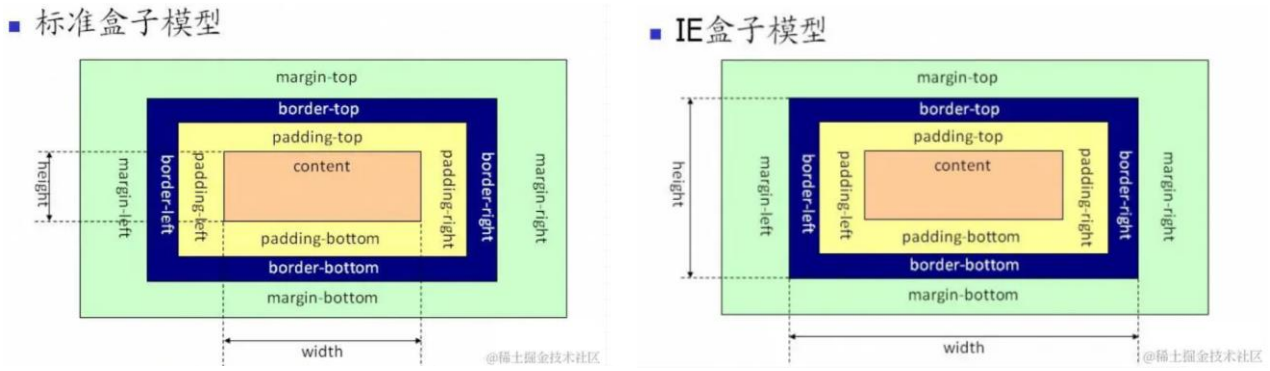
1.说一说 HTML 语义化?

HTML 语义化是指在编写 HTML 代码时，使用具有明确含义的标签（如<header>、<nav>、<article>等）来描述页面结构和内容含义，而非仅依赖无语义的通用标签（如 ）。其核心目标是让代码更易读、更易维护，同时提升搜索引擎优化（SEO）、无障碍访问等能力。

2.说一说盒模型?



在浏览器中，每个元素都可以看作一个矩形盒子，这就是盒模型。
盒模型由四层组成：content（内容）、padding（内边距）、border（边框）、margin（外边距）。
盒模型有两种类型：
content-box（默认）：width/height 只指 content 内容区域，padding 和 border 会额外增加元素占用空间。
border-box：width/height 包含 content 内容 + padding + border
以上两种可以通过 box-sizing 来设置

3.说一下浮动?

浮动（float）最初是为文字环绕图片设计的，现在也常用于布局。
设置浮动后，元素会脱离标准文档流，但仍在文本流中，因此可能导致父元素高度塌陷。
常见的清除浮动方式有四种：

1. 给父元素设置 overflow: hidden（触发 BFC）；
2. 给父元素固定高度；
3. 使用空 div 设置 clear: both；
4. 使用伪元素清除浮动（推荐方式）。

```
.clearfix::after {
  content: '';
  display: block;
  clear: both;
}
```

“浮动造成盒子塌陷”指的是：浮动的子元素脱离文档流后，父元素的高度无法被子元素撑开，导致父元素高度为 0。

4.说一说样式优先级的规则是什么

!important 最大，行内次之；
ID 高于类，类高于标签；
同级比较，后者覆盖前者。

类型	举例
!important	color: red !important;
内联样式（行内）	
ID 选择器	#box {}
类 / 属性 / 伪类选择器	.class 、[type="text"]、:hover
标签 / 伪元素选择器	div 、p::before
通配符、继承样式	* 、继承来的样式

5. 说一说 CSS 尺寸设置的单位

1. px：绝对单位，取决于屏幕分辨率。
2. rem：相对单位，相对于根元素的大小。
3. em：相对单位，相对于自身的大小，如果自己没设置 font-size，则根据父元素的大小。
4. vw：相对单位，相对于视口宽度的 1%。5. vh：相对单位，相对于视口高度的 1%。

6. 说一说 BFC

1.定义：块级格式化上下文，独立的渲染区域，不会影响边界外的元素
2.形成条件：①浮动 ②非静态定位 static ③overflow: hidden; ④display: table
3.布局规则：①区域内容 box 从上到下排列 ②box 垂直方向的距离由 margin 决定 ③同一个 BFC 内 box 的 margin 会重叠 ④BFC 不会与 float 元素重叠 ⑤BFC 计算高度也会计算 float 元素
4.解决的问题：①解决浮动元素重叠问题 ②解决父元素高度塌陷问题 ③解决 margin 重叠问题
定义：
BFC（Block Formatting Context，块级格式化上下文）是网页中的一个**独立渲染区域**，它内部的元素布局不会影响到外部，也不会被外部元素影响。可以理解为一个**相互隔离的“容器”**。

BFC 是块级格式化上下文，可以理解为一个独立的渲染区域，内部的布局不会影响外部。
它可以通过设置 float、overflow: hidden、position: absolute 或 display: flex 等方式触发。
BFC 的布局规则包括内部盒子垂直排列、同一 BFC 内 margin 会重叠、不同 BFC 间不会重叠、会包裹浮动元素等。
在实际开发中，它常用于解决三类问题：父元素高度塌陷、margin 重叠、浮动重叠。

7 说几个未知宽高元素水平垂直居中方法

要求： 元素宽高未知，仍需实现水平垂直居中。
→ 这意味着 **不能写固定宽高**，也**不能用 margin-left/top 的负值法**。
因此我们必须使用 **自适应布局** 的方式。

🥇 方法一：Flex 布局（最推荐 ✅）

```css
.parent {
  display: flex;
  justify-content: center; /* 水平居中 */
  align-items: center;     /* 垂直居中 */
}
```

🥉 方法三：绝对定位 + transform（经典万能法）

```css
.parent {
  position: relative;
}
.child {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}
```

优点： 兼容性好，不依赖 flex/grid。
原理： 先把左上角移动到父容器中心，再用 `translate` 把自身往回移动一半。

8 说一说三栏布局的实现方案

🧩 一、什么是"三栏布局"

典型结构：

```
diff

+-----------------------------+
|   左栏   |   中间内容   |   右栏   |
+-----------------------------+
```

要求：

- 左右两栏宽度固定；
- 中间栏宽度自适应；
- 三栏在同一行显示。

✅ 方案3：Flex 布局（推荐 ✅） 保存的记忆已满

```html
<div class="container">
  <div class="left">左</div>
  <div class="center">中</div>
  <div class="right">右</div>
</div>
```

```css
.container {
  display: flex;
}
.left, .right {
  width: 200px;
}
.center {
  flex: 1; /* 自适应 */
}
```

📌 **原理：**
Flex 让子元素在一行分布，左右两栏定宽，中间栏自适应填满剩余空间。

9.说一说 JS 数据类型有哪些,区别是什么?

JavaScript 一共有 8 种数据类型：7 种基本类型（Number、String、Boolean、Undefined、Null、Symbol、BigInt）和 1 种引用类型 Object（Object、Array、Function）。
基本类型存储在栈中，按值传递；引用类型存储在堆中，按引用传递。

10.说一说 null 和 undefined 的区别，如何让一个属性变为 null

在 JavaScript 中，undefined 表示变量已声明但未赋值，是系统默认的初始值；
null 表示空对象引用，通常由开发者主动赋值，表示"该变量目前没有对象"。
它们在类型上不同：typeof null === 'object' 是历史遗留 bug。
typeof undefined; // "undefined"
typeof null; // "object" <-- 历史遗留 bug
让一个属性变为 null 只需直接赋值：obj.prop = null;，这表示属性依然存在，只是值为空，而不是被删除。

11.说一说 JavaScript 有几种方法判断变量的类型?

JavaScript 中常用四种方式判断变量类型：

- typeof：适合判断基本类型；
- instanceof：判断引用类型实例；
- constructor：通过构造函数判断；
- Object.prototype.toString.call()：最准确、通用，推荐使用。
  实际开发中通常结合使用，例如 typeof + Array.isArray() 或 toString.call() 来确保准确性。

12.说一说数组去重都有哪些方法?

JavaScript 数组去重常见方法有：

1. 利用 Set（最推荐，简洁高效）；
2. 使用 indexOf 或 includes 遍历判断；

一、去重的核心目标

输入一个数组：

```js
let arr = [1, 2, 2, 3, 4, 4, 5, '5', NaN, NaN];
```

输出：

```js
[1, 2, 3, 4, 5, '5', NaN]
```

（要求重复元素只保留一个）

✅ 方法1：利用 `Set` 去重（最简单 & 推荐）

```js
let result = [...new Set(arr)];
console.log(result);
```

📌 原理：

> ES6 的 `Set` 结构会自动过滤掉重复的值，包括 `NaN`。

✅ 方法2：`indexOf` + 遍历

```js
let result = [];
for (let i = 0; i < arr.length; i++) {
  if (result.indexOf(arr[i]) === -1) {
    result.push(arr[i]);
  }
}
console.log(result);
```

📌 原理：

> 每次判断当前元素是否在结果数组中，不在则加入。

📈 优点：

- 兼容性最好；
- 理解简单。

📉 缺点：

- 效率较低（双层循环思想）。

✅ 方法3：`includes` + 遍历（语义更清晰）

```js
let result = [];
for (let value of arr) {
  if (!result.includes(value)) {
    result.push(value);
  }
}
console.log(result);
```

📌 区别：

> `includes` 比 `indexOf` 可检测 `NaN`。

📈 优点：

- 更语义化；
- 支持 `NaN` 去重。

13.说一说伪数组和数组的区别?

JavaScript 中的伪数组是**具有 length 属性和按索引访问特性，但不具备数组方法**的对象。常见的有 arguments、NodeList、HTMLCollection 等。
它与数组的区别在于：

1. 类型不同（伪数组是对象）；
2. 伪数组没有继承 Array.prototype；
3. 伪数组无法直接使用数组方法。
   如果要使用数组方法，可以通过 Array.from() 或 [...伪数组] 转换为真正的数组。

特性	`arguments`	`NodeList`	`HTMLCollection`
用途	函数的传入参数集合	DOM 节点（元素、文本等）集合	DOM 元素（仅元素节点）集合
产生	函数调用时自动创建	`document.querySelectorAll()`、`Node.childNodes` 等	`document.getElementsBy...`、`Element.children` 等
动态性	N/A	可能静态（`querySelectorAll`）或动态（`childNodes`）	**总是动态的（live）**
类型	类数组对象	Web API 接口	Web API 接口

14.说一说 map 和 forEach 的区别?

forEach 和 map 都用于遍历数组。

- forEach 只是单纯遍历，**没有返回值**，通常用于执行副作用（如打印、修改原数组等）；
- map 会**返回一个新数组**，每个元素是回调函数的返回结果，**不会修改原数组**。
  实际开发中，若需要数据转换用 map，只做遍历则用 forEach。

15.说一说 es6 中箭头函数?

箭头函数是 ES6 提供的一种更简洁的函数写法。
它没有自己的 this、arguments、super 或 new.target，this 的指向在定义时确定，取决于外层作用域。（window 或者 undifined）
箭头函数常用于回调函数（如 setTimeout、map、filter 等）中，可以避免普通函数中 this 丢失的问题。
但不能作为构造函数，也不适用于对象方法。
不能作为构造函数的原因：箭头函数缺少成为构造函数所需的所有底层机制（this 绑定和 prototype 属性）
不适用于对象方法：也是 this 指向问题

16.事件扩展符用过吗(...)，什么场景下?

ES6 中的 ... 有两种用法：
1 作为 **扩展运算符（spread）**：用于展开数组、对象、字符串等，可实现数组的克隆、合并，对象合并，浅拷贝等；
2 作为 **剩余参数（rest）**：用于函数形参或解构赋值中，用于收集多余参数或属性。
区别在于：扩展运算符用于“展开”，而剩余参数用于“收集”。

补充一个浅拷贝和深拷贝：

在 JavaScript 中，**浅拷贝**只复制对象的第一层属性，如果属性值是引用类型（比如对象或数组），拷贝的只是它的引用；而**深拷贝**会完整复制所有层级的数据，互不影响。
常见的浅拷贝方法有 Object.assign()、展开运算符 ... 等；深拷贝可以用 JSON.parse(JSON.stringify(obj) 或像 structuredClone()、lodash.cloneDeep() 这样的工具。
区别在于：修改浅拷贝对象的引用属性会影响原对象，而深拷贝不会。

浅拷贝示例

JavaScript

```
const original = {
  name: 'A',
  score: 10,
  data: {
    count: 5 // 这是一个嵌套对象
  },
  list: [1, 2] // 这是一个嵌套数组
};

// 使用展开运算符进行浅拷贝
const shallowCopy = { ...original };

// 1. 修改第一层属性（原始值）：互不影响
shallowCopy.score = 20;
console.log(original.score); // 10
console.log(shallowCopy.score); // 20

// 2. 修改嵌套对象的属性（引用值）：互相影响！
shallowCopy.data.count = 99;
console.log(original.data.count); // 99 (原对象也被修改了！)
```

JavaScript

```
const original = {
  name: 'A',
  data: {
    count: 5
  },
};

// 使用 JSON 方法实现深拷贝（注意其局限性）
const deepCopy = JSON.parse(JSON.stringify(original));

// 1. 修改第一层属性：互不影响
deepCopy.name = 'B';
console.log(original.name); // 'A'
console.log(deepCopy.name); // 'B'

// 2. 修改嵌套对象的属性：互不影响！
deepCopy.data.count = 99;
console.log(original.data.count); // 5 (原对象未被修改)
console.log(deepCopy.data.count); // 99
```

- JSON.stringify()可将对象转为标准化 JSON 字符串
- JSON.parse()可将符合规范的 JSON 字符串还原为 JavaScript 对象

17.说一说你对闭包的理解?

闭包（Closure）是指**一个函数可以访问其外层函数作用域中的变量**，即使外层函数已经执行结束。
本质上，闭包是 **函数和其外部词法环境（Lexical Environment）** 的组合。因为 JavaScript 是词法作用域，所以函数在定义时就确定了它能访问哪些变量。
闭包常用于**数据私有化（模拟私有变量）**、**延长变量生命周期**、以及**函数工厂**等场景。

例如：

js 复制代码

```
function createCounter() {
  let count = 0; // 外层作用域变量
  return function() {
    return ++count; // 内层函数访问外层变量 => 闭包
  };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

在这个例子中，`count` 并不会因为 `createCounter()` 执行结束而销毁，因为它被内部函数引用着，这就是闭包。

const counter = createCounter(); 这行代码的意思是：执行外部函数 createCounter，并将它返回的那个“内部函数”赋值给变量 counter。

闭包是指有权访问另一个函数作用域中变量的函数。它的创建方式是定义一个**嵌套函数**，内部函数可以访问外部函数的变量。
原理：基于 JavaScript 的**词法作用域**和**作用域链机制**，内部函数会“记住”它定义时所处的外部作用域，因此即使外部函数执行结束，变量依然能被访问。
作用：闭包可以让函数内部的局部变量在函数执行结束后依然保存在内存中，从而实现**数据持久化**或**模拟私有变量**等功能。
问题：由于闭包会使外部函数的变量一直被引用，垃圾回收器无法回收这些变量，若使用不当可能导致**内存泄漏**，长期累积可能引发**内存溢出**。

词法作用域：在哪里定义决定了**变量可访问的范围**，它是静态的、基于代码结构的。
作用域链：查找变量的**机制和顺序**，它基于词法作用域所确定的层级关系。
词法作用域是单向的：内层可以访问外层，但外层无法访问内层。

18.说一说 JS 变量提升?

JavaScript 的变量提升是指在代码执行前，解释器会先扫描当前作用域，把使用 var 声明的变量和函数声明提升到作用域顶部。
对 var 来说，只提升声明不提升赋值，因此在赋值前访问会得到 undefined；
对函数声明，整个函数会被提升，可以在声明前调用；
而 let 和 const 声明虽然也被记录，但会受到**暂时性死区**的限制，不能在声明前访问。

19.说一说 this 指向（普通函数、箭头函数）？

在 JavaScript 中，this 的指向由**函数调用方式**决定。
普通函数的 this 在运行时绑定，取决于谁调用它；而**箭头函数没有自己的 this**，它会在定义时绑定到外层作用域的 this。
普通函数的 this 可用 call、apply、bind 修改，而箭头函数的 this 无法被改变

20.说一说 call apply bind 的作用和区别?

all、apply、bind 都用于改变函数的 this 指向。
其中，call 和 apply 会立即执行函数，区别在于参数传递方式不同（前者逐个传，后者数组传）；
bind 则返回一个绑定了新 this 的函数，不会立即执行。

三、举个简单例子

```js
function sayHi(age, city) {
  console.log(`我是${this.name}，${age}岁，来自${city}`);
}

const person = { name: "Alice" };

sayHi.call(person, 20, "北京");   // ✅ 立即执行
sayHi.apply(person, [21, "上海"]); // ✅ 立即执行
const boundFn = sayHi.bind(person, 22, "广州");
boundFn(); // ✅ 手动执行
```

1. 对象方法复用：

```js
const obj1 = { name: "Tom" };
const obj2 = { name: "Jerry" };
function show() { console.log(this.name); }
show.call(obj1); // Tom
show.call(obj2); // Jerry
```

21.说一说 js 继承的方法和优缺点？ (难 之后再看看吧)

JavaScript 实现继承的方式主要有原型链继承、构造函数继承、组合继承、原型式继承、寄生式继承和寄生组合继承。
其中，**寄生组合继承**是 ES5 中最理想的方案，它避免了多次调用父构造函数的问题；
在 ES6 中，class 的 extends 本质上就是基于寄生组合继承的语法糖，语法更简洁、可读性更高。

22.说一说 new 会发生什么?

使用 new 调用构造函数时，JavaScript 会先创建一个空对象，并将这个对象的原型指向构造函数的 prototype；
然后以该对象为上下文执行构造函数，将构造函数内部的 this 绑定到这个新对象上；
最后，如果构造函数没有显式返回对象，就返回这个新创建的实例对象。

二、举个简单例子

```js
function Person(name) {
  this.name = name;
}
Person.prototype.say = function() {
  console.log("Hi, I'm " + this.name);
};

const p = new Person("Tom");
p.say(); // 输出：Hi, I'm Tom
```

1. 当调用 p.say() 时，JS 引擎首先在 p 实例本身查找 say 方法，没有找到。
2. 然后，它沿着原型链，通过 p.__proto__ 找到了 Person.prototype 上的 say 方法。
3. 执行 say 方法，此时 say 方法中的 this 动态地指向了调用它的对象，也就是实例 p。
4. 因此，this.name 就解析为 "Tom"，最终输出正确的结果。

以下是涉及到的相关知识：
什么是构造函数？
构造函数在本质上就是一个普通的 JavaScript 函数，但它有两个重要的约定和用途：

1. **用途：** 它的主要目的是作为**创建和初始化一系列相似对象（实例）的模板**。
2. **约定（命名）：** 构造函数通常以**大写字母开头**（例如：Person、Car、Date），这是社区约定俗成的规则，用来区分普通函数和构造函数。
3. **约定（调用）：** 构造函数**必须通过 new 关键字来调用**。

构造函数就是一套**可重复使用的对象创建蓝图**。它定义了新对象应该有哪些**独有的属性**（在函数体内用 this 定义），以及可以访问哪些**共享的方法**（在 prototype 上定义）。你使用 new 关键字来“激活”这个蓝图，从而批量地生产具有相同结构但数据不同的实例。

23.说一说 defer 和 async 区别?

defer 和 async 都能异步加载 JS。defer 会等 HTML 解析完再按顺序执行，async 谁先加载完谁先执行，一般 async 用于无依赖脚本，defer 用于有依赖的业务脚本。

一、背景：为什么需要 defer / async?

在浏览器加载 HTML 时，如果遇到

```html
<script src="xxx.js"></script>
```

浏览器会**停止解析 HTML，先下载并执行脚本。**
这会导致**页面阻塞**、首屏渲染变慢。

为了解决这个问题，就有了 `defer` 和 `async` 两个属性，用来**异步加载脚本。**

📘 **举例：**

```html
<script src="a.js" defer></script>
<script src="b.js" defer></script>
```

✅ `a.js` 和 `b.js` 会异步加载，不阻塞 HTML，
但会在 DOM 解析完成后按顺序执行。

```html
<script src="a.js" async></script>
<script src="b.js" async></script>
```

⚠️ `a.js`、`b.js` 加载完成后立即执行，谁先下载完谁先执行，**不保证顺序。**

24.说一说 promise 是什么与使用方法?

Promise 是 **ES6 引入的一种异步编程解决方案，**
它用来**管理异步任务**（比如请求接口、定时器、文件加载等），
解决传统回调函数（callback）出现的 “回调地狱”问题。
一句话总结：Promise 是一个用来表示异步操作最终完成（或失败）的对象。
它有三种状态：pending 初始状态、fulfilled 成功状态、rejected 失败状态；
用 then() 处理成功结果，用 catch() 处理错误，用 finally() 做收尾；
支持链式调用，常用静态方法有 all、race、allSettled、any。

25.说一说 JS 实现异步的方法?

JS 是单线程的，为了防止主线程阻塞，引入了异步编程。
实现异步的方法主要有五种：
早期用 **回调函数**，后来用 **Promise** 改善回调地狱，再到 **async/await** 提供同步化写法。
还包括基于 **Generator/yield** 的流程控制，以及浏览器原生的 **事件监听机制**。
目前主流是通过 Promise 和 async/await 实现异步操作。

一、为什么需要异步?

JavaScript 是**单线程语言**，同一时间只能做一件事。
如果执行了一个耗时操作（比如网络请求、文件读写、定时器），主线程就会被阻塞，页面就会“卡死”。

所以我们用 **异步编程**，让这些耗时任务**在后台执行**，等结果返回后再通知主线程继续处理。

26.说一说 cookie sessionStorage localStorage 区别?

Cookie、localStorage、sessionStorage 都是浏览器的存储机制。
Cookie 主要用于**与服务器交互**，大小约 4KB，并且会随请求自动发送。
localStorage 和 sessionStorage 是 HTML5 的本地存储方案，不会随请求发送。
区别在于生命周期：localStorage **永久保存**，sessionStorage **页面关闭即清除**。

27.如何实现可过期的 localstorage 数据?

1、惰性删除：存储数据时使用对象类型，添加一个 key 值为当前存储时间，当下一次使用时，判断与当前时间的间隔，如果超过过期时间就清除数据；
2、定时删除：每隔一段时间进行一次删除操作，通过限制删除操作执行的次数和频率，来减少删除操作对 CPU 的长期占用。获取所有设置过期时间的 key 判断是否过期，过期就存储到数组中，遍历数组，每隔 1S（固定时间）删除 5 个（固定个数），直到把数组中的 key 从 localstorage 中全部删除。

```js
// 惰性删除示例
function setExpire(key, value, expire) {
  const data = { value, time: Date.now(), expire };
  localStorage.setItem(key, JSON.stringify(data));
}

function getExpire(key) {
  const data = JSON.parse(localStorage.getItem(key));
  if (!data) return null;
  if (Date.now() - data.time > data.expire) {
    localStorage.removeItem(key);
    return null;
  }
  return data.value;
}
```

28.token 能放在 cookie 中吗?

Token 可以放在 Cookie 中，也可以放在 localStorage 或 sessionStorage 中，关键是看项目的安全策略。
如果放在 Cookie 中，要注意安全配置，比如添加 HttpOnly、Secure 和 SameSite=Strict 属性，这样可以防止 XSS 和 CSRF 攻击。
Cookie 的优点是浏览器会自动携带，适合 SSR 或传统 Web 应用。
但如果配置不当，容易被跨站请求伪造。
而如果放在 localStorage，就需要在请求头（通常是 Authorization）里手动携带 token，这样可以避免 CSRF 攻击，但要注意防范 XSS。
所以综合来说：**可以放在 Cookie 中，但必须正确设置安全属性**，否则会带来安全隐患。

1. Token（令牌）

名词	详细解释
Token（令牌）	**身份验证和授权的凭证。** 在用户成功登录后，服务器会生成一个加密的字符串（Token，通常是 JWT 格式），代表用户的身份信息和权限。后续客户端发送的每一个请求都会携带这个 Token，服务器验证 Token 有效后，才会处理请求。
应用场景	**代替传统的 Session/Cookie 机制**，实现**无状态认证**，尤其适用于前后端分离（SPA）架构和移动应用。

JWT（JSON Web Token）

2. 三种客户端存储位置

名词	存储机制	优点	缺点
Cookie	浏览器在客户端存储数据的小文件。**每次请求时，浏览器会自动在请求头中携带属于该域名的 Cookie。**	**自动发送**：无需手动操作，对开发者友好。适合传统 Web 应用（SSR）。	**容易被攻击**（如 CSRF、XSS，若配置不当）。存储空间小（约 4KB）。
localStorage	HTML5 提供的本地存储 API。数据永久保存在浏览器中，直到用户手动清除。	**存储空间大**（约 5MB）。需要手动携带 Token，**天然免疫 CSRF。**	**容易被 XSS 攻击窃取**（如果网站有 XSS 漏洞）。
sessionStorage	与 `localStorage` 类似，但数据只在当前会话（浏览器标签页或窗口）有效，**关闭会话后数据自动清除。**	适合存储敏感但不需要长期保留的数据。	同 `localStorage`，可能被 XSS 攻击窃取。

3. 安全攻击类型

名词	详细解释	风险/目标
XSS (跨站脚本攻击)	**Cross-Site Scripting。**攻击者向网页中注入恶意脚本（如 `<script>` 标签）。如果 Token 存储在 `localStorage`，恶意脚本可以直接访问并窃取 Token 发送到外部服务器。	**窃取敏感信息**（如 Token、Cookie）。
CSRF (跨站请求伪造)	**Cross-Site Request Forgery。**攻击者诱导用户点击外部链接或访问恶意网站，该网站利用用户在**另一个网站（如银行）**上已登录的身份（即已有的 Cookie）发送非本意的请求。	**利用用户身份发起非法操作**（如转账、修改密码）。

导出到 Google 表格

4. Cookie 的安全配置属性（防止攻击的关键）

为了防御上述攻击，Token 放在 Cookie 中时必须进行严格的安全配置：

属性	作用	防范的攻击	详细解释
HttpOnly	**禁止 JavaScript 访问 Cookie。**	**XSS 攻击**	设置此属性后，浏览器端（如 `document.cookie`）就无法读取或修改这个 Cookie。即使攻击者注入了恶意脚本，也无法窃取 Token。
Secure	**只允许通过 HTTPS 协议发送 Cookie。**	**中间人攻击**	确保 Cookie 在网络传输过程中是加密的。在非安全连接（HTTP）下，该 Cookie 将不会被发送。
SameSite	**控制 Cookie 跨站发送行为。**	**CSRF 攻击**	这是目前防御 CSRF 最有效的手段之一。它指示浏览器何时可以携带 Cookie 进行跨站请求。常见的配置有：`SameSite=Strict`：浏览器只在**同源请求**（即目标网站与当前网站域名完全一致）中发送 Cookie。 `SameSite=Lax`：允许少量例外情况（如顶级导航 GET 请求）。

5. 传输与架构名词

名词	详细解释
请求头（Authorization）	**HTTP 请求头的一部分。**当 Token 存储在 `localStorage` 时，前端需要在每次请求中手动将其添加到请求头的 `Authorization` 字段中（格式通常是 `Authorization: Bearer <Your Token>`）。
SSR（Server-Side Rendering）	**服务器端渲染。**传统 Web 应用架构，页面在服务器上生成 HTML 后发送给浏览器。在这种架构中，Cookie 机制工作得非常自然。
SPA（Single Page Application）	**单页应用。**现代前端架构（如 React/Vue/Angular），页面由客户端 JavaScript 渲染。在这种架构中，手动操作 `localStorage` 或 `sessionStorage` 携带 Token 更常见。

29.axios 的拦截器原理及应用?

axios 的拦截器本质上是基于 Promise 链实现的“中间件机制”。
当我们发起请求时，axios 会把请求拦截器、发送请求函数和响应拦截器串联成一个 Promise 链，按顺序执行。
请求拦截器在请求发送前执行，可以统一添加 token、修改请求头等；
响应拦截器在请求返回后执行，可以统一处理响应数据或错误，比如处理 401 跳转登录。

30.创建 ajax 过程?

创建 AJAX 的过程主要分为五步：
① 首先创建一个 XMLHttpRequest 对象；
② 然后调用 open() 方法，设置请求方式、URL 以及是否异步；
③ 如果需要，可以用 setRequestHeader() 设置请求头；
④ 接着监听 onreadystatechange 事件，通过判断 readyState === 4 && status === 200 来确定请求是否成功；
⑤ 最后调用 send() 方法发送请求。
成功返回后可以通过 xhr.responseText 获取响应数据。

```js
const xhr = new XMLHttpRequest();
xhr.open('GET', '/api/user', true);
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log(xhr.responseText);
  }
};
xhr.send();
```

AJAX 是通过 XMLHttpRequest 对象在浏览器中实现异步请求的技术。
核心步骤：创建对象 → open() → setHeader → 监听状态 → send()。

1、说一下 fetch 请求方式?

fetch() 是浏览器原生提供的用于发起 **HTTP 请求** 的接口，是对 XMLHttpRequest（XHR）的现代替代方案。它基于 **Promise**，语法更简洁、可读性更高，常用于前端与后端交互。

✅ 二、fetch 的基本使用方式

🧩 1. GET 请求（默认）

```js
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) throw new Error('网络响应失败');
    return response.json(); // 解析 JSON 数据
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('请求出错：', error);
  });
```

👉 **说明：**

- `fetch` 默认请求方式是 `GET`
- 需要手动解析返回体（如 `.json()` 或 `.text()`）
- `fetch` 不会自动抛出 404/500 错误，要手动判断 `response.ok`

🧩 2. POST 请求

```js
fetch('https://api.example.com/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    name: 'Tom',
    age: 20
  })
})
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

除了上述两图中的，还可以用 async 和 await

2、说一下有什么方法可以保持前后端实时通信?

实现前后端实时通信的常见方式有五种：

1. 短轮询（Polling）：定时向服务器请求数据，简单但实时性差；
2. 长轮询（Long Polling）：服务器挂起请求直到有新数据返回，实时性较高；
3. SSE（Server-Sent Events）：基于 HTTP 的单向服务器推送；
4. WebSocket：最常用的实时通信方式，支持双向通信、延迟低；
5. WebRTC：用于浏览器间实时音视频通信。

目前主流项目中，WebSocket 是最常见的实时通信方案，比如聊天室、消息推送、在线协作等场景都会用到它。

3、说一下浏览器输入 URL 发生了什么?

当浏览器输入 URL 并回车后，大致会经历以下几个过程：

1. **URL 解析**，判断浏览器输入的是搜索内容还是 URL；
2. **DNS 解析**，将域名解析成服务器 IP；
3. **TCP 三次握手** 建立连接（若是 HTTPS 还会进行 TLS 握手）；
4. **浏览器发送 HTTP 请求**；
5. **服务器处理并返回响应数据**；
6. 通过 TCP 四次挥手释放连接
7. **浏览器解析并渲染页面**，解析 HTML 构建 DOM 树，解析 CSS 构建 CSSOM 树，合成 Render Tree，最后经过布局（Layout）和绘制（Painting）显示到屏幕上。

4、说一下浏览器如何渲染页面的?

首先，解析 **HTML** 构建 **DOM 树**；
然后，解析 **CSS** 构建 **CSSOM 树**；
接着，将两者合并生成 **渲染树（Render Tree）**；
之后，浏览器根据渲染树进行 **布局（Layout）**，计算每个节点的大小和位置；
最后，执行 **绘制（Painting）**，将像素内容绘制到屏幕上。
在这个过程中，如果修改了 DOM 结构或样式，会触发 **回流（Reflow）** 或 **重绘（Repaint）**，从而影响页面性能。

5、说一下重绘、重排区别如何避免?

重排是指当元素的几何位置或尺寸发生变化时，浏览器需要重新计算布局；
重绘是指元素的外观发生变化但布局未变时，浏览器只需重新渲染样式。
重排比重绘的开销更大。
为了减少性能损耗，我们可以合并多次样式修改、避免频繁读取布局属性、使用 class 切换样式、或者通过 transform 和 opacity 实现动画。

🟦 重排（Reflow，也叫回流）

- **定义**：当页面的几何结构发生变化（元素的尺寸、位置、隐藏/显示等），浏览器需要重新计算布局。
- **会触发重排的操作**：
  - 改变元素的宽高、内外边距、边框；
  - 改变元素的显示状态（如 `display: none` → `block`）；
  - 插入或删除 DOM 节点；
  - 调整浏览器窗口大小；
  - 读取布局信息（如 `offsetHeight`、`clientWidth`、`getComputedStyle`）时。

🟩 重绘（Repaint）

- **定义**：当元素的外观改变但**布局未改变**时（例如颜色、背景、文字颜色等），浏览器只需重新绘制外观，而不重新计算布局。
- **触发重绘的操作**：
  - 改变颜色（`color`、`background-color`）；
  - 改变阴影、边框样式等外观属性。

📌 **关系**：
重排一定会引起重绘，但重绘不一定会引起重排。

6、说一下浏览器垃圾回收机制？（GC, Garbage Collection）

浏览器的垃圾回收机制主要是通过**标记清除算法**实现的，
它会从根对象出发，标记所有可达的对象，未被标记的对象就会被当作垃圾回收。
早期还有引用计数算法，每个对象有一个引用计数器，当有一个新的引用指向它，计数 +1；引用断开，计数 -1。
当计数为 0 时，对象就会被回收。但会出现循环引用问题。
常见的内存泄漏场景包括闭包滥用、全局变量、未清除的定时器和事件监听等。
实际开发中我们可以通过手动清除引用或使用 null 来帮助 GC 回收内存。

补充：
为什么需要垃圾回收？
JavaScript 是一种**自动内存管理语言**，开发者不需要手动释放内存。
浏览器会在合适的时机自动清理不再使用的对象，**回收内存空间**，这就是“垃圾回收机制”。

7、说一说事件循环 Event loop，宏任务与微任务?

JavaScript 是单线程的，它通过事件循环机制来处理异步任务。
所有同步任务会先执行，异步任务被分配到任务队列中。
每一轮事件循环中，会先执行宏任务（如 setTimeout），但要看看微任务（如 Promise.then）队列是否为空，只有微任务队列为空，才可以执行。如果微任务队列不为空，就要先执行微任务然后执行。
宏任务执行完后才会进入下一轮循环。
简单来说，执行顺序是：**同步任务 → 微任务 → 宏任务**。

js 在执行代码的时候，是从上到下依次来执行的，在这个过程中，如果遇到异步请求，会交给其他的线程去执行，当需要执行其他线程中的代码时，就会去事件队列中加入要执行的任务，当 js 引擎执行完同步代码后，js 引擎就会从事件队列中取出要执行的代码，进行执行，这就是事件循环。
微任务和宏任务 - 事件队列中维护这两个队列 - 宏任务队列：定时器，Ajax，Dom 监听等 - 微任务队列：Promise 的 then 回调，queueMicrotask()等 - 那么这两个执行队列的顺序是怎样的？ -1、mainscript 先执行 -2、**当执行每一个宏任务时，都要看看微任务队列是否为空，只有微任务队列为空，才可以执行 - 如果微任务队列不为空，就要先执行微任务**

- **举个例子**

```js
console.log('1');

setTimeout(() => console.log('2'), 0);

Promise.resolve().then(() => console.log('3'));

console.log('4');
```

执行顺序:

1. 同步：输出 1、4
2. 微任务（Promise）：输出 3
3. 宏任务（setTimeout）：输出 2

最终结果:

`1 → 4 → 3 → 2`

8、说一说跨域是什么？如何解决跨域问题?

浏览器出于安全考虑实行同源策略，如果协议、域名或端口不同，就会触发跨域问题。
常见的解决方式有：
CORS（在响应头添加 Access-Control-Allow-Origin，主流方案）；
跨域资源共享(Cross-Origin Resource sharing)
JSONP（利用 script 标签，只支持 GET）；
反向代理（由开发服务器或 Nginx 转发请求）；
postMessage（用于不同域 iframe 或窗口通信）。
其中 CORS 是目前最推荐和通用的解决方案。

🧠 一、什么是跨域？

跨域（Cross-Origin）是指浏览器出于同源策略（Same-Origin Policy）**的限制，
在执行某些请求（主要是 AJAX）时，**禁止一个域下的网页去请求另一个域下的资源。**

🔹 什么叫“同源”？

如果两个 URL 的 **协议、域名、端口号** 完全一致，就叫同源。

例如：

地址	是否同源
`https://a.com:443` 与 `https://a.com:443`	✅ 同源
`http://a.com` 与 `https://a.com`	❌ 不同协议
`https://a.com` 与 `https://b.com`	❌ 不同域名
`https://a.com:8080` 与 `https://a.com:443`	❌ 不同端口

9、说一说 vue 钩子函数？

Vue 的钩子函数是指在组件从创建到销毁的不同阶段自动调用的回调函数。

Vue2 钩子	Vue3 对应的钩子函数
beforeCreate	❌ 已合并到 `setup()`
created	❌ 已合并到 `setup()`
beforeMount	`onBeforeMount`
mounted	`onMounted`
beforeUpdate	`onBeforeUpdate`
updated	`onUpdated`
beforeDestroy	`onBeforeUnmount`
destroyed	`onUnmounted`

什么是钩子函数？
Vue 的钩子函数（生命周期函数）指的是：
在组件从创建到销毁的整个过程中，Vue 在不同阶段自动调用的函数。
开发者可以在这些钩子中插入逻辑，比如数据初始化、DOM 操作、接口请求、资源清理等。
简单理解：
📦 **钩子函数 = 生命周期阶段的回调函数**

10、 说一说组件通信的方式？

Vue 组件之间通信的方式主要根据组件关系不同来选择。
父子组件之间通过 props 和 $emit 通信；
兄弟组件可以借助事件总线 eventBus（Vue2）或第三方库 mitt（Vue3）；
跨层级组件可以使用 provide/inject；祖先和子孙

多组件共享数据时使用状态管理工具 Vuex（Vue2）或 Pinia（Vue3）；
此外，还可以通过 ref 或 $parent、$children 获取组件实例来调用方法，但这种方式耦合度高，不推荐在复杂项目中使用。

11、 说一说 computed 和 watch 的区别?

computed 是计算属性，依赖其他属性值，并且有缓存，只有当依赖的值发生了变化之后才会重新计算，不支持异步操作，而 watch 更多的是监听，当监听的值发生了变化，会立即执行回调进行操作，支持异步。
computed 是“算出来的值”，watch 是“观察到变化就干点事”。

12、 说一说 v-if 和 v-show 区别?

在 Vue 中，v-if 和 v-show 都可以控制元素的显示与隐藏。
v-if 是对 DOM 节点进行操作，即直接删除或添加；
v-show 是对 DOM 节点样式进行操作，相当于是否设置 display: none
v-if：适合**条件不常变化**的场景
v-show：适合**频繁切换显示状态**的场景

13、 说一说 vue 的 keep-alive ?

keep-alive 是 Vue 的内置抽象组件，用来**缓存被包裹的组件实例**，
可以避免组件重复渲染，从而提升性能。
当组件被缓存后，切换时不会重新创建和销毁实例，而是通过 activated 和 deactivated 生命周期来管理组件的激活与停用状态。
它常用于**路由切换**或**动态组件切换**的场景，比如从详情页返回列表页时保持原有状态。
此外，keep-alive 还可以通过 include、exclude 控制缓存范围，通过 max 限制缓存数量。

补充：
<keep-alive> 是 Vue 的**内置抽象组件**，
用于缓存被包裹的动态组件或路由组件，从而避免频繁销毁与重建，提高性能。
简单理解：
当你切换组件时，被 <keep-alive> 包裹的组件不会被卸载，而是被“暂存”在内存中，下次切换回来时会直接复用之前的实例。

14、 说一说 Vue 中 $nextTick 作用与原理?

$nextTick 是 Vue 提供的一个异步方法，用于在 **DOM 更新完成后** 执行回调。
因为 Vue 在更新数据时会进行 **异步批量更新**，数据改变后，DOM 不会立即更新。
$nextTick 可以让我们在 DOM 更新完成后获取到最新的视图状态。
其内部原理是通过 **Promise.then** 或 **MutationObserver** 实现的微任务队列，
在下一次事件循环中执行回调函数。
常用于在数据更新后立即操作 DOM，或者确保页面渲染完成后再执行逻辑。
$nextTick：等 DOM 更新完再执行回调；
原理：基于微任务（Promise.then / MutationObserver）实现的异步队列。

Vue 3 代码示例（使用 `Composition API`）

代码段

```
<template>
  <div>
    <button @click="editContent">编辑内容</button>

    <input v-if="visible" ref="inputRef" type="text" placeholder="请输入内容">
  </div>
</template>

<script setup>
import { ref, nextTick } from 'vue';

// 状态：控制 input 的显示/隐藏
const visible = ref(false);

// 模板引用：用于直接访问 input 元素
const inputRef = ref(null);
```

代码段

```
const editContent = async () => {
  // 步骤 A: 修改数据，控制文本框显示
  visible.value = true;

  // 步骤 B: 尝试获取焦点（错误的位置!）
  // inputRef.value.focus();
  // 这一行会报错，因为 visible = true 之后 DOM 还没来得及更新

  // 步骤 C: 使用 nextTick 确保在 DOM 更新后再执行操作
  await nextTick(() => {
    // 此时 DOM 已经更新完成，input 元素已经存在于页面上
    if (inputRef.value) {
      inputRef.value.focus();
      console.log('DOM 已更新，并成功获取焦点！');
    }
  });
};
</script>
```

场景：点击按钮修改内容并自动聚焦
假设我们有一个文本框，初始时是隐藏的。我们希望点击按钮后：
文本框出现（v-if 条件变为 true）。
同时，文本框自动获得焦点（focus()）

15、 说一说 Vue 列表为什么加 key?

在 Vue 中，key 是虚拟 DOM 节点的唯一标识，
Vue 通过 key 来精确判断哪些节点需要复用、哪些需要更新或销毁。
在列表渲染时，如果不加 key，Vue 会采用“就地复用”策略，
导致节点状态错误、列表渲染混乱，比如输入框内容错位、动画异常等问题。
因此，为每个节点设置唯一的 key（如 item.id）可以保证 diff 算法准确高效地更新视图。

key 是虚拟 DOM 的唯一标识，用来高效、准确地复用节点；
不加 key 会导致就地复用，出现渲染错误或状态错乱
<li v-for="item in list" :key="item.id">{{ item.name }}
Diff 算法是一种用于**计算新旧数据集合之间差异**的算法。

16、 说一说 vue-router 实现懒加载的方法?

在 Vue Router 中，路由懒加载是一种按需加载组件的方式，
可以通过动态 import() 实现，当路由被访问时才去加载对应的组件。
这样可以显著减少首屏加载体积，提高页面加载速度。
() => import('...')

```
const routes = [
  { path: '/home', component: () => import('@/views/Home.vue') }
]
```

17、 说一说 HashRouter 和 HistoryRouter 的区别和原理?

前端路由主要有两种实现方式：HashRouter 和 HistoryRouter。
Hash 模式通过监听 hashchange 事件实现路由切换，URL 中会带 #，刷新不会 404，兼容性好。
而 History 模式基于 HTML5 的 history.pushState 和 onpopstate，URL 更简洁，但需要服务端配置支持，否则刷新会 404。
简单来说，Hash 模式前端可独立运行，History 模式更优雅但需后端配合。

Hash 模式靠 # + hashchange 实现，History 模式靠 pushState + popstate 实现，
前者无需服务端支持，后者路径更自然但需后端配置。

18、 说一说 Vuex 是什么，每个属性是干嘛的，如何使用 ？

Vuex 是 Vue 的状态管理工具，用于集中管理组件之间共享的数据，实现可预测的数据流。
它包含五个核心属性：

- **state** 存储数据；
- **getters** 相当于计算属性；
- **mutations** 用于提交更新数据的方法；
- **actions** 用于处理异步操作并提交 mutations；
- **modules** 实现模块化管理。

在组件中通过 commit 提交同步操作、dispatch 触发异步操作。

```
vue

<script setup>
import { useStore } from 'vuex'
import { computed } from 'vue'

const store = useStore()

// 获取状态
const count = computed(() => store.state.count)
const double = computed(() => store.getters.doubleCount)

// 修改状态
const add = () => store.commit('increment')
const getUser = () => store.dispatch('fetchUser')
</script>

<template>
  <div>
    <p>Count: {{ count }} (Double: {{ double }})</p>
    <button @click="add">+1</button>
    <button @click="getUser">获取用户</button>
  </div>
</template>
```

```
import { createStore } from 'vuex'

export default createStore({
  state: {
    count: 0,
    user: null
  },
  getters: {
    doubleCount: state => state.count * 2
  },
  mutations: {
    increment(state) {
      state.count++
    },
    setUser(state, user) {
      state.user = user
    }
  },
  actions: {
    async fetchUser({ commit }) {
      const res = await fetch('/api/user')
      const data = await res.json()
      commit('setUser', data)
    }
  },
  modules: {
    // 例如 user 模块、cart 模块
  }
})
```

总结流程图

这个流程形成一个闭环，确保状态修改是单向且可预测的：

$$\text{View} \xrightarrow{\text{dispatch / commit}} \begin{cases} \text{Actions (异步/逻辑)} \xrightarrow{\text{commit}} \text{Mutations} \\ \text{Mutations (同步)} \end{cases} \xrightarrow{\text{Mutations}} \text{State} \xrightarrow{\text{响应式}}$$

面试精简总结：

> “Vuex 的单向数据流是严格的，它的流向是：视图通过 `dispatch` 触发 `Actions` 处理异步逻辑，或通过 `commit` 直接触发 `Mutations`。`Mutations` 是唯一能够 **同步** 修改 `State` 的地方。`State` 更新后，由于其响应式特性，最终会驱动 **视图** 自动更新。这个闭环保证了状态变更的可追踪性和可预测性。”

补充个 pinia：
Pinia 是 Vue3 官方推荐的状态管理库，可以看作 Vuex 的轻量替代。
它通过 defineStore 定义独立的状态模块，包含 state、getters、actions 三个核心属性。
与 Vuex 不同，Pinia 去掉了 mutations，在 actions 中即可直接修改状态，
同时支持组合式 API 和更好的 TypeScript 推断，使用更简单、性能更好。
典型使用是通过 useStore 获取 store 实例，再直接访问或调用方法。

四、Vuex vs Pinia 对比总结

对比点	Vuex	Pinia
定义方式	`createStore`	`defineStore`
状态管理结构	state、getters、mutations、actions	state、getters、actions（更简洁）
修改状态	必须通过 mutations	可直接在 actions 里改
模块化	modules	多 store，自然模块化
TypeScript 支持	一般	✅ 原生支持
语法风格	选项式 API	支持组合式 API
开发体验	模板化多、样板代码多	简洁易读、调试方便

19、 说一说 Vue2.0 双向绑定的原理与缺陷？（也就是响应式）

Vue2 的双向绑定是通过 Object.defineProperty() 实现的，
在数据初始化时会对每个属性进行劫持，并通过 getter 收集依赖、setter 派发更新。
当数据变化时，会通知 Watcher 更新视图，从而实现“数据驱动视图”。
它的主要缺陷是无法监听对象新增/删除属性、无法检测数组索引修改、
深层对象监听性能较差。Vue3 通过 Proxy 实现响应式系统，彻底解决了这些问题。

Vue2 用 defineProperty 劫持属性实现响应式，缺点是监听不全、性能较差；
Vue3 改用 Proxy，更强更快。

20、 说一说 Vue3.0 实现数据双向绑定的方法 ？

Vue3 的双向绑定基于 ES6 的 Proxy 实现，通过拦截对象的读取和设置操作来实现数据劫持。在读取属性时进行依赖收集，在修改属性时触发派发更新，从而实现数据和视图的同步。
相比 Vue2 使用 Object.defineProperty()，Proxy 可以监听对象新增、删除属性以及数组索引变化，并且支持按需监听和更好的性能。
Vue3 的响应式系统核心模块是 @vue/reactivity，其中包括 reactive、ref、computed 等 API

Vue3 的数据双向绑定是通过 **Proxy** 实现的。
它会在访问属性时拦截 get，进行**依赖收集**；
在修改属性时拦截 set，触发**视图更新**。
相比 Vue2 的 Object.defineProperty，Proxy 可以直接监听整个对象，
支持**新增属性、删除属性、数组下标修改**等操作，性能更好，也更灵活。

防抖（Debounce）
核心思想：等“静默期”
防抖的核心是：**在事件被触发后，延迟一段时间再执行函数。如果在延迟期内事件又被触发了，则重新开始计时。**
例子：实时搜索建议
节流（Throttle）
核心思想：限制频率
节流的核心是：**在指定时间周期内，函数最多只执行一次。**
例子：比如点击生成图片，5 秒冷却

21、 说一说前端性能优化手段?

前端性能优化可以分为加载、渲染和运行三个阶段：
加载阶段主要是**减少体积与请求次数**，比如代码压缩、图片优化、资源懒加载、CDN 加速；
渲染阶段关注 **DOM 更新和首屏体验**，比如减少重绘重排、骨架屏、SSR；
运行阶段则关注**用户交互和内存性能**，如节流防抖、Web Worker、虚拟列表；
同时通过缓存、HTTP/2 和构建优化进一步提升整体性能。

前端性能优化三件事：**加载快、渲染快、运行稳。**

22、 说一说性能优化有哪些性能指标，如何量化?

前端性能指标可分为三类：
加载性能（首屏时间、TTFB、DNS 查询时间、页面完全加载时间、可交互时间）；
渲染性能（重排、重绘、FPS）；
运行性能（长任务、内存占用、事件响应延迟）。
可通过 Chrome DevTools、Lighthouse、WebPageTest 等工具量化，并结合监控埋点进行持续优化。
前端性能指标 = **加载快（TTFB/FCP/TTI） + 渲染流畅（Reflow/Repaint/FPS） + 运行稳（长任务/内存/响应）**。

23、 说一说服务端渲染?

服务端渲染（SSR）是指在服务器端将前端框架的组件渲染成完整 HTML，再返回给浏览器显示。
优点是首屏快、SEO （搜索引擎优化）友好、白屏时间短；缺点是服务器压力大、开发复杂、缓存和更新处理难。
典型流程是：浏览器发请求 → 服务器生成 HTML → 浏览器渲染 → 前端 JS 激活交互（Hydration）

SSR（Server-Side Rendering） 是指在服务器端将 Vue / React 等前端框架的组件渲染成完整的 HTML 字符串，然后发送给浏览器显示。
传统 SPA（客户端渲染）：浏览器拿到空 HTML + JS 文件 → JS 解析 → 渲染页面
SSR：服务器先生成完整 HTML → 浏览器直接渲染 → JS 再接管交互（Hydration）

24、 XSS 攻击是什么?

XSS 是跨站脚本攻击，攻击者在网页中注入恶意 JS 脚本，当用户访问页面时执行，可能窃取 cookie、localStorage 或篡改页面。
类型包括 **反射型、存储型、DOM 型**。
防护方法：输入转义、模板自动转义、HTTP Only cookie、CSP、避免直接使用 innerHTML。

XSS（Cross-Site Scripting）跨站脚本攻击 是指攻击者在网页中注入恶意脚本，当其他用户浏览该页面时，这些脚本就会在用户浏览器中执行，从而窃取信息、劫持会话或篡改页面内容。
通俗理解：

- 用户在浏览网页时被攻击者植入的 JS 代码偷偷执行
- 典型目标：cookie、localStorage、DOM、页面行为

25、 CSRF 攻击是什么?

CSRF（跨站请求伪造）是攻击者利用用户已登录状态，诱导浏览器在用户不知情的情况下向受信任网站发送恶意请求，从而执行敏感操作。
常见防护措施有：

1. 服务端生成 **CSRF Token**，请求必须携带验证；
2. 设置 Cookie 的 **SameSite** 属性，禁止第三方请求携带 Cookie；
3. 验证请求来源 **Referer/Origin**；
4. 避免用 GET 执行敏感操作，重要操作可要求二次确认（密码、验证码等）。

CSRF（Cross-Site Request Forgery）跨站请求伪造
是指攻击者引导已登录用户，在不知情的情况下，向受信任的网站发送恶意请求，执行一些用户本不想执行的操作。
通俗理解👇：
利用用户已登录状态，让浏览器“帮用户”做坏事（比如转账、改密码）

⚙️ 二、CSRF 攻击原理

1. 用户登录受信任网站 A（如银行网站），浏览器保存登录的 Cookie
2. 攻击者诱导用户访问恶意网站 B
3. 网站 B 向网站 A 发起伪造请求（如转账请求）
4. 浏览器会自动带上网站 A 的 Cookie
5. 网站 A 误以为是用户本人操作，执行请求

26、 说一下 Diff 算法?

Diff 算法是虚拟 DOM 的核心更新算法，用于比较新旧虚拟 DOM 树，找出最小更新范围，只更新必要的真实 DOM，从而提升性能。

核心策略是**同层比对、类型判断、key 唯一标识、最小化移动**，Vue 3 还利用最长递增子序列优化列表节点移动。

面试讲时可以补一句：**避免频繁操作真实 DOM 是 Diff 算法设计的核心目的**。