2025.5-1

1.说一说样式优先级的规则是什么

优先级：也叫权重，当一个标签使用了多种选择器时，基于不同种类的选择器的匹配规则

规则：选择器优先级高的样式生效

公式：通配符选择器 < 标签选择器 < 类选择器 < id选择器 < 行内样式 < !important

（选中的标签范围越大，优先级越低）

优先级 - 叠加运算规则：

叠加运算：如果是复合选择器，则需要权重叠加计算（每一级之间不存在进位）

（行内样式，id选择器个数，类选择器个数，标签选择器个数）

规则：

- 从左向右依次比较个数，同一级个数多的优先级高，如果个数相同，则向后比较
- !important 权重最高
- 继承权重最低

例1：

```
<div id="box1" class="c1">
    <div id="box2" class="c2">
        <div id="box3" class="c3">
            这行文字是橙色的
        </div>
    </div>
</div>

/* (0,0,2,1) */
.c1 .c2 div {
    color: blue;
}
/* (0,1,0,1) */
div #box3 {
    color: green;
}
/* (0,1,1,0) */
#box1 .c3 {
    color: orange;
}
```

例2：

```
<div class="father">
    <p class="son">
        这行文字是红色的
    </p>
</div>

div p {
    color: red;
}
/* 继承权重最低 */
.father {
    color: blue;
}
```

例3：

```
<div id="father" class="c1">
    <p id="son" class="c2">
        这行文字是蓝色的
    </p>
</div>

/* (0,2,0,0) */
#father #son {
    color: blue;
}
/* (0,1,1,1) */
#father p .c2 {
    color: black;
}
/* (0,0,2,2) */
div .c1 p .c2 {
    color: red;
}
/* !important权重最高，但是样式设置给了父级，继承权重最低 */
#father {
    color: green !important;
}
/* 样式设置给了父级，继承权重最低 */
div #father .c1 {
    color: yellow;
}
```

2.说一说css尺寸设置的单位

常见的css尺寸设置单位包括像素（px）、百分比（%）、em、rem、vw / vh

- 像素（px）：

简单理解像素就是屏幕上最小的一个显示单元，就相当于画画时那一个小点点，像素分为两种类型：css 像素 和 物理像素

  - css像素：专门为开发者提供，在css中使用的一个抽象单位，就是我们在写样式的时候，比如 `width:100px` 中的 px
  - 物理像素：物理像素指的是屏幕上真实存在的小方块数量，一个css像素最终呈现在屏幕上可能是多个物理像素，也就是 DPR

- 百分比（%）：相对于父元素的尺寸
- em 和 rem：
  - em：相对于父元素的字体大小 1em = 父元素的 font-size
  - rem：相对于根元素的字体大小（html元素的 font-size） 1rem = html的font-size
- vw / vh：
  - vw：相对视口宽度 1vw = 视口宽度的1%
  - vh：相对视口高度 1vh = 视口高度的1%

> 视口：显示HTML网页的区域，用来约束HTMl尺寸

3.说一说BFC

BFC就是块格式化上下文（Block Formatting Context），简单来说就是网页中的一个独立布局环境（BFC是一个区域）。在这个环境里面的元素布局不会影响到外面的元素，外面的元素也不会影响到它。常见的触发方式是 `overflow: hidden`

创建BFC的条件：

- 和 根元素
- **元素设置浮动：float 除了 none 以外的值**
- 元素定位：position 设为 absolute 或 fixed（这两个会脱标）
- display 值为：flow-root、flex、inline-block
- overflow 值为：**hidden**、auto

BFC的特点：

- 垂直方向上，自上而下排列，和文档流的排列方式一致
- 触发BFC只会形成一个"独立布局环境"，不会改变元素的显示模式
- 在BFC中上下相邻的两个容器的margin会重叠（外边距合并问题的原因）
- BFC区域不会与浮动的容器（也是一个BFC区域）发生重叠
- BFC是独立的容器，容器内部元素不会影响外部元素
- 计算BFC高度时，子元素的margin也参与计算（解决外边距塌陷问题）

> 也就是元素触发BFC后，其中的浮动子元素也被包含在内

BFC的作用：

- **解决外边距合并问题：**给其中一个盒子设置 overflow: hidden ，创建BFC且包裹了该盒子，与另外一个盒子隔离，解决了合并问题
- **清除浮动：**给父元素设置 overflow: hidden ，其中的浮动子元素也被包含在内，父元素的高度就由浮动子元素撑开
- **创建自适应两栏布局：**左边宽度固定，右边宽度自适应

```
<div class="container">
    <div class="left"></div>
    <div class="right"></div>
</div>

.container {
    height: 100px;
    background-color: skyblue;
}
.container .left {
    /* 这里不能设置 overflow 来触发BFC，不能在一行显示（块级）*/
    /* 设置浮动（行内块）*/
    float: left;
    width: 100px;
    height: 100px;
    background-color: red;
}
/* 其实这样设置右边还是块级 */
/* 能一行显示是因为css的一个容错式排版行为*/
/* 浮动脱标 + BFC避免被包住 + 空间足够 => 块级元素可以排在浮动元素旁边 */
.container .right {
    /* 此时 .right 就会自动躲开 .left 浮动占的位置 */
    /* 从而实现在一行内右边自适应 */
    /* 创建BFC，BFC区域不会与浮动元素重叠 */
    overflow: hidden;
    height: 100px;
    background-color: green;
}
```

4.两栏布局的实现

一般两栏布局指的是**左侧一栏宽度固定，右侧一栏宽度自适应**，两栏布局的具体实现：

- **flex布局（推荐）**

```
<div class="container">
  <div class="left">左侧固定</div>
  <div class="right">右侧自适应</div>
</div>

.container {
      display: flex;
```

```
    height: 100px;
}

.left {
    height: 100px;
    width: 200px;
    background: skyblue;
}

.right {
    flex: 1;
    height: 100px;
    background: orange;
}
```

- float+ margin

```
<div class="container">
  <div class="left">左侧固定</div>
  <div class="right">右侧自适应</div>
</div>

.left {
  float: left;
  width: 200px;
  height: 100px;
  background: skyblue;
}
.right {
  margin-left: 200px;
  height: 100px;
  background: orange;
}
```

5.三栏布局的实现

三栏布局一般指的是页面中一共有三栏，**左右两栏宽度固定，中间自适应的布局**，三栏布局的具体实现：

- **flex布局（推荐）**

```
<div class="container">
  <div class="left">左侧固定</div>
  <div class="center">中间自适应</div>
  <div class="right">右侧固定</div>
</div>

.container {
    display: flex;
    height: 100px;
```

```
}

.left {
    width: 200px;
    height: 100px;
    background: skyblue;
}

.right {
    width: 200px;
    height: 100px;
    background: orange;
}

.center {
    flex: 1;
    height: 100px;
    background: red;
}
```

- 圣杯布局

```
<div class="container">
  <div class="center">中间自适应</div>
  <div class="left">左侧固定</div>
  <div class="right">右侧固定</div>
</div>

.container {
    height: 100px;
    padding-left: 100px;
    padding-right: 200px;
}

.center {
    float: left;
    width: 100%;
    height: 100px;
    background: lightgreen;
}

.left {
    position: relative;
    left: -100px;
    float: left;
    /* %是相对于当前元素的“包含块”（一般是父元素）的宽度 */
    margin-left: -100%;
    width: 100px;
    height: 100px;
    background: tomato;
}
```

```
.right {
    position: relative;
    right: -200px;
    float: left;
    margin-left: -200px;
    width: 200px;
    height: 100px;
    background: gold;
}
```

- 双飞翼布局

```
<div class="container">
  <div class="center-wrap">
    <div class="center">中间自适应</div>
  </div>
  <div class="left">左侧固定宽度</div>
  <div class="right">右侧固定宽度</div>
</div>

.center-wrap {
    float: left;
    width: 100%;
    height: 100px;
    background-color: orange;
}

.center {
    margin-left: 200px;
    margin-right: 100px;
    height: 100px;
    background-color: pink;
}

.left {
    float: left;
    margin-left: -100%;
    width: 200px;
    height: 100px;
    background-color: skyblue;
}

.right {
    float: left;
    margin-left: -100px;
    width: 100px;
    height: 100px;
    background-color: green;
}
```

> 圣杯布局和双飞翼布局核心都是利用浮动会触发BFC，使得左中右三个浮动块（行内显示模式）按照HTML顺序依次从左往右排列，然后在利用 margin、padding 和 边偏移微调实现三栏布局

6.JS 的数据类型有哪些？它们的区别是什么？

JS数据类型可以分为简单数据类型和复杂数据类型两大类

- 简单数据类型

  存储在**栈内存**中

  Number、String、Boolean、undefined、null、Symbol、BigInt

  > Symbol和BigInt 都是 ES6 新增，前者是用于创建一些独一无二的值，后者是用于表示超出 Number 能表示范围的极大整数

- 复杂数据类型（引用数据类型）

  通过 new 关键字创建的对象，存储在**堆内存**中，在**栈内存**中存储了地址（指针）

  Object、Array、Date（日期对象）、function

区别：

- **存储位置**不同，简单数据类型存储在栈内存中，复杂数据类型存储在堆内存中，在栈内存中存储了地址
- **比较方式**不同，简单数据类型是比较值，复杂数据类型是比较地址
- **拷贝效果**不同，简单数据类型只拷贝值，复杂数据类型拷贝地址

```
let a = 1
a = 2
```

第一步： `let a = 1`

- JS 引擎在执行时，在**栈内存中开辟一个空间给变量 a。**
- 把值 1 存进去（ 1 是简单数据类型类型，直接存在栈里）。
- 举个类比：a 是个标签，贴在一个小抽屉（栈中开辟的空间）上，里面放着值 1 。

第二步： `a = 2`

- 把**变量 a 指向的值（贴着 a 标签抽屉里面的值）修改成 2。**
- 还是在原来那块栈的空间，直接把里面的值从 1 改成 2 。
- 所以，**a 的栈地址没变，变的是它的值。**

```
let obj = { name: 'GPT' }
obj = { name: 'New' }
```

第一步： `let obj = { name: 'GPT' }`

- JS 引擎在执行时，在**栈内存中开辟一个空间给变量 obj。**

> - 把 { name: 'GPT' } 在堆中的地址存进去（{ name: 'GPT' } 是复杂数据类型类型）。
> - 举个类比：obj 是个标签，贴在一个小抽屉（栈中开辟的空间）上，里面放着 { name: 'GPT' } 在堆中地址。
>
> **第二步：** `obj = { name: 'New' }`
>
> - 把**贴着 obj 标签抽屉里面的地址修改成 { name: 'New' } 在堆中的地址 。**
> - 还是在原来那块栈的空间只不过里面的地址变了，变成新对象在堆中的地址了

7.说一下 undefined 和 null 的区别？如何让一个属性变为 null?

undefined 和 null 都是简单数据类型，这两个简单数据类型分别只有一个值，就是undefined 和 null

> 当对这两种类型使用 typeof 进行判断时，typeof null 会返回 “object”，这是一个历史遗留的问题（第一版本 JS 中的一个bug）

undefined 代表的含义是未定义，null 代表的含义是空值或者空对象，，一般情况下 undefined 是变量未初始化时的默认值，而 null 是开发者主动赋值表示空值

```
// 如何让一个属性变为 null?
obj.name = null // 直接赋值
const user = {
  name: null, // 初始化时设置
  age: 18
}
```

8.说一说JavaScript有几种方法判断变量的类型?

- typeof

  数组、对象、null用 `typeof` 判断都会被判断为 object，其他判断都正确

  ```
  console.log(typeof 2);               // number
  console.log(typeof true);            // boolean
  console.log(typeof 'str');           // string
  console.log(typeof []);              // object
  console.log(typeof function(){});    // function
  console.log(typeof {});              // object
  console.log(typeof undefined);       // undefined
  console.log(typeof null);            // object
  ```

- instanceof

  `instanceof` 只能正确判断复杂数据类型，而不能判断简单数据类型，其内部运行机制是判断在该复杂数据类型的原型链中能否找到该构造函数的 `prototype` 属性

  > instanceof 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上

```
console.log(2 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);                // false

console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

- Object.prototype.toString.call()**（最准确）**

```
Object.prototype.toString.call(42);      // "[object Number]"
Object.prototype.toString.call([]);      // "[object Array]"
Object.prototype.toString.call(null);    // "[object Null]"
Object.prototype.toString.call(undefined);// "[object Undefined]"
```

- constructor（不可靠）

  constructor 本质是一个原型上的属性，而这个属性可以被修改，导致我们用它来判断对象类型的时候不一定准确

```
// 自动包装为 => new Number(2).__proto__.constructor === Number
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
```

  下面这种就会出现问题：

```
function Person() {}
Person.prototype = {}  // 修改原型对象

const p = new Person()

// p.__proto__.constructor
console.log(p.constructor === Person) // false
console.log(p.constructor === Object) // true
```

9.说一说对原型和原型链的理解?

- **原型：**JS 规定，每一个构造函数都有一个 prototype 属性，指向另一个对象，所以我们也称为原型对象

- 构造函数通过原型分配的函数是所有对象 **共享的**，我们可以把那些不变的方法，直接定义在 prototype 对象上，这样所有对象的实例就可以共享这些方法
- 每个原型对象里面都有个 constructor 属性，该属性指向该原型对象的构造函数

> **对象原型：**
>
> - 对象都会有一个属性 `__proto__` 指向构造函数的 prototype 原型对象，之所以我们的对象可以使用构造函数 prototype 原型对象的属性和方法，就是因为对象有 `__proto__` 原型的存在
> - `__proto__` 是 JS 非标准属性，`[[prototype]]` 和 `__proto__` 意义相同

- **原型链：原型链其实是一个查找规则**，当访问一个对象的属性（包括方法）时，首先查找这个对象自身有没有该属性，如果没有就查找它的原型（也就是 `__proto__` 指向的 prototype 原型对象），如果还没有就查找原型对象的原型，依此类推一直找到 Object 为止（null）。`__proto__` 对象原型的意义就在于为对象成员查找机制提供一个方向，或者说一条路线

创建对象的三种方式：

- 利用对象字面量创建对象

```
const obj = {
  name: '佩奇',
  age: 18
}
```

- 利用 new Object() 创建对象

```
const obj = new Object({name: '佩奇'})
obj.age = 18
```

- 利用构造函数创建对象

> 构造函数是一种特殊的函数，主要用来初始化对象

```
function Pig (name, age) {
  this.name = name
  this.age = age
}
const pq = new Pig('佩奇', 18)
```

创建数组的方式：

- 利用数组字面量创建数组

```
const arr = [1, 2, 3]
```

- 利用 new Array() 创建数组

```
const arr = new Array()
```

- 利用构造函数创建数组（创建的是**伪数组**）

```
function MyArray() {
  this.length = 0
  this.push = function (item) {
    this[this.length] = item
    this.length++
  }
}

const arr = new MyArray();
arr.push(10)
arr.push(20)

console.log(arr)
// { '0': 10, '1': 20, length: 2, push: [Function] }
```

10.说一下伪数组和数组的区别?

伪数组是看起来像数组，但**不具备数组所有特性**的对象，两者的主要区别是伪数组没有继承 `Array.prototype` 不能直接使用数组方法

```
// 伪数组
const arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
}

console.log(arrayLike[0])      // 'a'
console.log(arrayLike.length)  // 3  有长度
console.log(arrayLike.map)     // undefined  不能使用数组的方法
```

> arguments是一个伪数组，而剩余参数是一个真数组，两者都只存在于函数中

怎么把伪数组变成真数组?

```
// 方法1 Array.from()
const realArray = Array.from(arrayLike)
```

```
// 方法2 扩展运算符
const realArray = [...arrayLike]
```

11.说一说 map 和 forEach 的区别?

```
const arr = [1, 2, 3, 4, 5]

// map
const newArr = arr.map((item, index) => {
  console.log(item)
  console.log(index)
  return item * 2
})

// forEach
arr.forEach((item, index) => {
  console.log(item)
  console.log(index)
})
```

map 是对每个元素进行处理之后返回新数组

forEach 是对每个元素进行操作，不返回新数组

12.说一说 ES6 中的箭头函数?

- 箭头函数相比普通函数书写更简洁，并且不会创建自己的 `this` ，它只会沿用自己作用域链上一层的 `this` 。
- 箭头函数没有 arguments 动态参数，但是有剩余参数（...others）
- 箭头函数不能用作构造函数
- 箭头函数没有 prototype 属性（构造函数才具有 prototype 属性）
- call()、apply()、bind()等方法不能改变箭头函数中this的指向

13.扩展运算符（...）用过吗，在什么场景?

> 也叫事件扩展符
>
> **与剩余参数的区别：**
>
> - 剩余参数：只存在于函数中，在函数参数中使用，得到真数组
> - 展开运算符：随时可用，可用于展开对象或数组

- 拷贝对象或数组（浅拷贝）

```
const obj = {
  name: 'zs',
  age: 18
}
```

```
const arr = [1, 2, 3, 4]

const newObj = {...obj} // {name: 'zs', age: 18}
const newArr = [...arr] // [1, 2, 3, 4]
```

- 合并对象或数组

```
const obj1 = {
  name: 'zs',
  age: 18
}
const obj2 = {
  hobby: '唱跳rap',
  sayHi: () => {
    console.log('hi~')
  }
}
const arr1 = [1, 2, 3, 4]
const arr2 = [11, 22, 33, 44]

const newObj = {...obj1, ...obj2}
// {name: 'zs', age: 18, hobby: '唱跳rap', sayHi: () => {console.log('hi~')}}
const newArr = [...arr1, ...arr2]
// [1, 2, 3, 4, 11, 22, 33, 44]
```

- 配合 Math数学对象 求最值

  数组自身没有求最值的方法，`Math.max()` 需要的参数只能是 `Math.max(1, 2, 3)` 这种形式，不能直接传数组，而展开运算符可以将数组展开 `...arr === 1, 2, 3` 虽然在控制台打印 `...arr` 为 `1 2 3`

```
const arr = [1, 2, 3]
console.log(Math.max(...arr)) // 3
console.log(Math.min(...arr)) // 1
```

14.说一下你对闭包的理解?

简单理解，**闭包 = 内层函数 + 外层函数的变量**

只要函数中用到了函数外的变量，并且这个函数在原作用域外被执行，就形成了闭包

闭包可以封闭数据，实现数据私有，函数外部也可以访问函数内部的变量，但是闭包容易造成内存泄露

```
function count () {
  // 封闭数据实现数据私有，函数外无法随便修改数据
  let i = 0
  function fn () {
    i++
```

```
    console.log('函数被调用了${i}次')
  }
  return fn
}
const fun = count()
// 相当于调用了 fn 间接的访问了函数内部的变量
fun()
```

15.说一说call、apply、bind的作用和区别?

- **call()**

  使用 call() 方法调用函数，同时指定被调用函数中 this 的值

```
const obj = {
    name: 'pink'
}
function fn(x, y){
  console.log(this)   // 指向 obj
  console.(x + y)     // 3
}
// 返回值就是函数的返回值，因为它就是调用函数
fn.call(obj, 1, 2)
```

- **apply()**

  使用 apply() 方法调用函数，同时指定被调用函数中 this 的值

```
const arr = [1, 2]
function fn(x, y){
  console.log(this)   // null
  console.(x + y)     // 3
}
// 返回值就是函数的返回值，因为它就是调用函数
fn.apply(null, arr)
```

  求数组最值：

```
const arr = [3, 4, 5, 6]
console.log(Math.max.apply(null, arr))
console.log(Math.max(...arr))
```

- **bind()**

  bind() 方法不会调用函数，但是能改变函数内部 this 指向

```
const obj = {
    name: 'pink'
}
function fn(){
  console.log(this)
}
// 返回值是已经改变 this 指向的新函数
const fun = fn.bind(obj)
fun() // obj
```

因此当我们只是想改变 this 指向，并且不想调用这个函数的时候，可以使用 bind()，比如改变定时器内部的 this 指向

```
const btn = document.querySelector('button')
btn.addEventListener('click', function(){
  // 禁用按钮
  this.disabled = true
  setTimeout(function(){
    // 将 this 由原来的 window 改为 btn
    this.disabled = false
  }.bind(btn), 2000)
})
```

- **区别**
  - call() 和 apply() 会调用函数，并且改变函数内部的 this 指向
  - call() 和 apply() 传递的参数不一样，call() 传递 1, 2... 正常形式，apply()必须以数组的形式
  - bind() 不会调用函数，可以改变函数内部的 this 指向

16.说一说js继承的方法和优缺点?

- **原型链继承**

  不会出现问题的写法：

```
function Person(){
  this.eyes = 2
  this.head = 1
}
// 男人
function Man(){}
Man.prototype = new Person()
Man.prototype.contructor = Man
Man.prototype.smoking = function(){}
const pink = new Man()

// 女人
function Woman(){}
```

```
Woman.prototype = new Person()
Woman.prototype.contructor = Woman
Woman.prototype.body = function(){}
const pink = new Woman()
// 这种写法不会造成引用数据类型的污染，因为 Man 和 Woman 两个构造函数的 prototype 是独立的
```

优点：

- 代码简单，继承了父类实例的属性和方法

缺点：

- 引用数据类型会被所有实例共享（数组或对象）

```
function Parent() {
  this.name = 'parent';
  this.colors = ['red', 'green'];
}
Parent.prototype.sayName = function () {
  console.log(this.name);
};

function Child() {}

Child.prototype = new Parent();  // 原型链继承发生在这里
Child.prototype.constructor = Child;

const child1 = new Child();
const child2 = new Child();

child1.colors.push('blue');

// child1实例通过__proto__找到Child()构造函数的prototype内的colors
console.log(child1.colors); // ['red', 'green', 'blue']
console.log(child2.colors); // ['red', 'green', 'blue'] 共享了引用类型
```

- 不能向父类构造函数传参

- **还有其他多种继承方式以后再挖？？？**

17.说一下new会发生什么?

使用 new 关键字调用函数的行为被称为实例化

```
function Person(name) {
  this.name = name
```

```
}
const p = new Person('Tom')
```

实例化的执行过程：

- 创建一个新的空对象
- 将这个对象的 `__proto__` 指向 构造函数.prototype
- 构造函数的 this 指向新对象，然后执行构造函数的代码，修改 this，添加新属性
- 返回新的对象

18.说一说defer和async的区别?

`defer` 和 `async` 都是用于控制外部 js脚本 加载和执行行为的属性，`<script>` 默认是同步阻塞渲染的，`defer` 和 `async`，它们都可以让脚本**异步加载**，但执行时机不同

- **defer：**脚本异步加载，但会等到 **HTML 解析完成后再按顺序执行**（如果有多个带有defer的脚本），适用于多个脚本有依赖、需要按顺序执行的场景
- **async：脚本也是异步加载，但一加载完成就立即执行**（会中断 HTML 解析），不等 HTML 解析，也不保证顺序。适合独立、无依赖的脚本，如广告等

实际开发中，更倾向于使用 `defer`，以确保脚本不会阻塞页面解析又能按需执行

19.说一说js实现异步的方法?

- **回调函数**

  最早期的异步方式，通过回调函数执行异步逻辑，比如 setTimeout、事件监听、ajax 等，但多个回调函数嵌套的时候会造成回调函数地狱

- **Promise**

  使用 `.then()`、`.catch()` 链式调用，让异步逻辑更清晰，避免了层层嵌套的问题

- **async / await**

  generator 和 promise 的语法糖，使异步代码像同步一样编写，增强可读性，但必须搭配 Promise 使用

  > await 只能等待一个 Promise 对象，如果 await 后不是 Promise，JS 会自动包装成一个立即 resolve 的 Promise，但这不是它设计的主要用途。

- **generator**

  早期 async/await 的替代方案，现在不常用了

20.说一说cookie sessionStorage localStorage 的区别?

cookie、sessionStorage 和 localStorage 都是前端用于在浏览器中存储数据的方式

- **cookie**通常用于与服务器通信，比如身份验证，会随着每次 HTTP 请求一起发送给服务器，**可设置过期时间**，大小限制约 4KB
- **sessionStorage**在**页面关闭后销毁**，适合临时数据，大小限制约5MB

- **localStorage数据持久保存**，除非手动清除，适合长期保存用户信息，大小限制约5MB

21.说一说如何实现可过期的 localStorage 数据?

封装两个函数：

- **setItem(key, value, expire)**

  存值的时候，保存

```
{
  value: 真正的值,
  expire: 过期时间戳 (ms)
}
过期时间戳 (ms)  = Date.now() + 过期时间 (ms)
```

- **getItem(key)**

  取值的时候，判断当前时间是否超过 expire，超过就说明过期了，删除并返回 null

22.说一下 token 能放在 cookie 中吗?

token 是可以放在 cookie 里的，这样浏览器每次发请求会自动带上，适合一些服务端渲染或者不需要频繁操作的场景。不过在前后端分离的项目中，通常会把 token 放在 localStorage 或 sessionStorage 里，然后在请求时手动加到请求头，这样可以更灵活一些，也能更好地防止 CSRF 攻击

23.说一说 axios 的拦截器原理及应用?

Axios 的拦截器本质是对请求和响应做统一处理的“中间层”，它在请求发出之前、响应到达之后执行对应的函数。

常见应用包括：

- 在请求拦截器（request）中统一加 token（如加到请求头 Authorization）
- 在响应拦截器（response）中统一处理返回的数据格式，或拦截 401 等错误码跳转登录

  > 拦截 401 等错误码跳转登录：
  >
  > 我在项目中会使用 axios 的响应拦截器来统一处理错误，比如常见的 401 状态码。401 一般是由于 token 过期或未登录导致的，我会在拦截器中判断这个状态码，然后跳转到登录页，并清除本地的 token 或用户信息，这样可以避免每个接口都单独写处理逻辑

- 做接口 loading 状态的统一处理（请求发出之前开 loading，响应到达之后关 loading）

它的原理是 Axios 内部维护了一个拦截器数组（请求和响应各一个），每次请求发出和响应回来都会按顺序执行这些拦截器。

24.说一说 vue 的 keep-alive ?

Vue 的 `<keep-alive>` 是一个抽象组件，用来缓存已经创建的组件实例。被包裹的组件在切换时不会被销毁，而是会被缓存起来，当再次激活时会直接复用，避免重复渲染，提高性能。常用于组件切换时需要保留状态的场景，比如 tab 切换

```
<keep-alive>
  <router-view></router-view>
</keep-alive>
```

当组件在 `<keep-alive>` 内被切换，它的 `activated` 和 `deactivated` 这两个生命周期钩子函数将会被对应执行

> `activated()`：组件被激活（再次显示）时调用
>
> `deactivated()`：组件被隐藏（缓存）时调用

常见的属性：

- include：只有名字匹配的组件会被缓存 `<keep-alive include="MyComponent">`
- exclude：排除不需要缓存的组件 `<keep-alive exclude="LoginPage">`
- max：缓存组件实例的最大数 `<keep-alive :max="5">`

25.说一说前端性能优化手段?

前端性能优化分为两类：一类是文件加载更快、另一类是文件渲染更快。

加载更快的方法：

- 让传输的数据包更小（压缩文件/图片）：图片压缩和文件压缩
- 减少网络请求的次数：雪碧图/精灵图、节流防抖
- 减少渲染的次数：缓存（HTTP缓存、本地缓存、Vue的keep-alive缓存等）

渲染更快的方法：

- 提前渲染：SSR服务端渲染
- 避免渲染阻塞：CSS 放在 HTML 的 `<head>` 中，JS 放在 HTML 的 `<body>` 底部
- 避免无用渲染：懒加载
- 减少渲染次数：对 DOM 查询进行缓存、将 DOM 操作合并、使用减少重排的标签

  > 缓存 DOM 查询，DOM 查询次数，避免反复遍历 DOM 树
  >
  > 合并 DOM 操作，减少重排/重绘次数（重排/重绘极其消耗资源）

26.说一说性能优化有哪些性能指标，如何量化?

核心的性能指标主要有：

- LCP（Largest Contentful Paint）- 最大内容绘制

视口中最大内容元素呈现所需时间

- FID（First Input Delay）- 首次输入延迟

  用户首次交互到浏览器响应的时间

- CLS（Cumulative Layout Shift）- 累计布局偏移

  页面生命周期内发生的意外布局偏移总和

测量这些指标常用的量化分析工具：

- Chrome DevTools - Performance性能面板分析运行时性能
- Lighthouse

27.说一说服务端渲染?

服务端渲染（SSR）指的是：在服务器上把页面 HTML 内容生成好，在返回给浏览器。浏览器拿到的就是"成品页面"，而不是像客户端渲染（CSR）那样拿到一堆 js 代码再在浏览器里生成页面。

SSR 的核心流程是：

- 用户请求页面（比如访问 https://www.baidu.com/home）
- 服务器拿到请求后调用页面中需要的后端接口，然后拿到数据
- 服务器把 HTML 和 数据 一起拼好，直接返回给浏览器
- 浏览器拿到完整的 HTML，直接展示，JS 在接管交互（Hydration）

SSR 的优点：

- **更快的首屏渲染速度：**因为 HTML 已经在服务器生成好了，用户打开页面就能看到内容（不用再等 js 执行、请求数据）
- **更利于 SEO：**搜索引擎能直接抓取 HTML 内容
- **更一致的性能体验：**不依赖客户端设备性能，避免了 CSR 的"白屏"问题

但 SSR 会**造成服务器压力更大**

28.XSS攻击是什么?

XSS 指的是：**攻击者往网页中注入恶意脚本代码，当用户浏览时，脚本就在用户浏览器里执行**，从而实现偷 cookie、钓鱼跳转等目的

29.CSRF攻击是什么?

CSRF指的是：用户登录某个网站后，在未退出的情况下，被第三方网站引导发起请求，利用用户登录状态**冒充用户**行为

例如：

用户登录了银行网站 `bank.com`，攻击者诱导用户点击一段代码：

```
<img src="https://bank.com/transfer?to=badguy&amount=10000">
```

浏览器会带上 cookie 自动访问 `bank.com`，相当于用户自己在转账（利用用户登录状态本地存储的 cookie 冒充用户伪造请求）

30.说一下 fetch 请求方式?

fetch 是一个原生的基于 Promise 设计的 HTTP 请求 API，没有使用XMLHttpRequest对象，语法更简洁，支持 async/await，默认返回一个 Promise，它不会自动处理 HTTP 错误（比如 404、500），所以需要手动检查 `res.ok` 来判断请求是否成功

```
fetch('/api/data')
  .then(res => {
    if (!res.ok) {
      throw new Error('请求失败，状态码：' + res.status)
    }
    return res.json()
  })
  .then(data => console.log(data))
  .catch(err => console.error('捕获错误:', err))
```

31.说一下有什么方法可以保持前后端实时通信?

保持前后端实时通信常用的方法有轮询、长轮询、WebSocket 和 SSE。我们平时比较常用 WebSocket，它可以实现真正的双向通信，适合做聊天、订单状态更新这些需要实时响应的功能。相比之下，轮询和长轮询简单但性能较差，更适合业务压力不大的场景

32.浏览器输入 URL 发生了什么?

- **DNS 解析：**将域名解析为对应的 IP 地址
- **建立 TCP 连接：**通过三次握手和服务器建立连接（如果是 HTTPS，还会有 TLS 握手）
- **发送 HTTP 请求：**浏览器构造请求报文发送到服务器
- **服务器处理请求：**服务器接收到请求后处理，并返回响应数据（HTML、JSON、图片等）
- **浏览器接收响应并渲染页面**
- **TCP 四次挥手断开连接（或进入 keep-alive）**

33.说一下浏览器如何渲染页面的?

- **解析 HTML，生成 DOM 树**

  浏览器从上到下解析 HTML 文件，遇到标签就创建对应的 DOM 节点，组织成一课 DOM树

  - DOM 是文档对象模型，是 HTML 结构的 JS 表达

    > DOM 是浏览器根据 HTML 构建出来的一种对象模型，它以 JavaScript 能操作的“对象”的形式，表达页面中的结构。DOM 树的每一个元素，其实就是 JS 中的对象

  - 会阻塞：遇到 `<script>` 会暂停 HTML 解析，执行完 JS 才继续

- **解析 CSS，生成 CSSOM 树**

并行地，浏览器也会下载、解析 `style` 或外部 css 文件，生成样式对象树 - CSSOM（CSS Object Model）

- 合并 DOM 和 CSSOM，生成渲染树（Render Tree）

  DOM 决定结构，CSSOM 决定样式，浏览器会将这两者结合，生成渲染树

- **布局 / 回流**

  浏览器会计算每个渲染树节点的精确位置和大小，这个过程叫 布局 / 回流

- **绘制**

  把每个节点的样式、颜色、边框、阴影等绘制到屏幕的多个图层中

- **合成图层**

  浏览器会为特定元素创建独立图层（例如 `tranform`、`position: fixed`），然后将这些图层合成一张最终图像

- **显示**

  最后，渲染后的像素图像送到屏幕，页面显示完成。

```
// DOM Tree 和 CSSOM Tree 的生成是同时进行的
HTML      →     DOM Tree
CSS       →     CSSOM Tree
                    ↓
            Render Tree（渲染树）
                    ↓
            Layout/Reflow（布局/回流）
                    ↓
                Paint（绘制）
                    ↓
            Composite（合成图层）
                    ↓
               Display（显示）
```

34.说一下重排、重绘的区别以及如何避免?

重排 / 回流：当渲染树中部分或者全部元素的尺寸、结构或者属性发生变化时，浏览器会重新渲染部分或者全部文档的过程就是重排

```
// 几何属性改变
element.style.width = '100px'
element.style.padding = '10px'

// 布局查询（强制同步布局）
const width = element.offsetWidth
```

```
// 内容变化
element.innerHTML = '<div>new content</div>'

// 窗口操作
window.resizeBy(100, 100)
window.scrollTo(0, 100)
```

重绘：当页面中某些元素的样式发生变化，但是不会影响其在文档流中的位置时，浏览器就会对元素进行重新绘制，这个过程就是重绘

```
// 外观属性改变
element.style.color = 'red'
element.style.backgroundColor = '#fff'

// 不改变几何布局的变换
element.style.opacity = '0.5'
element.style.transform = 'scale(1.1)'
```

当触发回流时，一定会触发重绘，但是重绘不一定会引发回流，重排的性能消耗更高

如何避免重排 / 重绘带来的性能问题?

- 批量修改样式时，用 class 代替多次 style 设置

```
// 错误写法
el.style.width = "100px";
el.style.height = "200px";
// 正确写法
el.classList.add("new-style");
```

- 避免频繁读写引起强制同步布局

> 一边写一边读，浏览器会“强制刷新布局”

```
// 频繁读写会引起强制重排
el.style.width = "100px"
console.log(el.offsetHeight) // 强制重新计算布局
el.style.height = "200px"
```

- 使用文档碎片（DocumentFragment）统一插入 DOM 节点
- 使用 CSS3 transform、opacity 做动画

> 这些属性**不会引起重排**，推荐用于动画性能优化

35.说一下事件循环Event loop?

JavaScript 是单线程的，为了处理异步操作，采用了事件循环机制，js执行代码时，

1. 从上到下将同步代码放入**调用栈**执行，遇到异步代码（宏任务/微任务）交给**宿主环境（浏览器）**，异步代码有结果则其回调函数进入**对应队列**
2. 当调用栈空闲时，先清空微任务队列内所有微任务的回调函数，然后执行宏任务队列内第一个宏任务的回调函数（微任务优先级大于宏任务）
3. 之后再检查微任务队列内是否有待执行的代码，如果有则清空微任务队列，如果没有则继续执行下一个宏任务的回调函数
4. 依次循环