# 断点续传怎么做的

断点续传是一种网络传输技术，可以在文件传输中出现中断后恢复传输而无需重新开始传输整个文件。这在文件传输较大、网络不稳定、带宽有限等情况下尤为重要。

通常情况下，实现断点续传需要客户端和服务端的相互配合。以下是一般的实现步骤：

客户端发起文件上传请求时，向服务端发送一个 HTTP 请求，包含文件名、文件大小等信息。

服务端接收到请求后，根据文件名和文件大小创建一个空文件，并返回已经上传的文件大小，如果之前有部分数据已经上传过，那么服务端可以根据已上传的大小确定从哪个位置开始上传。

客户端接收到服务端返回的已经上传的文件大小后，可以根据已上传的大小计算出下一次应该从哪个位置开始上传。

客户端将文件切分为多个小块，每个小块的大小一般为几十 KB 或几百 KB，将每个小块的数据上传到服务端，并在每个小块的末尾添加一个分界符，标记上传结束。

服务端接收到每个小块后，将其存储到文件对应的位置，等待下一次上传。

如果上传过程中出现中断，客户端可以记录已经上传的文件大小和已经上传的小块数，下次上传时从上一次中断的位置开始继续上传。

总的来说，断点续传的实现需要客户端和服务端的相互配合，并需要考虑文件上传的稳定性和可靠性。

# 秒传怎么实现

秒传是指上传一个文件时，如果已经存在相同的文件，就直接使用已经存在的文件，而不需要重新上传。实现秒传需要用到文件的唯一标识。

一种常见的实现方式是在上传文件时，先计算文件的哈希值或者MD5值，并将其作为文件的唯一标识。然后将该标识与后台数据库中已经存在的文件的标识进行比对。如果存在相同的标识，说明该文件已经上传过，那么就直接使用已经存在的文件；如果不存在相同的标识，说明该文件是一个新文件，那么就进行正常的上传操作。

在实现过程中，可以通过在前端计算文件哈希值或MD5值的方式，避免上传重复的文件，减少服务器的压力，提高文件上传的效率。同时，后台服务器也需要维护一个文件信息的数据库，用于存储已经上传过的文件的唯一标识和存储路径等信息，以便实现秒传功能。

需要注意的是，文件的唯一标识需要具备唯一性和不可修改性，否则就无法实现秒传功能。

# 箭头函数有哪些限制

箭头函数是ES6新增的语法，与传统的函数定义方式有一些不同之处，其中包括以下限制：

无法使用arguments对象，需通过剩余参数(rest parameters)或默认参数来获取传递给函数的参数列表；

无法使用new关键字调用函数，箭头函数没有自己的this值，它会捕获其所在的上下文中的this值；

无法使用yield关键字，因此箭头函数不能用作generator函数；

箭头函数没有自己的作用域，它会捕获所在上下文的作用域链(scope chain)，因此对于闭包的使用需要特别注意。

需要注意的是，虽然箭头函数有一些限制，但在很多情况下它们能够简化代码，提高可读性和可维护性，特别是在函数式编程和React等框架中得到了广泛的应用。

# 箭头函数为什么不能作为构造函数

箭头函数不能作为构造函数的原因是因为箭头函数没有自己的this值，而是从定义时的词法环境中继承this值，也就是说，箭头函数中的this值是在定义时确定的，不能通过new关键字来改变this的指向。而构造函数中，this的指向是随着实例化对象而改变的，因此箭头函数不具备构造函数的特性，不能用来创建对象。如果在箭头函数中使用了new关键字，会抛出一个TypeError错误。

# Map和Object的区别

在 JavaScript 中，Map 和 Object 都可以用来存储键值对。但是它们之间有几个关键的区别。

键的类型：Map 中的键可以是任意类型，包括原始类型和对象引用，而 Object 的键必须是字符串或者 Symbol 类型。

插入顺序：Map 保留了插入顺序，因此在迭代 Map 时会按照插入顺序返回元素，而 Object 不保留插入顺序。

大小：Map 可以使用 size 属性来获取元素的数量，而 Object 只能手动计算属性的数量。

迭代：Map 支持 for-of 循环和 forEach 方法来迭代元素，而 Object 只能使用 for-in 循环来迭代属性，但是需要手动判断是否是自身属性而不是继承属性。

性能：在非常大的数据集合下，Map 的查找和插入操作通常比 Object 更快。

综上所述，Map 更适合存储键值对，并需要保留插入顺序或使用非字符串类型作为键的情况，而 Object 则更适合存储属性。

# vue怎么实现双向绑定

Vue实现双向绑定主要是通过对模型数据的监听实现的，可以通过以下步骤进行实现：

创建一个Observer对象，对模型数据进行监听，当数据发生变化时，通知订阅者。

创建一个Directive对象，它是指令，负责将模型数据渲染到页面中。

创建一个Watcher对象，它是订阅者，当Observer对象监听到数据发生变化时，通知Watcher对象，Watcher对象再调用Directive对象，更新页面中的数据。

将Watcher对象添加到订阅者列表中，当模型数据发生变化时，Observer对象会遍历订阅者列表，通知所有的Watcher对象。

将Directive对象添加到节点上，当数据发生变化时，Directive对象会调用Watcher对象，更新页面中的数据。

Vue使用了ES5中的Object.defineProperty()方法来实现数据的劫持，通过get和set方法来实现对数据的监听，从而实现了双向绑定。当数据发生变化时，Vue会自动更新DOM中的数据。

# 介绍一下怎么做组件库和注意事项

首先，一个组件库是一个可重用的代码库，其中包含了多个组件，这些组件可以在多个应用程序中使用。构建一个高质量的组件库需要以下步骤：

确定组件的范围： 在开始构建组件库之前，您需要确定组件库所需的组件范围。您需要考虑到您的组件库将服务于哪些应用程序、所需的组件种类和组件的不同变体。

设计组件： 设计应该是在实现组件之前的一个关键步骤。为了确保所有的组件都具有一致的外观和行为，您需要进行UI设计和用户体验设计。这样可以帮助您在整个组件库中保持一致性和可维护性。

选择适当的技术栈： 您需要选择一种适合您的技术栈来实现组件库，这可能包括React、Vue、Angular等。这种技术栈应该能够提供对组件化开发的支持，并允许您实现可复用的代码。

构建组件： 您需要按照设计的规范，实现每个组件。确保代码的可重用性，并且对组件的行为进行单元测试，确保它们在不同的情况下都能够正常工作。

创建文档和样例： 组件库应该具有详细的文档和样例，以便其他开发人员能够使用和理解您的组件库。您需要创建一个文档站点，并提供文档和示例代码，以便其他开发人员能够快速上手。

在构建组件库时需要注意的一些事项包括：

组件的可重用性： 组件应该是可重用的，以便在不同的应用程序中使用。

组件的一致性： 组件应该在整个组件库中保持一致的外观和行为。

组件的可维护性： 组件应该是易于维护的，以便您可以轻松地对其进行修改和更新。

文档和样例的重要性： 组件库的文档和样例对于其他开发人员的使用非常重要。您应该创建一个易于理解和使用的文档和示例。

单元测试的重要性： 在构建组件时，进行单元测试非常重要。这可以确保组件在各种情况下都能够正常工作，并且可以减少在开发过程中出现的错误。

# ts是如何编译成js，js又如何在浏览器运行，介绍一下流程

TypeScript 编译成 JavaScript 的流程：

编写 TypeScript 代码：开发人员使用 TypeScript 语言编写代码，使用 TypeScript 所提供的类型检查、面向对象等特性。

编译 TypeScript 代码：在编写 TypeScript 代码后，需要使用 TypeScript 编译器将 TypeScript 代码编译成 JavaScript 代码。编译器会将 TypeScript 代码转换为 JavaScript 代码，并在转换的过程中进行类型检查等操作。

生成 JavaScript 代码：编译器会生成 JavaScript 代码文件，这些文件可以直接在浏览器或 Node.js 等 JavaScript 运行环境中运行。

JavaScript 在浏览器中运行的过程：

下载页面：浏览器通过网络下载 HTML、CSS 和 JavaScript 文件等页面资源。

解析 HTML：浏览器解析 HTML 文件，并构建出 DOM 树。

解析 CSS：浏览器解析 CSS 文件，并构建出 CSSOM 树。

JavaScript 解析与执行：浏览器解析 JavaScript 代码，并执行其中的代码逻辑，修改 DOM 树和 CSSOM 树，以及与服务器进行数据交互等操作。

渲染页面：浏览器将 DOM 树和 CSSOM 树合并，生成渲染树，并将其渲染到屏幕上。

页面交互：用户与页面进行交互，页面响应用户的操作，并根据用户的输入进行相应的处理。

综上所述，TypeScript 代码需要通过编译器将其编译成 JavaScript 代码，而 JavaScript 代码则需要在浏览器中下载、解析和执行，才能最终呈现出页面内容，并与用户进行交互。

# js在浏览器如何更优化快速运行

压缩 JavaScript 文件：将 JavaScript 文件进行压缩，可以减小文件大小，提高页面加载速度。压缩的方式包括删除注释、空格等无关内容，以及对变量、函数名进行简化等。

减少 HTTP 请求：通过将多个 JavaScript 文件合并成一个文件，可以减少 HTTP 请求的次数，提高页面加载速度。同时，也可以使用浏览器缓存机制，减少 JavaScript 文件的加载次数。

懒加载 JavaScript：对于一些不是必须立即加载的 JavaScript 文件，可以使用懒加载的方式，在用户需要时再进行加载，避免不必要的加载和执行。

使用 Web Workers：在进行大量计算或处理复杂任务时，可以使用 Web Workers 在后台进行运算，避免阻塞主线程，提高页面响应速度和用户体验。

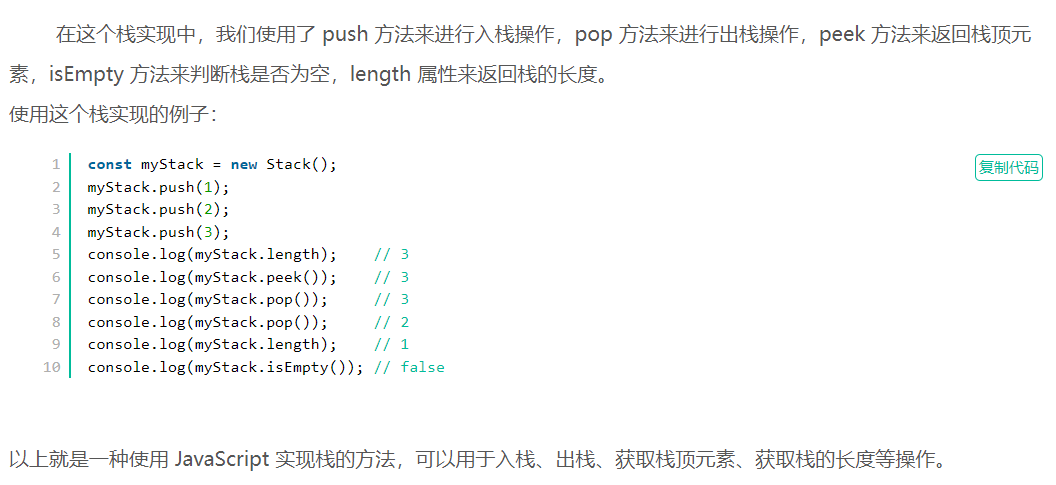
使用缓存：在 JavaScript 中可以使用缓存，将一些常用的数据进行缓存，避免重复计算和获取，提高代码执行效率。

使用事件委托：对于一些常见的事件，可以使用事件委托的方式，将事件绑定在父元素上，避免绑定过多的事件处理函数，提高性能。

避免使用全局变量：在 JavaScript 中，全局变量的访问速度较慢，同时也容易造成命名冲突等问题。因此，尽可能避免使用全局变量，采用模块化的方式组织代码。

综上所述，通过压缩 JavaScript 文件、减少 HTTP 请求、懒加载 JavaScript、使用 Web Workers、使用缓存、使用事件委托、避免使用全局变量等方式，可以有效地提高 JavaScript 在浏览器中的执行效率和性能，从而达到更优化和快速运行的效果。





# 权限怎么设计的

权限设计是现代web应用程序开发中的重要方面，它确保了用户只能访问他们被授权访问的资源。权限通常基于用户角色或权限组来定义。以下是一些在前端应用程序中设计权限的方法：

基于角色的访问控制：这是一种常用的方法，其中每个用户被分配到一个或多个角色，每个角色有一组允许访问的权限。例如，管理员角色可以访问所有资源，而普通用户角色只能访问受限资源。在前端应用程序中，可以使用路由守卫和组件级别的访问控制来实现基于角色的访问控制。

基于资源的访问控制：这是一种更细粒度的方法，其中每个资源都有一组允许访问的角色或用户。例如，某个特定的页面只允许管理员访问。在前端应用程序中，可以使用路由守卫和组件级别的访问控制来实现基于资源的访问控制。

基于功能的访问控制：这是一种将权限与应用程序的特定功能相关联的方法。例如，某个特定的按钮或操作只允许管理员执行。在前端应用程序中，可以使用条件渲染和事件处理程序来实现基于功能的访问控制。

无论采用哪种方法，都需要在服务器端进行认证和授权。前端应用程序可以使用令牌来验证用户身份，并将令牌传递给服务器来获取资源。在某些情况下，可以使用JWT（JSON Web Tokens）来实现跨应用程序的认证和授权。

# V8垃圾回收机制

V8是一个开源的JavaScript引擎，它被用于Chrome浏览器和Node.js等平台。V8的垃圾回收机制是V8对内存管理的实现。

V8的垃圾回收机制采用了分代垃圾回收策略，将内存分为新生代和老生代两个部分。新生代内存用于存储生命周期短的对象，采用Scavenge算法进行垃圾回收。老生代内存用于存储生命周期长的对象，采用Mark-Sweep和Mark-Compact算法进行垃圾回收。

在V8中，垃圾回收器会周期性地检查内存中不再使用的对象，并将其回收以释放内存空间。由于V8采用了分代垃圾回收策略，所以在回收新生代内存时，只需要扫描新生代内存中的少量存活对象即可，因此回收速度较快。而在回收老生代内存时，需要扫描的存活对象较多，因此回收速度较慢。

总的来说，V8的垃圾回收机制通过采用分代垃圾回收策略，提高了内存回收的效率，减少了内存碎片的产生，从而提高了JavaScript应用的性能和稳定性。

# tree- shaking的用途，原理

Tree-shaking 是一种优化 JavaScript 应用程序的技术，可以去除代码中从未被使用过的部分，以减少文件大小和提高性能。

Tree-shaking 的原理是基于 ES2015 模块的静态解析特性。当使用 ES2015 模块语法时，可以在代码中使用 import 和 export 关键字来声明依赖关系。这些依赖关系可以被静态分析，以确定哪些代码被使用，哪些代码未被使用。基于此，可以删除未使用的代码。

在实际应用中，需要使用支持 Tree-shaking 技术的构建工具，如 Webpack、Rollup 等。这些工具可以根据依赖关系构建依赖图，然后使用 Tree-shaking 算法来删除未使用的代码，最终生成精简的代码包。

# 介绍一下lru算法？

LRU（Least Recently Used）算法是一种常见的缓存淘汰策略，也被称为最近最少使用算法。LRU算法的核心思想是，当缓存空间不足时，优先淘汰最近最少使用的缓存数据。

具体来说，LRU算法维护一个缓存的列表，每次访问一个数据时，就将该数据移到列表的最前端，表示该数据最近被访问过。当需要淘汰数据时，就将列表尾部的数据淘汰，因为这些数据是最近最少被访问的。

LRU算法的优点是能够保证缓存中的数据都是热点数据，也就是最近被访问的数据，因此可以有效地提高缓存命中率。其缺点是实现较为复杂，需要维护一个有序的缓存列表，并且每次访问数据都需要更新列表，对性能有一定的影响。

1. 自我介绍

2. dns寻址过程

3. TCP三次握手 四次挥手

4. UDP TCP连接

5. 网络分层五层分别是？

6. 浏览器响应码 304

7. 强缓存和协商缓存

8. etag和last-modify区别

9. Cache-Contral:max-age和Expires的区别，http/1.0用哪个，http/1.1用哪个，为什么要替换这个字段

10. 离线缓存、启发式缓存

11. 浏览器响应码 403

12. 什么是同源策略，如何解决跨域，jsonp的原理是什么，是标签允许跨域还是某个属性允许跨域？

13. 浏览器渲染机制

14. DOM树和CSSOM树是同步渲染的吗？

15. DOM、CSS和JS的相互阻塞关系

16. 下面这段代码会执行几次回流

div.style.width = 10;

div.style.height = 10;

console.log(div.offsetWidth);

console.log(div.offsetHeight);

17. 下面这段代码的输出结果

var name='window'

var obj={

name:'obj',

say(){

console.log(this.name);

},

say1:()=>{

console.log(this.name);

}

}

obj.say()

obj.say1()

var fn=obj.say

var fn1=obj.say1

fn()

fn1()

18. 箭头函数和普通函数还有没有其他区别

19. 代码题：并发控制

// fetch已实现，可直接使用

// 向url发起请求，返回一个Promise

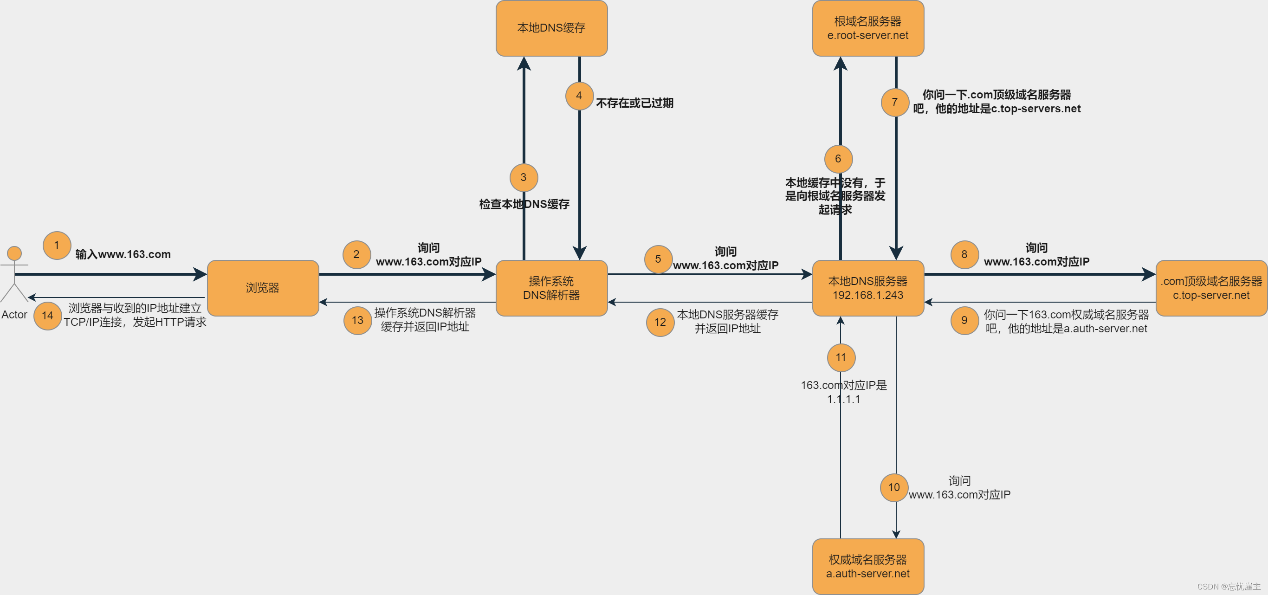
// fetch(url:string)=>Promise

// 请你实现一个函数，控制并发请求数量，接受两个参数

// urls:string[] 待请求的url

// k:number 并发请求数量

2.用户输入域名，浏览器查询本地dns缓存，本地dns服务器查询，根域名服务器查询，顶级域名服务器查询，权威域名服务器返回ip地址



3.三次握手：客户端发起连接请求，服务端接收到客户端发起的syn请求后，向客户响应syn+ack，客户端会返回ack确认，服务端收到第三次握手的ack后连接成功。四次挥手：客户端向服务端发送FIN发起断开连接请求，服务端收到客户端的请求，发生ACK给客户端告诉客户端我收到了你的请求，服务端发送FIN给客户端请求断开连接，客户端发送ACK给服务端，断开连接

4.他们都是传输层的协议，tcp是面向连接的，提供的是可靠的传输，传播速度相对较慢，适合需要高可靠性的场景，比如网页浏览、邮件、文件传输等。udp是面向数据报文的，不能保证传输的可靠性，传播速度比较快，适合视频、实时游戏等。

5.物理层，数据链路层，传输层，网络层，应用层

6.这个是在协商缓存的时候如果资源未改变就返回304

7.都是浏览器的缓存策略，先去看有没有命中强缓存，通过cache-control或者expires来看，如果失效了就去看有没有命中协商缓存，通过etag或者last-modified来看

8.tag是资源标识符，文件改变的时候，etag也会改变，更准确；last-modified是上次文件修改时间，不够准确

9.Expires是1.0，是一个具体的过期时间点（GMT时间），告诉客户端资源何时过期，可能受到客户端和服务器时钟偏差的影响不准确；cache-control是1.1，相对于资源被请求的时间，更准确

10启发式缓存主要用于那些没有明确指定缓存策略的资源。它是浏览器的一种兜底策略,为没有设置缓存策略的资源提供一定的缓存能力。本地存储缓存是指在客户端(如浏览器、移动应用等)本地保存数据的一种机制。它可以提高应用程序的性能和用户体验,减少对服务器的请求,并支持离线访问等功能。本地存储缓存主要包括以下几种技术: **1. Web Storage**包括 localStorage 和 sessionStorage 两种机制:**IndexedDB，Cache API**

11.权限不足

12协议域名端口号必须相同，proxy、jsonp、nginx、cors，原理是利用script标签的src属性来允许跨域请求数据

13解析html构建dom树，css构建cssom树，dom和cssom合并成render树，进行布局、绘制，当页面内容变化时候，发送回流重绘

14不是，css的加载和解析有可能会阻塞dom渲染

15 Css可能会阻塞DOM的渲染；Js会阻塞dom的解析（如果不用async或者defer）；Css可能会阻塞js的执行（如果js依赖css样式）；Js不会阻塞css的加载

16.1次？

17.‘obj'；'window'；'window'；'window'

18. 箭头函数没有自己的this，不能做构造函数，没有arguments对象，但可以使用剩余参数来代替

1. CDN 是如何理解的，有什么好处。他是怎么判断那个节点离他近的，原理

**什么是 CDN？**

**CDN**（Content Delivery Network）是一种分布式服务器网络，旨在通过将内容（如图像、视频、HTML 文件、样式表和脚本）复制并分发到全球各地的服务器节点上，从而提高内容交付的速度和可靠性。当用户访问网站或应用程序时，CDN 会自动选择最合适的节点来提供内容，以减少延迟并提高性能。

**CDN 的好处**

1. **降低延迟**：通过将内容存储在靠近用户的节点上，CDN 可以显著减少延迟，从而提高网站和应用程序的加载速度。
2. **提高可用性**：CDN 节点分布在多个地理位置，即使某个节点发生故障，其他节点仍然可以提供服务，从而提高网站的可用性和可靠性。
3. **减轻服务器负载**：通过将静态内容的请求分发到 CDN 节点，源服务器的负载得以减少，允许它处理更多的动态请求或降低其负载。
4. **带宽优化**：CDN 通过缓存内容来减少源服务器的带宽消耗，同时通过减少重复请求来优化网络带宽。
5. **安全性提升**：一些 CDN 提供了内置的安全功能，如 DDoS 保护、WAF（Web 应用防火墙）等，可以防止恶意攻击。

**CDN 如何判断最近的节点**

CDN 使用多种技术来判断哪个节点最接近用户，主要包括以下几种：

1. **地理位置**：
   * CDN 可以通过用户的 IP 地址来估计其地理位置，并将请求路由到地理上最近的 CDN 节点。
2. **DNS 重定向**：
   * 当用户发出请求时，DNS 服务器会将请求解析到最接近用户的 CDN 节点。这个过程可以基于地理位置、网络延迟等多个因素来决定。
3. **网络拓扑分析**：
   * CDN 服务提供商会实时分析全球网络流量和拥塞情况，从而动态选择最佳路径来传输内容。
4. **Anycast 路由**：
   * CDN 使用 Anycast 技术，在全球范围内使用相同的 IP 地址。当用户请求一个资源时，路由协议会自动将请求路由到网络路径最短的节点。

**原理**

* **请求分配**：当用户请求某个内容时，DNS 系统会将用户引导至距离最近且负载较轻的 CDN 节点。
* **缓存机制**：CDN 节点会将请求的内容缓存下来，下次有其他用户请求相同内容时，会直接从缓存中提供，而无需再次访问源服务器。
* **负载均衡**：通过分配流量到多个节点，CDN 能够确保网络流量在整个系统中均匀分布，从而避免某个节点过载。

1. WebPack 是如何打包的，前端模块化有什么好处

**Webpack** 是一个现代 JavaScript 应用程序的模块打包工具。它主要用于将多个模块（JavaScript、CSS、图片等）打包成一个或多个浏览器可以直接加载的文件（bundle）。Webpack 的打包过程可以分为以下几个步骤：

1. **入口（Entry）**：
   * Webpack 从配置文件中指定的入口点开始构建内部依赖图。入口点通常是应用程序的主文件，比如 index.js 或 main.js。

javascript

复制代码

module.exports = {

entry: './src/index.js',

// 其他配置...

};

1. **解析依赖（Dependency Graph）**：
   * Webpack 递归地解析入口文件的依赖项，识别出所有依赖的模块（包括子模块和它们的依赖），构建一个依赖图（dependency graph）。
2. **模块加载与转换（Loaders）**：
   * Webpack 支持各种类型的模块（JS、CSS、图片、字体等）。它通过 loaders 来转换这些模块，例如将 ES6 转换为 ES5，或将 SCSS 转换为 CSS。

javascript

复制代码

module.exports = {

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: 'babel-loader',

},

{

test: /\.css$/,

use: ['style-loader', 'css-loader'],

},

],

},

};

1. **插件（Plugins）**：
   * Webpack 提供了插件系统，允许在打包的各个阶段执行各种操作，如代码压缩、文件拆分、模板生成等。插件使得 Webpack 更加灵活和强大。

javascript

复制代码

const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {

plugins: [

new HtmlWebpackPlugin({

template: './src/index.html',

}),

],

};

1. **输出（Output）**：
   * 最后，Webpack 将处理好的模块打包成一个或多个文件，并输出到指定的目录。常见的输出配置包括文件名和路径。

javascript

复制代码

module.exports = {

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist'),

},

};

1. **代码拆分与优化**：
   * Webpack 通过代码拆分（code splitting）技术将应用程序拆分成多个 bundle，以提高加载性能。它还可以对代码进行压缩、去重和优化。

**前端模块化的好处**

前端模块化是一种将代码拆分成独立的、可复用的模块的技术，每个模块都封装了一个独立的功能或逻辑。模块化的主要好处包括：

1. **代码复用**：
   * 模块化使得开发者可以将常用的代码封装成模块，以便在多个项目或应用程序中复用，提高开发效率。
2. **维护性高**：
   * 模块化代码更容易维护。由于每个模块只负责单一功能，开发者可以更容易地理解、调试和修改代码。
3. **分工合作**：
   * 在大型项目中，模块化可以使得不同的开发者独立负责不同的模块，从而减少代码冲突，提升协作效率。
4. **依赖管理**：
   * 模块化系统（如 ES Modules、CommonJS）允许开发者明确地声明模块之间的依赖关系，减少了全局变量的使用，从而避免命名冲突和依赖问题。
5. **按需加载**：
   * 通过模块化，应用程序可以按需加载特定的模块，从而减少初始加载时间，提升用户体验。
6. **增强的测试能力**：
   * 模块化代码可以更容易地编写单元测试，因为每个模块独立且具有明确的输入输出。
7. **工具链支持**：
   * 现代前端工具（如 Webpack、Rollup、Parcel）高度依赖模块化系统，以实现自动化的打包、优化和部署工作流。
8. 如何减少网络请求，为什么要减少网络请求

**为什么要减少网络请求？**

1. **提高加载速度**：
   * 每个网络请求都有一定的开销，包括 DNS 查询、TCP 连接、TLS 握手、请求和响应时间等。减少请求数量可以降低这些开销，从而提升页面加载速度。
2. **减少带宽消耗**：
   * 每个请求都消耗一定的带宽。如果减少请求数量，可以减少总的带宽消耗，特别是对于移动网络用户来说，这一点尤为重要。
3. **降低服务器负担**：
   * 减少请求数量可以减轻服务器的负担，减少服务器处理的请求数量，从而提高整体系统的可扩展性和响应速度。
4. **优化用户体验**：
   * 页面加载速度是用户体验的关键因素。减少请求可以减少页面的总加载时间，提升用户的满意度和留存率。
5. **提升页面渲染性能**：
   * 浏览器在加载和渲染页面时会并行处理多个请求，减少请求可以减少页面渲染过程中的阻塞，提高渲染效率。

**如何减少网络请求？**

**合并文件**：

* + **CSS 和 JavaScript 合并**：将多个 CSS 文件和 JavaScript 文件合并成一个文件，减少请求数量。
  + **图片合并**：使用 CSS Sprite 技术将多个小图标合并为一个大图像，减少请求

**使用缓存**：

* **HTTP 缓存头**：通过配置适当的缓存头（如 Cache-Control 和 Expires）来利用浏览器缓存，减少对服务器的重复请求。
* **版本控制**：通过给资源添加版本号或哈希值，确保文件更新时浏览器能正确加载最新版本。

**懒加载**：

* **图片懒加载**：只在用户滚动到图片位置时才加载图片，减少初始页面加载时的请求。
* **懒加载 JavaScript**：按需加载 JavaScript 文件，避免一次性加载所有脚本

**压缩和优化资源**：

* **文件压缩**：使用压缩工具（如 UglifyJS、CSSNano）来减小 JavaScript 和 CSS 文件的大小，减少请求的大小。
* **图片优化**：使用适当的图片格式和压缩工具来减少图片文件的大小。

**使用数据 URL**：

* 对于小的图像或图标，可以将它们嵌入到 HTML 或 CSS 文件中作为 Data URL，这样可以减少额外的请求。

**异步加载资源**：

* 使用 async 或 defer 属性来异步加载 JavaScript 文件，避免阻塞页面渲染。

**使用 CDN**：

* 将静态资源（如库、框架、图片）托管到 Content Delivery Network (CDN)，利用 CDN 的缓存和分发机制减少请求的延迟和带宽消耗。

1. 当要频繁改变一个元素的状态时，应该怎么设置可以减少性能损耗

**1. 使用请求动画帧（requestAnimationFrame）**

requestAnimationFrame 是一个高效的方式来处理动画或频繁更新的 DOM 操作。它可以确保你的代码在浏览器的下一次重绘之前执行，从而提高性能并减少视觉上的撕裂。

**2. 批量更新**

如果可能，尽量将多个 DOM 更新操作合并为一个批量操作。这可以减少浏览器的重绘和重排次数。

**3. 虚拟 DOM**

使用框架或库（如 React 或 Vue）中的虚拟 DOM 来优化频繁的 DOM 更新。这些框架会将实际的 DOM 更新操作批量处理，从而减少性能损耗。

**4. 避免直接操作 DOM**

频繁的直接 DOM 操作可能导致性能问题。尽量将状态变化集中在 JavaScript 逻辑中，然后使用 innerHTML 或其他批量操作方法来更新 DOM。

**5. 应用防抖技术来减少频繁事件处理。**

**怎么判断一个元素出现在当前视窗**

**1. 使用 getBoundingClientRect**

getBoundingClientRect() 方法返回一个 DOMRect 对象，表示元素的大小及其相对于视口的位置。通过这个方法可以检查元素的边界是否与视口相交，从而判断是否在当前视窗中。

**function isElementInViewport(el) {**

**const rect = el.getBoundingClientRect();**

**return (**

**rect.top >= 0 &&**

**rect.left >= 0 &&**

**rect.bottom <= (window.innerHeight || document.documentElement.clientHeight) &&**

**rect.right <= (window.innerWidth || document.documentElement.clientWidth)**

**);**

**}**

**// 示例用法**

**const element = document.querySelector('#myElement');**

**console.log(isElementInViewport(element));**

**2. 使用 IntersectionObserver**

IntersectionObserver 是一种更现代的方法，它允许你异步地观察一个元素的可见性状态，通常用于实现懒加载和无限滚动等功能。

**const observer = new IntersectionObserver((entries) => {**

**entries.forEach(entry => {**

**if (entry.isIntersecting) {**

**console.log('Element is in the viewport');**

**} else {**

**console.log('Element is not in the viewport');**

**}**

**});**

**});**

**const element = document.querySelector('#myElement');**

**observer.observe(element);**

**http1.1 http2 http3 https都有什么特点**

**1. HTTP/1.1**

**发布年份：** 1997

**主要特点：**

* **持久连接（Persistent Connection）**：默认开启持久连接，可以在同一个TCP连接上复用多个HTTP请求，减少了TCP连接的开销。
* **管道化（Pipelining）**：允许在发送前一个请求的响应之前发送多个请求。但由于服务器响应必须按请求顺序返回，且存在阻塞问题，实际使用较少。
* **分块传输编码（Chunked Transfer Encoding）**：允许服务器在不知道内容长度的情况下分块传输数据，有利于流式数据传输。
* **缓存控制（Cache Control）**：引入了更细粒度的缓存控制头部字段，如 Cache-Control，便于客户端和服务器对资源进行缓存管理。
* **虚拟主机（Virtual Hosting）**：允许在同一个IP地址上托管多个域名，使用 Host 头部来区分请求是针对哪个域名的。

**局限性：**

* **队头阻塞（Head-of-Line Blocking）**：由于管道化和持久连接，后续请求必须等待前一个请求的响应返回，这可能导致延迟。
* **单一请求-响应模式**：每个请求只能对应一个响应，缺乏多路复用能力。

**2. HTTP/2**

\*\*发布年份：2015

**主要特点：**

* **二进制分帧（Binary Framing）**：将数据分为更小的帧，并以二进制格式传输，增加了传输效率和解析速度。
* **多路复用（Multiplexing）**：允许在同一个TCP连接上并发发送多个请求和响应，避免了HTTP/1.1的队头阻塞问题。
* **首部压缩（Header Compression）**：使用HPACK压缩算法，减少HTTP头部的冗余数据，减小带宽占用。
* **服务器推送（Server Push）**：允许服务器在客户端请求之前主动推送资源，减少请求延迟。
* **流量控制（Flow Control）**：支持对流量进行精细化控制，避免网络拥塞。

**改进点：**

* **提高了传输效率**：多路复用和二进制分帧机制大大提高了传输效率，特别是在高延迟网络环境下。
* **减少了延迟**：首部压缩和服务器推送有效减少了请求的延迟。

**3. HTTP/3**

\*\*发布年份：2018（草案）

**主要特点：**

* **基于QUIC协议**：HTTP/3不再使用TCP，而是基于UDP之上的QUIC协议，从而彻底解决了TCP协议中的队头阻塞问题。
* **更快的连接建立**：QUIC通过将握手和加密过程合并，极大地减少了连接建立的时间，特别是在高延迟环境下表现更优。
* **内置加密**：QUIC协议自带加密机制，简化了协议栈的设计，提高了安全性。
* **更好的丢包处理**：QUIC对数据包的丢失有更好的处理机制，确保数据传输的可靠性。

**优势：**

* **进一步降低了延迟**：通过减少握手时间和消除队头阻塞，HTTP/3显著降低了网络延迟。
* **更适合移动网络**：HTTP/3在不可靠网络环境下表现出色，特别适合移动网络和Wi-Fi切换等场景。

**4. HTTPS**

**特点：**

* **加密传输**：HTTPS使用SSL/TLS协议对数据进行加密，确保传输过程中的数据安全，防止窃听和篡改。
* **身份验证**：通过数字证书验证服务器的身份，确保客户端与合法的服务器通信。
* **数据完整性**：HTTPS提供了数据完整性校验，防止数据在传输过程中被篡改。

**工作原理：**

* **SSL/TLS握手过程**：客户端与服务器在通信前先进行加密算法协商、密钥交换等一系列握手操作，建立安全的通信通道。
* **加密通信**：一旦安全通道建立，所有HTTP数据都将通过SSL/TLS进行加密传输。

**缺点：**

* **性能开销**：由于加密和解密的过程，HTTPS在性能上比HTTP有一定的开销。
* **证书成本**：HTTPS需要购买数字证书，虽然现在可以通过Let’s Encrypt等机构免费获取证书，但还是增加了管理复杂性。

**总结**

* **HTTP/1.1** 是经典的、广泛支持的协议，但存在队头阻塞等问题。
* **HTTP/2** 引入了多路复用和二进制分帧，显著提高了传输效率和并发能力。
* **HTTP/3** 基于QUIC协议，进一步降低了延迟，优化了网络传输，特别是在不可靠的网络环境下。
* **HTTPS** 通过加密和身份验证增强了HTTP的安全性，已经成为Web通信的标准。

**浏览器中的事件循环原理和nodejs中的有什么区别？微任务有那些，宏任务又有那些？**

**1. 事件循环原理**

**浏览器中的事件循环**

浏览器的事件循环模型遵循 HTML 规范，其主要结构如下：

* **宏任务队列**：存储由 setTimeout、setInterval、I/O 事件、UI 渲染等产生的任务。每次循环会从宏任务队列中取出一个任务执行。
* **微任务队列**：存储由 Promise 的 .then、MutationObserver、queueMicrotask 产生的任务。微任务队列中的任务会在当前宏任务执行完成后立即执行，且会一次性清空微任务队列中的所有任务。

**事件循环过程**：

1. 从宏任务队列中取出一个任务执行。
2. 检查并执行微任务队列中的所有任务。
3. 执行渲染（如果有的话）。
4. 继续下一次循环。

**Node.js 中的事件循环**

Node.js 的事件循环基于 [libuv](https://github.com/libuv/libuv)，不同于浏览器，Node.js 的事件循环分为多个阶段，每个阶段都有特定的任务类型：

1. **timers**：执行到期的 setTimeout 和 setInterval 回调。
2. **pending callbacks**：执行延迟到下一个循环迭代的 I/O 回调。
3. **idle, prepare**：仅内部使用。
4. **poll**：检索新的 I/O 事件；执行与 I/O 相关的回调（几乎所有的异步操作都在此阶段处理）。
5. **check**：执行 setImmediate 回调。
6. **close callbacks**：执行 close 事件的回调，如 socket.on('close')。

**微任务队列**（nextTickQueue 和 microtaskQueue）：

* Node.js 事件循环中，微任务同样在每个阶段结束后执行，但特别之处在于 process.nextTick 的任务优先于其他微任务执行。

**事件循环过程**：

1. 执行 timers 阶段任务。
2. 执行所有微任务。
3. 执行 I/O 任务。
4. 执行所有微任务。
5. 执行 check 阶段的 setImmediate 任务。
6. 执行所有微任务。
7. 重复以上步骤。

**2. 微任务和宏任务的分类**

**微任务**

* **浏览器和 Node.js 都支持**：
  + Promise 的 .then() / .catch() / .finally()
  + MutationObserver
  + queueMicrotask
* **Node.js 特有**：
  + process.nextTick

**宏任务**

* **浏览器和 Node.js 都支持**：
  + setTimeout
  + setInterval
  + I/O 回调（在不同阶段处理，浏览器和 Node.js 都有）
* **浏览器特有**：
  + requestAnimationFrame
  + DOM 事件处理
* **Node.js 特有**：
  + setImmediate
  + 各种阶段处理（timers, poll, check等）

**3. 主要区别**

* **事件循环的阶段**：Node.js 的事件循环分为多个阶段，每个阶段负责不同类型的任务；而浏览器事件循环更简化，主要分为宏任务和微任务。
* **微任务的优先级**：在 Node.js 中，process.nextTick 比 Promise 的 .then() 等其他微任务优先级更高。而在浏览器中，所有微任务的优先级是一样的。
* **渲染机制**：浏览器中有一个独特的 UI 渲染步骤，事件循环在处理完微任务后可能会进行页面的重绘；而 Node.js 是服务器端环境，不涉及 UI 渲染。

**总结**

* **浏览器**：宏任务与微任务并行处理，事件循环关注UI渲染和用户交互。
* **Node.js**：事件循环更复杂，分多个阶段处理不同类型的任务，适用于服务器端高并发的I/O操作。

**大文件上传怎么做的，具体流程。从前端和后端角度**

**前端实现大文件上传**

1. **文件选择**：
   * 用户选择一个文件进行上传，通过 <input type="file"> 或 Drag & Drop。
2. **文件分片**：
   * 将大文件分成多个小块（例如 1MB 每块），使用 Blob.slice 方法。
   * 例如：

javascript

复制代码

const file = document.getElementById('fileInput').files[0];

const chunkSize = 1024 \* 1024; // 每块1MB

const totalChunks = Math.ceil(file.size / chunkSize);

for (let i = 0; i < totalChunks; i++) {

const start = i \* chunkSize;

const end = Math.min(start + chunkSize, file.size);

const chunk = file.slice(start, end);

// 发送这个chunk到服务器

uploadChunk(chunk, i, totalChunks);

}

1. **上传分片**：
   * 使用 fetch 或 XMLHttpRequest 将分片发送到服务器。
   * 可以发送每个分片的索引、总分片数、文件标识符等信息，以便服务器正确重组。
   * 例如：

javascript

复制代码

async function uploadChunk(chunk, index, totalChunks) {

const formData = new FormData();

formData.append('chunk', chunk);

formData.append('index', index);

formData.append('totalChunks', totalChunks);

await fetch('/upload', {

method: 'POST',

body: formData

});

}

1. **断点续传**：
   * 在上传之前，前端可以请求服务器检查已上传的分片，跳过已上传的部分。
   * 这种机制通常通过发送文件标识符和索引来实现。
2. **合并请求**：
   * 当所有分片上传完成后，前端发送一个“合并请求”给服务器，通知服务器将分片合并成完整的文件。

### ****后端处理大文件上传****

1. **接收分片**：
   * 后端接收到文件分片，使用 multipart/form-data 处理每个分片。
   * 根据分片的索引和文件标识符，将每个分片保存到临时存储（例如磁盘）。

**2.处理合并请求**：

* 后端接收到合并请求后，将所有分片按照索引顺序进行合并。
* 合并后删除临时存储的分片文件，保留完整文件。

**3.支持断点续传**：

* 后端应支持客户端请求已上传的分片信息，返回已上传的索引列表。
* 客户端根据响应决定跳过已上传的部分。

断点续传是一种允许在文件上传中断后，从中断的地方继续上传的技术。它的主要目的是提高上传的可靠性，避免因网络问题或其他原因导致整个文件上传失败的情况。下面详细解释断点续传的操作流程，包括前端和后端的实现步骤。

### ****断点续传的操作流程****

#### ****1. 前端实现****

1. **文件分片**：
   * 依然是将文件分成多个较小的块（分片）来上传。
   * 每个分片都有其索引，用来标识其在整个文件中的位置。

javascript

复制代码

const file = document.getElementById('fileInput').files[0];

const chunkSize = 1024 \* 1024; // 每块1MB

const totalChunks = Math.ceil(file.size / chunkSize);

1. **上传前检查已上传的分片**：
   * 在开始上传之前，前端会请求服务器，询问哪些分片已经成功上传。
   * 服务器返回已上传的分片索引列表。

javascript

复制代码

async function checkUploadedChunks(fileId) {

const response = await fetch('/check', {

method: 'POST',

body: JSON.stringify({ fileId }),

headers: { 'Content-Type': 'application/json' }

});

return await response.json(); // 返回已上传的分片索引

}

1. **跳过已上传的分片，继续上传剩余的分片**：
   * 根据服务器返回的已上传分片列表，跳过这些分片，只上传剩下的分片。

javascript

复制代码

async function uploadChunks(file, uploadedChunks, fileId) {

for (let i = 0; i < totalChunks; i++) {

if (!uploadedChunks.includes(i)) {

const start = i \* chunkSize;

const end = Math.min(start + chunkSize, file.size);

const chunk = file.slice(start, end);

await uploadChunk(chunk, i, totalChunks, fileId);

}

}

}

1. **上传分片**：
   * 每个分片单独上传到服务器，服务器端根据索引将分片存储起来。

javascript

复制代码

async function uploadChunk(chunk, index, totalChunks, fileId) {

const formData = new FormData();

formData.append('chunk', chunk);

formData.append('index', index);

formData.append('totalChunks', totalChunks);

formData.append('fileId', fileId);

await fetch('/upload', {

method: 'POST',

body: formData

});

}

1. **合并请求**：
   * 当所有分片都上传完毕后，前端会发送一个请求给服务器，通知服务器合并这些分片。

javascript

复制代码

async function mergeChunks(fileId, totalChunks) {

await fetch('/merge', {

method: 'POST',

body: JSON.stringify({ fileId, totalChunks }),

headers: { 'Content-Type': 'application/json' }

});

}

大文件上传怎么监控失败的情况呢？

**1. 使用 XMLHttpRequest 的事件监听**

* **适用场景**: 可以使用 XMLHttpRequest 来实现大文件上传，并通过监听它的事件来监控失败情况。
* **实现方法**:

javascript

复制代码

const xhr = new XMLHttpRequest();

xhr.upload.addEventListener('progress', (event) => {

if (event.lengthComputable) {

const percentComplete = (event.loaded / event.total) \* 100;

console.log(`Upload progress: ${percentComplete}%`);

}

});

xhr.addEventListener('load', () => {

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

console.log('Upload completed successfully');

} else {

console.error('Upload failed', xhr.status, xhr.statusText);

}

});

xhr.addEventListener('error', () => {

console.error('Upload failed due to a network error');

});

xhr.addEventListener('abort', () => {

console.warn('Upload was aborted');

});

xhr.open('POST', '/upload');

xhr.send(file); // Assuming `file` is a File object

* **优点**: 可以捕获网络错误、服务器错误、上传进度等详细信息。

**2. 使用 fetch 与 AbortController**

* **适用场景**: 使用 fetch API 来上传文件，并通过 AbortController 来监控和中止请求。
* **实现方法**:

javascript

复制代码

const controller = new AbortController();

const signal = controller.signal;

fetch('/upload', {

method: 'POST',

body: file, // Assuming `file` is a File object

signal: signal

})

.then(response => {

if (!response.ok) {

throw new Error('Upload failed: ' + response.statusText);

}

return response.json();

})

.then(data => {

console.log('Upload success:', data);

})

.catch(error => {

if (error.name === 'AbortError') {

console.warn('Upload was aborted');

} else {

console.error('Upload failed:', error);

}

});

// To abort the request if needed

// controller.abort();

* **优点**: 更现代的接口，更容易与 Promise 和 async/await 结合使用。

**3. 基于 FormData 的分片上传和重试机制**

* **适用场景**: 当上传非常大的文件时，通常会将其分片并分别上传，每一片上传成功后才上传下一片。
* **实现方法**:

javascript

复制代码

async function uploadFile(file) {

const chunkSize = 1024 \* 1024; // 1MB per chunk

const totalChunks = Math.ceil(file.size / chunkSize);

for (let i = 0; i < totalChunks; i++) {

const start = i \* chunkSize;

const end = Math.min(start + chunkSize, file.size);

const chunk = file.slice(start, end);

const formData = new FormData();

formData.append('chunk', chunk);

formData.append('chunkIndex', i);

formData.append('totalChunks', totalChunks);

try {

const response = await fetch('/upload-chunk', {

method: 'POST',

body: formData

});

if (!response.ok) {

throw new Error('Failed to upload chunk ' + i);

}

} catch (error) {

console.error('Chunk upload failed:', error);

// Optional: Implement retry logic here

i--; // Retry the current chunk

}

}

console.log('All chunks uploaded successfully');

}

uploadFile(file);

* **优点**: 如果某个分片上传失败，可以重试或暂停后继续，减少整体上传的风险。

**4. 服务器端返回状态监控**

* **适用场景**: 当文件上传完成后，服务器可以返回详细的状态信息，客户端可以根据状态信息来处理后续逻辑。
* **实现方法**:
  + 在服务器端实现逻辑，确保所有文件块都成功上传。
  + 返回上传结果，客户端根据响应中的状态信息判断成功或失败。
  + 客户端根据失败的状态信息做进一步的处理，如重试或用户提示。

**5. WebSocket 或其他实时通信**

* **适用场景**: 在需要实时监控上传过程，或者在长时间上传过程中获取实时反馈时。
* **实现方法**:
  + 使用 WebSocket 或 Server-Sent Events (SSE) 来接收服务器的实时反馈，监控上传过程中的失败情况。

跨端是什么，跨端的原理是什么

**跨端**指的是应用程序可以在多个平台或设备上运行，而无需为每个平台编写独立的代码。具体来说，跨端开发旨在使用一套代码，同时在不同的操作系统（如 iOS、Android、Windows、macOS 等）或设备类型（如手机、平板、PC、智能手表等）上运行应用程序。

**跨端开发的原理**

跨端开发通常通过以下几种方式实现：

1. **Web 技术封装**：
   * **原理**: 使用 HTML、CSS 和 JavaScript 等 Web 技术开发应用，然后通过封装工具将 Web 应用转化为可在各平台运行的应用。
   * **工具**:
     + **Cordova / PhoneGap**: 将 Web 应用打包成原生应用，可以访问设备的原生功能（如摄像头、加速度计等）。
     + **Electron**: 用于创建桌面跨平台应用，常用于将 Web 应用打包成 Windows、macOS 和 Linux 下的应用程序。
   * **优点**: 可以使用现有的 Web 技术，快速开发和迭代。
   * **缺点**: 性能上可能不如原生应用，特别是在需要高性能计算或复杂图形渲染的场景。
2. **跨平台框架**：
   * **原理**: 使用跨平台的开发框架编写应用代码，然后由框架将代码编译或转换为各平台的原生代码或可执行程序。
   * **工具**:
     + **React Native**: 由 Facebook 开发，使用 JavaScript 和 React 编写跨平台移动应用，最终生成原生代码。
     + **Flutter**: 由 Google 开发，使用 Dart 语言编写跨平台应用，能生成高性能的原生应用。
   * **优点**: 提供接近原生的性能和用户体验，且可以共享大部分代码。
   * **缺点**: 需要学习和使用特定框架的生态系统，部分原生功能可能需要额外的桥接或插件支持。
3. **PWA (Progressive Web Apps)**：
   * **原理**: PWA 是一种增强型的 Web 应用，它在浏览器中运行，但可以像原生应用一样离线工作、推送通知，并且可以安装到设备桌面。
   * **工具**: 通过标准的 Web 技术（如 Service Workers、Web App Manifest 等）构建，不依赖于特定的工具。
   * **优点**: 不需要通过应用商店分发，开发简单，具有跨平台特性。
   * **缺点**: 对于需要深度访问设备功能的应用，PWA 可能受限，且用户体验在某些平台上不如原生应用。
4. **编译层面的跨平台支持**：
   * **原理**: 使用特定语言和工具链编写代码，然后通过编译器将代码编译为适应不同平台的原生代码。
   * **工具**:
     + **Xamarin**: 使用 C# 语言开发，能够将代码编译成各平台的原生应用。
     + **Kotlin Multiplatform**: 使用 Kotlin 语言编写共享代码，并分别编译成适应不同平台的代码。
   * **优点**: 更好的原生性能和平台集成。
   * **缺点**: 编译和调试的复杂度较高，且需要对目标平台的特性有深入了解。

什么是ssr和ssg，ssr和ssg有什么不一样的地方

**SSR (Server-Side Rendering)** 和 **SSG (Static Site Generation)** 是两种常见的页面渲染方式，它们在数据获取、页面生成的时机和方式上存在显著的不同。

**SSR（Server-Side Rendering）**

**定义**：

* SSR 是指在服务器端生成完整的 HTML 页面，并将其发送给客户端浏览器进行显示。当用户请求一个页面时，服务器会获取数据，生成 HTML 内容，并将完整的页面发送给客户端。

**工作流程**：

1. 用户请求访问某个页面。
2. 服务器接收到请求，获取页面所需的数据。
3. 服务器将数据与模板结合，生成完整的 HTML 页面。
4. 服务器将生成的 HTML 页面发送给客户端。
5. 客户端浏览器接收并渲染页面。

**优点**：

* **SEO 友好**：搜索引擎能够直接抓取页面内容，有利于 SEO。
* **快速首屏渲染**：因为服务器直接返回完整的 HTML 页面，用户能更快地看到页面内容。
* **动态性强**：可以根据用户请求的不同生成不同的内容。

**缺点**：

* **服务器压力大**：每次请求都需要服务器生成页面，处理负载较高。
* **可能导致较高的延迟**：如果服务器响应速度慢，会影响用户体验。

**SSG（Static Site Generation）**

**定义**：

* SSG 是在构建阶段生成静态 HTML 页面，并将这些页面部署到服务器上。当用户请求页面时，服务器直接返回已经生成的静态 HTML 文件。

**工作流程**：

1. 在构建阶段，生成器（如 Next.js、Gatsby）获取所有必要的数据。
2. 生成器将数据与模板结合，生成静态的 HTML 页面。
3. 这些静态 HTML 页面被部署到服务器或 CDN。
4. 用户请求页面时，服务器或 CDN 直接返回已经生成的静态 HTML 页面。

**优点**：

* **高性能**：静态页面可以通过 CDN 高效分发，响应速度极快。
* **服务器负载轻**：不需要在每次请求时动态生成页面，降低了服务器的负担。
* **安全性高**：因为页面是静态的，减少了许多潜在的安全问题。

**缺点**：

* **缺乏动态性**：页面内容在构建时确定，动态更新的内容需要通过客户端渲染或重新构建页面。
* **生成时间较长**：如果网站规模大，构建阶段可能耗时较长。

**SSR 和 SSG 的区别**

* **渲染时机**：
  + **SSR**：页面在每次请求时动态生成。
  + **SSG**：页面在构建时静态生成，之后的每次请求都直接返回静态页面。
* **性能**：
  + **SSR**：每次请求都会触发服务器生成页面，性能取决于服务器的负载能力。
  + **SSG**：静态页面可以通过 CDN 缓存和分发，性能非常高。
* **动态性**：
  + **SSR**：适合需要频繁更新内容的网站，因为每次请求都会生成最新的页面。
  + **SSG**：适合内容较稳定的网站，动态更新需要特殊处理或使用客户端渲染。
* **SEO**：
  + **SSR 和 SSG**：两者都对 SEO 友好，因为最终用户和搜索引擎爬虫都能看到完整的 HTML 页面。

**总结**

* **SSR** 适合动态内容较多、需要实时更新的应用，比如社交媒体、新闻网站。
* **SSG** 适合内容更新频率较低、希望极致性能优化的网站，比如博客、文档站点。

根据项目的实际需求，可以选择 SSR 或 SSG，或者在一些现代框架中混合使用这两种技术（如 Next.js 支持同时使用 SSR 和 SSG）。

React Fiber 是 React 底层的重新实现，旨在提高其在复杂应用中的性能和灵活性。React Fiber 引入了一种新的协调（Reconciliation）算法，用于高效地更新用户界面。以下是关于 React Fiber 的详细解释：

**1. 背景和动机**

在 React Fiber 之前，React 使用的协调算法无法很好地处理非常复杂的应用，尤其是在涉及大量 UI 更新和用户交互的情况下。传统的 React 协调算法是同步的，一旦开始更新就不能被打断，导致在某些情况下用户界面会卡顿。

React Fiber 的目标是解决这些性能问题，使得 React 在处理复杂的更新时更流畅。

**2. Fiber 架构**

Fiber 是 React 用来描述应用程序 UI 树的内部数据结构。每个 Fiber 对象表示 UI 树中的一个节点。Fiber 的核心创新在于它将同步的递归渲染过程拆分为可中断的单元，使得 React 能够在更新过程中暂停、继续或丢弃工作。这种方法允许 React 更加流畅地处理大规模更新。

**3. 时间分片（Time Slicing）**

React Fiber 通过时间分片技术（Time Slicing）来分配 CPU 资源。它将渲染工作分解成多个小的任务，并在这些任务之间插入可以暂停的检查点，这样 React 可以更智能地决定何时暂停工作以响应用户交互。

* **优先级调度**：Fiber 可以为不同的更新任务分配不同的优先级，优先处理用户交互等高优先级任务，而延迟处理低优先级任务。
* **可中断的渲染**：传统的 React 渲染是同步且不可中断的，而 Fiber 允许渲染过程在合适的时机暂停并继续，以避免用户界面卡顿。

**4. 双缓存机制（Double Buffering）**

React Fiber 采用了一种双缓存机制来管理 UI 的更新状态。它维护两个 Fiber 树：

* **current fiber**：表示当前正在显示的 UI。
* **work in progress fiber**：表示正在构建的新的 UI 树。

在更新完成后，React 将工作中的 Fiber 树替换为当前的 Fiber 树，从而完成 UI 的更新。

**5. React Fiber 带来的好处**

* **更流畅的用户体验**：通过时间分片和可中断的渲染，React Fiber 减少了因大规模 UI 更新带来的卡顿现象。
* **更精细的控制**：开发者可以利用 Fiber 提供的生命周期方法（如 componentDidCatch 和 getDerivedStateFromProps）更好地控制组件的更新行为。
* **并发模式**：Fiber 是为 React 未来的并发模式奠定基础的一部分，支持并行处理多个任务，提高应用的响应速度和用户体验。

**6. 应用和影响**

React Fiber 自 React 16 起成为 React 的默认渲染引擎，开发者在使用 React 时可以自动享受到 Fiber 带来的性能优化，而无需做出额外的更改。

**总结**

React Fiber 是 React 框架的一次底层重构，旨在提升 React 应用的性能和响应速度。通过将渲染过程分解为可中断的任务，React Fiber 能够更加流畅地处理复杂的 UI 更新，是 React 面向未来发展的一次重要升级。