2020 年 web 前端最全面试题目及答案

目录

HTML 篇

1, doctype 的作用是什么

DOCTYPE 是 html5 标准网页声明，且必须声明在 HTML 文档的第一行。来告知浏览器的解析器用什么文档标准解析这个文档 ,不同的渲染模式会影响到浏览器对于 CSS 代码甚至 JavaScript 脚本的解析

文档解析类型有：

BackCompat：怪异模式，浏览器使用自己的怪异模式解析渲染页面。（如果没有声明 DOCTYPE，默认就是这个模式）

- CSS1Compat：标准模式，浏览器使用 W3C 的标准解析渲染页面。

IE8 还有一种介乎于上述两者之间的近乎标准的模式，但是基本淘汰了。

2，这三种模式的区别是什么？

- 标准模式(standards mode)：页面按照 HTML 与 CSS 的定义渲染
- 怪异模式(quirks mode)模式： 会模拟更旧的浏览器的行为
- 近乎标准(almost standards)模式： 会实施了一种表单元格尺寸的怪异行为(与 IE7 之前的单元格布局方式一致)，除此之外符合标准定义

3，HTML、XHTML、XML 有什么区别

- HTML(超文本标记语言): 在 html4.0 之前 HTML 先有实现再有标准，导致 HTML 非常混乱和松散
- XML(可扩展标记语言): 主要用于存储数据和结构，可扩展，大家熟悉的JSON 也是相似的作用，但是更加轻量高效，所以 XML 现在市场越来越小了
- XHTML(可扩展超文本标记语言): 基于上面两者而来，W3C 为了解决 HTML 混乱问题而生，并基于此诞生了 HTML5，开头加入<!DOCTYPE html>的做法因此而来，如果不加就是兼容混乱的 HTML，加了就是标准模式。

4，什么是 data-属性？

HTML 的数据属性，用于将数据储存于标准的HTML 元素中作为额外信息,我们可以通过 js 访问并操作它，来达到操作数据的目的。

5, 你对 HTML 语义化的理解？

语义化是指使用恰当语义的 html 标签，让页面具有良好的结构与含义，比如<p>标签就代表段落，<article>代表正文内容等等。

语义化的好处主要有两点：

- 开发者友好：使用语义类标签增强了可读性，开发者也能够清晰地看出网页的结构，也更为便于团队的开发和维护
- 机器友好：带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，语义类还可以支持读屏软件，根据文章可以自动生成目录

这对于简书、知乎这种富文本类的应用很重要 ,语义化对于其网站的内容传播有很大的帮助，但是对于功能性的 web 软件重要性大打折扣，比如一个按钮、Skeleton 这种组件根本没有对应的语义，也不需要什么 SEO。

6, HTML5 与 HTML4 的不同之处

文件类型声明（<!DOCTYPE>）仅有一型：<!DOCTYPE HTML>。

新的解析顺序：不再基于 SGML。

- 新的元素：section, video, progress, nav, meter, time, aside, canvas, command, datalist, details, embed, figcaption, figure, footer, header, hgroup, keygen, mark,

output, rp, rt, ruby, source, summary, wbr。

- input 元素的新类型：date, email, url 等等。
- 新的属性：ping（用于 a 与 area），charset（用于 meta），async（用于 script）。
- 全域属性：id, tabindex, repeat。
- 新的全域属性： contenteditable, contextmenu, draggable, dropzone, hidden, spellcheck。
- 移除元素： acronym, applet, basefont, big, center, dir, font, frame, frameset, isindex, noframes, strike, tt

7 有哪些常用的 meta 标签？

meta 标签由 name 和 content 两个属性来定义，来描述一个 HTML 网页文档的元信息，例如作者、日期和时间、网页描述、关键词、页面刷新等，除了一些 http 标准规定了一些 name 作为大家使用的共识，开发者也可以自定义 name。

- charset，用于描述 HTML 文档的编码形式
- http-equiv，顾名思义，相当于 http 的文件头作用,比如下面的代码就可以设置 http 的缓存过期日期
- viewport，移动前端最熟悉不过，Web 开发人员可以控制视口的大小和比例
- apple-mobile-web-app-status-bar-style,开发过 PWA 应用的开发者应该很熟悉 ,为了自定义苹果工具栏的颜色。

8, src 和 href 的区别？

- src 是指向外部资源的位置，指向的内容会嵌入到文档中当前标签所在的位置，在请求 src 资源时会将其指向的资源下载并应用到文档内，如 js 脚本，img 图片和 frame 等元素。当浏览器解析到该元素时，会暂停其他资源的下载和处理，知道将该资源加载、编译、执行完毕，所以一般 js 脚本会放在底部而不是头部。
- href 是指向网络资源所在位置（的超链接），用来建立和当前元素或文档之间的连接，当浏览器识别到它他指向的文件时，就会并行下载资源，不会停止对当前文档的处理。

9 知道 img 的 srcset 的作用是什么？

可以设计响应式图片，我们可以使用两个新的属性 srcset 和 sizes 来提供更多额外的资源图像和提示，帮助浏览器选择正确的一个资源。

srcset 定义了我们允许浏览器选择的图像集，以及每个图像的大小。

sizes 定义了一组媒体条件（例如屏幕宽度）并且指明当某些媒体条件为真时，什么样的图片尺寸是最佳选择。

所以，有了这些属性，浏览器会：

- 查看设备宽度
- 检查 sizes 列表中哪个媒体条件是第一个为真
- 查看给予该媒体查询的槽大小
- 加载 srcset 列表中引用的最接近所选的槽大小的图像

10, 还有哪一个标签能起到跟 srcset 相似作用？

<picture>元素通过包含零或多个 <source> 元素和一个 <img>元素来为不同的显示/设备场景提供图像版本。浏览器会选择最匹配的子 <source> 元素，如果没有匹配的，就选择 <img> 元素的 src 属性中的 URL。然后，所选图像呈现在<img>元素占据的空间中

11, script 标签中 defer 和 async 的区别?

- defer：浏览器指示脚本在文档被解析后执行，script 被异步加载后并不会立刻执行，而是等待文档被解析完毕后执行。
- async：同样是异步加载脚本，区别是脚本加载完毕后立即执行，这导致async 属性下的脚本是乱序的，对于 script 有先后依赖关系的情况，并不适用。

12, 有几种前端储存的方式?

cookies、localstorage、sessionstorage、Web SQL、IndexedDB

13, 这些方式的区别是什么？

- cookies： 在 HTML5 标准前本地储存的主要方式，优点是兼容性好，请求头自带 cookie 方便，缺点是大小只有 4k，自动请求头加入 cookie 浪费流量，每个 domain 限制 20 个 cookie，使用起来麻烦需要自行封装
- localStorage：HTML5 加入的以键值对(Key-Value)为标准的方式，优点是操作方便，

永久性储存（除非手动删除），大小为 5M，兼容 IE8+

- sessionStorage：与 localStorage 基本类似，区别是 sessionStorage 当页面关闭后会被清理，而且与 cookie、localStorage 不同，他不能在所有同源窗口中共享，是会话级别的储存方式
- Web SQL：2010 年被 W3C 废弃的本地数据库数据存储方案，但是主流浏览器（火狐除外）都已经有了相关的实现，web sql 类似于 SQLite，是真正意义上的关系型数据库，用 sql 进行操作，当我们用 JavaScript 时要进行转换，较为繁琐。
- IndexedDB： 是被正式纳入 HTML5 标准的数据库储存方案，它是 NoSQL 数据库，用键值对进行储存，可以进行快速读取操作，非常适合 web 场景，同时用 JavaScript 进行操作会非常方便。

CSS 篇

1，CSS 选择器的优先级是怎样的？

CSS 选择器的优先级是：内联 > ID 选择器 > 类选择器 > 标签选择器

到具体的计算层面，优先级是由 A 、B、C、D 的值来决定的，其中它们的值计算规则如下：

- A 的值等于 1 的前提是存在内联样式，否则 A = 0;
- B 的值等于 ID 选择器 出现的次数;
- C 的值等于 类选择器 和 属性选择器 和 伪类 出现的总次数;
- D 的值等于 标签选择器 和 伪元素 出现的总次数 。

2,link 和@import 的区别？

link 属于 XHTML 标签，而@import 是 CSS 提供的。

页面被加载时，link 会同时被加载，而@import 引用的 CSS 会等到页面被加载完再加载。

import 只在 IE 5 以上才能识别，而 link 是 XHTML 标签，无兼容问题。

link 方式的样式权重高于@import 的权重。

使用 dom 控制样式时的差别。当使用 javascript 控制 dom 去改变样式的时候，只能使用 link 标签，因为@import 不是 dom 可以控制的

2,em\px\rem 区别？

- px：绝对单位，页面按精确像素展示。
- em：相对单位，基准点为父节点字体的大小，如果自身定义了 font-size 按自身来计算（浏览器默认字体是 16px），整个页面内 1em 不是一个固定的值。
- rem：相对单位，可理解为”root em”，相对根节点 html 的字体大小来计算，CSS3 新加属性，chrome/firefox/IE9+支持

3,块级元素水平居中的方法？

- margin:0 auto 方法
- flex 布局，目前主流方法

- table 方法

4，CSS 有几种定位方式？

- static：正常文档流定位，此时 top, right, bottom, left 和 z-index 属性无效，块级元素从上往下纵向排布，行级元素从左向右排列。
- relative：相对定位，此时的『相对』是相对于正常文档流的位置。
- absolute：相对于最近的非 static 定位祖先元素的偏移，来确定元素位置，比如一个绝对定位元素它的父级、和祖父级元素都为 relative，它会相对他的父级而产生偏移。
- fixed：指定元素相对于屏幕视口（viewport）的位置来指定元素位置。元素的位置在屏幕滚动时不会改变，比如那种回到顶部的按钮一般都是用此定位方式。
- sticky：粘性定位，特性近似于 relative 和 fixed 的合体，其在实际应用中的近似效果就是 IOS 通讯录滚动的时候的『顶屁股』。

5，如何理解 z-index？

CSS 中的 z-index 属性控制重叠元素的垂直叠加顺序，默认元素的 z-index 为 0，我们可以修改 z-index 来控制元素的图层位置，而且 z-index 只能影响设置了 position 值的元素。

6，如何理解层叠上下文？

是什么？

层叠上下文是 HTML 元素的三维概念，这些HTML 元素在一条假想的相对于面向（电脑屏幕的）视窗或者网页的用户的 z 轴上延伸，HTML 元素依据其自身属性按照优先级顺序占用层叠上下文的空间。

如何产生？

触发一下条件则会产生层叠上下文：

根元素 (HTML),

z-index 值不为 "auto"的 绝对/相对定位，

一个 z-index 值不为 "auto"的 flex 项目 (flex item)，即：父元素 display: flex|inline-flex，

- opacity 属性值小于 1 的元素（参考 the specification for opacity），
- transform 属性值不为 "none"的元素，
- mix-blend-mode 属性值不为 "normal"的元素，
- filter 值不为“none”的元素，
- perspective 值不为“none”的元素，
- isolation 属性被设置为 "isolate"的元素，
- position: fixed

7，清除浮动有哪些方法？

- 空 div 方法：

- Clearfix 方法：上文使用.clearfix 类已经提到
- overflow: auto 或 overflow: hidden 方法，使用 BFC

8,你对 css sprites 的理解，好处是什么？

是什么 ？

雪碧图也叫 CSS 精灵， 是一 CSS 图像合成技术，开发人员往往将小图标合并在一起之后的图片称作雪碧图。

如何操作？

使用工具（PS 之类的）将多张图片打包成一张雪碧图，并为其生成合适的 CSS。 每张图片都有相应的 CSS 类，该类定义了 background-image、background-position 和 background-size 属性。 使用图片时，将相应的类添加到你的元素中。

好处：

减少加载多张图片的 HTTP 请求数（一张雪碧图只需要一个请求）

提前加载资源

不足：

CSS Sprite 维护成本较高，如果页面背景有少许改动，一般就要改这张合并的图片

加载速度优势在 http2 开启后荡然无存，HTTP2 多路复用，多张图片也可以重复利用一个连接通道搞定

9，你对媒体查询的理解？

是什么

媒体查询由一个可选的媒体类型和零个或多个使用媒体功能的限制了样式表范围的表达式组成，例如宽度、高度和颜色。媒体查询，添加自CSS3，允许内容的呈现针对一个特定范围的输出设备而进行裁剪，而不必改变内容本身,非常适合 web 网页应对不同型号的设备而做出对应的响应适配。

如何使用？

媒体查询包含一个可选的媒体类型和，满足 CSS3 规范的条件下，包含零个或多个表达式，这些表达式描述了媒体特征，最终会被解析为 true 或 false。如果媒体查询中指定的媒体类型匹配展示文档所使用的设备类型，并且所有的表达式的值都是 true，那么该媒体查询的结果为 true.那么媒体查询内的样式将会生效。

10，你对盒模型的理解

是什么？

当对一个文档进行布局（lay out）的时候，浏览器的渲染引擎会根据标准之一的 CSS 基础框盒模型（CSS basic box model），将所有元素表示为一个个矩形的盒子（box）。CSS 决定这些盒子的大小、位置以及属性（例如颜色、背景、边框尺寸…）。

盒模型由 content（内容）、padding（内边距）、border（边框）、margin（外边距）组成

11，标准盒模型和怪异盒模型有什么区别？

在 W3C 标准下，我们定义元素的 width 值即为盒模型中的 content 的宽度值，height 值即为盒模型中的 content 的高度值。

因此，标准盒模型下：

元素的宽度 = margin-left + border-left + padding-left + width + padding-right + border-right + margin-right

而 IE 怪异盒模型（IE8 以下）width 的宽度并不是 content 的宽度，而是 border-left + padding-left + content的宽度值 + padding-right + border-right之和 ,height 同理。

在怪异盒模型下：

元素占据的宽度 = margin-left + width + margin-right

虽然现代浏览器默认使用 W3C 的标准盒模型，但是在不少情况下怪异盒模型更好用，于是 W3C 在 css3 中加入 box-sizing

12，谈谈对 BFC 的理解

是什么？

书面解释：BFC(Block Formatting Context)这几个英文拆解

Block: Block 在这里可以理解为 Block-level Box指的是块级盒子的标准

Formatting context : 块级上下文格式化，它是页面中的一块渲染区域，并且有一套渲染规则，它决定了其子元素将如何定位，以及和其他元素的关系和相互作用

BFC 是指一个独立的渲染区域，只有 Block-level Box参与， 它规定了内部的 Block-level Box 如何布局，并且与这个区域外部毫不相干.

它的作用是在一块独立的区域，让处于 BFC 内部的元素与外部的元素互相隔离.

如何形成？

BFC 触发条件:

根元素，即 HTML 元素

- position: fixed/absolute
- float 不为 none
- overflow 不为 visible
- display 的值为 inline-block、table-cell、table-caption

作用是什么？

- 防止 margin 发生重叠
- 两栏布局，防止文字环绕等
- 防止元素塌陷

13, 为什么有时候人们用 translate 来改变位置而不是定位？

translate()是 transform 的一个值。改变 transform 或 opacity 不会触发浏览器重新布局（reflow）或重绘（repaint），只会触发复合（compositions）。而改变绝对定位会触发重新布局，进而触发重绘和复合。transform 使浏览器为元素创建一个 GPU 图层，但改变绝

对定位会使用到 CPU。 因此 translate()更高效，可以缩短平滑动画的绘制时间。

而 translate 改变位置时，元素依然会占据其原始空间，绝对定位就不会发生这种情况。

14, 伪类和伪元素的区别是什么？

是什么？

伪类（pseudo-class） 是一个以冒号(:)作为前缀，被添加到一个选择器末尾的关键字，当你希望样式在特定状态下才被呈现到指定的元素时 ,你可以往元素的选择器后面加上对应的伪类。

伪元素用于创建一些不在文档树中的元素 ,并为其添加样式。比如说 ,我们可以通过::before 来在一个元素前增加一些文本 , 并为这些文本添加样式。虽然用户可以看到这些文本 , 但是这些文本实际上不在文档树中。

区别

其实上文已经表达清楚两者区别了 ,伪类是通过在元素选择器上加入伪类改变元素状态 ,而伪元素通过对元素的操作进行对元素的改变。

我们通过 p::before 对这段文本添加了额外的元素，通过 p:first-child 改变了文本的样式。

15, 你对 flex 的理解？

web 应用有不同设备尺寸和分辨率 ,这时需要响应式界面设计来满足复杂的布局需求 ,Flex 弹性盒模型的优势在于开发人员只是声明布局应该具有的行为 ,而不需要给出具体的实现方

式，浏览器负责完成实际布局，当布局涉及到不定宽度，分布对齐的场景时，就要优先考虑弹性盒布局

JS 篇

1，解释下变量提升？

JavaScript 引擎的工作方式是，先解析代码，获取所有被声明的变量，然后再一行一行地运行。这造成的结果，就是所有的变量的声明语句，都会被提升到代码的头部，这就叫做变量提升（hoisting）。

2，理解闭包吗？

闭包是什么

MDN 的解释：闭包是函数和声明该函数的词法环境的组合。

按照我的理解就是：闭包 =『函数』和『函数体内可访问的变量总和』

闭包的作用

闭包最大的作用就是隐藏变量，闭包的一大特性就是内部函数总是可以访问其所在的外部函数中声明的参数和变量，即使在其外部函数被返回（寿命终结）了之后

基于此特性，JavaScript 可以实现私有变量、特权变量、储存变量等

我们就以私有变量举例，私有变量的实现方法很多，有靠约定的（变量名前加_），有靠 Proxy 代理的，也有靠 Symbol 这种新数据类型的。

但是真正广泛流行的其实是使用闭包。

3，JavaScript 的作用域链理解吗？

JavaScript 属于静态作用域，即声明的作用域是根据程序正文在编译时就确定的，有时也称为词法作用域。

其本质是 JavaScript 在执行过程中会创造可执行上下文，可执行上下文中的词法环境中含有外部词法环境的引用，我们可以通过这个引用获取外部词法环境的变量、声明等，这些引用串联起来一直指向全局的词法环境，因此形成了作用域链。

4，ES6 模块与 CommonJS 模块有什么区别？

ES6 Module 和 CommonJS 模块的区别：

CommonJS 是对模块的浅拷贝，ES6 Module 是对模块的引用，即 ES6 Module 只存只读，不能改变其值，具体点就是指针指向不能变，类似 const

import 的接口是 read-only（只读状态），不能修改其变量值。 即不能修改其变量的指针指向，但可以改变变量内部指针指向，可以对 commonJS 对重新赋值（改变指针指向），但是对 ES6 Module 赋值会编译报错。

ES6 Module 和 CommonJS 模块的共同点：

CommonJS 和 ES6 Module 都可以对引入的对象进行赋值，即对对象内部属性的值进行改变。

5, js 有哪些类型？

JavaScript 的类型分为两大类，一类是原始类型，一类是复杂(引用）类型。

原始类型:

- boolean
- null
- undefined
- number
- string
- symbol

复杂类型:

- Object

6, null 与 undefined 的区别是什么？

null 表示为空，代表此处不应该有值的存在，一个对象可以是 null，代表是个空对象，而

null 本身也是对象。

undefined 表示『不存在』,JavaScript 是一门动态类型语言 ,成员除了表示存在的空值外，

还有可能根本就不存在(因为存不存在只在运行期才知道),这就是 undefined 的意义所在。

7，谈谈你对原型链的理解？

这个问题关键在于两个点，一个是原型对象是什么，另一个是原型链是如何形成的

原型对象

绝大部分的函数(少数内建函数除外)都有一个 prototype 属性,这个属性是原型对象用来创

建新对象实例,而所有被创建的对象都会共享原型对象,因此这些对象便可以访问原型对象的

属性。

原型链

原因是每个对象都有 __proto__ 属性，此属性指向该对象的构造函数的原型。

对象可以通过 __proto__与上游的构造函数的原型对象连接起来 ,而上游的原型对象也有一

个__proto__，这样就形成了原型链。

8，谈一谈你对 this 的了解？

this 的指向不是在编写时确定的,而是在执行时确定的，同时，this 不同的指向在于遵循了

一定的规则。

首先，在默认情况下，this 是指向全局对象的，比如在浏览器就是指向 window。

其次，如果函数被调用的位置存在上下文对象时，那么函数是被隐式绑定的。

再次，显示改变 this 指向，常见的方法就是 call、apply、bind

最后，也是优先级最高的绑定 new 绑定。

用 new 调用一个构造函数 ,会创建一个新对象, 在创造这个新对象的过程中,新对象会自动绑定到 Person 对象的 this 上，那么 this 自然就指向这个新对象。

9, 那么箭头函数的 this 指向哪里？

箭头函数不同于传统 JavaScript 中的函数,箭头函数并没有属于自己的 this,它的所谓的 this 是捕获其所在上下文的 this 值 ,作为自己的 this 值,并且由于没有属于自己的 this,而箭头函数是不会被 new 调用的，这个所谓的 this 也不会被改变.

10, async/await 是什么？

async 函数，就是 Generator 函数的语法糖，它建立在Promises 上，并且与所有现有的基于 Promise 的 API 兼容。

Async—声明一个异步函数(async function someName(){...})

自动将常规函数转换成 Promise，返回值也是一个 Promise 对象

只有 async 函数内部的异步操作执行完，才会执行 then 方法指定的回调函数

异步函数内部可以使用 await

Await—暂停异步的功能执行(var result = await someAsyncCall()

放置在 Promise 调用之前，await 强制其他代码等待，直到 Promise 完成并返回结果

只能与 Promise 一起使用，不适用与回调

只能在 async 函数内部使用

11, async/await 相比于 Promise 的优势？

代码读起来更加同步，Promise 虽然摆脱了回调地狱，但是then 的链式调用也会带来额外的阅读负担

Promise 传递中间值非常麻烦，而 async/await 几乎是同步的写法，非常优雅

错误处理友好，async/await 可以用成熟的 try/catch，Promise 的错误捕获非常冗余

调试友好，Promise 的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个.then 代码块中使用调试器的步进(step-over)功能，调试器并不会进入后续的.then 代码块，因为调试器只能跟踪同步代码的『每一步』。

12, JavaScript 的参数是按照什么方式传递的？

基本类型传递方式

由于 js 中存在复杂类型和基本类型,对于基本类型而言,是按值传递的.

复杂类型按引用传递?

我们将外部 a 作为一个对象传入 test 函数.

按共享传递

复杂类型之所以会产生这种特性,原因就是在传递过程中,对象 a 先产生了一个副本 a,这个副本 a 并不是深克隆得到的副本 a,副本 a 地址同样指向对象 a 指向的堆内存.

13, 聊一聊如何在 JavaScript 中实现不可变对象？

实现不可变数据有三种主流的方法

- 深克隆，但是深克隆的性能非常差，不适合大规模使用
- Immutable.js，Immutable.js 是自成一体的一套数据结构，性能良好，但是需要学习额外的 API
- immer，利用 Proxy 特性，无需学习额外的 api，性能良好

14, JavaScript 的基本类型和复杂类型是储存在哪里的？

基本类型储存在栈中，但是一旦被闭包引用则成为常住内存，会储存在内存堆中。

复杂类型会储存在内存堆中。

Vue 篇

1, 你对 MVVM 的理解?

MVVM 模式，顾名思义即 Model-View-ViewModel 模式。它萌芽于 2005 年微软推出的基于 Windows 的用户界面框架 WPF ,前端最早的 MVVM 框架 knockout 在 2010 年发布。

- Model 层: 对应数据层的域模型，它主要做域模型的同步。通过 Ajax/fetch 等 API 完成客户端和服务端业务 Model 的同步。在层间关系里，它主要用于抽象出 ViewModel 中视图的 Model。
- View 层:作为视图模板存在，在 MVVM 里，整个 View 是一个动态模板。除了定义结构、布局外，它展示的是 ViewModel 层的数据和状态。View 层不负责处理状态，View 层做的是 数据绑定的声明、 指令的声明、 事件绑定的声明。
- ViewModel 层:把 View 需要的层数据暴露，并对 View 层的 数据绑定声明、 指令声明、 事件绑定声明 负责，也就是处理 View 层的具体业务逻辑。ViewModel 底层会做好绑定属性的监听。当 ViewModel 中数据变化，View 层会得到更新，而当 View 中声明了数据的双向绑定（通常是表单元素），框架也会监听 View 层（表单）值的变化。一旦值变化，View 层绑定的 ViewModel 中的数据也会得到自动更新。

2, MVVM 的优缺点?

优点:

分离视图（View）和模型（Model）,降低代码耦合，提高视图或者逻辑的重用性: 比如视图（View）可以独立于 Model 变化和修改，一个 ViewModel 可以绑定不同的"View"上，当 View 变化的时候 Model 不可以不变，当 Model 变化的时候 View 也可以不变。你可以把一些视图逻辑放在一个 ViewModel 里面，让很多 view 重用这段视图逻辑

提高可测试性: ViewModel 的存在可以帮助开发者更好地编写测试代码

自动更新 dom: 利用双向绑定,数据更新后视图自动更新,让开发者从繁琐的手动 dom 中解放

缺点:

Bug 很难被调试: 因为使用双向绑定的模式，当你看到界面异常了，有可能是你View 的代码有 Bug，也可能是 Model 的代码有问题。数据绑定使得一个位置的 Bug 被快速传递到别的位置，要定位原始出问题的地方就变得不那么容易了。另外，数据绑定的声明是指令式地写在 View 的模版当中的，这些内容是没办法去打断点 debug 的

一个大的模块中 model 也会很大，虽然使用方便了也很容易保证了数据的一致性，当时长期持有，不释放内存就造成了花费更多的内存

对于大型的图形应用程序，视图状态较多，ViewModel 的构建和维护的成本都会比较高

3, 你对 Vue 生命周期的理解？

Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模版、挂载Dom -> 渲染、更新 -> 渲染、卸载等一系列过程，我们称这是 Vue 的生命周期

4, 异步请求适合在哪个生命周期调用？

官方实例的异步请求是在 mounted 生命周期中调用的 ,而实际上也可以在 created 生命周期中调用

5, Vue 组件如何通信？

- props/$emit+v-on: 通过 props 将数据自上而下传递，而通过$emit 和 v-on 来向上传递信息。
- EventBus: 通过 EventBus 进行信息的发布与订阅
- vuex: 是全局数据管理库，可以通过 vuex 管理全局的数据流
- $attrs/$listeners: Vue2.4 中加入的$attrs/$listeners 可以进行跨级的组件通信
- provide/inject：以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在起上下游关系成立的时间里始终生效，这成为了跨组件通信的基础

6, computed 和 watch 有什么区别?

computed:

computed 是计算属性,也就是计算值,它更多用于计算值的场景

computed 具有缓存性,computed 的值在 getter 执行后是会缓存的 ,只有在它依赖的属性值改变之后，下一次获取 computed 的值时才会重新调用对应的 getter 来计算

computed 适用于计算比较消耗性能的计算场景

watch:

更多的是「观察」的作用类似于某些数据的监听回调,用于观察 props $emit 或者本组件的值,当数据变化时来执行回调进行后续操作

无缓存性，页面重新渲染时值不变化也会执行

小结:

当我们要进行数值计算,而且依赖于其他数据，那么把这个数据设计为 computed

如果你需要在某个数据变化时做一些事情，使用 watch 来观察这个数据变化

7, Vue 是如何实现双向绑定的?

利用 Object.defineProperty 劫持对象的访问器,在属性值发生变化时我们可以获取变化,然后根据变化进行后续响应,在 vue3.0 中通过 Proxy 代理对象进行类似的操作。

8, Proxy 与 Object.defineProperty 的优劣对比?

Proxy 的优势如下:

- Proxy 可以直接监听对象而非属性
- Proxy 可以直接监听数组的变化
- Proxy 有多达 13 种拦截方法,不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的
- Proxy 返回的是一个新对象,我们可以只操作新的对象达到目的,而 Object.defineProperty 只能遍历对象属性直接修改
- Proxy 作为新标准将受到浏览器厂商重点持续的性能优化,也就是传说中的新标准的性能红利

Object.defineProperty 的优势如下:

- 兼容性比较好

9，你是如何理解 Vue 的响应式系统的?

任何一个 Vue Component 都有一个与之对应的 Watcher 实例。

Vue 的 data 上的属性会被添加 getter 和 setter 属性。

当 Vue Component render 函数被执行的时候，data 上会被 触碰(touch)，即被读，getter 方法会被调用，此时 Vue 会去记录此 Vue component 所依赖的所有 data。(这一过程被称为依赖收集)

data 被改动时（主要是用户操作），即被写，setter 方法会被调用，此时 Vue 会去通知所有依赖于此 data 的组件去调用他们的 render 函数进行更新。

10，既然 Vue 通过数据劫持可以精准探测数据变化，为什么还需要虚拟 DOM 进行 diff 检测差异?

现代前端框架有两种方式侦测变化,一种是 pull 一种是 push

- pull: 其代表为 React,我们可以回忆一下 React 是如何侦测到变化的,我们通常会用 setStateAPI 显式更新,然后 React 会进行一层层的 Virtual Dom Diff 操作找出差异,然后 Patch 到 DOM 上,React 从一开始就不知道到底是哪发生了变化,只是知道「有变化了」,然后再进行比较暴力的 Diff 操作查找「哪发生变化了」,另外一个代表就是 Angular

的脏检查操作。

push: Vue的响应式系统则是 push 的代表,当 Vue 程序初始化的时候就会对数据 data 进行依赖的收集,一但数据发生变化,响应式系统就会立刻得知,因此 Vue 是一开始就知道是「在哪发生变化了」,但是这又会产生一个问题,如果你熟悉 Vue 的响应式系统就知道,通常一个绑定一个数据就需要一个 Watcher,一但我们的绑定细粒度过高就会产生大量的 Watcher,这会带来内存以及依赖追踪的开销,而细粒度过低会无法精准侦测变化,因此 Vue 的设计是选择中等细粒度的方案,在组件级别进行 push 侦测的方式,也就是那套响应式系统 ,通常我们会第一时间侦测到发生变化的组件 ,然后在组件内部进行 Virtual Dom Diff 获取更加具体的差异,而 Virtual Dom Diff 则是 pull 操作,Vue 是 push+pull 结合的方式进行变化侦测的

11, Vue 为什么没有类似于 React 中 shouldComponentUpdate 的生命周期?

根本原因是 Vue 与 React 的变化侦测方式有所不同

React 是 pull 的方式侦测变化,当 React 知道发生变化后,会使用 Virtual Dom Diff 进行差异检测,但是很多组件实际上是肯定不会发生变化的,这个时候需要用 shouldComponentUpdate 进行手动操作来减少 diff,从而提高程序整体的性能.

Vue 是 pull+push 的方式侦测变化的,在一开始就知道那个组件发生了变化,因此在 push 的阶段并不需要手动控制 diff,而组件内部采用的 diff 方式实际上是可以引入类似于 shouldComponentUpdate 相关生命周期的,但是通常合理大小的组件不会有过量的 diff,手动优化的价值有限,因此目前 Vue 并没有考虑引入 shouldComponentUpdate 这种手动

优化的生命周期

12, Vue 中的 key 到底有什么用？

key 是为 Vue 中的 vnode 标记的唯一 id,通过这个 key,我们的 diff 操作可以更准确、更快速

diff 算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的 key 与旧节点进行比对,然后超出差异.

diff 程可以概括为：oldCh 和 newCh 各有两个头尾的变量 StartIdx 和 EndIdx，它们的 2 个变量相互比较，一共有 4 种比较方式。如果 4 种比较都没匹配，如果设置了 key，就会用 key 进行比较，在比较的过程中，变量会往中间靠，一旦 StartIdx>EndIdx 表明 oldCh 和 newCh 至少有一个已经遍历完了，就会结束比较,这四种比较方式就是首、尾、旧尾新头、旧头新尾.

准确: 如果不加 key,那么 vue 会选择复用节点(Vue 的就地更新策略),导致之前节点的状态被保留下来,会产生一系列的 bug.

快速: key的唯一性可以被Map数据结构充分利用,相比于遍历查找的时间复杂度O(n),Map 的时间复杂度仅仅为 O(1).

React 篇

1, React 最新的生命周期是怎样的?

React 16 之后有三个生命周期被废弃(但并未删除)

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

官方计划在 17 版本完全删除这三个函数，只保留 UNSAVE_前缀的三个函数，目的是为了向下兼容，但是对于开发者而言应该尽量避免使用他们，而是使用新增的生命周期函数替代它们

目前 React 16.8 +的生命周期分为三个阶段,分别是挂载阶段、更新阶段、卸载阶段

挂载阶段:

constructor: 构造函数，最先被执行,我们通常在构造函数里初始化 state 对象或者给自定义方法绑定 this

getDerivedStateFromProps: static getDerivedStateFromProps(nextProps, prevState), 这是个静态方法,当我们接收到新的属性想去修改我们 state，可以使用 getDerivedStateFromProps

render: render 函数是纯函数，只返回需要渲染的东西，不应该包含其它的业务逻辑可以返回原生的 DOM、React 组件、Fragment、Portals、字符串和数字、Boolean 和 null 等内容

componentDidMount: 组件装载之后调用，此时我们可以获取到 DOM 节点并操作，比如对 canvas，svg 的操作，服务器请求，订阅都可以 写在这个里面 ，但是记 得在 componentWillUnmount 中取消订阅

更新阶段:

getDerivedStateFromProps: 此方法在更新个挂载阶段都可能会调用

shouldComponentUpdate: shouldComponentUpdate(nextProps, nextState),有两个参数 nextProps 和 nextState，表示新的属性和变化之后的 state，返回一个布尔值，true 表示会触发重新渲染，false 表示不会触发重新渲染，默认返回 true,我们通常利用此生命周期来优化 React 程序性能

render: 更新阶段也会触发此生命周期

getSnapshotBeforeUpdate: getSnapshotBeforeUpdate(prevProps, prevState),这个方法在 render 之后，componentDidUpdate 之前调用，有两个参数 prevProps 和 prevState，表示之前的属性和之前的 state，这个函数有一个返回值，会作为第三个参数传给 componentDidUpdate，如果你不想要返回值，可以返回 null，此生命周期必须与 componentDidUpdate 搭配使用

componentDidUpdate: componentDidUpdate(prevProps, prevState, snapshot)该方法在 getSnapshotBeforeUpdate 方法之后被调用，有三个参数 prevProps，prevState，snapshot，表示之前的 props，之前的 state，和 snapshot。第三个参数是 getSnapshotBeforeUpdate 返回的,如果触发某些回调函数时需要用到 DOM 元素的状态，则将对比或计算的过程迁移至 getSnapshotBeforeUpdate，然后在 componentDidUpdate 中统一触发回调或更新状态。

卸载阶段:

componentWillUnmount: 当我们的组件被卸载或者销毁了就会调用，我们可以在这个函数里去清除一些定时器，取消网络请求，清理无效的 DOM 元素等垃圾清理工作

2, React 的请求应该放在哪个生命周期中？

React 的异步请求到底应该放在哪个生命周期里,有人认为在 componentWillMount 中可以提前进行异步请求,避免白屏,其实这个观点是有问题的.

由于 JavaScript 中异步事件的性质，当您启动 API 调用时，浏览器会在此期间返回执行其他工作。当 React 渲染一个组件时，它不会等待 componentWillMount 它完成任何事情 - React 继续前进并继续 render,没有办法“暂停”渲染以等待数据到达。

而且在 componentWillMount 请求会有一系列潜在的问题,首先,在服务器渲染时,如果在 componentWillMount 里获取数据，fetch data 会执行两次，一次在服务端一次在客户端 ,这造成了多余的请求,其次,在 React 16进行 React Fiber重写后,componentWillMount 可能在一次渲染中多次调用.

目前官方推荐的异步请求是在 componentDidmount 中进行.

如果有特殊需求需要提前请求,也可以在特殊情况下在 constructor 中请求:

react 17 之后 componentWillMount 会被废弃，仅仅保留 UNSAFE_componentWillMount

3, setState 到底是异步还是同步?

先给出答案: 有时表现出异步,有时表现出同步

setState 只在合成事件和钩子函数中是“异步”的 ,在原生事件和 setTimeout 中都是同步的。

setState 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的 ,只是合成事件和钩子函数的调用顺序在更新之前 ,导致在合成事件和钩子函数中没法立马拿到更新后的值，形成了所谓的“异步”，当然可以通过第二个参

数 setState(partialState, callback) 中的 callback 拿到更新后的结果。

setState 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和 setTimeout 中不会批量更新，在“异步”中如果对同一个值进行多次 setState，setState 的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时 setState 多个不同的值，在更新时会对其进行合并批量更新

4, React 组件通信如何实现?

React 组件间通信方式:

父组件向子组件通讯: 父组件可以向子组件通过传 props 的方式，向子组件进行通讯

子组件向父组件通讯: props+回调的方式,父组件向子组件传递 props 进行通讯，此 props 为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中

兄弟组件通信: 找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信

跨层级通信: Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言,对于跨越多层的全局数据通过 Context 通信再适合不过

发布订阅模式: 发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入 event 模块进行通信

全局状态管理工具: 借助 Redux 或者 Mobx 等全局状态管理工具进行通信,这种工具会维护一个全局状态中心 Store,并根据不同的事件产生新的状态

5, React 如何进行组件/逻辑复用?

抛开已经被官方弃用的 Mixin,组件抽象的技术目前有三种比较主流:

高阶组件:

属性代理

反向继承

渲染属性

react-hooks

6, mixin、hoc、render props、react-hooks 的优劣如何?

Mixin 的缺陷:

组件与 Mixin 之间存在隐式依赖(Mixin 经常依赖组件的特定方法,但在定义组件时并不知道这种依赖关系)

多个 Mixin 之间可能产生冲突(比如定义了相同的 state 字段)

Mixin 倾向于增加更多状态,这降低了应用的可预测性(The more state in your application, the harder it is to reason about it.),导致复杂度剧增

隐式依赖导致依赖关系不透明,维护成本和理解成本迅速攀升:

难以快速理解组件行为,需要全盘了解所有依赖 Mixin 的扩展行为,及其之间的相互影响

组价自身的方法和 state 字段不敢轻易删改,因为难以确定有没有 Mixin 依赖它

Mixin 也难以维护 ,因为 Mixin 逻辑最后会被打平合并到一起 ,很难搞清楚一个 Mixin 的输入输出

HOC 相比 Mixin 的优势:

HOC 通过外层组件通过 Props 影响内层组件的状态，而不是直接改变其 State 不存在冲突和互相干扰,这就降低了耦合度

不同于 Mixin 的打平+合并，HOC 具有天然的层级结构（组件树结构），这又降低了复杂度

HOC 的缺陷:

扩展性限制： HOC 无法从外部访问子组件的 State 因此无法通过 shouldComponentUpdate 滤掉不必要的更新 ,React 在支持 ES6 Class 之后提供了 React.PureComponent 来解决这个问题

Ref 传递问题: Ref 被隔断,后来的 React.forwardRef 来解决这个问题

Wrapper Hell: HOC 可能出现多层包裹组件的情况,多层抽象同样增加了复杂度和理解成本

命名冲突: 如果高阶组件多次嵌套,没有使用命名空间的话会产生冲突,然后覆盖老属性

不可见性: HOC 相当于在原有组件外层再包装一个组件,你压根不知道外层的包装是啥,对于你是黑盒

Render Props 优点:

上述 HOC 的缺点 Render Props 都可以解决

Render Props 缺陷:

使用繁琐: HOC 使用只需要借助装饰器语法通常一行代码就可以进行复用 ,Render Props 无法做到如此简单

嵌套过深: Render Props 虽然摆脱了组件多层嵌套的问题,但是转化为了函数回调的嵌套

React Hooks 优点:

简洁: React Hooks 解决了 HOC 和 Render Props 的嵌套问题,更加简洁

解耦: React Hooks 可以更方便地把 UI 和状态分离,做到更彻底的解耦

组合: Hooks 中可以引用另外的 Hooks 形成新的 Hooks,组合变化万千

函数友好: React Hooks 为函数组件而生,从而解决了类组件的几大问题:

this 指向容易错误

分割在不同声明周期中的逻辑使得代码难以理解和维护

代码复用成本高（高阶组件容易使代码量剧增）

React Hooks 缺陷:

额外的学习成本（Functional Component 与 Class Component 之间的困惑）

写法上有限制（不能出现在条件、循环中），并且写法限制增加了重构成本

破坏了 PureComponent、React.memo 浅比较的性能优化效果（为了取最新的 props 和 state，每次 render()都要重新创建事件处函数）

在闭包场景可能会引用到旧的 state、props 值

内部实现上不直观（依赖一份可变的全局状态，不再那么“纯”）

React.memo 并不能完全替代 shouldComponentUpdate（因为拿不到 state change，只针对 props change）

7，你是如何理解 fiber 的?

React Fiber 是一种基于浏览器的单线程调度算法.

React 16 之前 ，reconcilation 算法实际上是递归，想要中断递归是很困难的，React 16

开始使用了循环来代替之前的递归.

Fiber：一种将 recocilation（递归 diff）, 拆分成无数个小任务的算法；它随时能够停止，恢复。停止恢复的时机取决于当前的一帧（16ms）内，还有没有足够的时间允许计算。

8, redux 的工作流程?

首先，我们看下几个核心概念：

Store：保存数据的地方，你可以把它看成一个容器，整个应用只能有一个Store。

State：Store 对象包含所有数据，如果想得到某个时点的数据，就要对 Store 生成快照，这种时点的数据集合，就叫做 State。

Action ：State 的变化，会导致 View 的变化。但是，用户接触不到 State，只能接触到 View。所以，State 的变化必须是 View 导致的。Action 就是 View 发出的通知，表示 State 应该要发生变化了。

Action Creator：View 要发送多少种消息，就会有多少种 Action。如果都手写，会很麻烦，所以我们定义一个函数来生成 Action，这个函数就叫 Action Creator。

Reducer：Store 收到 Action 以后，必须给出一个新的 State，这样 View 才会发生变化。这种 State 的计算过程就叫做 Reducer。Reducer 是一个函数，它接受 Action 和当前 State 作为参数，返回一个新的 State。

dispatch：是 View 发出 Action 的唯一方法。

然后我们过下整个工作流程：

首先，用户（通过 View）发出 Action，发出方式就用到了 dispatch 方法。

然后，Store 自动调用 Reducer，并且传入两个参数：当前 State 和收到的 Action，Reducer

会返回新的 State

State 一旦有变化，Store 就会调用监听函数，来更新 View。

到这儿为止，一次用户交互流程结束。可以看到 ,在整个流程中数据都是单向流动的，这种方式保证了流程的清晰。

9, react-redux 是如何工作的?

Provider: Provider 的作用是从最外部封装了整个应用，并向 connect 模块传递 store

connect: 负责连接 React 和 Redux

获取 state: connect 通过 context 获取 Provider 中的 store ,通过 store.getState()获取整个 store tree 上所有 state

包装原组件: 将 state 和 action 通过 props 的方式传入到原组件内部 wrapWithConnect 返回一个 ReactComponent 对象 Connect，Connect 重新 render 外部传入的原组件 WrappedComponent ，并把 connect 中传入的 mapStateToProps, mapDispatchToProps 与组件上原有的 props 合并后，通过属性的方式传给 WrappedComponent

监听 store tree 变化: connect 缓存了 store tree 中 state 的状态,通过当前 state 状态和变更前 state 状态进行比较,从而确定是否调用 this.setState()方法触发 Connect 及其子组件的重新渲染

10, redux 与 mobx 的区别?

两者对比:

redux 将数据保存在单一的 store 中，mobx 将数据保存在分散的多个 store 中

redux 使用 plain object 保存数据，需要手动处理变化后的操作；mobx 适用 observable 保存数据，数据变化后自动处理响应的操作

redux 使用不可变状态，这意味着状态是只读的，不能直接去修改它，而是应该返回一个新的状态，同时使用纯函数；mobx 中的状态是可变的，可以直接对其进行修改

mobx 相对来说比较简单，在其中有很多的抽象，mobx 更多的使用面向对象的编程思维；redux 会比较复杂，因为其中的函数式编程思想掌握起来不是那么容易，同时需要借助一系列的中间件来处理异步和副作用

mobx 中有更多的抽象和封装，调试会比较困难，同时结果也难以预测；而 redux 提供能够进行时间回溯的开发工具，同时其纯函数以及更少的抽象，让调试变得更加的容易

场景辨析:

基于以上区别,我们可以简单得分析一下两者的不同使用场景.

mobx 更适合数据不复杂的应用: mobx 难以调试,很多状态无法回溯,面对复杂度高的应用时,往往力不从心.

redux 适合有回溯需求的应用: 比如一个画板应用、一个表格应用，很多时候需要撤销、重做等操作，由于 redux 不可变的特性，天然支持这些操作.

mobx 适合短平快的项目: mobx 上手简单,样板代码少,可以很大程度上提高开发效率.

当然 mobx 和 redux 也并不一定是非此即彼的关系,你也可以在项目中用 redux 作为全局状态管理,用 mobx 作为组件局部状态管理器来用.

11，redux 中如何进行异步操作?

当然,我们可以在 componentDidmount 中直接进行请求无须借助 redux.

但是在一定规模的项目中,上述方法很难进行异步流的管理,通常情况下我们会借助 redux 的异步中间件进行异步处理.

redux 异步流中间件其实有很多 ,但是当下主流的异步中间件只有两种 redux-thunk、redux-saga，当然 redux-observable 可能也有资格占据一席之地,其余的异步中间件不管是社区活跃度还是 npm 下载量都比较差了.

12, redux 异步中间件之间的优劣?

redux-thunk 优点:

体积小: redux-thunk 的实现方式很简单,只有不到 20 行代码

使用简单: redux-thunk 没有引入像 redux-saga 或者 redux-observable 额外的范式,上手简单

redux-thunk 缺陷:

样板代码过多: 与 redux 本身一样,通常一个请求需要大量的代码,而且很多都是重复性质的

耦合严重: 异步操作与 redux 的 action 偶合在一起,不方便管理

功能孱弱: 有一些实际开发中常用的功能需要自己进行封装

redux-saga 优点:

异步解耦: 异步操作被被转移到单独 saga.js 中，不再是掺杂在 action.js 或 component.js 中

action 摆脱 thunk function: dispatch 的参数依然是一个纯粹的 action (FSA)，而不是充满 “黑魔法” thunk function

异常处理: 受益于 generator function 的 saga 实现，代码异常/请求失败 都可以直接通过 try/catch 语法直接捕获处理

功能强大: redux-saga 提供了大量的 Saga 辅助函数和 Effect 创建器供开发者使用,开发者无须封装或者简单封装即可使用

灵活: redux-saga可以将多个 Saga 可以串行/并行组合起来,形成一个非常实用的异步 flow

易测试，提供了各种 case 的测试方案，包括 mock task，分支覆盖等等

redux-saga 缺陷:

额外的学习成本: redux-saga 不仅在使用难以理解的 generator function,而且有数十个 API,学习成本远超 redux-thunk,最重要的是你的额外学习成本是只服务于这个库的 ,与 redux-observable 不同,redux-observable 虽然也有额外学习成本但是背后是 rxjs 和一整套思想

体积庞大: 体积略大,代码近 2000 行，min 版 25KB 左右

功能过剩: 实际上并发控制等功能很难用到,但是我们依然需要引入这些代码

ts 支持不友好: yield 无法返回 TS 类型

redux-observable 优点:

功能最强: 由于背靠 rxjs 这个强大的响应式编程的库,借助 rxjs 的操作符,你可以几乎做任何你能想到的异步处理

背靠 rxjs: 由于有 rxjs 的加持,如果你已经学习了 rxjs,redux-observable 的学习成本并不高,而且随着 rxjs 的升级 redux-observable 也会变得更强大

redux-observable 缺陷:

学习成本奇高: 如果你不会 rxjs,则需要额外学习两个复杂的库

社区一般: redux-observable 的下载量只有 redux-saga 的 1/5,社区也不够活跃,在复杂异

步流中间件这个层面 redux-saga 仍处于领导地位