https://juejin.cn/post/6926794697553739784#heading-22

1. Interface 和 type的区别

type和interface都可以使用泛型，都可以描述一个对象和函数

type并没有创建一个真正的类型，而是创建了一个对名字指向了一个类型，

type可以声明基本类型别名，比如元组类型，布尔类型，

type可以使用typeof获取实例的类型

个人认为两种场景可能会使用到type而不能用interface：

具体定义数组每个位置的类型  
type PetList = [Dog, Pet]

限定具体几个值的基本类型联合类型  
type someAnimal = 'dog' | 'cat'

interface可以声明合并，这里的意思是，声明两个一样的interface会联合里面的所有属性，而声明两个type会报错

interface可以是用extends，implements运算符，type不可以但是可以使用联合类型达到类似的效果

所以一般不需用面向对象的场景或者不需要修改全局类型的场合，我一般都是用 type 来定义类型

1. never有什么作用？

never表示从不发生的类型，是任何类型的子集，可以检查一些unreachable 的代码，比如，一个从来不会有返回值的函数，一个总是会排除错误的函数，使用never避免出现了新增联合类型但是没有对应实现的情况，目的是保证写出来的代码绝对安全，（Exhaustive Checks）

<https://www.zhihu.com/question/354601204>

1. node中的子进程

https://juejin.cn/post/6844903998068097037

子进程的创建方式有4中，分别是spawn，fork，exec，execFile，后面三个方法都是spawn的扩展应用。

他们之间的区别，exec和execFile类似，都有回调函数。进程的类型也可以是任意类型，不同的是exec执行的是命令,会衍生shell，然后在shell中执行命令，相当于又开了一个窗口，执行了命令，execFile执行的是可执行文件文件，默认不会衍生shell，可执行文件会被直接衍生作为新进程，比exec高效一些。spawn和fork都没有回调函数，spawn执行的也是命令，可以使任意进程类型，fork是spawn的特例，只能执行node进程，并且是文件，返回的是ChildProcess对象，并且内置了通信通道，允许父进程和子进程之间通信（其中options参数中的serialization用于指定进程之间发送消息的序列化类型，默认为json）。

进程之间的通讯：

子进程对象使用send()方法实现想子进程发送数据，message事件实现收听子进程发送来的数据。

主线程和子线程之间通过onmessage和postMessage进行通讯。

实现进程间的通讯（IPC）的方式有很多，命名管道，匿名管道，信号量，socket，共享内存，消息队列

node中是用管道技术实现的进程间通讯（IPC），父进程在实际创建子进程之前，会创建IPC通道并且监听他，然后才真正创建出子进程比通过环境变量告诉子进程这个IPC通道的文件描述符，子进程在启动的过程中，根据文件描述符去连接这个已存在的IPC通道，从而完成父子进程之间的链接。

进程间通讯原理：

子进程对象send（）方法可以发送的句柄类型包括以下几种：

Net.socket TCP套接字

Net.server TCP服务器，

Net.Native C++层面的TCP套接字

Dgram。Socket UDP套接字

send方法在将消息发送到IPC管道前，将消息组装成两个对象，一个参数是handle，另一个是message

Message: {

Cmd :’NODE\_HANDLE’,

Type: ‘net.Server’,

Mas: message,

}

发送到IPC通道中的实际上是我们要发送的句柄文件描述符，是一个整数值，在写入到IPC通道之前会对message进行JSON.stringfy()序列化，最终发送到IPC通道的都是字符串。子进程通过IPC通道读取到父进程发送的消息，通过JSON.parse解析还原为对象后，才触发message事件，然后在使用消息。

1. node中的工作线程

Work\_threads可以共享内存，通过传输ArrayBuffer实例或共享SharedArrayBuffer实例来实现

1. TCP三次握手，四次挥手

浏览器发送一个syn（同步序列确认号）给服务器，期望建立连接，并进去syn\_send状态

服务端收到syn包，发送syn+ack包确认收到建立连接的请求，并进入syn\_rec状态

浏览器收到syn+ack包，发送ack包，服务器端收到后，双方进入established状态

浏览器发送一个FIN包，期待结束连接，此时浏览器仍然能接受数据，进去FIN\_wait状态

服务端收到FIN包，发送ACK包，表示确认收到，但是服务端可能还有数据需要传递，进入CLOSE\_wait状态

等服务端发送完数据之后，服务端发送FIN+ACK包，进入LAST\_ACK状态

浏览器收到FIN+ACK包后，发送ACK包，双方进入CLOSE状态

1. 拥塞控制，流量控制

流量控制指的是，发送方的发送滑动窗口的改变，接收方在收到数据后会在发送确认包的时候，额外加上一个窗口大小的信息，告诉发送方此时自己的缓存还剩多少，发送方根据这个窗口信息的大小调整自己的发送窗口。一般来说接受窗口>=发送窗口，因为接受放在发送完接受窗口信息会继续处理缓存中的信息。

拥塞控制主要是由于网络状态影响的，发送方在发送完数据后，一直没有等待接收方的ACK确认，就会认为出现了丢包情况，但是又可能是数据在网络中发生了拥堵，发送方会在再次发送一个数据包，依次重复，这样越来越多的数据包在网络中会造成更拥堵的情况。所以这个时候需要，探测网络的拥堵情况。

发送方发送1个数据包，看看有没有超时，在发送2个看看有没有超时，在发送4个看看有没有超时，依次发送8,16,32个，指数增长，等到到一个阈值的时候（我们自己设置的），就开始一个一个线性增长。在这个情况下，指数的增长我们称为慢启动，线性的增长我们称为避免拥塞。

不管是指数的增长还是线性的增长都会最终发生超时情况，此时我们把阈值设置为最大超时发送量的一般，继续开始上面的探测网络拥堵情况的测试。

1. rollup和webpack

rollup基于es6模块系统，同时支持tree shaking，配置上相对轻量。针对js库级别的应用非常适合使用rollup。对应业务场景丰富的应用，如需要各种loader来处理图片、css、字体等、需要按需加载，代码分割还是webpack更适合

8. tree shaking

基于ES6的静态引用，tree shaking 通过扫描所有 ES6 的export，找出被import 的内容并添加到最终代码中。 webpack 的实现是把所有import 标记为有使用/无使用两种，在后续压缩时进行区别处理。

webpack 负责对代码进行标记，把import&export标记为 3 类：

1. 所有import标记为/\* harmony import \*/
2. 被使用过的export标记为/\* harmony export ([type]) \*/，其中[type]和 webpack 内部有关，可能是binding, immutable等等。
3. 没被使用过的export标记为/\* unused harmony export [FuncName] \*/，其中 [FuncName]为export的方法名称

之后在 Uglifyjs (或者其他类似的工具) 步骤进行代码精简，把没用的都删除。

1. react diff

// https://juejin.cn/post/6844903944796258317

传统的diff算法是使用循环递归对节点进行依次对比，这边我们知道递归函数的特点，内存使用量大，需要保存每一层递归的数据栈，时间复杂度较高。

现在的diff算法，主要重3个方面做了diff策略

Tree diff：对于DOM节点中的跨层级的操作进行忽略，这个意思就是，比如对于两棵树只对同一层级进行比较，只要有一个节点不存在了，就把这个节点和他的所有子节点都删除，只需要层次遍历一次就能完成对DOM树的比较

Component diff：

同一类型的组件，按照原定的tree diff策略进行比较

同类型组件，比如组件A转换为组件B，如果virtual Dom无变化，可以通过shouldComponentUpdate方法判断

不同类型的组件，diff算法会把要改变的组件判断为dirty component，替换整个组件的所有节点

Element diff：

对于同一层级的组件，提供3中操作节点的方法：插入，移动，删除

· 插入： 新的组件不在原来的集合中，而是全新的节点，则对集合进行插入操作。

· 删除：组件已经在集合中，但集合已经更新，此时节点就需要删除。

· 移动：组件已经存在于集合中，并且集合更新时，组件并没有发生更新，只是位置发生改变，例如：(A,B,C,D) → (A,D,B,C), 如果为传统diff则会在检测到旧集合中第二位为B，新集合第二位为D时删除B，插入D，并且后面的所有节点都要重新加载，而 React diff 则是通过向同一层的节点添加 唯一key 进行区分，并且移动。

1. React中key的作用

key属性的使用，则涉及到diff算法中同级节点的对比策略，当我们指定key值时，key值会作为当前组件的id，diff算法会根据这个id来进行匹配。如果遍历新的dom结构时，发现组件的id在旧的dom结构中存在，那么react会认为当前组件只是位置发生了变化，因此不会将旧的组件销毁重新创建，只会改变当前组件的位置，然后再检查组件的属性有没有发生变化，

1. for和forEach的性能效率
2. React 16新增加了哪些声明周期方法？去掉了哪些声明周期方法？为什么这么做？

https://juejin.cn/post/6844903600309665799

React 16新增加了getDerivedStateFromProps, getSnapshotBeforeUpdate,

在React 17将要移除的componentWillMount, componentWillReceiveProps, componentWillUpdate。

首先componentWillMount的使用场景，为了避免第一次渲染的时候页面因为没有获取到异步数据导致白屏，而将部分代码放在了componentWillMount，但是componentWillMount执行完之后，并不会等到异步数据返回就会继续向下进行，页面的首次渲染仍然会处于没有异步数据的状态，就是不管在什么生命周期执行，都会出现首屏因为没有异步数据而白屏的现象。另外React官方建议也是将首屏的异步请求放在constructor中。另一个场景是，在componentWillMount中，订阅时间，在componentWillUnmount中，取消订阅，但是在React开启异步渲染模式后（concuttent模式）后，渲染是可以被打断的，也就是会导致componentWillUnmount不会被调用。而componentDidMount这个生命周期出现在Render之后，之后一定会调用componentWillUnmount，所以最好的解决方案是在componentDidMount中注册事件。

componentWIllReceiveProps参数为nextProps。在props变化的时候，我们先判断nextProps和当前的props是否相同，然后在执行之后的逻辑，有一个场景是，我们要根据props的值来更新state，比如，从外部链接跳转到列表的时候，直接根据传入的某个值，定位到某个Tab，如果componentWIllReceiveProps中，我们需要先判断nextProps和当前的props是否相同，然后在执行之后的操作，比如请求新的接口数据，这些操作都是写在componentWIllReceiveProps里面的。而在getDerivedStateFromProps中，参数直接就是nextProps和preState,给我们提供了用新的props更新state的声明周期，并且在getDerivedStateFromProps还禁止访问this.props，通过这种方式强制比较nextProps和prestate的值，防止去做一些让组件自身状态变的更加不可预测的事情。getDerivedStateFromProps返回的值会自动setState所以没有变化的时候，我们需要返回一个null。在getDerivedStateFromProps中更新了状态后，我们可以再componentDidUpdate中做我们后续的逻辑，相当于把之前componentWIllReceiveProps的逻辑分成了两个部分，逻辑颗粒化，更加利于我们维护。

componentWIllUpdate于componentWillReceviueProps类似，很多开发者也会在componentWIllUpdate中根据props的变化触发一些回调或其他逻辑，但是这两个声明周期都有可能在一次更新中被调用多次，里面的逻辑显然也会调用多次。而componentDidUpdate在一次更新中只会调用一次。

另一个场景是在组件更新之前读取DOM元素的状态，在React的异步渲染阶段render阶段（生成虚拟DOM）和commit阶段（虚拟DOM渲染到真是的DOM）并不是无缝衔接的，在render读取到的DOM的元素状态并不和commit阶段相同，getSnapshotBeforeUpdate(preProps, preState)会保证在最终commit之前调用，读取到的DOM元素状态和componentDidUpdate相同，应该尽量在getDerivedStateFromProps中返回一个值，这个值会被传到componentDidUpdate中。

1. React中的setState的执行情况

在React中创建更新，会在enqueueSetState方法中将创建的更新通过一个链表结构连接起来，并且默认会进行batchedUpdate更新优化，batchedUpdate就是批处理，不会每一次调用setState的时候，都会进行调度，进行渲染，否则如果我们在某一个方法中频繁的执行setState就会造成频繁的更新，造成性能问题。在batchedUpdated会设置一个isBatchedUpdated为true，并且有一个try,finally结构

isBatchedUpdates = true;

Try{

Return fn();

} finally {

isBatchedUpdates = false；

performSyncWork();

}

其中fn，就是经过React事件系统包装过的setState的事件，里面包括有调用requestWork，requestWork就是每次调用setState的时候，都会进行的函数调用，在requestWork里面判断到全局变量isBatchedUpdate为true就会返回，最后返回到batchedUpdated的调用栈的finally里面，执行prefromWork，进行批量更新，所以看起来直接调用setState是异步的，但是其实是React默认的批处理的性能优化，另外大部分的事件调用都会经过React的事件系统处理，默认进行batchedUpdated

还有一种情况是，包裹了setTimeout的setState调用，这个是因为setTimeout的异步执行，将事件加入事件队列执行的时候，batchedUpdate已经执行到finally中，并将isBatchedUpdated变成了false，此时会在requestWork里面执行，performSyncWork，所以是同步的。

在React中还提供了一个api，batchupdate（），如果我们将更新调用放在里面，那么所有的更新调用都会是批处理的，也就是看起来是异步的

当然，在React中的concorrent模式中，异步更新渲染，不管是在setTimeout中，还是其他情况，都是异步的。

1. Redux-thunk和Redux-saga有什么不同，优缺点？

在redux原生的action概念中，action是一个普通的js对象，Redux-thunk改变了action类型，使得action可以是一个函数，Redux-action的实现较为简单，在dispatch的时候，判断action是否是函数，如果是函数，就执行这个函数，并且将dispatch和getState作为参数传入进去。在下一次异步结果返回的时候，我们可以根据dispatch参数做一些其他操作。这样做的缺点是，所有的逻辑都集成在action中，使得action逻辑变得复杂。

saga自己独创的一套sagas体系，是基于Generator，将所有通过saga创建的操作逻辑集中处理，操作也就是Effects，在saga中的effect分为3中，1、发起一个异步调用（ajax）2.发起其他的action更新store，3. 调用其他的saga。

saga可以保证action的纯洁，action还是原来的js对象，另外saga提供了很多创建不同effects的api，可以一些async函数做不到的事情，比如无阻塞并发，取消请求等等。另外还有一个功能是，可以通过监听action进行打点记录，减少侵入式打点对代码的侵入程度。

1. fork是非阻塞的，非阻塞就是遇到它，不需要等它执行完, 就可以直接往下运行；

1. fork的另外一个同胞兄弟call是阻塞，阻塞的意思就是一定要等它执行完, 才可以直接往下运行；
2. fork是返回一个任务，这个任务是可以被取消的；而call就是它执行的正常返回结果！

4. take是阻塞的，主要用来监听action.type的，只有监听到了，才会继续往下执行

1. 如何计算白屏时间，首屏时间？

https://juejin.cn/post/6844904020482457613

如果只是粗略的了解页面性能，比如白屏时间，首屏时间，可以使用chrome自带的分析performance工具，其中FP(first paint) 代表从浏览器从进去页面，浏览器开始第一帧渲染的时间，也就是白屏时间。首屏时间可以用FMP（first meaning paint）粗略的表示。

如果不支持performance.time.navigationStart另外一种比较粗略的计算是，在dom 的head下面的第一个位置，记录一个时间，在head标签下面的最后一个位置，记录一个时间，计算他们之间的差值。

比较精准的计算页面性能，可以使用performance Api，提供了一些比较精细的性能指标，比如DNS解析时间，TCP链接时间，DOM loading 时间等等。计算白屏时间可以用，responseStart减去performance,.time.navigationStart，首屏时间无法确切的计算，一般计算最大图片的加载时间到performance.time.navigationStart的时间差。

1. 怎么优化白屏时间？
2. dns链接优化，预链接

加载资源的时候，首先会对域名进行dns解析，获取到ip地址后，进而和服务器进行通讯。一般来说一个dns的解析查找过程包括，浏览器缓存---系统缓存---isp dns缓存---递归搜索。一般在前端优化中与DNS有关的两点：1、减少请求次数，2、提前对DNS预获取。Dns-prefetch可以进行提前dns解析，可以先在link标签中加入<link rel="preconnect" href="https://exapmle.com/" crossorigin>,进行预先的tcp链接，并且在link标签中<link rel="dns-prefetch" href="https://example.com" />。

1. 使用HTTP2协议，

在http1.1中虽然默认的是长连接，但是对于每一个域名下的链接，浏览器会有一个限制，最多开启6个链接，这个时候的解决办法，首先可以将静态资源放到cdn上，其次，使用http2协议也可以，首先对于HTTP2的协议来说，是基于二进制格式解析，而http1.1基于文本解析，解析更高效，同时多路复用对于同一个域名只建立一个TCP链接，在逻辑上分成多个流，每个流可以传递若干消息，每个消息由若干二进制消息帧组成，不同请求之间的消息是可以顺序不同的，会对每一个消息加上一个id，在接受的对话根据id识别是哪一个请求。其他，还有头部压缩，http请求是无状态的，每一次请求都可能会带着重复的请求头，HTTP/2对头部信息采用HPACK压缩算法来减少报文头的大小。具体做法是把报文头信息中常见的名和值对应一个索引，维护了一张静态字典，index从1到61，比如把：method:GET映射成2，这样就能达到压缩头部的作用。但是对于一些动态的资源，比如，user-agent，需要维护一份可动态添加内容的共同动态字典，这份动态字典在数据传输的过程中逐步建立，index从62开始。然后将映射之后的数据用huffman编码。

1. 加载一个网页触发事件的顺序？

当我们打开网页，网页的加载过程大致为：

解析html文档结构

加载外部样式表及js脚本

执行js脚本，如果是同步的话，

DOM树渲染完成

加载未完成的其他资源

页面加载完成

对于以上过程，发生的事件有

Document.readystatechange事件，

readyState属性描述了当前文档的加载状态，在整个加载过程中document.readyState会不断变化，每次变化都会触发readystatechange事件。

readyState有一下状态uninitialized - 还未开始载入

loading - 载入中

interactive - 互动文档已经完成加载，文档已被解析，但是诸如图像，样式表和框架之类的子资源仍在加载，文档与用户可以开始交互

complete - 载入完成

DOMContentLoaded事件

DOMContentLoaded事件是docoment对象上的事件。指的是DOM已经加载完毕。

DOM树渲染完成时触发DOMContentLoaded事件，此时外部资源可能并未加载完成。

window.load事件

load事件是window对象上的事件。指的是网页资源已经加载完毕（包括但不限于DOM、图片、音频、脚本、插件资源以及CSS）

所有的资源全部加载完成会触发window 的 load事件。

1. webpack是如何打包文件的？

https://mp.weixin.qq.com/s/UWtkwNjiCybvSposnxfDsw

webpack打包文件其实返回的是一个立即执行函数，立即执行函数的入参是一个对象，对象的key就是文件的名字，value是一个经过webpack包装过函数，参数是module，\_\_webpack\_exports\_\_,\_\_webpack\_require\_\_，其中exports就是为这个文件提供一个export的能力，require就是提供一个引入其他文件的能力。在webpack返回的立即执行函数中，返回的是一个对入口文件的\_\_webpack\_\_require，之后的文件对递归的调用\_\_webpack\_require引入，在调用\_webpack\_require的时候会做一个缓存的优化，将所有加载过的资源缓存起来，之后在加载的时候可以直接冲缓存中得到。其中webpack是支持动态引入的，动态引入其实本质上就是创建了一个script标签，里面是一个jsonp请求，并通过 webpackJsonpCallback 判断加载的结果。

1. 浏览器的事件循环和nodejs的事件循环

https://mp.weixin.qq.com/s/VZwAZcmAJGWeWrqRbiveaw

浏览器的事件循环，可以分为宏任务和微任务，浏览器在执行的时候，为异步的代码提供了一个单独的运行空间，当异步的代码运行完毕以后，会将代码的回调送入到task queue（任务队列）中，等到调用栈为空的时候，再将队列中的回调函数调入栈中执行，等待栈空以及任务队列为空的时候，会不断坚持任务队列，看看是不是有任务要执行，任务队列分为宏队列和微队列，在检查任务队的时候是有优先级的，会先检查微队列，如果微队列不为空，则执行所有的微队列任务，知道微队列为空，再去检查宏队列，取出一个宏队列的任务执行，之后再去检查微队列，重复以上步骤。浏览器环境中的宏队列，包括settimeout，setinterval，和ui render

微队列包括Promise，requestAnimation

nodejs的事件循环，分为6个阶段，

Times:计时器阶段，用于处理setTimeout以及setInverval的回调函数

Pending callback: 用于执行某些系统操作的回调，比如TCP错误

Idle，prepose：Node内部使用

Poll：轮询阶段，执行队列的中的i/o队列，并检查定时器是否到时

Check：执行setImemediate的回调

我们能控制的只有times，poll，check，close callback阶段，这几个阶段都有自己的宏队列和微队列执行流程和浏览器环境一致

1. webpack打包流程

webpack的运行流程是一个串行的过程，从启动到结束会依次执行以下流程

1. 初始化参数：从配置文件或者shell语句中读取与合并参数，并得出最终的参数
2. 开始编译：用上一步得到的参数初始化一个compiler对象，并加载所有的配置插件，执行对象的run方法，开始执行编译
3. 确定入口根据配置中的entry找出所有的入口文件
4. 编译模块：从入口文件中，调用所有配置的loader对象对模块进行翻译，在找到该模块依赖的模块，递归的知道所有的入口依赖文件都进行了处理。
5. 在进行编译的时候，比如举一个使用babel-loader，es6转换成es5的例子，第一步会使用@babel/parser进行内部的语法分析，并生成一个AST（抽象语法树），第二步其实babel也提供了@babel/traverse这个包，来遍历语法树，在这个环节可以找到所有的依赖模块，在然后就是使用@babel/core里面的@babel/preset和@babel/env将AST生成es5的代码，最后对于我们上面找出来的依赖项，递归解析依赖项
6. 输出资源：根据入口文件和依赖项目的关系，组装成一个个包含多个模块的chunk，再把每个chunk转换成一个单独的文件加入到输出列表
7. 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名。