## 浏览器的渲染流程

这个问题应该就是输入url到页面呈现问题的变种，只不过此时的侧重点是获取完数据之后进行的渲染流程。

第一步，HTML转换成DOM

第二步，CSS转换成浏览器可理解的styleSheets，然后计算DOM节点的样式

第三步，创建布局树，计算元素的布局信息

第四步，对布局树进行分层，构建分层树

第五步，为每个图层生产绘制列表，并将其提交到合成线程

第六步，合成线程将图层转化为图块，进而将图块转化成位图

第七步，合成线程发送绘制命令给浏览器

第八步，浏览器根据绘制命令生成页面，并显示到显示器上。

## 输入网址到实现的过程

1. DNS 解析:将域名解析成 IP 地址
2. TCP 连接：TCP 三次握手

为什么要三次握手?

谢希仁著《计算机网络》中讲“三次握手”的目的是“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”。

1. 发送 HTTP 请求
2. 服务器处理请求并返回 HTTP 报文
3. 浏览器解析渲染页面
4. 断开连接：TCP 四次挥手

## 网页前端性能优化的方式有哪些？

1.压缩 css, js, 图片

2.减少 http 请求次数， 合并 css、js 、合并图片（雪碧图）

3.使用 CDN

4.减少 dom 元素数量

5.图片懒加载

6.静态资源另外用无 cookie 的域名

7.减少 dom 的访问（缓存 dom）

8.巧用事件委托

9.样式表置顶、脚本置低

## 移动端性能优化？

尽量使用css3动画，开启硬件加速

适当使用touch事件代替click时间

避免使用css3渐变阴影效果

可以用transform: translateZ(0) 来开启硬件加速

不滥用float。float在渲染时计算量比较大，尽量减少使用

不滥用web字体。web字体需要下载，解析，重绘当前页面

合理使用requestAnimationFrame动画代替setTimeout

css中的属性（css3 transitions、css3 3D transforms、opacity、webGL、video）会触发GUP渲染，耗电

## 节流与防抖的原理和应用场景

节流：就是限制一个动作在一段时间内只能执行一次

防抖：就是 当一个动作连续触发，只执行最后一次

/\*\*

\* 防抖

\*

\* @export

\* @param {\*} handle

\* @param {\*} delay

\* @returns

\*/

export function debounce(handle, delay) {

var timer = null;

return function () {

var \_self = this,

\_args = arguments;

clearTimeout(timer);

timer = setTimeout(function () {

handle.apply(\_self, \_args)

}, delay)

}

}

/\*\*

\* 节流

\*

\* @export

\* @param {\*} handler

\* @param {\*} wait

\* @returns

\*/

export function throttle(handler, wait) {

var lastTime = 0;

return function (e) {

var nowTime = new Date().getTime();

if (nowTime - lastTime > wait) {

handler.apply(this, arguments);

lastTime = nowTime;

}

}

}

## 浏览器的缓存

**流程**：浏览器中的缓存主要分为强缓存和协商缓存。

1. 浏览器在加载资源时，根据请求头的 expires 和 cache-control 判断是否命中强缓存，是则直 接从缓存读取资源，不会发请求到服务器。
2. 如果没有命中强缓存，浏览器一定会发送一个请求到服务器，通过 last-modified 和 etag 验证 资源是否命中协商缓存，如果命中，服务器会将这个请求返回，但是不会返回这个资源的数据，依 然是从缓存中读取资源。
3. 如果前面两者都没有命中，直接从服务器加载资源。

### 缓存机制

浏览器发送请求前，根据请求头的expires和cache-control判断是否命中（包括是否过期）强缓存策略，如果命中，直接从缓存获取资源，并不会发送请求。如果没有命中，则进入下一步。 2. 没有命中强缓存规则，浏览器会发送请求，根据请求头的last-modified和etag判断是否命中协商缓存，如果命中，直接从缓存获取资源。如果没有命中，则进入下一步。 3. 如果前两步都没有命中，则直接从服务端获取资源。

### 2、强缓存

强缓存：不会向服务器发送请求，直接从缓存中读取资源。

#### 2.1 强缓存原理

强制缓存就是向浏览器缓存查找该请求结果，并根据该结果的缓存规则来决定是否使用该缓存结果的过程，强制缓存的情况主要有三种(暂不分析协商缓存过程)，如下：

* 第一次请求，不存在缓存结果和缓存标识，直接向服务器发送请求
* 存在缓存标识和缓存结果，但是已经失效，强制缓存是啊比，则使用协商缓存（暂不分析）
* 存在该缓存结果和缓存标识，且该结果尚未失效，强制缓存生效，直接返回该结果

那么强制缓存的缓存规则是什么？ 当浏览器向服务器发起请求时，服务器会将缓存规则放入HTTP响应报文的HTTP头中和请求结果一起返回给浏览器，控制强制缓存的字段分别是Expires和Cache-Control，其中Cache-Control优先级比Expires高。

### 协商缓存

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：

* 协商缓存生效，返回304和Not Modified
* 协商缓存失效，返回200和请求结果

## cookie、sessionStorage、localStorage

### cookie

* 会话性：默认不设置过期时间，保存在内存中；关闭浏览器、用户手动清除 则cookie失效
* 持久性：设置了过期时间，保存在硬盘中；过期时间到、用户手动清除 则cookie失效
* 同一浏览器下的同一个域名下共享(同一浏览器的不同网页窗口共享)
* 单个4KB
* 每次请求都会携带在http请求头中，存储过多cookie会造成性能的浪费
* 可用于登录识别用户

### session

* 关闭浏览器的网页窗口、用户手动清除 则失效
* 同一浏览器的不同网页窗口不共享，只在当前网页窗口有效
* 单个5MB
* 仅在客户端（即浏览器）中保存，不参与和服务器的通信

### localStorage

* 长期有效，不管是关闭网页窗口还是浏览器都不会失效，只能手动清除
* 单个5MB
* 仅在客户端（即浏览器）中保存，不参与和服务器的通信

### 总结

| **类型** | **时效性** | **共享性** | **大小** | **参与** |
| --- | --- | --- | --- | --- |
| cookie | 设置过期时间，时间到则失效；未设置过期时间，浏览器关闭失效 | 同域名下共享 | 单个4KB，一般可存储50个，Opera为30个 | 参与，携带在http请求头中 |
| sessionStorage | 网页窗口关闭则失效 | 只在当前网页窗口中可访问 | 单个5MB | 不参与 |
| localStorage | 永久存储，除非手动清除 | 在特定页面协议内访问 | 单个5MB | 不参与 |

从**数据存储位置、生命周期、存储大小、写入方式、数据共享、发送请求时是否携带、应用场景**这几个方面来回答。

1.存储位置：Cookie、SessionStorage、 LocalStorage都是浏览器的本地存储。 它们的共同点：都是存储在浏览器本地的，它们的区别：cookie是由服务器端写入的，而SessionStorage、 LocalStorage都是由前端写入的，

2.生命周期：cookie的生命周期是由服务器端在写入的时候就设置好的，LocalStorage是写入就一直存在，除非手动清除，SessionStorage是页面关闭的时候就会自动清除。

3.存储大小：cookie的存储空间比较小大概4KB，SessionStorage、LocalStorage存储空间比较大，大概5M。

4.数据共享：Cookie、SessionStorage、LocalStorage数据共享都遵循同源原则，SessionStorage还限制必须是同一个页面。

5.是否携带：在前端给后端发送请求的时候会自动携带Cookie中的数据，但是SessionStorage、LocalStorage不会

总结：由于它们的以上区别，所以它们的应用场景也不同，Cookie一般用于存储登录验证信息SessionID或者token，LocalStorage常用于存储不易变动的数据，减轻服务器的压力，SessionStorage可以用来检测用户是否是刷新进入页面，如音乐播放器恢复播放进度条的功能。

## vue和react的异同

react和vue大体上是相同的，比如都使用[虚拟DOM](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9FDOM&spm=1001.2101.3001.7020" \t "/Users/gdj/Documents\\x/_blank)高效的更新视图，都提倡组件化，都实现了数据驱动视图，都使用diff算法，也都对diff算法进行了优化，都有router库实现url到组件的映射，都有状态管理等等

### 组件化的异同

相同点：react和vue都推崇组件化，通过将页面拆分成一个一个小的可复用单元来提高代码的复用率和开发效率。在开发时react和vue有相同的套路，比如都有父子组件传参，都有数据状态管理，都有前端路由等。

差异：

React推荐的做法是JSX + inline style, 也就是把 HTML 和 CSS 全都写进 JavaScript 中,即 all in js;

Vue 推荐的做法是 template 的单文件组件格式(简单易懂，从传统前端转过来易于理解),即 html,css,JS 写在同一个文件(vue也支持JSX写法)

### 2.虚拟DOM异同

相同点：

Vue与React都使用了 Virtual DOM + Diff算法， 不管是Vue的Template模板+options api 写法， 还是React的Class或者Function写法,最后都是生成render函数，而render函数执行返回VNode(虚拟DOM的数据结构，本质上是棵树)。

当每一次UI更新时，总会根据render重新生成最新的VNode，然后跟以前缓存起来老的VNode进行比对，再使用Diff算法（框架核心）去真正更新真实DOM（虚拟DOM是JS对象结构，同样在JS引擎中，而真实DOM在浏览器渲染引擎中，所以操作虚拟DOM比操作真实DOM开销要小的多）

不同点：

react函数式组件思想，在发生数据（setState）更改后，会重新生成新的虚拟dom树，然后进行新旧虚拟dom树的diff（自顶向下的全量diff）

vue是组件响应思想，采用代理监听（watcher）数据，当一个组件内数据更改，可以明确知道并响应这个组件进行diff比较（局部订阅）

#### diff算法源码异同

在处理老节点部分，都需要把节点处理 key - value 的 Map 数据结构，方便在往后的比对中可以快速通过节点的 key 取到对应的节点。同样在比对两个新老节点是否相同时，key 是否相同也是非常重要的判断标准。所以不同是 React, 还是 Vue，在写动态列表的时候，都需要设置一个唯一值 key，这样在 diff 算法处理的时候性能才最大化。

不同点：react 会自顶向下全diff。vue会跟踪每一个组件的依赖关系,不需要重新渲染整个组件树。

在react中，当状态发生改变时，组件树就会自顶向下的全diff, 重新render页面， 重新生成新的虚拟dom tree, 新旧dom tree进行比较， 进行patch打补丁方式，局部更新dom。所以react为了避免父组件更新而引起不必要的子组件更新， 可以在shouldComponentUpdate做逻辑判断，减少没必要的render， 以及重新生成虚拟dom，做差量对比过程。

在vue中， 通过Object.defineProperty 把 data 属性全部转为 getter/setter。同时watcher实例对象会在组件渲染时，将属性记录为dep, 当dep 项中的 setter被调用时，通知watch重新计算，使得关联组件更新。（vue中的diff是用双端比较的方式实现的，但是vue2与vue3有区别）

### 3.数据驱动视图

#### vue中的数据驱动视图

Vuejs的数据驱动是通过MVVM这种框架来实现的。MVVM框架主要包含3个部分:model、view和 viewModel。

* Model:指的是数据部分，对应到前端就是javascript对象
* View:指的是视图部分，对应前端就是dom
* ViewModel:就是连接视图与数据的中间件

ViewModel是实现数据驱动视图的核心，当数据变化的时候，ViewModel能够监听到这种变化，并及时的通知view做出修改。同样的，当页面有事件触发时，ViewModel也能够监听到事件，并通知model进行响应。ViewModel就相当于一个观察者，监控着双方的动作，并及时通知对方进行相应的操作。

首先，vuejs在实例化的过程中，会对遍历传给实例化对象选项中的data 选项，遍历其所有属性并使用 Object.defineProperty 把这些属性全部转为 getter/setter。

同时每一个实例对象都有一个watcher实例对象，他会在模板编译的过程中,用getter去访问data的属性，watcher此时就会把用到的data属性记为依赖，这样就建立了视图与数据之间的联系。当之后我们渲染视图的数据依赖发生改变（即数据的setter被调用）的时候，watcher会对比前后两个的数值是否发生变化，然后确定是否通知视图进行重新渲染。这样就实现了所谓的数据对于视图的驱动。

#### react的数据驱动视图

首先了解一些列内容：

* pending：当前所有等待更新的state队列。
* isBatchingUpdates：React中用于标识当前是否处理批量更新状态，默认false。
* dirtyComponent：当前所有待更新state的组件队列。

React通过setState实现数据驱动视图，通过setState来引发一次组件的更新过程从而实现页面的重新渲染(除非shouldComponentUpdate返回false)。

* setState()首先将接收的第一个参数state存储在pending队列中；（state）
* 判断当前React是否处于批量更新状态，是的话就将需要更新state的组件添加到dirtyComponents中；（组件）
* 不是的话，它会遍历dirtyComponents的所有组件，调用updateComponent方法更新每个dirty组件（开启批量更新事务）

## AST抽象

概念

AST 全称为 Abstract Syntax Tree，译为抽象语法树。在 JavaScript 中，任何一个对象（变量、函数、表达式等）都可以转化为一个抽象语法树的形式。抽象语法树本质就是一个树形结构的对象。

用途

* 常用各类转义、编译的插件中。比如最典型的 ES6 转换为 ES6 工具 、JSX 语法转换为 JavaScript 语法。即 babel 模块。
* 代码语法的检查，比如代码规范工具 ESLint 模块。
* 各类 JS/CSS/HTML 压缩工具。
* 代码的格式化、高亮。
* 代码错误提示。
* 代码自动补全。

## React最大的限制(缺点)是什么？

* React 只是一个库。它不是一个完整的框架。
* React有一个庞大的库，需要时间来理解。
* 新程序员可能很难理解和编码。
* React 使用内联模板和 JSX，这可能会很困难并充当障碍。它也使编码变得复杂。

## React调用 setState 之后发生了什么？

在代码中调用 setState 函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个 UI 界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

## react 生命周期函数

初始化阶段：

* getDefaultProps:获取实例的默认属性
* getInitialState:获取每个实例的初始化状态
* componentWillMount：组件即将被装载、渲染到页面上
* render:组件在这里生成虚拟的 DOM 节点
* componentDidMount:组件真正在被装载之后

运行中状态：

* componentWillReceiveProps:组件将要接收到属性的时候调用
* shouldComponentUpdate:组件接受到新属性或者新状态的时候（可以返回 false，接收数据后不更新，阻止 render 调用，后面的函数不会被继续执行了）
* componentWillUpdate:组件即将更新不能修改属性和状态
* render:组件重新描绘
* componentDidUpdate:组件已经更新

销毁阶段：

* componentWillUnmount:组件即将销毁

## shouldComponentUpdate 是做什么的，（react 性能优化是哪个周期函数？）

shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

## 为什么虚拟 dom 会提高性能?(必考)

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。

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，视图就更新了。

## react diff 原理（常考，必考）

* 把树形结构按照层级分解，只比较同级元素。
* 给列表结构的每个单元添加唯一的 key 属性，方便比较。
* React 只会匹配相同 class 的 component（这里面的 class 指的是组件的名字）
* 合并操作，调用 component 的 setState 方法的时候, React 将其标记为 dirty.到每一个事件循环结束, React 检查所有标记 dirty 的 component 重新绘制.
* 选择性子树渲染。开发人员可以重写 shouldComponentUpdate 提高 diff 的性能。

## React组件怎样可以返回多个组件

* 使用HOC（高阶函数）
* 使用React.Fragment,可以让你将元素列表加到一个分组中，而且不会创建额外的节点（类似vue的template)
* 使用数组返回

## React中元素和组件的区别

* react组件有类组件、函数组件
* react元素是通过jsx创建的

## React性能优化手段

1. shouldComponentUpdate
2. memo
3. getDerviedStateFromProps
4. 使用Fragment
5. 循环元素使用正确的key
6. 拆分尽可能小的可复用组件，ErrorBoundary
7. 使用React.lazy和React.Suspense延迟加载不需要立马使用的组件

## 闭包问题

得分点： 变量背包、作用域链、局部变量不销毁、函数体外访问函数的内部变量、内存泄漏、内存溢出、形成块级作用域、柯里化、构造函数中定义特权方法、Vue中数据响应式Observer

标准回答：

**一个函数和词法作用域的引用捆绑在一起，这样的组合**就是闭包（closure）。一般就是一个函数A，return其内部的函数B，被return出去的B函数能够在外部访问A函数内部的变量，这时候就形成了一个B函数的变量背包，A函数执行结束后这个变量背包也不会被销毁，并且这个变量背包在A函数外部只能通过B函数访问。

闭包形成的原理：作用域链，当前作用域可以访问上级作用域中的变量

闭包解决的问题：能够让函数作用域中的变量在函数执行结束之后不被销毁，同时也能在函数外部可以访问函数内部的局部变量。

闭包带来的问题：由于垃圾回收器不会将闭包中变量销毁，于是就造成了内存泄露，内存泄露积累多了就容易导致内存溢出。

加分回答： 闭包的应用，能够模仿块级作用域，能够实现柯里化，在构造函数中定义特权方法、Vue中数据响应式Observer中使用闭包等。

## ajax、axios 前后端通信的过程（原生的ajax怎么写）

原生ajax请求步骤get：

1.创建核心的对象（new xhrhttprequest对象）

2.编写回调函数（判断是否请求成功 状态码等于4或者等于200）

3.调用open方法 设置请求方式和请求路径

4.调用send方法 发送

post：要多设置一个请求头setrequestheader（在发送的前面）

## deffer和async的区别

<script src="demo\_defer.js" defer></script>

如果defer设置了该属性，则指定脚本与解析页面并行下载，并在页面解析完成后执行。

注意：该属性仅适用于外部脚本（仅当存在defer该属性时才应使用）。 src

注意：可以通过多种方式执行外部脚本：

如果async存在：脚本与解析页面并行下载，并在可用时立即执行（在解析完成之前）

如果 defer存在（而不是async）：脚本与解析页面并行下载，并在页面完成解析后执行

如果两者都不存在async或 defer不存在：脚本被立即下载并执行，阻塞解析直到脚本完成

## map和forEach的区别

forEach返回undefined，map会返回新的数组。

forEach没办法中止循环，但是map可以通过返回false或者出错来中止。

## call、apply以及bind的区别。

call、aply和bind都可以改变函数中的this指向。

* apply和call基本类似，他们的区别只是传入的参数不同。
* apply传入的参数是包含多个参数的数组
* call传入的参数是若干个参数列表
* bind方法会创建一个新的函数，当被调用的时候，将其this关键字设置为提供的值，我们必须手动去调用

### apply：

apply() 方法调用一个函数，其具有一个指定的this值，以及作为一个数组（或者类似数组的对象）提供的参数，fun.apply(thisArg, [argsArray])

thisArg：在fun函数运行时指定的this值。指定this的值并不一定是函数执行时真正的this值，如果是原始值的this会指向该原始值的自动包装对象。

argsArray：一个数组或者类数组对象，其中的数组元素将作为单独的参数传给fun函数。参数为null或者undefined，则表示不需要传入任何参数。

### call：

call() 调用一个函数，其具有一个指定的this值，以及若干个参数列表，fun.call(thisArg, arg1, arg2, …)

thisArg：在fun函数运行时指定的this值。指定this的值并不一定是函数执行时真正的this值，如果是原始值的this会指向该原始值的自动包装对象。

arg1, arg2, …：若干个参数列表

### bind：

bind() 创建一个新的函数，当被调用时，将其this的关键字设置为提供的值，在调用新函数时，在任何提供一个给定的参数序列。

bind创建了一个新函数，必须手动去调用。

## 数组常用函数

join()：用指定的分隔符将数组每一项拼接为字符串

push() ：向数组的末尾添加新元素

pop()：删除数组的最后一项

shift()：删除数组的第一项

unshift()：向数组首位添加新元素

slice()：按照条件查找出其中的部分元素

splice()：对数组进行增删改

fill(): 方法能使用特定值填充数组中的一个或多个元素

filter():“过滤”功能

concat()：用于连接两个或多个数组

indexOf()：检测当前值在数组中第一次出现的位置索引

lastIndexOf()：检测当前值在数组中最后一次出现的位置索引

every()：判断数组中每一项都是否满足条件

some()：判断数组中是否存在满足条件的项

includes()：判断一个数组是否包含一个指定的值

sort()：对数组的元素进行排序

reverse()：对数组进行倒序

forEach()：ES5 及以下循环遍历数组每一项

map()：ES6 循环遍历数组每一项

copyWithin():用于从数组的指定位置拷贝元素到数组的另一个指定位置中

find():返回匹配的值

findIndex():返回匹配位置的索引

toLocaleString()、toString():将数组转换为字符串

flat()、flatMap()：扁平化数组

entries() 、keys() 、values():遍历数组

## 事件循环

JavaScript是单线程执行模型，执行的时候将会区分为主线程和任务队列。主线程执行完毕，会从任务队列中读取新的任务放入主线程进行执行，这个读取过程是**循环读取**，所以也叫**事件循环**。

任务队列分为**微任务和宏任务**，同层次，先执行微任务，再执行宏任务。

微任务：promise.then()、process.nextTick()

宏任务：setTimeOut()、setInterval()

## get和post请求区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被添加书签(Bookmark)，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST没有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

GET产生一个TCP数据包；POST产生两个TCP数据包。

GET与POST都有自己的语义，不能随便混用。

## 双向绑定原理

vue数据双向绑定是通过数据劫持结合发布者-订阅者模式的方式来实现的

我们已经知道实现数据的双向绑定，首先要对数据进行劫持监听，所以我们需要设置一个监听器Observer，用来监听所有属性。如果属性发上变化了，就需要告诉订阅者Watcher看是否需要更新。因为订阅者是有很多个，所以我们需要有一个消息订阅器Dep来专门收集这些订阅者，然后在监听器Observer和订阅者Watcher之间进行统一管理的。接着，我们还需要有一个指令解析器Compile，对每个节点元素进行扫描和解析，将相关指令（如v-model，v-on）对应初始化成一个订阅者Watcher，并替换模板数据或者绑定相应的函数，此时当订阅者Watcher接收到相应属性的变化，就会执行对应的更新函数，从而更新视图。

## Async和Await原理

async、await从字面上理解，async就是异步的意思，await就是等待的意思，而两者的用法上也是这样的，async用于申明一个function是异步的，而await用于等待一个异步方法执行完成

过程就是：

1，async就是一个async函数，而await只能在这个函数中使用

2，await表示这里等待await后面的操作执行完毕，再执行下一句代码

3，await后面紧跟的最好是一个定时操作或者一个异步操作

async和await的优点

1，解决了回调地狱的问题

2，支持并发执行

3，可以添加返回值 return xxx

4，可以在代码中添加try/catch捕获错误

## webpack工作原理

### 1、核心概念

（1）entry：一个可执行模块或者库的入口。

（2）chunk：多个文件组成一个代码块。可以将可执行的模块和他所依赖的模块组合成一个chunk，这是打包。

（3）loader：文件转换器。例如把es6转为es5，scss转为css等

（4）plugin：扩展webpack功能的插件。在webpack构建的生命周期节点上加入扩展hook，添加功能

### 2、webpack构建流程（原理）

从启动构建到输出结果一系列过程：

（1）初始化参数：解析webpack配置参数，合并shell传入和webpack.config.js文件配置的参数，形成最后的配置结果。

（2）开始编译：上一步得到的参数初始化compiler对象，注册所有配置的插件，插件监听webpack构建生命周期的事件节点，做出相应的反应，执行对象的 run 方法开始执行编译。

（3）确定入口：从配置的entry入口，开始解析文件构建AST语法树，找出依赖，递归下去。

（4）编译模块：递归中根据文件类型和loader配置，调用所有配置的loader对文件进行转换，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。

（5）完成模块编译并输出：递归完事后，得到每个文件结果，包含每个模块以及他们之间的依赖关系，根据entry配置生成代码块chunk。

（6）输出完成：输出所有的chunk到文件系统。

注意：在构建生命周期中有一系列插件在做合适的时机做合适事情，比如UglifyPlugin会在loader转换递归完对结果使用UglifyJs压缩覆盖之前的结果

## webpack的loader和plugin区别

【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。

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。

【Plugin】：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。

## promise 与 async await

async/await是写异步代码的新方式，以前的方法有回调函数和Promise。

　　async/await是基于Promise实现的，它不能用于普通的回调函数。

　　async/await与Promise一样，是非阻塞的。

　　async/await使得异步代码看起来像同步代码，这正是它的魔力所在

函数前面多了一个aync关键字。await关键字只能用在aync定义的函数内。async函数会隐式地返回一个promise，该promise的reosolve值就是函数return的值。(示例中reosolve值就是字符串”done”)

为什么Async/Await更好？

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）错误处理：

　　　　Async/Await 让 try/catch 可以同时处理同步和异步错误。在下面的promise示例中，try/catch 不能处理 JSON.parse 的错误，因为它在Promise中。我们需要使用 .catch，这样错误处理代码非常冗余。并且，在我们的实际生产代码会更加复杂

3）使用async/await的话，代码会变得异常简单和直观。

4）错误栈

　 如果 Promise 连续调用，对于错误的处理是很麻烦的。你无法知道错误出在哪里。

async/await中的错误栈会指向错误所在的函数。在开发环境中，这一点优势并不大。但是，当你分析生产环境的错误日志时，它将非常有用。这时，知道错误发生在makeRequest比知道错误发生在then链中要好。

5）调试

　 async/await能够使得代码调试更简单。2个理由使得调试Promise变得非常痛苦:

《1》不能在返回表达式的箭头函数中设置断点

《2》如果你在.then代码块中设置断点，使用Step Over快捷键，调试器不会跳到下一个.then，因为它只会跳过异步代码。

使用await/async时，你不再需要那么多箭头函数，这样你就可以像调试同步代码一样跳过await语句。

## 重绘和回流有什么区别

什么是回流，什么是重绘，有什么区别？

html 加载时发生了什么：

在页面加载时，浏览器把获取到的HTML代码解析成1个DOM树，DOM树里包含了所有HTML标签，包括display:none隐藏，还有用JS动态添加的元素等。

浏览器把所有样式(用户定义的CSS和用户代理)解析成样式结构体

DOM Tree 和样式结构体组合后构建render tree, render tree类似于DOM tree，但区别很大，因为render tree能识别样式，render tree中每个NODE都有自己的style，而且render tree不包含隐藏的节点(比如display:none的节点，还有head节点)，因为这些节点不会用于呈现，而且不会影响呈现的，所以就不会包含到 render tree中。我自己简单的理解就是DOM Tree和我们写的CSS结合在一起之后，渲染出了render tree。

什么是回流：

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(reflow)。每个页面至少需要一次回流，就是在页面第一次加载的时候，这时候是一定会发生回流的，因为要构建render tree。在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程成为重绘。

什么是重绘：

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。

区别：

他们的区别很大：

回流必将引起重绘，而重绘不一定会引起回流。比如：只有颜色改变的时候就只会发生重绘而不会引起回流

当页面布局和几何属性改变时就需要回流

比如：添加或者删除可见的DOM元素，元素位置改变，元素尺寸改变——边距、填充、边框、宽度和高度，内容改变

扩展：

浏览器的帮忙

所以我们能得知回流比重绘的代价要更高，回流的花销跟render tree有多少节点需要重新构建有关系

因为这些机制的存在，所以浏览器会帮助我们优化这些操作，浏览器会维护1个队列，把所有会引起回流、重绘的操作放入这个队列，等队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会flush队列，进行一个批处理。这样就会让多次的回流、重绘变成一次回流重绘。

自己的优化

靠浏览器不如靠自己，我们可以改变一些写法减少回流和重绘

比如改变样式的时候，不去改变他们每个的样式，而是直接改变className 就要用到cssText 但是要注意有一个问题，会把原有的cssText清掉，比如原来的style中有’display:none;’，那么执行完上面的JS后，display就被删掉了。

为了解决这个问题，可以采用cssText累加的方法，但是IE不支持累加，前面添一个分号可以解决。

还有添加节点的时候比如要添加一个div里面有三个子元素p，如果添加div再在里面添加三次p，这样就触发很多次回流和重绘，我们可以用cloneNode(true or false) 来避免，一次把要添加的都克隆好再appened就好了，还有其他很多的方法就不依依说了

## JS 数据类型 ?

数据类型主要包括两部分：

基本数据类型(也叫简单数据类型)：Number 、String、Boolean、BigInt、Symbol、Null、Undefined

引用数据类型(复杂数据类型)： 用Object代表，普通对象，数组，正则，日期，Math数学函数都属于Object。

两大类的**本质区别**：基本数据类型和引用数据类型它们在**内存中的存储方式不同**。

基本数据类型是直接存储在栈内存中的简单数据段，占据空间小，属于被频繁使用的数据。

引用数据类型是存储在堆内存中，占据空间大。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，会检索其在栈中的地址，取得地址后从堆中获得实体。

加分回答： Symbol是ES6新出的一种数据类型，这种数据类型的特点就是没有重复的数据，可以作为object的key。

BigInt是ES10新出的一种数据类型，这种数据类型的特点就是数据涵盖的范围大，能够解决超出普通数据类型范围报错的问题。

## 说一说Vuex是什么，每个属性是干嘛的，如何使用 ？

得分点： state、mutations、getters、actions、module、store.commit、store.dispatch

标准回答 ：

Vuex是集中管理项目公共数据的。Vuex 有state、mutations 、getters、actions、module属性。

state 属性用来存储公共管理的数据。

mutations 属性定义改变state中数据的方法， 注意：不要在mutation中的方法中写异步方法ajax，那样数据就不可跟踪了。

getters 属性可以认为是定义 store 的计算属性。就像计算属性一样，getter的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。

action属性类似于mutation，不同在于：Action 提交的是 mutation，而不是直接变更状态。Action 可以包含任意异步操作。

moudle属性是将store分割成模块。每个模块拥有自己的state、mutation、action、getter、甚至是嵌套子模块，从上至下进行同样方式的分割

使用方法：

state：直接以对象方式添加属性

mutations ：通过store.commit调用

action：通过 store.dispatch方法触发

getters：直接通过store.getters.调用

加分回答：可以使用辅助函数mapState、mapMutations、mapAction、mapGetters一次性获取每个属性下对应的多个方法。 VueX在大型项目中比较常用，非关系组件传递数据比较方便。

### 工作流程

（1）通过dispatch去提交一个actions，

（2） actions接收到这个事件之后，在actions中可以执行一些异步|同步操作，根据不同的情况去分发给不同的mutations，

（3）actions通过commit去触发mutations，

（4）mutations去更新state数据，state更新之后，就会通知vue进行渲染

## 说一说你是怎么理解BFC的？

得分点： 块级格式化上下文、独立的渲染区域、不会影响边界以外的元素、形成BFC条件、float、position、overflow、display

标准回答： BFC(Block Formatting Context)块级格式化上下文，是Web页面一块独立的渲染区域，内部元素的渲染不会影响边界以外的元素。

BFC布局规则：

内部盒子会在垂直方向，一个接一个地放置。

Box垂直方向的距离由margin决定。属于同一个BFC的两个相邻Box的margin会发生重叠。

每个盒子（块盒与行盒）的margin box的左边，与包含块border box的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。

BFC的区域不会与float box重叠。

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。

计算BFC的高度时，浮动元素也参与计算。

BFC形成的条件:

根元素：body；

元素设置浮动：float 除 none 以外的值；

元素设置绝对定位：position(absolute、fixed)；

display值为：inline-block、table-cell、table-caption、flex 等；

overflow值为：hidden、auto、scroll；

BFC解决能的问题：

解决 margin 的重叠问题：由于 BFC 是一个独立的区域，内部的元素和外部的元素互不影响，将两个元素变为两个 BFC，就解决了 margin 重叠的问题。

解决高度塌陷的问题：在对子元素设置浮动后，父元素会发生高度塌陷，也就是父元素的高度变为 0。解决这个问题，只需要把父元素变成一个BFC。常用的办法是给父元素设置overflow:hidden。

创建自适应两栏布局：可以用来创建自适应两栏布局：左边的宽度固定，右边的宽度自适应。

## 说一说promise是什么与使用方法？

得分点： pendding、rejected、resolved、微任务、then、catch、Promise.resolve()、Promise.reject()、Promise.all() Promise.any()、Promise.race()

标准回答：

Promise的作用：Promise是异步微任务，解决了异步多层嵌套回调的问题，让代码的可读性更高，更容易维护

Promise使用：Promise是ES6提供的一个构造函数，可以使用Promise构造函数new一个实例，Promise构造函数接收一个函数作为参数，这个函数有两个参数，分别是两个函数 resolve和reject resolve将Promise的状态由等待变为成功，将异步操作的结果作为参数传递过去； reject则将状态由等待转变为失败，在异步操作失败时调用，将异步操作报出的错误作为参数传递过去。 实例创建完成后，可以使用then方法分别指定成功或失败的回调函数，也可以使用catch捕获失败，then和catch最终返回的也是一个Promise，所以可以链式调用。

Promise的特点：

对象的状态不受外界影响（Promise对象代表一个异步操作，有三种状态）。 - pending（执行中） - Resolved（成功，又称Fulfilled） - rejected（拒绝） 其中pending为初始状态，fulfilled和rejected为结束状态（结束状态表示promise的生命周期已结束）。

一旦状态改变，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（状态凝固了，就不会再变了，会一直保持这个结果）： - 从Pending变为Resolved - 从Pending变为Rejected

resolve 方法的参数是then中回调函数的参数，reject 方法中的参数是catch中的参数

then 方法和 catch方法 只要不报错，返回的都是一个fullfilled状态的promise

加分回答 ：Promise的其他方法：

Promise.resolve() :返回的Promise对象状态为fulfilled，并且将该value传递给对应的then方法.

Promise.reject()：返回一个状态为失败的Promise对象，并将给定的失败信息传递给对应的处理方法。

Promise.all()：返回一个新的promise对象，该promise对象在参数对象里所有的promise对象都成功的时候才会触发成功，一旦有任何一个iterable里面的promise对象失败则立即触发该promise对象的失败。

Promise.any()：接收一个Promise对象的集合，当其中的一个 promise 成功，就返回那个成功的promise的值。

Promise.race()：当参数里的任意一个子promise被成功或失败后，父promise马上也会用子promise的成功返回值或失败详情作为参数调用父promise绑定的相应句柄，并返回该promise对象。

## 说一说跨域是什么？如何解决跨域问题？

得分点： 同源限制、协议、域名、端口、CORS、node中间件、JSONP、postmessage

标准回答 跨域：当前页面中的某个接口请求的地址和当前页面的地址如果协议、域名、端口其中有一项不同，就说该接口跨域了。 跨域限制的原因：浏览器为了保证网页的安全，出的同源协议策略。 跨域报错信息：

跨域解决方案

cors：目前最常用的一种解决办法，通过设置后端允许跨域实现。 res.setHeader(‘Access-Control-Allow-Origin’, ‘\*’); res.setHeader(“Access-Control-Allow-Methods”, “GET, PUT, OPTIONS,POST”);

node中间件、nginx反向代理：跨域限制的时候浏览器不能跨域访问服务器，node中间件和nginx反向代理，都是让请求发给代理服务器，静态页面面和代理服务器是同源的，然后代理服务器再向后端服务器发请求，服务器和服务器之间不存在同源限制。

JSONP：利用的原理是script标签可以跨域请求资源，将回调函数作为参数拼接在url中。后端收到请求，调用该回调函数，并将数据作为参数返回去，注意设置响应头返回文档类型，应该设置成javascript。

postmessage：H5新增API，通过发送和接收API实现跨域通信。

加分回答 跨域场景：前后端分离式开发、调用第三方接口

## 说一说JavaScript有几种方法判断变量的类型？

得分点： typeof、instanceof、Object.prototype.toString.call()（对象原型链判断方法）、 constructor (用于引用数据类型)

标准回答：

JavaScript有4种方法判断变量的类型，分别是typeof、instanceof、Object.prototype.toString.call()（对象原型链判断方法）、 constructor (用于引用数据类型)

typeof：常用于判断基本数据类型，对于引用数据类型除了function返回’function‘，其余全部返回’object’。

instanceof：主要用于区分引用数据类型，检测方法是检测的类型在当前实例的原型链上，用其检测出来的结果都是true，不太适合用于简单数据类型的检测，检测过程繁琐且对于简单数据类型中的undefined, null, symbol检测不出来。

constructor：用于检测引用数据类型，检测方法是获取实例的构造函数判断和某个类是否相同，如果相同就说明该数据是符合那个数据类型的，这种方法不会把原型链上的其他类也加入进来，避免了原型链的干扰。

Object.prototype.toString.call()：

适用于所有类型的判断检测，检测方法是Object.prototype.toString.call(数据) 返回的是该数据类型的字符串。

这四种判断数据类型的方法中，各种数据类型都能检测且检测精准的就是Object.prototype.toString.call()这种方法。

加分回答

instanceof的实现原理：验证当前类的原型prototype是否会出现在实例的原型链\_\_proto\_\_上，只要在它的原型链上，则结果都为true。因此，instanceof 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 prototype，找到返回true，未找到返回false。

Object.prototype.toString.call()原理：Object.prototype.toString 表示一个返回对象类型的字符串，call()方法可以改变this的指向，那么把Object.prototype.toString()方法指向不同的数据类型上面，返回不同的结果

## 说一说样式优先级的规则是什么？

得分点： !important、行内样式、嵌入样式、外链样式、id选择器、类选择器、标签选择器、复合选择器、通配符、继承样式

标准回答

CSS样式的优先级应该分成五大类

第一类!important，无论引入方式是什么，选择器是什么，它的优先级都是最高的。

第二类引入方式，行内样式的优先级要高于嵌入和外链，嵌入和外链如果使用的选择器相同就看他们在页面中插入的顺序，在后面插入的会覆盖前面的。

第三类选择器，选择器优先级：id选择器>（类选择器 | 伪类选择器 | 属性选择器 ）> （后代选择器 | 伪元素选择器 ）> （子选择器 | 相邻选择器） > 通配符选择器 。

第四类继承样式，是所有样式中优先级比较低的。

第五类浏览器默认样式优先级最低。

## Vue 的生命周期

Vue实例有⼀个完整的⽣命周期，也就是从**开始创建、初始化数据、编译模版、挂载Dom -> 渲染、更新->渲染、卸载**等⼀系列过程，称这是Vue的⽣命周期。

1.beforeCreate（创建前）：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不 能访问到 data、computed、watch、methods 上的方法和数据。

2.created（创建后） ：实例创建完成，实例上配置的 options 包 括 data、computed、watch、methods 等都配置完成，但是此时渲染 得节点还未挂载到 DOM，所以不能访问到 $el 属性。

3.beforeMount（挂载前）：在挂载开始之前被调用，相关的 render 函数首次被调用。实例已完成以下的配置：编译模板，把 data 里面 的数据和模板生成 html。此时还没有挂载 html 到页面上。

4.mounted（挂载后）：在 el 被新创建的 vm.$el 替换，并挂载到实 例上去之后调用。实例已完成以下的配置：用上面编译好的 html 内 容替换 el 属性指向的 DOM 对象。完成模板中的 html 渲染到 html 页 面中。此过程中进行 ajax 交互。

5.beforeUpdate（更新前）：响应式数据更新时调用，此时虽然响应 式数据更新了，但是对应的真实 DOM 还没有被渲染。

6.updated（更新后）：在由于数据更改导致的虚拟 DOM 重新渲染和 打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调 用时，组件 DOM 已经更新，所以可以执行依赖于 DOM 的操作。然而 在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更 新无限循环。该钩子在服务器端渲染期间不被调用。

7.beforeDestroy（销毁前）：实例销毁之前调用。这一步，实例仍 然完全可用，this 仍能获取到实例。

8.destroyed（销毁后）：实例销毁后调用，调用后，Vue 实例指示 的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例 也会被销毁。该钩子在服务端渲染期间不被调用。

另外还有 keep-alive 独有的生命周期，分别为 activated 和

deactivated。用 keep-alive 包裹的组件在切换时不会进行销毁，而 是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执 行 activated 钩子函数。

## 什么是useCallback和useMemo？

useCallback 和 useMemo 都是react可用于性能优化的内置hooks。

两者的区别在于：useCallback缓存的是一个函数，而useMemo缓存的是计算结果。

### 为什么使用useCallback和useMemo?

在函数式组件中，每次UI的变化，都是通过重新执行整个函数来完成的，这和传统的类组件有很大区别：函数组件中并没有一个直接的方式在多次渲染之间维持一个状态。

在重新执行整个函数组件的过程中，其中的函数和引用类型的变量会创建新的（指向新的引用），导致函数组件在re-render前后，其中函数和引用类型变量是不相等的，这又会导致其他非必要的re-render。

### 什么时候使用useCallback和useMemo?

#### useCallback

当子组件接收一个函数props时，一般会使用useCallback来缓存这个函数，减少不必要的re-render。以下例子：向子组件传递一个函数，在父组件每次re-render的时候，函数会重新创建新的，这会导致使用这个函数props的子组件也re-render，但这是不必要的，可以用useCallback来解决。

#### useMemo

useMemo常用在以下两种场景的优化中：

1. 引用类型的变量
2. 需要大量时间执行的计算函数。

useMemo 和 useCallback 接收的参数都是一样,第一个参数为回调 第二个参数为要依赖的数据

### 异同点

共同作用：

1.仅仅 依赖数据 发生变化, 才会重新计算结果，也就是起到缓存的作用。

两者区别：

1.useMemo 计算结果是 return 回来的值, 主要用于 缓存计算结果的值 ，应用场景如： 需要 计算的状态

2.useCallback 计算结果是 函数, 主要用于 缓存函数，应用场景如: 需要缓存的函数，因为函数式组件每次任何一个 state 的变化 整个组件 都会被重新刷新，一些函数是没有必要被重新刷新的，此时就应该缓存起来，提高性能，和减少资源浪费。

注意： 不要滥用会造成性能浪费，react中减少render就能提高性能，所以这个仅仅只针对缓存能减少重复渲染时使用和缓存计算结果。

### 总结

useMemo返回的是一个变量的值，useCallback返回的是一个函数。对应到function组件最后要return的html代码部分，useMemo就是作为一个值来使用的，而useCallback则是被绑定到onClick上，作为要执行的函数。这就是它俩的本质区别。

useCallback(fn, deps) 等价于 useMemo(() => fn, deps).

## setState是同步还是异步？

**v18之前：**

React是希望setState表现为异步的，因为批量更新可以优化性能。因此在React能够管控到的地方，比如生命周期钩子和合成事件回调函数内，表现为异步。

在定时器和原生事件里，因为React管控不到，所以表现为同步。

在某些情况下，我们需要立即获取更新后的状态，这时可以使用第二个可选参数callback，在状态更新后立即执行回调函数来获取更新后的状态。例如：

this.setState({ counter: this.state.counter + 1 }, () => {

console.log(this.state.counter); // 输出更新后的值

});

**v18之后：**

React18之后，默认所有的操作都放到批处理中，因此setState不管在那儿调用都是异步的了。

如果希望同步更新，可以使用flushSync这个API。

## 使用Hooks有踩过哪些坑？

useEffect中没有正确设置依赖数组导致死循环。

useEffect中没有清除副作用导致内存泄漏。

在条件语句和循环中使用Hooks导致报错。

闭包陷阱。

## React逻辑复用方式有哪些？

Mixin：有很多缺点，已被弃用，可以不考虑。

HOC(高阶组件)：高阶组件是一个函数，它接收一个组件作为参数并返回一个新的组件。高阶组件可以将一些通用的逻辑（如：数据获取、权限验证、错误处理等）封装到一个函数中，并将其作为高阶组件的参数传递给其他组件使用，HOC一般以withXxx命名，并可以结合装饰器优雅地使用。

Render Props：通过在组件中传递一个函数作为prop，该函数将用于渲染组件的内容。这个函数可以接收组件需要的数据和方法，并返回React元素。

Hooks：自定义Hooks，将通用逻辑封装到useXxx函数中，可以在多个组件内使用，常见的像数据请求、表单、防抖节流、拖拽等。

## useRef有什么作用？

获取DOM。

存储上一次渲染的值，可以用useRef创建一个对象来存储setState前的旧值。

## React代码层面有哪些性能优化的方式？

React.memo()：可以缓存组件的渲染结果，避免不必要的重渲染。它接受一个函数组件，并返回一个新的组件，新组件将只在props发生变化时才重新渲染。

useMemo和useCallback。

shouldComponentUpdate：在类组件中，可以通过实现shouldComponentUpdate()方法来判断组件是否需要重新渲染。SCU接收两个参数：nextProps和nextState，我们可以在这个方法中比较当前props和state与下一个props和state的变化来决定是否需要重新渲染组件。

使用React.lazy()和Suspense进行组件懒加载。

对于大型应用，可以使用不可变数据的三方库比如Immer.js结合shouldComponentUpdate来做性能优化。