传来沙盒脚本的办法，避开浏览器的同源策略，他是可以支持大部分的http请求方式的.

* target：接口的域名，这里不能直接写成'http://localhost:8888/static/cnWord.json'，亲测无效~
* changeOrigin：开启代理
* pathRewrite：可以看到我使用了/api来匹配请求接口的域名，而接口名称是/static/cnWord.json，因此在实际请求中应该写成/api/static/cnWord.json，但是我实际请求的地址又是没有api前缀的，因此需要通过pathRewrite重写地址，将接口请求的时候前缀去除。

16、es6：

（arr方法，set，map，sort）：

使用Flex布局会大大降低CSS的难度

使用Flux可以很好的隔离业务（我们的业务逻辑都在Store里）,大大降低了单元测试的难度

基于React的控件（这里说的是基本控件）是对React理解的一个体现

尽量把React组件（Component）做到小，做到细，也就是尽量拆分React组件

基于数据驱动的方式（Data-Driven）开发，理解到这一点就会发现React很神奇，可能这也是最不好理解的地方

ImmutableJS带来了大量的好处，所有组件都使用了ShouldComponentUpdate来优化

实现了I18N的动态加载

使用了Url来保存Router信息，没有使用Hash，感觉很简洁

存在的一些问题

Flux的循环调用问题，可能是一个弊端，但是，总是可以让我们重新思考，我们这样的流程是否完全必要，还是有逻辑不清晰的地方。我遇到过几次循环调用的问题，最终都能通过不使用setTimeout来解决

还是存在一些冗余的代码，个人认为还是需要引入Model层（虽然我们没有这么做），因为我们是做的数据驱动，所以，有些model内容的验证（如表单验证）可能做在Model里会好些

还需要改进的地方

Webpack打包的size还是有些大，可能代码有些冗余

WebSocket应该在后续加入系统，可能使用Socket.io

CSS可能还是需要规范，主要参考BEM,OOCSS等规范

增加更多的动画效果

开发一套属于自己的组件库，逐步抛弃material-ui库

js的设计模式

1.单体（Singleton）模式： 绝对是JavaScript中最基本最有用的模式。

2.工厂（Factory）模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。

3.桥接（bridge）模式：在实现API的时候，桥梁模式灰常有用。在所有模式中，这种模式最容易立即付诸实施。

4.装饰者（Decorator）模式：这个模式就是为对象增加功能(或方法)。

5.组合（Composite）模式：将对象组合成树形结构以表示“部分-整体”的层次结构。它使得客户对单个对象和复合对象的使用具有一致性。

6.门面（facade）模式：门面模式是几乎所有JavaScript库的核心原则

7.适配置器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作，使用这种模式的对象又叫包装器，因为他们是在用一个新的接口包装另一个对象。

8.享元（Flyweight）模式：运用共享技术有效地支持大量细粒度的对象。

9.代理（Proxy）模式：此模式最基本的形式是对访问进行控制。代理对象和另一个对象（本体）实现的是同样的接口，可是实际上工作还是本体在做，它才是负责执行所分派的任务的那个对象或类，代理对象不会在另以对象的基础上修改任何方法，也不会简化那个对象的接口。

10.观察者（Observer）模式：定义对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新。

11.命令（Command）模式：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作

12.职责链（Chain Of Responsibility）模式：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。

箭头函数与普通函数的区别

1.箭头函数是匿名函数，不能作为构造函数，不能使用new

2.箭头函数不绑定arguments，取而代之用rest参数...解决

3.箭头函数不绑定this，会捕获其所在的上下文的this值，作为自己的this值

4.箭头函数通过 call() 或 apply() 方法调用一个函数时，只传入了一个参数，对 this 并没有影响。

5.箭头函数没有原型属性

6.箭头函数不能当做Generator函数,不能使用yield关键字

总结

箭头函数的 this 永远指向其上下文的 this ，任何方法都改变不了其指向，如 call() , bind() , apply()

普通函数的this指向调用它的那个对象

redux缺点

1、 action和reducer太繁琐。一套或者几套action和reducer的组合，看起来很不错，但是一旦功能和需求多了，

action和reducer就会很混乱，如果管理不善，都不能愉悦的写代码了。

比如你一个人开发你自己的博客的时候，独自面对大量的action和reducer和store，不知道会不会头疼。所以redux是不适合小场景的。微服务好，也不是什么系统都适合的。

2、store和state的模棱两可。没有严格的定义哪些存store，哪些存internal state。如果不是资深redux玩家，想必也说不出个所以然来。

3、dispatch是同步的，而且dispatch没办法确认action是否执行成功

首先要承认的是redux是非常棒的框架，但是只适合资深redux玩家和中大型项目。

redux 阻止组件渲染

1.shouldComponentUpdate

2.Component和PureComponent有一个不同点

除了为你提供了一个具有浅比较的shouldComponentUpdate方法，PureComponent和Component基本上完全相同。当props或者state改变时，PureComponent将对props和state进行浅比较。另一方面，Component不会比较当前和下个状态的props和state。因此，每当shouldComponentUpdate被调用时，组件默认的会重新渲染。

不要在props和state中改变对象和数组，如果你在你的父组件中改变对象，你的“pure”子组件不将更新。虽然值已经被改变，但是子组件比较的是之前props的引用是否相同，所以不会检测到不同。

因此，你可以通过使用es6的assign方法或者数组的扩展运算符或者使用第三方库，强制返回一个新的对象。

　强制返回一个新的对象1、ES6的扩展语法Object.assign()

　　2、深拷贝/浅拷贝或利用JSON.parse(JSON.stringify(data))相当于深拷贝，但使用受一定限制

　　3、引入immutable.js react官方推荐的第三方库

redux 性能优化

1. render里面尽量减少新建变量和bind函数，传递参数是尽量减少传递参数的数量。

2.定制shouldComponentUpdate函数

3.Immutable.js

4.多个react组件性能优化，key的优化

5.redux性能优化：reselect（数据获取时优化）我们都是优化渲染来提高性能的，

既然react和redux都是通过数据驱动的的方式驱动渲染过程，那么处理优化渲染过程，获取数据的过程也是需要考虑的一个优化点。

当数组比较大的时候，则会降低性能。

这个时候，reselect就应运而生了，它的动作原理：只要相关的状态没有改变，那么就直接使用上一次的缓存结果。

http三次握手

第一次握手

        建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN\_SEND状态，等待服务器的确认；

第二次握手

        服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，此时服务器进入SYN\_RECV状态；

第三次握手

       客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。

为什么要三次握手

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

具体例子：“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”

---------------------

四次挥手

第一次分手

主机1（可以使客户端，也可以是服务器端），设置Sequence Number，向主机2发送一个FIN报文段；此时，主机1进入FIN\_WAIT\_1状态；这表示主机1没有数据要发送给主机2了；

第二次分手

主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为Sequence Number加1；主机1进入FIN\_WAIT\_2状态；主机2告诉主机1，我“同意”你的关闭请求；

第三次分手

主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入LAST\_ACK状态；

第四次分手

es6新特性

这是为忙碌的开发者准备的ES6中最棒的十个特性（无特定顺序）：

默认参数

模版表达式

多行字符串

结构赋值

改进的对象表达式

箭头函数 =&>

Promise

块级作用域的let和const

类

模块化

主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME\_WAIT状态；主机2收到主机1的ACK报文段以后，就关闭连接；此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

为什么要四次分手

       TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。

表单验证

getFieldDecorator：getFieldDecorator是一个方法，这个方法接收两个参数，第一个是表单的字段对象，第二个是验证规则。这个方法本身返回一个方法，需要将需要获取值的标签包裹进去

中间键的作用

因为dispatch处理action时异步处理的不好，所以引用中间件。

Diff算法

DIFF算法在执行时有三个维度，分别是Tree DIFF、Component DIFF和Element DIFF，执行时按顺序依次执行，它们的差异仅仅因为DIFF粒度不同、执行先后顺序不同。

1. Tree DIFF是对树的每一层进行遍历，如果某组件不存在了，则会直接销毁
2. Element DIFF紧接着以上统一类型组件继续比较下去，常见类型就是列表

总结：diff比较新旧节点的时候只在同层比较，不会跨层级比较，key作用是高效比较虚拟dom