import和require区别

1. 模块系统：
   - `import` 是 ES6（ECMAScript 2015）引入的一种模块化机制，使用的是 ES6 模块（也称为 ECMAScript 模块，简称 ESM）
   - `require` 是 CommonJS 模块系统的一部分，广泛用于 Node.js 环境
2. 语法：
   - `import` 语句是静态的，必须放在模块的顶层作用域，不能在条件语句、函数内部或动态导入中使用
   - `require` 是动态的，可以在代码的任何地方使用，包括条件语句、函数内部等
3. 加载：
   - `import` 语句会在编译时解析模块路径，可以在需要时异步加载模块，支持 Tree Shaking（按需加载）
   - `require` 语句是同步的，模块会立即加载和解析
4. 作用域提升：
   - `import` 语句有作用域提升（hoisting），它会在模块顶部被提升。这意味着你可以在模块的任何地方使用它们，而不需要担心它们是否已经被解析
   - `require` 语句没有作用域提升，它们按照代码的执行顺序解析。这意味着你需要在代码中显式地确保模块在使用前已经被加载
5. 兼容性：
   - `import` 语句在现代 JavaScript 环境中得到广泛支持，但在使用 CommonJS 的 Node.js 环境中，需要在 Node.js 14 及以上版本，并在 package.json 中设置 `"type": "module"` 才能使用
   - `require` 语句在 Node.js 环境中得到广泛支持，但在浏览器环境中，通常需要借助工具（如 Browserify 或 Webpack）来打包使用

Tree Shaking的实现

- 树摇（Tree Shaking）是一种优化技术，用于消除代码中未使用的部分。它通过静态分析源代码和模块依赖关系，然后删除未被引用的代码。通常与 ES6 模块和工具（如 Webpack）一起使用。

SPA应用的优缺点

- 优点：
  - 更快的页面切换，因为页面刷新不需要重新加载整个页面。
  - 更流畅的用户体验，因为页面在客户端动态加载数据。
  - 可以更好地利用浏览器缓存，减少服务器负载。
- 缺点：
  - 初次加载可能较慢，因为需要下载整个应用的代码和资源。
  - 不利于 SEO，因为搜索引擎通常只会抓取页面的静态内容。
  - 前进/后退按钮可能不太友好，需要手动处理路由状态。

前端路由原理

- 前端路由是指在单页应用中，通过 JavaScript 控制页面的 URL，而不是传统的服务器端路由。它的原理是监听浏览器的 URL 变化（通常是通过 `hashchange` 事件或 HTML5 的 `history` API），然后根据 URL 的变化加载相应的组件或内容，从而实现页面的切换和导航。

应用首次加载优化方法：

- 代码分割（Code Splitting）：将应用拆分为多个小模块，并按需加载。
- 懒加载（Lazy Loading）：延迟加载页面中不是首次必需的资源和组件。
- 使用 CDN 加速静态资源加载。
- 图片优化：压缩图片、使用适当的图片格式等。
- 使用服务端渲染（SSR）：将页面的初始 HTML 内容直接由服务器生成并发送，加速首次渲染
- performance监控

vue文件怎么编程浏览器可执行的js文件

Vue.js 是一个基于 JavaScript 的前端框架，它本身不需要编译成 JavaScript，因为 Vue.js 本身就是 JavaScript。但是，在实际开发中，我们通常会使用 Vue 的单文件组件（.vue 文件），这些文件需要通过构建工具进行编译打包，最终输出为浏览器可执行的 JavaScript。

在使用 Vue.js 开发项目时，通常的编译流程如下：

1. **编写 Vue 单文件组件：** 开发者编写 `.vue` 文件，其中包括 `<template>`、`<script>` 和 `<style>` 三个部分，分别用来定义模板、逻辑和样式。
2. **使用构建工具进行编译：** 使用像 Webpack、Rollup 或 Vue CLI 这样的构建工具，配置相应的 loader 或插件来处理 `.vue` 文件。例如，可以使用 vue-loader 来处理单文件组件。
3. **构建项目：** 运行构建命令，构建工具会将 `.vue` 文件编译成浏览器可执行的 JavaScript 代码。这包括将模板编译成 render 函数、合并组件代码、处理样式等。
4. **打包输出：** 构建工具会将编译后的 JavaScript 文件打包输出到指定的目录中，用于部署到生产环境或开发服务器上

setState() 方法是异步的

主要是为了提高性能和优化用户体验

1. **性能优化：** 当多次连续调用 `setState()` 时，React 会将这些更新请求进行合并，并在适当的时机进行批处理。这样可以减少不必要的重复渲染，提高性能。
2. **优化用户体验：** 异步更新状态可以确保 React 在执行更新时，能够将多个状态更新请求合并为单个更新，从而避免多次不必要的重绘和布局计算。这样用户在与页面交互时会感觉更流畅，不会因频繁的 UI 变化而造成卡顿或闪烁。
3. **避免循环依赖和死锁：** 如果 `setState()` 是同步的，而且在更新状态后立即访问该状态，可能会导致循环依赖或死锁的情况。异步更新状态可以避免这种情况发生

React比Vue好在哪里

1. **组件化和灵活性**

   JSX 语法：React 使用 JSX 语法，将 HTML 和 JavaScript 结合在一起，这种方式允许开发者更紧密地控制组件的结构和逻辑。JSX 提供了更强的灵活性和表达力

   灵活的组件设计：React 允许开发者以多种方式创建和管理组件，包括类组件和函数组件（以及 Hooks），这种灵活性适合各种开发风格和需求

2. **虚拟 DOM 和性能优化**

   虚拟 DOM：React 使用虚拟 DOM 来高效地更新界面，通过最小化对真实 DOM 的操作来提高性能。虽然 Vue 也使用虚拟 DOM，但 React 在这个方面的性能优化和成熟度得到了广泛认可

   细粒度控制：React 的更新策略和细粒度控制使得开发者可以更精确地优化组件渲染和性能

3. **Hooks 和功能增强**

   Hooks API：React 的 Hooks API 提供了一种更简洁的方式来管理组件状态和副作用，使得函数组件变得更加强大和灵活。Hooks 让组件逻辑更加可重用和易于测试

   Context API：React 的 Context API 提供了一种更简便的全局状态管理方式，避免了复杂的状态管理库

4. **函数式组件**（优点）

   简洁性：函数式组件通常比类组件更简洁，代码更易于阅读和维护。它们仅由一个函数组成，没有复杂的类结构和生命周期方法

   性能优化：函数式组件由于没有 `this` 绑定和生命周期方法的复杂性，通常比类组件更轻量。此外，React 的 `React.memo` 可以用来优化函数式组件的性能，避免不必要的重新渲染

   生命周期简化：函数式组件中的 `useEffect` Hook 可以代替类组件中的多个生命周期方法，如 `componentDidMount`、`componentDidUpdate` 和 `componentWillUnmount`。这使得处理副作用和生命周期更为简洁

   类型支持：在 TypeScript 中，函数式组件的类型定义通常比类组件更简单，类型推断更加直观

Vue2和Vue3有什么不一样的

1. 性能优化

   Vue 3 使用基于 Proxy 的响应式系统，替代了 Vue 2 中基于 Object.defineProperty 的实现

2. 组合式Api(Composition API)，Vue 2 主要使用 Options API(选项式)
3. TypeScript 支持更好，开发体验更好
4. Vue 3 更加模块化，支持 Tree-shaking，从而减小打包体积
5. 支持返回多个根元素

new一个新对象，底层做了什么

1. 创建一个**新对象**：

   JavaScript 引擎会创建一个新的空对象，这个对象没有任何属性和方法

2. 链接到**原型**：

   新创建的对象会被关联到其构造函数的原型对象（prototype）。这通过 `Object.setPrototypeOf` 或者内部实现方式来完成，确保对象可以访问构造函数原型上的属性和方法

3. 将**构造函数**作为上下文调用：

   构造函数会被调用，并将新创建的对象作为 this 上下文

   这意味着在构造函数内部，可以使用 this 关键字来引用新创建的对象，从而设置对象的初始状态（即属性和方法）

4. 返回新对象：

   如果构造函数内部没有显式返回一个对象，则 new 表达式将返回这个新创建的对象。

   如果构造函数内部显式返回一个对象，则 new 表达式将返回这个返回的对象。

SSR

服务端渲染（Server-Side Rendering，简称 SSR）是一种渲染网页内容的方法，在这种方法中，网页内容在服务器端生成，然后发送到客户端进行展示。与客户端渲染（Client-Side Rendering, CSR）不同，SSR 在服务器上生成 HTML，确保客户端接收到的是一个完整的 HTML 页面。这对 SEO 和初次页面加载性能有显著的提升

- **SSR提升初始加载速度**：通过预先生成完整HTML，SSR可以显著提升页面的首屏渲染速度和SEO效果。
- **水合过程带来一定延迟**：初次交互时可能会受到水合过程的影响，导致响应稍慢。
- **动态更新性能与CSR相似**：一旦水合完成，页面的动态更新性能与纯客户端渲染的性能基本相同

为什么使用 SSR?

1. **SEO 优化**：搜索引擎爬虫可以更容易地抓取和索引页面内容，因为在请求时内容已经生成。
2. **更快的首屏加载时间**：首屏内容可以更快地呈现给用户，因为服务器生成了完整的 HTML，而不需要等待客户端 JavaScript 的执行和数据的获取。
3. **更好的性能**：由于初始内容加载更快，用户感觉到应用更快，更流畅。

SSR缺点:

1. **服务器负载增加**
2. **开发复杂性增加**
3. **响应时间增加**
4. **缓存和静态内容处理**

前端实现SSR

通过使用 Next.js 和 Nuxt.js，可以在现有的 React 和 Vue.js SPA 中轻松引入 SSR。这些框架提供了简便的配置和强大的功能，使得开发者能够在享受 SPA 动态交互优势的同时，获得更好的 SEO 和首屏加载性能

1. **React with Next.js**：
   - 使用 `getServerSideProps` 实现服务器端渲染
   - 使用文件路由系统，将页面组件放在 `pages` 目录下
2. **Vue.js with Nuxt.js**：
   - 使用 `asyncData` 钩子实现服务器端渲染
   - 采用类似文件路由系统，将页面组件放在 `pages` 目录下

SSR适用场景

- **推荐使用SSR的场景**：SEO非常重要、需要加快首屏渲染速度、页面内容相对静态且开发团队能够处理SSR的复杂性。
- **不推荐使用SSR的场景**：应用动态交互性强、实时数据需求多、内容高度个性化、或者项目资源有限，CSR可能是更好的选择

CSR

客户端渲染（Client-Side Rendering，简称 CSR）是一种渲染网页内容的方法，其中大部分或所有内容由客户端（通常是浏览器）处理。在这种方法中，服务器主要负责发送一个基础的 HTML 框架，JavaScript 在客户端运行并生成动态内容

为什么使用客户端渲染?

1. **动态交互**：客户端渲染使得应用可以在用户与页面交互时动态更新内容，而不需要重新加载整个页面。

2. **单页应用（SPA）**：CSR 是构建单页应用的基础。SPA 使得应用看起来和感觉更像是一个桌面应用，通过路由实现无刷新导航。
3. **模块化和组件化开发**：现代前端框架（如 React、Vue.js、Angular）使得前端开发更加模块化和组件化，易于管理和维护。

跨站脚本攻击（XSS）

XSS 攻击是通过注入恶意脚本代码，使其在受害者的浏览器中执行。攻击者可以利用 XSS 获取用户的敏感信息，如 cookies、会话令牌等。

防御措施：

- **输入验证和输出编码**：对所有用户输入进行严格验证，并对输出进行适当编码
- **使用安全的 API**：避免使用 `innerHTML`，改用 `textContent` 或 `innerText`
- **设置 Content Security Policy (CSP)**：CSP 可以防止未授权的脚本执行

跨站请求伪造（CSRF）

CSRF 攻击通过伪造用户请求，使用户在不知情的情况下执行某些操作，如转账、修改密码等。

防御措施：

- **使用 CSRF 令牌**：每次请求时生成一个唯一的令牌，并验证其有效性
- **验证请求来源**：检查请求头中的 `Origin` 和 `Referer` 字段，确保请求来自可信的源

Object.definePropery和Proxy

Vue2中**Object.definePropery**的问题

1. **深度监听的问题**

   使用 `Object.defineProperty()` 实现响应式时，需要递归地为对象的每个属性设置 getter 和 setter，以实现深度监听。这样在处理嵌套数据结构时，性能开销较大，且实现复杂

2. **新增或删除属性的监听**

   `Object.defineProperty()` 只能劫持对象的已有属性，无法监听新添加的属性（`Vue.set()` 的实现）或者属性的删除（`Vue.delete()` 的实现）

3. **性能问题**

   每次调用 `Object.defineProperty()` 都会直接修改对象，可能会影响 JavaScript 引擎的优化，造成性能损失，特别是在大型数据和频繁更新的场景下

4. **数组变化的监听**

   数组的变化（如修改元素、添加/删除元素）难以通过 `Object.defineProperty()` 直接监听到，需要额外的处理

Proxy

1. **更强大的拦截能力**：

   `Proxy` 可以代理整个对象而非属性，并通过附加的拦截器函数捕获对目标对象的所有操作，包括属性的读取、赋值、删除、属性遍历等

2. **更简洁的 API**：

   相比于 `Object.defineProperty()`，`Proxy` 提供了更为简单和一致的 API，可以监听整个对象以及其深层次的属性变化

3. **更好的性能和扩展性**

   `Proxy` 的拦截器函数在 JavaScript 引擎层面进行优化，比较 `Object.defineProperty()` 有更好的性能表现，特别是在处理大型数据和复杂对象时。

4. **支持数组变化的监听**：

   `Proxy` 可以直接监听数组的变化，如元素的修改、添加和删除操作，无需额外处理

```
// 创建一个目标对象
let target = {
    message: 'hello'
};

// 创建一个 Proxy 实例
let handler = {
    get: function(target, prop, receiver) {
        console.log(`Getting property "${prop}"`);
        return target[prop];
    },
    set: function(target, prop, value, receiver) {
        console.log(`Setting property "${prop}" to "${value}"`);
        target[prop] = value;
        return true;
    }
};

let proxy = new Proxy(target, handler);

// 通过 Proxy 实例访问和修改目标对象
console.log(proxy.message); // 输出: "Getting property "message" / "hello"
proxy.message = 'world'; // 输出: "Setting property "message" to "world"
console.log(proxy.message); // 输出: "Getting property "message" / "world"
```

浏览器地址栏中输入一个网址后做了什么

1. 用户输入网址
2. 浏览器解析 URL
3. DNS 解析
4. 建立 TCP 连接
5. 发送 HTTP 请求
6. 服务器处理请求
7. 服务器发送 HTTP 响应
8. 浏览器处理响应
9. 资源加载
10. 执行脚本
11. 页面呈现

vue模板绑定基本原理

Vue 的核心特性之一是响应式数据系统。它通过**劫持对象属性**的 getter 和 setter 实现数据的响应式。每当数据发生变化时，Vue 会自动更新绑定到这些数据的 DOM

响应式系统：通过数据劫持和依赖收集，Vue 实现了数据的响应式

模板编译：Vue 将模板编译为渲染函数，生成虚拟 DOM 树

虚拟 DOM：通过虚拟 DOM 和差分算法，Vue 高效地更新实际 DOM

v-model双向数据绑定基本原理

v-model vue官网

v-model组件绑定，简单的说就是把绑定的值和事件简写成v-model语法糖形式，实际上传过去的还是一个值，一个事件。子组件依旧需要注册监听事件

前端工程化

前端工程化旨在通过工具、流程和规范来提高前端开发的效率、质量和可维护性。

1. 比如自定义配置构建工具 Vite ，通过 vite 生成我们想要的工程文件目录，让最终文件包结构更加人性化，还可以通过vite的配置减少首屏加载的时间
2. 其次是 `eslint` 配置，应该如何规范一个工程项目的开发代码，提高代码的可维护性
3. 再然后是组件化开发，组件在项目里应该如何拆分。工程文件目录结构框架，以提升项目的开发效率
4. 最后是项目性能优化

前端性能优化

前端性能优化是提高 Web 应用程序响应速度、减少加载时间和提升用户体验的重要工作。以下是一些常用的前端性能优化策略：

1. **减少 HTTP 请求**

- 合并文件：将多个 CSS 和 JavaScript 文件合并成一个文件，以减少 HTTP 请求的数量
- 使用图像精灵（Sprite）：将多个小图标合并成一张图片，然后使用 CSS 背景定位显示所需的部分

2. **资源压缩与优化**

- 压缩 CSS、JavaScript 和 HTML：使用工具如 Terser、UglifyJS、cssnano、HTMLMinifier 等压缩代码，去除不必要的空白和注释
- 图像优化：使用工具如 ImageOptim、TinyPNG 对图片进行无损压缩，减少文件大小

3. **延迟加载与懒加载**

- 懒加载图片和视频：使用 `loading="lazy"` 或者 `IntersectionObserver` API 懒加载页面上的图片和视频，只在用户滚动到视口时加载
- 延迟加载 JavaScript：使用 `defer` 或 `async` 属性延迟加载非关键的 JavaScript 文件，避免阻塞页面渲染

4. **使用内容分发网络（CDN）**

- CDN：通过 CDN 分发静态资源，利用地理上接近用户的服务器来加快资源加载速度

5. **缓存优化**

- HTTP 缓存控制：使用 `Cache-Control` 和 `Expires` 头来指定静态资源的缓存策略，避免用户重复下载相同资源
- Service Workers：通过 Service Workers 实现离线缓存，为用户提供更快的加载体验

6. **代码分割**

- 按需加载：在大型单页应用中，使用 Webpack 的 `code splitting` 功能，将应用分成多个代码块，按需加载，减少初始加载时间
- 树摇（Tree Shaking）：使用 Webpack 或 Rollup 移除未使用的代码，减小打包后的文件大小

7. **减少重绘和重排**

- 优化 CSS 和 JavaScript：避免使用触发重排（如 `offsetWidth`、`clientHeight`）的 DOM 操作，尽量合并多次修改样式的操作，减少重排次数
- 使用 CSS3 动画：使用 GPU 加速的 CSS3 动画而非 JavaScript 动画，以提高动画性能

8. **使用现代浏览器特性**

- 预加载资源：使用 `<link rel="preload">` 预加载关键资源，如字体和关键 CSS 文件，加快页面渲染
- Prefetching：使用 `<link rel="prefetch">` 预取用户可能会访问的资源，减少未来加载时间

9. **减少第三方库的使用**

- 选择轻量级库：如果需要使用第三方库，选择轻量级的库（如 Lodash 的按需加载替代）
- 自定义解决方案：尽量使用原生 JavaScript 和 CSS 实现功能，减少对第三方库的依赖

10. **优先处理关键内容**

- 关键渲染路径优化：确保首屏内容尽早显示，减少阻塞渲染的资源（如同步 JavaScript）
- 字体加载优化：使用 `font-display: swap` 避免因字体加载导致的文字不可见

11. **网络请求优化**

- 使用 HTTP/2：HTTP/2 支持多路复用、压缩头部信息、服务器推送等特性，有助于加快页面加载速度
- 减少 cookie 大小：减小或避免在静态资源请求中附带过多的 cookie，减少请求头大小

12. **监控与分析**

- 性能监控工具：使用 Chrome DevTools、Lighthouse、WebPageTest 等工具定期检查和分析页面性能
- 用户体验监控：通过 Google Analytics、Sentry 等监控用户的实际体验，及时发现并修复性能瓶颈

http和https的主要区别

HTTP（HyperText Transfer Protocol）和HTTPS（HyperText Transfer Protocol Secure）是用于在Web浏览器和Web服务器之间传输数据的协议。虽然它们在功能上相似，但在安全性、数据传输方式等方面存在一些关键差别

1. **证书**
    - **HTTP**：**无证书要求**：HTTP协议不需要使用任何证书。客户端和服务器之间的通信是开放的，没有加密，也没有身份验证机制。这意味着任何人都可以访问HTTP网站，而无需进行额外的身份验证
    - **HTTPS**：**需要SSL/TLS证书**：HTTPS协议要求服务器必须拥有一个有效的SSL/TLS证书，以便建立加密连接并验证服务器的身份。SSL（Secure Sockets Layer）和TLS（Transport Layer Security）是加密通信的协议，TLS是SSL的升级版，目前大部分网站使用的是TLS
2. **安全性**：
    - **HTTP**：不加密，数据以明文形式传输，容易被窃听或篡改
    - **HTTPS**：使用SSL/TLS加密数据传输，提供数据的机密性、完整性和身份验证

    端口：
    - **HTTP**：默认使用端口80
    - **HTTPS**：默认使用端口443

    身份验证：
    - **HTTP**：没有身份验证机制，无法验证服务器身份
    - **HTTPS**：通过数字证书验证服务器身份，防止钓鱼网站

    SEO和浏览器提示：
    - **HTTP**：浏览器通常会标记为“不安全”，对SEO不利
    - **HTTPS**：浏览器显示锁形图标，提升用户信任度，有助于SEO优化

HTTP/1.1和HTTP/2的区别

1. **数据格式**：
   - HTTP/1.1使用文本格式传输数据
   - HTTP/2使用二进制格式，传输效率更高
2. **多路复用**：
   - HTTP/1.1每个连接只能处理一个请求，存在队头阻塞问题
   - HTTP/2支持多路复用，可以在一个连接中并行处理多个请求，解决了队头阻塞问题
3. **头部压缩**：
   - HTTP/1.1不压缩头部信息，增加传输开销
   - HTTP/2使用HPACK算法对头部信息进行压缩，减少冗余数据
4. **服务器推送**：
   - HTTP/1.1没有服务器推送功能
   - HTTP/2支持服务器推送，可以提前向客户端发送资源，提高加载速度
5. **连接管理**：
   - HTTP/1.1通过长连接（Keep-Alive）维持连接，但请求是顺序进行的
   - HTTP/2默认使用长连接，并支持并行处理多个请求

这些改进使HTTP/2在速度和效率上比HTTP/1.1有了显著提升