1、说一说 flex 布局

Flex 是 CSS3 的一种弹性布局，用来让元素在容器中灵活排列。
它通过 display: flex 启用，有两根轴线：主轴和交叉轴。
常用的容器属性有：
flex-direction（主轴方向）、justify-content（主轴对齐）、
align-items（交叉轴对齐）、flex-wrap（是否换行）。
是现代前端开发中最常用的布局方式之一。

2、响应式布局

响应式布局是指网页能根据不同设备的屏幕大小自动调整结构和样式，实现多端适配。
常用实现方式有：
① **媒体查询（@media）** 根据屏幕宽度切换样式；
② **弹性布局（Flex/Grid）** 自动分配空间；
③ **rem + viewport** 实现移动端缩放；
④ 还可以结合框架如 Bootstrap。
核心目标是：一套代码，多端适配，提升用户体验和维护效率。

响应式布局（Responsive Layout） 是指网页能够根据**不同设备的屏幕大小、分辨率、方向** 自动调整排版和样式，以在 **PC、平板、手机** 等设备上都能获得良好体验的布局方式。
核心目标是：
一套代码，多端适配。

3、margin 坍塌

margin 坍塌是指 **垂直方向相邻的两个块级元素外边距会合并**，最终取两者中较大的值，而不是相加。
常见的三种情况是：
1.相邻兄弟元素之间；
2.父子元素之间；
3.空块元素自身上下 margin。
解决方法主要有：

- 给父元素加 overflow: hidden;（形成 BFC）；
- 或加 padding/border；
- 或使用 display: flex 等新布局。

核心思想：**让元素形成独立的布局上下文，就能避免 margin 坍塌。**

4、var，let，const

一、三者的定义

- **var**：ES5 定义变量
- **let / const**：ES6 新增，用来替代 `var`，拥有更合理的作用域与行为。

var、let、const 的主要区别在作用域、变量提升和可修改性。
var 是函数作用域，有变量提升，可重复声明；
let 和 const 都是块级作用域，没有变量提升，不能重复声明；const 声明时必须赋值，且不能修改绑定；
一般 **const 定义常量、let 定义变量**，避免使用 var。

特性	`var`	`let` / `const`
作用域类型	**函数作用域**	**块级作用域**（由 `{}` 界定）
可见性	在整个函数内部都可见。	只在声明它的 `{}` 块内部可见。
示例	在 `if` 或 `for` 循环中声明的变量，在外部仍然可访问。	在 `if` 或 `for` 循环中声明的变量，在外部会报错（`ReferenceError`）。

5、数组常见方法

🧩 一、不会改变原数组的方法（纯函数类）

这些方法返回一个新数组或值，不会修改原数组内容。

方法	作用	返回值
map()	对每个元素执行回调函数	新数组
filter()	过滤符合条件的元素	新数组
slice(start, end)	截取部分数组	新数组
concat()	合并数组	新数组
reduce()	累积计算（如求和）	计算结果
find() / findIndex()	查找第一个符合条件的元素或索引	元素 / 索引
includes()	判断数组是否包含某值	布尔值
every() / some()	判断所有/部分元素是否满足条件	布尔值
join()	将数组转为字符串	字符串

🧩 二、会改变原数组的方法（原地操作类）

这些方法直接修改原数组的内容。

方法	作用	备注
push()	末尾添加元素	返回新长度
pop()	删除末尾元素	返回删除的元素
shift()	删除首个元素	返回删除的元素
unshift()	在开头添加元素	返回新长度
splice(start, deleteCount, ...items)	删除/替换/插入元素	强大但会改原数组
sort()	排序	改变原数组
reverse()	反转数组	改变原数组

数组常见方法分为两类：
不改变原数组的有：map、filter、slice、concat、reduce、find、includes、every、some 等。
会改变原数组的有：push、pop、shift、unshift、splice、sort、reverse 等。
实际开发中，我们通常优先使用**不改变原数组的函数式方法**，保证数据的可控性和可维护性。

6、事件冒泡

事件冒泡是 DOM 事件流的**第三个阶段**，指的是事件从其触发的**目标元素**开始，沿着 DOM 树向它的父元素、祖父元素依次**向上传播**，直至 document 和 window 的过程。在默认情况下，大部分事件监听器都在这个阶段被触发。事件冒泡的核心应用是实现 **事件委托（Event Delegation）**，通过在父元素上统一处理子元素事件，从而达到**优化性能**的目的。如果需要，我们可以使用 event.stopPropagation() 来阻止事件继续向上传播。

事件流的三个阶段

在浏览器事件机制中，事件流分为三个阶段：

1. **捕获阶段（Capturing Phase）**：事件从 window 向下传播到目标元素；
2. **目标阶段（Target Phase）**：事件到达目标元素；
3. **冒泡阶段（Bubbling Phase）**：事件从目标元素向上传递回 window。

📌 事件监听函数默认在**冒泡阶段**触发。

事件委托：**事件委托**是指将子元素上的事件监听器绑定到它们的**共同父元素**上，利用事件冒泡机制，由父元素统一接收并处理所有子元素触发的事件。

原理总结

1. 用户点击 <li> Item 1。
2. 事件首先在 <li> 上触发，然后进入 冒泡阶段。
3. 事件沿着 DOM 树向上冒泡到 。
4. 上的监听器接收到这个事件。
5. 通过 event.target 属性，我们依然可以精确地知道最初被点击的是 <li> 元素。
6. 上的函数根据 event.target 来执行对应的操作。

HTML

```
<ul id="list">
  <li data-id="1">Item 1</li>
  <li data-id="2">Item 2</li>
  <li data-id="3">Item 3 (新增)</li>
</ul>
<script>
  const list = document.getElementById('list');

  // 只需要在父元素上绑定一个监听器
  list.addEventListener('click', function(event) {
    // 关键步骤 1: 识别触发事件的元素
    const targetElement = event.target;

    // 关键步骤 2: 确保点击的是我们想要的子元素（例如 <li>）
    // 通过判断标签名或使用自定义属性（如 data-id）
    if (targetElement.tagName === 'LI') {
        const id = targetElement.getAttribute('data-id');
        console.log(`事件委托：点击了 ID 为 ${id} 的项`);

        // 在这里执行对该子元素的操作，例如：
        targetElement.style.backgroundColor = 'yellow';
    }
  });
</script>
```

7、vue2 和 vue3 区别

Vue3 在底层、语法和性能上都进行了重大优化。

底层使用 **Proxy** 代替 Vue2 的 Object.defineProperty 实现响应式，能监听对象新增/删除属性、数组索引变化。

新增 **Composition API 组合式 API**（如 setup、ref、reactive），相对于 vue2 的选项式 API （Options API）逻辑更聚合、可复用性更强。

生命周期命名变化。

同时 Vue3 用 TypeScript 重写，性能更高、体积更小。

8、 vue 为什么会引入虚拟 dom

Vue 引入虚拟 DOM 是为了 **提升性能和可维护性**。
因为直接操作真实 DOM 成本很高，Vue 用 **JS 对象模拟真实 DOM 结构**，在内存中通过 Diff 算法对比新旧虚拟 DOM 树，只更新变化的部分，从而 **减少 DOM 操作、提升渲染效率**。
同时虚拟 DOM 也让 Vue 能实现 **跨平台渲染**，不仅能在浏览器中运行，也能用于 SSR 或原生端。

9、 mvvm

MVVM 是一种前端架构模式，全称是 Model–View–ViewModel。
它通过 ViewModel 把数据（Model）和视图（View）连接起来，实现了 **数据驱动视图**。
当数据变化时，视图会自动更新；当用户操作视图时，也会自动更新数据，实现 **双向绑定**。
这种模式能减少 DOM 操作，让代码更清晰、维护更方便。

一、什么是 MVVM?

MVVM 是一种前端架构设计模式，全称 **Model–View–ViewModel**，它的核心思想是：
👉 **通过数据驱动视图，让视图和数据自动保持同步，而不是直接操作 DOM。**

二、MVVM 的三个核心部分

模块	说明
Model（模型层）	负责存储和处理数据，比如接口返回的数据或本地状态。
View（视图层）	用户界面部分（DOM / 模板），负责展示数据。
ViewModel（视图模型层）	连接 View 和 Model，负责数据绑定、事件监听等逻辑。Vue 实例本质上就是 ViewModel。

三、MVVM 的工作原理

1️⃣ **数据变化时：**
Model → ViewModel 通过数据监听（Vue2 用 `Object.defineProperty` / Vue3 用 `Proxy`）侦测变化；
ViewModel → 自动更新绑定的 View（模板）。

2️⃣ **视图变化时：**
View 上的事件（如输入框 input）通过双向绑定（v-model）反馈到 ViewModel；
ViewModel 再更新 Model 中的数据。

🔁 从而实现了数据与视图的 **双向绑定、自动同步。**

10、 单向数据流是什么

单向数据流指的是数据只能从父组件流向子组件，不能反向修改。
在 Vue 中，父组件通过 props 向子组件传值，子组件如需修改数据，只能通过$emit 触发事件通知父组件，再由父组件更新后重新传递。
这种方式让数据流向清晰、状态可控，便于调试和维护。

11、 为什么会有 react 和 vue 这些框架的出现

React 和 Vue 的出现，主要是为了解决传统前端开发中 **手动操作 DOM、数据与视图不同步、项目难维护** 等问题。它们通过 **数据驱动视图、虚拟 DOM、组件化开发** 等机制，让开发者只关注数据逻辑，框架自动处理页面更新，从而大幅提升了性能、可维护性和开发效率。

12、 作用域链

作用域链是 JavaScript 中的一种变量查找机制。
当访问一个变量时，JS 会先在当前作用域查找，如果找不到，就沿着函数的父级作用域一直向上查找，直到全局作用域为止。
这种由多个作用域层层嵌套形成的链式结构，就叫做作用域链。
它保证了内部函数可以访问外部函数的变量。

一、什么是作用域（Scope）

> **作用域**：指变量和函数能被访问的范围。
> 在 JS 中，主要有三种作用域：

1. **全局作用域**（最外层）
2. **函数作用域**
3. **块级作用域（let / const 引入）**

二、作用域链的定义

> **作用域链（Scope Chain）** 是一种 **变量查找机制**，当访问一个变量时，
> JS 引擎会 **从当前作用域开始**，逐层向上查找父级作用域，直到全局作用域为止。
> 如果找不到，就会报 `ReferenceError`。

简单来说：
内部作用域可以访问外部作用域的变量，反之不行。

三、作用域链的形成过程

当 JS 代码执行时，每个函数都会创建一个执行上下文（Execution Context）。
这个上下文中保存着：

- 当前函数的变量对象（变量、函数声明）
- 以及对外层作用域的引用（`[[Scope]]`）

多个执行上下文按调用关系形成一条链，这就是**作用域链**。

🔗 举个例子：

```js
var a = 1
function outer() {
  var b = 2
  function inner() {
    var c = 3
    console.log(a, b, c)
  }
  inner()
}
outer()
```

执行 `console.log(a, b, c)` 时：
👉 JS 会先在 `inner` 查找 → 没找到再去 `outer` → 最后去全局 → 找到后停止。
这就是作用域链的查找规则。

其它补充：
map/set/weaimap 是什么呢

特性	Map	Set	WeakMap
存储内容	键值对	唯一的值	键值对
键/元素类型限制	任何类型	任何类型	**键必须是对象**
对键的引用	**强引用**（阻止垃圾回收）	**强引用**	**弱引用**（不阻止垃圾回收）
可迭代性	是（可获取 `.size`）	是（可获取 `.size`）	**否**（不可迭代，无 `.size` 属性）
主要应用	灵活的键值存储	数组去重、成员检查	避免内存泄漏、DOM 元素关联数据

您提出的这三个都是 ES6 引入的新的数据结构，它们提供了比传统对象（{}）和数组（[]）更强大、更高效或更有内存管理意识的功能。
其中 Map 和 Set 是**强引用**数据结构，而 WeakMap 和 WeakSet 是**弱引用**数据结构。

三次握手（Three-Way Handshake）”和“四次挥手（Four-Way Handshake）”是 **TCP (Transmission Control Protocol, 传输控制协议)** 建立和终止连接的核心机制。

1. TCP 三次握手（建立连接）

三次握手的目的是在客户端（Client）和服务器（Server）之间建立可靠的连接，并确认双方的**发送和接收能力**。

核心过程

假设客户端 (A) 主动发起连接，服务器 (B) 响应。

步骤	发送方	接收方	报文内容	作用（确认能力）
第一次握手	A → B	B	**SYN=1, Seq=x**	**A：** 客户端发送连接请求（SYN），并告诉服务器自己想发送的初始序列号 x。
第二次握手	B → A	A	**SYN=1, ACK=1, Seq=y, Ack=x+1**	**B：** 服务器确认收到了 A 的请求（ACK=1, Ack=$x+1$），同时发送自己的连接请求（SYN=1, Seq=y）。**B 确认了 A 的发送能力。**
第三次握手	A → B	B	**ACK=1, Ack=y+1**	**A：** 客户端确认收到了 B 的响应和请求（ACK=1, Ack=$y+1$）。**A 确认了 B 的接收和发送能力。**

2. TCP 四次挥手（终止连接）

四次挥手的目的是安全地关闭连接，确保**双方数据都传输完毕**。由于 TCP 是全双工通信（数据可以同时双向流动），因此关闭连接需要分别关闭这两个方向的连接。

核心过程

假设客户端 (A) 主动发起断开连接。

步骤	发送方	接收方	报文内容	状态变化 & 作用
第一次挥手	A → B	B	**FIN=1, Seq=u**	**A：** 客户端发出 FIN 报文，表示 **A 不再向 B 发送数据。** A 进入 `FIN_WAIT_1` 状态。
第二次挥手	B → A	A	**ACK=1, Ack=u+1**	**B：** 服务器发送 ACK 确认收到 A 的关闭请求。B 进入 `CLOSE_WAIT` 状态。**此时连接处于半关闭状态：A 不能发，但 B 仍然可以向 A 发送数据。**
第三次挥手	B → A	A	**FIN=1, Seq=w**	**B：** 服务器的数据发送完毕后，也发送 FIN 报文，表示 **B 也不再向 A 发送数据。** B 进入 `LAST_ACK` 状态。
第四次挥手	A → B	B	**ACK=1, Ack=w+1**	**A：** 客户端发送 ACK 确认收到 B 的关闭请求。A 进入 `TIME_WAIT` 状态，等待 2MSL 时间后，关闭连接。B 收到 ACK 后立即关闭连接。

为什么需要四次？

- **全双工特性：** 第一次和第二次挥手（FIN + ACK）只保证了**客户端到服务器**这个方向的数据发送完毕。
- 服务器收到客户端的 FIN 后，不能立即关闭连接，因为它可能还有**未发送完的数据**给客户端。
- 因此，服务器需要分成两步：
    1. 先发送 ACK 确认收到关闭请求（第二次挥手）。
    2. 等待所有数据发送完毕后，再发送自己的 FIN 报文（第三次挥手）。

这就是为什么需要四次往返来确保双向连接都安全关闭。

Hooks 是 Vue 3 (Composition API) 和 React 框架中，用于在**函数组件**中实现**状态逻辑复用**和**生命周期管理**的核心机制。

在 Vue 3 中，Hooks 通常被称为 **Composition API 函数**，用于在 `setup` 块中构建组件逻辑。

Hook 名称	类别	作用和用途
`ref()`	状态管理	定义**响应式数据**。它接收一个内部值，返回一个具有 `.value` 属性的对象，通过 `.value` 读写可以触发视图更新。适用于原始类型数据。
`reactive()`	状态管理	定义**响应式对象或数组**。它返回一个原始对象的响应式代理。适用于复杂数据类型。
`computed()`	状态管理	创建一个**计算属性**。当其依赖的响应式数据变化时，`computed` 会自动重新计算其值并更新视图。
`watch()`	侦听器	侦听一个或多个响应式数据源，并在数据源变化时执行副作用函数。用于**响应数据变化**执行异步或复杂逻辑。
`onMounted()`	生命周期	在组件**挂载到 DOM 之后**调用。常用于 DOM 操作、使用第三方库或发送不依赖数据的初始化请求。
`onUnmounted()`	生命周期	在组件**从 DOM 卸载之前**调用。常用于清理副作用，如清除定时器、移除事件监听器。
`toRefs()`	工具	将一个 `reactive` 对象的**所有属性**转换为独立的 `ref` 引用，方便结构化后使用。

ES6（ECMAScript 2015）是 JavaScript 语言发展历史上的一个**里程碑式**的版本，引入了大量新的语法特性和功能，极大地改善了现代 JavaScript 的开发体验、可读性和工程化能力。

一、 声明与作用域的改进（核心）

新特性	描述	为什么重要?
`let` 和 `const`	引入了**块级作用域**（Block Scope）。`let` 用于声明变量，`const` 用于声明常量（必须初始化且不可重新赋值）。	解决了 `var` 的**变量提升**和**函数作用域**带来的变量污染和覆盖问题，使作用域管理更清晰、更安全。
箭头函数（Arrow Functions）	语法更简洁的函数定义方式，如 `(a, b) => a + b`。	最重要的特性是它**没有自己的 `this` 绑定**，`this` 始终指向其定义时的外部作用域（词法作用域）。解决了传统函数中 `this` 指向不明确的问题。

二、 数据结构与集合

新特性	描述	为什么重要?
`Map` 和 `Set`	`Map` 是一种支持任意类型键（包括对象）的键值对集合；`Set` 是一种值唯一的集合。	提供了比普通对象和数组更高效、更灵活的数据存储方式，特别是 `Set` 可以轻松实现数组去重。
`WeakMap` 和 `WeakSet`	弱引用的 Map 和 Set。键（`WeakMap`）或元素（`WeakSet`）只能是对象，且是**弱引用**。	解决了**内存泄漏**问题。当键或元素对象在其他地方不再被引用时，垃圾回收机制会自动将其从集合中清除。

三、 函数与面向对象

新特性	描述	为什么重要?
Class（类）	引入了基于原型的面向对象编程的语法糖。使用 `class`、`constructor`、`extends` 等关键字。	让 JavaScript 的面向对象编程更贴近传统面向对象语言的习惯，但其底层本质仍是基于原型链。
模块化（`import` / `export`）	引入了官方的模块系统（ES Modules）。	解决了早期的 CommonJS/AMD 模块化方案不统一的问题，是前端工程化和 Tree Shaking（摇树优化）的基础。
默认参数（Default Parameters）	允许在函数定义时为参数设置默认值。	简化了参数缺失时的判断逻辑，避免了使用 `a = a`

四、 语法糖与便捷操作

新特性	描述	为什么重要?
模板字面量（Template Literals）	使用反引号 `` ` `` 定义字符串，支持多行和**字符串插值**（`${}`）。	极大地简化了拼接字符串和嵌入变量的操作，提高了代码可读性。
解构赋值（Destructuring）	允许从数组和对象中提取值，对变量进行赋值，语法简洁。	简化了从对象或数组中获取多个属性或元素的常见操作。
展开运算符（Spread Operator）	使用三个点 `...`，用于数组或对象的展开。	简化了数组合并、克隆、函数参数传递等操作，是函数式编程中不可或缺的工具。
`for...of` **循环**	专门用来遍历**可迭代对象**（如数组、`Map`、`Set`、`String`、`TypedArray`）的值。	解决了 `for...in` 循环的缺点（遍历键名，且会遍历原型链上的属性），是推荐的数组遍历方式。

五、 异步编程

新特性	描述	为什么重要?
Promise	引入了处理异步操作的统一标准解决方案。	解决了传统的回调地狱（Callback Hell）问题，使异步代码的流程控制更加清晰和链式化。
Generator	一种特殊的函数，可以通过 `yield` 关键字暂停和恢复执行。	为更复杂的异步控制（如 `async/await` 的底层实现）提供了基础。