开课吧 Web 高级工程师 - 4月份面试题汇总

1.解释css sprites， 如何使用?

> 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。
>
> CSS Sprites为一些大型的网站节约了带宽，让提高了用户的加载速度和用户体验，不需要加载更多的图片。

2.解释下为什么接下来这段代码不是IIFE(立即调用的函数表达式)?

```
function foo(){//code... }()
```

> **解析**
>
> 以function关键字开头的语句会被解析为函数声明，而函数声明是不允许直接运行的。只有当解析器把这句话解析为函数表达式，才能够直接运行，怎么办呢？以运算符开头就可以了。
>
> `(function foo(){// code.. })()`
>
> 更加详细的解释可参考以下链接：
>
> https://swordair.com/function-and-exclamation-mark/

3.CSS中link和@import的区别是什么?

> (1) link属于HTML标签，而@import是CSS提供的
>
> (2) 页面被加载的时，link会同时被加载，而@import被引用的CSS会等到引用它的 CSS文件被加载完再加载
>
> (3) import只在IE5以上才能识别，而link是HTML标签，无兼容问题
>
> (4) link方式的样式的权重 高于@import的权重

4.什么叫优雅降级和渐进增强?

> 渐进增强(progressive enhancement)：针对低版本浏览器进行构建页面，保证最基本的功能，然后再针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。
>
> 优雅降级(graceful degradation)：一开始就构建完整的功能，然后再针对低版本浏览器进行兼容。
>
> 区别：
>
> (1) 优雅降级是从复杂的现状开始，并试图减少用户体验的供给

(2) 渐进增强则是从一个非常基础的，能够起作用的版本开始，并不断扩充，以适应未来环境的需要

(3) 降级（功能衰减）意味着往回看；而渐进增强则意味着朝前看，同时保证其根基处于安全地带

5.前端需要注意哪些SEO?

（1）合理的 title、description、keywords：搜索对着三项的权重逐个减小，title 值强调重点即可，重 要关键词出现不要超过2次，而且要靠前，不同页面 title 要有所不同；description 把页面内容高度 概括，长度合适，不可过分堆砌关键词，不同页面 description 有所不同；keywords 列举出重要 关键词即可。

（2）语义化的 HTML 代码，符合 W3C 规范：语义化代码让搜索引擎容易理解网页。

（3）重要内容 HTML 代码放在最前：搜索引擎抓取 HTML 顺序是从上到下，有的搜索引擎对抓取长度 有限制，保证重要内容肯定被抓取。

（4）重要内容不要用 js 输出：爬虫不会执行 js 获取内容

（5）少用 iframe：搜索引擎不会抓取 iframe 中的内容

（6）非装饰性图片必须加 alt

（7）提高网站速度：网站速度是搜索引擎排序的一个重要指标

6.网页验证码是干嘛的，是为了解决什么安全问题?

（1）区分用户是计算机还是人的公共全自动程序。可以防止恶意破解密码、刷票、论坛灌水

（2）有效防止黑客对某一个特定注册用户用特定程序暴力破解方式进行不断的登陆尝试

7 .如果需要手动写动画，你认为最小时间间隔是多久，为什么?

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

8. Object.is() 与原来的比较操作符 “===”、“==” 的区别?

(1) 两等号判等，会在比较时进行类型转换；

(2)三等号判等（判断严格），比较时不进行隐式类型转换，（类型不同则会返回false）；

(3)Object.is 在三等号判等的基础上特别处理了 NaN 、-0 和 +0 ，保证 -0 和 +0 不再相同，但 Object.is(NaN, NaN) 会返回 true。

Object.is 应被认为有其特殊的用途，而不能用它认为它比其它的相等对比更宽松或严格。

9.用于预格式化文本的标签是?

预格式化就是保留文字在源码中的格式 最后显示出来样式与源码中的样式一致 所见即所得。

定义预格式文本，保持文本原有的格式

10. png、jpg、gif 这些图片格式解释一下，分别什么时候用。有没有了解过webp?

（1）BMP，是无损的、既支持索引色也支持直接色的、点阵图。这种图片格式几乎没有对数据进行压缩，所以BMP格式的图片通常具有较大的文件大小。

（2）GIF是无损的、采用索引色的、点阵图。采用LZW压缩算法进行编码。文件小，是GIF格式的优点，同时，GIF格式还具有支持动画以及透明的优点。但，GIF格式仅支持8bit的索引色，所以GIF格式适用于对色彩要求不高同时需要文件体积较小的场景。

（3）JPEG是有损的、采用直接色的、点阵图。JPEG的图片的优点，是采用了直接色，得益于更丰富的色彩，JPEG非常适合用来存储照片，与GIF相比，JPEG不适合用来存储企业Logo、线框类的图。因为有损压缩会导致图片模糊，而直接色的选用，又会导致图片文件较GIF更大。

（4）PNG-8是无损的、使用索引色的、点阵图。PNG是一种比较新的图片格式，PNG-8是非常好的GIF格式替代者，在可能的情况下，应该尽可能的使用PNG-8而不是GIF，因为在相同的图片效果下，PNG-8具有更小的文件体积。除此之外，PNG-8还支持透明度的调节，而GIF并不支持。现在，除非需要动画的支持，否则我们没有理由使用GIF而不是PNG-8。

（5）PNG-24是无损的、使用直接色的、点阵图。PNG-24的优点在于，它压缩了图片的数据，使得同样效果的图片，PNG-24格式的文件大小要比BMP小得多。当然，PNG24的图片还是要比JPEG、GIF、PNG-8大得多。

（6）SVG是无损的、矢量图。SVG是矢量图。这意味着SVG图片由直线和曲线以及绘制它们的方法组成。当你放大一个SVG图片的时候，你看到的还是线和曲线，而不会出现像素点。这意味着SVG图片在放大时，不会失真，所以它非常适合用来绘制企业Logo、Icon等。

（7）WebP是谷歌开发的一种新图片格式，WebP是同时支持有损和无损压缩的、使用直接色的、点阵图。使用webp格式的最大的优点是，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小，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，目前在兼容性上还不是太好。

11.一个tcp链接能发几个http请求?

答案:

如果是 HTTP 1.0 版本协议，一般情况下，不支持长连接，因此在每次请求发送完毕之后，TCP 连接即会断开，因此一个 TCP 发送一个 HTTP 请求，但是有一种情况可以将一条 TCP 连接保持在活跃状态，那就是通过 Connection 和 Keep-Alive 首部，在请求头带上 Connection: Keep-Alive，并且可以通过 Keep-Alive 通用首部中指定的，用逗号分隔的选项调节 keep-alive 的行为，如果客户端和服务端都支持，那么其实也可以发送多条，不过此方式也有限制，可以关注《HTTP 权威指南》4.5.5 节对于 Keep-Alive 连接的限制和规则。

而如果是 HTTP 1.1 版本协议，支持了长连接，因此只要 TCP 连接不断开，便可以一直发送 HTTP 请求，持续不断，没有上限； 同样，如果是 HTTP 2.0 版本协议，支持多用复用，一个 TCP 连接是可以并发多个 HTTP 请求的，同样也是支持长连接，因此只要不断开 TCP 的连接，HTTP 请求数也是可以没有上限地持续发送

12.Virtual DOM的优势在哪里?

「Virtual Dom 的优势」其实这道题目面试官更想听到的答案不是上来就说「直接操作/频繁操作 DOM 的性能差」，如果 DOM 操作的性能如此不堪，那么 jQuery 也不至于活到今天。所以面试官更想听到 VDOM 想解决的问题以及为什么频繁的 DOM 操作会性能差。

首先我们需要知道：DOM 引擎、JS 引擎 相互独立，但又工作在同一线程（主线程） JS 代码调用 DOM API 必须 挂起 JS 引擎、转换传入参数数据、激活 DOM 引擎，DOM 重绘后再转换可能有的返回值，最后激活 JS 引擎并继续执行若有频繁的 DOM API 调用，且浏览器厂商不做“批量处理”优化， 引擎间切换的单位代价将迅速积累若其中有强制重绘的 DOM API 调用，重新计算布局、重新绘制图像会引起更大的性能消耗。其次是 VDOM 和真实 DOM 的区别和优化：

- 虚拟 DOM 不会立马进行排版与重绘操作
- 虚拟 DOM 进行频繁修改，然后一次性比较并修改真实 DOM 中需要改的部分，最后在真实 DOM 中进行排版与重绘，减少过多DOM节点排版与重绘损耗
- 虚拟 DOM 有效降低大面积真实 DOM 的重绘与排版，因为最终与真实 DOM 比较差异，可以只渲染局部

13.common.js和es6中模块引入的区别?

答案:

CommonJS 是一种模块规范，最初被应用于 Nodejs，成为 Nodejs 的模块规范。运行在浏览器端的 JavaScript 由于也缺少类似的规范，在 ES6 出来之前，前端也实现了一套相同的模块规范 (例如: AMD)，用来对前端模块进行管理。自 ES6 起，引入了一套新的 ES6 Module 规范，在语言标准的层面上实现了模块功能，而且实现得相当简单，有望成为浏览器和服务器通用的模块解决方案。但目前浏览器对 ES6 Module 兼容还不太好，我们平时在 Webpack 中使用的 export 和 import，会经过 Babel 转换为 CommonJS 规范。在使用上的差别主要有：

- 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 模块是运行时加载，ES6 模块是编译时输出接口。
- CommonJs 是单个值导出，ES6 Module可以导出多个
- CommonJs 是动态语法可以写在判断里，ES6 Module 静态语法只能写在顶层
- CommonJs 的 this 是当前模块，ES6 Module的 this 是 undefined

14.首屏和白屏时间如何计算?

答案:

首屏时间的计算，可以由 Native WebView 提供的类似 onload 的方法实现，在 ios 下对应的是 webViewDidFinishLoad，在 android 下对应的是onPageFinished事件。白屏的定义有多种。可以认为“没有任何内容”是白屏，可以认为“网络或服务异常”是白屏，可以认为“数据加载中”是白屏，可以认为“图片加载不出来”是白屏。场景不同，白屏的计算方式就不相同。

方法1：当页面的元素数小于x时，则认为页面白屏。比如“没有任何内容”，可以获取页面的DOM节点数，判断DOM节点数少于某个阈值X，则认为白屏。 方法2：当页面出现业务定义的错误码时，则认为是白屏。比如“网络或服务异常”。 方法3：当页面出现业务定义的特征值时，则认为是白屏。比如“数据加载中”。

15.小程序和H5什么区别?

渲染方式与 H5 不同，小程序一般是通过 Native 原生渲染的，但是小程序同时也支持 web 渲染，如果使用 web 渲染的方式，我们需要初始化一个WebView 组件，然后在 WebView 中加载 H5 页面；

所以当我们开发一个小程序时，通常会使用 hybrid 的方式，即会根据具体情况选择部分功能用小程序原生的代码来开发，部分功能通过 WebView 加载 H5 页面来实现。Native 与 Web 渲染混合使用，以实现项目的最优解；

这里值得注意的是，小程序下，native 方式通常情况下性能要优于 web 方式。小程序特有的双线程设计。 H5 下我们所有资源通常都会打到一个 bundle.js 文件里（不考虑分包加载），而小程序编译后的结果会有两个bundle，index.js封装的是小程序项目的 view 层，以及 index.worker.js 封装的是项目的业务逻辑，在运行时，会有两条线程来分别处理这两个bundle，一个是主渲染线程，它负责加载并渲染 index.js 里的内容，另外一个是 Service Worker线 程，它负责执行 index.worker.js 里封装的业务逻辑，这里面会有很多对底层api调用。

16.了解v8引擎吗，一段js代码是如何执行的?

答案:

在执行一段代码时，JS 引擎会首先创建一个执行栈然后JS引擎会创建一个全局执行上下文，并push到执行栈中, 这个过程JS引擎会为这段代码中所有变量分配内存并赋一个初始值（undefined），在创建完成后，JS引擎会进入执行阶段，这个过程JS引擎会逐行的执行代码，即为之前分配好内存的变量逐个赋值(真实值)。

如果这段代码中存在function的声明和调用，那么JS引擎会创建一个函数执行上下文，并push到执行栈中，其创建和执行过程跟全局执行上下文一样。但有特殊情况，即当函数中存在对其它函数的调用时，JS引擎会在父函数执行的过程中，将子函数的全局执行上下文push到执行栈，这也是为什么子函数能够访问到父函数内所声明的变量。

还有一种特殊情况是，在子函数执行的过程中，父函数已经return了，这种情况下，JS引擎会将父函数的上下文从执行栈中移除，与此同时，JS引擎会为还在执行的子函数上下文创建一个闭包，这个闭包里保存了父函数内声明的变量及其赋值，子函数仍然能够在其上下文中访问并使用这边变量/常量。当子函数执行完毕，JS引擎才会将子函数的上下文及闭包一并从执行栈中移除。

最后，JS引擎是单线程的，那么它是如何处理高并发的呢？即当代码中存在异步调用时JS是如何执行的。比如setTimeout或fetch请求都是non-blocking的，当异步调用代码触发时，JS引擎会将需要异步执行的代码移出调用栈，直到等待到返回结果，JS引擎会立即将与之对应的回调函数push进任务队列中等待被调用，当调用(执行)栈中已经没有需要被执行的代码时，JS引擎会立刻将任务队列中的回调函数逐个push进调用栈并执行。这个过程我们也称之为事件循环。

附言：需要更深入的了解JS引擎，必须理解几个概念，执行上下文，闭包，作用域，作用域链，以及事件循环。建议去网上多看看相关文章，这里推荐一篇非常精彩的博客，对于JS引擎的执行做了图形化的说明，更加便于理解。

17.请解释原型设计模式

答案:

原型模式可用于创建新对象，但它创建的不是非初始化的对象，而是使用原型对象（或样本对象）的值进行初始化的对象。原型模式也称为属性模式。

原型模式在初始化业务对象时非常有用，业务对象的值与数据库中的默认值相匹配。原型对象中的默认值被复制到新创建的业务对象中。

经典的编程语言很少使用原型模式，但作为原型语言的 JavaScript 在构造新对象及其原型时使用了这个模式。

18.判断一个给定的字符串是否是重构的

答案:

如果两个字符串是同构的，那么字符串 A 中所有出现的字符都可以用另一个字符替换，以便获得字符串 B，而且必须保留字符的顺序。字符串 A 中的每个字符必须与字符串 B 的每个字符一对一对应。

- paper 和 title 将返回 true。
- egg 和 sad 将返回 false。
- dgg 和 add 将返回 true。

```
isIsomorphic("egg", 'add'); // true
isIsomorphic("paper", 'title'); // true
isIsomorphic("kick", 'side'); // false

function isIsomorphic(firstString, secondString) {

  // 检查长度是否相等，如果不相等，它们不可能是同构的
  if (firstString.length !== secondString.length) return false

  var letterMap = {};

  for (var i = 0; i < firstString.length; i++) {
    var letterA = firstString[i],
        letterB = secondString[i];

    // 如果 letterA 不存在，创建一个 map，并将 letterB 赋值给它
    if (letterMap[letterA] === undefined) {
      letterMap[letterA] = letterB;
    } else if (letterMap[letterA] !== letterB) {
      // 如果 letterA 在 map 中已存在，但不是与 letterB 对应，
      // 那么这意味着 letterA 与多个字符相对应。
      return false;
    }
  }, ,     // 迭代完毕，如果满足条件，那么返回 true。
  // 它们是同构的。
  return true;
}
```

19.调用setState之后发生了什么?

答案:

在代码中调用 setState 函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个 UI 界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

20.为什么虚拟DOM会提高性能?

> 答案：虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而提高性能。用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

---

21. 简述URL和URI的区别?

> 答案：
>
> URI: Uniform Resource Identifier 指的是统一资源标识符
>
> URL: Uniform Resource Location 指的是统一资源定位符
>
> URI 指的是统一资源标识符，用唯一的标识来确定一个资源，它是一种抽象的定义，也就是说，不管使用什么方法来定义，只要能唯 一的标识一个资源，就可以称为 URI。
>
> URL 指的是统一资源定位符，URN 指的是统一资源名称。URL 和 URN 是 URI 的子集，URL 可以理解为使用地址来标识资源。

22.Vue中computed和watch的区别?

> 答案：computed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计算值变化才会返回内容。 watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。

---

23. Vue中$route和$router的区别?

> 答案：
>
> $route 是“路由信息对象”，包括 path，params，hash，query，fullPath，matched，name 等路由信息参数。
>
> $router是“路由实例”对象包括了路由的跳转方法，钩子函数等。

---

24. 如何实现图片懒加载?

> 答案：
>
> 懒加载也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载，这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。它适用于图片很多，页面很长的电商网站的场景。懒加载的实现原理是，将页面上的图片的 src 属性设置为空字符串，将图片的真实路径保存在一个自定义属性中，当页面滚动的时候，进行判断，如果图片进入页面可视区域内，则从自定义属性中取出真实路径赋值给图片的 src 属性，以此来实现图片的延迟加载。

---

25. 下面代码的输出结果是什么？(D)

```
function sayHi() {
  console.log(name);
  console.log(age);
  var name = 'Lydia';
  let age = 21;
}
```

A. undefined 和 undefined;

B. Lydia 和 ReferenceError;

C. ReferenceError 和 21;

D. undefined 和 ReferenceError;

> **解析：**
>
> 本题的考点主要是var与let的区别以及var的预解析问题。var所声明的变量会被预解析，var name;提升到作用域最顶部，所以在开始的console.log(name)时，name已经存在，但是由于没有赋值，所以是undefined；而let会有暂时性死区，也就是在let声明变量之前，你都无法使用这个变量，会抛出一个错误，故选D。

---

26. 下面代码的输出结果是什么？(B)

```
const arr = [1, 2, [3, 4, [5]]];
console.log(arr.flat(1));
```

A. [1, 2, [3, 4, [5]]];

B. [1, 2, 3, 4, [5]];

C. [1, 2, [3, 4, 5]];

D. [1, 2, 3, 4, 5];

> **解析：**
>
> 这里主要是考察Array.prototype.flat方法的使用，扁平化会创建一个新的，被扁平化的数组，扁平化的深度取决于传入的值；这里传入的是1也就是默认值，所以数组只会被扁平化一层，相当于[].concat([1, 2], [3, 4, [5]])，故选B。

27.下面代码的输出结果是什么？(C)

```
const name = 'Lydia Hallie';
const age = 21;

console.log(Number.isNaN(name));
console.log(Number.isNaN(age));

console.log(isNaN(name));
console.log(isNaN(age));
```

A. true false true false

B. true false false false

C. false false true false

D. false true false true

> **解析：**

本题主要考察isNaN和Number.isNaN的区别；首先isNaN在调用的时候，会先将传入的参数转换为数字类型，所以非数字值传入也有可能返回true，所以第三个和第四个打印分别是true false；Number.isNaN不同的地方是，他会首先判断传入的值是否为数字类型，如果不是，直接返回false，本题中传入的是字符串类型，所以第一个和第二个打印均为false，故选C。

28. 请回答其他类型值转换为字符串时的规则

答案：

规范的 9.8 节中定义了抽象操作 ToString，它负责处理非字符串到字符串的强制类型转换。

（1）Null 和 Undefined 类型，null 转换为 "null"，undefined 转换为 "undefined"，（2）Boolean 类型，true 转换为 "true"，false 转换为 "false"。

（3）Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。（4）Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。（5）对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString((Object.prototype.toString())来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。

29. 请回答其他类型值转换为数字时的规则

答案：

有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义了抽象操作 ToNumber。

（1）Undefined 类型的值转换为 NaN。

（2）Null 类型的值转换为 0。

（3）Boolean 类型的值，true 转换为 1，false 转换为 0。

（4）String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。

（5）Symbol 类型的值不能转换为数字，会报错。

（6）对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有valueOf() 方法。

如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用toString() 的返回值（如果存在）来进行强制类型转换.如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

30. Object.is()和"==="、"=="的区别?

答案：

使用双等号进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。 使用三等号进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。 使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两 个 NaN 认定为是相等的。

31. 简述一下src与href的区别?

src用于替换当前元素，href用于在当前文档和引用资源之间确立联系。src是source的缩写，指向外部资源的位置，指向的内容将会嵌入到文档中当前标签所在位置；在请求src资源时会将其指向的资源下载并应用到文档内，例如js脚本，img图片和frame等元素。<script src =”js.js">当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等元素也如此，类似于将所指向资源嵌入当前标签内。这也是为什么将js脚本放在底部而不是头部。

href是Hypertext Reference的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加那么浏览器会识别该文档为css文件，就会并行下载资源并且不会停止对当前文档的处理。

---

32. JS有几种方法判断变量的类型?

(1) 使用typeof检测 当需要变量是否是number, string, boolean, function, undefined, json类型时，可以使用typeof进行判断。 arr, json, nul, date, reg, error 全部被检测为object类型，，其他的变量能够被正确检测出来。

(2) 使用instanceof检测 instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。

(3). 使用constructor检测 constructor本来是原型对象上的属性，指向构造函数。但是根据实例对象寻找属性的顺序，若实例对象上没有实例属性或方法时，就去原型链上寻找，因此，实例对象也是能使用constructor属性的。

---

33. js数组去重，能用几种方法实现?

(1).使用es6 set方法 [...new Set(arr)] let arr = [1,2,3,4,3,2,3,4,6,7,6]; let unique = (arr)=> [...new Set(arr)]; unique(arr);//[1, 2, 3, 4, 6, 7] (2).利用新数组indexOf查找 indexOf() 方法可返回某个指定的元素在数组中首次出现的位置。如果没有就返回-1。 (3).for 双重循环 通过判断第二层循环，去重的数组中是否含有该元素，如果有就退出第二层循环，如果没有j==result.length就相等，然后把对应的元素添加到最后的数组里面。

```
let arr = [1,2,3,4,3,2,3,4,6,7,6];
let result = [];
for(var i = 0 ; i< arr.length; i++) {
    for(var j = 0 ; j < result.length ; j++) {
if( arr[i] === result[j]){
  break;
}
    }
    if(j == result.length){
result.push(arr[i]);
    }
}
console.log(result);
```

(4).利用for嵌套for，然后splice去重

```
function unique(arr){
    for(vari=0; i<arr.length; i++){
    for(varj=i+1; j<arr.length; j++){
if(arr[i]==arr[j]){
//第一个等同于第二个，splice方法删除第二个 arr.splice(j,1); j--;
}
}
}
returnarr;
}
```

(5).

```
let arr = [1,2,3,4,3,2,3,4,6,7,6];
let unique = (arr) => {
    return arr.filter((item,index) => {
return arr.indexOf(item) === index;
    })
};
unique(arr);
```

(6).利用Map数据结构去重

```
let arr = [1,2,3,4,3,2,3,4,6,7,6];
let unique = (arr)=> {
let seen = new Map();
return arr.filter((item) => {
return !seen.has(item) && seen.set(item,1);
    });
};
unique(arr);
```

34. 请分别简述一下防抖与节流?

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；

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频率。

35. Window.onload和DOMContentLoaded区别是什么?

页面加载过程不同

36. 请你谈谈Cookie的弊端

1.`Cookie`数量和长度的限制。每个domain最多只能有20条cookie，每个cookie长度不能超过4KB，否则会被截掉。

2.安全性问题。如果cookie被人拦截了，那人就可以取得所有的session信息。即使加密也与事无补，因为拦截者并不需要知道cookie的意义，他只要原样转发cookie就可以达到目的了。

3.有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务器<端保存一个计数器。如果我们把这个计数器保存在客户端，那么它起不到任何作用。

---

37. JS实现九九乘法表

```
<style>
html,body,ul,li {
padding: 0;
margin: 0;
border: 0;
}
ul {
width: 900px;
overflow: hidden;
margin-top: 4px;
font-size: 12px;
line-height: 36px;
}
li {
float: left;
width: 90px;
margin: 0 4px;
display: inline-block;
text-align: center;
border: 1px solid #333;
background:yellowgreen;
}
</style>

<script>
for(var i = 1; i <= 9; i++){
var myUl = document.createElement('ul');
for(var j = 1; j <= i; j++){
  var myLi = document.createElement('li');
  var myText = document.createTextNode(j + " × " + i + " = " + i*j);
  myLi.appendChild(myText);
  myUl.appendChild(myLi);
}
document.getElementsByTagName('body')[0].appendChild(myUl);
}
</script>
```

---

38. 浏览器输入url到页面呈现出来发生了什么?

1)、进行地址解析 解析出字符串地址中的主机，域名，端口号，参数等 2) 、根据解析出的域名进行DNS解析 1.首先在浏览器中查找DNS缓存中是否有对应的ip地址，如果有就直 接使用，没有机执行第二步 2.在操作系统中查找DNS缓存是否有对应的ip地址，如果有就直接使 用，没有就执行第三步 3.向本地DNS服务商发送请求查找时候有DNS对应的ip地址。如果仍 然没有最后向Root

> Server服务商查询。
>
> 3)、根据查询到的ip地址寻找目标服务器
>
> 1.与服务器建立连接
>
> 2.进入服务器，寻找对应的请求
>
> 4)、浏览器接收到响应码开始处理。
>
> 5)、浏览器开始渲染dom，下载css、图片等一些资源。直到这次请求完成

---

39. 数据库中怎么创建二维表?

> 首先要知道什么是二维表,想必大家都上学的时候都见过课程表吧,最顶部的一行标注星期,最左边的一列标注时间,就是一个X轴,一个Y轴,里面每个单元格的内容对应着两个字段,星期和上课时间,这样的表格就是二维表,那么,怎么创建二维表呢?简单的方法可以通过Excel表格直接导入,在Excel表格中写好一个课程表,在数据库中导入即可.

---

40. 以下代码运行输出为（A）

```
var a = [1, 2, 3],
    b = [1, 2, 3],
    c = [1, 2, 4];
    console.log(a == b);
    console.log(a === b);
    console.log(a > c);
    console.log(a < c);
```

A: false, false, false, true B: false, false, false, false C: true, true, false, true D: other

解析:

> JavaScript中Array的本质也是对象，所以前两个的结果都是false， 而JavaScript中Array的'>'运算符和'<'运算符的比较方式类似于字符串比较字典序，会从第一个元素开始进行比较，如果一样比较第二个，还一样就比较第三个，如此类推，所以第三个结果为false，第四个为true。 综上所述，结果为false, false, false, true，选A

---

41. 以下代码运行结果为(D)

```
var val = 'smtg';
console.log('Value is ' + (val === 'smtg') ? 'Something' : 'Nothing');
```

A: Value is Something B: Value is Nothing C: NaN D: other

解析:

> 这题考的javascript中的运算符优先级，这里'+'运算符的优先级要高于'?'所以运算符，实际上是'Value is true'?'Something' : 'Nothing'，当字符串不为空时，转换为bool为true，所以结果为'Something'，选D

42.[,,,].join(", ")运行结果为 (C)

A: ", , , " B: "undefined, undefined, undefined, undefined" C: ", , " D: ""

解析:

> JavaScript中使用字面量创建数组时，如果最末尾有一个逗号','，会被省略，所以实际上这个数组只有三个元素（都是undefined）： console.log([,].length);//输出结果：//3 而三个元素，使用join方法，只需要添加两次，所以结果为", , "，选C

43.以下代码运行结果为(D)

```
function sidEffecting(ary) {
  ary[0] = ary[2];
}
function bar(a,b,c) {
  c = 10
  sidEffecting(arguments);
  return a + b + c;
}
bar(1,1,1)
```

A: 3 B: 12 C: error D: other

解析:

> 这题考的是JS的函数arguments的概念： 在调用函数时，函数内部的arguments维护着传递到这个函数的参数列表。它看起来是一个数组，但实际上它只是一个有length属性的Object，不从Array.prototype继承。所以无法使用一些Array.prototype的方法。 arguments对象其内部属性以及函数形参创建getter和setter方法，因此改变形参的值会影响到arguments对象的值，反过来也是一样 具体例子可以参见Javascript秘密花园#arguments 所以，这里所有的更改都将生效，a和c的值都为10，a+b+c的值将为21，选D

44.以下代码运行结果为(A)

```
var name = 'World!';
(function () {
    if (typeof name === 'undefined') {
        var name = 'Jack';
        console.log('Goodbye ' + name);
    } else {
        console.log('Hello ' + name);
    }
})();
```

A: Goodbye Jack B: Hello Jack C: Hello undefined D: Hello World

解析:

> 这题考的是javascript作用域中的变量提升，javascript的作用于中使用var定义的变量都会被提升到所有代码的最前面，于是乎这段代码就成了： var name = 'World!'; (function () { var name;// 现在还是undefined if (typeof name === 'undefined') { name = 'Jack'; console.log('Goodbye ' + name); } else { console.log('Hello ' + name); } })(); 这样就很好理解了，typeof name === 'undefined'的结果为true，所以最后会输出'Goodbye Jack'，选A

45.以下代码运行结果为(B)

```
var a = [0];
if ([0]) {
  console.log(a == true);
} else {
  console.log("wut");
}
```

A: true B: false C: "wut" D: other

解析:

> 同样是一道隐式类型转换的题，不过这次考虑的是"运算符，a本身是一个长度为1的数组，而当数组不为空时，其转换成bool值为true。 而左右的转换，会使用如果一个操作值为布尔值，则在比较之前先将其转换为数值的规则来转换，Number([0])，也就是0，于是变成了0 == true，结果自然是false，所以最终结果为B

46.解释css sprites ， 如何使用?

解析:

> 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。

CSS Sprites为一些大型的网站节约了带宽，让提高了用户的加载速度和用户体验，不需要加载更多的图片

47.以下代码运行结果为(D)

```
var ary = Array(3);
ary[0]=2
ary.map(function(elem) { return '1'; });
```

A: [2, 1, 1] B: ["1", "1", "1"] C: [2, "1", "1"] D: other

解析:

> 又是考的Array.prototype.map的用法，map在使用的时候，只有数组中被初始化过元素才会被触发，其他都是undefined，所以结果为[“1”, undefined ,undefined]，选D

48.以下代码运行结果为（B）

```
var a = 11111111111111110000,
    b = 1111;
    console.log(a + b);
```

A: 11111111111111111111 B: 11111111111111110000 C: NaN D: Infinity

解析: 又是一道考查JavaScript数字的题，由于JavaScript实际上只有一种数字形式IEEE 754标准的64位双精度浮点数，其所能表示的整数范围为-253~253(包括边界值)。这里的11111111111111110000已经超过了2^53次方，所以会发生精度丢失的情况。综上选B

49.以下代码运行结果为(C)

```
(function(){
  var x = y = 1;
})();
console.log(y);
console.log(x);
```

A: 1, 1 B: error, error C: 1, error D: other

解析:

> 变量提升和隐式定义全局变量的题，也是一个JavaScript经典的坑... 还是那句话，在作用域内，变量定义和函数定义会先行提升，所以里面就变成了: (function(){ var x; y = 1; x = 1; })(); 这点会问了，为什么不是var x, y;，这就是坑的地方...这里只会定义第一个变量x，而y则会通过不使用var的方式直接使用，于是乎就隐式定义了一个全局变量y 所以，y是全局作用域下，而x则是在函数内部，结果就为1, error，选C

---

50. 伪类和伪元素的区别

> 伪类和伪元素是为了修饰不在文档树中的部分，比如一句话的第一个字母，列表中第一个元素。
>
> **伪类**用于当已有元素处于某种状态时候，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。虽然他和普通的css类似，可以为已有的元素添加样式，但是他只有处于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**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。

---

51. Object.is() 与原来的比较操作符 “===”、“==” 的区别?

回答:

> 使用双等号进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。
>
> 使用三等号进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。
>
> 使用 Object.is 来进行

---

52. 手写 call、apply 及 bind 函数

call函数实现

```
Function.prototype.myCall = function (context) {
  // 判断调用对象
  if (typeof this !== "function") {
    console.error("type error");
  }

  // 获取参数
  let args = [...arguments].slice(1),
    result = null;
```

```
  // 判断 context 是否传入，如果未传入则设置为 window
  context = context || window;

  // 将调用函数设为对象的方法
  context.fn = this;

  // 调用函数
  result = context.fn(...args);

  // 将属性删除
  delete context.fn;

  return result;
}
```

apply 函数实现

```
Function.prototype.myApply = function (context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }

  let result = null;

  // 判断 context 是否存在，如果未传入则为 window
  context = context || window;

  // 将函数设为对象的方法
  context.fn = this;

  // 调用方法
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }

  // 将属性删除
  delete context.fn;

  return result;
}
```

bind 函数实现

```
Function.prototype.myBind = function (context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }

  // 获取参数
  var args = [...arguments].slice(1),
    fn = this;
```

```
    return function Fn() {

      // 根据调用方式，传入不同绑定值
      return fn.apply(this instanceof Fn ? this : context,
args.concat(...arguments));
    }

  }
```

53. 事件委托是什么?

答案

> 事件委托本质上是利用了浏览器事件冒泡的机制。因为事件在冒泡过程中会上传到父节点，并且父节点可以通过事件对象获取到 目标节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件 代理。
>
> 使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实 现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的 监听函数来处理。

54. javascript 代码中的 "use strict"; 是什么意思 ? 使用它区别是什么?

> use strict 指的是严格运行模式，在这种模式对 js 的使用添加了一些限制。比如说禁止 this 指向全局对象，还有禁止使 用 with 语句等。设立严格模式的目的，主要是为了消除代码使用中的一些不安全的使用方式，也是为了消除 js 语法本身的一 些不合理的地方，以此来减少一些运行时的怪异的行为。同时使用严格运行模式也能够提高编译的效率，从而提高代码的运行速度。 我认为严格模式代表了 js 一种更合理、更安全、更严谨的发展方向。

相关知识点:

> use strict 是一种 ECMAscript5 添加的（严格）运行模式，这种模式使得 Javascript 在更严格的条件下运行。
>
> 设立"严格模式"的目的，主要有以下几个:
>
> - 消除 Javascript 语法的一些不合理、不严谨之处，减少一些怪异行为;
> - 消除代码运行的一些不安全之处，保证代码运行的安全;
> - 提高编译器效率，增加运行速度;
> - 为未来新版本的 Javascript 做好铺垫。
>
> 区别:
>
> （1）禁止使用 with 语句。 （2）禁止 this 关键字指向全局对象。 （3）对象不能有重名的属性。

54. get 和 post 请求在缓存方面的区别

> 缓存一般只适用于那些不会更新服务端数据的请求。
>
> 一般 get 请求都是查找请求，不会对服务器资源数据造成修改
>
> 而 post 请求一般都会对服务器数据造成修改，所以，一般会对 get 请求进行缓存，很少会对 post 请求进行缓存。

55. 简单说一下图片的懒加载和预加载

> **懒加载**也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。
>
> 它适用于图片很多，页面很长的电商网站的场景。懒加载的实现原理是，将页面上的图片的 src 属性设置为空字符串，将图片的真实路径保存在一个自定义属性中，当页面滚动的时候，进行判断，如果图片进入页面可视区域内，则从自定义属性中取出真实路径赋值给图片的 src 属性，以此来实现图片的延迟加载。
>
> **预加载**指的是将所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。通过预加载能够减少用户的等待时间，提高用户的体验。我了解的预加载的最常用的方式是使用 js 中的 image 对象，通过为 image 对象来设置 scr 属性，来实现图片的预加载。
>
> 这两种方式都是提高网页性能的方式，两者主要区别是一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓 解压力作用，预加载则会增加服务器前端压力。

---

56. 手写async await

```
function asyncToGenerator(generatorFunc) {
    return function() {
      const gen = generatorFunc.apply(this, arguments)
      return new Promise((resolve, reject) => {
        function step(key, arg) {
          let generatorResult
          try {
            generatorResult = gen[key](arg)
          } catch (error) {
            return reject(error)
          }
          const { value, done } = generatorResult
          if (done) {
            return resolve(value)
          } else {
            return Promise.resolve(value).then(val => step('next', val), err =>
step('throw', err))
          }
        }
        step("next")
      })
    }
}
```

解析

```
function asyncToGenerator(generatorFunc) {
  // 返回的是一个新的函数
  return function() {

    // 先调用generator函数 生成迭代器
    // 对应 var gen = testG()
    const gen = generatorFunc.apply(this, arguments)

    // 返回一个promise 因为外部是用.then的方式 或者await的方式去使用这个函数的返回值的
    // var test = asyncToGenerator(testG)
```

```
    // test().then(res => console.log(res))
    return new Promise((resolve, reject) => {

      // 内部定义一个step函数 用来一步一步的跨过yield的阻碍
      // key有next和throw两种取值，分别对应了gen的next和throw方法
      // arg参数则是用来把promise resolve出来的值交给下一个yield
      function step(key, arg) {
        let generatorResult

        // 这个方法需要包裹在try catch中
        // 如果报错了 就把promise给reject掉 外部通过.catch可以获取到错误
        try {
          generatorResult = gen[key](arg)
        } catch (error) {
          return reject(error)
        }

        // gen.next() 得到的结果是一个 { value, done } 的结构
        const { value, done } = generatorResult

        if (done) {
          // 如果已经完成了 就直接resolve这个promise
          // 这个done是在最后一次调用next后才会为true
          // 以本文的例子来说 此时的结果是 { done: true, value: 'success' }
          // 这个value也就是generator函数最后的返回值
          return resolve(value)
        } else {
          // 除了最后结束的时候外，每次调用gen.next()
          // 其实是返回 { value: Promise, done: false } 的结构，
          // 这里要注意的是Promise.resolve可以接受一个promise为参数
          // 并且这个promise参数被resolve的时候，这个then才会被调用
          return Promise.resolve(
            // 这个value对应的是yield后面的promise
            value
          ).then(
            // value这个promise被resove的时候，就会执行next
            // 并且只要done不是true的时候 就会递归的往下解开promise
            // 对应gen.next().value.then(value => {
            //    gen.next(value).value.then(value2 => {
            //       gen.next()
            //
            //      // 此时done为true了 整个promise被resolve了
            //      // 最外部的test().then(res => console.log(res))的then就开始执行
了

            //    })
            // })
            function onResolve(val) {
              step("next", val)
            },
            // 如果promise被reject了 就再次进入step函数
            // 不同的是，这次的try catch中调用的是gen.throw(err)
            // 那么自然就被catch到 然后把promise给reject掉啦
            function onReject(err) {
              step("throw", err)
            },
          )
        }
      }
```

```
      step("next")
    })
  }
}
```

57. 什么是回流，什么是重绘，有什么区别?

> 当render tree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)**
>
> 当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为**重绘**
>
> **区别:**
>
> 回流必将引起重绘，而重绘不一定会引起回流。比如：只有颜色改变的时候就只会发生重绘而不会引起回流 当页面布局和几何属性改变时就需要回流 比如：添加或者删除可见的DOM元素，元素位置改变，元素尺寸改变——边距、填充、边框、宽度和高度，内容改变

58. 怎么做性能优化

> 1、JavaScript外联文件引用放在html文档底部；CSS外联文件引用在html文档头部，位于head内；
>
> 2、http静态资源尽量用多个子域名；
>
> 3、服务器端提供html文档和http静态资源时，尽量开启gzip压缩；
>
> 4、尽量减少HTTP Requests的数量；
>
> 5、使用雪碧图来减少CSS背景图片的HTTP请求次数；
>
> 6、首屏不需要展示的较大尺寸图片，请使用lazyload；
>
> 7、图片无损压缩的优化；
>
> 8、减少cookies的大小：尽量减少cookies的体积对减少用户获得响应的时间十分重要；
>
> 9、引入textarea/script元素做延迟解析异步渲染

59. vue有什么生命周期？在new Vue 到 vm.$destory的过程经历了什么?

> **生命周期:**
>
> - **初始化阶段**：beforeCreate和create
> - **挂载阶段**：beforeMount和mounted
> - **更新阶段**：beforeUpdate和update
> - **卸载阶段**：beforeDestory和destory
>
> **过程**
>
> 当`new Vue()`后，首先会**初始化事**和**生命周期**，接着会执行**beforeCreate**生命周期钩子，`在这个钩子里面还拿不到`this.$el`和`this.$data`；
>
> 接着往下走会**初始化inject**和**将data的数据进行侦测也就是进行双向绑定**；

> 接着会执`**create钩子函数**，在这个钩子里面能够拿到`this.$data`还拿不到`this.$el`;到这里初始化阶段就走完了。
>
> 然后会进入一个**模版编译阶段**，在这个阶段首先会判断有没有`el`选项如果有的话就继续往下走，如果没有的话会调用`vm.$mount(el)`;
>
> 接着继续判断有没有**template**选项，如果有的话，会将`template`提供的模版编译到`render函数中`；如果没有的话，会通过`el选项`选择模版；到这个编译阶段就结束了。（温馨提示：这个阶段只有完整版的Vue.js才会经历，也是就是通过cmd引入的方式；在单页面应用中，没有这个编译阶段，因为vue-loader已经提前帮编译好，因此，单页面使用的vue.js是运行时的版本）。
>
> 模版编译完之后（这里说的是完整版，如果是运行时的版本会在初始化阶段结束后直接就到挂载阶段），然后进入**挂载阶段**，在挂在阶段首先或**触发`beforeMount钩子**`，在这个钩子里面只能拿到`this.$data`还是拿不到`this.$el`;
>
> 接着会执`**mounted钩子**，在这个钩子里面就既能够拿到`this.$el`也能拿到`this.$data`；到这个挂载阶段就已经走完了，整个实例也已经挂载好了。
>
> 当数据发生变更的时候，就会进入**更新阶段**，首先会触发**beforeUpdate钩子**，然后触**updated钩**`，这个阶段会重新计算生成新的vnode,然后通过patch函数里面的diff算法,将新生成的vnode和缓存中的旧vnode进行一个比对，最后将差异部分更新到视图中。`
>
> 当**vm.$destory**被调用的时候，就会进入**卸载阶段**，在这个阶段，首先触发**beforeDestory钩子**接着触发**destoryed钩子**，在这个阶段Vue会将自身从父组件中删除，取消实例上的所有追踪并且移除所有的事件监听。
>
> 到这里Vue整个生命周期就结束了。
