目录

一. HTML 篇

1.页面导入样式时，使用 link 和 @import 有什么区别。

a.从属关系区别。@import 只能导入样式表，link 还可以定义 RSS、rel 连接属性、引入网站图标等；

b.加载顺序区别；加载页面时，link 标签引入的 CSS 被同时加载；@import 引入的 CSS 将在页面加载完毕后被加载；

c.兼容性区别；

2. 常见浏览器内核

内核

- Trident：IE 浏览器内核；
- Gecko：Firefox 浏览器内核；
- Presto：Opera 浏览器内核；
- Webkit：Safari 浏览器内核；
- Blink：谷歌浏览器内核，属于 Webkit 的一个分支，与 Opera 一起在研发；

浏览器

- IE：Trident，IE 内核；
- Chrome：以前是 Webkit，现在是 Blink 内核；
- Firefox：Gecko 内核；
- Safari：Webkit 内核；
- Opera：一起是 Presto，现在是 Blink 内核；
- 360、猎豹浏览器内核：IE + Blink 双内核；
- 搜狗、遨游、QQ 浏览器内核：Trident（兼容模式）+ Webkit（高速模式）；
- 百度浏览器、世界之窗内核：IE 内核；
- 2345 浏览器：以前是 IE 内核，现在是 IE + Blink 双内核；
- UC 浏览器内核：Webkit + Trident；

3.浏览器的渲染原理

1).首先解析收到的文档，根据文档定义构建一颗 DOM 树，DOM 树是由 DOM 元素及属性节点组成的；

2)然后对 CSS 进行解析，生成 CSSOM 规则树；

3).根据 DOM 树和 CSSOM 规则树构建 Render Tree。渲染树的节点被称为渲染对象，渲染对象是一个包含有颜色和大小等属性的矩形，渲染对象和 DOM 对象相对应，但这种对应关系不是一对一的，不可见的 DOM 元素不会被插入渲染树。

4).当渲染对象被创建并添加到树中，它们并没有位置和大小，所以当浏览器生成渲染树以后，就会根据渲染树来进行布局（也可以叫做回流）。这一阶

段浏览器要做的事情就是要弄清楚各个节点在页面中的确切位置和大小。通常这一行为也被称为“自动重排”。

5).布局阶段结束后是绘制阶段，比那里渲染树并调用对象的 paint 方法将它们的内容显示在屏幕上，绘制使用 UI 基础组件。

为了更好的用户体验，渲染引擎会尽可能早的将内容呈现到屏幕上，并不会等到所有的 html 解析完成之后再去构建和布局 render tree。它是解析完一部分内容就显示一部分内容，同时可能还在网络下载其余内容。

4.HTML5 的 form 的自动完成功能是什么？

autocomplete 属性规定输入字段是否应该启用自动完成功能，默认为启用，设置为 autocomplete=off 可以关闭该功能。自动完成允许浏览器预测对字段的输入。在用户在字段开始键入时，浏览器基于之前键入过的值，应该显示出在字段中填写的选项。

5.如何实现浏览器内多个标签页之间的通信？

实现多个标签页之间的通信，本质上都是通过中介者模式来实现的。因为标签页之间没有办法直接通信，因此我们可以找一个中介者来让标签页和中介者进行通信，然后让这个中介者来进行消息的转发。

1).使用 Websocket，通信的标签页连接同一个服务器，发送消息到服务器后，服务器推送消息给所有连接的客户端；

2).可以地调用 localStorage，localStorage 在另一个浏览上下文里被添加、修改或删除时，它都会触发一个 storage 事件，我们可以通过监听 storage 事件，控制它的值来进行页面信息通信；

3).如果我们能够获得对应标签页的引用，通过 postMessage 方法也是可以实现多个标签页通信的；

6.简述前端性能优化

页面内容方面

1).通过文件合并、css 雪碧图、使用 base64 等方式来减少 HTTP 请求数，避免过多的请求造成等待的情况；

2).通过 DNS 缓存等机制来减少 DNS 的查询次数；

3).通过设置缓存策略，对常用不变的资源进行缓存；

4).通过延迟加载的方式，来减少页面首屏加载时需要请求的资源，延迟加载的资源当用户需要访问时，再去请求加载；

5).通过用户行为，对某些资源使用预加载的方式，来提高用户需要访问资源时的响应速度；

服务器方面

1).使用 CDN 服务，来提高用户对于资源请求时的响应速度；

2).服务器端自用 Gzip、Deflate 等方式对于传输的资源进行压缩，减少传输文件的体积；

3).尽可能减小 cookie 的大小，并且通过将静态资源分配到其他域名下，来避免对静态资源请求时携带不必要的 cookie；

7.什么是 webp？

WebP 是谷歌开发的一种新图片格式，它是支持有损和无损两种压缩方式的使用直接色的点阵图。使用 webp 格式的最大优点是是，在相同质量的文件下，它拥有更小的文件体积。因此它非常适合于网络图片的传输，因为图片体积的减少，意味着请求时间的减少，这样会提高用户的体验。这是谷歌开发的一种新的图片格式。

浏览器如何判断是否支持 webp 格式图片？

通过创建 Image 对象，将其 src 属性设置为 webp 格式的图片，然后在 onload 事件中获取图片的宽高，如果能够获取，则说明浏览器支持 webp 格式图片。如果不能获取或者处罚了 onerror 函数，那么就说明浏览器不支持 webp 格式的图片。

二. CSS 篇

1.介绍下 BFC 及其应用

BFC（Block Format Context）块级格式化上下文，是页面盒模型中的一种 css 渲染模式，相当于一个独立的容器，里面的元素和外部的元素相互不影响。

创建 BFC 的方式有：

1）.html 根元素

2).float 浮动

3).绝对定位

4).overflow 不为 visible

5).display 为表格布局或者弹性布局；

BFC 主要的作用是：

1).清除浮动

2).防止同一 BFC 容器中的相邻元素间的外边距重叠问题

2.怎么让一个 div 水平垂直居中？

```
<div class="parent">
  <div class="child"></div></div>
<!-- 1 -->
div.parent {
  display: flex;
  justify-content: center;
  align-items: center;
}

<!-- 2 -->
div.parent {
  position: relative;
}div.child {
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%, -50%);
```

```
}


<!-- 3 -->
div.parent {
  display: grid;
}div.child {
  justify-self: center;
  align-self: center;
}


<!-- 4 -->
div.parent {
  font-size: 0;
  text-align: center;
  &::before {
    content: "";
    display: inline-block;
    width: 0;
    height: 100%;
    vertical-align: middle;
  }
}div.child {
  display: inline-block;
  vertical-align: middle;
}
```

2.怎么让一个 div 水平垂直居中？

```
<div class="parent">
  <div class="child"></div></div>
<!-- 1 -->
div.parent {
  display: flex;
  justify-content: center;
  align-items: center;
}

<!-- 2 -->
div.parent {
  position: relative;
}div.child {
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%, -50%);
}

<!-- 3 -->
div.parent {
  display: grid;
}div.child {
  justify-self: center;
```

```
  align-self: center;
}

<!-- 4 -->
div.parent {
  font-size: 0;
  text-align: center;
  &::before {
    content: "";
    display: inline-block;
    width: 0;
    height: 100%;
    vertical-align: middle;
  }
}div.child {
  display: inline-block;
  vertical-align: middle;
}
```

3.介绍下重绘和回流（Repaint & Reflow），以及如何进行优化？

浏览器渲染机制

- 浏览器采用流式布局模型（Flow Based Layout）；

- 浏览器会把 HTML 解析成 DOM，把 CSS 解析成 CSSOM,DOM 和 CSSOM 合并就产生了渲染树（Render Tree）；
- 有了 RenderTree，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上；
- 由于浏览器使用流式布局，对 Render Tree 的计算通常只需要遍历一次就可以完成，但 table 及其内部元素除外，他们可能需要多次计算，通常要花 3 倍于同等元素的时间，这也是为什么要避免使用 table 布局的原因之一；

重绘

由于节点的集合属性发生改变或者由于样式改变而不会影响布局的，成为重绘，例如 outline、visibility、color、background-color 等，重绘的代价是高昂的，因此浏览器必须验证 DOM 树上其他节点元素的可见性。

回流

回流是布局或者几何属性需要改变就称为回流。回流是影响浏览器性能的关键因素，因为其变化涉及到部分页面（或是整个页面）的布局更新。一个元素的回流可能会导致其素有子元素以及 DOM 中紧随其后的节点、祖先节点元素的随后的回流。大部分的回流将导致页面的重新渲染。

回流必定会发生重绘，重绘不一定会引发回流。

浏览器优化

现代浏览器大多是通过队列机制来批量更新布局，浏览器会把修改操作放在队列中，至少一个浏览器刷新（即 16.6ms）才会清空队列，但当你获取布局信息的时候，队列中可能会有影响这些属性或方法返回值的操作，即使没有，浏览器也会强制清空队列，触发回流和重绘来确保返回正确的值。

例如 offsetTop、clientTop、scrollTop、getComputedStyle()、width、height、getBoundingClientRect()，应避免频繁使用这些属性，他们都会强制渲染刷新队列。

减少重绘和回流

1. CSS

- 使用 transform 代替 top；
- 使用 visibility 替换 display: none，前者引起重绘，后者引发回流；
- 避免使用 table 布局；
- 尽可能在 DOM 树的最末端改变 class；
- 避免设置多层内联样式，CSS 选择符从右往左匹配查找，避免节点层级过多；
- 将动画效果应用到 position 属性为 absolute 或 fixed 的元素上，避免影响其他元素的布局；
- 避免使用 CSS 表达式，可能会引发回流；
- CSS 硬件加速；

1. Javascript

- 避免频繁操作样式，修改 class 最好；
- 避免频繁操作 DOM，合并多次修改为一次；
- 避免频繁读取会引发回流/重绘的属性，将结果缓存；
- 对具有复杂动画的元素使用绝对定位，使它脱离文档流；

4.分析比较 opacity: 0、visibility: hidden、display: none 优劣和适用场景

1).display: none - 不占空间，不能点击，会引起回流，子元素不影响

2).visibility: hidden - 占据空间，不能点击，引起重绘，子元素可设置 visible 进行显示

3).opacity: 0 - 占据空间，可以点击，引起重绘，子元素不影响

5.简述 CSS 盒模型

盒子由 margin、border、padding、content 组成；

标准盒模型：box-sizing: content-box;

IE 盒模型：box-sizing: border-box;

6.简述 Rem 及其转换原理

rem 是 CSS3 新增的相对长度单位，是指相对于根元素 html 的 font-size 计算值的大小。

默认根元素的 font-size 都是 16px 的。如果想要设置 12px 的字体大小也就是 12px/16px = 0.75rem。

- 由于 px 是相对固定单位，字号大小直接被定死，无法随着浏览器进行缩放；
- rem 直接相对于根元素 html，避开层级关系，移动端新型浏览器对其支持较好；

个人用 vw + 百分比布局用的比较多，可以使用 webpack 的 postcss-loader 的一个插件 postcss-px-to-viewport 实现对 px 到 vw 的自动转换，非常适合开发。

7.移动端视口配置

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0, user-scalable=no" />
```

- initial-scale: 初始的缩放比例；
- minimum-scale: 允许用户缩放到的最小比例；
- maximum-scale: 允许用户缩放到的最大比例；
- user-scalable: 用户是否可以手动缩放；

8.简述伪类和伪元素

伪类

伪类用于当已有元素处于某种状态时，为其添加对应的样式，这个状态是根据用户行为变化而变化的。比如说 :hover。它只有处于 dom 树无法描述的状态才能为元素添加样式，所以称为伪类。

伪元素

伪元素用于创建一些原本不在文档树中的元素，并为其添加样式，比如说 ::before。虽然用户可以看到这些内容，但是其实他不在文档树中。

区别

伪类的操作对象是文档树中已存在的元素，而伪元素是创建一个文档树外的元素。

css 规范中用双冒号 :: 表示伪元素，用一个冒号 : 表示伪类。

9.行内元素的 margin 和 padding

- 水平方向：水平方向上，都有效；
- 垂直方向：垂直方向上，都无效；（padding-top 和 padding-bottom 会显示出效果，但是高度不会撑开，不会对周围元素有影响）

10.CSS 中哪些属性可以继承？

1).字体系列属性

- font-family
- font-size
- font-weight
- font-style

2).文本系列属性

- text-indent
- text-align
- line-hight
- word-spacing
- letter-spacing
- color

3) .其他

- cursor
- Visibility

11.CSS3 新增伪类有那些？（例如 nth-child）

- elem:nth-child(n): 选中父元素下的第 n 个标签名为 elem 的元素；
- elem:nth-last-child(n): 作用同上，从后开始查找；
- elem:last-child：最后一个子元素
- elem:only-child：如果 elem 是父元素下唯一的子元素，则选中；
- elem:nth-of-type(n)：选择父元素下第 n 个 elem 类型元素；
- elem:empty：选中不包含子元素和内容的 elem 类型元素；
- :not(elem)：选择非 elem 元素的每个元素；
- :enabled：启用状态的表单组件

12.用纯 CSS 创建一个三角形

```
#demo {
  width: 0;
  height: 0;
  border-width: 20;
  border-style: "solid";
  border-color: transparent transparent red transparent;
}
```

原理是相邻边框连接处是均分的原理。

13.min-width/max-width 和 min-height/max-height 属性间的覆盖规则？

1).max-width 会覆盖 width，即使 width 是行内样式或者设置了 !important。

2).min-width 会覆盖 max-width，此规则发生在 min-width 和 max-width 冲突的时候；

三. Javascript 篇

1.Vue 的响应式原理中 Object.defineProperty 有什么缺陷？为什么在 Vue3.0 采用了 Proxy，抛弃了 Object.defineProperty？

原因如下：

1）.Object.defineProperty 无法低耗费的监听到数组下标的变化，导致通过数组下标添加元素，不能实时响应；

2).Object.defineProperty 只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历。如果属性值是对象，还需要深度遍历。 Proxy 可以劫持整个对象， 并返回一个新的对象。

3).Proxy 不仅可以代理对象，还可以代理数组。还可以代理动态增加的属性。

2.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么？

vue 和 react 都是采用 diff 算法来对比新旧虚拟节点，从而更新节点。在 vue 的 diff 函数交叉对比中，当新节点跟旧节点头尾交叉对比没有结果时，会根据新节点的 key 去对比旧节点数组中的 key，从而找到相应旧节点（这里对应的是一个 key => index 的 map 映射）。如果没有找到就认为是一个新增节点。而如果没有 key，那么就会采用遍历查找的方式去找到对应的旧节点。一种一个 map 映射，另一种是遍历查找。相比而言，map 映射的速度更快。

3.['1', '2', '3'].map(parseInt) what & why ?

`['1', '2', '3'].map(parseInt)` 的输出结果为 [1, NaN, NaN]。

因为 `parseInt(string, radix)` 将一个字符串 `string` 转换为 `radix` 进制的整数， `radix` 为介于 `2-36` 之间的数。

在数组的 `map` 方法的回调函数中会传入 `item`（遍历项） 和 `index`（遍历下标) 作为前两个参数，所以这里的 `parseInt` 执行了对应的三次分别是

- parseInt(1, 0)
- parseInt(2, 1)
- parseInt(3, 2)

对应的执行结果分别为 `1`、`NaN`、`NaN`。

4.（挖财）什么是防抖和节流？有什么区别？如何实现？

防抖

触发高频事件后 n 秒内函数只会执行一次，如果 n 秒内高频事件再次被触发，则重新计算时间。

```
function debounce(fn, timing) {
  let timer;
  return function() {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn();
    }, timing);
  }}
```

节流

高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的执行效率。

```
function throttle(fn, timing) {
  let trigger;
  return function() {
    if (trigger) return;
    trigger = true;
    fn();
    setTimeout(() => {
```

```
    trigger = false;
  }, timing);
}}
```

Tips：我记这个很容易把两者弄混，总结了个口诀，就是 DTTV（Debounce Timer Throttle Variable - 防抖靠定时器控制，节流靠变量控制）。

5.介绍下 Set、Map、WeakSet 和 WeakMap 的区别？

Set

1).成员不能重复；

2).只有键值，没有键名，有点类似数组；

3).可以遍历，方法有 `add`、`delete`、`has`

WeakSet

1).成员都是对象（引用）；

2).成员都是弱引用，随时可以消失（不计入垃圾回收机制）。可以用来保存 DOM 节点，不容易造成内存泄露；

3).不能遍历，方法有 `add`、`delete`、`has`；

Map

1).本质上是键值对的集合，类似集合；

2).可以遍历，方法很多，可以跟各种数据格式转换；

WeakMap

1).只接收对象为键名（null 除外），不接受其他类型的值作为键名；

2).键名指向的对象，不计入垃圾回收机制；

3).不能遍历，方法同 get、set、has、delete；

6.ES5/ES6 的继承除了写法以外还有什么区别？

1).class 声明会提升，但不会初始化赋值。（类似于 let、const 声明变量；

2).class 声明内部会启用严格模式；

3).class 的所有方法（包括静态方法和实例方法）都是不可枚举的；

4).class 的所有方法（包括静态方法和实例方法）都没有原型对象 prototype，所以也没有 [[constructor]]，不能使用 new 来调用；

5).必须使用 new 来调用 class；

6).class 内部无法重写类名；

7.setTimeout、Promise、Async/Await 的区别

setTimeout: setTimeout 的回调函数放到宏任务队列里，等到执行栈清空以后执行；

Promise: Promise 本身是同步的立即执行函数，当在 executor 中执行 resolve 或者 reject 的时候，此时是异步操作，会先执行 then/catch 等，当主栈完成时，才会去调用 resolve/reject 方法中存放的方法。

async: async 函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

8.（头条、微医）Async/Await 如何通过同步的方式实现异步

Async/Await 是一个自执行的 generate 函数。利用 generate 函数的特性把异步的代码写成"同步"的形式。

```
var fetch = require("node-fetch");

function *gen() { // 这里的 * 可以看成 async

  var url = "https://api.github.com/users/github";

  var result = yield fetch(url); // 这里的 yield 可以看成 await

  console.log(result.bio);}

var g = gen();var result = g.next();result.value.then(data =>
data.json()).then(data  => g.next(data));
```

9.简述一下 Generator 函数

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做“协程”（coroutine），意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像线程，它的运行流程大致如下：

- 第一步，协程 A 开始执行；
- 第二步，协程 A 执行到一半，进入暂停，执行权转移到协程 B；
- 第三步，（一段时间后）协程 B 交还执行权；
- 第四步，协程 A 恢复执行；

上面流程的协程 A，就是异步任务，因为它分成两段（或多段）执行。

举例来说，读取文件的协程写法如下：

```
function* asyncJob() {
 // ...
 var f = yield readFile(fileA);
 // ...}
```

上面代码的函数 asyncJob 是一个协程，它的奥妙就在其中的 yield 命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield 命令是异步两个阶段的分界线。协程遇到 yield 命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。

Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。

```
function* gen(x) {
  var y = yield x + 2;
  return y;}
```

```
var g = gen(1);g.next() // { value: 3, done: false }g.next(2) // { value: 2, done: false }
```

next 是返回值的 value 属性，是 Generator 函数向外输出数据；next 方法还可以接受参数，向 Generator 函数体内输入数据。

上面代码中，第一个 next 方法的 value 属性，返回表达式 x + 2 的值 3。第二个 next 方法带有参数 2，这个参数可以传入 Generator 函数，作为 上个阶段 异步任务的返回结果，被函数体内的变量 y 接收。因此，这一步的 value 属性，返回的就是 2（变量 y 的值）。

10.（滴滴、挖财、微医、海康）JS 异步解决方案的发展历程以及优缺点。

回调函数

优点：解决了同步的问题（整体任务执行时长）；

缺点：回调地狱，不能用 try catch 捕获错误，不能 return;

Promise

优点：解决了回调地狱的问题；

缺点：无法取消 Promise，错误需要通过回调函数来捕获；

Generator

特点：可以控制函数的执行。

Async/Await

优点：代码清晰，不用像 Promise 写一大堆 then 链，处理了回调地狱的问题；

缺点：await 将异步代码改造成同步代码，如果多个异步操作没有依赖性而使用 await 会导致性能上的降低；

11.简述浏览器缓存读取规则

浏览器缓存可以优化性能，比如直接使用缓存而不发起请求，或者发起了请求但后端存储的数据和前端一致，则使用缓存从而减少响应数据。

缓存位置

Service Worker

Service Worker 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。

使用 Service Worker 的话，传输协议必须为 HTTPS。Service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由缓存哪些文件、如何匹配缓存、如何读取缓存，而缓存是可持续性的。Service Worker 也是 PWA 的核心技术。

Memory Cache

Memory Cache 也就是内存中的缓存，主要包含的是当前页面中已经抓取到的资源，例如页面上已经下载的样式、脚本、图片等。读取内存中的数据很高效，但是缓存持续性很短，会随着进程的释放而释放。一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。

Disk Cache

Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。

在所有浏览器缓存中，Disk Cache 覆盖面基本上是最大的。它会根据 HTTP Header 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据。绝大部分的缓存都来自 Disk Cache。

Push Cache

Push Cache（推送缓存）是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被时候用。它只在会话（Session）中存在，一旦会话结束就被释放，并且缓存时间也很短暂（大约 5 分钟）。

缓存过程分析

浏览器与服务器通信的方式为应答模式，即是：浏览器发起 HTTP 请求 - 服务器响应该请求。浏览器第一次向服务器发起该请求后拿到请求结果后，将请求结果和缓存表示存入浏览器缓存，浏览器对于缓存的处理是根据第一次请求资源返回的响应头来确定的。

- 浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识；
- 浏览器每次拿到返回的请求结果都会将该结果和缓存表示存入浏览器缓存中；

12.为什么 Vuex 的 mutation 和 Redux 的 reducer 中不能做异步操作？

纯函数，给定同样的输入返回同样的输出，可预测性。

13.（京东）下面代码中 a 在什么情况下会打印 1？

```
var a = ?;if(a == 1 && a == 2 && a == 3){

        console.log(1);}
```

解答：

```
var a = {
  value: 0,

  valueOf() {
    return ++this.value;
  }};
```

14.在 Vue 中，子组件为何不可以修改父组件传递的 Prop，如果修改了，Vue 是如何监控到属性的修改并给出警告的。

1.因为 Vue 是单项数据流，易于检测数据的流动，出现了错误可以更加迅速的定位到错误发生的位置；

2.通过 setter 属性进行检测，修改值将会触发 setter，从而触发警告；

15. 实现一个 sleep 函数

比如 sleep(1000) 意味着等待 1000 毫秒，可从 Promise、Generator、Async/Await 等角度实现

```
// Promise
function sleep1(time) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve();
    }, time);
  })}
sleep1(1000).then(() => console.log("sleep1"));
```

```
// Generator
function* sleep2(time) {
  return yield sleep1(time);}
const s = sleep2(1500);s.next().value.then(() => console.log("sleep2"));

// Async/Await
async function sleep3(time) {
  await sleep1(time);}
(async () => {
  await sleep3(2000);
  console.log("sleep3")})()
```

16.双向绑定和 vuex 是否冲突

当在严格模式中使用 Vuex 时，在属于 Vuex 的 state 上使用 v-model 会导致出错。

解决方案：

1)给 <Input> 中绑定 value，然后侦听 input 或者 change 事件，在事件回调中调用一个方法；

2).使用带有 setter 的双向绑定计算属性；

17.call 和 apply 的区别是什么，哪个性能更好一些

1)Function.prototype.apply 和 Function.prototype.call 的作用是一样的，区别在于传入参数的不同；

2).第一个参数都是指定函数体内 this 的指向；

3)第二个参数开始不同，apply 是传入带下标的集合，数组或者类数组，apply 把它传给函数作为参数，call 从第二个开始传入的参数是不固定的，都会传给函数作为参数；

4)call 比 apply 的性能要好，call 传入参数的格式正式内部所需要的格式；

18.为什么通常在发送数据埋点请求的时候使用的是 1x1 像素的透明 gif 图片？

1）能够完成整个 HTTP 请求+响应（尽管不需要响应内容）；

2）.触发 GET 请求之后不需要获取和处理数据，服务器也不需要发送数据；

3).跨域友好；

4).执行过程无阻塞；

5).相比 XMLHttpRequest 对象发送 GET 请求，性能上更好；

6).GIF 的最低合法体积最小（合法的 GIF 只需要 43 个字节）

19.（百度）实现 (5).add(3).minus(2) 功能

```
Number.prototype.add = function(n) {
  return this + n;}
Number.prototype.minus = function(n) {
  return this - n;}
```

20.操作题

某公司 1 到 12 月份的销售额存在一个对象里面 如下：{1:222, 2:123, 5:888}，

请把数据处理为如下结构：[222, 123, null, null, 888, null, null, null, null, null, null, null]

解答：

```
function convert(obj) {

  return Array.from({ length: 12 }).map((item, index) => obj[index] || null).slice(1);}
```

21.操作题

要求设计 LazyMan 类，实现以下功能。

```
LazyMan('Tony');// Hi I am Tony

LazyMan('Tony').sleep(10).eat('lunch');// Hi I am Tony// 等待了 10 秒...// I am eating lunch

LazyMan('Tony').eat('lunch').sleep(10).eat('dinner');// Hi I am Tony// I am eating lunch// 等待了 10 秒...// I am eating diner

LazyMan('Tony').eat('lunch').eat('dinner').sleepFirst(5).sleep(10).eat('junk food');// Hi I am Tony// 等待了 5 秒...// I am eating lunch// I am eating dinner// 等待了 10 秒...// I am eating junk food
```

解答：

查看同目录下 test.js；

22.箭头函数与普通函数（function）的区别是什么？构造函数（function）可以使用 new 生成实例，那么箭头函数可以吗？为什么？

箭头函数是普通函数的简写，可以更优雅的定义一个函数，和普通函数相比，有以下几点差异：

1).函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象；

2).不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替；

3).不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数；

4).不可以使用 new 命令，因为：

A.没有自己的 this，无法调用 call、apply；

B.没有 prototype 属性，而 new 命令在执行时需要将钩子函数的 prototype 赋值给新的对象的 __proto__

23.redux 为什么要把 reducer 设计成纯函数

redux 的设计思想就是不产生副作用，数据更改的状态可回溯，所以 redux 中处处都是纯函数。

24.ES6 代码转成 ES5 代码的实现思路是什么？

Babel 的实现方式：

1).将代码字符串解析成抽象语法树，即所谓的 AST；

2).对 AST 进行处理，在这个阶段可以对 ES6 AST 进行相应转换，即转换成 ES5 AST；

3).根据处理后的 AST 再生成代码字符串；

25.Vue 的父组件和子组件生命周期钩子执行顺序是什么

1. 加载渲染过程：父 beforeCreate -> 父 created -> 父 beforeMount -> 子 beforeCreate -> 子 created -> 子 beforeMount -> 子 mounted -> 父 mounted；
2. 子组件更新过程：父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated；
3. 父组件更新过程：父 beforeUpdate -> 父 updated；
4. 销毁过程：父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed；

26. react-router 里的 标签和 标签有什么区别

1. 有 onClick 则执行 OnClick；
2. 阻止 a 标签默认事件（跳转页面）；
3. 在取得跳转 href（to 属性值），用 history/hash 跳转，此时只是链接发现改变，并没有刷新页面；

27.vue 在 v-for 时给每项元素绑定事件需要用事件代理吗？为什么？

在 v-for 中使用事件代理可以使监听器数量和内存占用率都减少，vue 内部并不会自动做事件代理，所以在 v-for 上使用事件代理在性能上会更优。

28.谈谈对 MVC、MVP、MVVM 模式的理解

在开发图形界面应用程序的时候，会把管理用户界面的层次称为 View，应用程序的数据为 Model，Model 提供数据操作的接口，执行相应的业务逻辑。

MVC

MVC 除了把应用程序分为 View、Model 层，还额外的加了一个 Controller 层，它的职责是进行 Model 和 View 之间的协作（路由、输入预处理等）的应由逻辑（application logic）；Model 进行处理业务逻辑。

用户对 View 操作以后，View 捕获到这个操作，会把处理的权利交移给 Controller（Pass calls）；Controller 会对来自 View 数据进行预处理、决定调用哪个 Model 的接口；然后由 Model 执行相关的业务逻辑；当 Model 变更了以后，会通过观察者模式（Observer Pattern）通知 View；View 通过观察者模式收到 Model 变更的消息以后，会向 Model 请求最新的数据，然后重新更新界面。

MVP

和 MVC 模式一样，用户对 View 的操作都会从 View 交易给 Presenter。Presenter 会执行相应的应用程序逻辑，并且会对 Model 进行相应的操作；而这时候 Model 执行业务逻辑以后，也是通过观察者模式把自己变更的消息传递出去，但是是传给 Presenter 而不是 View。Presenter 获取到 Model 变更的消息以后，通过 View 提供的接口更新界面。

MVVM

MVVM 可以看做是一种特殊的 MVP（Passive View）模式，或者说是对 MVP 模式的一种改良。

MVVM 代表的是 Model-View-ViewModel，可以简单把 ViewModel 理解为页面上所显示内容的数据抽象，和 Domain Model 不一样，ViewModel 更适合用来描述 View。 MVVM 的依赖关系和 MVP 依赖关系一致，只不过是把 P 换成了 VM。

MVVM 的调用关系：

MVVM 的调用关系和 MVP 一样。但是，在 ViewModel 当中会有一个叫 Binder，或者是 Data-binding engine 的东西。以前全部由 Presenter 负责的 View 和 Model 之间数据同步操作交由给 Binder 处理。你只需要在 View 的模板语法当中，指令式声明 View 上的显示的内容是和 Model 的哪一块数据绑定的。当 ViewModel 对进行 Model 更新的时候，Binder 会自动把数据更新到 View 上，当用户对 View 进行操作（例如表单输入），Binder 也会自动把数据更新到 Model 上。这种方式称为：Two-way data-binding，双向数据绑定。可以简单而不恰当地理解为一个模板引擎，但是会根据数据变更实时渲染。

29.简单说说 js 中有哪几种内存泄露的情况

1. 意外的全局变量；
2. 闭包；
3. 未被清空的定时器；
4. 未被销毁的事件监听；
5. DOM 引用；

30.跨域问题如何解决

1. JSONP
2. CORS（Cross-Origin-Resource-Share，跨域资源共享），由服务端设置响应头通过浏览器的同源策略限制
    1. Access-Control-Allow-Origin: *;
    2. Access-Control-Allow-Methods: *;
    3. Access-Control-Allow-Headers: *;
    4. Access-Control-Allow-Credentials: true;

31.instanceof 的实现原理

```
while (x.__proto__) {
  if (x.__proto__ === y.prototype) {
    return true;
  }
  x.__proto__ = x.__proto__.__proto__;}if (x.__proto__ === null) {
  return false;}
```

32.react 组件的生命周期

初始化阶段

1. constructor(): 用于绑定事件，初始化 state
2. componentWillMount()：组件将要挂载，在 render 之前调用，可以在服务端调用。
3. render()：用作渲染 dom；

4. componentDidMount()：在 render 之后，而且是所有子组件都 render 之后才调用。

更新阶段

1. getDerivedStateFromProps：getDerivedStateFromProps 会在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新 state，如果返回 null 则不更新任何内容；
2. componentWillReceiveProps(nextProps): 在这里可以拿到即将改变的状态，可以在这里通过 setState 方法设置 state
3. shouldComponentUpdate(nextProps, nextState): 他的返回值决定了接下来的声明周期是否会被调用，默认返回 true
4. componentWillUpdate(): 不能在这里改变 state，否则会陷入死循环
5. componentDidUpdate(): 和 componentDidMount()类似，在这里执行 Dom 操作以及发起网络请求

析构阶段

1. componentWillUnmount()：主要执行清除工作，比如取消网络请求，清除事件监听。

33.简述 Flux 思想

Flux 最大的特点就是，数据单向流动

1. 用户访问 View

2. View 发出用户的 Action
3. Dispatcher 收到 Action，要求 Store 进行对应的更新；
4. Store 更新后，发出一个 “change” 事件；

34.简述执行上下文和执行栈

执行上下文

- 全局执行上下文：默认的上下文，任何不在函数内部的代码都在全局上下文里面。它会执行两件事情：创建一个全局的的 window 对象，并且设置 this 为这个全局对象。一个程序只有一个全局对象。
- 函数执行上下文：每当一个函数被调用时，就会为该函数创建一个新的上下文，每个函数都有自己的上下文，不过是在被函数调用的时候创建的。函数上下文可以有任意多个，每当一个新的执行上下文被创建，他会按照定义的顺序执行一系列的步骤。
- Eval 函数执行上下文：执行在 `eval` 函数内部的代码有他自己的执行上下文。

执行栈

执行栈就是一个调用栈，是一个后进先出数据结构的栈，用来存储代码运行时创建的执行上下文。

this 绑定

全局执行上下文中，`this` 指向全局对象。

函数执行上下文中，this 取决于函数是如何被调用的。如果他被一个引用对象调用，那么 `this` 会设置成那个对象，否则是全局对象。

5. View 收到 “change” 后，更新页面；

35.什么是 CSP？

CSP（Content-Security-Policy）指的是内容安全策略，它的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截由浏览器自己来实现。

通常有两种方式来开启 CSP，一种是设置 HTTP 首部中的 Content-Security-Policy，一种是设置 meta 标签的方式 `<meta http-equiv="Content-Security-Policy">`

CSP 也是解决 XSS 攻击的一个强力手段。

36.什么是 CSRF 攻击？如何防范 CSRF 攻击？

CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态（cookie），绕过后台的用户验证，冒充用户向服务器执行一些操作。

CSRF 攻击的本质是利用了 cookie 会在同源请求中携带发送给服务器的特点，以此来实现用户的冒充。

防护方法：

1. 同源检测，服务器检测请求来源；
2. 使用 token 来进行验证；
3. 设置 cookie 时设置 Samesite，限制 cookie 不能作为被第三方使用；

37.谈一谈你理解的函数式编程？

“函数式变成”是一种“编程范式”，也就是如何编写程序的方法论。

它具有以下特性：闭包和高阶函数、惰性运算、递归、函数是“第一等公民”、只用“表达式”。

38.什么是尾调用，使用尾调用有什么好处？

尾调用指的是函数的最后一步调用另一个函数。我们代码执行是基于执行栈的，所以当我们在一个函数里调用另一个函数时，我们会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话，因为已经是函数的最后一步，所以这个时候我们可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

39.Vue 组件间如何通信？

父子组件通信

1. props + emit
2. $refs + $parent
3. provider/inject

兄弟组件通信

1. eventBus
2. $parent.$refs

40.Vue 中 computed 和 watch 的差异？

1. computed 是计算一个新的属性，并将该属性挂载到 Vue 实例上，而 watch 是监听已经存在且已挂载到 Vue 实例上的数据，所以用 watch 同样可以监听 computed 计算属性的变化；
2. computed 本质是一个惰性求值的观察者，具有缓存性，只有当依赖变化后，第一次访问 computed 值，才会计算新的值。而 watch 则是当数据发送变化便会调用执行函数；
3. 从使用场景上来说，computed 适用一个数据被多个数据影响，而 watch 使用一个数据影响多个数据。

41.简述一下 PWA

PWA（Progressive Web App）渐进式网页应用，目的是提升 Web App 的性能，改善 Web App 的用户体验。

特点

1. 可安装：可以像原生 APP 在主屏幕上留有图标。
2. 离线应用：可以离线使用，背后用的是技术是 Service Worker
3. Service Worker 实际上是一段脚本，在后台运行。作为一个独立的线程，运行环境和普通脚本不同，所以不能直接参与 Web 交互行为，属于一种客户端代理。
4. Service Worker 可以创建有效的离线体验，拦截网络请求，并根据网络是否可用判断是否使用缓存数据或者更新缓存数据。
5. 消息推送

42.（阿里巴巴）介绍下 CacheStorage

CacheStorage 接口表示 Cache 对象的存储。它提供了一个 ServiceWorker、其他类型 woker 或者 window 范围内可以访问到的所有命名 cache 的主目录（它并不是一定要和 service workers 一起使用，即使它是在 service workers 规范中定义的），并维护一份字符串名称到相应 Cache 对象的映射。

CacheStorage 和 Cache，是两个与缓存相关的接口，用于管理当前网页/Web App 的缓存；在使用 Service Worker 时基本都会用到。它们与数据库有点类似，我们可以用 mongodb 来打个比喻：

- CacheStorage 管理者所有的 Cache，是整个缓存 api 的入口，类似于 mongo；
- Cache 是单个缓存库，通常一个 app 会有一个，类似 mongo 里的每个 db；

无论在 ServiceWorker 域或 window 域下，你都可以用 caches 来访问全局的 CacheStorage。

43.（阿里巴巴）Vue 双向数据绑定原理

vue 通过双向数据绑定，来实现了 View 和 Model 的同步更新。vue 的双向数据绑定主要是通过数据劫持和发布订阅者模式来实现的。

首先我们通过 Object.defineProperty() 方法来对 Model 数据各个属性添加访问器属性，以此来实现数据的劫持，因此当 Model 中的数据发生变化的时候，我们可以通过配置的 setter 和 getter 方法来实现对 View 层数据更新的通知。

对于文本节点的更新，我们使用了发布订阅者模式，属性作为一个主题，我们为这个节点设置一个订阅者对象，将这个订阅者对象加入这个属性主题的订阅者列表中。当 Model 层数据发生改变的时候，Model 作为发布者向主题发出通知，主题收到通知再向它的所有订阅者推送，订阅者收到通知后更改自己的数据。

44.页面的可用性时间的计算

Performance 接口可以获取到当前页面中与性能相关的信息。

- Performance.timing：`Performance.timing` 对象包含延迟相关的性能信息；

45. 简述一下 WebAssembly

WebAssembly 是一种新的编码方式，可以在现代的网络浏览器中运行 - 它是一种低级的类汇编语言，具有紧凑的二进制格式，可以接近原生的性能运行，并为诸如 C/C++ 等语言提供一个编译目标，以便它们可以在 Web 上运行。它也被设计为可以与 Javascript 共存，允许两者一起工作。

WebAssembly 提供了一条途径，以使得以各种语言编写的代码都可以以接近原生的速度在 Web 中运行。

46.（阿里巴巴）谈谈移动端点击

移动端 300 ms 点击（click 事件）延迟

由于移动端会有双击缩放的这个操作，因此浏览器在 click 之后要等待 300ms，判断这次操作是不是双击。

解决方案：

1. 禁用缩放：user-scalable=no
2. 更改默认的视口宽度
3. CSS touch-action

点击穿透问题

因为 click 事件的 300ms 延迟问题，所以有可能会在某些情况触发多次事件。

解决方案：

1. 只用 touch
2. 只用 click

47.（阿里巴巴）谈谈 Git-Rebase

1. 可以合并多次提交记录，减少无用的提交信息；
2. 合并分支并且减少 commit 记录；

48.（阿里巴巴）简述懒加载

懒加载也叫延迟加载，指的是在长网页中延迟加载图像，是一种很好优化网页性能的方式。

懒加载的优点：

1. 提升用户体验，加快首屏渲染速度；
2. 减少无效资源的加载；
3. 防止并发加载的资源过多会阻塞 js 的加载；

懒加载的原理：

首先将页面上的图片的 src 属性设为空字符串，而图片的真实路径则设置

在 data-original 属性中，当页面滚动的时候需要去监听 scroll 事件，

在 scroll 事件的回调中，判断我们的懒加载的图片是否进入可视区域，如果图

片在可视区内则将图片的 src 属性设置为 data-original 的值，这样就可以实现延迟加载。

49.（腾讯）webpack 中 loader 和 plugin 的区别是什么？

loader：loader 是一个转换器，将 A 文件进行编译成 B 文件，属于单纯的文件转换过程；

plugin：plugin 是一个扩展器，它丰富了 webpack 本身，针对是 loader 结束后，webpack 打包的整个过程，它并不直接操作文件，而是基于事件机制工作，会监听 webpack 打包过程中的某些节点，执行广泛的任务。

四. 函数执行结果

1.第一题（考察 This 指针）

请给出如下代码的打印结果（答案在最下面）：

```
function Foo() {
    Foo.a = function() {
        console.log(1)
    }
    this.a = function() {
        console.log(2)
    }}Foo.prototype.a = function() {
    console.log(3)}Foo.a = function() {
    console.log(4)}Foo.a();let obj = new Foo(); obj.a();Foo.a();
```

打印结果： 4 2 1

2.第二题（考察对象应用）

请给出如下代码的打印结果（答案在最下面）：

```
function changeObjProperty(o) {

  o.siteUrl = "http://www.baidu.com"

  o = new Object()

  o.siteUrl = "http://www.google.com"} let webSite = new
Object();changeObjProperty(webSite);console.log(webSite.siteUrl);
```

3.第三题（考察事件循环/异步）

请写出下面代码的运行结果

```
async function async1() {

    console.log('async1 start');

    await async2();

    console.log('async1 end');}async function async2() {

    console.log('async2');}console.log('script start');setTimeout(function()
{

    console.log('setTimeout');}, 0)async1();new Promise(function(resolve) {

    console.log('promise1');

    resolve();}).then(function() {

    console.log('promise2');});console.log('script end');
```

运行结果：

- script start
- async1 start
- async2
- promise1
- script end

- async1 end
- promise2
- setTimeout

4.第四题（考察 React 的使用）

请写出下面代码的运行结果

```
class Example extends React.Component {
  constructor() {
    super();
    this.state = {
      val: 0
    };
  }

  componentDidMount() {
    this.setState({val: this.state.val + 1});
    console.log(this.state.val);    // 第 1 次 log

    this.setState({val: this.state.val + 1});
    console.log(this.state.val);    // 第 2 次 log

    setTimeout(() => {
      this.setState({val: this.state.val + 1});
      console.log(this.state.val);  // 第 3 次 log

      this.setState({val: this.state.val + 1});
```

```
    console.log(this.state.val);  // 第 4 次 log
  }, 0);
}

render() {
  return null;
}};
```

输出

- 0
- 0
- 2
- 3

解答：

1. 第一次和第二次都是在 react 自身声明周期内，触发时 `isBatchingUpdates` 为 `true`，所以并不会直接执行更新 `state`，而是加入了 `dirtyComponents`，所以打印的时获取的都是更新前的状态 0；
2. 两次 `setState`，获取到 `this.state.val` 都是 0，所以执行时都是将 0 设置成 1，在 `react` 内部会被合并掉，只执行一次，设置完成后 `state.val` 值为 1.
3. `setTimeout` 中的代码，触发时 `isBatchingUpdate` 为 `false`，所以能够直接进行更新，所以连着输出 2、3

5.第五题（考察作用域）

下面的代码打印什么内容，为什么？

```
var b = 10;(function b(){
    b = 20;
    console.log(b); })();
```

输出：

```
ƒ b(){
    b = 20;
    console.log(b); }
```

原因:

作用域：执行上下文中包含作用域链；

特性：声明提前：一个声明在函数体内都是可见的，函数声明优先于变量声明；

在非匿名自执行函数中，函数变量为只读状态无法修改。

6.第六题（考察作用域）

下面代码输出什么？

```
var a = 10;(function () {
    console.log(a)
    a = 5
    console.log(window.a)
    var a = 20;
    console.log(a)})()
```

输出 undefined 10 20

7.第七题（考察数组）

下面代码输出什么？

```
var obj = {
    '2': 3,
    '3': 4,
    'length': 2,
    'splice': Array.prototype.splice,
    'push': Array.prototype.push}obj.push(1)obj.push(2)console.log(obj)
```

输出：[,,1,2] length 为 4

解释：Array.prototype.push 将根据 length 将元素填充到对应位置并修改 length 属性 +1，所以输出的结果就是上述结果。

8.第八题（考察赋值表达式）

下面代码会输出什么？

```
var a = {n: 1};var b = a;a.x = a = {n: 2};
console.log(a.x)          console.log(b.x)
```

9.第九题（考察赋值表达式）

下面代码会输出什么？

```
// example 1var a={}, b='123', c=123;  a[b]='b';a[c]='c';  console.log(a[b]);
```

```
----------------------// example 2var a={}, b=Symbol('123'), c=Symbol('123');
a[b]='b';a[c]='c';  console.log(a[b]);

----------------------// example 3var a={}, b={key:'123'}, c={key:'456'};
a[b]='b';a[c]='c';  console.log(a[b]);
```

五．源码相关

1.如何实现函数的柯里化？

什么是函数柯里化？

把接收多个参数的函数变换为接收一个单一参数（最初函数的第一个参数）的函数，并返回接收剩余参数而且返回结果的新函数的技术。

JS 函数柯里化的优点：

- 可以延迟计算，即如果调用柯里化函数传入参数是不调用的，会将参数添加到数组中存储，等到没有参数传入的时候进行调用；
- 参数复用，当在多次调用同一个函数，并且传递的参数绝大多数是相同的，那么该函数可能是一个很好的柯里化候选；

怎么实现？

```
function curringAdd() {

  let args = [].slice.call(arguments, 0);


  function add() {

    args = [...args, [].slice.call(arguments, 0)];

    return add

  }


  add.toString = function() {
```

```
    return args.reduce((t, a) => t + +a, 0);
  }

  return add;}

console.log(curringAdd(1)(2)(3)) // 6console.log(curringAdd(1, 2, 3)(4))  //
10console.log(curringAdd(1)(2)(3)(4)(5)) // 15console.log(curringAdd(2, 6)(1))
// 9console.log(curringAdd(1)) // 1
```

2.手写 bind、call、apply

```
Function.prototype.MyCall = function (context) {
  const args = [...arguments].slice(1);

  context.fn = this;

  const result = context.fn(...args);
  delete context.fn;

  return result;}
Function.prototype.MyApply = function (context) {
  const args = arguments[1] || [];

  context.fn = this;
  const result = context.fn(...args);
  delete context.fn;

  return result;}
Function.prototype.MyBind = function (context) {
```

```
  const args = [...arguments].slice(1);


  return function () {
    context.MyApply(context, args);
  }}
```

3.模拟 new 的实现

```
function myNew(fn) {
  const newObj = Object.create(fn.prototype);
  result = fn.apply(newObj, [...arguments].slice(1));
  return typeof result === "object" ? result : newObj;}
```

4.请使用 Proxy + Fetch 实现类似于 axios 的基础 API

```
const fetch = require("node-fetch");
const axiosOriginal = {
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS']};const axios = new
Proxy(axiosOriginal, {
  set() {
    throw new Error("Can't set any property");
  },


  get(target, name) {
    const method = name.toLocaleUpperCase();
    if (target.methods.indexOf(method) === -1) throw new Error(`Can't support
method ${method}`);
```

```
    return (url, options) => {
      return fetch(url, {
        method,
        ...options
      }).then(res => res.text())
    }
  }});
```

```
axios.get("http://www.baidu.com").then(res =>
console.log(res));axios.post("http://www.baidu.com").then(res =>
console.log(res));
```

5.手写 Promise

```
const PENDING = "pending";const RESOLVED = "resolved";const REJECTED =
"rejected";
```

```
class MyPromise {
  constructor(fn) {
    this.state = PENDING;
    this.resolvedHandlers = [];
    this.rejectedHandlers = [];
    fn(this.resolve.bind(this), this.reject.bind(this));
    return this;
  }

  resolve(props) {
    setTimeout(() => {
      this.state = RESOLVED;
      const resolveHandler = this.resolvedHandlers.shift();
```

```
    if (!resolveHandler) return;

    const result = resolveHandler(props);
    if (result && result instanceof MyPromise) {
      result.then(...this.resolvedHandlers);
    }
  });
}

reject(error) {
  setTimeout(() => {
    this.state = REJECTED;
    const rejectHandler = this.rejectedHandlers.shift();
    if (!rejectHandler) return;

    const result = rejectHandler(error);
    if (result && result instanceof MyPromise) {
      result.catch(...this.rejectedHandlers);
    }
  });
}

then(...handlers) {
  this.resolvedHandlers = [...this.resolvedHandlers, ...handlers];
  return this;
```

```
  }


  catch(...handlers) {
    this.rejectedHandlers = [...this.rejectedHandlers, ...handlers];
    return this;
  }}
MyPromise.all = function (promises) {
  return new MyPromise((resolve, reject) => {
    const results = [];
    for (let i = 0; i < promises.length; i++) {
      const promise = promises[i];
      promise.then(res => {
        results.push(res);
        if (results.length === promises.length) resolve(results);
      }).catch(reject);
    }
  });}
MyPromise.race = function (promises) {
  return new MyPromise((resolve, reject) => {
    for (let i = 0; i < promises.length; i++) {
      const promise = promises[i];
      promise.then(resolve).catch(reject);
    }
  });}
```

7.聊聊 redux-thunk 是如何实现异步 action 的？

在 redux-thunk 中会判断 action 的类型，如果 action 的类型为函数，则执行该 action 函数，并且将 dispatch 作为参数，将自身的 dispatch 操作延迟到 action 函数中执行，由 action 函数决定何时（可能是异步操作后）执行 dispatch.

8.简单聊聊 new Vue 以后发生的事情

1).new Vue 会调用 Vue 原型链上的 _init 方法对 Vue 实例进行初始化；

2).首先是 initLifecycle 初始化生命周期，对 Vue 实例内部的一些属性（如 children、parent、isMounted）进行初始化；

3).initEvents，初始化当前实例上的一些自定义事件（Vue.$on）；

4).initRender，解析 slots 绑定在 Vue 实例上，绑定 createElement 方法在实例上；

5).完成对生命周期、自定义事件等一系列属性的初始化后，触发生命周期钩子 beforeCreate；

6).initInjections，在初始化 data 和 props 之前完成依赖注入（类似于 React.Context）；

7).initState，完成对 data 和 props 的初始化，同时对属性完成数据劫持内部，启用监听者对数据进行监听（更改）；

8).initProvide，对依赖注入进行解析；

9).完成对数据（state 状态）的初始化后，触发生命周期钩子 created；

10).进入挂载阶段，将 vue 模板语法通过 vue-loader 解析成虚拟 DOM 树，虚拟 DOM 树与数据完成双向绑定，触发生命周期钩子 beforeMount；

11).将解析好的虚拟 DOM 树通过 vue 渲染成真实 DOM，触发生命周期钩子 mounted；

9.介绍下 webpack 热更新原理，是如何做到在不刷新浏览器的前提下更新页面的

1).当修改了一个或多个文件；

2).文件系统接收更改并通知 webpack；

3).webpack 重新编译构建一个或多个模块，并通知 HMR（Hot Module Replacement）服务器进行更新；

4).HMR Server 使用 Websocket 通知 HMR runtime 需要更新，HMR runtime 通过 HTTP 请求更新 jsonp；

5).HMR runtime 替换更新中的模块，如果确定这些模块无法更新，则触发整个页面刷新；

10.简述一下 React 的源码实现

1).React 的实现主要分为 Component 和 Element；

2).Component 属于 React 实例，在创建实例的过程中会在实例中注册 state 和 props 属性，还会依次调用内置的生命周期函数；

3).Component 中有一个 render 函数，render 函数要求返回一个 Element 对象（或 null）；

4).Element 对象分为原生 Element 对象和组件式对象，原生 Element + 组件式对象会被一起解析成虚拟 DOM 树，并且内部使用的 state 和 props 也以 AST 的形式注入到这棵虚拟 DOM 树之中；

5).在渲染虚拟 DOM 树的前后，会触发 React Component 的一些生命周期钩子函数，比如 componentWillMount 和 componentDidMount，在虚拟 DOM 树解析完成后将被渲染成真实 DOM 树；

6).调用 setState 时，会调用更新函数更新 Component 的 state，并且触发内部的一个 updater，调用 render 生成新的虚拟 DOM 树，利用 diff 算法与旧的虚拟 DOM 树进行比对，比对以后利用最优的方案进行 DOM 节点的更新，这也是 React 单向数据流的原理（与 Vue 的 MVVM 不同之处）。

六. 网络相关

1.HTTP1.0 和 HTTP1.1 有什么区别？

HTTP1.0 最早在网页中使用是在 1996 年，那个时候只是使用一些较为简单的网页上和网络请求上。而 HTTP1.1 则在 1999 年才开始广泛应用于现在的各大

浏览器网络请求中，同时 HTTP1.1 也是当前使用最为广泛的 HTTP 协议。主要区别体现在：

1.缓存处理：在 HTTP1.0 中主要使用 header 里的 If-Modified-Since、Expires 来作为缓存判断的标准，HTTP1.1 则引入了更多的缓存控制策略例如 Entity tag、If-Unmodified-Since、If-Match、If-None-Match 等更多可供选择的缓存头来控制缓存策略。

2.带宽优化及网络连接的使用：HTTP1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，而且不支持断点续传功能，HTTP1.1 则在请求头中引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（Partial Content），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。

3.错误通知的管理：在 HTTP1.1 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）表示服务器的某个资源被永久性的删除；

4.Host 头处理：在 HTTP1.0 中认为每台服务器都绑定唯一的 IP 地址，因此，请求信息中的 URL 并没有传递主机名（hostname）。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），并且它们可以共享一个 IP 地址。HTTP1.1 的请求信息和响应信息都应支持 Host 头域，且请求信息中如果没有 Host 头域会报告一个错误（400 Bad Request）。

5.长连接：HTTP1.1 支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减

少了建立和关闭连接的消耗和延迟，在 HTTP1.1 中默认开启 Connection: keep-alive，一定程度上弥补了 HTTP1.0 每次请求都要创建连接的缺点。

2.（网易）简单讲解一下 http2 的多路复用

在 HTTP/1 中，每次请求都会建立一次 HTTP 连接，也就是我们常说的 3 次握手和 4 次挥手，这个过程在一次请求过程中占用了相当长的时间，即使开启了 `Keep-Alive`，解决了多次连接的问题，但是依然有两个效率上的问题，一是串行的文件传输，二是连接数过多导致的性能问题。

HTTP/2 的多路复用就是为了解决上述的两个性能问题。

在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。

多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。

3.介绍 HTTPS 握手过程

1. 客户端使用 `https` 的 `url` 访问 web 服务器，要求与服务器建立 `ssl` 连接；
2. web 服务器收到客户端请求后，会将网站的证书（包含公钥）传送一份给客户端；
3. 客户端收到网站证书后会检查证书的颁发机构以及过期时间，如果没有问题就随机产生一个秘钥；
4. 客户端利用公钥将会话秘钥加密，并传送给服务端，服务端利用自己的私钥解密出会话秘钥；
5. 之后服务器与客户端使用秘钥加密传输；

4.HTTPS 握手过程中，客户端如何验证证书的合法性

1. 首先浏览器读取证书中的证书所有者、有效期等信息进行校验，校验证书的网站域名是否与证书颁发的域名一致，校验证书是否在有效期内；
2. 浏览器开始查找操作系统中已内置的受新人的证书发布机构 CA，与服务器发来的证书中的颁发者 CA 比对，用于校验证书是否为合法机构颁发；
3. 如果找不到，浏览器就会报错，说明浏览器发来的证书是不可信任的；
4. 如果找到，那么浏览器就会从操作系统中取出颁发者 CA 的公钥（多数浏览器开发商发布版本时，会实现在内部植入常用认证机关的公开密钥），然后对服务器发来的证书里面的签名进行解密；
5. 浏览器使用相同的 hash 算法计算出服务器发来的证书的 hash 值，将这个计算的 hash 值与证书中签名做迪比；
6. 对比结果一致，则证明服务器发来的证书合法，没有被冒充

5.介绍下如何实现 token 加密

JWT 加密：

1. 需要一个 secret（随机数）；
2. 后端利用 secret 和加密算法（如：HMAC-SHA256）对 payload（如账号密码）生成一个字符串（token），返回前端；
3. 前端每次 request 在 header 中带上 token;
4. 后端用同样的算法解密；

6.介绍下 HTTPS 中间人攻击

https 协议由 http + ssl 协议构成。

中间人攻击过程如下：

1. 服务器向客户端发送公钥；
2. 攻击者截获公钥，保留在自己手上；
3. 然后攻击者自己生成一个【伪造的】公钥，发给客户端；
4. 客户端收到伪造的公钥后，生成加密 hash（秘钥） 值发给服务器；
5. 攻击者获得加密 hash 值，用自己的私钥解密获得真秘钥；
6. 同时生成假的加密 hash 值，发给服务器；
7. 服务器用私钥解密获得假秘钥；
8. 服务器用假秘钥加密传输信息；

防范方法：

1. 服务器在发送浏览器的公钥中加入 CA 证书，浏览器可以验证 CA 证书的有效性；（现有 HTTPS 很难被劫持，除非信任了劫持者的 CA 证书）。

7.说出几个你知道的 HTTP 状态码及其功能

100~199: 信息提示

200~299：成功

300~399：重定向

400~499：客户端错误

500~599：服务端错误

- 200 成功
- 204 无内容
- 301 永久移动（回应 GET 响应时会自动将请求者转到新位置）

- 304 未修改（协商缓存）
- 400 Bad Request
- 401 未授权
- 403 服务器拒绝请求
- 404 未找到
- 409 请求发生冲突
- 500 服务器内部错误
- 502 错误网关
- 503 服务不可用

8.从输入 URL 到页面加载的全过程

1. 浏览器获取用户输入，等待 url 输入完毕，触发 enter 事件；
2. 解析 URL，分析协议头，再分析主机名是域名还是 IP 地址；
3. 如果主机名是域名的话，则发送一个 DNS 查询请求到 DNS 服务器，获得主机 IP 地址；
4. 使用 DNS 获取到主机 IP 地址后，向目的地址发送一个（http/https/protocol）请求，并且在网络套接字上自动添加端口信息（http 80 https 443）；
5. 等待服务器响应结果；
6. 将响应结果（html）经浏览器引擎解析后得到 `Render tree`，浏览器将 `Render tree` 进行渲染后显示在显示器中，用户此时可以看到页面被渲染。

9.简述 HTTP2.0 与 HTTP1.1 相较于之前版本的改进

HTTP2.0

1. HTTP2.0 基本单位为二进制，以往是采用文本形式，健壮性不是很好，现在采用二进制格式，更方便更健壮。
2. HTTP2.0 的多路复用，把多个请求当做多个流，请求响应数据分成多个帧，不同流中的帧交错发送，解决了 TCP 连接数量多，TCP 连接慢，所以对于同一个域名只用创建一个连接就可以了。
3. HTTP2.0 压缩消息头，避免了重复请求头的传输，又减少了传输的大小；
4. HTTP2.0 服务端推送，浏览器发送请求后，服务端会主动发送与这个请求相关的资源，之后浏览器就不用再次发送后续的请求了；
5. HTTP2.0 可以设置请求优先级，可以按照优先级来解决阻塞的问题；

HTTP1.1

1. 缓存处理新增 E-Tag、If-None-Match 之类的缓存来来控制缓存；
2. 长连接，可以在一个 TCP 连接上发送多个请求和响应；

10.SSL 连接断开后如何恢复？

Session ID

每一次的会话都有一个编号，当对话中断后，下一次重新连接时，只要客户端给出这个编号，服务器如果有这个编号的记录，那么双方就可以继续使用以前的密钥，而不用重新生成一把。

Session Ticket

session ticket 是服务器在上一次对话中发送给客户的，这个 ticket 是加密的，只有服务器可能够解密，里面包含了本次会话的信息，比如对话密钥和加密方法等。这样不管我们的请求是否转移到其他的服务器上，当服务器将 ticket 解密以后，就能够获取上次对话的信息，就不用重新生成对话秘钥了。

11.什么是 CDN 服务？

CDN 是一个内容分发网络，通过对源网站资源的缓存，利用本身多台位于不同地域、不同运营商的服务器，向用户提供资源就近访问的功能。也就是说，用户的请求并不是直接发送给源网站，而是发送给 CDN 服务器，由 CDN 服务器将请求定位到最近的含有该资源的服务器上去请求。这样有利于提高网站的访问速度，同时通过这种方式也减轻了源服务器的访问压力。

CDN 访问过程

1. 用户输入访问的域名,操作系统向 LocalDns 查询域名的 ip 地址.
2. LocalDns 向 ROOT DNS 查询域名的授权服务器(这里假设 LocalDns 缓存过期)
3. ROOT DNS 将域名授权 dns 记录回应给 LocalDns
4. LocalDns 得到域名的授权 dns 记录后,继续向域名授权 dns 查询域名的 ip 地址
5. 域名授权 dns 查询域名记录后(一般是 CNAME)，回应给 LocalDns
6. LocalDns 得到域名记录后,向智能调度 DNS 查询域名的 ip 地址
7. 智能调度 DNS 根据一定的算法和策略(比如静态拓扑，容量等),将最适合的 CDN 节点 ip 地址回应给 LocalDns
8. LocalDns 将得到的域名 ip 地址，回应给 用户端

9. 用户得到域名 ip 地址后，访问站点服务器

10. CDN 节点服务器应答请求，将内容返回给客户端.(缓存服务器一方面在本地进行保存，以备以后使用，二方面把获取的数据返回给客户端，完成数据服务过程)

七. 设计模式

1.什么是设计模式？设计模式如何解决复杂问题？

设计模式描述了一个在我们周围不断发生的问题，以及解决该问题方案的核心。有了设计模式，我们就可以一次又一次的使用该方案而不用重复劳动。

设计模式主要通过两个方面来解决复杂问题：

1. 分解：将复杂问题分解成多个简单问题。
2. 抽象：忽略问题的本质细节，去处理泛化和理想化了的对象模型。

2.什么是白箱复用和黑箱复用？

白箱复用就是 B 类继承 A 类的功能，同时需要了解 A 类的内部细节，从而达到复用的效果，耦合性较强。

在黑箱复用中，B 类只需要关注 A 类所暴露的一些外部方法即可达到复用的效果，达到了解耦的效果。

3.介绍下观察者模式和订阅-发布模式的区别，各自适用于什么场景

观察者模式中主体和观察者是互相感知的，发布-订阅模式是借助第三方来实现调度的，发布者和订阅者之间互不感知。

一对多时使用观察者模式，多对多时使用订阅-发布模式。

4.简述面向对象的设计原则

1. 依赖倒置原则
    1. 高层模块（稳定）不应该依赖低层模块（变化），两者都应该依赖于抽象（稳定）；
    2. 抽象（稳定）不应该依赖于实现细节（变化），实现细节（变化）应该依赖于抽象（稳定）；
2. 开放封闭原则
    1. 对扩展开放，对更改封闭；
    2. 类模块应该是可扩展的，但是不可修改；
3. 单一职责原则
    1. 一个类应该只有一个引起它变化的原因；
    2. 变化的方向隐含了类的责任；
4. Liskov 替换原则
    1. 子类必须能够替换他们的基类（IS-A）；
    2. 继承表达类型抽象；
5. 接口隔离原则
    1. 不应该强迫客户端使用他们不用的方法；
    2. 接口应该小而完备；
6. 优先使用对象组合，而不是类继承
    1. 类继承通常为“白箱复用”，对象组合通常为“黑箱复用”；
    2. 继承在某种程度上破坏了封装性，子类父类耦合度高；
    3. 而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低；
7. 封装变化点

1. 使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实现层次间的松耦合；

8. 针对接口编程，而不是针对实现编程

    1. 不将变量类型声明为具体的某个类，而是声明为某个接口；
    2. 客户程序无需获取对象的具体类型，只需要知道对象所具有的接口；
    3. 减少系统中的各部分依赖关系，从而实现"高内聚、低耦合"的类型设计方案；

5.简述你了解的设计模式及应用场景

组件协作

1. Template Method 模式：
    1. 模式定义：定义一个操作中的算法的骨架（稳定），将一些步骤（变化）延迟到子类中；
    2. 应用场景：React 生命周期；
2. Observer 模式：
    1. 模式定义：定义对象间的一种一对多（变化）的依赖关系，以便当一个对象（Subject）的状态发生变化时，所有依赖于它的对象都得到通知并自动更新。
    2. 应用场景：Redux 实现。
3. Strategy 模式：
    1. 定义：定义一系列算法，把它们一个个封装起来，并且使它们可互相替换（变化）。该模式使得算法可独立于使用它的客户程序（稳定）而变化（扩展、子类化）。
    2. 应用：国际化、多种输入一种输出。

单一职责

1. Decorator 模式：
    1. 模式定义：动态（组合）地给一个对象增加一些额外的职责，就增加功能而言，Decorator 模式比生成子类（继承）更加灵活（消除重复代码、减少子类个数）；
    2. 应用场景：React 高阶组件、中间件。
2. Bridge 模式：
    1. 模式定义：将抽象业务（业务功能）与实现部分（平台实现）分离，使它们都可以独立地变化。
    2. 应用场景：组件组合功能。

对象创建

1. Factory Method 模式：
    1. 模式定义：定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使得一个类的实例化延迟（目的：解耦，手段：抽象）到子类。
    2. 应用场景：React.createElement(...)、gin.Default(...)
2. Abstract Factory
    1. 模式定义：提供一个接口，让该接口负责创建一系列"相关或者相互依赖的对象"，无需指定他们具体的类。

2. 应用场景：SQL API 类有多个成员类，比如连接类、创建类、操作类，将这些组合起来放在一个 Factory 类中，由这个类完成多个成员类的创建工作。

对象性能

1. Singleton 模式：
    1. 模式定义：保证一个类只有一个实例，并提供一个该实例的全局访问点。
    2. 应用场景：游戏中的主角类实例。
2. FlyWeight（享元模式）：
    1. 模式定义：运行共享技术有效地支持大量细粒度的对象。
    2. 应用场景：对象池优化。

接口隔离

1. Facade 模式：
    1. 模式定义：为子系统中的一组接口提供一致（稳定）的界面，Facade 模式定义了一个高层接口，这个接口使得这个子系统更加容易使用（复用）。
    2. 应用场景：各种硬件驱动。
2. Proxy 模式：
    1. 模式定义：为其他对象提供一种代理以控制（隔离、使用接口）对这种对象的访问。
    2. 应用场景：使用闭包导出方法（代理操作对象）。
3. Adapter 模式：

1. 模式定义：将一个类的接口转换为客户希望的另一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
2. 应用场景：Go 中的 `http.ListenAndServe` 所接受的结构体只需要实现 `ServeHTTP` 方法即可满足 `http` 的接口条件。

4. Composite 模式：

1. 模式定义：将对象组合成树形结构以表示"部分-整体"的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性（稳定）。
2. 应用场景：在树形结构中，`Parent Node` 和 `Child Node` 都继承于同一抽象类 `Node`，都使用 `process` 方法来执行自身的方法。

八. 算法相关

1.选择排序

选择排序：首先找到数组中最小的元素，其次，将它和数组的第一个元素交换位置（如果第一个元素就是最小元素那么它就和自己交换）。其次，在剩下的元素中找到最小的元素，将它与数组的第二个元素交换位置。如此往复，直到将整个数组排序。这种方法叫做选择排序，因为它在不断地选择剩余元素之中的最小者。

2.使用迭代的方式实现 flatten 函数

```
function flatten(arr) {
  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(arr[i])) {
      arr = arr.concat(arr[i])
      arr.splice(i, 1)
```

```
    }
  }
  return arr;}
let arr = [1, 2, [3, 4, 5, [6, 7], 8], 9, 10, [11, [12, 13]]]
console.log(flatten(arr))
```

3.介绍下深度优先遍历和广度优先遍历，如何实现？

图的遍历

两种遍历算法：

- 深度优先遍历；
- 广度优先遍历；

深度优先遍历（DFS）

深度优先遍历（Depth-First-Search），是搜索算法的一种，它沿着树的深度遍历树的节点，尽可能深地搜索树的分支。当节点 v 的所有边都已被探寻过，将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已探寻源节点到其他所有节点为止，如果还有未被发现的节点，则选择其中一个未被发现的节点为源节点并重复以上操作，直到所有节点都被探寻完成。

简单的说，DFS 就是从图中的一个节点开始追溯，直到最后一个节点，然后回溯，继续追溯下一条路径，直到到达所有的节点，如此往复，直到没有路径为止。

DFS 可以产生相应图的拓扑排序表，利用拓扑排序表可以解决很多问题，例如最大路径问题。一般用堆数据结构来辅助实现 DFS 算法。

注意：深度 DFS 属于盲目搜索，无法保证搜索到的路径为最短路径，也不是在搜索特定的路径，而是通过搜索来查看图中有哪些路径可以选择。

广度优先遍历（BFS）

广度优先遍历（Breadth-First-Search）是从根节点开始，沿着图的宽度遍历节点，如果所有的节点均被访问过，则算法终止，BFS 同样属于盲目搜索，一般用队列数据结构来辅助实现 BFS。

BFS 从一个节点开始，尝试访问尽可能接近它的目标节点。本质上这种遍历在图上是逐层移动的，首先检查最靠近第一个节点的层，再逐渐向下移动到离起始节点最远的层。

4.（携程）算法手写题

已知如下数组：

```
var arr = [ [1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14] ] ] ], 10];
```

编写一个程序将数组扁平化去并除其中重复部分数据，最终得到一个升序且不重复的数组

```
// 思路型 function flatten(arr) {
  let flattenArr = [];
  for (let i = 0; i < arr.length; i++) {
    const item = arr[i];
    if (Array.isArray(item)) {
      flattenArr = flattenArr.concat(flatten(item))
    } else {
      if (flattenArr.indexOf(item) > -1) continue;
      flattenArr.push(item);
```

```
    }
  }
  return flattenArr.sort((a, b) => a - b);}
// API 型 function flatten2(arr) {
  return Array.from(new Set(arr.flat(Infinity))).sort((a, b) => a - b);}
```

5.给定两个数组，写一个方法来计算它们的交集。

例如：给定 nums1 = [1, 2, 2, 1]，nums2 = [2, 2]，返回 [2, 2]。

答案：同目录下 test.js

6.数组编程题

随机生成一个长度为 10 的整数类型的数组，例如 [2, 10, 3, 4, 5, 11, 10, 11, 20]，将其排列成一个新数组，要求新数组形式如下，例如 [[2, 3, 4, 5], [10, 11], [20]]。

```
function generateRandomArr(len) {
  const arr = new Array(len);
  arr.fill();
  return arr.map(() => Math.round(Math.random() * 100));}
function reFlatten() {
  const sortArr = [];
  let arr = generateRandomArr(10);
  arr.sort((a, b) => a - b);
  arr = [...new Set(arr)];
  for (let i = 0; i < arr.length; i++) {
    const n = Math.floor(arr[i] / 10);
```

```
    sortArr[n] = sortArr[n] || [];
    sortArr[n].push(arr[i]);
  }
  return sortArr.filter(item => item);}
console.log(reFlatten());
```

7.如何把一个字符串的大小写取反（大写变小写小写变大写），例如 'AbC' 变成 'aBc' 。

```
function
convert(str)
{
              let convertStr = "";
              for (let i = 0; i < str.length; i++) {
              const code = str.charCodeAt(i);
              if (code >= 97) {
              convertStr += String.fromCharCode(code - 32);
              } else {
              convertStr += String.fromCharCode(code + 32);
              }
              }
              return convertStr;
              }

              console.log(convert("AbC"));
```

8.实现一个字符串匹配算法，从长度为 n 的字符串 S 中，查找是否存在字符串 T，T 的长度是 m，若存在返回所在位置。

```
function findStr(str, targetStr) {
  if (str.length < targetStr.length) return -1;
  let cStr = str;
  let tLen = targetStr.length;
  let i = 0;
```

```
  while(cStr.length > 0) {
    if (cStr.slice(0, tLen) === targetStr) {
      return i;
    }
    cStr = cStr.slice(1);
    i++;
  }
  return -1;}
```

9.算法题「旋转数组」

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

```
function
moveRight(arr,
n) {
                const movedArr = [];
                const l = arr.length;
                for (let i = 0; i < l; i++) {
                movedArr[i] = arr[Math.abs(-n % l)];
                }
                return movedArr;
                }

                console.log(moveRight([1, 2, 3, 4, 5, 6, 7], 1));
```

10.（京东、快手）周一算法题之「两数之和」

给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。 你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。 示例：

给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9

所以返回 [0, 1]

```
function computed(arr, target) {
  const map = new Map();
  for (let i = 0; i < arr.length; i++) {
    map.set(arr[i], i);
  }
  for (let i = 0; i < arr.length; i++) {
    const value = map.get(target - arr[i]);
    if (value && value !== i) return [i, value];
  }}
```

11.（bilibili）编程算法题

用 JavaScript 写一个函数，输入 int 型，返回整数逆序后的字符串。如：输入整型 1234，返回字符串“4321”。要求必须使用递归函数调用，不能用全局变量，输入函数必须只有一个参数传入，必须返回字符串。

```
function reverse(n) {
  let y = n % 10;
  let s = String(y);
  if (n / 10 >= 1) {
    s += reverse((n - y) / 10);
  }
  return s;}
```

12.如何实现数组的随机排序？

```
// 随机数排序 function random1(arr) {
  return arr.sort(() => Math.random() - .5);}
```

```
// 随机插入排序 function random2(arr) {
  const cArr = [...arr];
  const newArr = [];
  while (cArr.length) {
    const index = Math.floor(Math.random() * cArr.length);
    newArr.push(cArr[index]);
    cArr.splice(index, 1);
  }
  return newArr;}
// 洗牌算法，随机交换排序 function random3(arr) {
  const l = arr.length;
  for (let i = 0; i < l; i++) {
    const index = Math.floor(Math.random() * (l - i)) + i;
    const temp = arr[index];
    arr[index] = arr[i];
    arr[i] = temp;
  }
  return arr;}
```

13.将数字变成 0 的操作次数

给你一个非负整数 num ，请你返回将它变成 0 所需要的步数。 如果当前数字是偶数，你需要把它除以 2 ；否则，减去 1 。

示例 1：

输入：num = 14
输出：6
解释：

步骤 1) 14 是偶数，除以 2 得到 7 。
步骤 2） 7 是奇数，减 1 得到 6 。
步骤 3） 6 是偶数，除以 2 得到 3 。
步骤 4） 3 是奇数，减 1 得到 2 。
步骤 5） 2 是偶数，除以 2 得到 1 。
步骤 6） 1 是奇数，减 1 得到 0 。

示例 2：

输入：num = 8
输出：4
解释：
步骤 1） 8 是偶数，除以 2 得到 4 。
步骤 2） 4 是偶数，除以 2 得到 2 。
步骤 3） 2 是偶数，除以 2 得到 1 。
步骤 4） 1 是奇数，减 1 得到 0 。

示例 3：

输入：num = 123
输出：12

14.实现 Trie (前缀树)

实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

示例:

```
Trie trie = new Trie();


trie.insert("apple");

trie.search("apple");   // 返回 true

trie.search("app");     // 返回 false

trie.startsWith("app"); // 返回 true

trie.insert("app");

trie.search("app");     // 返回 true
```

说明:

- 你可以假设所有的输入都是由小写字母 a-z 构成的。
- 保证所有输入均为非空字符串。

15.朋友圈

班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。

如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。

给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。如果 M[i][j] = 1，

表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

示例 1:

输入:

```
[[1,1,0],
 [1,1,0],
 [0,0,1]]
```

输出: 2

说明：已知学生 0 和学生 1 互为朋友，他们在一个朋友圈。

第 2 个学生自己在一个朋友圈。所以返回 2。

示例 2:

输入：

```
[[1,1,0],
 [1,1,1],
 [0,1,1]]
```

输出：1

说明：已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，所以学生 0 和学生 2 也是朋友，所以他们三个在一个朋友圈，返回 1。

注意：

- N 在[1,200]的范围内。
- 对于所有学生，有 M[i][i] = 1。
- 如果有 M[i][j] = 1，则有 M[j][i] = 1。

16.解压缩编码列表

给你一个以行程长度编码压缩的整数列表 nums 。

考虑每对相邻的两个元素 freq, val] = [nums[2i], nums[2i+1]] （其中 i >= 0 ），

每一对都表示解压后子列表中有 freq 个值为 val 的元素，你需要从左到右连接所有子列表以生成解压后的列表。

请你返回解压后的列表。

示例：

输入：nums = [1,2,3,4]

输出：[2,4,4,4]

解释：第一对 [1,2] 代表着 2 的出现频次为 1，所以生成数组 [2]。

第二对 [3,4] 代表着 4 的出现频次为 3，所以生成数组 [4,4,4]。

最后将它们串联到一起 [2] + [4,4,4] = [2,4,4,4]。

示例 2：

输入：nums = [1,1,2,3]

输出：[1,3,3]

提示：

- 2 <= nums.length <= 100
- nums.length % 2 == 0
- 1 <= nums[i] <= 100

17.整数的各位积和之差

给你一个整数 n，请你帮忙计算并返回该整数「各位数字之积」与「各位数字之和」的差。

示例 1：

输入：n = 234
输出：15
解释：
各位数之积 = 2 * 3 * 4 = 24
各位数之和 = 2 + 3 + 4 = 9
结果 = 24 - 9 = 15

示例 2：

输入：n = 4421
输出：21
解释：
各位数之积 = 4 * 4 * 2 * 1 = 32

```
各位数之和 = 4 + 4 + 2 + 1 = 11
结果 = 32 - 11 = 21
```

提示：

```
- 1 <= n <= 10^5
```

18.猜数字

小 A 和 小 B 在玩猜数字。小 B 每次从 1, 2, 3 中随机选择一个，小 A 每次也从 1, 2, 3 中选择一个猜。他们一共进行三次这个游戏，请返回 小 A 猜对了几次？

输入的 guess 数组为 小 A 每次的猜测，answer 数组为 小 B 每次的选择。guess 和 answer 的长度都等于 3。

示例 1：

```
输入：guess = [1,2,3], answer = [1,2,3]
输出：3
解释：小 A 每次都猜对了。
```

示例 2：

```
输入：guess = [2,2,3], answer = [3,2,1]
输出：1
解释：小 A 只猜对了第二次。
```

限制：

- guess 的长度 = 3
- answer 的长度 = 3
- guess 的元素取值为 {1, 2, 3} 之一。
- answer 的元素取值为 {1, 2, 3} 之一。

19.统计位数为偶数的数字

给你一个整数数组 nums，请你返回其中位数为 偶数 的数字的个数。

示例 1：

```
输入：nums = [12,345,2,6,7896]
输出：2
解释：
12 是 2 位数字（位数为偶数）
345 是 3 位数字（位数为奇数）
2 是 1 位数字（位数为奇数）
6 是 1 位数字 位数为奇数）
7896 是 4 位数字（位数为偶数）
因此只有 12 和 7896 是位数为偶数的数字
```

示例 2：

```
输入：nums = [555,901,482,1771]
输出：1
解释：
只有 1771 是位数为偶数的数字。
```

提示：

```
- 1 <= nums.length <= 500
- 1 <= nums[i] <= 10^5
```

20.交换数字

编写一个函数，不用临时变量，直接交换 numbers = [a, b]中 a 与 b 的值。

示例：

```
输入：numbers = [1,2]
输出：[2,1]
```

提示：

- numbers.length == 2

21.删除链表中的节点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:

示例 1：

```
输入：head = [4,5,1,9], node = 5
输出：[4,1,9]
```

解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.

示例 2：

```
输入：head = [4,5,1,9], node = 1
输出：[4,5,9]
```

解释：给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

22.子集[sku]

给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

```
输入: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

23.螺旋矩阵 II

给定一个正整数 n，生成一个包含 1 到 n^2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例:

```
输入: 3
输出:
```

```
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

24.IP 地址无效化

给你一个有效的 IPv4 地址 address，返回这个 IP 地址的无效化版本。

所谓无效化 IP 地址，其实就是用 "[.]" 代替了每个 "."。

示例 1：

输入：address = "1.1.1.1"

输出："1[.]1[.]1[.]1"

示例 2：

输入：address = "255.100.50.0"

输出："255[.]100[.]50[.]0"

提示：

- 给出的 address 是一个有效的 IPv4 地址

25.二进制链表转整数

给你一个单链表的引用结点 head。链表中每个结点的值不是 0 就是 1。已知此链表是一个整数数字的二进制表示形式。

请你返回该链表所表示数字的 十进制值 。

示例 1：

输入：head = [1,0,1]

输出：5

解释：二进制数 (101) 转化为十进制数 (5)

示例 2：

输入：head = [0]

输出：0

示例 3：

输入：head = [1]

输出：1

示例 4：

输入：head = [1,0,0,1,0,0,1,1,1,0,0,0,0,0,0]

输出：18880

示例 5：

输入：head = [0,0]

输出：0

提示：

- 链表不为空。
- 链表的结点总数不超过 30。
- 每个结点的值不是 0 就是 1。

26.反转链表

反转一个单链表。

示例:

输入：1->2->3->4->5->NULL

输出：5->4->3->2->1->NULL

进阶:

- 你可以迭代或递归地反转链表。你能否用两种方法解决这道题？

27.删去字符串中的元音

给你一个字符串 S，请你删去其中的所有元音字母（'a'，'e'，'i'，'o'，'u'），并返回这个新字符串。

示例 1：

输入："leetcodeisacommunityforcoders"

输出："ltcdscmmntyfrcdrs"

示例 2：

输入："aeiou"

输出：""

提示：

- S 仅由小写英文字母组成。
- 1 <= S.length <= 1000

28.找出变位映射

给定两个列表 Aand B，并且 B 是 A 的变位（即 B 是由 A 中的元素随机排列后组成的新列表）。

我们希望找出一个从 A 到 B 的索引映射 P 。一个映射 P[i] = j 指的是列表 A 中的第 i 个元素出现于列表 B 中的第 j 个元素上。

列表 A 和 B 可能出现重复元素。如果有多于一种答案，输出任意一种。

例如，给定

```
A = [12, 28, 46, 32, 50]

B = [50, 12, 32, 46, 28]
```

需要返回

```
[1, 4, 3, 2, 0]
```

P[0] = 1 ，因为 A 中的第 0 个元素出现于 B[1]，而且 P[1] = 4 因为 A 中第 1 个元素出现于 B[4]，以此类推。

注：

- A, B 有相同的长度，范围为 [1, 100]。
- A[i], B[i] 都是范围在 [0, 10^5] 的整数。

29.TinyURL 的加密与解密

TinyURL 是一种 URL 简化服务， 比如：当你输入一个 URL https://leetcode.com/problems/design-tinyurl 时，它将返回一个简化的 URL http://tinyurl.com/4e9iAk.

要求：设计一个 TinyURL 的加密 encode 和解密 decode 的方法。你的加密和解密算法如何设计和运作是没有限制的，你只需要保证一个 URL 可以被加密成一个 TinyURL，并且这个 TinyURL 可以用解密方法恢复成原本的 URL。

30.访问所有点的最小时间

平面上有 n 个点，点的位置用整数坐标表示 points[i] = [xi, yi]。请你计算访问所有这些点需要的最小时间（以秒为单位）。

你可以按照下面的规则在平面上移动：

每一秒沿水平或者竖直方向移动一个单位长度，或者跨过对角线（可以看作在一秒内向水平和竖直方向各移动一个单位长度）。 必须按照数组中出现的顺序来访问这些点。

示例 1：

输入：points = [[1,1],[3,4],[-1,0]]

输出：7

解释：一条最佳的访问路径是： [1,1] -> [2,2] -> [3,3] -> [3,4] -> [2,3] -> [1,2] -> [0,1] -> [-1,0]

从 [1,1] 到 [3,4] 需要 3 秒

从 [3,4] 到 [-1,0] 需要 4 秒

一共需要 7 秒

示例 2：

输入：points = [[3,2],[-2,2]]

输出：5

提示：

- points.length == n
- 1 <= n <= 100
- points[i].length == 2
- -1000 <= points[i][0], points[i][1] <= 1000

31.无重复字符串的排列组合

无重复字符串的排列组合。编写一种方法，计算某字符串的所有排列组合，字符串每个字符均不相同。

示例 1:

输入：S = "qwe"

输出：["qwe", "qew", "wqe", "weq", "ewq", "eqw"]

示例 2：

输入：S = "ab"

输出：["ab", "ba"]

提示:

- 字符都是英文字母。
- 字符串长度在[1, 9]之间。

32.统计有序矩阵中的负数

给你一个 m * n 的矩阵 grid，矩阵中的元素无论是按行还是按列，都以非递增顺序排列。

请你统计并返回 grid 中 负数 的数目。

示例 1：

输入：grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[-1,-1,-2,-3]]

输出：8

解释：矩阵中共有 8 个负数。

示例 2：

输入：grid = [[3,2],[1,0]]

输出：0

示例 3：

输入：grid = [[1,-1],[-1,-1]]

输出：3

33.链表中倒数第 k 个节点

输入一个链表，输出该链表中倒数第 k 个节点。为了符合大多数人的习惯，本题从 1 开始计数，即链表的尾节点是倒数第 1 个节点。例如，一个链表有 6 个节点，从头节点开始，它们的值依次是 1、2、3、4、5、6。这个链表的倒数第 3 个节点是值为 4 的节点。

示例：

给定一个链表: 1->2->3->4->5, 和 k = 2.

返回链表 4->5.

示例 4：

输入：grid = [[-1]]

输出：1

提示：

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 100
- -100 <= grid[i][j] <= 100

34.单行键盘

我们定制了一款特殊的力扣键盘，所有的键都排列在一行上。

我们可以按从左到右的顺序，用一个长度为 26 的字符串 keyboard（索引从 0 开始，到 25 结束）来表示该键盘的键位布局。

现在需要测试这个键盘是否能够有效工作，那么我们就需要个机械手来测试这个键盘。

最初的时候，机械手位于左边起第一个键（也就是索引为 0 的键）的上方。当机械手移动到某一字符所在的键位时，就会在终端上输出该字符。

机械手从索引 i 移动到索引 j 所需要的时间是 |i - j|。

当前测试需要你使用机械手输出指定的单词 word，请你编写一个函数来计算机械手输出该单词所需的时间。

示例 1：

输入：keyboard = "abcdefghijklmnopqrstuvwxyz", word = "cba"

输出：4

解释：

机械手从 0 号键移动到 2 号键来输出 'c'，又移动到 1 号键来输出 'b'，接着移动到 0 号键来输出 'a'。

总用时 = 2 + 1 + 1 = 4.

示例 2：

输入：keyboard = "pqrstuvwxyzabcdefghijklmno", word = "leetcode"

输出：73

提示：

- keyboard.length == 26
- keyboard 按某种特定顺序排列，并包含每个小写英文字母一次。
- 1 <= word.length <= 10^4
- word[i] 是一个小写英文字母

35.二叉树的深度

输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

例如：

给定二叉树 [3,9,20,null,null,15,7]，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

提示：

- 节点总数 <= 10000

36.打印从 1 到最大的 n 位数

输入数字 n，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

示例 1:

输入：n = 1

输出：[1,2,3,4,5,6,7,8,9]

说明：

- 用返回一个整数列表来代替打印
- n 为正整数

37.数组中数字出现的次数 II

在一个数组 nums 中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

示例 1：

输入：nums = [3,4,3,3]

输出：4

示例 2：

输入：nums = [9,1,7,9,7,9,7]

输出：1

限制：

- 1 <= nums.length <= 10000

- 1 <= nums[i] < 2^31

38.单词频率

设计一个方法，找出任意指定单词在一本书中的出现频率。

你的实现应该支持如下操作：

WordsFrequency(book)构造函数，参数为字符串数组构成的一本书 get(word)查询指定单词在数中出现的频率 示例：

```
WordsFrequency wordsFrequency = new WordsFrequency({"i", "have", "an", "apple", "he", "have", "a", "pen"});

wordsFrequency.get("you"); //返回 0，"you"没有出现过

wordsFrequency.get("have"); //返回 2，"have"出现 2 次

wordsFrequency.get("an"); //返回 1

wordsFrequency.get("apple"); //返回 1

wordsFrequency.get("pen"); //返回 1
```

提示：

- book[i]中只包含小写字母
- 1 <= book.length <= 100000
- 1 <= book[i].length <= 10
- get 函数的调用次数不会超过 100000

39.替换空格

请实现一个函数，把字符串 s 中的每个空格替换成"%20"。

示例 1：

输入：s = "We are happy."

输出："We%20are%20happy."

限制：

0 <= s 的长度 <= 10000

40.分割平衡字符串

在一个「平衡字符串」中，'L' 和 'R' 字符的数量是相同的。

给出一个平衡字符串 s，请你将它分割成尽可能多的平衡字符串。

返回可以通过分割得到的平衡字符串的最大数量。

示例 1：

输入：s = "RLRRLLRLRL"

输出：4

解释：s 可以分割为 "RL"，"RRLL"，"RL"，"RL"，每个子字符串中都包含相同数量的 'L' 和 'R'。

示例 2：

输入：s = "RLLLLRRRLR"

输出：3

解释：s 可以分割为 "RL"，"LLLRRR"，"LR"，每个子字符串中都包含相同数量的 'L' 和 'R'。

示例 3：

输入：s = "LLLLRRRR"

输出：1

解释：s 只能保持原样 "LLLLRRRR".

提示：

- 1 <= s.length <= 1000
- s[i] = 'L' 或 'R'

41.删除中间节点

实现一种算法，删除单向链表中间的某个节点（除了第一个和最后一个节点，不一定是中间节点），假定你只能访问该节点。

示例：

输入：单向链表 a->b->c->d->e->f 中的节点 c
结果：不返回任何数据，但该链表变为 a->b->d->e->f

42.从尾到头打印链表

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

输入：head = [1,3,2]
输出：[2,3,1]

43.反转链表

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例:

输入: 1->2->3->4->5->NULL

输出: 5->4->3->2->1->NULL

限制：

0 <= 节点个数 <= 5000

44.6 和 9 组成的最大数字

给你一个仅由数字 6 和 9 组成的正整数 num。

你最多只能翻转一位数字，将 6 变成 9，或者把 9 变成 6 。

请返回你可以得到的最大数字。

示例 1：

输入：num = 9669

输出：9969

解释：

改变第一位数字可以得到 6669 。

改变第二位数字可以得到 9969 。

改变第三位数字可以得到 9699 。

改变第四位数字可以得到 9666 。

其中最大的数字是 9969 。

示例 2：

输入：num = 9996

输出：9999

解释：将最后一位从 6 变到 9，其结果 9999 是最大的数。

示例 3：

输入：num = 9999

输出：9999

解释：无需改变就已经是最大的数字了。

提示：

- 1 <= num <= 10^4
- num 每一位上的数字都是 6 或者 9 。

45.最小元素各数位之和

给你一个正整数的数组 A。

然后计算 S，使其等于数组 A 当中最小的那个元素各个数位上数字之和。

最后，假如 S 所得计算结果是 奇数 的请你返回 0，否则请返回 1。

示例 1：

输入：[34,23,1,24,75,33,54,8]

输出：0

解释：

最小元素为 1，该元素各个数位上的数字之和 S = 1，是奇数所以答案为 0。

示例 2：

输入：[99,77,33,66,55]

输出：1

解释：

最小元素为 33，该元素各个数位上的数字之和 S = 3 + 3 = 6，是偶数所以答案为 1。

提示：

- 1 <= A.length <= 100
- 1 <= A[i].length <= 100

46.合并二叉树

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

示例 1:

输入:

```
        Tree 1                     Tree 2
          1                         2
         / \                       / \
        3   2                     1   3
       /                           \   \
      5                             4   7
```

输出:

合并后的树:

```
            3
           / \
          4   5
         / \   \
        5   4   7
```

注意: 合并必须从两个树的根节点开始。

47.汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 x 和 y，计算它们之间的汉明距离。

注意： 0 ≤ x, y < 231.

示例:

输入: x = 1, y = 4

输出: 2

解释:

```
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑
```

上面的箭头指出了对应二进制位不同的位置。

48.唯一摩尔斯密码词

国际摩尔斯密码定义一种标准编码方式，将每个字母对应于一个由一系列点和短线组成的字符串， 比如: "a" 对应 ".-", "b" 对应 "-...", "c" 对应 "-.-.", 等等。

为了方便，所有 26 个英文字母对应摩尔斯密码表如下：

```
[".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-",".-..",
"--","-.","---",".--.","--.-",".-.","...","-","..-","...-",".--","-..-","-.
--","--.."]
```

给定一个单词列表，每个单词可以写成每个字母对应摩尔斯密码的组合。例如，"cab" 可以写成 "-.-.--..."，(即 "-.-." + "-..." + ".-"字符串的结合)。我们将这样一个连接过程称作单词翻译。

返回我们可以获得所有词不同单词翻译的数量。

例如:

输入: words = ["gin", "zen", "gig", "msg"]

输出: 2

解释:

各单词翻译如下:

"gin" -> "--...-."

"zen" -> "--...-."

"gig" -> "--...--."

"msg" -> "--...--."

共有 2 种不同翻译, "--...-." 和 "--...--.".

注意:

- 单词列表 words 的长度不会超过 100。
- 每个单词 words[i]的长度范围为 [1, 12]。
- 每个单词 words[i]只包含小写字母。

49.自除数

自除数 是指可以被它包含的每一位数除尽的数。

例如，128 是一个自除数，因为 128 % 1 == 0，128 % 2 == 0，128 % 8 == 0。

还有，自除数不允许包含 0 。

给定上边界和下边界数字，输出一个列表，列表的元素是边界（含边界）内所有的自除数。

示例 1：

输入：

上边界 left = 1, 下边界 right = 22

输出： [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]

注意：

- 每个输入参数的边界满足 1 <= left <= right <= 10000。

50.二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例： 给定二叉树 [3,9,20,null,null,15,7]，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

51.斐波那契数

斐波那契数，通常用 F(n) 表示，形成的序列称为斐波那契数列。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0, F(1) = 1 F(N) = F(N - 1) + F(N - 2), 其中 N > 1. 给定 N，计算 F(N)。

示例 1：

输入：2

输出：1

解释：F(2) = F(1) + F(0) = 1 + 0 = 1.

示例 2：

输入：3

输出：2

解释：F(3) = F(2) + F(1) = 1 + 1 = 2.

示例 3：

输入：4

输出：3

解释：F(4) = F(3) + F(2) = 2 + 1 = 3.

提示：

0 ≤ N ≤ 30

八. SQL 算法题

1.查找重复的电子邮箱

编写一个 SQL 查询，查找 Person 表中所有重复的电子邮箱。

示例：

```
+----+---------+
| Id | Email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

根据以上输入，你的查询应返回以下结果：

```
+---------+
| Email   |
+---------+
| a@b.com |
+---------+
```

说明：所有电子邮箱都是小写字母

2.大的国家

这里有张 World 表

```
+-----------------+------------+------------+--------------+---------------
+
| name            | continent  | area       | population   | gdp           |
+-----------------+------------+------------+--------------+---------------
+
| Afghanistan     | Asia       | 652230     | 25500100     | 20343000      |
| Albania         | Europe     | 28748      | 2831741      | 12960000      |
```

```
| Algeria         | Africa     | 2381741    | 37100000     | 188681000     |
| Andorra         | Europe     | 468        | 78115        | 3712000       |
| Angola          | Africa     | 1246700    | 20609294     | 100990000     |
+-----------------+------------+------------+--------------+---------------
+
```

如果一个国家的面积超过 300 万平方公里，或者人口超过 2500 万，那么这个国家就是大国家。

编写一个 SQL 查询，输出表中所有大国家的名称、人口和面积。

例如，根据上表，我们应该输出：

```
+--------------+-------------+--------------+
| name         | population  | area         |
+--------------+-------------+--------------+
| Afghanistan  | 25500100    | 652230       |
| Algeria      | 37100000    | 2381741      |
+--------------+-------------+--------------+
```

九. Nodejs 篇

1.介绍一下 Node 里的模块是什么？

Node 中，每个文件模块都是一个对象，它的定义如下：

```
function Module(id, parent) {
  this.id = id;
  this.exports = {};
  this.parent = parent;
  this.filename = null;
  this.loaded = false;
  this.children = [];}
module.exports = Module;
var module = new Module(filename, parent);
```

2.请介绍一下 require 的模块加载机制

1. 计算模块绝对路径；
2. 如果缓存中有该模块，则从缓存中取出该模块；
3. 按优先级依次寻找并编译执行模块，将模块推入缓存（require.cache）中；
4. 输出模块的 exports 属性；

3.请介绍一下 Node 中的内存泄露问题和解决方案

内存泄露原因

1. 全局变量：全局变量挂在 root 对象上，不会被清除掉；
2. 闭包：如果闭包未释放，就会导致内存泄露；
3. 事件监听：对同一个事件重复监听，忘记移除（removeListener），将造成内存泄露。

解决方案

最容易出现也是最难排查的就是事件监听造成的内存泄露，所以事件监听这块需要格外注意小心使用。

如果出现了内存泄露问题，需要检测内存使用情况，对内存泄露的位置进行定位，然后对对应的内存泄露代码进行修复。

4.在 Node 中两个模块互相引用会发生什么？

假设 A 和 B 模块互相引用，此时运行 A 模块的话，先运行的 A 模块将会被缓存，但是此时缓存的是一个未执行完毕的 A 模块，而 A 模块中引入的 B 模

块将会被完整加载并且正常使用，而 B 模块中调用的 A 模块将会是个默认的空对象（module.exports 的默认值），不具备 A 模块的任何功能。

5.Node 如何实现热更新？

Node 中有一个 api 是 require.cache，如果这个对象中的引用被清楚后，下次再调用就会重新加载，这个机制可以用来热加载更新的模块。

```
function clearCache(modulePath) {
  const path = require.resolve(modulePath);
  if (require.cache[path]) {
    require.cache[path] = null;
  }}
```

然后使用 fs.watchFile 监听文件的更改，文件更改后调用 clearCache 传入对应的模块名即可。

使用 pm2 reload 也可以实现暴力热更新，它会保证在新的实例重启成功后才会把旧的进程杀死，可以保持服务一直能够响应，服务能够一直保证在可响应状态。

6.为什么 Node.js 不给每一个.js 文件以独立的上下文来避免作用域被污染?

Nodejs 模块正常情况下对作用域不会造成污染（模块函数内执行），意外创建全局变量是一种例外，可以采用严格模式来避免。

```
function fn() {
  a = 1;}
fn();
var b = 10;
console.log(a); // 1console.log(this.a); // undefinedconsole.log(global.a); // 1 意外的全局上下文污染
```

```
console.log(b); // 10console.log(this.b); // undefinedconsole.log(global.b);
// undefined
```

7.Node 更适合处理 I/O 密集型任务还是 CPU 密集型任务？为什么？

Node 更适合处理 I/O 密集型的任务。因为 Node 的 I/O 密集型任务可以异步调用，利用事件循环的处理能力，资源占用极少，并且事件循环能力避开了多线程的调用，在调用方面是单线程，内部处理其实是多线程的。

并且由于 Javascript 是单线程的原因，Node 不适合处理 CPU 密集型的任务，CPU 密集型的任务会导致 CPU 时间片不能释放，使得后续 I/O 无法发起，从而造成阻塞。但是可以利用到多进程的特点完成对一些 CPU 密集型任务的处理，不过由于 Javascript 并不支持多线程，所以在这方面的处理能力会弱于其他多线程语言（例如 Java、Go）。

8.聊一聊 Node 的垃圾回收机制

Node 的 Javascript 脚本引擎是 Chrome 的 V8 引擎，所以垃圾回收机制也属于 V8 的内部垃圾回收机制。

V8 的垃圾回收机制根据对象的存活时间采用了不同的算法，使得垃圾回收变得更高效。

在 V8 中，内存分为新生代与老生代。

对于新生代的内存采取的是将内存区一分为二，将存活的对象从一个区复制到另一个区，然后对原有的区进行内存释放，反复如此。当一个对象经过多次复制依然存活时，这个较长生命周期的对象会被移动到老生代中。

对于老生代的垃圾回收采用的是标记清除算法，遍历所有对象并标记仍然存在的对象，然后在清除阶段将没有标记的对象进行清除，最后将清除后的空间进行内存释放。

9.简单聊聊 Node 的异步 I/O

在进程启动时，Node 便会创建一个类似于 while(true) 的循环，每执行一次循环体的过程成为 Tick。每个 Tick 的过程就是查看是否有事件待处理，如果有，就取出事件及其相关的回调函数。如果存在关联的回调函数，就执行它们。然后进入下个循环，如果不再有事件处理，就退出进程。（每一次 Tick 都会把观察者中可执行的事件执行完毕后，再进行下一次的 Tick）

事件循环是一个典型的生产者/消费者模型。异步 I/O、网络请求等则是事件的生产者，生产出的事件被传递到对应的观察者，事件循环从观察者取出事件并处理（消费者）。

```
// 简易型 tick
const events = [];
function observer() {
  setInterval(() => {
    console.log('checking...')
    if (events.length > 0) {
      const event = events.shift();
      event();
    }
  }, 50);}
const fn = () => {
  console.log('event callback');}
setInterval(() => {
  events.push(fn);}, 500);
observer();
```

Node 内部的异步 I/O 流程

1. 发起异步调用
2. 封装异步执行对象，设置回调函数和参数
3. 将异步执行对象推入线程池（主线程继续往下执行）
4. （一段时间后）异步执行对象执行完毕，将执行后的结果连同线程一起交还给主线程；
5. 主线程的 Tick 检测到有执行完成的异步任务，将执行对象取出，执行对应的回调函数；
6. 完成

10.进程的当前工作目录是什么？有什么作用？

进程的当前工作目录默认值是当前进程启动的目录，通过 `process.cwd()` 可以获取当前工作目录（current working directory），文件操作等使用相对路径时会相对当前工作目录来获取文件。

一些获取配置的第三方模块（例如 webpack）就是通过你的当前工作目录来获取对应的配置文件的，在程序中可以通过 `process.chdir()` 来改变当前的工作目录。

11.console.log 是同步还是异步？如何实现一个 console.log？

`console.log` 内部实现是 `process.stdout`，将输入的内容打印到 `stdout`，异步同步取决于 `stdout` 连接的数据流的类型（需要写入的位置）以及不同的操作系统。

- 文件：在 Windows 和 POSIX 上是同步的；
- TTY（终端）：在 Windows 上是异步的，在 POSIX 上是同步；

- 管道（和 socket）：在 Windows 上是同步的，在 POSIX 上是异步的；

造成这种差异的原因是因为一些历史遗留问题，不过这个问题并不会影响正常的输出结果。

12.父进程或子进程的死亡是否会影响对方？什么是孤儿进程？

子进程死亡不会影响父进程，不过子进程死亡时，会向它的父进程发送死亡信号。反之父进程死亡，一般情况下子进程也会随之死亡，但如果此时子进程处于可运行状态、僵死状态等等的话，子进程将被 `init` 进程手痒，从而成为孤儿进程。

另外，子进程死亡的时候（处于“终止状态”），父进程没有及时调

用 `wait()` 或 `waitpid()` 来返回死亡进程的相关信息，此时子进程还有一个 PCB 残留在进程表中，被成为僵尸进程。

13.简单介绍一下 IPC

IPC（Inner-Process Communication）又称进程间通信技术，是用于 Node 内部父子进程之间进行通信的方法。

Node 的 IPC 是通过不同平台的管道技术实现的，特点是本地网络通信，速度快，效率高。

Node 在启动子进程的时候，主进程先建立 IPC 通道，然后将 IPC 通道的 fd（文件描述符）通过环境变量（NODE_CHANNEL_FD）的方式传递给子进程，然后子进程通过 fd 与 父进程建立 IPC 连接。

14. 什么是守护进程？Node 如何实现守护进程？

守护进程是不依赖终端（tty）的进程，不会因为用户退出终端而停止运行的进程。

Node 实现守护进程的思路：

1. 创建一个进程 A；
2. 在进程 A 中创建进程 B，可以使用 child_process.fork 或者其他方法；
3. 启动子进程时，设置 detached 属性为 true，保证子进程在父进程退出后继续运行；
4. 进程 A 退出，进程 B 由 init 进程接管。此时进程 B 为守护进程。

15.简单介绍一下 Buffer

Buffer 是 Node 中用于处理二进制数据的类，其中与 IO 相关的操作（网络/文件等）均基于 Buffer。Buffer 类的实例非常类似于整数数组，但其大小是固定不变的，并且其内存在 V8 堆栈外分配原始内存空间。Buffer 类的实例创建之后，其所占用的内存大小就不能再进行调整。

16.简单介绍一下 Stream

流（stream）是 Node 中处理流式数据的抽象接口，stream 模块用于构建实现了流接口的对象。

Node 中提供了多种流对象，例如 HTTP 服务器的请求 和 process.stdout。

流可以是可读的、可写的、或者可读可写的，所有的流都是 EventEmitter 的实例。

17.什么是粘包问题，如何解决？

默认情况下，TCP 连接会采用延迟传送算法（Nagle 算法），在数据发送之前缓存他们。如果短时间有多个数据发送，会缓冲到一起作一次发送（缓冲大小是 socket.bufferSize），这样可以减少 IO 消耗提高性能。（TCP 会出现这个问题，HTTP 协议解决了这个问题）

解决方法

1. 多次发送之前间隔一个等待时间：处理简单，但是影响传输效率；
2. 关闭 Nagle 算法：消耗资源高，整体性能下降；
3. 封包/拆包：使用一些有标识来进行封包拆包（类似 HTTP 协议头尾）；

18.cookie 与 session 的区别？服务端如何清除 cookie?

主要区别在于，session 存在服务端，cookie 存在客户端。session 比 cookie 更安全，而且 cookie 不一定一直能用（可能被浏览器禁止）。服务端可以通过设 cookie 的值为空并设置一个及时的 expires 来清除存在客户端上的 cookie。

cookie 可能会包含一些关键信息，而 session 一般都是一个加密串。

19.hosts 文件是什么？

hosts 文件是个没有扩展名的系统文件，其作用就是将网址域名和其对应的 IP 地址建立一个关联"数据库"，当用户在浏览器中输入一个 url 时，系统会首先自动从 hosts 文件中寻找对应的 IP 地址。

十．消息队列

1.消息队列的应用场景有哪些？

消息队列的应用场景主要有四个：异步处理、应用解耦、流量削锋和消息通讯。

异步处理：引入消息队列，将不是必须的业务逻辑，推入消息队列做异步处理，从而提高系统并发量与吞吐量。

应用解耦：当两个系统出现强耦合时，可以引入消息队列将两个系统进行解耦，比如订单系统与库存系统。

流量削锋：流量削锋也是消息队列中的常用场景，一般在秒杀和团抢活动中使用广泛！用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

日志处理：日志采集客户端，负责日志数据采集，定时写入 Kafka 队列；Kafka 消息队列：负责日志数据的接收、存储和转发；日志处理应用：订阅并消费 kafka 队列中的日志数据。

十一．大厂面试题

1.（bilibili）编程算法题

用 JavaScript 写一个函数，输入 int 型，返回整数逆序后的字符串。如：输入整型 1234，返回字符串“4321”。要求必须使用递归函数调用，不能用全局变量，输入函数必须只有一个参数传入，必须返回字符串。

```
function reverse(n) {
  let y = n % 10;
  let s = String(y);
  if (n / 10 >= 1) {
    s += reverse((n - y) / 10);
  }
  return s;}
```

2.（携程）算法手写题

已知如下数组：

```
var arr = [ [1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14] ] ] ], 10];
```

编写一个程序将数组扁平化去并除其中重复部分数据，最终得到一个升序且不重复的数组

```
// 思路型 function flatten(arr) {
  let flattenArr = [];
```

```
  for (let i = 0; i < arr.length; i++) {
    const item = arr[i];
    if (Array.isArray(item)) {
      flattenArr = flattenArr.concat(flatten(item))
    } else {
      if (flattenArr.indexOf(item) > -1) continue;
      flattenArr.push(item);
    }
  }
  return flattenArr.sort((a, b) => a - b);}
// API 型 function flatten2(arr) {
  return Array.from(new Set(arr.flat(Infinity))).sort((a, b) => a - b);}
```

3.介绍下深度优先遍历和广度优先遍历，如何实现？

图的遍历

两种遍历算法：

- 深度优先遍历；
- 广度优先遍历；

深度优先遍历（DFS）

深度优先遍历（Depth-First-Search），是搜索算法的一种，它沿着树的深度遍历树的节点，尽可能深地搜索树的分支。当节点 v 的所有边都已被探寻过，将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已探寻源节点到

其他所有节点为止，如果还有未被发现的节点，则选择其中一个未被发现的节点为源节点并重复以上操作，直到所有节点都被探寻完成。

简单的说，DFS 就是从图中的一个节点开始追溯，直到最后一个节点，然后回溯，继续追溯下一条路径，直到到达所有的节点，如此往复，直到没有路径为止。

DFS 可以产生相应图的拓扑排序表，利用拓扑排序表可以解决很多问题，例如最大路径问题。一般用堆数据结构来辅助实现 DFS 算法。

注意：深度 DFS 属于盲目搜索，无法保证搜索到的路径为最短路径，也不是在搜索特定的路径，而是通过搜索来查看图中有哪些路径可以选择。

广度优先遍历（BFS）

广度优先遍历（Breadth-First-Search）是从根节点开始，沿着图的宽度遍历节点，如果所有的节点均被访问过，则算法终止，BFS 同样属于盲目搜索，一般用队列数据结构来辅助实现 BFS。

BFS 从一个节点开始，尝试访问尽可能接近它的目标节点。本质上这种遍历在图上是逐层移动的，首先检查最靠近第一个节点的层，再逐渐向下移动到离起始节点最远的层。

4.（网易）简单讲解一下 http2 的多路复用

在 HTTP/1 中，每次请求都会建立一次 HTTP 连接，也就是我们常说的 3 次握手和 4 次挥手，这个过程在一次请求过程中占用了相当长的时间，即使开启了 `Keep-Alive`，解决了多次连接的问题，但是依然有两个效率上的问题，一是串行的文件传输，二是连接数过多导致的性能问题。

HTTP/2 的多路复用就是为了解决上述的两个性能问题。

在 HTTP/2 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。

多路复用，就是在一个 TCP 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 HTTP 旧版本中的队头阻塞问题，极大的提高传输性能。

5.（挖财）什么是防抖和节流？有什么区别？如何实现？

防抖

触发高频事件后 n 秒内函数只会执行一次，如果 n 秒内高频事件再次被触发，则重新计算时间。

```
function debounce(fn, timing) {
  let timer;
  return function() {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn();
    }, timing);
  }}
```

节流

高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的执行效率。

```
function throttle(fn, timing) {
  let trigger;
```

```
  return function() {
    if (trigger) return;
    trigger = true;
    fn();
    setTimeout(() => {
      trigger = false;
    }, timing);
  }}
```

Tips：我记这个很容易把两者弄混，总结了个口诀，就是 DTTV（Debounce Timer Throttle Variable - 防抖靠定时器控制，节流靠变量控制）。

6.（头条、微医）Async/Await 如何通过同步的方式实现异步

Async/Await 是一个自执行的 generate 函数。利用 generate 函数的特性把异步的代码写成"同步"的形式。

```
var fetch = require("node-fetch");
function *gen() { // 这里的 * 可以看成 async
  var url = "https://api.github.com/users/github";
  var result = yield fetch(url); // 这里的 yield 可以看成 await
  console.log(result.bio);}
```

```
var g = gen();var result = g.next();result.value.then(data =>
data.json()).then(data  => g.next(data));
```

7.（滴滴、挖财、微医、海康）JS 异步解决方案的发展历程以及优缺点。

回调函数

优点：解决了同步的问题（整体任务执行时长）；

缺点：回调地狱，不能用 `try catch` 捕获错误，不能 `return`;

Promise

优点：解决了回调地狱的问题；

缺点：无法取消 `Promise`，错误需要通过回调函数来捕获；

Generator

特点：可以控制函数的执行。

Async/Await

优点：代码清晰，不用像 `Promise` 写一大堆 `then` 链，处理了回调地狱的问题；

缺点：`await` 将异步代码改造成同步代码，如果多个异步操作没有依赖性而使用 `await` 会导致性能上的降低；

8.（兑吧）情人节福利题，如何实现一个 new

```
function _new(fn, ...args) {

  let obj = Object.create(fn.prototype);
```

```
  let ret = fn.apply(obj, args);
  return ret instanceof Object ? ret : obj;}
```

9.（京东）下面代码中 a 在什么情况下会打印 1？

```
var a = ?;if(a == 1 && a == 2 && a == 3){
        console.log(1);}
```

解答：

```
var a = {
  value: 0,

  valueOf() {
    return ++this.value;
  }};
```

10.（百度）实现 (5).add(3).minus(2) 功能

```
Number.prototype.add = function(n) {
  return this + n;}
Number.prototype.minus = function(n) {
  return this - n;}
```

11.写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么？

vue 和 react 都是采用 diff 算法来对比新旧虚拟节点，从而更新节点。在 vue 的 diff 函数交叉对比中，当新节点跟旧节点头尾交叉对比没有结果时，会根据新节点的 key 去对比旧节点数组中的 key，从而找到相应旧节点（这里对应的是一个 key

=> index 的 map 映射）。如果没有找到就认为是一个新增节点。而如果没有 key，那么就会采用遍历查找的方式去找到对应的旧节点。一种一个 map 映射，另一种是遍历查找。相比而言，map 映射的速度更快。

12.简述执行上下文和执行栈

执行上下文

- 全局执行上下文：默认的上下文，任何不在函数内部的代码都在全局上下文里面。它会执行两件事情：创建一个全局的的 window 对象，并且设置 this 为这个全局对象。一个程序只有一个全局对象。
- 函数执行上下文：每当一个函数被调用时，就会为该函数创建一个新的上下文，每个函数都有自己的上下文，不过是在被函数调用的时候创建的。函数上下文可以有任意多个，每当一个新的执行上下文被创建，他会按照定义的顺序执行一系列的步骤。
- Eval 函数执行上下文：执行在 `eval` 函数内部的代码有他自己的执行上下文。

执行栈

执行栈就是一个调用栈，是一个后进先出数据结构的栈，用来存储代码运行时创建的执行上下文。

this 绑定

全局执行上下文中，`this` 指向全局对象。

函数执行上下文中，this 取决于函数是如何被调用的。如果他被一个引用对象调用，那么 `this` 会设置成那个对象，否则是全局对象。

13.Vue 组件间如何通信？

父子组件通信

1. props + emit
2. $refs + $parent
3. provider/inject

兄弟组件通信

1. eventBus
2. $parent.$refs

14.简述前端性能优化

页面内容方面

1. 通过文件合并、css 雪碧图、使用 base64 等方式来减少 HTTP 请求数，避免过多的请求造成等待的情况；
2. 通过 DNS 缓存等机制来减少 DNS 的查询次数；
3. 通过设置缓存策略，对常用不变的资源进行缓存；
4. 通过延迟加载的方式，来减少页面首屏加载时需要请求的资源，延迟加载的资源当用户需要访问时，再去请求加载；
5. 通过用户行为，对某些资源使用预加载的方式，来提高用户需要访问资源时的响应速度；

服务器方面

1. 使用 CDN 服务，来提高用户对于资源请求时的响应速度；
2. 服务器端自用 Gzip、Deflate 等方式对于传输的资源进行压缩，减少传输文件的体积；

3. 尽可能减小 cookie 的大小，并且通过将静态资源分配到其他域名下，来避免对静态资源请求时携带不必要的 cookie；

15.如何实现数组的随机排序？

```
// 随机数排序 function random1(arr) {
  return arr.sort(() => Math.random() - .5);}
// 随机插入排序 function random2(arr) {
  const cArr = [...arr];
  const newArr = [];
  while (cArr.length) {
    const index = Math.floor(Math.random() * cArr.length);
    newArr.push(cArr[index]);
    cArr.splice(index, 1);
  }
  return newArr;}
// 洗牌算法，随机交换排序 function random3(arr) {
  const l = arr.length;
  for (let i = 0; i < l; i++) {
    const index = Math.floor(Math.random() * (l - i)) + i;
    const temp = arr[index];
    arr[index] = arr[i];
    arr[i] = temp;
  }
  return arr;}
```

16.（阿里巴巴）介绍下 CacheStorage

CacheStorage 接口表示 Cache 对象的存储。它提供了一个 ServiceWorker、其他类型 woker 或者 window 范围内可以访问到的所有命名 cache 的主目录（它并不是一定要和 service workers 一起使用，即使它是在 service workers 规范中定义的），并维护一份字符串名称到相应 Cache 对象的映射。

CacheStorage 和 Cache，是两个与缓存相关的接口，用于管理当前网页/Web App 的缓存；在使用 Service Worker 时基本都会用到。它们与数据库有点类似，我们可以用 mongodb 来打个比喻：

- CacheStorage 管理者所有的 Cache，是整个缓存 api 的入口，类似于 mongo；
- Cache 是单个缓存库，通常一个 app 会有一个，类似 mongo 里的每个 db；

无论在 ServiceWorker 域或 window 域下，你都可以用 caches 来访问全局的 CacheStorage。

17.（阿里巴巴）Vue 双向数据绑定原理

vue 通过双向数据绑定，来实现了 View 和 Model 的同步更新。vue 的双向数据绑定主要是通过数据劫持和发布订阅者模式来实现的。

首先我们通过 Object.defineProperty() 方法来对 Model 数据各个属性添加访问器属性，以此来实现数据的劫持，因此当 Model 中的数据发生变化的时候，

我们可以通过配置的 setter 和 getter 方法来实现对 View 层数据更新的通知。

对于文本节点的更新，我们使用了发布订阅者模式，属性作为一个主题，我们为这个节点设置一个订阅者对象，将这个订阅者对象加入这个属性主题的订阅者列表中。当 Model 层数据发生改变的时候，Model 作为发布者向主题发出通知，主题收到通知再向它的所有订阅者推送，订阅者收到通知后更改自己的数据。

18.页面的可用性时间的计算

Performance 接口可以获取到当前页面中与性能相关的信息。

- Performance.timing：`Performance.timing` 对象包含延迟相关的性能信息；

19.简述一下 WebAssembly

WebAssembly 是一种新的编码方式，可以在现代的网络浏览器中运行 - 它是一种低级的类汇编语言，具有紧凑的二进制格式，可以接近原生的性能运行，并为诸如 C/C++ 等语言提供一个编译目标，以便它们可以在 Web 上运行。它也被设计为可以与 Javascript 共存，允许两者一起工作。

WebAssembly 提供了一条途径，以使得以各种语言编写的代码都可以以接近原生的速度在 Web 中运行。

20.（阿里巴巴）谈谈移动端点击

移动端 300 ms 点击（click 事件）延迟

由于移动端会有双击缩放的这个操作，因此浏览器在 click 之后要等待 300ms，判断这次操作是不是双击。

解决方案：

1. 禁用缩放：user-scalable=no
2. 更改默认的视口宽度
3. CSS touch-action

点击穿透问题

因为 click 事件的 300ms 延迟问题，所以有可能会在某些情况触发多次事件。

解决方案：

1. 只用 touch
2. 只用 click

21.（阿里巴巴）谈谈 Git-Rebase

1. 可以合并多次提交记录，减少无用的提交信息；
2. 合并分支并且减少 commit 记录；

22.（腾讯）webpack 中 loader 和 plugin 的区别是什么？

loader：loader 是一个转换器，将 A 文件进行编译成 B 文件，属于单纯的文件转换过程；

plugin：plugin 是一个扩展器，它丰富了 webpack 本身，针对是 loader 结束后，webpack 打包的整个过程，它并不直接操作文件，而是基于事件机制工作，会监听 webpack 打包过程中的某些节点，执行广泛的任务。

23.谈谈对 MVC、MVP、MVVM 模式的理解

在开发图形界面应用程序的时候，会把管理用户界面的层次称为 View，应用程序的数据为 Model，Model 提供数据操作的接口，执行相应的业务逻辑。

MVC

MVC 除了把应用程序分为 View、Model 层，还额外的加了一个 Controller 层，它的职责是进行 Model 和 View 之间的协作（路由、输入预处理等）的应由逻辑（application logic）；Model 进行处理业务逻辑。

用户对 View 操作以后，View 捕获到这个操作，会把处理的权利交移给 Controller（Pass calls）；Controller 会对来自 View 数据进行预处理、决定调用哪个 Model 的接口；然后由 Model 执行相关的业务逻辑；当 Model 变更了以后，会通过观察者模式（Observer Pattern）通知 View；View 通过观察者模式收到 Model 变更的消息以后，会向 Model 请求最新的数据，然后重新更新界面。

MVP

和 MVC 模式一样，用户对 View 的操作都会从 View 交易给 Presenter。Presenter 会执行相应的应用程序逻辑，并且会对 Model 进行相应的操作；而这时候 Model 执行业务逻辑以后，也是通过观察者模式把自己变更的消息传递出去，但是是传给 Presenter 而不是 View。Presenter 获取到 Model 变更的消息以后，通过 View 提供的接口更新界面。

MVVM

MVVM 可以看做是一种特殊的 MVP（Passive View）模式，或者说是对 MVP 模式的一种改良。

MVVM 代表的是 Model-View-ViewModel，可以简单把 ViewModel 理解为页面上所显示内容的数据抽象，和 Domain Model 不一样，ViewModel 更适合用来描述 View。 MVVM 的依赖关系和 MVP 依赖关系一致，只不过是把 P 换成了 VM。

MVVM 的调用关系：

MVVM 的调用关系和 MVP 一样。但是，在 ViewModel 当中会有一个叫 Binder，或者是 Data-binding engine 的东西。以前全部由 Presenter 负责的 View 和 Model 之间数据同步操作交由给 Binder 处理。你只需要在 View 的模板语法当中，指令式声明 View 上的显示的内容是和 Model 的哪一块数据绑定的。当 ViewModel 对进行 Model 更新的时候，Binder 会自动把数据更新到 View 上，当用户对 View 进行操作（例如表单输入），Binder 也会自动把数据更新到 Model 上。这种方式称为：Two-way data-binding，双向数据绑定。可以简单而不恰当地理解为一个模板引擎，但是会根据数据变更实时渲染。

```
// "http://www.baidu.com"
```