端面试题整理

已同步到掘金、CSDN

掘金地址： https://juejin.cn/post/7075332630417244173

CSDN 地址：https://blog.csdn.net/z1832729975/article/details/123431083

个人整理了很多网上常见的面试题，希望也能通过这来复习

内容有点多，可能 CSDN 上预览效果不好，想要 markdown 文档的可以私信我，推荐使用Typora看

比较好的面试题

2021 年我的前端面试准备

2021 年前端面试必读文章【超三百篇文章/赠复习导图】

CSS、HTML

浏览器内核

> IE: trident 内核 Firefox：gecko 内核 Safari: webkit 内核 Opera: 以前是 presto 内核，Opera 现已改用 GoogleChrome 的 Blink 内核 Chrome: Blink(基于 webkit，Google 与 Opera Software 共同开发)

你是怎么理解 HTML 语义化

HTML 语义化简单来说就是用正确的标签来做正确的事。

比如表示段落用 p 标签、表示标题用 h1-h6 标签、表示文章就用 article 等。

DOCTYPE 的作用

> Doctype 作用？严格模式与混杂模式如何区分？它们有何差异？

1. <!DOCTYPE> 声明位于文档中的最前面，处于 <html> 标签之前。告知浏览器以何种模式来渲染文档。
2. 严格模式的排版和 JS 运作模式是 以该浏览器支持的最高标准运行。
3. 在混杂模式中，页面以宽松的向后兼容的方式显示。模拟老式浏览器的行为以防止站 点无法工作。
4. DOCTYPE 不存在或格式不正确会导致文档以混杂模式呈现。复制代码 你知道多少种 Doctype 文档类型？ 该标签可声明三种 DTD 类型，分别表示严格版本、过渡版本以及基于框架的 HTML 文档。 HTML 4.01 规定了三种文档类型：Strict、Transitional 以及 Frameset。 XHTML 1.0 规定了三种 XML 文档类型：Strict、Transitional 以及 Frameset。 Standards （标准）模式（也就是严格呈现模式）用于呈现遵循最新标准的网页，而 Quirks （包容）模式（也就是松散呈现模式或者兼容模式）用于呈现为传统浏览器而设计的网页。

行内元素、块级元素、 空元素有那些?

- 行内元素 (不能设置宽高，设置宽高无效) a,span,i,em,strong,label
- 行内块元素：img, input
- 块元素： div, p, h1-h6, ul,li,ol,dl,table...
- 知名的空元素 br, hr,img, input,link,meta

可以通过 display 修改 inline-block, block, inline

注意

只有文字才能组成段落，因此 p 标签里面不能放块级元素，特别是 p 标签不能放 div。同理还有这些标签h1,h2,h3,h4,h5,h6,dt ，他们都是文字类块级标签，里面不能放其他块级元素。

meta viewport 是做什么用的，怎么写

使用目的

告诉浏览器，用户在移动端时如何缩放页面

```
<meta
  name="viewport"
```

```
  content="width=device-width,
                initial-scale=1,
                maximum-scale-1, minimum-scale=1"
/>

```

with=device-width 将布局视窗（layout viewport）的宽度设置为设备屏幕分辨率的宽度
initial-scale=1 页面初始缩放比例为屏幕分辨率的宽度
maximum-scale=1 指定用户能够放大的最大比例
minimum-scale=1 指定用户能够缩小的最大比例

label 标签的作用

label 标签来定义表单控制间的关系,当用户选择该标签时，浏览器会自动将焦点转到和标签相关的表单控件上。

```
<label for="Name">Number:</label>
<input type="text" name="Name" id="Name" />

<label
  >Date:
  <input type="text" name="B" />
</label>

```

canvas 在标签上设置宽高 和在 style 中设置宽高有什么 区别

> canvas 标签的 width 和 height 是画布实际宽度和高度，绘制的图形都是在这个上面。 而 style 的 width 和 height 是 canvas 在浏览器中被渲染的高度和宽度。 如果 canvas 的 width 和 height 没指定或值不正确，就被设置成默认值 。

html5 新特性

> HTML5 新特性

1. 语义化标签， header, footer, nav, aside,article,section
2. 增强型表单
3. 视频 video 和音频 audio
4. Canvas 绘图
5. SVG绘图
6. 地理定位
7. 拖放 API

8. WebWorker
9. WebStorage(本地离线存储 localStorage、sessionStorage)
10. WebSocket

css3 新特性

> CSS3 有哪些新特性？CSS3 新特性详解

1、圆角效果；2、图形化边界；3、块阴影与文字阴影；4、使用 RGBA 实现透明效果；5、渐变效果；6、使用“@Font-Face”实现定制字体；7、多背景图；8、文字或图像的变形处理；9、多栏布局；10、媒体查询等。

```
1、颜色：新增RGBA、HSLA模式
2、文字阴影：（text-shadow）
3、边框：圆角（border-radius）边框阴影：box-shadow
4、盒子模型：box-sizing
5、背景：background-size,background-origin background-clip(削弱)
6、渐变：linear-gradient(线性渐变):
eg: background-image: linear-gradient(100deg, #237b9f, #f2febd);
radial-gradient (径向渐变)
7、过渡：transition可实现动画
8、自定义动画： animate@keyfrom
9、媒体查询：多栏布局@media screen and (width:800px)
10、border-image
11、2D转换:transform:translate(x,y) rotate(x,y)旋转 skew(x,y)倾斜 scale(x,y)缩放
12、3D转换
13、字体图标：Font-Face
14、弹性布局：flex

```

css 选择器

id 选择器（ #myid）
类选择器（.myclassname）
标签选择器（div, h1, p）相邻选择器（h1 + p）
子选择器（ul > li）后代选择器（li a）
属性选择器（a[rel = "external"]）
伪类选择器（a: hover, li:nth-child）
通配符选择器（ * ）

```
        !Important > 行内式 > id > 类/伪类/属性 > 标签选择器  >  全局
```

```
    （对应权重：无穷大∞ > 1000> 100 >  10        >  1    >      0）

```

盒模型

一个盒子，会有 content,padding,border,margin 四部分，
标准的盒模型的宽高指的是 content 部分
ie 的盒模型的宽高包括了 content+padding+border
我们可以通过 box-sizing 修改盒模型，box-sizing border-box content-box

margin 合并

在垂直方向上的两个盒子，他们的 margin 会发生合并（会取最大的值），比如上边盒子设置margin-bottom:20px，下边盒子设置margin-top:30px;，那么两个盒子间的间距只有30px，不会是50px
解决 margin 合并，我们可以给其中一个盒子套上一个父盒子，给父盒子设置 BFC

margin 塌陷

效果： 一个父盒子中有一个子盒子，我们给子盒子设置margin-top:xxpx结果发现会带着父盒子一起移动（就效果和父盒子设置margin-top:xxpx的效果一样）
解决： 1、给父盒子设置 border，例如设置border:1px solid red; 2、给父盒子设置 BFC

BFC

> 块级格式化上下文（block format context）

BFC 的布局规则 *

- 内部的 Box 会在垂直方向，一个接一个地放置。
- Box 垂直方向的距离由 margin 决定。属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠。
- 每个盒子（块盒与行盒）的 margin box 的左边，与包含块 border box 的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。
- BFC 的区域不会与 float box 重叠。
- BFC 就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。
- 计算 BFC 的高度时，浮动元素也参与计算。

触发 BFC 的条件 *

- 根元素 html
- float 的值不是 none。
- position 的值 absoute、fixed
- display 的值是 inline-block、table-cell、flex、table-caption 或者 inline-flex
- overflow 的值不是 visible

解决什么问题

1. 可以用来解决两栏布局BFC 的区域不会与 float box 重叠.left { float: flet; } .right { overflow: hidden; }
2. 解决 margin 塌陷和 margin 合并问题
3. 解决浮动元素无法撑起父元素

flex

> 设为 Flex 布局以后，子元素的 float、clear 和 vertical-align 属性将失效

什么是 rem、px、em 区别

> rem 是一个相对单位，rem 的是相对于 html 元素的字体大小，没有继承性 em 是一个相对单位，是相对于父元素字体大小有继承性 px 是一个“绝对单位”，就是 css 中定义的像素，利用 px 设置字体大小及元素的宽高等，比较稳定和精确。

响应式布局

响应式布局有哪些实现方式？什么是响应式设计？响应式设计的基本原理是什么？

> 1.百分比布局，但是无法对字体，边框等比例缩放 2.弹性盒子布局 display:flex 3.rem 布局，1rem=html 的 font-size 值的大小 css3 媒体查询 @media screen and(max-width: 750px){} 5.vw+vh 6.使用一些框架（bootstrap，vant） 什么是响应式设计：响应式网站设计是一个网站能够兼容多个终端，智能地根据不同设备环境进行相对应的布局 响应式设计的基本原理：基本原理是通过媒体查询检测不同的设备屏幕尺寸设置不同的 css 样式 页面头部必须有 meta 声明的

布局

- 两栏布局,左边定宽，右边自适应
- 三栏布局、圣杯布局、双飞翼布局

水平垂直居中

方法一：给父元素设置成弹性盒子，子元素横向居中，纵向居中

方法二：父相子绝后，子部分向上移动本身宽度和高度的一半，也可以用 transfrom:translate(-50%,-50%)（最常用方法）

方法三：父相子绝，子元素所有定位为 0，margin 设置 auto 自适应

iframe 有哪些缺点?

iframe 是一种框架，也是一种很常见的网页嵌入方

iframe 的优点：

1. iframe 能够原封不动的把嵌入的网页展现出来。
2. 如果有多个网页引用 iframe，那么你只需要修改 iframe 的内容，就可以实现调用的每一个页面内容的更改，方便快捷。

3. 网页如果为了统一风格，头部和版本都是一样的，就可以写成一个页面，用 iframe 来嵌套，可以增加代码的可重用。
4. 如果遇到加载缓慢的第三方内容如图标和广告，这些问题可以由 iframe 来解决。

iframe 的缺点：

1. 会产生很多页面，不容易管理。
2. iframe 框架结构有时会让人感到迷惑，如果框架个数多的话，可能会出现上下、左右滚动条，会分散访问者的注意力，用户体验度差。
3. 代码复杂，无法被一些搜索引擎索引到，这一点很关键，现在的搜索引擎爬虫还不能很好的处理 iframe 中的内容，所以使用 iframe 会不利于搜索引擎优化。
4. 很多的移动设备（PDA 手机）无法完全显示框架，设备兼容性差。
5. iframe 框架页面会增加服务器的 http 请求，对于大型网站是不可取的。现在基本上都是用 Ajax 来代替 iframe，所以 iframe 已经渐渐的退出了前端开发。

link @import 导入 css

> link 是 XHTML 标签，除了加载 CSS 外，还可以定义 RSS 等其他事务； @import 属于 CSS 范畴， 只能加载 CSS。 link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载。link 无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持。 link 支持使用 Javascript 控制 DOM 去改变样式；而@import 不支持。

DOM 事件机制/模型

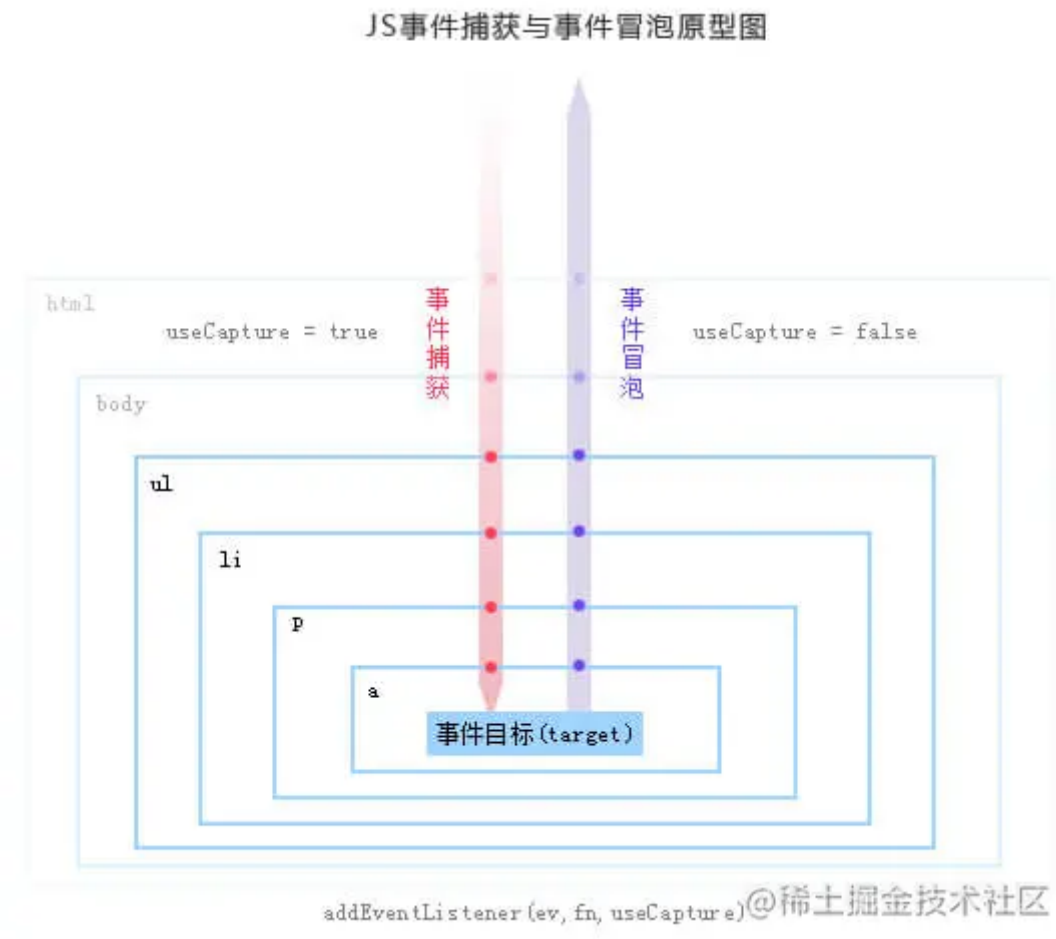
> DOM0 级模型、IE 事件模型、DOM2 级事件模型

就比如用户触发一个点击事件，有一个触发的过程

事件捕获-阶段（从上大小，从外到内）--->处于目标事件-阶段---->事件冒泡-阶段（从下到上，从内到外）

```
window.addEventListener(
  "click",
  function (event) {
    event = event || window.event /*ie*/;
    const target = event.target || event.srcElement; /*ie*/ // 拿到事件目标

    // 阻止冒泡
    // event.stopPropagation()
    // event.cancelBubble=true; // ie

    // 阻止默认事件
    // event.preventDefault();
    // event.returnValue=false; // ie
```

```
  },
  /* 是否使用捕获，默认是fasle, */ fasle
);

```



事件委托

简介：事件委托指的是，不在事件的发生地（直接 dom）上设置监听函数，而是在其父元素上设置监听函数，通过事件冒泡，父元素可以监听到子元素上事件的触发，通过判断事件发生元素 DOM 的类型，来做出不同的响应。

举例：最经典的就是 ul 和 li 标签的事件监听，比如我们在添加事件时候，采用事件委托机制，不会在 li 标签上直接添加，而是在 ul 父元素上添加。

好处：比较合适动态元素的绑定，新添加的子元素也会有监听函数，也可以有事件触发机制

如果需要手动写动画，你认为最小时间间隔是多久

> 多数显示器默认频率是 60Hz，即 1 秒刷新 60 次，所以理论上最小间隔为 1/60＊1000ms = 16.7ms

::before和:after中双冒号和单冒号有什么区别

单冒号(:)用于 CSS3 伪类，双冒号(::)用于 CSS3 伪元素。 ::before 就是以一个子元素的存在，定义在元素主体内容之前的一个伪元素。并不存在于 dom 之中，只存在在页面之中。 :before 和 :after 这两个伪元素，是在 CSS2.1 里新出现的。起初，伪元素的前缀使用的是单 冒号语法，但随着 Web 的进化，在 CSS3 的规范里，伪元素的语法被修改成使用双冒号，成 为::before ::after

CSS sprites 精灵图

CSS Sprites 其实就是把网页中一些背景图片整合到一张图片文件中，再利用 CSS 的 “background-image”， “ background-repeat ”， “ background-position” 的组合进行背景定位，background-position 可以用数字能精确的定位出背景图片的位置。这样可以减少很多图片请 求的开销，因为请求耗时比较长；请求虽然可以并发，但是也有限制，一般浏览器都是 6 个

重排和重绘

重绘（repaint 或 redraw）：当盒子的位置、大小以及其他属性，例如颜色、字 体大小等都确定下来之后，浏览器便把这些原色都按照各自的特性绘制一遍，将 内容呈现在页面上。重绘是指一个元素外观的改变所触发的浏览器行为，浏览器 会根据元素的新属性重新绘制，使元素呈现新的外观。
触发重绘的条件：改变元素外观属性。如：color，background-color 等。 注意：table 及其内部元素可能需要多次计算才能确定好其在渲染树中节点的属性值，比同等元素要多花两倍时间，这就是我们尽量避免使用 table 布局页面的 原因之一。
重排（重构/回流/reflow）：当渲染树中的一部分(或全部)因为元素的规模尺寸， 布局，隐藏等改变而需要重新构建, 这就称为回流(reflow)。每个页面至少需要 一次回流，就是在页面第一次加载的时候。
重绘和重排的关系：在回流的时候，浏览器会使渲染树中受到影响的部分失效， 并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕 中，该过程称为重绘。所以，重排必定会引发重绘，但重绘不一定会引发重排。

JavaScript

js 数据类型

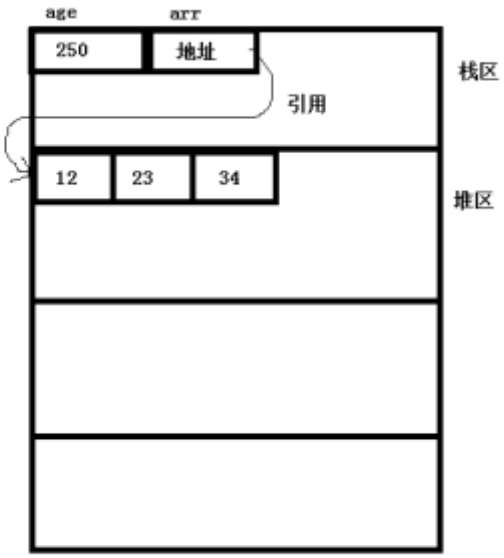
8 中, ES6出的 Symbol BigInt

```
Number String Boolean undefined null Object Symbol BigInt

```

js 的基本数据类型和复杂数据类型的区别（在堆和栈中，赋值时的不同,一个拷贝值一个拷贝地址）
基本类型和引用类型在内存上存储的区别

```
function test(){
      var age = 250; //值类型

      var arr = new Array(12,23,34);

}
```

内存的地址，就是个编号



null 与 undefined 的异同

相同点:

- Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null

不同点:

- null 转换成数字是 0, undefined 转换数字是NaN
- undefined 代表的含义是未定义， null 代表的含义是空对象。
- typeof null 返回'object'，typeof undefined 返回'undefined'
- null == undefined; // true null === undefined; // false
- 其实 null 不是对象，虽然 typeof null 会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，然而 null 表示为全零，所以将它错误的判断为 object 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

说说 JavaScript 中判断数据类型的几种方法

typeof

- typeof一般用来判断基本数据类型，除了判断 null 会输出"object"，其它都是正确的
- typeof判断引用数据类型时，除了判断函数会输出"function",其它都是输出"object"

instanceof

> Instanceof 可以准确的判断引用数据类型，它的原理是检测构造函数的prototype属性是否在某个实例对象的原型链上， 不能判断基本数据类型

```
// instanceof 的实现
function instanceofOper(left, right) {
  const prototype = right.prototype;
  while (left) {
    if ((left = left.__proto__) === prototype) {
      return true;
```

```
    }
  }
  return false;
}
// let obj  = {}
// Object.getPrototypeOf(obj) === obj.__proto__ ==> true

```

```
// 实现 instanceof 2
function myInstanceof(left, right) {
  // 这里先用typeof来判断基础数据类型，如果是，直接返回false
  if (typeof left !== "object" || left === null) return false;
  // getProtypeOf是Object对象自带的API，能够拿到参数的原型对象
  let proto = Object.getPrototypeOf(left);
  while (true) {
    if (proto === null) return false;
    if (proto === right.prototype) return true; //找到相同原型对象，返回true
    proto = Object.getPrototypeof(proto);
  }
}

```

Object.prototype.toString.call() 返回 [object Xxxx] 都能判断

深拷贝和浅拷贝

```
let obj = { b: "xxx" };
let arr = [{ a: "ss" }, obj, 333];

// 赋值
let arr2 = arr;
// 浅拷贝-只拷贝了一层，深层次的引用还是存在
// Object.assign(), ...扩展运算符，slice等
let arr2 = arr.slice();
let arr2 = [...arr];
obj.b = "222"; // arr2[1].b => 222
// arr[2] = 4444 ==> arr2[2] ===> 333

// 深拷贝
// 1. 最简单的，JSON.stringify，但这个有问题，看下面有说明
```

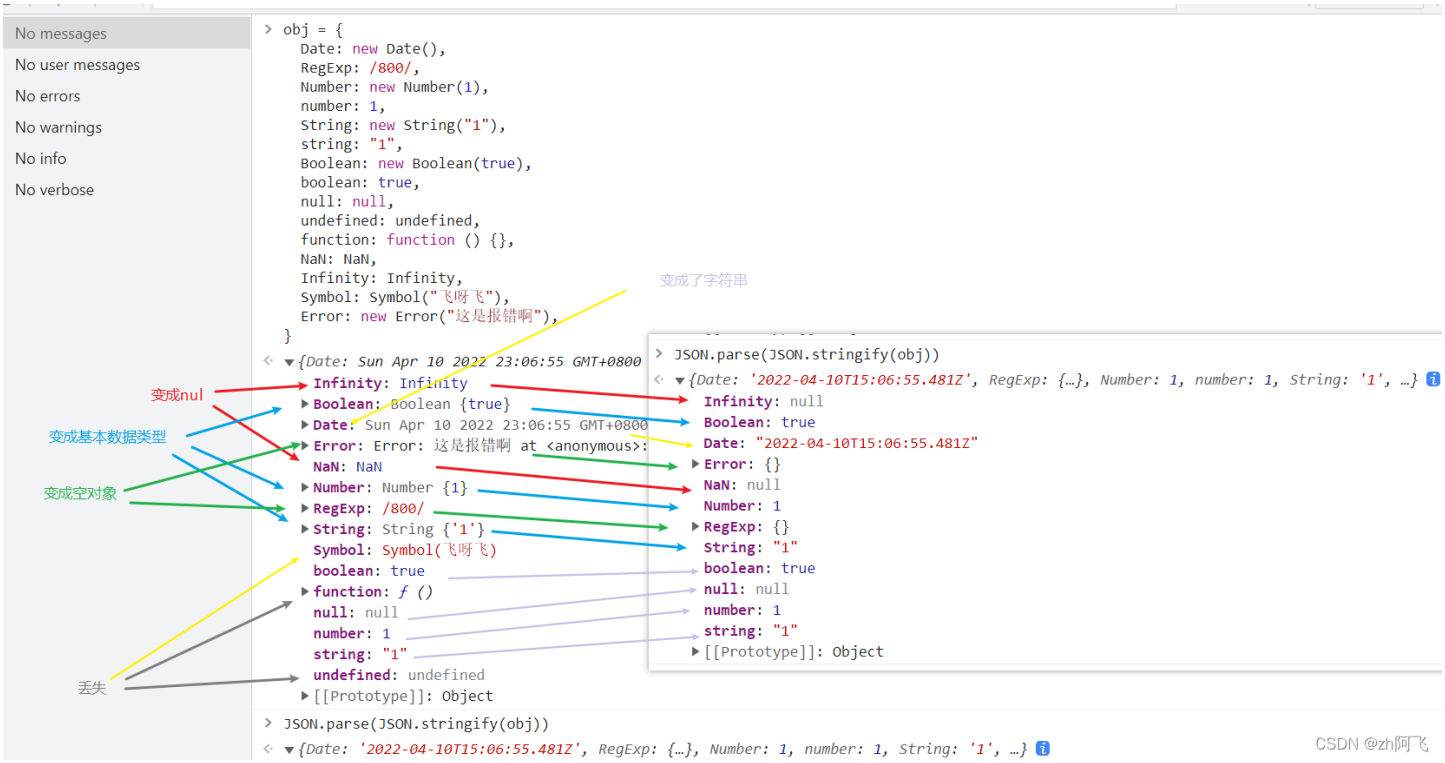
```
let arr2 = JSON.parse(JSON.stringify(arr));

// 2. 自己封装，要递归处理

```

实现深拷贝-简单版

```
export function deepClone(obj, map = new Map()) {
  if (!obj && typeof obj !== "object") return obj;
  if (obj instanceof Date) return new Date(obj);
  if (obj instanceof RegExp) return new RegExp(obj.source, obj.flags);

  if (map.get(obj)) {
    //  如果有循环引用、就返回这个对象
    return map.get(obj);
  }

  const cloneObj = obj.constructor(); // 数组的就是[],对象就是{}

  map.set(obj, cloneObj); // 缓存对象，用于循环引用的情况

  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      cloneObj[key] = deepClone(obj[key], map);
    }
  }

  return cloneObj;
}

```

JSON.stringify 问题

1. 如果有循环引用就报错
2. Symbol、function、undefined会丢失
3. 布尔值、数字、字符串的包装对象会转换成原始值
4. NaN、Infinity 变成 null
5. Date类型的日期会变成字符串
6. RegExp、Error被转换成了空对象 {}



模块化

- commonjs// 由nodejs实现 const fs = require("fs"); module.exports = {};
- ESM// 由es6实现 import $ from "jquery"; export default $;
- AMD（异步加载模块）// 由RequireJS实现 define(["juqery", "vue"], function ($, Vue) { // 依赖必须一开始就写好 $("#app"); new Vue({}); });
- CMD// 由SeaJS 实现 define(function (require, exports, module) { var a = require("./a"); a.doSomething(); // .... var b = require("./b"); // 依赖可以就近书写 b.doSomething(); // ... });
- UMD (通用加载模块)(function (global, factory) { typeof exports === 'object' && typeof module !== 'undefined' ? module.exports = factory() : typeof define === 'function' && define.amd ? define(factory) : (global = global || self, global.Vue = factory()); }(this, function () { 'use strict';

AMD 和 CMD 的区别有哪些

> https://blog.csdn.net/qq_38912819/article/details/80597101

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）
2. CMD 推崇依赖就近，AMD 推崇依赖前置

CommonJS 与 ES6 Module 的差异

CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

- CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。
- ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，

ES6 的import有点像 Unix 系统的“符号连接”，原始值变了，import加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

- 运行时加载: CommonJS 模块就是对象；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。
- 编译时加载: ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，import时采用静态命令的形式。即在import时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”。

CommonJS 加载的是一个对象（即 module.exports 属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

JS 延迟加载的方式

JavaScript 会阻塞 DOM 的解析，因此也就会阻塞 DOM 的加载。所以有时候我们希望延迟 JS 的加载来提高页面的加载速度。

- 把 JS 放在页面的最底部
- script 标签的 defer 属性：脚本会立即下载但延迟到整个页面加载完毕再执行。该属性对于内联脚本无作用 (即没有 「src」 属性的脚本）。
- Async 是在外部 JS 加载完成后，浏览器空闲时，Load 事件触发前执行，标记为 async 的脚本并不保证按照指定他们的先后顺序执行，该属性对于内联脚本无作用 (即没有 「src」 属性的脚本）。
- 动态创建 script 标签，监听 dom 加载完毕再引入 js 文件

call、apply 、bind

> call，apply，bind 都是改变 this 指向，bind 不会立即执行，会返回的是一个绑定 this 的新函数 面试官问：能否模拟实现 JS 的 call 和 apply 方法

```
obj.call(this指向, 参数1,  参数2)ss
obj.apply(this指向, [参数1,  参数2])

function fn(age) {
    console.log(this, age)
}
const obj = {name:''}
const result = fn.bind(obj) // bind会返回一个新的函数
result(20)

```

```
// 实现一个 apply
Function.prototype.myApply = function (context) {
  context = context || window;
```

```
  const fn = Symbol();
  context[fn] = this;
  var res = context[fn](...arguments[1]);
  delete context[fn];
  return res;
};

```

实现一个 bind

```
// 最终版 删除注释 详细注释版请看上文
Function.prototype.bind =
  Function.prototype.bind ||
  function bind(thisArg) {
    if (typeof this !== "function") {
      throw new TypeError(this + " must be a function");
    }
    var self = this;
    var args = [].slice.call(arguments, 1);
    var bound = function () {
      var boundArgs = [].slice.call(arguments);
      var finalArgs = args.concat(boundArgs);
      if (this instanceof bound) {
        if (self.prototype) {
          function Empty() {}
          Empty.prototype = self.prototype;
          bound.prototype = new Empty();
        }
        var result = self.apply(this, finalArgs);
        var isObject = typeof result === "object" && result !== null;
        var isFunction = typeof result === "function";
        if (isObject || isFunction) {
          return result;
        }
        return this;
      } else {
        return self.apply(thisArg, finalArgs);
      }
    };
    return bound;
  };
```

```
32
```

防抖

> debounce 所谓防抖，就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

```
function debounce(func, wait, immediate) {
  let timeout;
  return function () {
    const context = this;
    const args = [...arguments];
    if (timeout) clearTimeout(timeout);
    if (immediate) {
      const callNow = !timeout;
      timeout = setTimeout(() => {
        timeout = null;
      }, wait);
      if (callNow) func.apply(context, args);
    } else {
      timeout = setTimeout(() => {
        func.apply(context, args);
      }, wait);
    }
  };
}

```

节流

> 就是指连续触发事件但是在 n 秒中只执行一次函数

```
function throttle(fn, wait) {
  let pre = 0;
  return function (...args) {
    let now = Date.now();
    if (now - pre >= wait) {
      fn.apply(this, args);
      pre = now;
    }
```

```
    };
}

```

闭包

闭包是指有权访问另一个函数作用域中的变量的函数 ——《JavaScript 高级程序设计》
当函数可以记住并访问所在的词法作用域时，就产生了闭包，
即使函数是在当前词法作用域之外执行 ——《你不知道的 JavaScript》

- 闭包用途：
  a. 能够访问函数定义时所在的词法作用域(阻止其被回收)
  b. 私有化变量
  c. 模拟块级作用域
  d. 创建模块
- 闭包缺点：会导致函数的变量一直保存在内存中，过多的闭包可能会导致内存泄漏

原型、原型链(高频)

原型: 对象中固有的__proto__属性，该属性指向对象的prototype原型属性。
原型链: 当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是Object.prototype所以这就是我们新建的对象为什么能够使用toString()等方法的原因。
特点: JavaScript对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

this 指向、new 关键字

this对象是是执行上下文中的一个属性，它指向最后一次调用这个方法的对象，在全局函数中，this等于window，而当函数被作为某个对象调用时，this 等于那个对象。 在实际开发中，this 的指向可以通过四种调用模式来判断。

1. 函数调用，当一个函数不是一个对象的属性时，直接作为函数来调用时，this指向全局对象。
2. 方法调用，如果一个函数作为一个对象的方法来调用时，this指向这个对象。
3. 构造函数调用，this指向这个用new新创建的对象。
4. 第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。apply接收参数的是数组，call接受参数列表，`` bind方法通过传入一个对象，返回一个this绑定了传入对象的新函数。这个函数的this指向除了使用new `时会被改变，其他情况下都不会改变。

new

> 面试官问：能否模拟实现 JS 的 new 操作符

1. 首先创建了一个新的空对象
2. 设置原型，将对象的原型设置为函数的prototype对象。
3. 让函数的this指向这个对象，执行构造函数的代码（为这个新对象添加属性）
4. 判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

```
// new 操作符的实现
function newOperator(ctor) {
  if (typeof ctor !== "function") {
    throw "newOperator function the first param must be a function";
  }
  newOperator.target = ctor;
  var newObj = Object.create(ctor.prototype);
  var argsArr = [].slice.call(arguments, 1);
  var ctorReturnResult = ctor.apply(newObj, argsArr);
  var isObject =
    typeof ctorReturnResult === "object" && ctorReturnResult !== null;
  var isFunction = typeof ctorReturnResult === "function";
  if (isObject || isFunction) {
    return ctorReturnResult;
  }
  return newObj;
}

```

作用域、作用域链、变量提升

作用域负责收集和维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。(全局作用域、函数作用域、块级作用域)。 作用域链就是从当前作用域开始一层一层向上寻找某个变量，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链。

继承(含 es6)、多种继承方式

```
function Animal(name) {
  // 属性
  this.name = name || "Animal";
  // 实例方法
  this.sleep = function () {
    console.log(this.name + "正在睡觉！");
  };
```

```
}
// 原型方法
Animal.prototype.eat = function (food) {
  console.log(this.name + "正在吃: " + food);
};

```

（1）第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

```
// 原型链继承
function Cat() {}
Cat.prototype = new Animal("小黄"); // 缺点 无法实现多继承 来自原型对象的所有属性被所有实例共享
Cat.prototype.name = "cat";

```

（2）第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

```
// 借用构造函数继承
function Cat() {
  Animal.call(this, "小黄");
  // 缺点 只能继承父类实例的属性和方法，不能继承原型上的属性和方法。
}

```

（3）第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

（4）第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 Object.create() 方法就是原型式继承的实现。缺点与原型链方式相同。

```
function object(o) {
  function F() {}
  F.prototype = o;
```

```
  return new F();
}

```

（5）第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是我们的自定义类型时。缺点是没有办法实现函数的复用。

```
function createAnother(original) {
  var clone = object(original); //通过调用object函数创建一个新对象
  clone.sayHi = function () {
    //以某种方式来增强这个对象
    alert("hi");
  };
  return clone; //返回这个对象
}

```

（6）第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

```
function extend(subClass, superClass) {
  var prototype = object(superClass.prototype); //创建对象
  prototype.constructor = subClass; //增强对象
  subClass.prototype = prototype; //指定对象
}

```

类型转换

大家都知道 JS 中在使用运算符号或者对比符时，会自带隐式转换，规则如下:

-、*、/、% ： 一律转换成数值后计算

+：

- 数字 + 字符串 = 字符串， 运算顺序是从左到右
- 数字 + 对象， 优先调用对象的 valueOf -> toString
- 数字 + boolean/null -> 数字
- 数字 + undefined -> NaN
- [1].toString() === '1' 内部调用 .join 方法

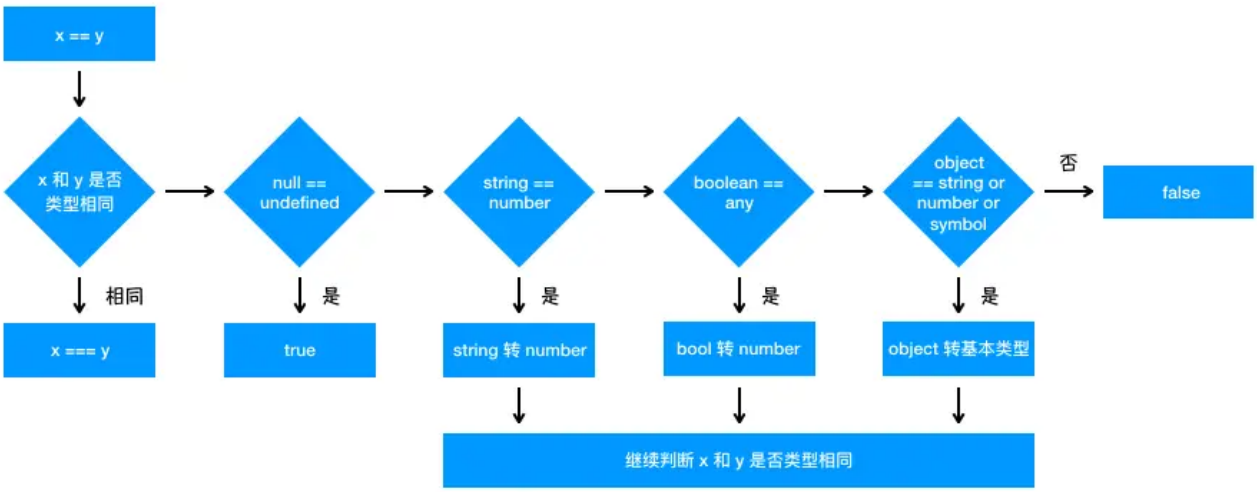
- {}.toString() === '[object object]'
- NaN !== NaN 、+undefined 为 NaN

Object.is()与比较操作符==、===的区别?

- ==会先进行类型转换再比较
- ===比较时不会进行类型转换，类型不同则直接返回 false
- Object.is()在===基础上特别处理了NaN,-0,+0,保证-0 与+0 不相等，但 NaN 与 NaN 相等

==操作符的强制类型转换规则

- 字符串和数字之间的相等比较，将字符串转换为数字之后再进行比较。
- 其他类型和布尔类型之间的相等比较，先将布尔值转换为数字后，再应用其他规则进行比较。
- null 和 undefined 之间的相等比较，结果为真。其他值和它们进行比较都返回假值。
- 对象和非对象之间的相等比较，对象先调用 ToPrimitive 抽象操作后，再进行比较。
- 如果一个操作值为 NaN ，则相等比较返回 false （ NaN 本身也不等于 NaN ）。
- 如果两个操作值都是对象，则比较它们是不是指向同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 true，否则，返回 false。



ES6

1. 新增 Symbol 类型 表示独一无二的值，用来定义独一无二的对象属性名;
2. const/let 都是用来声明变量,不可重复声明，具有块级作用域。存在暂时性死区，不存在变量提升。(const 一般用于声明常量);
3. 变量的解构赋值(包含数组、对象、字符串、数字及布尔值,函数参数),剩余运算符(...rest);
4. 模板字符串(${data});
5. ...扩展运算符(数组、对象);;

6. 箭头函数;
7. Set 和 Map 数据结构;
8. Proxy/Reflect;
9. Promise;
10. async 函数;
11. Class;
12. Module 语法(import/export)。

let/const

> const声明一个只读的常量。一旦声明，常量的值就不能改变 https://es6.ruanyifeng.com/#docs/let

var 在全局作用域中声明的变量会变成全局变量

let、const 和 var 的区别

- 不允许重复声明
- 不存在变量提升// var 的情况 console.log(foo); // 输出undefined var foo = 2; // let 的情况 console.log(bar); // 报错 ReferenceError let bar = 2;
- 暂时性死区（不能在未声明之前使用） 注意暂时性死区和不存在变量提升不是同一个东西 只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。var tmp = 123; // 声明了 tmp if (true) { tmp = "abc"; // ReferenceError let tmp; }
- 块级作用域：用 let 和 const 声明的变量，在这个块中会形成块级作用域**es5 只有函数作用域和全局作用域**IIFE 立即执行函数表达式// IIFE 写法 (function () { var tmp = ...; ... }()); // 块级作用域写法 { let tmp = ...; ... } // 函数声明 function a() {} // 函数表达式 const b = function () {};

ES6 的一些叫法

- reset 参数function add(...values) { let sum = 0; for (var val of values) { sum += val; } return sum; } add(2, 5, 3); // 10
- 扩展运算符console.log(...[1, 2, 3]); // 1 2 3 const b = { ...{ a: "2", b: "3" } };
- ?. 可选链运算符 左侧的对象是否为null或undefined。如果是的，就不再往下运算，而是返回undefined a?.b; // 等同于 a == null ? undefined : a.b; // 注意 undefined == null ==> true
- ?? Null 判断运算符

> https://es6.ruanyifeng.com/#docs/operator#Null-%E5%88%A4%E6%96%AD%E8%BF%90%E7%AE%97%E7%AC%A6

```
const headerText = response.settings.headerText ?? "Hello, world!";
const animationDuration = response.settings.animationDuration ?? 300;
const showSplashScreen = response.settings.showSplashScreen ?? true;

```

但左侧的为 undefined或者null是就返回右边的，否则就直接返回左边的

箭头函数和普通函数的区别

1. 箭头函数没有this，this是继承于当前的上下文，不能通过call,apply,bind去改变 this
2. 箭头函数没有自己的 arguments 对象，但是可以访问外围函数的 arguments 对象
3. 不能通过 new 关键字调用(不能作为构造函数)，同样也没有 new.target 和原型

如何解决异步回调地狱

promise、generator、async/await

mouseover 和 mouseenter 的区别

mouseover：当鼠标移入元素或其子元素都会触发事件，所以有一个重复触发，
冒泡的过程。对应的移除事件是 mouseout
mouseenter：当鼠标移除元素本身（不包含元素的子元素）会触发事件，也就是
不会冒泡，对应的移除事件是 mouseleave

setTimeout、setInterval 和 requestAnimationFrame 之间的区别

与 setTimeout 和 setInterval 不同，requestAnimationFrame 不需要设置时间 间隔， 大多数电脑显示器的刷新频率是 60Hz，大概相当于每秒钟重绘 60 次。大多数浏 览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频 率用户体验也不会有提升。因此，最平滑动画的最佳循环间隔是 1000ms/60，约 等于 16.6ms。 RAF 采用的是系统时间间隔，不会因为前面的任务，不会影响 RAF，但是如果前 面的任务多的话，会响应 setTimeout 和 setInterval 真正运行时的时间间隔。 特点：
（1）requestAnimationFrame 会把每一帧中的所有 DOM 操作集中起来，在一次 重绘或回流中就完成，并且重绘或回流的时间间隔紧紧跟随浏览器的刷新频率。
（2）在隐藏或不可见的元素中，requestAnimationFrame 将不会进行重绘或回 流，这当然就意味着更少的 CPU、GPU 和内存使用量
（3）requestAnimationFrame 是由浏览器专门为动画提供的 API，在运行时浏览 器会自动优化方法的调用，并且如果页面不是激活状态下的话，动画会自动暂停， 有效节省了 CPU 开销。

vue

vue2 是通过Object.defineProperty来实现响应式的，所以就会有一些缺陷

1. 当修改一个对象的某个键值属性时，当这个键值没有在这个对象中，vue 不能做响应式处理
2. 但直接修改数组的某一项（arr[index]='xxx'）vue 不能做响应式处理

可用下面的解决响应式

1. Vue.set ==> this.$set(对象\数组， key 值、index， value)

2. 修改数组length, 调用数据的 splice 方法

vue 生命周期

```
beforeCreate 实例化之前这里能拿到this，但是还不能拿到data里面的数据
created  实例化之后
beforeMount()
mounted() $el
beforeUpdate
updated

beforeDestroy 清除定时/移除监听事件
destroyed

// 被keep-alive 包裹的
// keep-alive 标签 include exclude max
activated() {},
deactivated() {},

// 父子
父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted。

// 离开页面：实例销毁 --> DOM卸载
parent  beforeDestroy
child   beforeDestroy
child   destroyed
parent  destroyed

```

Vue 的 data 为什么是一个函数

因为 Vue 的组件可能会在很多地方使用， 会产生多个实例，如果返回的是对象的， 这些组件之间的数据是同一份（引用关系），那么修改其中一个组件的数据，另外一个组件的数据都会被修改到

Vue key 值的作用

> 看这个视频，你能给面试官说这些，你就很不错了，vue 和 react 的差不多 https://www.bilibili.com/video/BV1wy4y1D7JT?p=48

...待更新

Vue 双向数据绑定原理

> 下面是照抄的一段话，个人觉得这个主要看个人理解，如果看过源码理解 MVVM 这方面的，就很简单

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty()来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。
具体步骤：
第一步：需要 observe 的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter 和 getter
这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化
第二步：compile 解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图
第三步：Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要做的事情是:
1、在自身实例化时往属性订阅器(dep)里面添加自己
2、自身必须有一个 update()方法
3、待属性变动 dep.notice()通知时，能调用自身的 update()方法，并触发 Compile 中绑定的回调，则功成身退。
第四步：MVVM 作为数据绑定的入口，整合 Observer、Compile 和 Watcher 三者，通过 Observer
来监听自己的 model 数据变化，通过 Compile 来解析编译模板指令，最终利用 Watcher 搭起 Observer 和 Compile 之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据 model 变更的双向绑定效果。
所以也可以根据这个来说明为什么 给Vue对象不存在的属性设置值的时候不生效，直接修改数组的index不生效
Vue 提供了 Vue.set(对象|数组, key|index, 值)修改触发响应式，重新数组的原型方法实现响应式

Vue extend 和 mixins

vue extend 和 mixins 的区别， mixins 里面的 函数和本身的函数重名了使用哪一个，mixins 里面的生命周期和本身的生命周期哪一个先执行
...待更新

动态组件

```
1 // component 动态组件，通过is设置要显示的组件
2 <component is="UserInfo" >
3
```

递归组件

就是给组件设置name，之后就可以在当前组件去递归使用组件

Vue 组件间的传值的几种方式

```
// Vue组件间的传值的几种方式
1. props/emit
2. $attrs/$listeners // $attrs 除了父级作用域 props、class、style 之外的属性

// $listeners 父组件里面的所有的监听方法
3. $refs/$parent/$children/$root/
4. vuex
5. 事件总线，通过new Vue去实现 / mitt <==> vue3
6. provide/inject

    // 父组件
    props: {},
    provide() {
        name: this.name,
        user: this.user
    }
    // 子组件
    props: {},
    inject: ['user']
7. 本地存储、全局变量


```

watch、mixins、组件顺序、组件配置

```
export default {
  name: "MyComponentName",
  mixins: [tableMixin],
  components: {},
  inject: ["xxx"],
  // props: ['value', 'visible'],
  props: {
    id: String,
```

```
    type: {
      // required: true,
      type: String,
      default: "warning",
      validator(val) {
        return ["primary", "warning", "danger", "success", "info"].includes(
          val
        );
      },
    },
    list: {
      type: Array,
      default: () => [],
    },
  },
  data() {
    return {
      name: "张三",
      user: { name: "张三", age: 18 },
      loading: true,

      // vue2
      obj: {
        name: "李四~",
      },
      // vue2 会进行深度合并
      // obj  {"name":"李四~","age":19}

      // vue3 { name: "李四~" }
    };
  },
  // provide 不支持响应式，想支持响应式的话我们要传对象
  provide() {
    return {
      userName: this.name,
      user: this.user,
    };
  },
  computed: {
    // fullName() {
```

```
    //   return 'xxxxx'
    // }
    fullName: {
      get() {
        return this.$store.state.userName;
        // return '李四'
      },
      set(val) {
        this.$store.commit("SET_NAME", val);
      },
    },
  },

  watch: {
    // name(value) {
    //   this.handlerName()
    // }
    // name: {
    //   immediate: true,
    //   deep: true, //
    //   handler(val, oldValue) {
    //     this.handlerName()
    //   },
    // },
    // this.obj.name = 'xxxx' 这样不会执行
    // this.obj = {name: 'xxx'} 这样才会执行
    // obj(value) {
    //   console.log(' value: ', value)
    // }
    //  和上面等价
    // obj: {
    //   handler(value) {
    //     console.log(" value: ", value)
    //   },
    // },
    // this.obj.name = 'xxxx' 这样去修改也能监听
    // obj: {
    //   deep: true, // 深度监听
    //   immediate: true, // 第一次就用执行这个方法，可以理解为在 created 的时候会执行
handler
```

```
    //   handler(value) {
    //     console.log(" value: ", value)
    //   },
    // },
    //
    // obj: {
    //   deep: true, // 深度监听
    //   immediate: true, // 第一次就用执行这个方法，可以理解为在 created 的时候会执行 handler
    //   handler: 'handlerName',
    // },
    // ==》
    // obj: 'handlerName'
    // '$route.path': {},
    // 'obj.a' : {}
  },

  beforeCreate() {
    console.log("this", this);
  },
  mounted() {
    // this.handlerName()
    this.fullName = "xxxx";

    //  this.fullName '李四'
  },

  methods: {
    handlerName() {
      this.obj.name = "xxxx";
    },
  },
};

```

指令

常用指令

- v-show dispaly none 的切换
- v-if/v-else

- v-html
- v-text
- v-for (vue2 v-for比v-if优先级高，vu3v-if优先级比v-for高)
- v-cloak [v-cloak] {dispaly:none}
- v-once 静态内容
- v-bind => : v-on => @<!--- 可以直接 v-bind="object" v-on="object" --> <Child v-bind="$attrs" v-on="$listeners"> </Child>
- v-model<el-input v-model="keyword"></el-input> <!--- 等价下面这个 --> <el-input :value="keyword" @input="keyword = $event"></el-input>

```
Vue.directive("指令名", {
  // 生命周期
  // 只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
  bind(el, binding, vnode, oldVnode) {
    //
    // binging.value 拿到指令值
    // binding.modifiers 修饰符对象
  },
  // 被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)
  inserted() {},
  update() {},
  componentUpdated() {},
  // 只调用一次，指令与元素解绑时调用
  unbind() {},
});

// 默认绑定 bind update 的生命周期
Vue.directive("指令名", function (el, binding, vnode, oldVnode) {});

```

修饰符

- .lazy、.number、.trim、.enter、.prevent、.self
- .sync<Dialog :visible.sync="visible"></Child> <!--- 等价下面这个 --> <Dialog :visible="visible" @update:visible="visible = $event"></Child>

scoped

加了 scoped 就只作用于当前组件

```
<style scoped></style>

```

渲染规则

```
.a .b {
}
== > .a .b[data-v-xx] {
}
.a /deep/ .b {
}
== > .a[data-v-xxx] .b {
}
.a >>> .b {
}
== > .a[data-v-xxx] .b {
}
.a ::v-deep .b {
}
== > .a[data-v-xxx] .b {
}

```

vue-router

```
// 全局路由守卫
router.beforeEach((to, from, next) => {})
router.afterEach((to, from) => {})

new VueRouter({
    mode: 'hash', // hash | history | abstract
    // 滚动位置
    scrollBehavior(to, from, savedPosition) {
       if (savedPosition) return savedPosition
       return { y: 0 }
        },
    routes: [
        {
            path: '/',
```

```
            // 路由独享守卫
            beforeEnter(to, from, next) {}
        }
    ]
})

// 组件内的路由
beforeRouteEnter(to, from, next) {}
beforeRouteUpdate(to, from, next) {}
beforeRouteLeave(to, from, next) {}

// 跳转
this.$router.push({name: '', path: '', query: {}})
// 路由信息
this.$route.query this.$route.params

```

vuex

state getters mutations actions modules

```
// state
this.$store.state.userInfo;
// getters
this.$store.getters.userInfo;

// mutations
this.$store.commit("SET_USER_INFO", "传递数据");

// actions
this.$store.dispatch("logout").then((res) => {});

// ----------------------------------
// modules > user
// namespaced: true,

// state 拿 name
this.$store.state.user.avatar;
// getters
this.$store.getters.user.avatar;
```

```
20
21 // mutations
22 this.$store.commit("user/SET_TOKEN", "传递数据");
23
24 // actions
25 this.$store.dispatch("user/login").then((res) => {});
26
27 // ----------------------------------
28 // modules > user
29 // namespaced: false,
30
31 // state 拿 name
32 this.$store.state.user.avatar;
33 // getters
34 this.$store.getters.user.avatar;
35
36 // mutations
37 this.$store.commit("SET_TOKEN", "传递数据");
38
39 // actions
40 this.$store.dispatch("login").then((res) => {});
41
```

辅助函数

```
1 mapState, mapGetters, mapMutations, mapActions;
2
```

vue3

> Vue3 的 8 种和 Vue2 的 12 种组件通信，值得收藏 聊一聊 Vue3 的 9 个知识点

Vue3 有哪些变化

- 新增了三个组件：Fragment 支持多个根节点、Suspense 可以在组件渲染之前的等待时间显示指定内容、Teleport 可以让子组件能够在视觉上跳出父组件(如父组件 overflow:hidden)
- 新增指令 v-memo，可以缓存 html 模板，比如 v-for 列表不会变化的就缓存，简单说就是用内存换时间
- 支持 Tree-Shaking，会在打包时去除一些无用代码，没有用到的模块，使得代码打包体积更小
- 新增 Composition API 可以更好的逻辑复用和代码组织，同一功能的代码不至于像以前一样太分散，虽然 Vue2 中可以用 minxin 来实现复用代码，但也存在问题，比如方法或属性名会冲突，代码来源也不清楚等
- 用 Proxy 代替 Object.defineProperty 重构了响应式系统，可以监听到数组下标变化，及对象新增属性，因为监

听的不是对象属性，而是对象本身，还可拦截 apply、has 等 13 种方法

- 重构了虚拟 DOM，在编译时会将事件缓存、将 slot 编译为 lazy 函数、保存静态节点直接复用(静态提升)、以及添加静态标记、Diff 算法使用 最长递增子序列 优化了对比流程，使得虚拟 DOM 生成速度提升 200%
- 支持在 <style></style> 里使用 v-bind，给 CSS 绑定 JS 变量(color: v-bind(str))
- 用 setup 代替了 beforeCreate 和 created 这两个生命周期
- 新增了开发环境的两个钩子函数，在组件更新时 onRenderTracked 会跟踪组件里所有变量和方法的变化、每次触发渲染时 onRenderTriggered 会返回发生变化的新旧值，可以让我们进行有针对性调试
- 毕竟 Vue3 是用 TS 写的，所以对 TS 的支持度更好
- Vue3 不兼容 IE11

vue3 生命周期

选项式 API	Hook inside setup
beforeCreate	Not needed*
created	Not needed*
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeUnmount	onBeforeUnmount
unmounted	onUnmounted
errorCaptured	onErrorCaptured
renderTracked	onRenderTracked
renderTriggered	onRenderTriggered
activated	onActivated
deactivated	onDeactivated

基本代码

main.js

```
// main.js
import { createApp } from "vue";
import App from "./App.vue";

import HelloWorld from "./components/HelloWorld.vue";
const app = createApp(App);

// 全局组件
app.component("HelloWorld", HelloWorld);

// 全局属性
// vue2.0 Vue.prototype.$http
app.config.globalProperties.$http = () => {
  console.log("http ==");
};

app.mount("#app");

```

App.vue

```
<!--- App.vue  -->
<template>
  <div>
    <!-- v-model="xxx"  <==> v-model:modelValue="xxx" -->
    <!-- :value="xxx" @input="xxx = $event" -->
    <!-- value $emit('input', '传递') -->

    <!--
      visible.sync="visible"
      ==>
      :visible="visible" @update:visible="visible = $event"
      -->

    <!-- vue3 把 .sync 去掉，==>
        v-model:visible="visible"
       -->

```

```
    <!--
    <div :ref="setDivRef">
      count: {{ count }}
      <p>
        <button @click="add">+</button>
        <button @click="reduce">-</button>
      </p>
    </div>

    <ul>
      <li>姓名：{{ user.name }}</li>
      <li>年龄：{{ user.age }}</li>
    </ul> -->

    <!-- v-model="num" -->
    <Child
      title="父组件传递的title"
      :modelValue="num"
      @update:modelValue="num = $event"
      @change="onChildChange"
      v-model:visible="visible"
      ref="childRef"
    ></Child>
    <!-- <HelloWorld></HelloWorld> -->
  </div>
</template>

<script>
  import Child from "./Child-setup.vue";
  import { reactive, ref } from "@vue/reactivity";
  import { onMounted, provide } from "@vue/runtime-core";
  export default {
    components: { Child },
    // data() {
    //   return {
    //     msg: '哈哈哈',
    //   }
    // },
    setup() {
      const msg = ref("哈哈哈2"); // => reactive({value: 哈哈哈2 })
```

```
58        const obj = ref({ x: "xx" });
59        console.log(" obj.value: ", obj.value);
60        const user = reactive({ name: "张三", age: 18 });
61        const count = ref(0);
62
63        provide("count", count);
64        provide("http", () => {
65          console.log("$http >>>");
66        });
67
68        const add = () => {
69          count.value++;
70        };
71
72        const reduce = () => {
73          count.value--;
74        };
75
76        const num = ref(1);
77        const visible = ref(false);
78
79        // this.$refs.childRef x
80        // refs
81        // 1. 用字符串
82        const childRef = ref(null);
83        onMounted(() => {
84          console.log(" childRef.value: ", childRef.value);
85        });
86
87        let divRef;
88        const setDivRef = (el) => {
89          console.log(" el: ", el);
90          divRef = el;
91        };
92
93        return {
94          msg,
95          user,
96          count,
```

```
        add,
        reduce,
        num,
        visible,
        childRef,
        setDivRef,
      };
    },

    methods: {
      onChildChange() {},
    },
  };
</script>

<style>
  #app {
    font-family: Avenir, Helvetica, Arial, sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    text-align: center;
    color: #2c3e50;
    margin-top: 60px;
  }
</style>

```

Child-composition （组合式 api）

```
<template>
  <!--
  1. 多个片段， 多个根标签
  2. v-for v-if 优先级变化 v3 v-if > v-for
-->
  <div>
    <button @click="triggerEvent">触发事件</button>

    <div>num2: {{ num2 }}</div>
    <div>count: {{ count }}</div>
```

```
11
12      modelValue: {{ modelValue }}
13      <button @click="add">+</button>
14      <hr />
15      visible: {{ visible }}
16      <button @click="updateVisible">更新visible</button>
17
18      <!--   -->
19      <teleport to="body">
20        <div v-if="visible">对话框</div>
21      </teleport>
22    </div>
23  </template>
24
25  <script>
26    import {
27      computed,
28      inject,
29      onActivated,
30      onBeforeMount,
31      onBeforeUnmount,
32      onBeforeUpdate,
33      onDeactivated,
34      onMounted,
35      onUnmounted,
36      onUpdated,
37      watch,
38      watchEffect,
39    } from "@vue/runtime-core";
40    export default {
41      props: {
42        title: String,
43        modelValue: Number,
44        visible: Boolean,
45      },
46      // computed: {
47      //   num2() {
48      //     return this.modelValue * 2
49      //   }
50      // },
```

```
51      emits: ["change", "update:modelValue", "update:visible"],
52      // 发生在 beforeCreate
53      // attrs 除了 class style,props 之外的属性
54      //
55
56      // watch: {
57      //   title: {
58      //     deep: true, // 深度简单
59
60      //   }
61      // },
62      // 组合式API(composition),  选项式API(options)
63      setup(props, { emit, attrs, slots }) {
64        console.log(" attrs: ", attrs);
65        console.log(" props: ", props);
66
67        // computed
68        const num2 = computed(() => props.modelValue * 2);
69        // const num2 = computed({
70        //   get: () => props.modelValue * 2,
71        //   set: (val) => {
72        //     ssss
73        //   }
74        // })
75
76        //
77        const count = inject("count");
78        console.log(" count: ", count);
79
80        // watch
81        // this.$watch()
82        const unwatch = watch(
83          () => props.modelValue,
84          (newVal, oldValue) => {
85            console.log(" newVal: ", newVal);
86            if (newVal >= 10) {
87              // 取消监听
88              unwatch();
89            }
```

```
90          },
91          {
92            deep: true,
93            // immediate: true
94          }
95        );
96
97        // 自动收集依赖，所以会初始化的时候就执行一次
98        watchEffect(() => {
99          console.log(" props.modelValue: ", props.modelValue);
100       });
101
102       // hooks
103       onBeforeMount(() => {});
104       onMounted(() => {
105         console.log("哈哈哈");
106       });
107       onBeforeUpdate(() => {});
108       onUpdated(() => {});
109       onBeforeUnmount(() => {});
110       onUnmounted(() => {});
111
112       // keep-alive
113       onActivated(() => {});
114       onDeactivated(() => {});
115
116       // methods
117       const triggerEvent = () => {
118         emit("change", "传递的数据");
119       };
120
121       const add = () => {
122         emit("update:modelValue", props.modelValue + 1);
123       };
124       const updateVisible = () => {
125         console.log(" props.visible: ", props.visible);
126         emit("update:visible", !props.visible);
127       };
128
```

```
      return {
        triggerEvent,
        add,
        updateVisible,
        num2,
        count,
      };
    },
    // beforeCreate() {
    //   console.log('beforeCreate')
    // },
    // created() {
    //   console.log('created')
    // },

    // beforeDestroy beforeUnmount
    // destroyed unmounted
  };
</script>

```

Child-setup

```
<template>
  <div>
    <p>title: {{ title }}</p>
    <p>num2: {{ num2 }}</p>
    <p>count: {{ count }}</p>

    <div>
      modelValue: {{ modelValue }}
      <button @click="add">+</button>
    </div>

    <button @click="triggerEvent">触发事件</button>

    <!-- <input type="text" v-model="inputValue"> -->
    <!--   -->
    <input type="text" :value="inputValue" @input="onInputUpdate" />
```

```
17
18      <!-- volar -->
19      <Foo></Foo>
20    </div>
21  </template>
22
23  <!--- vue 3.2.x -->
24  <script setup>
25    import {
26      computed,
27      getCurrentInstance,
28      inject,
29      ref,
30      useAttrs,
31      useSlots,
32    } from "@vue/runtime-core";
33    import Foo from "./foo.vue";
34
35    // props
36    const props = defineProps({
37      title: String,
38      modelValue: Number,
39    });
40    // computed
41    const num2 = computed(() => props.modelValue * 2);
42    const count = inject("count");
43
44    // emit
45    const emit = defineEmits(["change", "update:modelValue", "update:visible"]);
46    const triggerEvent = () => {
47      emit("change", "传递的数据");
48    };
49    const add = () => {
50      emit("update:modelValue", props.modelValue + 1);
51    };
52
53    // 向父组件暴露自己的属性和方法
54    defineExpose({
55      num2,
```

```
    test() {
      console.log("888");
    },
  });

  const attrs = useAttrs();
  console.log(" attrs: ", attrs);
  const solts = useSlots();

  const ctx = getCurrentInstance();

  const http = ctx.appContext.config.globalProperties.$http;
  http("xxx");

  const $http = inject("http");
  $http();

  // $ref: ref(false)

  const inputValue = ref("");

  const onInputUpdate = (event) => {
    console.log(" event: ", event);
    inputValue.value = event.target.value;
  };
</script>

```

项目相关

git

常用命令

https://shfshanyue.github.io/cheat-sheets/git

git pull 和 git featch 的区别

怎么样进行合并，比如把 mater 分支合并到 dev 分支

Webpack 一些核心概念：

> 【万字】透过分析 webpack 面试题，构建 webpack5.x 知识体系

- Entry：入口，指示 Webpack 应该使用哪个模块，来作为构建其内部 依赖图(dependency graph) 的开始。
- Output：输出结果，告诉 Webpack 在哪里输出它所创建的 bundle，以及如何命名这些文件。
- Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
- Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
- Loader：模块代码转换器，让 webpack 能够去处理除了 JS、JSON 之外的其他类型的文件，并将它们转换为有效 模块，以供应用程序使用，以及被添加到依赖图中。
- Plugin：扩展插件。在 webpack 运行的生命周期中会广播出许多事件，plugin 可以监听这些事件，在合适的时机通过 webpack 提供的 api 改变输出结果。常见的有：打包优化，资源管理，注入环境变量。
- Mode：模式，告知 webpack 使用相应模式的内置优化
- hash: 每次构建的生成唯一的一个 hash，且所有的文件 hash 串是一样的
- chunkhash: 每个**入口文件**都是一个 chunk，每个 chunk 是由入口文件与其依赖所构成，**异步加载**的文件也被视为是一个 chunk, **chunkhash**是由每次编译模块，根据模块及其依赖模块构成 chunk 生成对应的 chunkhash, 这也就表明了**每个 chunk 的 chunkhash 值**都不一样， 也就是说每个 chunk 都是独立开来的，互不影响，每个 chunk 的更新不会影响其他 chunk 的编译构建
- contenthash：由文件内容决定，文件变化 contenthash 才会变化，一般配合 mini-css-extract-plugin插件提取出 cssconst MiniCssExtractPlugin = require("mini-css-extract-plugin"); const HTMLWebpackPlugin = require("html-webpack-plugin"); module.exports = { // ... module: { rules: [{ test: /\.css$/, use: [{ // 把 style-loader替换掉，不要使用 style-loader了 loader: MiniCssExtractPlugin.loader, options: { outputPath: "css/", }, }, "css-loader",], },], }, plugins: [// .... new MiniCssExtractPlugin({ filename: "css/[name].[contenthash].css", }),], };

提升 webpack 打包速度

> 一套骚操作下来，webpack 项目打包速度飞升 🚀、体积骤减 ↓ 玩转 webpack，使你的打包速度提升 90% 带你深度解锁 Webpack 系列(优化篇) 学习 Webpack5 之路（优化篇）- 近 7k 字

- 速度分析，可以使用 speed-measure-webpack-plugin
- 提升基础环境，nodejs 版本，webpack 版本
- CDN 分包 html-webpack-externals-plugin, externals
- 多进程、多实例构建 thread-loader happypack(不再维护)
- 多进程并行构建打包uglifyjs-webpack-plugin terser-webpack-plugin
- 缓存: webpack5 内置了cache模块 、babel-loader 的 cacheDirectory 标志、cache-loader, HardSourceWebpackPluginmodule.exports = { // webpack5内置缓存 cache: { type: "filesystem", // 使用文件缓存 }, };
- 构建缩小范围 include,exclude
- 加快文件查找速度resolve.alias,resolve.extensions, module.noParse

- DllPlugin
- babel 配置的优化

webpack 常用 loader，plugin

loader

- babel-loader 将 es6 转换成 es5 , ts-loader、vue-loader
- eslint-loader 配置 enforce: 'pre' 这个 loader 最先执行
- css-loader、style-loader、postcss-loader、less-loader、sass-loader
- file-loader 把文件转换成路径引入, url-loader（比file-loader多了小于多少的能转换成 base64）
- image-loader
- svg-sprite-loader 处理 svg
- thread-loader 开启多进程，会在一个单独的 worker 池（worker pool）中运行
- cache-loader 缓存一些性能开销比较大的 loader 的处理结果

plugin

- html-webpack-plugin 将生成的 css，js 自动注入到 html 文件中，能对 html 文件压缩
- copy-webpack-plugin 拷贝某个目录
- clean-webpack-plugin 清空某个目录
- webpack.HotModuleReplacementPlugin 热重载
- webpack.DefinePlugin 定义全局变量
- mini-css-extract-plugin 提取 CSS 到独立 bundle 文件。 extract-text-webpack-plugin
- optimize-css-assets-webpack-plugin 压缩 css webpack5 推荐css-minimizer-webpack-plugin
- purgecss-webpack-plugin 会单独提取 CSS 并清除用不到的 CSS（会有问题把有用的 css 删除）
- uglifyjs-webpack-plugin ✕（不推荐） 压缩 js、多进程 parallel: true
- terser-webpack-plugin 压缩 js， 可开启多进程压缩、推荐使用module.exports = { optimization: { minimize: true, minimizer: [new TerserPlugin({ parallel: true, // 多进程压缩 }),], }, };
- Happypack ✕（不再维护） 可开启多进程
- HardSourceWebpackPlugin 缓存
- speed-measure-webpack-plugin 打包构建速度分析、查看编译速度
- webpack-bundle-analyzer 打包体积分析
- compression-webpack-plugin gzip 压缩

前端性能优化

> 前端性能优化 24 条建议（2020）

1. 减少 http 请求
2. 使用 http2
3. 静态资源使用 CDN
4. 将 CSS 放在文件头部，JavaScript 文件放在底部

5. 使用字体图标 iconfont 代替图片图标
6. 设置缓存，强缓存，协商缓存
7. 压缩文件，css(MiniCssExtractPlugin),js(UglifyPlugin),html(html-webpack-plugin)文件压缩，清除无用的代码，tree-shaking（需要 es6 的 import 才支持），gzip 压缩(compression-webpack-plugin)
8. splitChunks 分包配置，optimization.splitChunks 是基于 SplitChunksPlugin 插件实现的
9. 图片优化、图片压缩
10. webpack 按需加载代码，hash，contenthash
11. 减少重排重绘
12. 降低 css 选择器的复杂性

babel

> 不容错过的 Babel7 知识

核心库 @babel/core

Polyfill 垫片

CLI 命令行工具 @babel/cli

插件

预设：包含了很多插件的一个组合，@babel/preset-env @babel/preset-react @babel/preset-typescript

polyfill

Babel 默认只转换新的 JavaScript 句法（syntax），而不转换新的 API，比如Iterator、Generator、Set、Map、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转码。

举例来说，ES6 在Array对象上新增了Array.from方法。Babel 就不会转码这个方法。如果想让这个方法运行，可以使用core-js和regenerator-runtime(后者提供 generator 函数的转码)，为当前环境提供一个垫片。

@babel/plugin-transform-runtime

Babel 会使用很小的辅助函数来实现类似 _createClass 等公共方法。默认情况下，它将被添加(inject)到需要它的每个文件中。

如果你有 10 个文件中都使用了这个 class，是不是意味着 _classCallCheck、_defineProperties、_createClass 这些方法被 inject 了 10 次。这显然会导致包体积增大，最关键的是，我们并不需要它 inject 多次。

@babel/plugin-transform-runtime 是一个可以重复使用 Babel 注入的帮助程序，以节省代码大小的插件。

```
npm install --save-dev @babel/plugin-transform-runtime
npm install --save @babel/runtime

```

```
//.babelrc
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "useBuiltIns": "usage", // 配置 polyfill 动态导入
                "corejs": 3 // core-js@3
            }
        ]
    ],
    "plugins": [
        [
            "@babel/plugin-transform-runtime"
        ]
    ]
}

```

浏览器

跨域、同源策略

参考：https://blog.csdn.net/weixin_43745075/article/details/115482227

> 同源策略是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSRF等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

同源策略限制内容有：

- Cookie、LocalStorage、IndexedDB 等存储性内容
- DOM 节点
- AJAX 请求发送后，结果被浏览器拦截了

但是有三个标签是允许跨域加载资源：

```
<img src="XXX">
<link href="XXX">
<script src="XXX">

```

跨域解决方案

1. JSONP：通过动态创建script，再请求一个带参网址实现跨域通信。
2. 开发环境：前端做代理
3. nginx反向代理
4. CORS: 服务端设置 Access-Control-Allow-Origin 即可，前端无须设置，若要带 cookie 请求，前后端都需要设置。
5. websocket---下面的跨域通信、注意只是页面之间的跨域，不是前后端服务跨域，别人问前后端跨域就不要回答下面的了
6. postMessage
7. window.name + iframe
8. document.domain + iframe
9. location.hash + iframe

垃圾回收机制

- 标记清除： 进入环境、离开环境
- 引用计数（不常用）：值被引用的次数， 当引用次数为零时会被清除（缺陷，相互引用的会有问题）

缓存机制

强缓存

> 如果命中强缓存--就不用像服务器去请求

1. Expires 设置时间，过期时间 expires: Tue, 15 Oct 2019 13:30:54 GMT通过本地时间和 expires 比较是否过期，如果过期了就去服务器请求，没有过期的话就直接使用本地的缺点：本地时间可能会更改， 导致缓存出错
2. Cache-Control HTTP1.1 中新增的
    - max-age 最大缓存多少毫秒，列如 Cache-Control: max-age=2592000
    - no-store (每次都要请求，就连协商缓存都不走)表示不进行缓存，缓存中不得存储任何关于客户端请求和服务端响应的内容。每次 由客户端发起的请求都会下载完整的响应内容。Cache-Control: no-store
    - no-cache（默认值）表示不缓存过期的资源，缓存会向源服务器进行有效期确认后处理资源，也许称 为 do-notserve-from-cache-without-revalidation 更合适。浏览器默认开启的是 no-cache，其 实这里也可理解为开启协商缓存
    - public 和 privatepublic 与 private 是针对资源是否能够被代理服务缓存而存在的一组对立概念当我们为资源

设置了 pubile，那么它既可以被浏览器缓存也可被代理服务器缓存。设置为private 的时候，则该资源只能被浏览器缓存，其中默认值是 private。

- max-age 和 s-maxages-maxage 只适用于供多用户使用的公共服务器上(如 CND cache)，并只对 public 缓存有效

协商缓存

> 需要向服务器请求，如果没有过期，服务器会返回 304，

1. ETag 和 If-None-Match 唯一标识

- 服务器响应 ETag 值，浏览器携带的是 If-None-Match（携带的是上一次响应的 ETag），服务拿到这 If-None-Match 值后判断过期--> 没有过期 304，并且返回 ETag二者的值都是服务器为每份资源分配的唯一标识字符串。• 浏览器请求资源，服务器会在响应报文头中加入 ETag 字段。资源更新的时候，服务端的ETag 值也随之更新• 浏览器再次请求资源，会在请求报文头中添加 If-None-Match 字段，它的值就是上次响应报文中的 ETag 值，服务器会对比 ETag 和 If-None-Match 的值是否一致。如果不一致，服务器则接受请求，返回更新后的资源，状态码返回 200；如果一致，表明资源未更新，则返回状态码 304，可继续使用本地缓存，值得注意的是此时响应头会加上 ETag 字段，即使它没有变化
- **Last-Modified 和 If-Modified-Since 时间戳** 缺点： 某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说 1s 内修改了 N 次)，If-Modified-Since 可查到的是秒级，这种修改无法判断

预编译

四部曲

1. 创建AO对象
2. 找形参和变量声明，将变量和形参名作为AO的属性名，值为undefined
3. 将实参值和形参值相统一
4. 在函数体里面找到函数声明，值赋予函数体

```
// 预编译
function foo(test /* 形参 */) {
  console.log(" test: ", test); // function(){}
  var test = 2;
  var str = "bs";
  console.log(" test: ", test); // 2
  // 函数声明
  function test() {}
  // 函数表达式
  str = function () {};
  console.log(" test: ", test); // 2
}

```

```
// 预编译 四部曲
// 1. 创建AO对象
// 2. 找形参和变量声明，将变量和形参名作为AO的属性名，值为undefined
// 3. 将实参值和形参值相统一
// 4. 在函数体里面找到函数声明，值赋予函数体

// AO {
//   test: undefined
//   str: undefined
// }
// AO {
//   test: 1
//   str: undefined
// }
// AO {
//   test: 1
//   str: function() {}
// }

foo(1 /*实参*/);

function foo(a, b, c) {
  console.log(a);
  console.log(b);
  var a = "222";
  function a() {}
  var b = function () {};
  console.log(a);
  console.log(b);

  console.log(" a, b, c: ", a, b, c);
}
// AO {
//   a : '222',
//   b : function() {},
//   c : 3
// }
foo(1, 2, 3);

```

```
var a = 22;
// let a = 22
// window.a ==> 22

```

全局

1. 创建 GO 对象==window
2. 变量声明，将变量作为 GO 的属性名，值为undefined
3. 找到函数声明，值赋予函数体

event-loop(事件循环)

> 一次弄懂 Event Loop（彻底解决此类面试问题）

JS是单线程的，为了防止一个函数执行时间过长阻塞后面的代码，所以会先将同步代码压入执行栈中，依次执行，将异步代码推入异步队列，异步队列又分为宏任务队列和微任务队列，因为宏任务队列的执行时间较长，所以微任务队列要优先于宏任务队列。微任务队列的代表就是，Promise.then，MutationObserver，宏任务的话就是setImmediate setTimeout setInterval

MacroTask（宏任务）*

- script全部代码、setTimeout、setInterval、setImmediate（浏览器暂时不支持，只有 IE10 支持，具体可见 MDN）、I/O、UI Rendering。

MicroTask（微任务）

- Process.nextTick（Node独有）、Promise.then、Object.observe(废弃)、MutationObserver

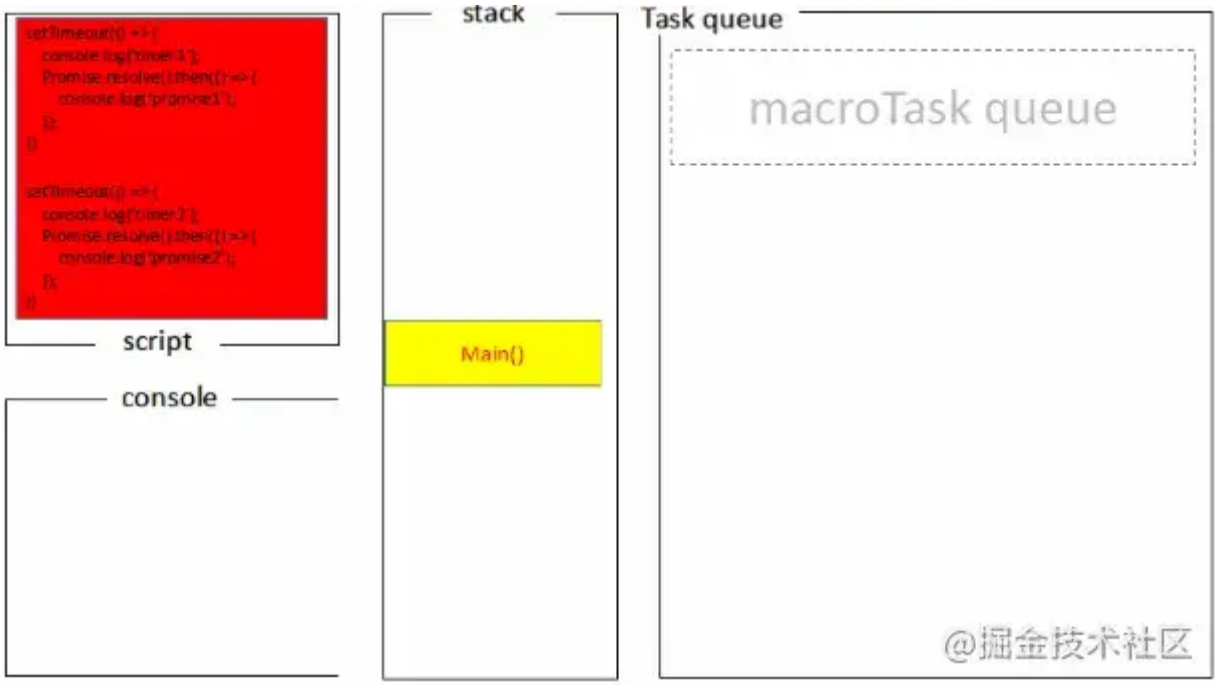
浏览器中

> 执行完一个宏任务，会执行所有的微任务

```
console.log("script start");

setTimeout(function () {
  console.log("setTimeout");
}, 0);

new Promise((resolve) => {
  console.log("promise1");
  resolve();
}).then(function () {
  console.log("promise2");
});
console.log("script end");

```

执行结果

```
script start
promise1
script end
promise2
setTimeout

```



nodejs 中

> 在 11 之前的版本，会在每个阶段之后执行所有的微任务 在 11 版本及之后，会每执行完一个宏任务，就会清空所用的微任务（和浏览器保存一致）

```
new Promise((resolve) => {
  console.log("new Promise 1");
  resolve();
}).then(() => {
  console.log("new Promise then");
});

setTimeout(() => {
  console.log("timer1");
  new Promise((resolve) => {
    console.log("timer1 new Promise");
```

```
    resolve();
  }).then(() => {
    console.log("timer1 new Promise then");
  });
  Promise.resolve().then(() => {
    console.log("timer1 Promise then");
  });
});

setTimeout(() => {
  console.log("timer2");
  Promise.resolve().then(() => {
    console.log("timer2 Promise then");
  });
});

console.log("start end");

```

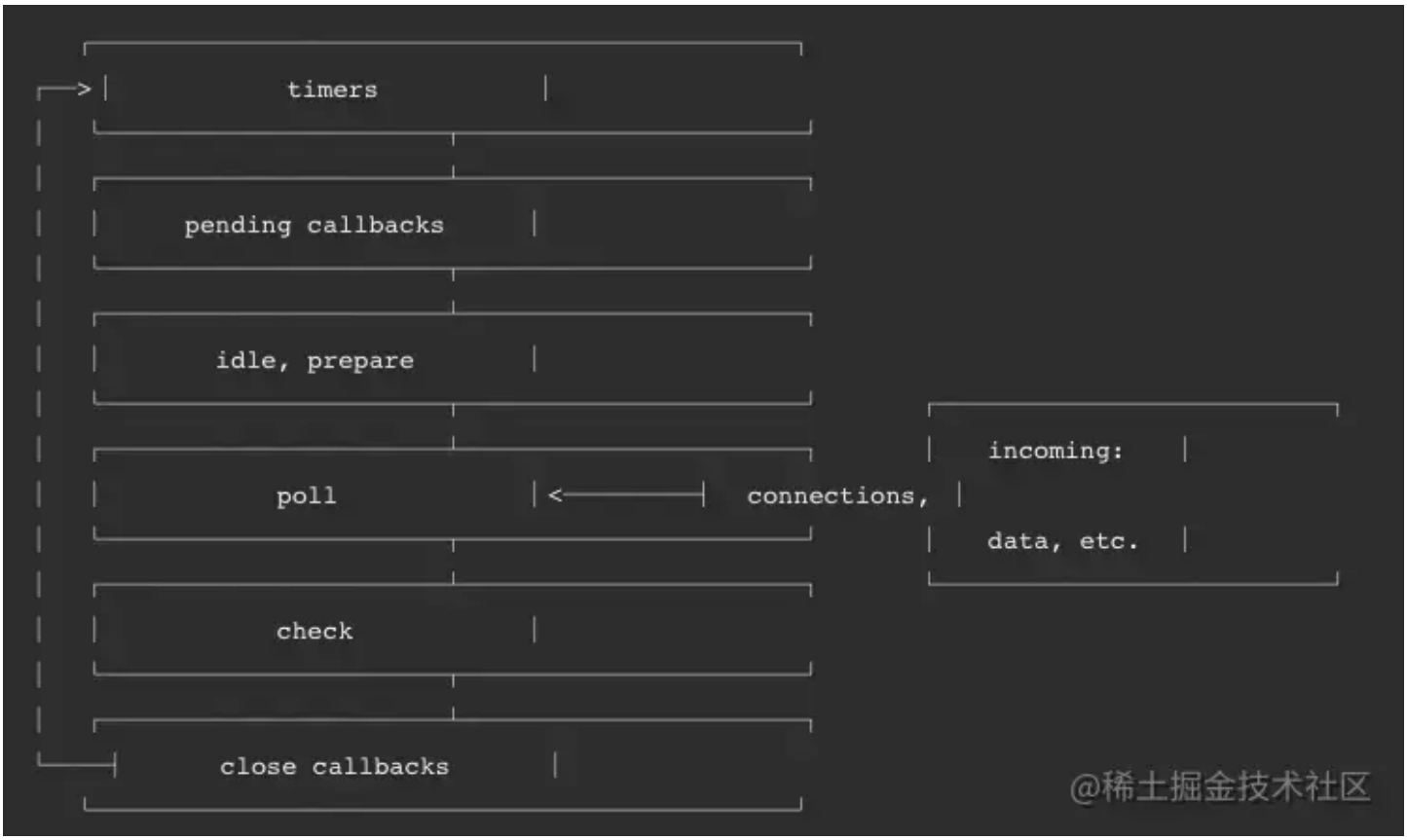
在 node11 版本之前（不包含 11）

```
new Promise 1
start end
new Promise then
timer1
timer1 new Promise
timer2
timer1 new Promise then
timer1 Promise then
timer2 Promise then

```

在 node11 版本及之后

```
new Promise 1
start end
new Promise then
timer1
timer1 new Promise
timer1 new Promise then
timer1 Promise then
```

```
timer2
timer2 Promise then

```



Node的Event loop一共分为 6 个阶段，每个细节具体如下：

1. timers: 执行setTimeout和setInterval中到期的callback。
2. pending callback: 上一轮循环中少数的callback会放在这一阶段执行。
3. idle, prepare: 仅在内部使用。
4. poll: 最重要的阶段，执行pending callback，在适当的情况下回阻塞在这个阶段。
5. check: 执行setImmediate(setImmediate()是将事件插入到事件队列尾部，主线程和事件队列的函数执行完成之后立即执行setImmediate指定的回调函数)的callback。
6. close callbacks: 执行close事件的callback，例如socket.on('close'[,fn])或者http.server.on('close, fn)。

网络

常见状态码

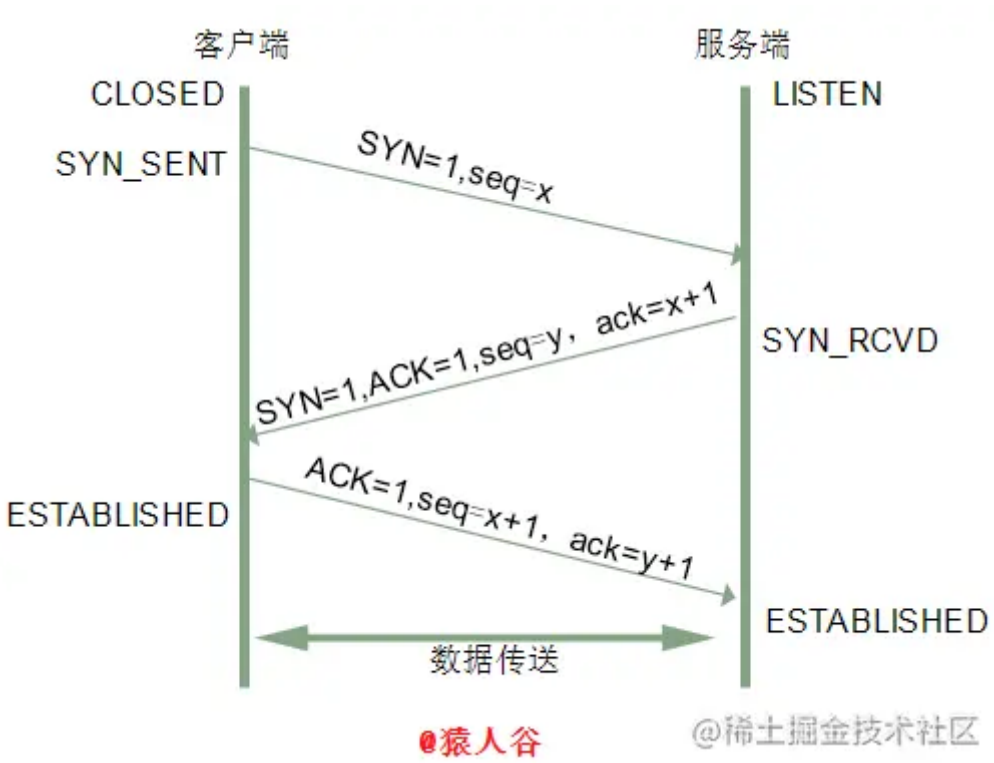
```
1xx: 接受，继续处理
200: 成功，并返回数据
201: 已创建
202: 已接受
203: 成为，但未授权
```

```
204: 成功，无内容
205: 成功，重置内容
206: 成功，部分内容
301: 永久移动，重定向
302: 临时移动，可使用原有 URI
304: 资源未修改，可使用缓存
305: 需代理访问
400: 请求语法错误
401: 要求身份认证
403: 拒绝请求
404: 资源不存在
500: 服务器错误


```

TCP

> 面试官，不要再问我三次握手和四次挥手

三次握手



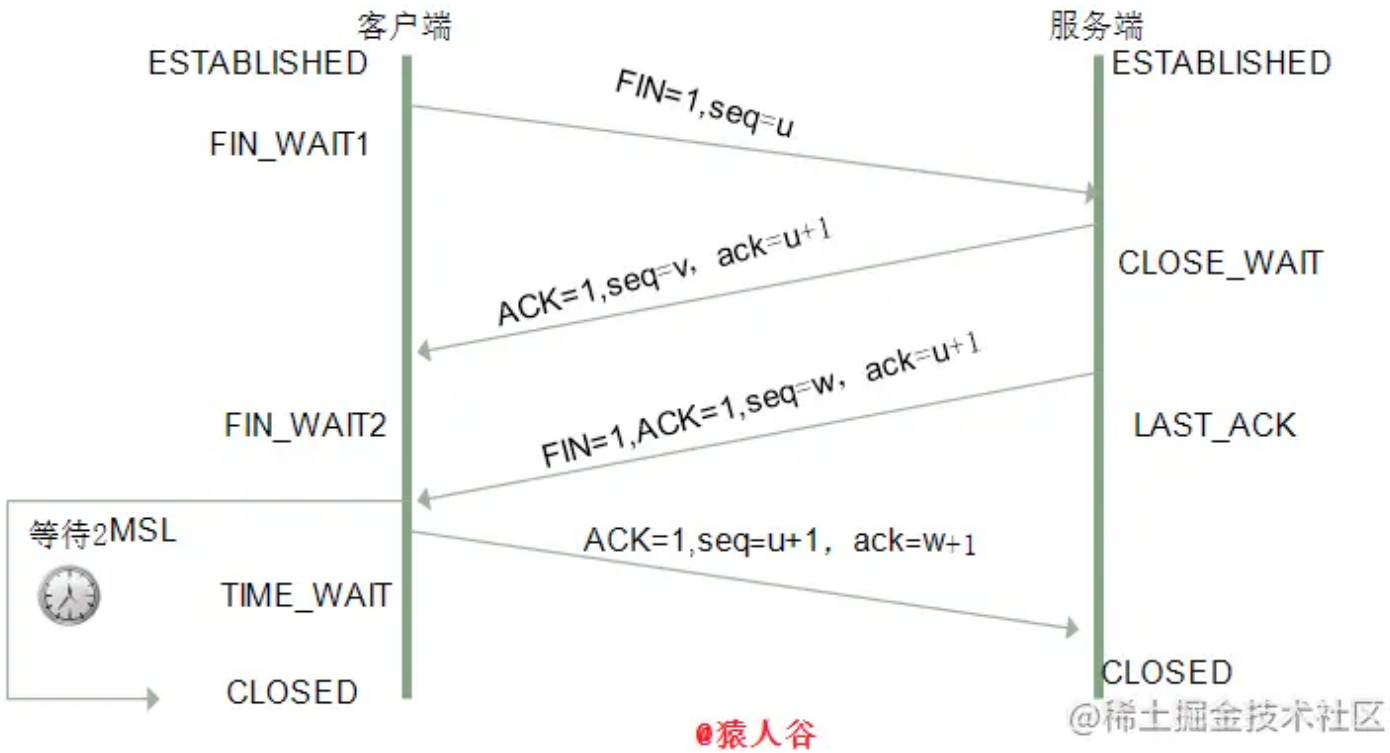
为什么需要三次握手，两次不可以吗

```
为了防止失效的连接请求又传送到主机，因而产生错误。
如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请
```

求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于 TCP 的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

四次挥手



挥手为什么需要四次?

因为当服务端收到客户端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中**ACK 报文是用来应答的，SYN 报文是用来同步的。**但是关闭连接时，当服务端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉客户端，"你发的 FIN 报文我收到了"。只有等到我服务端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四次挥手。

2MSL等待状态

TIME_WAIT 状态也成为2MSL等待状态。每个具体 TCP 实现必须选择一个报文段最大生存时间 MSL（Maximum Segment Lifetime），它是任何报文段被丢弃前在网络内的最长时间。这个时间是有限的，因为 TCP 报文段以 IP 数据报在网络内传输，而 IP 数据报则有限制其生存时间的 TTL 字段。

对一个具体实现所给定的 MSL 值，处理的原则是：当 TCP 执行一个主动关闭，并发回最后一个 ACK，该连接必须在 TIME_WAIT 状态停留的时间为 2 倍的 MSL。这样可让 TCP 再次发送最后的 ACK 以防这个 ACK 丢失（另一端超时并重发最后的 FIN）。
这种 2MSL 等待的另一个结果是这个 TCP 连接在 2MSL 等待期间，定义这个连接的插口（客户的 IP 地址和端口号，服务器的 IP 地址和端口号）不能再被使用。这个连接只能在 2MSL 结束后才能再被使用。

HTTP 版本

HTTP/1.0
最早的 http 只是使用一些简单的网页上和网络请求上，每次请求都打开一个新的 TCP 连接， 收到响应后立即断开连接
HTTP/1.1
缓存处理，HTTP/1.1 更多的引入了缓存策略，如 Cache-Control，Entity tag，If-Unmodified-Since, If-Match, If-None-Match 等
宽带优化及网络连接的使用，在 HTTP/1.0 中，存在一些浪费宽带的现象，列如客户端只需要某个对象的一部分，而服务器把整个对象都送过来了，并且不支持断点续传，HTTP1.1 则
在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（PartialContent），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
错误通知的管理，在 HTTP/1.1 中新增了 24 个错误状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
Host 头处理，在 HTTP1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。HTTP1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有 Host 头域会报告一个错误（400 Bad Request）
长连接， HTTP/1.1 默认开启持久连接（默认：keep-alive），在一个 TCP 连接上可以传递多个 HTTP 请求和响应，减少了建立与关闭连接的消耗和延迟
HTTP/2.0
在 HTTP/2.0 中，有两个重要的概念，分别是帧（frame） 和 流（stream），帧代表数据传输的最小单位，每个帧都有序列标识标明该帧属于哪个流，流也就是多个帧组成的数据流，每个流表示一个请求。
新的二进制格式： HTTP/1.x 的解析是基于文本的。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认 0 和 1 的组合。基于这种考虑 HTTP2.0 的协议解析决定采用二进制格式，实现方便且健壮。
多路复用： HTTP/2.0 支持多路复用，这是 HTTP/1.1 持久连接的升级版。多路复用，就是在一个 TCP 连接中存在多个条流，也就是多个请求，服务器则可以通过帧中的标识知道该帧属于哪个流（即请求），通过重新排序还原请求。多路复用允许并发多个请求，每个请求及该请求的响应不需要等待其他的请求或响应，避免了线头阻塞问题。这样某个请求任务耗时严

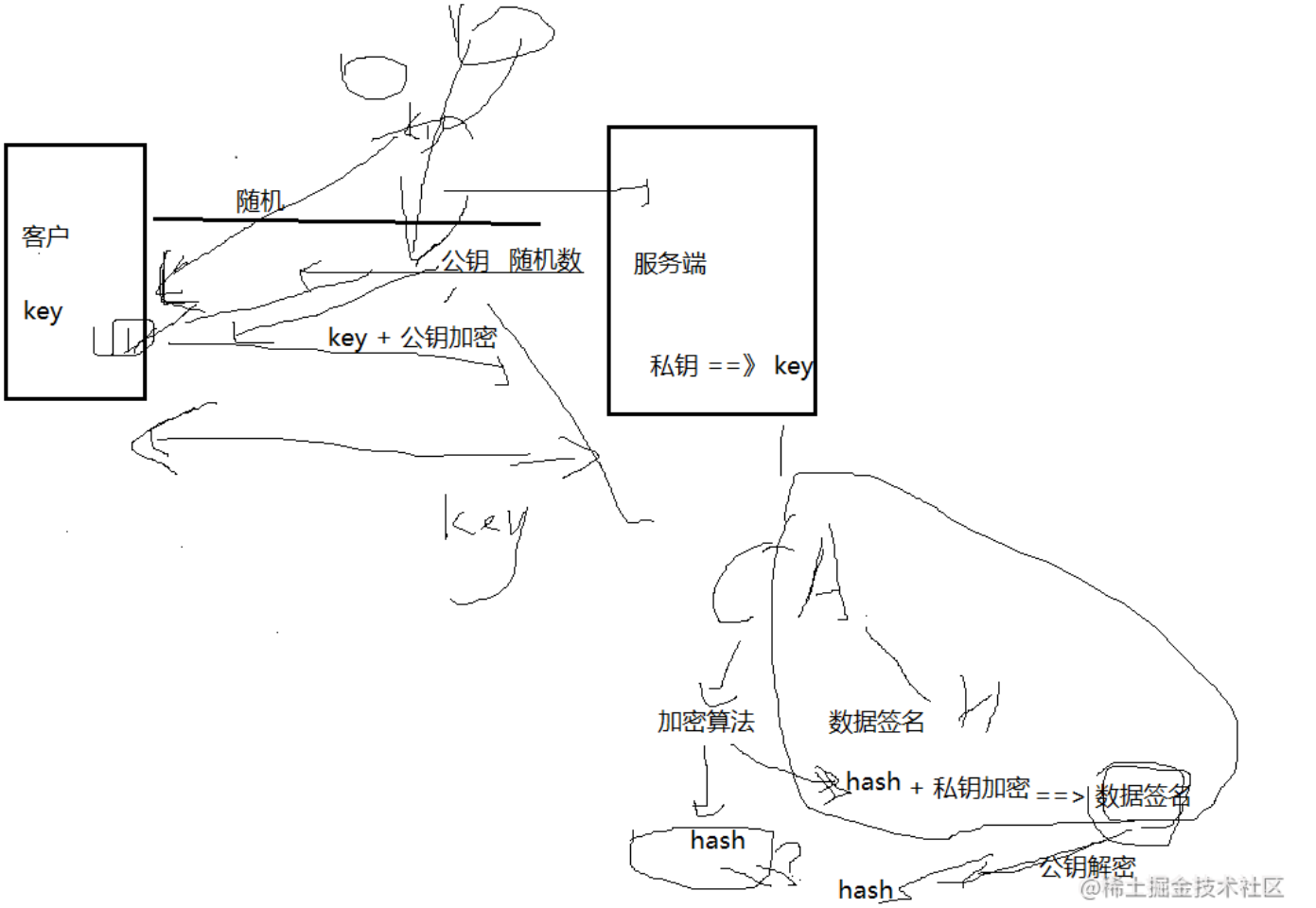
重，不会影响到其它连接的正常执行,极大的提高传输性能。
头部压缩： 对前面提到的 HTTP/1.x 的 header 带有大量信息，而且每次都要重复发送，HTTP/2.0 使用 encoder 来减少需要传输的头部大小，通讯双方各自 cache 一份头部 fields 表，既避免了重复头部的传输，又减小了需要传输的大小。
服务端推送： 服务端推送指把客户端所需要的 css/js/img 资源伴随着 index.html 一起发送到客户端，省去了客户端重复请求的步骤（从缓存中取）。正因为没有发起请求，建立连接等操作，所以静态资源通过服务端推送的方式极大的提升了速度HTTP/3.0
HTTP/2.0 使用了多路复用，一般来说同一域名下只需要使用一个 TCP 连接。但当这个连接中出现了丢包的情况，会导致整个 TCP 都要开始等待重传，也就导致了后面所有的数据都阻塞了。
避免包阻塞： 多个流的数据包在 TCP 连接上传输时，若一个流中的数据包传输出现问题，TCP 需要等待该包重传后，才能继续传输其它流的数据包。但在基于 UDP 的 QUIC 协议中，不同的流之间的数据传输真正实现了相互独立互不干扰，某个流的数据包在出问题需要重传时，并不会对其他流的数据包传输产生影响。
快速重启会话: 普通基于 tcp 的连接，是基于两端的 ip 和端口和协议来建立的。在网络切换场景，例如手机端切换了无线网，使用 4G 网络，会改变本身的 ip，这就导致 tcp 连接必须重新创建。而 QUIC 协议使用特有的 UUID 来标记每一次连接，在网络环境发生变化的时候，只要 UUID 不变，就能不需要握手，继续传输数据。
HTTP2.0 的多路复用和 HTTP1.X 中的长连接有什么区别?
HTTP/1.* 一次请求-响应，建立一个连接，用完关闭；每一个请求都要建立一个连接；
HTTP/1.1 在一个 TCP 连接上可以传递多个 HTTP 请求和响应，后面的请求等待前面的请求返回才能获得执行机会，一旦有某个请求超时，后续请求只能被阻塞，毫无办法，也就是常说的线头阻塞
HTTP/2.0 多个请求可同时在一个连接上并行执行.某个请求任务耗时严重，不影响其他连接的正常执行。

https(http + ssl/tls)

http: 最广泛网络协议，BS 模型，浏览器高效。
https: 安全版，通过 SSL 加密，加密传输，身份认证，密钥
1 https 相对于 http 加入了 ssl 层, 加密传输, 身份认证;
2 需要到 ca 申请收费的证书;
3 安全但是耗时多，缓存不是很好;
4 注意兼容 http 和 https;
5 连接方式不同, 端口号也不同, http 是 80, https 是 443

- 明文： 普通的文本
- 密钥：把明文加密的那个钥匙
- 密文： 把明文加密明文+密钥==>密文==>密钥==解密=>明文
- 对称加密 解密的 key（密钥）和解密的 key 是同一个 3 + 1

- 非对称加密 私钥和公钥



手写

> 10 个常见的前端手写功能，你全都会吗

最近面试 2022 年 3 月问到了很多手写，这个一定要准备下

防抖

```
function debounce(func, wait, immediate) {
  let timeout;
  return function () {
    const context = this;
    const args = [...arguments];
    if (timeout) clearTimeout(timeout);
    if (immediate) {
      const callNow = !timeout;
      timeout = setTimeout(() => {
```

```
        timeout = null;
      }, wait);
      if (callNow) func.apply(context, args);
    } else {
      timeout = setTimeout(() => {
        func.apply(context, args);
      }, wait);
    }
  };
}

```

节流

```
function throttle(fn, wait) {
  let pre = 0;
  return function (...args) {
    let now = Date.now();
    if (now - pre >= wait) {
      fn.apply(this, args);
      pre = now;
    }
  };
}

```

event bus 事件总线 | 发布订阅模式

```
// event bus

class EventBus {
  constructor() {
    this.events = {};
  }
  on(name, callback) {
    const { events } = this;
    if (!events[name]) {
      events[name] = [];
    }
```

```
    events[name].push(callback);
  }
  emit(name, ...args) {
    const handlers = this.events[name];
    handlers &&
      handlers.forEach((fn) => {
        fn.apply(this, args);
      });
  }
  off(name, callback) {
    const { events } = this;
    if (!events[name]) return;
    events[name] = events[name].filter((fn) => fn !== callback);
  }
  once(name, callback) {
    const handler = function () {
      callback.apply(this, arguments);
      this.off(name, handler);
    };
    this.on(name, handler);
  }
  clear() {
    this.events = {};
  }
}

```

数据偏平化

```
// 数据偏平化
function flatter(arr) {
  return arr.reduce((prev, curr) => {
    return Array.isArray(curr) ? [...prev, ...flatter(curr)] : [...prev, curr];
  }, []);
}

```

手写 new

```
// 手写 new
function myNew(ctr, ...args) {
  const obj = Object.create(ctr.prototype);
  myNew.target = ctr;
  const result = ctr.apply(obj, args);
  if (
    result &&
    (typeof result === "function" || typeof result === "function")
  ) {
    return result;
  }
  return obj;
}

```

call、bind

```
Function.prototype.myCall = function (context, ...args) {
  context = context || window;
  const fn = Symbol();
  context[fn] = this;
  const result = context[fn](...args);
  delete context[fn];
  return result;
};

//bind实现要复杂一点  因为他考虑的情况比较多 还要涉及到参数合并(类似函数柯里化)

Function.prototype.myBind = function (context, ...args) {
  if (!context || context === null) {
    context = window;
  }
  // 创造唯一的key值  作为我们构造的context内部方法名
  let fn = Symbol();
  context[fn] = this;
  let _this = this;
  //  bind情况要复杂一点
  const result = function (...innerArgs) {
    // 第一种情况 :若是将 bind 绑定之后的函数当作构造函数，通过 new 操作符使用，则不绑定传入的 this，而是将 this 指向实例化出来的对象
```

```
    // 此时由于new操作符作用  this指向result实例对象  而result又继承自传入的_this 根据原型链知识可得出以下结论
    // this.__proto__ === result.prototype   //this instanceof result =>true
    // this.__proto__.__proto__ === result.prototype.__proto__ === _this.prototype; //this instanceof _this =>true
    if (this instanceof _this === true) {
      // 此时this指向指向result的实例  这时候不需要改变this指向
      this[fn] = _this;
      this[fn](...[...args, ...innerArgs]); //这里使用es6的方法让bind支持参数合并
    } else {
      // 如果只是作为普通函数调用  那就很简单了 直接改变this指向为传入的context
      context[fn](...[...args, ...innerArgs]);
    }
  };
  // 如果绑定的是构造函数 那么需要继承构造函数原型属性和方法
  // 实现继承的方式: 使用Object.create
  result.prototype = Object.create(this.prototype);
  return result;
};

```

异步控制并发数

```
function limitRequest(requests, limit = 3) {
  requests = requests.slice();
  return new Promise((resolve, reject) => {
    let count = 0;
    const len = requests.length;
    while (limit > 0) {
      start();
      limit--;
    }

    function start() {
      const promiseFn = requests.shift();

      promiseFn?.().finally(() => {
        count++; // 一定要通过 count 判断、不能通过 requests.length 判断是否为空，这样不对的
        if (count === len) {
          // 最后一个
```

```
          resolve();
        } else {
          start();
        }
      });
    }
  });
}
// 测试
const arr = [];
for (let value of "12345") {
  arr.push(() => fetch(`https://www.baidu.com/s?ie=UTF-8&wd=${value}`));
}
limitRequest(arr);

```

算法/特殊题目

最近面试 2022 年 3 月问到了很多手写，这个一定要准备下、下面都是我被问到的

台阶问题

有 N 个台阶，一步可以走一梯或者两梯，请问有多少种走法

解答：https://blog.csdn.net/z1832729975/article/details/123836190

有效括号

> 我面试才几家，这个有两家都问到了 力扣原题

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。

左括号必须以正确的顺序闭合。

```
示例 1：
输入：s = "()"
输出：true


示例 2：
```

```
输入：s = "()[]{}"
输出：true

示例 3：
输入：s = "("
输出：false
```

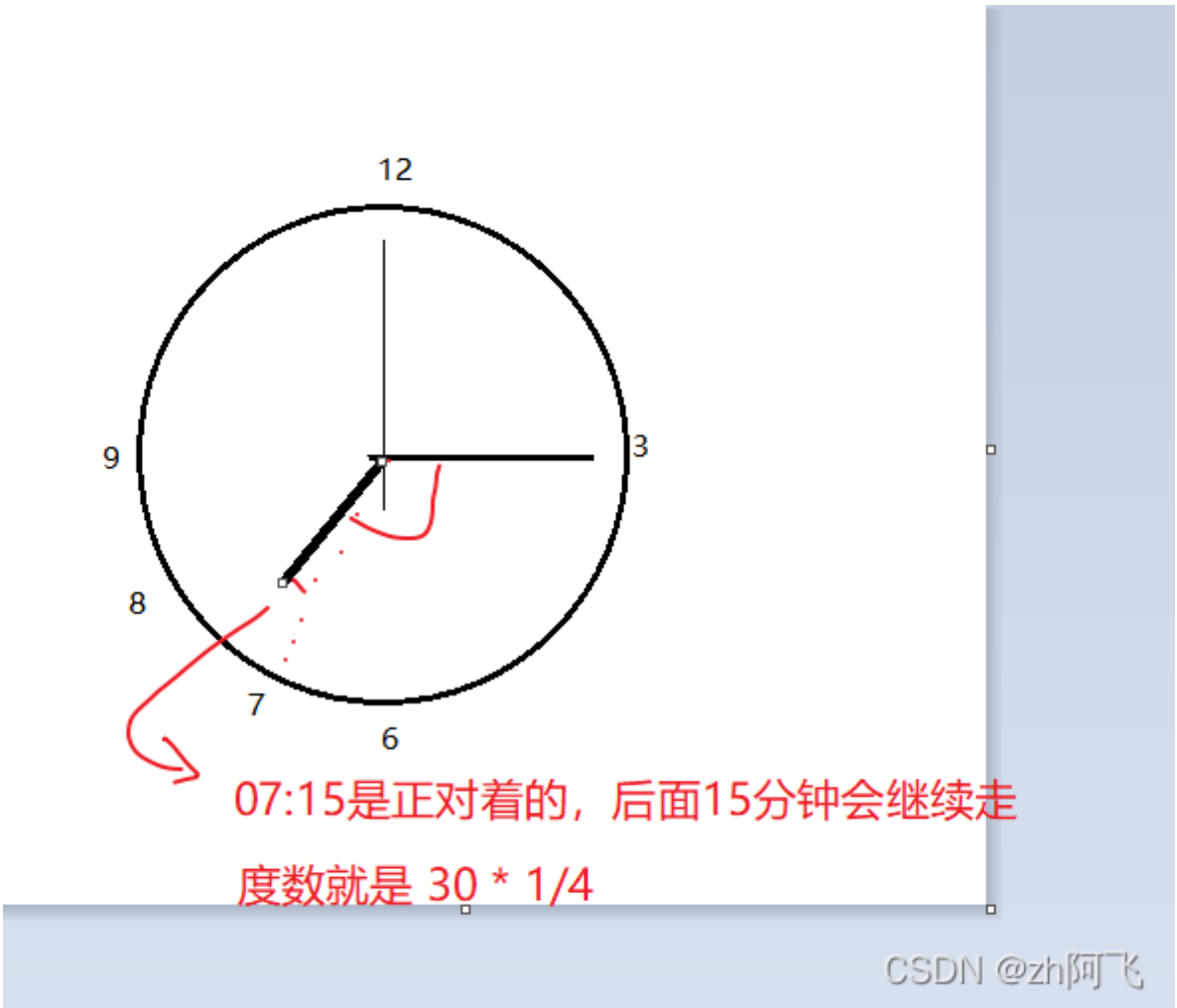
实现

我们可以通过栈来实现、当遇到左括号的时候就把对应的右括号值push到栈中，否则的话我们就把栈定的元素pop和当前字符比较是否相等，不相信的话直接返回 false

```
/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function (s) {
  if (!s) return true;
  const leftFlags = {
    "(": ")",
    "{": "}",
    "[": "]",
  };
  const stack = [];
  for (let chart of s) {
    const flag = leftFlags[chart];
    // 是左括号
    if (flag) {
      stack.push(flag);
    }
    // 是右括号
    else if (chart !== stack.pop()) {
      return false;
    }
  }
  return stack.length === 0;
};

```

现在时间 07:15，请问分针和时针的夹角是多少

先看看时钟，要了解 07:15 在哪，这个不知道在哪就尴尬了



画图，结果如下

7 点 15 分时针和分针所形成的角是

120 + 30*1/4=127.5

这题需要注意时针还好继续走，不会固定，不然容易被坑

写 IP 地址的正则表达式

分析ip地址

让 a==1 && a==2 && a==3 为 true

> 因为这个是 ==，会存在隐式类型转换

- 利用对象Symbol.toStringvalueOftoStringvar a = { value: 1, // 这三种方法都可以，优先级也是这个顺序 [Symbol.toString]() { return a.value++; }, // valueOf() { // return a.value++ // }, // toString() { // return a.value++ // } };
- 利用数组a.valueOf = a.shift; // 一样有 //a[Symbol.toPrimitive] = a.shift //a.toString = a.shift
- 通过Object.defineProperty拦截let value = 1; Object.defineProperty(window, "a", { get() { return value++; }, });
- 通过 Proxy 拦截let value = 1; const a = new Proxy({}, { get() { return function () { return value++; }; }, });

typescript

> 建议先把基础的东西学会，推荐看这篇文章、基础的学会，就能应付大多数的 typescript 面试了 2021 typescript 史上最强学习入门文章(2w 字)

const和readonly的区别

const常量：表示这个变量的指针地址不可以在改变，可以更改对象内部的属性

readonly只读：指针地址不可以改变，并且对象内部的属性也不可以改变

1. const 用于变量，readonly 用于属性
2. const 在运行时检查，readonly 在编译时检查
3. 使用 const 变量保存的数组，可以使用 push，pop 等方法。但是如果使用ReadonlyArray声明的数组不能使用 push，pop 等方法。

type和interface的区别

参考：https://juejin.cn/post/7018805943710253086#heading-63

type-类型别名

interface-接口

1. 接口重名会合并、类型别名重名会报错interface Person { name: string; } interface Person { age: number; } // 这个接口合并，变成下面的 interface Person { name: string; age: number; } type Aanimal = { name: string }; type Aanimal = { age: number }; // 会报错、重名了
2. 两者都可以用来描述对象或函数的类型，但是语法不同interfaceinterface Point { x: number; y: number; } interface SetPoint { (x: number, y: number): void; } typetype Point = { x: number; y: number; }; type SetPoint =

(x: number, y: number) => void;

3. 类型别名可以为任何类型引入名称。例如基本类型，联合类型等// primitive type Name = string; // object type PartialPointX = { x: number }; type PartialPointY = { y: number }; // union type PartialPoint = PartialPointX | PartialPointY; // tuple type Data = [number, string]; // dom let div = document.createElement("div"); type B = typeof div;
4. 扩展两者的扩展方式不同，但并不互斥。接口可以扩展类型别名，同理，类型别名也可以扩展接口。接口的扩展就是继承，通过 extends 来实现。类型别名的扩展就是交叉类型，通过 & 来实现。接口扩展接口interface PointX { x: number; } interface Point extends PointX { y: number; }

类型别名扩展类型别名

```
type PointX = {
  x: number;
};

type Point = PointX & {
  y: number;
};

```

接口扩展类型别名

```
type PointX = {
  x: number;
};
interface Point extends PointX {
  y: number;
}

```

类型别名扩展接口

```
interface PointX {
  x: number;
}
type Point = PointX & {
  y: number;
};

```

keyof 和 typeof 关键字的作用?

> keyof 索引类型查询操作符 获取索引类型的属性名，构成联合类型。 typeof 获取一个变量或对象的类型。

unknown, any 的区别

> unknown 类型和 any 类型类似。与 any 类型不同的是。unknown 类型可以接受任意类型赋值，但是 unknown 类型赋值给其他类型前，必须被断言

> CSS3 有哪些新特性？CSS3 新特性详解

1、圆角效果；2、图形化边界；3、块阴影与文字阴影；4、使用 RGBA 实现透明效果；5、渐变效果；6、使用“@Font-Face”实现定制字体；7、多背景图；8、文字或图像的变形处理；9、多栏布局；10、媒体查询等。

```
1、颜色：新增RGBA、HSLA模式
2、文字阴影：（text-shadow）
3、边框：圆角（border-radius）边框阴影：box-shadow
4、盒子模型：box-sizing
5、背景：background-size,background-origin background-clip(削弱)
6、渐变：linear-gradient(线性渐变):
eg: background-image: linear-gradient(100deg, #237b9f, #f2febd);
radial-gradient (径向渐变)
7、过渡：transition可实现动画
8、自定义动画： animate@keyfrom
9、媒体查询：多栏布局@media screen and (width:800px)
10、border-image
11、2D转换:transform:translate(x,y) rotate(x,y)旋转 skew(x,y)倾斜 scale(x,y)缩放
12、3D转换
13、字体图标：Font-Face
14、弹性布局：flex
```

```
17
```

css 选择器

id 选择器（ #myid）
类选择器（.myclassname）
标签选择器（div, h1, p）相邻选择器（h1 + p）
子选择器（ul > li）后代选择器（li a）
属性选择器（a[rel = "external"]）
伪类选择器（a: hover, li:nth-child）
通配符选择器（ * ）

```
1          !Important > 行内式 > id > 类/伪类/属性 > 标签选择器   >  全局
2       （对应权重：无穷大∞ > 1000> 100 >  10         >  1    >       0）
3
```

盒模型

一个盒子，会有 content,padding,border,margin 四部分，
标准的盒模型的宽高指的是 content 部分
ie 的盒模型的宽高包括了 content+padding+border
我们可以通过 box-sizing 修改盒模型，box-sizing border-box content-box

margin 合并

在垂直方向上的两个盒子，他们的 margin 会发生合并（会取最大的值），比如上边盒子设置margin-bottom:20px，下边盒子设置margin-top:30px;，那么两个盒子间的间距只有30px，不会是50px
解决 margin 合并，我们可以给其中一个盒子套上一个父盒子，给父盒子设置 BFC

margin 塌陷

效果： 一个父盒子中有一个子盒子，我们给子盒子设置margin-top:xxpx结果发现会带着父盒子一起移动（就效果和父盒子设置margin-top:xxpx的效果一样）
解决： 1、给父盒子设置 border，例如设置border:1px solid red; 2、给父盒子设置 BFC

BFC

> 块级格式化上下文（block format context）

BFC 的布局规则 *

- 内部的 Box 会在垂直方向，一个接一个地放置。
- Box 垂直方向的距离由 margin 决定。属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠。

- 每个盒子（块盒与行盒）的 margin box 的左边，与包含块 border box 的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。
- BFC 的区域不会与 float box 重叠。
- BFC 就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。
- 计算 BFC 的高度时，浮动元素也参与计算。

触发 BFC 的条件 *

- 根元素 html
- float 的值不是 none。
- position 的值 absoute、fixed
- display 的值是 inline-block、table-cell、flex、table-caption 或者 inline-flex
- overflow 的值不是 visible

解决什么问题

1. 可以用来解决两栏布局BFC 的区域不会与 float box 重叠.left { float: flet; } .right { overflow: hidden; }
2. 解决 margin 塌陷和 margin 合并问题
3. 解决浮动元素无法撑起父元素

flex

> 设为 Flex 布局以后，子元素的 float、clear 和 vertical-align 属性将失效

什么是 rem、px、em 区别

> rem 是一个相对单位，rem 的是相对于 html 元素的字体大小，没有继承性 em 是一个相对单位，是相对于父元素字体大小有继承性 px 是一个“绝对单位”，就是 css 中定义的像素，利用 px 设置字体大小及元素的宽高等，比较稳定和精确。

响应式布局

响应式布局有哪些实现方式？什么是响应式设计？响应式设计的基本原理是什么？

> 1.百分比布局，但是无法对字体，边框等比例缩放 2.弹性盒子布局 display:flex 3.rem 布局，1rem=html 的 font-size 值的大小 css3 媒体查询 @media screen and(max-width: 750px){} 5.vw+vh 6.使用一些框架（bootstrap，vant） 什么是响应式设计：响应式网站设计是一个网站能够兼容多个终端，智能地根据不同设备环境进行相对应的布局 响应式设计的基本原理：基本原理是通过媒体查询检测不同的设备屏幕尺寸设置不同的 css 样式 页面头部必须有 meta 声明的

布局

- 两栏布局,左边定宽，右边自适应
- 三栏布局、圣杯布局、双飞翼布局

水平垂直居中

方法一：给父元素设置成弹性盒子，子元素横向居中，纵向居中

方法二：父相子绝后，子部分向上移动本身宽度和高度的一半，也可以用 transfrom:translate(-50%,-50%)（最常用方法）

方法三：父相子绝，子元素所有定位为 0，margin 设置 auto 自适应

iframe 有哪些缺点?

iframe 是一种框架，也是一种很常见的网页嵌入方

iframe 的优点：

1. iframe 能够原封不动的把嵌入的网页展现出来。
2. 如果有多个网页引用 iframe，那么你只需要修改 iframe 的内容，就可以实现调用的每一个页面内容的更改，方便快捷。
3. 网页如果为了统一风格，头部和版本都是一样的，就可以写成一个页面，用 iframe 来嵌套，可以增加代码的可重用。
4. 如果遇到加载缓慢的第三方内容如图标和广告，这些问题可以由 iframe 来解决。

iframe 的缺点：

1. 会产生很多页面，不容易管理。
2. iframe 框架结构有时会让人感到迷惑，如果框架个数多的话，可能会出现上下、左右滚动条，会分散访问者的注意力，用户体验度差。
3. 代码复杂，无法被一些搜索引擎索引到，这一点很关键，现在的搜索引擎爬虫还不能很好的处理 iframe 中的内容，所以使用 iframe 会不利于搜索引擎优化。
4. 很多的移动设备（PDA 手机）无法完全显示框架，设备兼容性差。
5. iframe 框架页面会增加服务器的 http 请求，对于大型网站是不可取的。现在基本上都是用 Ajax 来代替 iframe，所以 iframe 已经渐渐的退出了前端开发。

link @import 导入 css

> link 是 XHTML 标签，除了加载 CSS 外，还可以定义 RSS 等其他事务； @import 属于 CSS 范畴， 只能加载 CSS。 link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载。link 无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持。 link 支持使用 Javascript 控制 DOM 去改变样式；而@import 不支持。

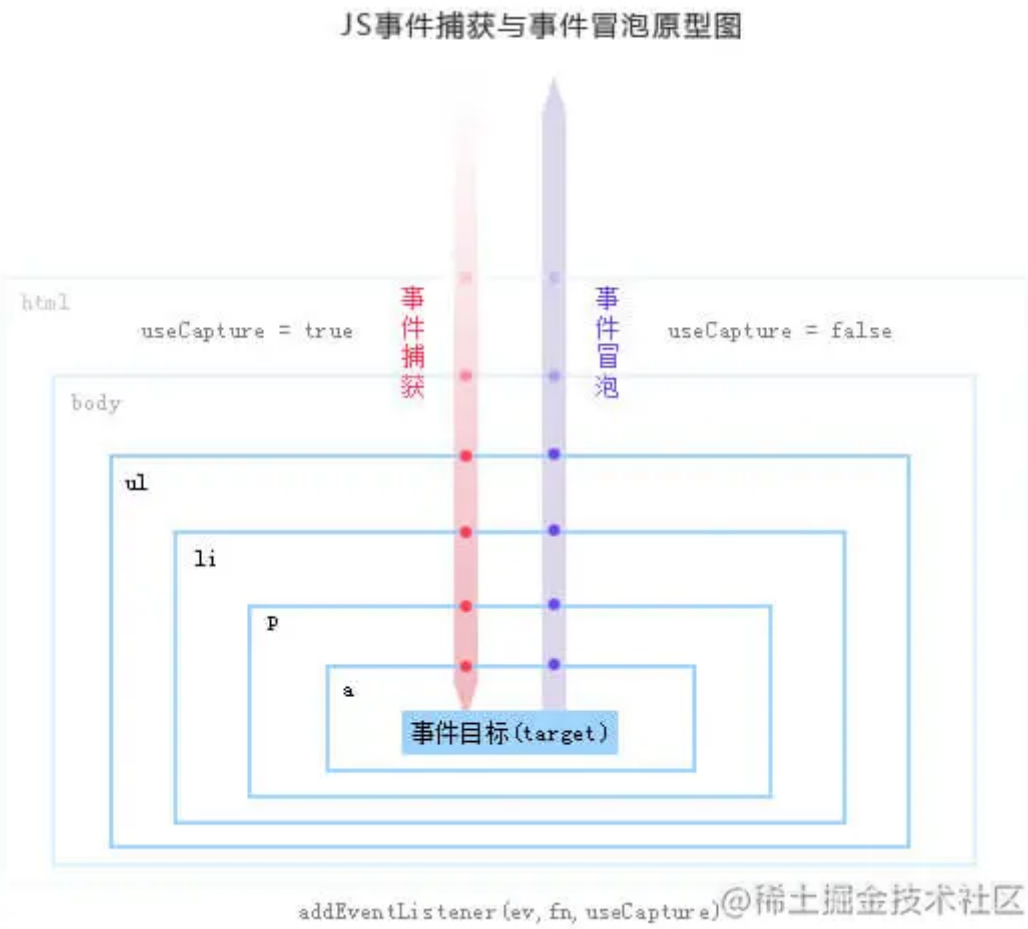
DOM 事件机制/模型

> DOM0 级模型、IE 事件模型、DOM2 级事件模型

就比如用户触发一个点击事件，有一个触发的过程

事件捕获-阶段（从上大小，从外到内）--->处于目标事件-阶段---->事件冒泡-阶段（从下到上，从内到外）

```
window.addEventListener(
  "click",
  function (event) {
    event = event || window.event /*ie*/;
    const target = event.target || event.srcElement; /*ie*/ // 拿到事件目标

    // 阻止冒泡
    // event.stopPropagation()
    // event.cancelBubble=true; // ie

    // 阻止默认事件
    // event.preventDefault();
    // event.returnValue=false; // ie
  },
  /* 是否使用捕获，默认是fasle, */ fasle
);

```



事件委托

简介：事件委托指的是，不在事件的发生地（直接 dom）上设置监听函数，而是在其父元素上设置监听函数，通过事件冒泡，父元素可以监听到子元素上事件的触发，通过判断事件发生元素 DOM 的类型，来做出不同的响应。

举例：最经典的就是 ul 和 li 标签的事件监听，比如我们在添加事件时候，采用事件委托机制，不会在 li 标签上直接添加，而是在 ul 父元素上添加。

好处：比较合适动态元素的绑定，新添加的子元素也会有监听函数，也可以有事件触发机制

如果需要手动写动画，你认为最小时间间隔是多久

> 多数显示器默认频率是 60Hz，即 1 秒刷新 60 次，所以理论上最小间隔为 1/60＊1000ms = 16.7ms

::before和:after中双冒号和单冒号有什么区别

单冒号(:)用于 CSS3 伪类，双冒号(::)用于 CSS3 伪元素。 ::before 就是以一个子元素的存在，定义在元素主体内容之前的一个伪元素。并不存在于 dom 之中，只存在在页面之中。 :before 和 :after 这两个伪元素，是在 CSS2.1 里新出现的。起初，伪元素的前缀使用的是单 冒号语法，但随着 Web 的进化，在 CSS3 的规范里，伪元素的语法被修改成使用双冒号，成 为::before ::after

CSS sprites 精灵图

CSS Sprites 其实就是把网页中一些背景图片整合到一张图片文件中，再利用 CSS 的 “background-image”， “ background-repeat ” ， “ background-position” 的组合进行背景定位，background-position 可以用数字能精确的定位出背景图片的位置。这样可以减少很多图片请 求的开销，因为请求耗时比较长；请求虽然可以并发，但是也有限制，一般浏览器都是 6 个

重排和重绘

重绘（repaint 或 redraw）：当盒子的位置、大小以及其他属性，例如颜色、字 体大小等都确定下来之后，浏览器便把这些原色都按照各自的特性绘制一遍，将 内容呈现在页面上。重绘是指一个元素外观的改变所触发的浏览器行为，浏览器 会根据元素的新属性重新绘制，使元素呈现新的外观。

触发重绘的条件：改变元素外观属性。如：color，background-color 等。 注意：table 及其内部元素可能需要多次计算才能确定好其在渲染树中节点的属性值，比同等元素要多花两倍时间，这就是我们尽量避免使用 table 布局页面的 原因之一。

重排（重构/回流/reflow）：当渲染树中的一部分(或全部)因为元素的规模尺寸， 布局，隐藏等改变而需要重新构建, 这就称为回流(reflow)。每个页面至少需要 一次回流，就是在页面第一次加载的时候。

重绘和重排的关系：在回流的时候，浏览器会使渲染树中受到影响的部分失效， 并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕 中，该过程称为重绘。所以，重排必定会引发重绘，但重绘不一定会引发重排。

JavaScript

js 数据类型

8 中, ES6出的 Symbol BigInt

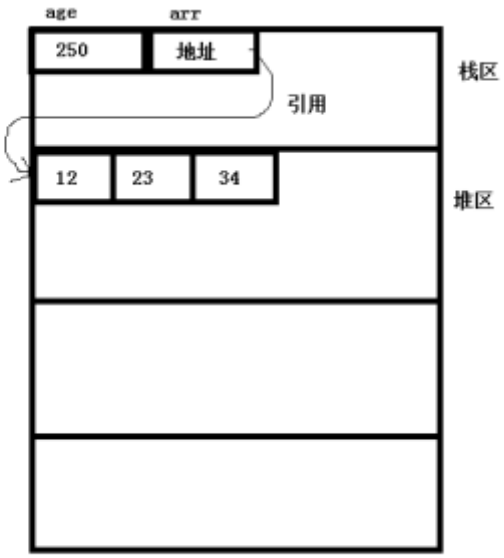
```
Number String Boolean undefined null Object Symbol BigInt

```

js 的基本数据类型和复杂数据类型的区别（在堆和栈中，赋值时的不同,一个拷贝值一个拷贝地址）

基本类型和引用类型在内存上存储的区别

```
function test(){
      var age = 250; //值类型

      var arr = new Array(12,23,34);

}
```

内存的地址，就是个编号



null 与 undefined 的异同

相同点:

- Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null

不同点:

- null 转换成数字是 0, undefined 转换数字是NaN
- undefined 代表的含义是未定义， null 代表的含义是空对象。
- typeof null 返回'object'，typeof undefined 返回'undefined'
- null == undefined; // true null === undefined; // false
- 其实 null 不是对象，虽然 typeof null 会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，然而 null 表示为全零，所以将它错误的判断为 object 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

说说 JavaScript 中判断数据类型的几种方法

typeof

- typeof一般用来判断基本数据类型，除了判断 null 会输出"object"，其它都是正确的
- typeof判断引用数据类型时，除了判断函数会输出"function",其它都是输出"object"

instanceof

> Instanceof 可以准确的判断引用数据类型，它的原理是检测构造函数的prototype属性是否在某个实例对象的原型链上， 不能判断基本数据类型

```
// instanceof 的实现
function instanceofOper(left, right) {
  const prototype = right.prototype;
  while (left) {
    if ((left = left.__proto__) === prototype) {
      return true;
```

```
    }
  }
  return false;
}
// let obj  = {}
// Object.getPrototypeOf(obj) === obj.__proto__ ==> true

```

```
// 实现 instanceof 2
function myInstanceof(left, right) {
  // 这里先用typeof来判断基础数据类型，如果是，直接返回false
  if (typeof left !== "object" || left === null) return false;
  // getProtypeOf是Object对象自带的API，能够拿到参数的原型对象
  let proto = Object.getPrototypeOf(left);
  while (true) {
    if (proto === null) return false;
    if (proto === right.prototype) return true; //找到相同原型对象，返回true
    proto = Object.getPrototypeof(proto);
  }
}

```

Object.prototype.toString.call() 返回 [object Xxxx] 都能判断

深拷贝和浅拷贝

```
let obj = { b: "xxx" };
let arr = [{ a: "ss" }, obj, 333];

// 赋值
let arr2 = arr;
// 浅拷贝-只拷贝了一层，深层次的引用还是存在
// Object.assign(), ...扩展运算符，slice等
let arr2 = arr.slice();
let arr2 = [...arr];
obj.b = "222"; // arr2[1].b => 222
// arr[2] = 4444 ==> arr2[2] ===> 333

// 深拷贝
// 1. 最简单的，JSON.stringify，但这个有问题，看下面有说明
```

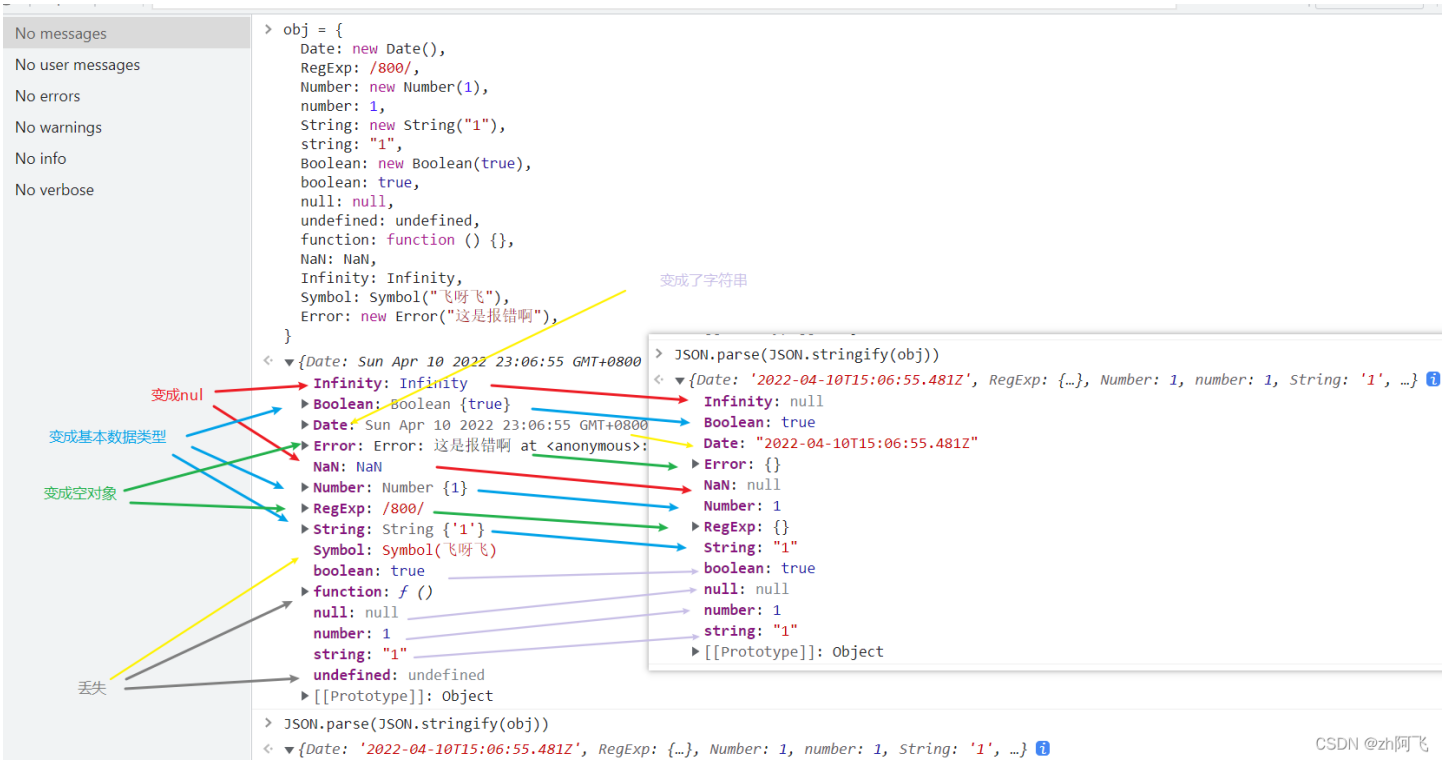
```
let arr2 = JSON.parse(JSON.stringify(arr));

// 2. 自己封装，要递归处理

```

实现深拷贝-简单版

```
export function deepClone(obj, map = new Map()) {
  if (!obj && typeof obj !== "object") return obj;
  if (obj instanceof Date) return new Date(obj);
  if (obj instanceof RegExp) return new RegExp(obj.source, obj.flags);

  if (map.get(obj)) {
    //  如果有循环引用、就返回这个对象
    return map.get(obj);
  }

  const cloneObj = obj.constructor(); // 数组的就是[],对象就是{}

  map.set(obj, cloneObj); // 缓存对象，用于循环引用的情况

  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      cloneObj[key] = deepClone(obj[key], map);
    }
  }

  return cloneObj;
}

```

JSON.stringify 问题

1. 如果有循环引用就报错
2. Symbol、function、undefined会丢失
3. 布尔值、数字、字符串的包装对象会转换成原始值
4. NaN、Infinity 变成 null
5. Date类型的日期会变成字符串
6. RegExp、Error被转换成了空对象 {}



模块化

- commonjs// 由nodejs实现 const fs = require("fs"); module.exports = {};
- ESM// 由es6实现 import $ from "jquery"; export default $;
- AMD（异步加载模块）// 由RequireJS实现 define(["juqery", "vue"], function ($, Vue) { // 依赖必须一开始就写好 $("#app"); new Vue({}); });
- CMD// 由SeaJS 实现 define(function (require, exports, module) { var a = require("./a"); a.doSomething(); // .... var b = require("./b"); // 依赖可以就近书写 b.doSomething(); // ... });
- UMD (通用加载模块)(function (global, factory) { typeof exports === 'object' && typeof module !== 'undefined' ? module.exports = factory() : typeof define === 'function' && define.amd ? define(factory) : (global = global || self, global.Vue = factory()); }(this, function () { 'use strict';

AMD 和 CMD 的区别有哪些

> https://blog.csdn.net/qq_38912819/article/details/80597101

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）
2. CMD 推崇依赖就近，AMD 推崇依赖前置

CommonJS 与 ES6 Module 的差异

CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

- CommonJS 模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。
- ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，

ES6 的import有点像 Unix 系统的“符号连接”，原始值变了，import加载的值也会跟着变。因此，ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

- 运行时加载: CommonJS 模块就是对象；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。
- 编译时加载: ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，import时采用静态命令的形式。即在import时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”。

CommonJS 加载的是一个对象（即 module.exports 属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

JS 延迟加载的方式

JavaScript 会阻塞 DOM 的解析，因此也就会阻塞 DOM 的加载。所以有时候我们希望延迟 JS 的加载来提高页面的加载速度。

- 把 JS 放在页面的最底部
- script 标签的 defer 属性：脚本会立即下载但延迟到整个页面加载完毕再执行。该属性对于内联脚本无作用 (即没有 「src」 属性的脚本）。
- Async 是在外部 JS 加载完成后，浏览器空闲时，Load 事件触发前执行，标记为 async 的脚本并不保证按照指定他们的先后顺序执行，该属性对于内联脚本无作用 (即没有 「src」 属性的脚本）。
- 动态创建 script 标签，监听 dom 加载完毕再引入 js 文件

call、apply 、bind

> call，apply，bind 都是改变 this 指向，bind 不会立即执行，会返回的是一个绑定 this 的新函数 面试官问：能否模拟实现 JS 的 call 和 apply 方法

```
obj.call(this指向, 参数1,  参数2)ss
obj.apply(this指向, [参数1,  参数2])

function fn(age) {
    console.log(this, age)
}
const obj = {name:''}
const result = fn.bind(obj) // bind会返回一个新的函数
result(20)

```

```
// 实现一个 apply
Function.prototype.myApply = function (context) {
  context = context || window;
```

```
  const fn = Symbol();
  context[fn] = this;
  var res = context[fn](...arguments[1]);
  delete context[fn];
  return res;
};

```

实现一个 bind

```
// 最终版 删除注释 详细注释版请看上文
Function.prototype.bind =
  Function.prototype.bind ||
  function bind(thisArg) {
    if (typeof this !== "function") {
      throw new TypeError(this + " must be a function");
    }
    var self = this;
    var args = [].slice.call(arguments, 1);
    var bound = function () {
      var boundArgs = [].slice.call(arguments);
      var finalArgs = args.concat(boundArgs);
      if (this instanceof bound) {
        if (self.prototype) {
          function Empty() {}
          Empty.prototype = self.prototype;
          bound.prototype = new Empty();
        }
        var result = self.apply(this, finalArgs);
        var isObject = typeof result === "object" && result !== null;
        var isFunction = typeof result === "function";
        if (isObject || isFunction) {
          return result;
        }
        return this;
      } else {
        return self.apply(thisArg, finalArgs);
      }
    };
    return bound;
  };
```

```
32
```

防抖

> debounce 所谓防抖，就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

```
function debounce(func, wait, immediate) {
  let timeout;
  return function () {
    const context = this;
    const args = [...arguments];
    if (timeout) clearTimeout(timeout);
    if (immediate) {
      const callNow = !timeout;
      timeout = setTimeout(() => {
        timeout = null;
      }, wait);
      if (callNow) func.apply(context, args);
    } else {
      timeout = setTimeout(() => {
        func.apply(context, args);
      }, wait);
    }
  };
}

```

节流

> 就是指连续触发事件但是在 n 秒中只执行一次函数

```
function throttle(fn, wait) {
  let pre = 0;
  return function (...args) {
    let now = Date.now();
    if (now - pre >= wait) {
      fn.apply(this, args);
      pre = now;
    }
```

```
    };
}

```

闭包

闭包是指有权访问另一个函数作用域中的变量的函数 ——《JavaScript 高级程序设计》
当函数可以记住并访问所在的词法作用域时，就产生了闭包，
即使函数是在当前词法作用域之外执行 ——《你不知道的 JavaScript》

- 闭包用途：
  a. 能够访问函数定义时所在的词法作用域(阻止其被回收)
  b. 私有化变量
  c. 模拟块级作用域
  d. 创建模块
- 闭包缺点：会导致函数的变量一直保存在内存中，过多的闭包可能会导致内存泄漏

原型、原型链(高频)

原型: 对象中固有的__proto__属性，该属性指向对象的prototype原型属性。
原型链: 当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是Object.prototype所以这就是我们新建的对象为什么能够使用toString()等方法的原因。
特点: JavaScript对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

this 指向、new 关键字

this对象是是执行上下文中的一个属性，它指向最后一次调用这个方法的对象，在全局函数中，this等于window，而当函数被作为某个对象调用时，this 等于那个对象。 在实际开发中，this 的指向可以通过四种调用模式来判断。

1. 函数调用，当一个函数不是一个对象的属性时，直接作为函数来调用时，this指向全局对象。
2. 方法调用，如果一个函数作为一个对象的方法来调用时，this指向这个对象。
3. 构造函数调用，this指向这个用new新创建的对象。
4. 第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。apply接收参数的是数组，call接受参数列表，`` bind方法通过传入一个对象，返回一个this绑定了传入对象的新函数。这个函数的this指向除了使用new `时会被改变，其他情况下都不会改变。

new

> 面试官问：能否模拟实现 JS 的 new 操作符

1. 首先创建了一个新的空对象
2. 设置原型，将对象的原型设置为函数的prototype对象。
3. 让函数的this指向这个对象，执行构造函数的代码（为这个新对象添加属性）
4. 判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

```
// new 操作符的实现
function newOperator(ctor) {
  if (typeof ctor !== "function") {
    throw "newOperator function the first param must be a function";
  }
  newOperator.target = ctor;
  var newObj = Object.create(ctor.prototype);
  var argsArr = [].slice.call(arguments, 1);
  var ctorReturnResult = ctor.apply(newObj, argsArr);
  var isObject =
    typeof ctorReturnResult === "object" && ctorReturnResult !== null;
  var isFunction = typeof ctorReturnResult === "function";
  if (isObject || isFunction) {
    return ctorReturnResult;
  }
  return newObj;
}

```

作用域、作用域链、变量提升

作用域负责收集和维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。(全局作用域、函数作用域、块级作用域)。 作用域链就是从当前作用域开始一层一层向上寻找某个变量，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链。

继承(含 es6)、多种继承方式

```
function Animal(name) {
  // 属性
  this.name = name || "Animal";
  // 实例方法
  this.sleep = function () {
    console.log(this.name + "正在睡觉！");
  };
```

```
}
// 原型方法
Animal.prototype.eat = function (food) {
  console.log(this.name + "正在吃：" + food);
};

```

（1）第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

```
// 原型链继承
function Cat() {}
Cat.prototype = new Animal("小黄"); // 缺点 无法实现多继承 来自原型对象的所有属性被所有实例共享
Cat.prototype.name = "cat";

```

（2）第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

```
// 借用构造函数继承
function Cat() {
  Animal.call(this, "小黄");
  // 缺点 只能继承父类实例的属性和方法，不能继承原型上的属性和方法。
}

```

（3）第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

（4）第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 Object.create() 方法就是原型式继承的实现。缺点与原型链方式相同。

```
function object(o) {
  function F() {}
  F.prototype = o;
```

```
  return new F();
}

```

（5）第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是我们的自定义类型时。缺点是没有办法实现函数的复用。

```
function createAnother(original) {
  var clone = object(original); //通过调用object函数创建一个新对象
  clone.sayHi = function () {
    //以某种方式来增强这个对象
    alert("hi");
  };
  return clone; //返回这个对象
}

```

（6）第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

```
function extend(subClass, superClass) {
  var prototype = object(superClass.prototype); //创建对象
  prototype.constructor = subClass; //增强对象
  subClass.prototype = prototype; //指定对象
}

```

类型转换

大家都知道 JS 中在使用运算符号或者对比符时，会自带隐式转换，规则如下:

-、*、/、%： 一律转换成数值后计算

+：

- 数字 + 字符串 = 字符串， 运算顺序是从左到右
- 数字 + 对象， 优先调用对象的 valueOf -> toString
- 数字 + boolean/null -> 数字
- 数字 + undefined -> NaN
- [1].toString() === '1' 内部调用 .join 方法

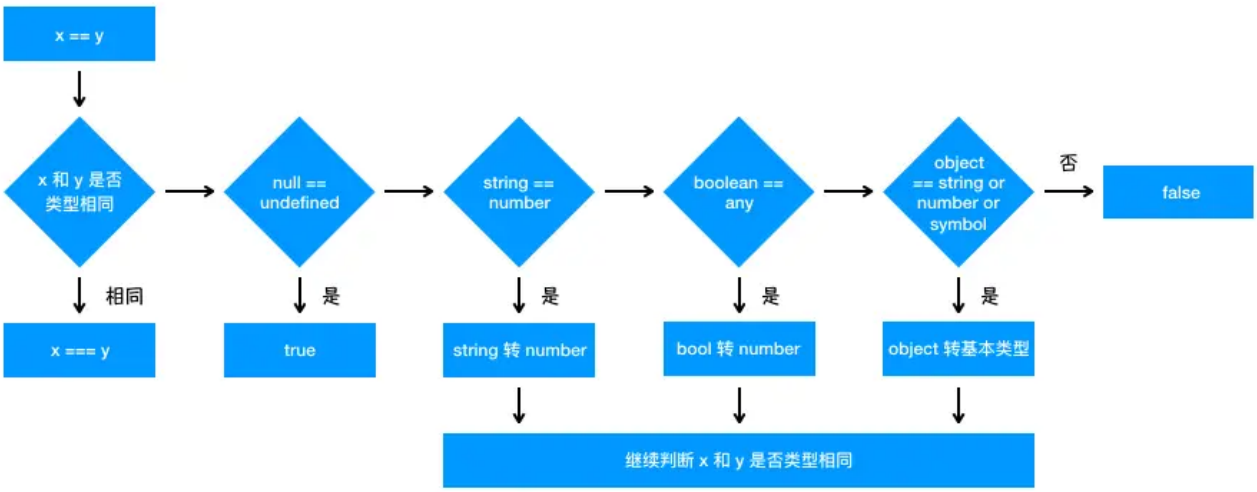
- {}.toString() === '[object object]'
- NaN !== NaN 、+undefined 为 NaN

Object.is()与比较操作符==、===的区别?

- ==会先进行类型转换再比较
- ===比较时不会进行类型转换，类型不同则直接返回 false
- Object.is()在===基础上特别处理了NaN,-0,+0,保证-0 与+0 不相等，但 NaN 与 NaN 相等

==操作符的强制类型转换规则

- 字符串和数字之间的相等比较，将字符串转换为数字之后再进行比较。
- 其他类型和布尔类型之间的相等比较，先将布尔值转换为数字后，再应用其他规则进行比较。
- null 和 undefined 之间的相等比较，结果为真。其他值和它们进行比较都返回假值。
- 对象和非对象之间的相等比较，对象先调用 ToPrimitive 抽象操作后，再进行比较。
- 如果一个操作值为 NaN ，则相等比较返回 false （ NaN 本身也不等于 NaN ）。
- 如果两个操作值都是对象，则比较它们是不是指向同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 true，否则，返回 false。



ES6

1. 新增 Symbol 类型 表示独一无二的值，用来定义独一无二的对象属性名;
2. const/let 都是用来声明变量,不可重复声明，具有块级作用域。存在暂时性死区，不存在变量提升。(const 一般用于声明常量);
3. 变量的解构赋值(包含数组、对象、字符串、数字及布尔值,函数参数),剩余运算符(...rest);
4. 模板字符串(${data});
5. ...扩展运算符(数组、对象);;

6. 箭头函数;
7. Set 和 Map 数据结构;
8. Proxy/Reflect;
9. Promise;
10. async 函数;
11. Class;
12. Module 语法(import/export)。

let/const

> const声明一个只读的常量。一旦声明，常量的值就不能改变 https://es6.ruanyifeng.com/#docs/let

var 在全局作用域中声明的变量会变成全局变量

let、const 和 var 的区别

- 不允许重复声明
- 不存在变量提升// var 的情况 console.log(foo); // 输出undefined var foo = 2; // let 的情况 console.log(bar); // 报错 ReferenceError let bar = 2;
- 暂时性死区（不能在未声明之前使用） 注意暂时性死区和不存在变量提升不是同一个东西 只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。var tmp = 123; // 声明了 tmp if (true) { tmp = "abc"; // ReferenceError let tmp; }
- 块级作用域：用 let 和 const 声明的变量，在这个块中会形成块级作用域**es5 只有函数作用域和全局作用域**IIFE 立即执行函数表达式// IIFE 写法 (function () { var tmp = ...; ... }()); // 块级作用域写法 { let tmp = ...; ... } // 函数声明 function a() {} // 函数表达式 const b = function () {};

ES6 的一些叫法

- reset 参数function add(...values) { let sum = 0; for (var val of values) { sum += val; } return sum; } add(2, 5, 3); // 10
- 扩展运算符console.log(...[1, 2, 3]); // 1 2 3 const b = { ...{ a: "2", b: "3" } };
- ?. 可选链运算符 左侧的对象是否为null或undefined。如果是的，就不再往下运算，而是返回undefined a?.b; // 等同于 a == null ? undefined : a.b; // 注意 undefined == null ==> true
- ?? Null 判断运算符

> https://es6.ruanyifeng.com/#docs/operator#Null-%E5%88%A4%E6%96%AD%E8%BF%90%E7%AE%97%E7%AC%A6

```
const headerText = response.settings.headerText ?? "Hello, world!";
const animationDuration = response.settings.animationDuration ?? 300;
const showSplashScreen = response.settings.showSplashScreen ?? true;

```

但左侧的为 undefined或者null是就返回右边的，否则就直接返回左边的

箭头函数和普通函数的区别

1. 箭头函数没有this，this是继承于当前的上下文，不能通过call,apply,bind去改变 this
2. 箭头函数没有自己的 arguments 对象，但是可以访问外围函数的 arguments 对象
3. 不能通过 new 关键字调用(不能作为构造函数)，同样也没有 new.target 和原型

如何解决异步回调地狱

promise、generator、async/await

mouseover 和 mouseenter 的区别

mouseover：当鼠标移入元素或其子元素都会触发事件，所以有一个重复触发，
冒泡的过程。对应的移除事件是 mouseout
mouseenter：当鼠标移除元素本身（不包含元素的子元素）会触发事件，也就是
不会冒泡，对应的移除事件是 mouseleave

setTimeout、setInterval 和 requestAnimationFrame 之间的区别

与 setTimeout 和 setInterval 不同，requestAnimationFrame 不需要设置时间 间隔， 大多数电脑显示器的刷新频率是 60Hz，大概相当于每秒钟重绘 60 次。大多数浏 览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频 率用户体验也不会有提升。因此，最平滑动画的最佳循环间隔是 1000ms/60，约 等于 16.6ms。 RAF 采用的是系统时间间隔，不会因为前面的任务，不会影响 RAF，但是如果前 面的任务多的话，会响应 setTimeout 和 setInterval 真正运行时的时间间隔。 特点：
（1）requestAnimationFrame 会把每一帧中的所有 DOM 操作集中起来，在一次 重绘或回流中就完成，并且重绘或回流的时间间隔紧紧跟随浏览器的刷新频率。
（2）在隐藏或不可见的元素中，requestAnimationFrame 将不会进行重绘或回 流，这当然就意味着更少的 CPU、GPU 和内存使用量
（3）requestAnimationFrame 是由浏览器专门为动画提供的 API，在运行时浏览 器会自动优化方法的调用，并且如果页面不是激活状态下的话，动画会自动暂停， 有效节省了 CPU 开销。

vue

vue2 是通过Object.defineProperty来实现响应式的，所以就会有一些缺陷

1. 当修改一个对象的某个键值属性时，当这个键值没有在这个对象中，vue 不能做响应式处理
2. 但直接修改数组的某一项（arr[index]='xxx'）vue 不能做响应式处理

可用下面的解决响应式

1. Vue.set ==> this.$set(对象\数组， key 值、index， value)

2. 修改数组length, 调用数据的 splice 方法

vue 生命周期

```
beforeCreate 实例化之前这里能拿到this，但是还不能拿到data里面的数据
created  实例化之后
beforeMount()
mounted() $el
beforeUpdate
updated

beforeDestroy 清除定时/移除监听事件
destroyed

// 被keep-alive 包裹的
// keep-alive 标签 include exclude max
activated() {},
deactivated() {},

// 父子
父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted。

// 离开页面：实例销毁 --> DOM卸载
parent  beforeDestroy
child   beforeDestroy
child   destroyed
parent  destroyed

```

Vue 的 data 为什么是一个函数

因为 Vue 的组件可能会在很多地方使用， 会产生多个实例，如果返回的是对象的， 这些组件之间的数据是同一份（引用关系），那么修改其中一个组件的数据，另外一个组件的数据都会被修改到

Vue key 值的作用

> 看这个视频，你能给面试官说这些，你就很不错了，vue 和 react 的差不多 https://www.bilibili.com/video/BV1wy4y1D7JT?p=48

...待更新

Vue 双向数据绑定原理

> 下面是照抄的一段话，个人觉得这个主要看个人理解，如果看过源码理解 MVVM 这方面的，就很简单

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty()来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。
具体步骤：
第一步：需要 observe 的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter 和 getter
这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化
第二步：compile 解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图
第三步：Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要做的事情是:
1、在自身实例化时往属性订阅器(dep)里面添加自己
2、自身必须有一个 update()方法
3、待属性变动 dep.notice()通知时，能调用自身的 update()方法，并触发 Compile 中绑定的回调，则功成身退。
第四步：MVVM 作为数据绑定的入口，整合 Observer、Compile 和 Watcher 三者，通过 Observer
来监听自己的 model 数据变化，通过 Compile 来解析编译模板指令，最终利用 Watcher 搭起 Observer 和 Compile 之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据 model 变更的双向绑定效果。
所以也可以根据这个来说明为什么 给Vue对象不存在的属性设置值的时候不生效，直接修改数组的index不生效
Vue 提供了 Vue.set(对象|数组, key|index, 值)修改触发响应式，重新数组的原型方法实现响应式

Vue extend 和 mixins

vue extend 和 mixins 的区别， mixins 里面的 函数和本身的函数重名了使用哪一个，mixins 里面的生命周期和本身的生命周期哪一个先执行
...待更新

动态组件

```
1 // component 动态组件，通过is设置要显示的组件
2 <component is="UserInfo" >
3
```

递归组件

就是给组件设置name，之后就可以在当前组件去递归使用组件

Vue 组件间的传值的几种方式

```
// Vue组件间的传值的几种方式
1. props/emit
2. $attrs/$listeners // $attrs 除了父级作用域 props、class、style 之外的属性

// $listeners 父组件里面的所有的监听方法
3. $refs/$parent/$children/$root/
4. vuex
5. 事件总线，通过new Vue去实现 / mitt <==> vue3
6. provide/inject

    // 父组件
    props: {},
    provide() {
        name: this.name,
        user: this.user
    }
    // 子组件
    props: {},
    inject: ['user']
7. 本地存储、全局变量


```

watch、mixins、组件顺序、组件配置

```
export default {
  name: "MyComponentName",
  mixins: [tableMixin],
  components: {},
  inject: ["xxx"],
  // props: ['value', 'visible'],
  props: {
    id: String,
```

```
    type: {
      // required: true,
      type: String,
      default: "warning",
      validator(val) {
        return ["primary", "warning", "danger", "success", "info"].includes(
          val
        );
      },
    },
    list: {
      type: Array,
      default: () => [],
    },
  },
  data() {
    return {
      name: "张三",
      user: { name: "张三", age: 18 },
      loading: true,

      // vue2
      obj: {
        name: "李四~",
      },
      // vue2 会进行深度合并
      // obj  {"name":"李四~","age":19}

      // vue3 { name: "李四~" }
    };
  },
  // provide 不支持响应式，想支持响应式的话我们要传对象
  provide() {
    return {
      userName: this.name,
      user: this.user,
    };
  },
  computed: {
    // fullName() {
```

```
49      //   return 'xxxxx'
50      // }
51      fullName: {
52        get() {
53          return this.$store.state.userName;
54          // return '李四'
55        },
56        set(val) {
57          this.$store.commit("SET_NAME", val);
58        },
59      },
60    },
61
62    watch: {
63      // name(value) {
64      //   this.handlerName()
65      // }
66      // name: {
67      //   immediate: true,
68      //   deep: true, //
69      //   handler(val, oldValue) {
70      //     this.handlerName()
71      //   },
72      // },
73      // this.obj.name = 'xxxx' 这样不会执行
74      // this.obj = {name: 'xxx'} 这样才会执行
75      // obj(value) {
76      //   console.log(' value: ', value)
77      // }
78      //  和上面等价
79      // obj: {
80      //   handler(value) {
81      //     console.log(" value: ", value)
82      //   },
83      // },
84      // this.obj.name = 'xxxx' 这样去修改也能监听
85      // obj: {
86      //   deep: true, // 深度监听
87      //   immediate: true, // 第一次就用执行这个方法，可以理解为在 created 的时候会执行
    handler
```

```
    //   handler(value) {
    //     console.log(" value: ", value)
    //   },
    // },
    //
    // obj: {
    //   deep: true, // 深度监听
    //   immediate: true, // 第一次就用执行这个方法，可以理解为在 created 的时候会执行 handler
    //   handler: 'handlerName',
    // },
    // ==》
    // obj: 'handlerName'
    // '$route.path': {},
    // 'obj.a' : {}
  },

  beforeCreate() {
    console.log("this", this);
  },
  mounted() {
    // this.handlerName()
    this.fullName = "xxxx";

    //  this.fullName '李四'
  },

  methods: {
    handlerName() {
      this.obj.name = "xxxx";
    },
  },
};

```

指令

常用指令

- v-show dispaly none 的切换
- v-if/v-else

- v-html
- v-text
- v-for (vue2 v-for比v-if优先级高，vu3v-if优先级比v-for高)
- v-cloak [v-cloak] {dispaly:none}
- v-once 静态内容
- v-bind => : v-on => @<!--- 可以直接 v-bind="object" v-on="object" --> <Child v-bind="$attrs" v-on="$listeners"> </Child>
- v-model<el-input v-model="keyword"></el-input> <!--- 等价下面这个 --> <el-input :value="keyword" @input="keyword = $event"></el-input>

```
Vue.directive("指令名", {
  // 生命周期
  // 只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
  bind(el, binding, vnode, oldVnode) {
    //
    // binging.value 拿到指令值
    // binding.modifiers 修饰符对象
  },
  // 被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)
  inserted() {},
  update() {},
  componentUpdated() {},
  // 只调用一次，指令与元素解绑时调用
  unbind() {},
});

// 默认绑定 bind update 的生命周期
Vue.directive("指令名", function (el, binding, vnode, oldVnode) {});

```

修饰符

- .lazy、.number、.trim、.enter、.prevent、.self
- .sync<Dialog :visible.sync="visible"></Child> <!--- 等价下面这个 --> <Dialog :visible="visible" @update:visible="visible = $event"></Child>

scoped

加了 scoped 就只作用于当前组件

```
<style scoped></style>

```

渲染规则

```
.a .b {
}
== > .a .b[data-v-xx] {
}
.a /deep/ .b {
}
== > .a[data-v-xxx] .b {
}
.a >>> .b {
}
== > .a[data-v-xxx] .b {
}
.a ::v-deep .b {
}
== > .a[data-v-xxx] .b {
}

```

vue-router

```
// 全局路由守卫
router.beforeEach((to, from, next) => {})
router.afterEach((to, from) => {})

new VueRouter({
    mode: 'hash', // hash | history | abstract
    // 滚动位置
    scrollBehavior(to, from, savedPosition) {
       if (savedPosition) return savedPosition
       return { y: 0 }
        },
    routes: [
        {
            path: '/',
```

```
            // 路由独享守卫
            beforeEnter(to, from, next) {}
        }
    ]
})

// 组件内的路由
beforeRouteEnter(to, from, next) {}
beforeRouteUpdate(to, from, next) {}
beforeRouteLeave(to, from, next) {}

// 跳转
this.$router.push({name: '', path: '', query: {}})
// 路由信息
this.$route.query this.$route.params

```

vuex

state getters mutations actions modules

```
// state
this.$store.state.userInfo;
// getters
this.$store.getters.userInfo;

// mutations
this.$store.commit("SET_USER_INFO", "传递数据");

// actions
this.$store.dispatch("logout").then((res) => {});

// ----------------------------------
// modules > user
// namespaced: true,

// state 拿 name
this.$store.state.user.avatar;
// getters
this.$store.getters.user.avatar;
```

```
20
21 // mutations
22 this.$store.commit("user/SET_TOKEN", "传递数据");
23
24 // actions
25 this.$store.dispatch("user/login").then((res) => {});
26
27 // ----------------------------------
28 // modules > user
29 // namespaced: false,
30
31 // state 拿 name
32 this.$store.state.user.avatar;
33 // getters
34 this.$store.getters.user.avatar;
35
36 // mutations
37 this.$store.commit("SET_TOKEN", "传递数据");
38
39 // actions
40 this.$store.dispatch("login").then((res) => {});
41
```

辅助函数

```
1 mapState, mapGetters, mapMutations, mapActions;
2
```

vue3

> Vue3 的 8 种和 Vue2 的 12 种组件通信，值得收藏 聊一聊 Vue3 的 9 个知识点

Vue3 有哪些变化

- 新增了三个组件：Fragment 支持多个根节点、Suspense 可以在组件渲染之前的等待时间显示指定内容、Teleport 可以让子组件能够在视觉上跳出父组件(如父组件 overflow:hidden)
- 新增指令 v-memo，可以缓存 html 模板，比如 v-for 列表不会变化的就缓存，简单说就是用内存换时间
- 支持 Tree-Shaking，会在打包时去除一些无用代码，没有用到的模块，使得代码打包体积更小
- 新增 Composition API 可以更好的逻辑复用和代码组织，同一功能的代码不至于像以前一样太分散，虽然 Vue2 中可以用 minxin 来实现复用代码，但也存在问题，比如方法或属性名会冲突，代码来源也不清楚等
- 用 Proxy 代替 Object.defineProperty 重构了响应式系统，可以监听到数组下标变化，及对象新增属性，因为监

听的不是对象属性，而是对象本身，还可拦截 apply、has 等 13 种方法

- 重构了虚拟 DOM，在编译时会将事件缓存、将 slot 编译为 lazy 函数、保存静态节点直接复用(静态提升)、以及添加静态标记、Diff 算法使用 最长递增子序列 优化了对比流程，使得虚拟 DOM 生成速度提升 200%
- 支持在 <style></style> 里使用 v-bind，给 CSS 绑定 JS 变量(color: v-bind(str))
- 用 setup 代替了 beforeCreate 和 created 这两个生命周期
- 新增了开发环境的两个钩子函数，在组件更新时 onRenderTracked 会跟踪组件里所有变量和方法的变化、每次触发渲染时 onRenderTriggered 会返回发生变化的新旧值，可以让我们进行有针对性调试
- 毕竟 Vue3 是用 TS 写的，所以对 TS 的支持度更好
- Vue3 不兼容 IE11

vue3 生命周期

选项式 API	Hook inside setup
beforeCreate	Not needed*
created	Not needed*
beforeMount	onBeforeMount
mounted	onMounted
beforeUpdate	onBeforeUpdate
updated	onUpdated
beforeUnmount	onBeforeUnmount
unmounted	onUnmounted
errorCaptured	onErrorCaptured
renderTracked	onRenderTracked
renderTriggered	onRenderTriggered
activated	onActivated
deactivated	onDeactivated

基本代码

main.js

```
// main.js
import { createApp } from "vue";
import App from "./App.vue";

import HelloWorld from "./components/HelloWorld.vue";
const app = createApp(App);

// 全局组件
app.component("HelloWorld", HelloWorld);

// 全局属性
// vue2.0 Vue.prototype.$http
app.config.globalProperties.$http = () => {
  console.log("http ==");
};

app.mount("#app");

```

App.vue

```
<!--- App.vue  -->
<template>
  <div>
    <!-- v-model="xxx"  <==> v-model:modelValue="xxx" -->
    <!-- :value="xxx" @input="xxx = $event" -->
    <!-- value $emit('input', '传递') -->

    <!--
      visible.sync="visible"
      ==>
      :visible="visible" @update:visible="visible = $event"
      -->

    <!-- vue3 把 .sync 去掉，==>
        v-model:visible="visible"
       -->

```

```
    <!--
    <div :ref="setDivRef">
      count: {{ count }}
      <p>
        <button @click="add">+</button>
        <button @click="reduce">-</button>
      </p>
    </div>

    <ul>
      <li>姓名：{{ user.name }}</li>
      <li>年龄：{{ user.age }}</li>
    </ul> -->

    <!-- v-model="num" -->
    <Child
      title="父组件传递的title"
      :modelValue="num"
      @update:modelValue="num = $event"
      @change="onChildChange"
      v-model:visible="visible"
      ref="childRef"
    ></Child>
    <!-- <HelloWorld></HelloWorld> -->
  </div>
</template>

<script>
  import Child from "./Child-setup.vue";
  import { reactive, ref } from "@vue/reactivity";
  import { onMounted, provide } from "@vue/runtime-core";
  export default {
    components: { Child },
    // data() {
    //   return {
    //     msg: '哈哈哈',
    //   }
    // },
    setup() {
      const msg = ref("哈哈哈2"); // => reactive({value: 哈哈哈2 })
```

```
58        const obj = ref({ x: "xx" });
59        console.log(" obj.value: ", obj.value);
60        const user = reactive({ name: "张三", age: 18 });
61        const count = ref(0);
62
63        provide("count", count);
64        provide("http", () => {
65          console.log("$http >>>");
66        });
67
68        const add = () => {
69          count.value++;
70        };
71
72        const reduce = () => {
73          count.value--;
74        };
75
76        const num = ref(1);
77        const visible = ref(false);
78
79        // this.$refs.childRef x
80        // refs
81        // 1. 用字符串
82        const childRef = ref(null);
83        onMounted(() => {
84          console.log(" childRef.value: ", childRef.value);
85        });
86
87        let divRef;
88        const setDivRef = (el) => {
89          console.log(" el: ", el);
90          divRef = el;
91        };
92
93        return {
94          msg,
95          user,
96          count,
```

```
        add,
        reduce,
        num,
        visible,
        childRef,
        setDivRef,
      };
    },

    methods: {
      onChildChange() {},
    },
  };
</script>

<style>
  #app {
    font-family: Avenir, Helvetica, Arial, sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    text-align: center;
    color: #2c3e50;
    margin-top: 60px;
  }
</style>

```

Child-composition （组合式 api）

```
<template>
  <!--
  1. 多个片段， 多个根标签
  2. v-for v-if 优先级变化 v3 v-if > v-for
-->
  <div>
    <button @click="triggerEvent">触发事件</button>

    <div>num2: {{ num2 }}</div>
    <div>count: {{ count }}</div>
```

```
11
12      modelValue: {{ modelValue }}
13      <button @click="add">+</button>
14      <hr />
15      visible: {{ visible }}
16      <button @click="updateVisible">更新visible</button>
17
18      <!--   -->
19      <teleport to="body">
20        <div v-if="visible">对话框</div>
21      </teleport>
22    </div>
23  </template>
24
25  <script>
26    import {
27      computed,
28      inject,
29      onActivated,
30      onBeforeMount,
31      onBeforeUnmount,
32      onBeforeUpdate,
33      onDeactivated,
34      onMounted,
35      onUnmounted,
36      onUpdated,
37      watch,
38      watchEffect,
39    } from "@vue/runtime-core";
40    export default {
41      props: {
42        title: String,
43        modelValue: Number,
44        visible: Boolean,
45      },
46      // computed: {
47      //   num2() {
48      //     return this.modelValue * 2
49      //   }
50      // },
```

```
51      emits: ["change", "update:modelValue", "update:visible"],
52      // 发生在 beforeCreate
53      // attrs 除了 class style,props 之外的属性
54      //
55
56      // watch: {
57      //   title: {
58      //     deep: true, // 深度简单
59
60      //   }
61      // },
62      // 组合式API(composition),  选项式API(options)
63      setup(props, { emit, attrs, slots }) {
64        console.log(" attrs: ", attrs);
65        console.log(" props: ", props);
66
67        // computed
68        const num2 = computed(() => props.modelValue * 2);
69        // const num2 = computed({
70        //   get: () => props.modelValue * 2,
71        //   set: (val) => {
72        //     ssss
73        //   }
74        // })
75
76        //
77        const count = inject("count");
78        console.log(" count: ", count);
79
80        // watch
81        // this.$watch()
82        const unwatch = watch(
83          () => props.modelValue,
84          (newVal, oldValue) => {
85            console.log(" newVal: ", newVal);
86            if (newVal >= 10) {
87              // 取消监听
88              unwatch();
89            }
```

```
    },
    {
      deep: true,
      // immediate: true
    }
  );

  // 自动收集依赖，所以会初始化的时候就执行一次
  watchEffect(() => {
    console.log(" props.modelValue: ", props.modelValue);
  });

  // hooks
  onBeforeMount(() => {});
  onMounted(() => {
    console.log("哈哈哈");
  });
  onBeforeUpdate(() => {});
  onUpdated(() => {});
  onBeforeUnmount(() => {});
  onUnmounted(() => {});

  // keep-alive
  onActivated(() => {});
  onDeactivated(() => {});

  // methods
  const triggerEvent = () => {
    emit("change", "传递的数据");
  };

  const add = () => {
    emit("update:modelValue", props.modelValue + 1);
  };
  const updateVisible = () => {
    console.log(" props.visible: ", props.visible);
    emit("update:visible", !props.visible);
  };

```

```
      return {
        triggerEvent,
        add,
        updateVisible,
        num2,
        count,
      };
    },
    // beforeCreate() {
    //   console.log('beforeCreate')
    // },
    // created() {
    //   console.log('created')
    // },

    // beforeDestroy beforeUnmount
    // destroyed unmounted
  };
</script>

```

Child-setup

```
<template>
  <div>
    <p>title: {{ title }}</p>
    <p>num2: {{ num2 }}</p>
    <p>count: {{ count }}</p>

    <div>
      modelValue: {{ modelValue }}
      <button @click="add">+</button>
    </div>

    <button @click="triggerEvent">触发事件</button>

    <!-- <input type="text" v-model="inputValue"> -->
    <!--   -->
    <input type="text" :value="inputValue" @input="onInputUpdate" />
```

```
17
18      <!-- volar -->
19      <Foo></Foo>
20    </div>
21  </template>
22
23  <!--- vue 3.2.x -->
24  <script setup>
25    import {
26      computed,
27      getCurrentInstance,
28      inject,
29      ref,
30      useAttrs,
31      useSlots,
32    } from "@vue/runtime-core";
33    import Foo from "./foo.vue";
34
35    // props
36    const props = defineProps({
37      title: String,
38      modelValue: Number,
39    });
40    // computed
41    const num2 = computed(() => props.modelValue * 2);
42    const count = inject("count");
43
44    // emit
45    const emit = defineEmits(["change", "update:modelValue", "update:visible"]);
46    const triggerEvent = () => {
47      emit("change", "传递的数据");
48    };
49    const add = () => {
50      emit("update:modelValue", props.modelValue + 1);
51    };
52
53    // 向父组件暴露自己的属性和方法
54    defineExpose({
55      num2,
```

```
    test() {
      console.log("888");
    },
  });

  const attrs = useAttrs();
  console.log(" attrs: ", attrs);
  const solts = useSlots();

  const ctx = getCurrentInstance();

  const http = ctx.appContext.config.globalProperties.$http;
  http("xxx");

  const $http = inject("http");
  $http();

  // $ref: ref(false)

  const inputValue = ref("");

  const onInputUpdate = (event) => {
    console.log(" event: ", event);
    inputValue.value = event.target.value;
  };
</script>

```

项目相关

git

常用命令

https://shfshanyue.github.io/cheat-sheets/git

git pull 和 git featch 的区别

怎么样进行合并，比如把 mater 分支合并到 dev 分支

Webpack 一些核心概念：

> 【万字】透过分析 webpack 面试题，构建 webpack5.x 知识体系

- Entry：入口，指示 Webpack 应该使用哪个模块，来作为构建其内部 依赖图(dependency graph) 的开始。
- Output：输出结果，告诉 Webpack 在哪里输出它所创建的 bundle，以及如何命名这些文件。
- Module：模块，在 Webpack 里一切皆模块，一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。
- Chunk：代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。
- Loader：模块代码转换器，让 webpack 能够去处理除了 JS、JSON 之外的其他类型的文件，并将它们转换为有效 模块，以供应用程序使用，以及被添加到依赖图中。
- Plugin：扩展插件。在 webpack 运行的生命周期中会广播出许多事件，plugin 可以监听这些事件，在合适的时机通过 webpack 提供的 api 改变输出结果。常见的有：打包优化，资源管理，注入环境变量。
- Mode：模式，告知 webpack 使用相应模式的内置优化
- hash: 每次构建的生成唯一的一个 hash，且所有的文件 hash 串是一样的
- chunkhash: 每个**入口文件**都是一个 chunk，每个 chunk 是由入口文件与其依赖所构成，**异步加载**的文件也被视为是一个 chunk, **chunkhash**是由每次编译模块，根据模块及其依赖模块构成 chunk 生成对应的 chunkhash, 这也就表明了**每个 chunk 的 chunkhash 值**都不一样， 也就是说每个 chunk 都是独立开来的，互不影响，每个 chunk 的更新不会影响其他 chunk 的编译构建
- contenthash：由文件内容决定，文件变化 contenthash 才会变化，一般配合 mini-css-extract-plugin插件提取出 cssconst MiniCssExtractPlugin = require("mini-css-extract-plugin"); const HTMLWebpackPlugin = require("html-webpack-plugin"); module.exports = { // ... module: { rules: [{ test: /\.css$/, use: [{ // 把 style-loader替换掉，不要使用 style-loader了 loader: MiniCssExtractPlugin.loader, options: { outputPath: "css/", }, }, "css-loader",], },], }, plugins: [// .... new MiniCssExtractPlugin({ filename: "css/[name].[contenthash].css", }),], };

提升 webpack 打包速度

> 一套骚操作下来，webpack 项目打包速度飞升 🚀、体积骤减 ↓ 玩转 webpack，使你的打包速度提升 90% 带你深度解锁 Webpack 系列(优化篇) 学习 Webpack5 之路（优化篇）- 近 7k 字

- 速度分析，可以使用 speed-measure-webpack-plugin
- 提升基础环境，nodejs 版本，webpack 版本
- CDN 分包 html-webpack-externals-plugin, externals
- 多进程、多实例构建 thread-loader happypack(不再维护)
- 多进程并行构建打包uglifyjs-webpack-plugin terser-webpack-plugin
- 缓存: webpack5 内置了cache模块 、babel-loader 的 cacheDirectory 标志、cache-loader, HardSourceWebpackPluginmodule.exports = { // webpack5内置缓存 cache: { type: "filesystem", // 使用文件缓存 }, };
- 构建缩小范围 include,exclude
- 加快文件查找速度resolve.alias,resolve.extensions, module.noParse

- DllPlugin
- babel 配置的优化

webpack 常用 loader，plugin

loader

- babel-loader 将 es6 转换成 es5 , ts-loader、vue-loader
- eslint-loader 配置 enforce: 'pre' 这个 loader 最先执行
- css-loader、style-loader、postcss-loader、less-loader、sass-loader
- file-loader 把文件转换成路径引入, url-loader（比file-loader多了小于多少的能转换成 base64）
- image-loader
- svg-sprite-loader 处理 svg
- thread-loader 开启多进程，会在一个单独的 worker 池（worker pool）中运行
- cache-loader 缓存一些性能开销比较大的 loader 的处理结果

plugin

- html-webpack-plugin 将生成的 css，js 自动注入到 html 文件中，能对 html 文件压缩
- copy-webpack-plugin 拷贝某个目录
- clean-webpack-plugin 清空某个目录
- webpack.HotModuleReplacementPlugin 热重载
- webpack.DefinePlugin 定义全局变量
- mini-css-extract-plugin 提取 CSS 到独立 bundle 文件。 extract-text-webpack-plugin
- optimize-css-assets-webpack-plugin 压缩 css webpack5 推荐css-minimizer-webpack-plugin
- purgecss-webpack-plugin 会单独提取 CSS 并清除用不到的 CSS（会有问题把有用的 css 删除）
- uglifyjs-webpack-plugin ✕（不推荐） 压缩 js、多进程 parallel: true
- terser-webpack-plugin 压缩 js， 可开启多进程压缩、推荐使用module.exports = { optimization: { minimize: true, minimizer: [new TerserPlugin({ parallel: true, // 多进程压缩 }),], }, };
- Happypack ✕（不再维护） 可开启多进程
- HardSourceWebpackPlugin 缓存
- speed-measure-webpack-plugin 打包构建速度分析、查看编译速度
- webpack-bundle-analyzer 打包体积分析
- compression-webpack-plugin gzip 压缩

前端性能优化

> 前端性能优化 24 条建议（2020）

1. 减少 http 请求
2. 使用 http2
3. 静态资源使用 CDN
4. 将 CSS 放在文件头部，JavaScript 文件放在底部

5. 使用字体图标 iconfont 代替图片图标
6. 设置缓存，强缓存，协商缓存
7. 压缩文件，css(MiniCssExtractPlugin),js(UglifyPlugin),html(html-webpack-plugin)文件压缩，清除无用的代码，tree-shaking（需要 es6 的 import 才支持），gzip 压缩(compression-webpack-plugin)
8. splitChunks 分包配置，optimization.splitChunks 是基于 SplitChunksPlugin 插件实现的
9. 图片优化、图片压缩
10. webpack 按需加载代码，hash，contenthash
11. 减少重排重绘
12. 降低 css 选择器的复杂性

babel

> 不容错过的 Babel7 知识

核心库 @babel/core

Polyfill 垫片

CLI 命令行工具 @babel/cli

插件

预设：包含了很多插件的一个组合，@babel/preset-env @babel/preset-react @babel/preset-typescript

polyfill

Babel 默认只转换新的 JavaScript 句法（syntax），而不转换新的 API，比如Iterator、Generator、Set、Map、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转码。

举例来说，ES6 在Array对象上新增了Array.from方法。Babel 就不会转码这个方法。如果想让这个方法运行，可以使用core-js和regenerator-runtime(后者提供 generator 函数的转码)，为当前环境提供一个垫片。

@babel/plugin-transform-runtime

Babel 会使用很小的辅助函数来实现类似 _createClass 等公共方法。默认情况下，它将被添加(inject)到需要它的每个文件中。

如果你有 10 个文件中都使用了这个 class，是不是意味着 _classCallCheck、_defineProperties、_createClass 这些方法被 inject 了 10 次。这显然会导致包体积增大，最关键的是，我们并不需要它 inject 多次。

@babel/plugin-transform-runtime 是一个可以重复使用 Babel 注入的帮助程序，以节省代码大小的插件。

```
npm install --save-dev @babel/plugin-transform-runtime
npm install --save @babel/runtime

```

```
//.babelrc
{
    "presets": [
        [
            "@babel/preset-env",
            {
                "useBuiltIns": "usage", // 配置 polyfill 动态导入
                "corejs": 3 // core-js@3
            }
        ]
    ],
    "plugins": [
        [
            "@babel/plugin-transform-runtime"
        ]
    ]
}

```

浏览器

跨域、同源策略

参考：https://blog.csdn.net/weixin_43745075/article/details/115482227

> 同源策略是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSRF等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

同源策略限制内容有：

- Cookie、LocalStorage、IndexedDB 等存储性内容
- DOM 节点
- AJAX 请求发送后，结果被浏览器拦截了

但是有三个标签是允许跨域加载资源：

```
<img src="XXX">
<link href="XXX">
<script src="XXX">

```

跨域解决方案

1. JSONP：通过动态创建script，再请求一个带参网址实现跨域通信。
2. 开发环境：前端做代理
3. nginx反向代理
4. CORS: 服务端设置 Access-Control-Allow-Origin 即可，前端无须设置，若要带 cookie 请求，前后端都需要设置。
5. websocket---下面的跨域通信、注意只是页面之间的跨域，不是前后端服务跨域，别人问前后端跨域就不要回答下面的了
6. postMessage
7. window.name + iframe
8. document.domain + iframe
9. location.hash + iframe

垃圾回收机制

- 标记清除： 进入环境、离开环境
- 引用计数（不常用）：值被引用的次数， 当引用次数为零时会被清除（缺陷，相互引用的会有问题）

缓存机制

强缓存

> 如果命中强缓存--就不用像服务器去请求

1. Expires 设置时间，过期时间 expires: Tue, 15 Oct 2019 13:30:54 GMT通过本地时间和 expires 比较是否过期，如果过期了就去服务器请求，没有过期的话就直接使用本地的缺点：本地时间可能会更改， 导致缓存出错
2. Cache-Control HTTP1.1 中新增的
    - max-age 最大缓存多少毫秒，列如 Cache-Control: max-age=2592000
    - no-store (每次都要请求，就连协商缓存都不走)表示不进行缓存，缓存中不得存储任何关于客户端请求和服务端响应的内容。每次 由客户端发起的请求都会下载完整的响应内容。Cache-Control: no-store
    - no-cache（默认值）表示不缓存过期的资源，缓存会向源服务器进行有效期确认后处理资源，也许称 为 do-notserve-from-cache-without-revalidation 更合适。浏览器默认开启的是 no-cache，其 实这里也可理解为开启协商缓存
    - public 和 privatepublic 与 private 是针对资源是否能够被代理服务缓存而存在的一组对立概念当我们为资源

设置了 pubile，那么它既可以被浏览器缓存也可被代理服务器缓存。设置为private 的时候，则该资源只能被浏览器缓存，其中默认值是 private。

- max-age 和 s-maxages-maxage 只适用于供多用户使用的公共服务器上(如 CND cache)，并只对 public 缓存有效

协商缓存

> 需要向服务器请求，如果没有过期，服务器会返回 304，

1. ETag 和 If-None-Match 唯一标识

- 服务器响应 ETag 值，浏览器携带的是 If-None-Match（携带的是上一次响应的 ETag），服务拿到这 If-None-Match 值后判断过期--> 没有过期 304，并且返回 ETag二者的值都是服务器为每份资源分配的唯一标识字符串。• 浏览器请求资源，服务器会在响应报文头中加入 ETag 字段。资源更新的时候，服务端的ETag 值也随之更新• 浏览器再次请求资源，会在请求报文头中添加 If-None-Match 字段，它的值就是上次响应报文中的 ETag 值，服务器会对比 ETag 和 If-None-Match 的值是否一致。如果不一致，服务器则接受请求，返回更新后的资源，状态码返回 200；如果一致，表明资源未更新，则返回状态码 304，可继续使用本地缓存，值得注意的是此时响应头会加上 ETag 字段，即使它没有变化
- **Last-Modified 和 If-Modified-Since 时间戳** 缺点： 某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说 1s 内修改了 N 次)，If-Modified-Since 可查到的是秒级，这种修改无法判断

预编译

四部曲

1. 创建AO对象
2. 找形参和变量声明，将变量和形参名作为AO的属性名，值为undefined
3. 将实参值和形参值相统一
4. 在函数体里面找到函数声明，值赋予函数体

```
// 预编译
function foo(test /* 形参 */) {
  console.log(" test: ", test); // function(){}
  var test = 2;
  var str = "bs";
  console.log(" test: ", test); // 2
  // 函数声明
  function test() {}
  // 函数表达式
  str = function () {};
  console.log(" test: ", test); // 2
}

```

```
// 预编译 四部曲
// 1. 创建AO对象
// 2. 找形参和变量声明，将变量和形参名作为AO的属性名，值为undefined
// 3. 将实参值和形参值相统一
// 4. 在函数体里面找到函数声明，值赋予函数体

// AO {
//   test: undefined
//   str: undefined
// }
// AO {
//   test: 1
//   str: undefined
// }
// AO {
//   test: 1
//   str: function() {}
// }

foo(1 /*实参*/);

function foo(a, b, c) {
  console.log(a);
  console.log(b);
  var a = "222";
  function a() {}
  var b = function () {};
  console.log(a);
  console.log(b);

  console.log(" a, b, c: ", a, b, c);
}
// AO {
//   a : '222',
//   b : function() {},
//   c : 3
// }
foo(1, 2, 3);

```

```
var a = 22;
// let a = 22
// window.a ==> 22

```

全局

1. 创建 GO 对象==window
2. 变量声明，将变量作为 GO 的属性名，值为undefined
3. 找到函数声明，值赋予函数体

event-loop(事件循环)

> 一次弄懂 Event Loop（彻底解决此类面试问题）

JS是单线程的，为了防止一个函数执行时间过长阻塞后面的代码，所以会先将同步代码压入执行栈中，依次执行，将异步代码推入异步队列，异步队列又分为宏任务队列和微任务队列，因为宏任务队列的执行时间较长，所以微任务队列要优先于宏任务队列。微任务队列的代表就是，Promise.then，MutationObserver，宏任务的话就是setImmediate setTimeout setInterval

MacroTask（宏任务）*

- script全部代码、setTimeout、setInterval、setImmediate（浏览器暂时不支持，只有 IE10 支持，具体可见 MDN）、I/O、UI Rendering。

MicroTask（微任务）

- Process.nextTick（Node独有）、Promise.then、Object.observe(废弃)、MutationObserver

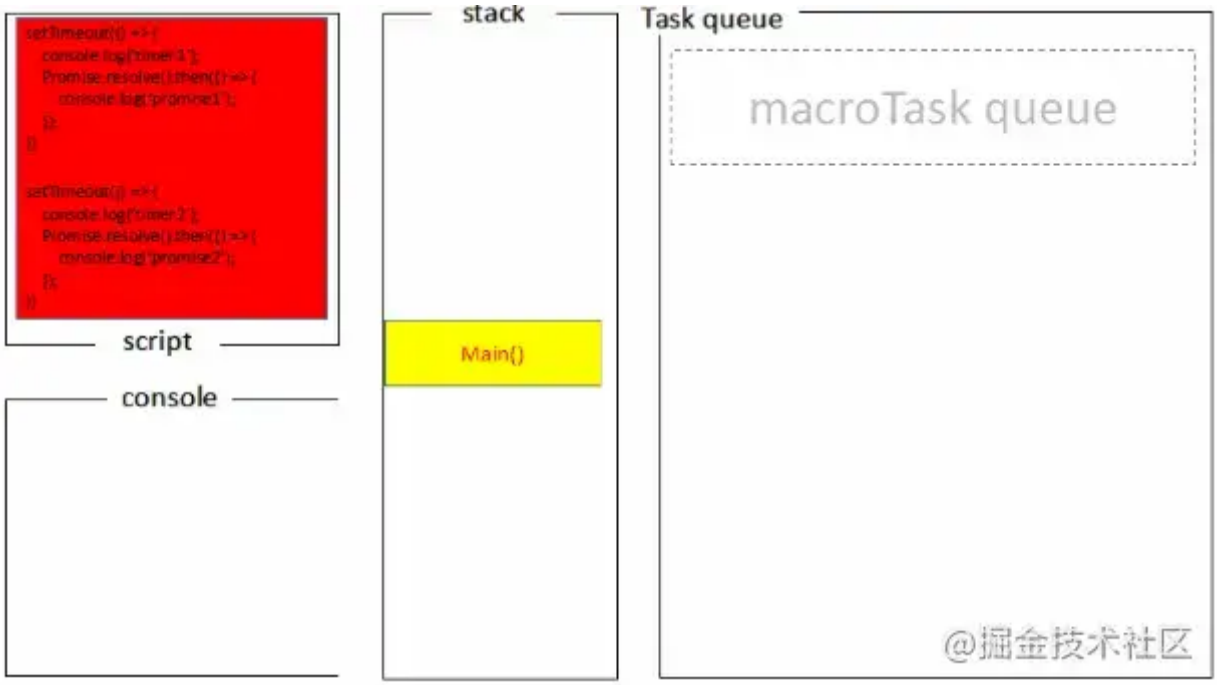
浏览器中

> 执行完一个宏任务，会执行所有的微任务

```
console.log("script start");

setTimeout(function () {
  console.log("setTimeout");
}, 0);

new Promise((resolve) => {
  console.log("promise1");
  resolve();
}).then(function () {
  console.log("promise2");
});
console.log("script end");

```

执行结果

```
script start
promise1
script end
promise2
setTimeout

```



nodejs 中

> 在 11 之前的版本，会在每个阶段之后执行所有的微任务 在 11 版本及之后，会每执行完一个宏任务，就会清空所用的微任务（和浏览器保存一致）

```
new Promise((resolve) => {
  console.log("new Promise 1");
  resolve();
}).then(() => {
  console.log("new Promise then");
});

setTimeout(() => {
  console.log("timer1");
  new Promise((resolve) => {
    console.log("timer1 new Promise");
```

```
    resolve();
  }).then(() => {
    console.log("timer1 new Promise then");
  });
  Promise.resolve().then(() => {
    console.log("timer1 Promise then");
  });
});

setTimeout(() => {
  console.log("timer2");
  Promise.resolve().then(() => {
    console.log("timer2 Promise then");
  });
});

console.log("start end");

```

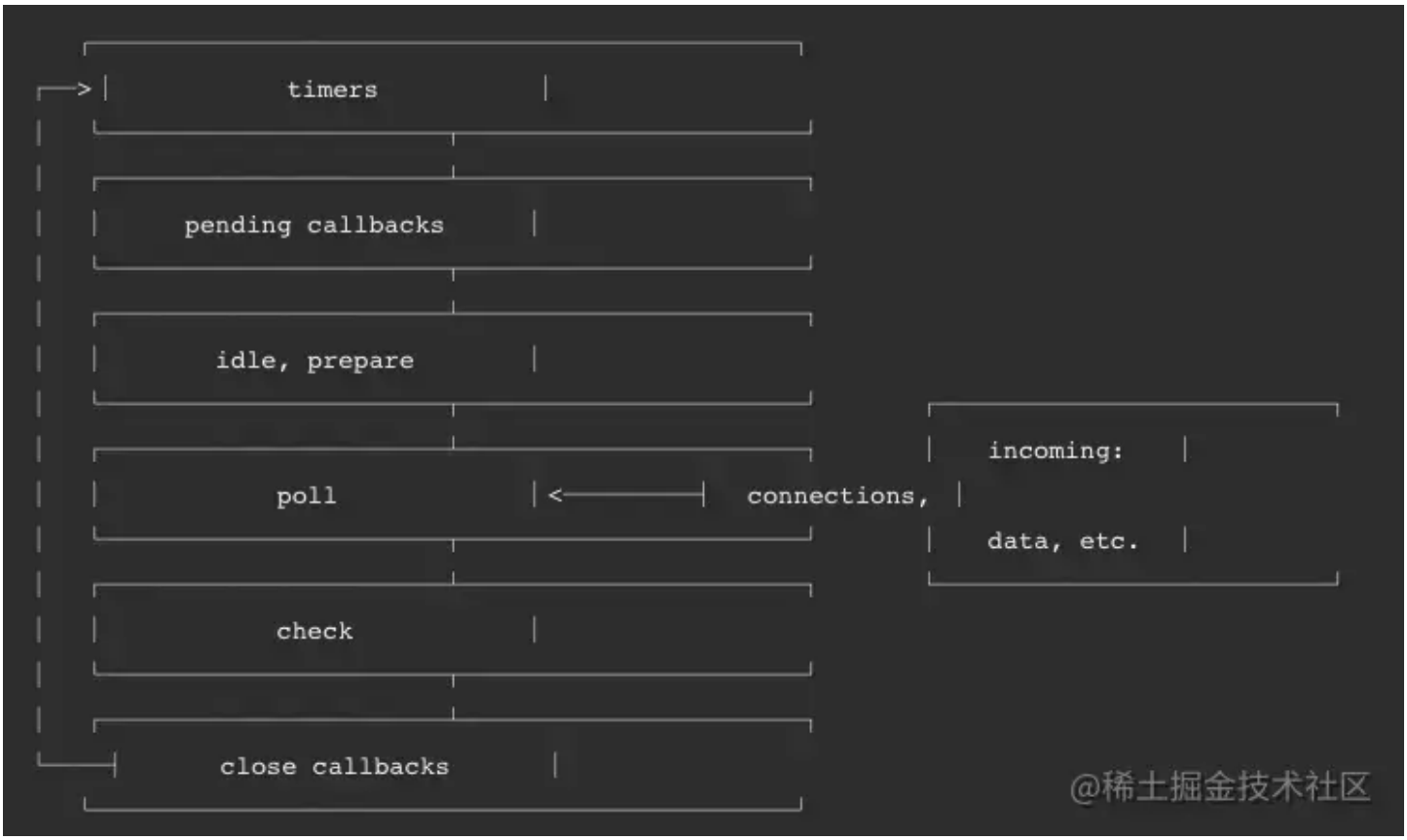
在 node11 版本之前（不包含 11）

```
new Promise 1
start end
new Promise then
timer1
timer1 new Promise
timer2
timer1 new Promise then
timer1 Promise then
timer2 Promise then

```

在 node11 版本及之后

```
new Promise 1
start end
new Promise then
timer1
timer1 new Promise
timer1 new Promise then
timer1 Promise then
```

```
timer2
timer2 Promise then

```



Node的Event loop一共分为 6 个阶段，每个细节具体如下：

1. timers: 执行setTimeout和setInterval中到期的callback。
2. pending callback: 上一轮循环中少数的callback会放在这一阶段执行。
3. idle, prepare: 仅在内部使用。
4. poll: 最重要的阶段，执行pending callback，在适当的情况下回阻塞在这个阶段。
5. check: 执行setImmediate(setImmediate()是将事件插入到事件队列尾部，主线程和事件队列的函数执行完成之后立即执行setImmediate指定的回调函数)的callback。
6. close callbacks: 执行close事件的callback，例如socket.on('close'[,fn])或者http.server.on('close, fn)。

网络

常见状态码

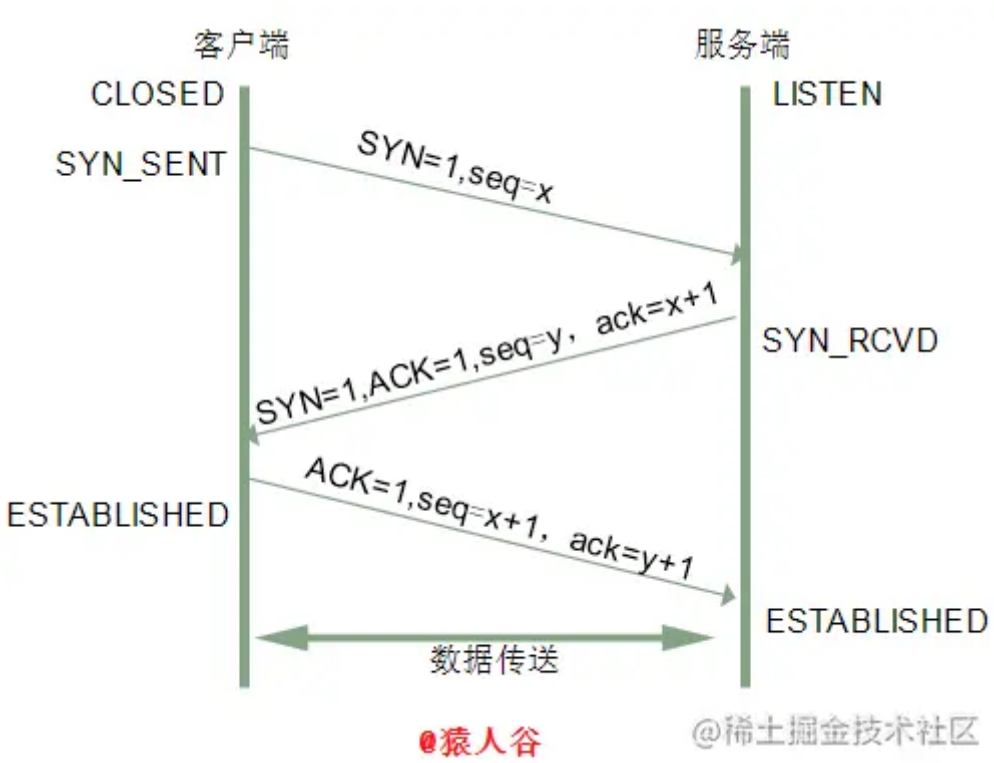
```
1xx: 接受，继续处理
200: 成功，并返回数据
201: 已创建
202: 已接受
203: 成为，但未授权
```

```
204: 成功，无内容
205: 成功，重置内容
206: 成功，部分内容
301: 永久移动，重定向
302: 临时移动，可使用原有 URI
304: 资源未修改，可使用缓存
305: 需代理访问
400: 请求语法错误
401: 要求身份认证
403: 拒绝请求
404: 资源不存在
500: 服务器错误


```

TCP

> 面试官，不要再问我三次握手和四次挥手

三次握手



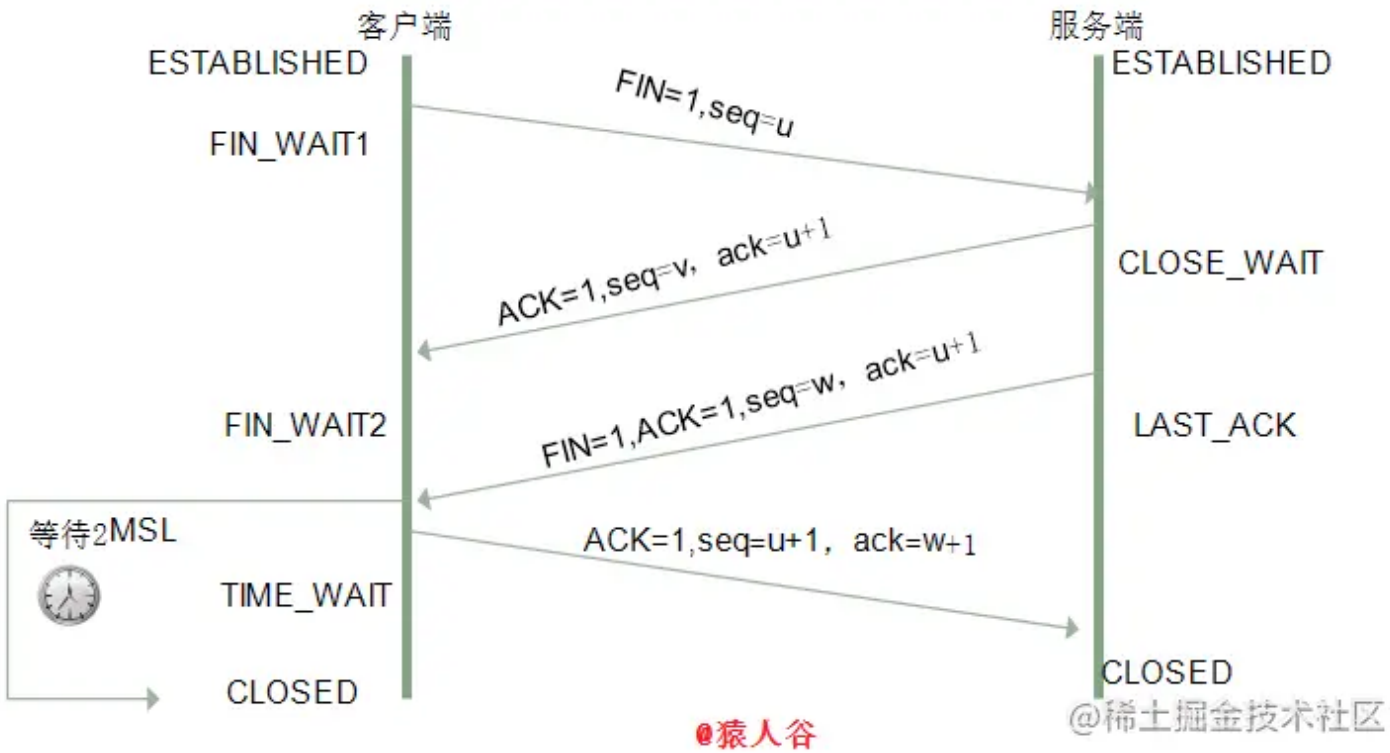
为什么需要三次握手，两次不可以吗

```
为了防止失效的连接请求又传送到主机，因而产生错误。
如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请
```

求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于 TCP 的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

四次挥手



挥手为什么需要四次?

因为当服务端收到客户端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中**ACK 报文是用来应答的，SYN 报文是用来同步的。**但是关闭连接时，当服务端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉客户端，"你发的 FIN 报文我收到了"。只有等到我服务端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四次挥手。

2MSL等待状态

TIME_WAIT 状态也成为2MSL等待状态。每个具体 TCP 实现必须选择一个报文段最大生存时间 MSL（Maximum Segment Lifetime），它是任何报文段被丢弃前在网络内的最长时间。这个时间是有限的，因为 TCP 报文段以 IP 数据报在网络内传输，而 IP 数据报则有限制其生存时间的 TTL 字段。

对一个具体实现所给定的 MSL 值，处理的原则是：当 TCP 执行一个主动关闭，并发回最后一个 ACK，该连接必须在 TIME_WAIT 状态停留的时间为 2 倍的 MSL。这样可让 TCP 再次发送最后的 ACK 以防这个 ACK 丢失（另一端超时并重发最后的 FIN）。
这种 2MSL 等待的另一个结果是这个 TCP 连接在 2MSL 等待期间，定义这个连接的插口（客户的 IP 地址和端口号，服务器的 IP 地址和端口号）不能再被使用。这个连接只能在 2MSL 结束后才能再被使用。

HTTP 版本

HTTP/1.0
最早的 http 只是使用一些简单的网页上和网络请求上，每次请求都打开一个新的 TCP 连接， 收到响应后立即断开连接
HTTP/1.1
缓存处理，HTTP/1.1 更多的引入了缓存策略，如 Cache-Control，Entity tag，If-Unmodified-Since, If-Match, If-None-Match 等
宽带优化及网络连接的使用，在 HTTP/1.0 中，存在一些浪费宽带的现象，列如客户端只需要某个对象的一部分，而服务器把整个对象都送过来了，并且不支持断点续传，HTTP1.1 则
在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（PartialContent），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
错误通知的管理，在 HTTP/1.1 中新增了 24 个错误状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
Host 头处理，在 HTTP1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。HTTP1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有 Host 头域会报告一个错误（400 Bad Request）
长连接， HTTP/1.1 默认开启持久连接（默认：keep-alive），在一个 TCP 连接上可以传递多个 HTTP 请求和响应，减少了建立与关闭连接的消耗和延迟
HTTP/2.0
在 HTTP/2.0 中，有两个重要的概念，分别是帧（frame） 和 流（stream），帧代表数据传输的最小单位，每个帧都有序列标识标明该帧属于哪个流，流也就是多个帧组成的数据流，每个流表示一个请求。
新的二进制格式： HTTP/1.x 的解析是基于文本的。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认 0 和 1 的组合。基于这种考虑 HTTP2.0 的协议解析决定采用二进制格式，实现方便且健壮。
多路复用： HTTP/2.0 支持多路复用，这是 HTTP/1.1 持久连接的升级版。多路复用，就是在一个 TCP 连接中存在多个条流，也就是多个请求，服务器则可以通过帧中的标识知道该帧属于哪个流（即请求），通过重新排序还原请求。多路复用允许并发多个请求，每个请求及该请求的响应不需要等待其他的请求或响应，避免了线头阻塞问题。这样某个请求任务耗时严

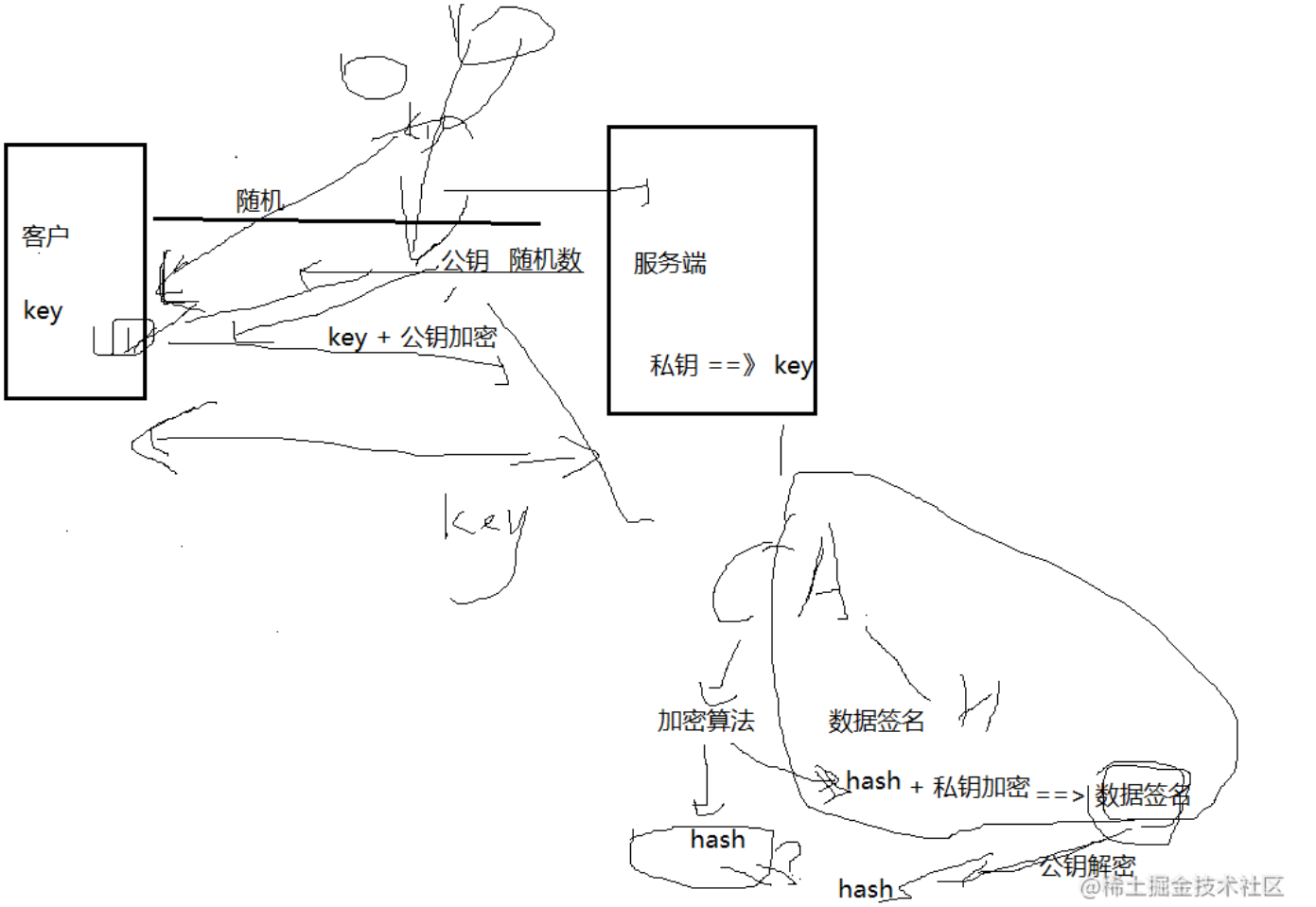
重，不会影响到其它连接的正常执行,极大的提高传输性能。
头部压缩： 对前面提到的 HTTP/1.x 的 header 带有大量信息，而且每次都要重复发送，HTTP/2.0 使用 encoder 来减少需要传输的头部大小，通讯双方各自 cache 一份头部 fields 表，既避免了重复头部的传输，又减小了需要传输的大小。
服务端推送： 服务端推送指把客户端所需要的 css/js/img 资源伴随着 index.html 一起发送到客户端，省去了客户端重复请求的步骤（从缓存中取）。正因为没有发起请求，建立连接等操作，所以静态资源通过服务端推送的方式极大的提升了速度HTTP/3.0
HTTP/2.0 使用了多路复用，一般来说同一域名下只需要使用一个 TCP 连接。但当这个连接中出现了丢包的情况，会导致整个 TCP 都要开始等待重传，也就导致了后面所有的数据都阻塞了。
避免包阻塞： 多个流的数据包在 TCP 连接上传输时，若一个流中的数据包传输出现问题，TCP 需要等待该包重传后，才能继续传输其它流的数据包。但在基于 UDP 的 QUIC 协议中，不同的流之间的数据传输真正实现了相互独立互不干扰，某个流的数据包在出问题需要重传时，并不会对其他流的数据包传输产生影响。
快速重启会话: 普通基于 tcp 的连接，是基于两端的 ip 和端口和协议来建立的。在网络切换场景，例如手机端切换了无线网，使用 4G 网络，会改变本身的 ip，这就导致 tcp 连接必须重新创建。而 QUIC 协议使用特有的 UUID 来标记每一次连接，在网络环境发生变化的时候，只要 UUID 不变，就能不需要握手，继续传输数据。
HTTP2.0 的多路复用和 HTTP1.X 中的长连接有什么区别?
HTTP/1.* 一次请求-响应，建立一个连接，用完关闭；每一个请求都要建立一个连接；
HTTP/1.1 在一个 TCP 连接上可以传递多个 HTTP 请求和响应，后面的请求等待前面的请求返回才能获得执行机会，一旦有某个请求超时，后续请求只能被阻塞，毫无办法，也就是常说的线头阻塞
HTTP/2.0 多个请求可同时在一个连接上并行执行.某个请求任务耗时严重，不影响其他连接的正常执行。

https(http + ssl/tls)

http: 最广泛网络协议，BS 模型，浏览器高效。
https: 安全版，通过 SSL 加密，加密传输，身份认证，密钥
1 https 相对于 http 加入了 ssl 层, 加密传输, 身份认证;
2 需要到 ca 申请收费的证书;
3 安全但是耗时多，缓存不是很好;
4 注意兼容 http 和 https;
5 连接方式不同, 端口号也不同, http 是 80, https 是 443

- 明文： 普通的文本
- 密钥：把明文加密的那个钥匙
- 密文： 把明文加密明文+密钥==>密文==>密钥==解密=>明文
- 对称加密 解密的 key（密钥）和解密的 key 是同一个 3 + 1

- 非对称加密 私钥和公钥



手写

> 10 个常见的前端手写功能，你全都会吗

最近面试 2022 年 3 月问到了很多手写，这个一定要准备下

防抖

```
function debounce(func, wait, immediate) {
  let timeout;
  return function () {
    const context = this;
    const args = [...arguments];
    if (timeout) clearTimeout(timeout);
    if (immediate) {
      const callNow = !timeout;
      timeout = setTimeout(() => {
```

```
        timeout = null;
      }, wait);
      if (callNow) func.apply(context, args);
    } else {
      timeout = setTimeout(() => {
        func.apply(context, args);
      }, wait);
    }
  };
}

```

节流

```
function throttle(fn, wait) {
  let pre = 0;
  return function (...args) {
    let now = Date.now();
    if (now - pre >= wait) {
      fn.apply(this, args);
      pre = now;
    }
  };
}

```

event bus 事件总线 | 发布订阅模式

```
// event bus

class EventBus {
  constructor() {
    this.events = {};
  }
  on(name, callback) {
    const { events } = this;
    if (!events[name]) {
      events[name] = [];
    }
```

```
    events[name].push(callback);
  }
  emit(name, ...args) {
    const handlers = this.events[name];
    handlers &&
      handlers.forEach((fn) => {
        fn.apply(this, args);
      });
  }
  off(name, callback) {
    const { events } = this;
    if (!events[name]) return;
    events[name] = events[name].filter((fn) => fn !== callback);
  }
  once(name, callback) {
    const handler = function () {
      callback.apply(this, arguments);
      this.off(name, handler);
    };
    this.on(name, handler);
  }
  clear() {
    this.events = {};
  }
}

```

数据偏平化

```
// 数据偏平化
function flatter(arr) {
  return arr.reduce((prev, curr) => {
    return Array.isArray(curr) ? [...prev, ...flatter(curr)] : [...prev, curr];
  }, []);
}

```

手写 new

```
// 手写 new
function myNew(ctr, ...args) {
  const obj = Object.create(ctr.prototype);
  myNew.target = ctr;
  const result = ctr.apply(obj, args);
  if (
    result &&
    (typeof result === "function" || typeof result === "function")
  ) {
    return result;
  }
  return obj;
}

```

call、bind

```
Function.prototype.myCall = function (context, ...args) {
  context = context || window;
  const fn = Symbol();
  context[fn] = this;
  const result = context[fn](...args);
  delete context[fn];
  return result;
};

//bind实现要复杂一点  因为他考虑的情况比较多 还要涉及到参数合并(类似函数柯里化)

Function.prototype.myBind = function (context, ...args) {
  if (!context || context === null) {
    context = window;
  }
  // 创造唯一的key值  作为我们构造的context内部方法名
  let fn = Symbol();
  context[fn] = this;
  let _this = this;
  //  bind情况要复杂一点
  const result = function (...innerArgs) {
    // 第一种情况 :若是将 bind 绑定之后的函数当作构造函数，通过 new 操作符使用，则不绑定传入的 this，而是将 this 指向实例化出来的对象
```

```
    // 此时由于new操作符作用  this指向result实例对象  而result又继承自传入的_this 根据原型链知识可得出以下结论
    // this.__proto__ === result.prototype   //this instanceof result =>true
    // this.__proto__.__proto__ === result.prototype.__proto__ === _this.prototype; //this instanceof _this =>true
    if (this instanceof _this === true) {
      // 此时this指向指向result的实例  这时候不需要改变this指向
      this[fn] = _this;
      this[fn](...[...args, ...innerArgs]); //这里使用es6的方法让bind支持参数合并
    } else {
      // 如果只是作为普通函数调用  那就很简单了 直接改变this指向为传入的context
      context[fn](...[...args, ...innerArgs]);
    }
  };
  // 如果绑定的是构造函数 那么需要继承构造函数原型属性和方法
  // 实现继承的方式: 使用Object.create
  result.prototype = Object.create(this.prototype);
  return result;
};

```

异步控制并发数

```
function limitRequest(requests, limit = 3) {
  requests = requests.slice();
  return new Promise((resolve, reject) => {
    let count = 0;
    const len = requests.length;
    while (limit > 0) {
      start();
      limit--;
    }

    function start() {
      const promiseFn = requests.shift();

      promiseFn?.().finally(() => {
        count++; // 一定要通过 count 判断、不能通过 requests.length 判断是否为空，这样不对的
        if (count === len) {
          // 最后一个
```

```
          resolve();
        } else {
          start();
        }
      });
    }
  });
}
// 测试
const arr = [];
for (let value of "12345") {
  arr.push(() => fetch(`https://www.baidu.com/s?ie=UTF-8&wd=${value}`));
}
limitRequest(arr);

```

算法/特殊题目

最近面试 2022 年 3 月问到了很多手写，这个一定要准备下、下面都是我被问到的

台阶问题

有 N 个台阶，一步可以走一梯或者两梯，请问有多少种走法

解答：https://blog.csdn.net/z1832729975/article/details/123836190

有效括号

> 我面试才几家，这个有两家都问到了 力扣原题

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。

左括号必须以正确的顺序闭合。

```
示例 1：
输入：s = "()"
输出：true


示例 2：
```

```
输入：s = "()[]{}"
输出：true

示例 3：
输入：s = "(]"
输出：false

```

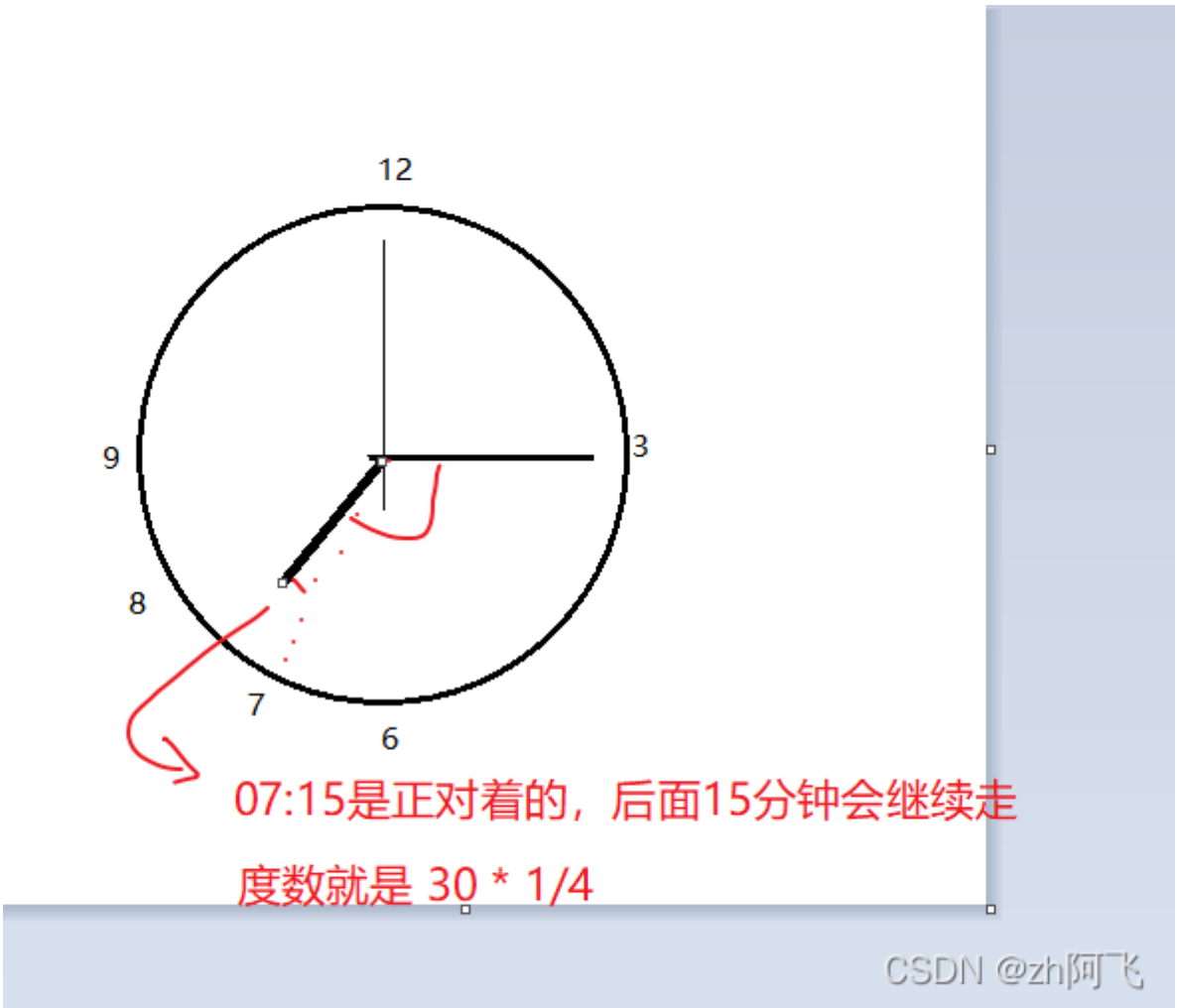
实现

我们可以通过栈来实现、当遇到左括号的时候就把对应的右括号值push到栈中，否则的话我们就把栈定的元素pop和当前字符比较是否相等，不相信的话直接返回 false

```
/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function (s) {
  if (!s) return true;
  const leftFlags = {
    "(": ")",
    "{": "}",
    "[": "]",
  };
  const stack = [];
  for (let chart of s) {
    const flag = leftFlags[chart];
    // 是左括号
    if (flag) {
      stack.push(flag);
    }
    // 是右括号
    else if (chart !== stack.pop()) {
      return false;
    }
  }
  return stack.length === 0;
};

```

现在时间 07:15，请问分针和时针的夹角是多少

先看看时钟，要了解 07:15 在哪，这个不知道在哪就尴尬了



画图，结果如下

7 点 15 分时针和分针所形成的角是

120 + 30*1/4=127.5

这题需要注意时针还好继续走，不会固定，不然容易被坑

写 IP 地址的正则表达式

分析ip地址

让 a==1 && a==2 && a==3 为 true

> 因为这个是 ==，会存在隐式类型转换

- 利用对象Symbol.toStringvalueOftoStringvar a = { value: 1, // 这三种方法都可以，优先级也是这个顺序 [Symbol.toString]() { return a.value++; }, // valueOf() { // return a.value++ // }, // toString() { // return a.value++ // } };
- 利用数组a.valueOf = a.shift; // 一样有 //a[Symbol.toPrimitive] = a.shift //a.toString = a.shift
- 通过Object.defineProperty拦截let value = 1; Object.defineProperty(window, "a", { get() { return value++; }, });
- 通过 Proxy 拦截let value = 1; const a = new Proxy({}, { get() { return function () { return value++; }; }, });

typescript

> 建议先把基础的东西学会，推荐看这篇文章、基础的学会，就能应付大多数的 typescript 面试了 2021 typescript 史上最强学习入门文章(2w 字)

const和readonly的区别

const常量：表示这个变量的指针地址不可以在改变，可以更改对象内部的属性

readonly只读：指针地址不可以改变，并且对象内部的属性也不可以改变

1. const 用于变量，readonly 用于属性
2. const 在运行时检查，readonly 在编译时检查
3. 使用 const 变量保存的数组，可以使用 push，pop 等方法。但是如果使用ReadonlyArray声明的数组不能使用 push，pop 等方法。

type和interface的区别

参考：https://juejin.cn/post/7018805943710253086#heading-63

type-类型别名

interface-接口

1. 接口重名会合并、类型别名重名会报错interface Person { name: string; } interface Person { age: number; } // 这个接口合并，变成下面的 interface Person { name: string; age: number; } type Aanimal = { name: string }; type Aanimal = { age: number }; // 会报错、重名了
2. 两者都可以用来描述对象或函数的类型，但是语法不同interfaceinterface Point { x: number; y: number; } interface SetPoint { (x: number, y: number): void; } typetype Point = { x: number; y: number; }; type SetPoint = (x: number, y: number) => void;
3. 类型别名可以为任何类型引入名称。例如基本类型，联合类型等// primitive type Name = string; // object type

PartialPointX = { x: number }; type PartialPointY = { y: number }; // union type PartialPoint = PartialPointX | PartialPointY; // tuple type Data = [number, string]; // dom let div = document.createElement("div"); type B = typeof div;

4. 扩展两者的扩展方式不同，但并不互斥。接口可以扩展类型别名，同理，类型别名也可以扩展接口。接口的扩展就是继承，通过 extends 来实现。类型别名的扩展就是交叉类型，通过 & 来实现。接口扩展接口interface PointX { x: number; } interface Point extends PointX { y: number; }

类型别名扩展类型别名

```
type PointX = {
  x: number;
};

type Point = PointX & {
  y: number;
};

```

接口扩展类型别名

```
type PointX = {
  x: number;
};
interface Point extends PointX {
  y: number;
}

```

类型别名扩展接口

```
interface PointX {
  x: number;
}
type Point = PointX & {
  y: number;
};

```

keyof 和 typeof 关键字的作用?

> keyof 索引类型查询操作符 获取索引类型的属性名，构成联合类型。 typeof 获取一个变量或对象的类型。

unknown, any 的区别

> unknown 类型和 any 类型类似。与 any 类型不同的是。unknown 类型可以接受任意类型赋值，但是 unknown 类型赋值给其他类型前，必须被断言