第一章 HTML

一、基础

1. DOCTYPE 作用？严格模式与混杂模式是什么？

DOCTYPE 是 HTML5 中一种标准通用标记语言的文档类型声明，它的目的是告诉浏览器（解析器）应该以什么样（html 或 xhtml）的文档类型定义来解析文档，不同的渲染模式会影响浏览器对 CSS 代码甚至 JavaScript 脚本的解析。它必须声明在 HTML文档的第一行。

浏览器渲染页面的两种模式：

CSS1Compat：标准模式（Strick mode），默认模式，浏览器使用 W3C 的标准解析渲染页面。在标准模式中，浏览器以其支持的最高标准呈现页面。

BackCompat：怪异模式(混杂模式)(Quick mode)，浏览器使用自己的怪异模式解析渲染页面。在怪异模式中，页面以一种比较宽松的向后兼容的方式显示。

2. 列出常见的标签，并简单介绍这些标签用在什么场景

行内元素有：a b span u b i s strong；

块级元素有：div ul ol li dl dt dd h1 h2 h3 h4 h5 h6 p；

行内块有：img input select button；

空元素，即没有内容的 HTML 元素。空元素是在开始标签中关闭的，也就是空元素没有闭合标签：

常见的有：
、<hr>、<img>、<input>、<link>、<meta>；。

3. 页面出现了乱码，是怎么回事？如何解决

产生乱码的原因：

网页源代码是 gbk 的编码，而内容中的中文字是 utf-8 编码的，这样浏览器打开即会出现 html 乱码，反之也会出现乱码；

html 网页编码是 gbk，而程序从数据库中调出呈现是 utf-8 编码的内容也会造成编码乱码；

浏览器不能自动检测网页编码，造成网页乱码。

解决办法：

使用软件编辑 HTML 网页内容；

如果网页设置编码是 gbk，而数据库储存数据编码格式是 UTF-8，此时需要程序查询数据库数据显示数据前进程序转码；

如果浏览器浏览时候出现网页乱码，在浏览器中找到转换编码的菜单进行转换

4. title 属性和 alt 属性分别有什么作用

alt 是 img 标签的属性，当图片加载出错、无法显示图像时会显示的替代文本。IE7 以下 IE 浏览器在鼠标滑过时会展示 alt 属性，IE8 以后不会。alt 属性是搜索引擎唯一能识别的图片信息，因此，在优化网站时，尽可能利用 alt 属性阐述图片的主题内容，但是又要避免关键词的重叠堆砌。

title 属性适用于所有标签，规定关于元素的额外信息，通常会在鼠标移到元素上是显示一段提示文本，所有浏览器都支持。

5. HTML5 为什么只写 <!DOCTYPE html> ？文档声明（Doctype）和<!Doctype html>有何作用？严格模式与混杂模式如何区分？它们有何意义

文档声明的作用： 文档声明是为了告诉浏览器，当前 HTML 文档使用什么版本的 HTML 来写的，这样浏览器才

能按照声明的版本来正确的解析。

<!doctype html> 的作用：就是让浏览器进入标准模式，使用最新的 HTML5 标准来解析渲染页面；如果不写，浏览器就会进入混杂模式，我们需要避免此类情况发生。

严格模式与混杂模式的区分：

严格模式： 又称为标准模式，指浏览器按照 W3C 标准解析代码；

混杂模式： 又称怪异模式、兼容模式，是指浏览器用自己的方式解析代码。混杂模式通常模拟老式浏览器的行为，以防止老站点无法工作；

区分：网页中的 DTD，直接影响到使用的是严格模式还是浏览模式，可以说 DTD 的使用与这两种方式的区别息息相关。

如果文档包含严格的 DOCTYPE ，那么它一般以严格模式呈现（严格 DTD ——严格模式）；

包含过渡 DTD 和 URI 的 DOCTYPE ，也以严格模式呈现，但有过渡 DTD 而没有 URI （统一资源标识符，就是声明最后的地址）会导致页面以混杂模式呈现（有 URI 的过渡 DTD ——严格模式；没有 URI 的过渡 DTD ——混杂模式）；

DOCTYPE 不存在或形式不正确会导致文档以混杂模式呈现（DTD 不存在或者格式不正确——混杂模式）；

HTML5 没有 DTD ，因此也就没有严格模式与混杂模式的区别，HTML5 有相对宽松的 法，实现时，已经尽可能大的实现了向后兼容(HTML5 没有严格和混杂之分)。

总之，严格模式让各个浏览器统一执行一套规范兼容模式保证了旧网站的正常运行。

6. 常用的 meta 标签有哪些

（1）charset：<meta charset="UTF-8" >用来描述 HTML 文档的编码类型

（2） keywords：<meta name="keywords" content="关键词" />页面关键词

（3）description：<meta name="description" content="页面描述内容" />页面描述

（4）refresh：<meta http-equiv="refresh" content="0;url=" />页面重定向和刷新

（5）viewport：<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">适配移动端，可以控制视口的大小和比例，其中，content 参数有以下几种：

width viewport ： 宽度(数值/device-width)

height viewport ： 高度(数值/device-height)

initial-scale ： 初始缩放比例

maximum-scale ： 最大缩放比例

minimum-scale ： 最小缩放比例

user-scalable ： 是否允许用户缩放(yes/no)

7. Web 标准以及 W3C 标准是什么

Web 标准不是某一个标准，而是一系列标准的集合：

web 标准简单来说可以分为结构、表现和行为

结构 主要是有 HTML 标签组成

表现 即指 css 样式表

行为 主要是有 js、dom 组成

web 标准一般是将该三部分独立分开，使其更具有模块化。但一般产生行为时，就会有结构或者表现的变化，也使这三者的界限并不那么清晰。

W3C 对于 WEB 标准提出了规范化的要求：

1）标签和属性名字母要小写

2）标签要闭合

3）标签不允许随意嵌套

4）尽量使用外链 css 样式表和 js 脚本。让结构、表现和行为分为三块，符合规范。同时提高页面渲染速度，提高用户的体验。

5）样式尽量少用行间样式表，使结构与表现分离

6）标签的 id 和 class 等属性命名要做到见文知义，更利于 seo，代码便于维护

8. HTML 全局属性（Global Attribute）有哪些

accesskey 规定激活元素的快捷键。

class 规定元素的一个或多个类名（引用样式表中的类）。

contenteditable 规定元素内容是否可编辑。

contextmenu 规定元素的上下文菜单。上下文菜单在用户点击元素时显示。

*data-** 用于存储页面或应用程序的私有定制数据。

dir 规定元素中内容的文本方向。

draggable 规定元素是否可拖动。

dropzone 规定在拖动被拖动数据时是否进行复制、移动或链接。

hidden 规定元素仍未或不再相关。

id 规定元素的唯一 id。

lang 规定元素内容的语言。

spellcheck 规定是否对元素进行拼写和语法检查。

style 规定元素的行内 CSS 样式。

tabindex 规定元素的 tab 键次序。

title 规定有关元素的额外信息。

translate 规定是否应该翻译元素内容。

9. script 标签中 defer 和 async 的区别

如果没有 defer 或 async 属性，浏览器会立即加载并执行相应的脚本。它不会等待后续加载的文档元素，读取到就会开始加载和执行，这样就阻塞了后续文档的加载。

defer 和 async 属性都是去异步加载外部的 JS 脚本文件，它们都不会阻塞页面的解析，其区别如下：

执行顺序： 多个带 async 属性的标签，不能保证加载的顺序；多个带 defer 属性的标签，按照加载顺序执行；

脚本是否并行执行：async 属性，表示后续文档的加载和执行与 js 脚本的加载和执行是并行进行的，即异步执行；defer 属性，加载后续文档的过程和 js 脚本的加载(此时仅加载不执行)是并行进行的(异步)，js 脚本需要等到文档所有元素解析完成之后才执行，DOMContentLoaded 事件触发执行之前。

10. 说一下 web worker

在 HTML 页面中，如果在执行脚本时，页面的状态是不可相应的，直到脚本执行完成后，页面才变成可相应。web worker 是运行在后台的 js，独立于其他脚本，不会影响页面的性能。 并且通过 postMessage 将结果回传到主线程。这样在进行复杂操作的时候，就不会阻塞主线程了。

如何创建 web worker：

检测浏览器对于 web worker 的支持性

创建 web worker 文件（js，回传函数等）

创建 web worker 对象

11. 渐进增强和优雅降级之间的区别

（1）渐进增强 progressive enhancement：主要是针对低版本的浏览器进行页面重构，保证基本的功能情况下，再针对高级浏览器进行效果、交互等方面的改进和追加功能，以达到更好的用户体验。

（2）优雅降级 graceful degradation： 一开始就构建完整的功能，然后再针对低版本的浏览器进行兼容。

两者区别：

优雅降级是从复杂的现状开始的，并试图减少用户体验的供给；而渐进增强是从一个非常基础的，能够起作用的版本开始的，并在此基础上不断扩充，以适应未来环境的需要；

降级（功能衰竭）意味着往回看，而渐进增强则意味着往前看，同时保证其根基处于安全地带。

“优雅降级”观点认为应该针对那些最高级、最完善的浏览器来设计网站。而将那些被认为“过时”或有功能缺失的浏览器下的测试工作安排在开发周期的最后阶段，并把测试对象限定为主流浏览器（如 IE、Mozilla 等）的前一个版本。 在这种设计范例下，旧版的浏览器被认为仅能提供“简陋却无妨 (poor, but passable)” 的浏览体验。可以做一些小的调整来适应某个特定的浏览器。但由于它们并非我们所关注的焦点，因此除了修复较大的错误之外，其它的差异将被直接忽略。

“渐进增强”观点则认为应关注于内容本身。内容是建立网站的诱因，有的网站展示它，有的则收集它，有的寻求，有的操作，还有的网站甚至会包含以上的种种，但相同点是它们全都涉及到内容。这使得“渐进增强”成为一种更为合理的设计范例。这也是它立即被 Yahoo 所采纳并用以构建其“分级式浏览器支持 (Graded Browser Support)”策略的原因所在。

12. head 标签有什么作用，其中什么标签必不可少

用于定义文档的头部，它是所有头部元素的容器。 中的元素可以引用脚本、指示浏览器在哪里找到样式表、提供元信息等。

文档的头部描述了文档的各种属性和信息，包括文档的标题、在 Web 中的位置以及和其他文档的关系等。绝大多数文档头部包含的数据都不会真正作为内容显示给读者。

下面这些标签可用在 head 部分：<base>, <link>, <meta>, <script>, <style>, <title>。

其中 <title> 定义文档的标题，它是 head 部分中唯一必需的元素。

13. src 和 href 的区别

src： 表示对资源的引用，它指向的内容会嵌入到当前标签所在的位置。src 会将其指向的资源下载并应用到文档内，如请求 js 脚本。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，所以一般 js 脚本会放在页面底部。

href： 表示超文本引用，它指向一些网络资源，建立和当前元素或本文档的链接关系。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。 常用在 a、link 等标签上。

14. img 的 srcset 属性的作用

响应式页面中经常用到根据屏幕密度设置不同的图片。这时就用到了 img 标签的 srcset 属性。srcset 属性用于设置不同屏幕密度下，img 会自动加载不同的图片。用法如下：

```
<img src="image-128.png" srcset="image-256.png 2x" />
```

使用上面的代码，就能实现在屏幕密度为 1x 的情况下加载 image-128.png, 屏幕密度为 2x 时加载 image-256.png。

按照上面的实现，不同的屏幕密度都要设置图片地址，目前的屏幕密度有 1x,2x,3x,4x 四种，如果每一个图片都设置 4 张图片，加载就会很慢。所以就有了新的 srcset 标准。代码如下：

```
<img src="image-128.png"
     srcset="image-128.png 128w, image-256.png 256w, image-512.png 512w"
     sizes="(max-width: 360px) 340px, 128px" />
```

其中 srcset 指定图片的地址和对应的图片质量。sizes 用来设置图片的尺寸零界点。对于 srcset 中的 w 单位，可以理解成图片质量。如果可视区域小于这个质量的值，就可以使用。浏览器会自动选择一个最小的可用图片。

sizes 语法如下：

```
sizes="[media query] [length], [media query] [length] ... "
```

sizes 就是指默认显示 128px，如果视区宽度大于 360px，则显示 340px。

15. style 标签写在 body 后与 body 前有什么区别

页面加载自上而下 当然是先加载样式。

写在 body 标签后由于浏览器以逐行方式对 HTML 文档进行解析，当解析到写在尾部的样式表（外联或写在 style 标签）会导致浏览器停止之前的渲染，等待加载且解析样式表完成之后重新渲染，在 windows 的 IE 下可能会出现 FOUC 现象（即样式失效导致的页面闪烁问题）。

二、HTML5

16. HTML5 有哪些更新?

移除了部分元素：

纯表现的元素：basefont，big，center，font, s，strike，tt，u;

对可用性产生负面影响的元素：frame，frameset，noframes；

新增了部分元素：

（1）新增语义化标签：nav、header、footer、aside、section、article

（2）音频、视频标签：audio、video

（3）数据存储：localStorage、sessionStorage

（4）canvas（画布）、Geolocation（地理定位）、websocket（通信协议）

（5）input 标签新增属性：placeholder、autocomplete、autofocus、required

（6）history API：go、forward、back、pushstate

（7）表单

表单类型：

email：能够验证当前输入的邮箱地址是否合法

url： 验证 URL

number： 只能输入数字，其他输入不了，而且自带上下增大减小箭头，max 属性可以设置为最大值，min 可以设置为最小值，value 为默认值。

search： 输入框后面会给提供一个小叉，可以删除输入的内容，更加人性化。

range： 可以提供给一个范围，其中可以设置 max 和 min 以及 value，其中 value 属性可以设置为默认值

color： 提供了一个颜色拾取器

time： 时分秒

data： 日期选择年月日

datatime： 时间和日期(目前只有 Safari 支持)

datatime-local：日期时间控件

week：周控件

month：月控件

表单属性：

placeholder ：提示信息

autofocus ：自动获取焦点

autocomplete=“on” 或者 autocomplete=“off” 使用这个属性需要有两个前提：

表单必须提交过

必须有 name 属性。

required：要求输入框不能为空，必须有值才能够提交。

pattern=" " 里面写入想要的正则模式，例如手机号 patte="^(+86)?\d{10}$"

multiple：可以选择多个文件或者多个邮箱

form=" form 表单的 ID"

表单事件：

oninput 每当 input 里的输入框内容发生变化都会触发此事件。

oninvalid 当验证不通过时触发此事件。

17. HTML5 的离线储存怎么使用，它的工作原理是什么?

离线存储指的是：在用户没有与因特网连接时，可以正常访问站点或应用，在用户与因特网连接时，更新用户机器上的缓存文件。

原理：HTML5 的离线存储是基于一个新建的 .appcache 文件的缓存机制(不是存储技术)，通过这个文件上的解析清单离线存储资源，这些资源就会像 cookie 一样被存储了下来。之后当网络在处于离线状态下时，浏览器会通过被离线存储的数据进行页面展示

使用方法：

（1）创建一个和 html 同名的 manifest 文件，然后在页面头部加入 manifest 属性：

```
<html lang="en" manifest="index.manifest">
```

（2）在 cache.manifest 文件中编写需要离线存储的资源：

```
CACHE MANIFEST
    #v0.11
    CACHE:
    js/app.js
    css/style.css
    NETWORK:
    resourse/logo.png
    FALLBACK:
    / /offline.html
```

CACHE: 表示需要离线存储的资源列表，由于包含 manifest 文件的页面将被自动离线存储，所以不需要把页面自身也列出来。

NETWORK: 表示在它下面列出来的资源只有在在线的情况下才能访问，他们不会被离线存储，所以在离线情况下无法使用这些资源。不过，如果在 CACHE 和 NETWORK 中有一个相同的资源，那么这个资源还是会被离线存储，也就是说 CACHE 的优先级更高。

FALLBACK: 表示如果访问第一个资源失败，那么就使用第二个资源来替换他，比如上面这个文件表示的就是如果访问根目录下任何一个资源失败了，那么就去访问 offline.html 。

（3）在离线状态时，操作 window.applicationCache 进行离线缓存的操作。

如何更新缓存：

（1）更新 manifest 文件

（2）通过 javascript 操作

（3）清除浏览器缓存

注意事项：

（1）浏览器对缓存数据的容量限制可能不太一样（某些浏览器设置的限制是每个站点 5MB）。

（2）如果 manifest 文件，或者内部列举的某一个文件不能正常下载，整个更新过程都将失败，浏览器继续全部使用老的缓存。

（3）引用 manifest 的 html 必须与 manifest 文件同源，在同一个域下。

（4）FALLBACK 中的资源必须和 manifest 文件同源。

（5）当一个资源被缓存后，该浏览器直接请求这个绝对路径也会访问缓存中的资源。

（6）站点中的其他页面即使没有设置 manifest 属性，请求的资源如果在缓存中也从缓存中访问。

（7）当 manifest 文件发生改变时，资源请求本身也会触发更新

18. 浏览器是如何对 HTML5 的离线储存资源进行管理和加载?

在线的情况下，浏览器发现 html 头部有 manifest 属性，它会请求 manifest 文件，如果是第一次访问页面 ，那么浏览器就会根据 manifest 文件的内容下载相应的资源并且进行离线存储。如果已经访问过页面并且资源已经进行离线存储了，那么浏览器就会使用离线的资源加载页面，然后浏览器会对比新的 manifest 文件与旧的 manifest 文件，如果文件没有发生改变，就不做任何操作，如果文件改变了，就会重新下载文件中的资源并进行离线存储。

离线的情况下，浏览器会直接使用离线存储的资源。

19. 说一下 HTML5 drag API?

dragstart：事件主体是被拖放元素，在开始拖放被拖放元素时触发。

darg：事件主体是被拖放元素，在正在拖放被拖放元素时触发。

dragenter：事件主体是目标元素，在被拖放元素进入某元素时触发。

dragover：事件主体是目标元素，在被拖放在某元素内移动时触发。

dragleave：事件主体是目标元素，在被拖放元素移出目标元素是触发。

drop：事件主体是目标元素，在目标元素完全接受被拖放元素时触发。

dragend：事件主体是被拖放元素，在整个拖放操作结束时触发。

20. 对 HTML 语义化的理解，常见的语义化标签有哪些?

语义化是指根据内容的结构化（内容语义化），选择合适的标签（代码语义化）。通俗来讲就是用正确的标签做正确的事情。

语义化的优点如下：

对机器友好，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，有利于 SEO。除此之

外，语义类还支持读屏软件，根据文章可以自动生成目录；

对开发者友好，使用语义类标签增强了可读性，结构更加清晰，开发者能清晰的看出网页的结构，便于团队的开发与维护。

常见的语义化标签：

<header></header> 头部

<nav></nav> 导航栏

<section></section> 区块（有语义化的 div）

<main></main> 主要区域

<article></article> 主要内容

<aside></aside> 侧边栏

<footer></footer> 底部

21. html5 中 data-*属性的作用是什么?

data-* 属性用于存储页面或应用程序的私有自定义数据。

data-* 属性赋予我们在所有 HTML 元素上嵌入自定义 data 属性的能力。

存储的（自定义）数据能够被页面的 JavaScript 中利用，以创建更好的用户体验（不进行 Ajax 调用或服务器端数据库查询）。

data-* 属性包括两部分：

属性名不应该包含任何大写字母，并且在前缀 "data-" 之后必须有至少一个字符

属性值可以是任意字符串

第二章 CSS

参考：https://juejin.cn/post/6844903832552472583

一、基础

1. CSS 选择器及其优先级?

id 选择器	#id	100
类选择器	#classname	10
属性选择器	a[ref=“eee”]	10
伪类选择器	li:last-child	10
标签选择器	div	1
伪元素选择器	li:after	1
相邻兄弟选择器	h1+p	0
子选择器	ul>li	0
后代选择器	li a	0
通配符选择器	*	0

注意事项：

!important 声明的样式的优先级最高；

如果优先级相同，则最后出现的样式生效；

继承得到的样式的优先级最低；

通用选择器（*）、子选择器（>）和相邻同胞选择器（+）并不在这四个等级中，所以它们的权值都为 0 ；

样式表的来源不同时，优先级顺序为：内联样式 > 内部样式 > 外部样式 > 浏览器用户自定义样式 > 浏览器默认样式

2. CSS 中可继承与不可继承属性有哪些

一、无继承性的属性

display：规定元素应该生成的框的类型

文本属性：

vertical-align：垂直文本对齐

text-decoration：规定添加到文本的装饰

text-shadow：文本阴影效果

white-space：空白符的处理

unicode-bidi：设置文本的方向

盒子模型的属性：width、height、margin、border、padding

背景属性：background、background-color、background-image、background-repeat、background-position、background-attachment

定位属性：float、clear、position、top、right、bottom、left、min-width、min-height、max-width、max-height、overflow、clip、z-index

生成内容属性：content、counter-reset、counter-increment

轮廓样式属性：outline-style、outline-width、outline-color、outline

页面样式属性：size、page-break-before、page-break-after

声音样式属性：pause-before、pause-after、pause、cue-before、cue-after、cue、play-during

二、有继承性的属性

字体系列属性

font-family：字体系列

font-weight：字体的粗细

font-size：字体的大小

font-style：字体的风格

文本系列属性

text-indent：文本缩进

text-align：文本水平对齐

line-height：行高

word-spacing：单词之间的间距

letter-spacing：中文或者字母之间的间距

text-transform：控制文本大小写（就是 uppercase、lowercase、capitalize 这三个）

color：文本颜色

元素可见性 visibility：控制元素显示隐藏

列表布局属性 list-style：列表风格，包括 list-style-type、list-style-image 等

光标属性 cursor：光标显示为何种形态

3. 隐藏元素的方法有哪些

display: none：渲染树不会包含该渲染对象，因此该元素不会在页面中占据位置，也不会响应绑定的监听事

件。

visibility: hidden：元素在页面中仍占据空间，但是不会响应绑定的监听事件。

opacity: 0：将元素的透明度设置为 0，以此来实现元素的隐藏。元素在页面中仍然占据空间，并且能够响应元素绑定的监听事件。

position: absolute：通过使用绝对定位将元素移除可视区域内，以此来实现元素的隐藏。

z-index: 负值：来使其他元素遮盖住该元素，以此来实现隐藏。

clip/clip-path ：使用元素裁剪的方法来实现元素的隐藏，这种方法下，元素仍在页面中占据位置，但是不会响应绑定的监听事件。

transform: scale(0,0)：将元素缩放为 0，来实现元素的隐藏。这种方法下，元素仍在页面中占据位置，但是不会响应绑定的监听事件。

4. link 和@import 的区别

伪元素：在内容元素的前后插入额外的元素或样式，但是这些元素实际上并不在文档中生成。它们只在外部显示可见，但不会在文档的源代码中找到它们，因此，称为“伪”元素。

伪类：将特殊的效果添加到特定选择器上。它是已有元素上添加类别的，不会产生新的元素。

5. 伪元素和伪类的区别和作用

两者都是外部引用 CSS 的方式，它们的区别如下：

link 是 XHTML 标签，除了加载 CSS 外，还可以定义 RSS 等其他事务；@import 属于 CSS 范畴，只能加载 CSS。

link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载。

link 是 XHTML 标签，无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持。

link 支持使用 Javascript 控制 DOM 去改变样式；而@import 不支持。

6. li 与 li 之间有看不见的空白间隔是什么原因引起的？如何解决

浏览器会把 inline 内联元素间的空白字符（空格、换行、Tab 等）渲染成一个空格。为了美观，通常是一个<li>放在一行，这导致<li>换行后产生换行字符，它变成一个空格，占用了一个字符的宽度。

解决办法：

（1）为<li>设置 float:left。不足：有些容器是不能设置浮动，如左右切换的焦点图等。

（2）将所有<li>写在同一行。不足：代码不美观。

（3）将<ul>内的字符尺寸直接设为 0，即 font-size:0。不足：<ul>中的其他字符尺寸也被设为 0，需要额外重新设定其他字符尺寸，且在 Safari 浏览器依然会出现空白间隔。

（4）消除<ul>的字符间隔 letter-spacing:-8px，不足：这也设置了<li>内的字符间隔，因此需要将<li>内的字符间隔设为默认 letter-spacing:normal。

7. 常见的图片格式及使用场景

（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 格式的图片通常是较大的文件。

（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。但是 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。除非需要动画的支持，否则没有理由使用 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 是矢量图意味着 SVG 图片由直线和曲线以及绘制它们的方法组成。当放大 SVG 图片时，看到的还是线和曲线，而不会出现像素点。SVG 图片在放大时，不会失真，所以它适合用来绘制 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。从名字就可以看出来它是为 Web 而生的，什么叫为 Web 而生呢？就是说相同质量的图片，WebP 具有更小的文件体积。现在网站上充满了大量的图片，如果能够降低每一个图片的文件大小，那么将大大减少浏览器和服务器之间的数据传输量，进而降低访问延迟，提升访问体验。目前只有 Chrome 浏览器和 Opera 浏览器支持 WebP 格式，兼容性不太好。

在无损压缩的情况下，相同质量的 WebP 图片，文件大小要比 PNG 小 26%；

在有损压缩的情况下，具有相同图片精度的 WebP 图片，文件大小要比 JPEG 小 25%~34%；

WebP 图片格式支持图片透明度，一个无损压缩的 WebP 图片，如果要支持透明度只需要 22%的格外文件大小。

8. 对 CSS Sprites 的理解

CSSSprites（精灵图），将一个页面涉及到的所有图片都包含到一张大图中去，然后利用 CSS 的 background-image，background-repeat，background-position 属性的组合进行背景定位。

优点：利用 CSS Sprites 能很好地减少网页的 http 请求，从而大大提高了页面的性能，这是 CSS Sprites 最大的优点；CSS Sprites 能减少图片的字节，把 3 张图片合并成 1 张图片的字节总是小于这 3 张图片的字节总和。

缺点：在图片合并时，要把多张图片有序的、合理的合并成一张图片，还要留好足够的空间，防止板块内出现不必要的背景。在宽屏及高分辨率下的自适应页面，如果背景不够宽，很容易出现背景断裂；CSSSprites 在开发的时候相对来说有点麻烦，需要借助 photoshop 或其他工具来对每个背景单元测量其准确的位置。

维护方面：CSS Sprites 在维护的时候比较麻烦，页面背景有少许改动时，就要改这张合并的图片，无需改的地方尽量不要动，这样避免改动更多的 CSS，如果在原来的地方放不下，又只能（最好）往下加图片，这样图片的字节就增加了，还要改动 CSS。

9. 什么是物理像素，逻辑像素和像素密度，为什么在移动端开发时需要用到@3x, @2x 这种图片

以 iPhone XS 为例，当写 CSS 代码时，针对于单位 px，其宽度为 414px＆896px，也就是说当赋予一个 DIV 元素宽度为 414px，这个 DIV 就会填满手机的宽度；

而如果有一把尺子来实际测量这部手机的物理像素，实际为 1242*2688 物理像素；经过计算可知，1242/414=3，也就是说，在单边上，一个逻辑像素=3 个物理像素，就说这个屏幕的像素密度为 3，也就是常说的 3 倍屏。

对于图片来说，为了保证其不失真，1 个图片像素至少要对应一个物理像素，假如原始图片是 500300 像素，那么在 3 倍屏上就要放一个 1500900 像素的图片才能保证 1 个物理像素至少对应一个图片像素，才能不失真。

10. z-index 属性在什么情况下会失效

通常 z-index 的使用是在有两个重叠的标签，在一定的情况下控制其中一个在另一个的上方或者下方出现。

z-index 值越大就越是在上层。z-index 元素的 position 属性需要是 relative，absolute 或是 fixed。

z-index 属性在下列情况下会失效：

父元素 position 为 relative 时，子元素的 z-index 失效。解决：父元素 position 改为 absolute 或 static；

元素没有设置 position 属性为非 static 属性。解决：设置该元素的 position 属性为 relative，absolute 或是 fixed 中的一种；

元素在设置 z-index 的同时还设置了 float 浮动。解决：float 去除，改为 display：inline-block；

11. 对媒体查询的理解

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，适合 web网页应对不同型号的设备而做出对应的响应适配。

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。

简单来说，使用 @media 查询，可以针对不同的媒体类型定义不同的样式。@media 可以针对不同的屏幕尺寸设置不同的样式，特别是需要设置设计响应式的页面，@media 是非常有用的。当重置浏览器大小的过程中，页面也会根据浏览器的宽度和高度重新渲染页面。

12. 对 CSS 工程化的理解

CSS 工程化是为了解决以下问题：

宏观设计：CSS 代码如何组织、如何拆分、模块结构怎样设计？

编码优化：怎样写出更好的 CSS？

构建：如何处理我的 CSS，才能让它的打包结果最优？

可维护性：代码写完了，如何最小化它后续的变更成本？如何确保任何一个同事都能轻松接手？

以下三个方向都是时下比较流行的、普适性非常好的 CSS 工程化实践：

预处理器：Less、 Sass 等；

重要的工程化插件： PostCss；

Webpack loader 等；

基于这三个方向，可以衍生出一些具有典型意义的子问题，这里我们逐个来看：

（1）预处理器：为什么要用预处理器？它的出现是为了解决什么问题？

预处理器，其实就是 CSS 世界的“轮子”。预处理器支持我们写一种类似 CSS、但实际并不是 CSS 的语言，然后把它编译成 CSS 代码：

那为什么写 CSS 代码写得好好的，偏偏要转去写“类 CSS”呢？这就和本来用 JS 也可以实现所有功能，但最后却写 React 的 jsx 或者 Vue 的模板语法一样——为了爽！要想知道有了预处理器有多爽，首先要知道的是传统 CSS 有多不爽。随着前端业务复杂度的提高，前端工程中对 CSS 提出了以下的诉求：

宏观设计上：我们希望能优化 CSS 文件的目录结构，对现有的 CSS 文件实现复用；

编码优化上：我们希望能写出结构清晰、简明易懂的 CSS，需要它具有一目了然的嵌套层级关系，而不是无差别的一铺到底写法；我们希望它具有变量特征、计算能力、循环能力等等更强的可编程性，这样我们可以少写一些无用的代码；

可维护性上：更强的可编程性意味着更优质的代码结构，实现复用意味着更简单的目录结构和更强的拓展能

力，这两点如果能做到，自然会带来更强的可维护性。

这三点是传统 CSS 所做不到的，也正是预处理器所解决掉的问题。预处理器普遍会具备这样的特性：

嵌套代码的能力，通过嵌套来反映不同 css 属性之间的层级关系 ；

支持定义 css 变量；

提供计算函数；

允许对代码片段进行 extend 和 mixin；

支持循环语句的使用；

支持将 CSS 文件模块化，实现复用。

（2）PostCss：PostCss 是如何工作的？我们在什么场景下会使用 PostCss？

它和预处理器的不同就在于，预处理器处理的是 类 CSS，而 PostCss 处理的就是 CSS 本身。Babel 可以将高版本的 JS 代码转换为低版本的 JS 代码。PostCss 做的是类似的事情：它可以编译尚未被浏览器广泛支持的先进的 CSS 语法，还可以自动为一些需要额外兼容的语法增加前缀。更强的是，由于 PostCss 有着强大的插件机制，支持各种各样的扩展，极大地强化了 CSS 的能力。

PostCss 在业务中的使用场景非常多：

提高 CSS 代码的可读性：PostCss 其实可以做类似预处理器能做的工作；

当我们的 CSS 代码需要适配低版本浏览器时,PostCss 的 Autoprefixer 插件可以帮助我们自动增加浏览器前缀；

允许我们编写面向未来的 CSS：PostCss 能够帮助我们编译 CSS next 代码；

（3）Webpack 能处理 CSS 吗？如何实现？

Webpack 能处理 CSS 吗：

Webpack 在裸奔的状态下，是不能处理 CSS 的，Webpack 本身是一个面向 JavaScript 且只能处理 JavaScript 代码的模块化打包工具；

Webpack 在 loader 的辅助下，是可以处理 CSS 的。

如何用 Webpack 实现对 CSS 的处理：

Webpack 中操作 CSS 需要使用的两个关键的 loader：css-loader 和 style-loader

注意，答出“用什么”有时候可能还不够，面试官会怀疑你是不是在背答案，所以你还需要了解每个 loader 都做了什么事情：

css-loader：导入 CSS 模块，对 CSS 代码进行编译处理；

style-loader：创建 style 标签，把 CSS 内容写入标签。

在实际使用中，css-loader 的执行顺序一定要安排在 style-loader 的前面。因为只有完成了编译过程，才可以对 css 代码进行插入；若提前插入了未编译的代码，那么 webpack 是无法理解这坨东西的，它会无情报错。

13. 一个满屏品字布局如何设计

第一种真正的品字：

三块高宽是确定的；

上面那块用 margin: 0 auto;居中；

下面两块用 float 或者 inline-block 不换行；

用 margin 调整位置使他们居中。

第二种全屏的品字布局:

上面的 div 设置成 100%，下面的 div 分别宽 50%，然后使用 float 或者 inline 使其不换行。

14. 对 BFC 规范(块级格式化上下文：block formatting context)的理解

BFC 规定了内部的 Block Box 如何布局。

定位方案：

内部的 Box 会在垂直方向上一个接一个放置。

Box 垂直方向的距离由 margin 决定，属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠。

每个元素的 margin box 的左边，与包含块 border box 的左边相接触。

BFC 的区域不会与 float box 重叠。

BFC 是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。

计算 BFC 的高度时，浮动元素也会参与计算。

满足下列条件之一就可触发 BFC

根元素，即 html

float 的值不为 none（默认）

overflow 的值不为 visible（默认）

display 的值为 inline-block、table-cell、table-caption

position 的值为 absolute 或 fixed

15. 浏览器是怎样解析 CSS 选择器的

CSS 选择器的解析是从右向左解析的。若从左向右的匹配，发现不符合规则，需要进行回溯，会损失很多性能。若从右向左匹配，先找到所有的最右节点，对于每一个节点，向上寻找其父节点直到找到根元素或满足条件的匹配规则，则结束这个分支的遍历。两种匹配规则的性能差别很大，是因为从右向左的匹配在第一步就筛选掉了大量的不符合条件的最右节点（叶子节点），而从左向右的匹配规则的性能都浪费在了失败的查找上面。

而在 CSS 解析完毕后，需要将解析的结果与 DOM Tree 的内容一起进行分析建立一棵 Render Tree，最终用来进行绘图。在建立 Render Tree 时（WebKit 中的「Attachment」过程），浏览器就要为每个 DOM Tree 中的元素根据 CSS 的解析结果（Style Rules）来确定生成怎样的 Render Tree。

16. CSS 优化、提高性能的方法有哪些

避免过度约束

避免后代选择符

避免链式选择符

使用紧凑的语法

避免不必要的命名空间

避免不必要的重复

最好使用表示语义的名字。一个好的类名应该是描述他是什么而不是像什么

避免！important，可以选择其他选择器

尽可能的精简规则，你可以合并不同类里的重复规则。

17. 在网页中的应该使用奇数还是偶数的字体？为什么呢

p{font-size:10px;-webkit-transform:scale(0.8);} //0.8 是缩放比例。

18. 怎么让 Chrome 支持小于 12px 的文字

使用偶数字体。偶数字号相对更容易和 web 设计的其他部分构成比例关系。Windows 自带的点阵宋体（中易宋体）从 Vista 开始只提供 12、14、16 px 这三个大小的点阵，而 13、15、17 px 时用的是小一号的点。（即每个字占的空间大了 1 px，但点阵没变），于是略显稀疏。

19. Sass、Less 是什么？为什么要使用他们

他们都是 CSS 预处理器，是 CSS 上的一种抽象层。他们是一种特殊的语法/语言编译成 CSS。例如 Less 是一种动态样式语言，将 CSS 赋予了动态语言的特性，如变量，继承，运算， 函数，LESS 既可以在客户端上运行 (支持 IE 6+, Webkit, Firefox)，也可以在服务端运行 (借助 Node.js)。

为什么要使用它们？

结构清晰，便于扩展。 可以方便地屏蔽浏览器私有语法差异。封装对浏览器语法差异的重复处理， 减少无意义的机械劳动。

可以轻松实现多重继承。 完全兼容 CSS 代码，可以方便地应用到老项目中。LESS 只是在 CSS 语法上做了扩展，所以老的 CSS 代码也可以与 LESS 代码一同编译。

20. 什么是物理像素，逻辑像素和像素密度，为什么在移动端开发时需要用到@3x, @2x 这种图片

以 iPhone XS 为例，当写 CSS 代码时，针对于单位 px，其宽度为 414px＆896px，也就是说当赋予一个 DIV 元素宽度为 414px，这个 DIV 就会填满手机的宽度；

而如果有一把尺子来实际测量这部手机的物理像素，实际为 1242*2688 物理像素；经过计算可知，1242/414=3，也就是说，在单边上，一个逻辑像素=3 个物理像素，就说这个屏幕的像素密度为 3，也就是常说的 3 倍屏。

对于图片来说，为了保证其不失真，1 个图片像素至少要对应一个物理像素，假如原始图片是 500300 像素，那么在 3 倍屏上就要放一个 1500900 像素的图片才能保证 1 个物理像素至少对应一个图片像素，才能不失真。

当然，也可以针对所有屏幕，都只提供最高清图片。虽然低密度屏幕用不到那么多图片像素，而且会因为下载多余的像素造成带宽浪费和下载延迟，但从结果上说能保证图片在所有屏幕上都不会失真。

21. 替换元素的概念及计算规则

通过修改某个属性值呈现的内容就可以被替换的元素就称为“替换元素”。

替换元素除了内容可替换这一特性以外，还有以下特性：

内容的外观不受页面上的 CSS 的影响：用专业的话讲就是在样式表现在 CSS 作用域之外。如何更改替换元素本身的外观需要类似 appearance 属性，或者浏览器自身暴露的一些样式接口。

有自己的尺寸：在 Web 中，很多替换元素在没有明确尺寸设定的情况下，其默认的尺寸（不包括边框）是 300 像素×150 像素，如

在很多 CSS 属性上有自己的一套表现规则：比较具有代表性的就是 vertical-align 属性，对于替换元素和非替换元素，vertical-align 属性值的解释是不一样的。比方说 vertical-align 的默认值的 baseline，很简单的属性值，基线之意，被定义为字符 x 的下边缘，而替换元素的基线却被硬生生定义成了元素的下边缘。

所有的替换元素都是内联水平元素：也就是替换元素和替换元素、替换元素和文字都是可以在一行显示的。但是，替换元素默认的 display 值却是不一样的，有的是 inline，有的是 inline-block。

替换元素的尺寸从内而外分为三类：

固有尺寸： 指的是替换内容原本的尺寸。例如，图片、视频作为一个独立文件存在的时候，都是有着自己的宽度和高度的。

HTML 尺寸： 只能通过 HTML 原生属性改变，这些 HTML 原生属性包括的 width 和 height 属性、的 size 属性。

CSS 尺寸： 特指可以通过 CSS 的 width 和 height 或者 max-width/min-width 和 max-height/min-height 设置的尺寸，对应盒尺寸中的 content box。

这三层结构的计算规则具体如下：

（1）如果没有 CSS 尺寸和 HTML 尺寸，则使用固有尺寸作为最终的宽高。

（2）如果没有 CSS 尺寸，则使用 HTML 尺寸作为最终的宽高。

（3）如果有 CSS 尺寸，则最终尺寸由 CSS 属性决定。

（4）如果“固有尺寸”含有固有的宽高比例，同时仅设置了宽度或仅设置了高度，则元素依然按照固有的宽高比例显示。

（5）如果上面的条件都不符合，则最终宽度表现为 300 像素，高度为 150 像素。

（6）内联替换元素和块级替换元素使用上面同一套尺寸计算规则。

22. 常见的图片格式及使用场景

（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 格式的图片通常是较大的文件。

（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。但是 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。除非需要动画的支持，否则没有理由使用 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 是矢量图意味着 SVG 图片由直线和曲线以及绘制它们的方法组成。当放大 SVG 图片时，看到的还是线和曲线，而不会出现像素点。SVG 图片在放大时，不会失真，所以它适合用来绘制 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。从名字就可以看出来它是为 Web 而生的，什么叫为 Web 而生呢？就是说相同质量的图片，WebP 具有更小的文件体积。现在网站上充满了大量的图片，如果能够降低每一个图片的文件大小，那么将大大减少浏览器和服务器之间的数据传输量，进而降低访问延迟，提升访问体验。目前只有 Chrome 浏览器和 Opera 浏览器支持 WebP 格式，兼容性不太好。

在无损压缩的情况下，相同质量的 WebP 图片，文件大小要比 PNG 小 26%；

在有损压缩的情况下，具有相同图片精度的 WebP 图片，文件大小要比 JPEG 小 25%~34%；

WebP 图片格式支持图片透明度，一个无损压缩的 WebP 图片，如果要支持透明度只需要 22%的格外文件大小。

23. display:inline-block 什么时候会显示间隙

有空格时候会有间隙 解决：移除空格

margin 正值的时候 解决：margin 使用负值

使用 font-size 时候 解决：font-size:0、letter-spacing、word-spacing

二、CSS3

参考：https://juejin.cn/post/6844903518520901639

https://hieeyh.github.io/2017/07/06/css3-of-interview/

24. CSS3 中有哪些新特性?

新增各种 CSS 选择器 （: not(.input)：所有 class 不是“input”的节点）

圆角 （border-radius:8px）

多列布局 （multi-column layout）

阴影和反射 （Shadoweflect）

文字特效 （text-shadow）

文字渲染 （Text-decoration）

线性渐变 （gradient）

旋转 （transform）

增加了旋转,缩放,定位,倾斜,动画,多背景

25. CSS3 新增伪类有那些?

p:first-of-type 选择属于其父元素的首个元素

p:last-of-type 选择属于其父元素的最后元素

p:only-of-type 选择属于其父元素唯一的元素

p:only-child 选择属于其父元素的唯一子元素

p:nth-child(2) 选择属于其父元素的第二个子元素

:enabled :disabled 表单控件的禁用状态。

:checked 单选框或复选框被选中。

26. :before 和 :after 中双冒号和单冒号有什么区别？解释一下这 2 个伪元素的作用?

单冒号(:)用于 CSS3 伪类，双冒号(::)用于 CSS3 伪元素。

::before 就是以一个子元素的存在，定义在元素主体内容之前的一个伪元素。并不存在于 dom 之中，只存在在页面之中。

:before 和 :after 这两个伪元素，是在 CSS2.1 里新出现的。起初，伪元素的前缀使用的是单冒号语法，但随着 Web 的进化，在 CSS3 的规范里，伪元素的语法被修改成使用双冒号，成为::before ::after

27. 伪元素和伪类的区别和作用?

伪元素：在内容元素的前后插入额外的元素或样式，但是这些元素实际上并不在文档中生成。它们只在外部显示可见，但不会在文档的源代码中找到它们，因此，称为“伪”元素。

伪类：将特殊的效果添加到特定选择器上。它是已有元素上添加类别的，不会产生新的元素。

总结： 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。

三、布局

28. 常见的 CSS 布局单位?

常用的布局单位包括像素（px），百分比（%），em，rem，vw/vh。

（1）像素（px）是页面布局的基础，一个像素表示终端（电脑、手机、平板等）屏幕所能显示的最小的区域，像素分为两种类型：CSS 像素和物理像素：

CSS 像素：为 web 开发者提供，在 CSS 中使用的一个抽象单位；

物理像素：只与设备的硬件密度有关，任何设备的物理像素都是固定的。

（2）百分比（%），当浏览器的宽度或者高度发生变化时，通过百分比单位可以使得浏览器中的组件的宽和高随着浏览器的变化而变化，从而实现响应式的效果。一般认为子元素的百分比相对于直接父元素。

（3）em 和 rem 相对于 px 更具灵活性，它们都是相对长度单位，它们之间的区别：em 相对于父元素，rem 相对于根元素。

em： 文本相对长度单位。相对于当前对象内文本的字体尺寸。如果当前行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸(默认 16px)。(相对父元素的字体大小倍数)。

rem： rem 是 CSS3 新增的一个相对单位，相对于根元素（html 元素）的 font-size 的倍数。作用：利用 rem 可以实现简单的响应式布局，可以利用 html 元素中字体的大小与屏幕间的比值来设置 font-size 的值，以此实现当屏幕分辨率变化时让元素也随之变化。

（4）vw/vh 是与视图窗口有关的单位，vw 表示相对于视图窗口的宽度，vh 表示相对于视图窗口高度，除了 vw 和 vh 外，还有 vmin 和 vmax 两个相关的单位。

vw：相对于视窗的宽度，视窗宽度是 100vw；

vh：相对于视窗的高度，视窗高度是 100vh；

vmin：vw 和 vh 中的较小值；

vmax：vw 和 vh 中的较大值；

vw/vh 和百分比很类似，两者的区别：

百分比（%）：大部分相对于祖先元素，也有相对于自身的情况比如（border-radius、translate 等)

vw/vm：相对于视窗的尺寸。

29. px、em、rem 的区别及使用场景?

三者的区别：

px 是固定的像素，一旦设置了就无法因为适应页面大小而改变。

em 和 rem 相对于 px 更具有灵活性，他们是相对长度单位，其长度不是固定的，更适用于响应式布局。

em 是相对于其父元素来设置字体大小，这样就会存在一个问题，进行任何元素设置，都有可能需要知道他父元素的大小。而 rem 是相对于根元素，这样就意味着，只需要在根元素确定一个参考值。

使用场景：

对于只需要适配少部分移动设备，且分辨率对页面影响不大的，使用 px 即可 。

对于需要适配各种移动设备，使用 rem，例如需要适配 iPhone 和 iPad 等分辨率差别比较挺大的设备。

30. 如何根据设计稿进行移动端适配?

移动端适配主要有两个维度：

适配不同像素密度， 针对不同的像素密度，使用 CSS 媒体查询，选择不同精度的图片，以保证图片不会失真；

适配不同屏幕大小， 由于不同的屏幕有着不同的逻辑像素大小，所以如果直接使用 px 作为开发单位，会使得开发的页面在某一款手机上可以准确显示，但是在另一款手机上就会失真。为了适配不同屏幕的大小，应按照比例来还原设计稿的内容。

为了能让页面的尺寸自适应，可以使用 rem，em，vw，vh 等相对单位。

31. 响应式设计的概念及基本原理?

在实际项目中，我们可能需要综合上面的方案，比如用 rem 来做字体的适配，用 srcset 来做图片的响应式，宽度可以用 rem，flex，栅格系统等来实现响应式，然后可能还需要利用媒体查询来作为响应式布局的基础，因此综合上面的实现方案，项目中实现响应式布局需要注意下面几点：

设置 viewport
媒体查询
字体的适配（字体单位）
百分比布局
图片的适配（图片的响应式）
结合 flex，grid，BFC，第三方 UI 库等已经成型的栅格系统方案。

32. 对 Flex 布局的理解及其使用场景?

Flex 是 FlexibleBox 的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。任何一个容器都可以指定为 Flex 布局。行内元素也可以使用 Flex 布局。注意，设为 Flex 布局以后，子元素的 float、clear 和 vertical-align 属性将失效。采用 Flex 布局的元素，称为 Flex 容器（flex container），简称"容器"。它的所有子元素自动成为容器成员，称为 Flex 项目（flex item），简称"项目"。容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis），项目默认沿水平主轴排列。

以下 6 个属性设置在容器上：

flex-direction 属性决定主轴的方向（即项目的排列方向）。
flex-wrap 属性定义，如果一条轴线排不下，如何换行。
flex-flow 属性是 flex-direction 属性和 flex-wrap 属性的简写形式，默认值为 row nowrap。
justify-content 属性定义了项目在主轴上的对齐方式。
align-items 属性定义项目在交叉轴上如何对齐。
align-content 属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用。

以下 6 个属性设置在项目上：

order 属性定义项目的排列顺序。数值越小，排列越靠前，默认为 0。
flex-grow 属性定义项目的放大比例，默认为 0，即如果存在剩余空间，也不放大。
flex-shrink 属性定义了项目的缩小比例，默认为 1，即如果空间不足，该项目将缩小。
flex-basis 属性定义了在分配多余空间之前，项目占据的主轴空间。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为 auto，即项目的本来大小。
flex 属性是 flex-grow，flex-shrink 和 flex-basis 的简写，默认值为 0 1 auto。
align-self 属性允许单个项目有与其他项目不一样的对齐方式，可覆盖 align-items 属性。默认值为 auto，表示继承父元素的 align-items 属性，如果没有父元素，则等同于 stretch。

简单来说：

flex 布局是 CSS3 新增的一种布局方式，可以通过将一个元素的 display 属性值设置为 flex 从而使它成为一个 flex 容器，它的所有子元素都会成为它的项目。一个容器默认有两条轴：一个是水平的主轴，一个是与主轴垂直的交叉轴。可以使用 flex-direction 来指定主轴的方向。可以使用 justify-content 来指定元素在主轴上的排列方式，使用 align-items 来指定元素在交叉轴上的排列方式。还可以使用 flex-wrap 来规定当一行排列不下时的换行方式。对于容器中的项目，可以使用 order 属性来指定项目的排列顺序，还可以使用 flex-grow 来指定当排列空间有剩余的时候，项目的放大比例，还可以使用 flex-shrink 来指定当排列空间不足时，项目的缩小比例。

33. Grid 布局?

http://alloween.top/2018/03/18/CSS%E6%A0%85%E6%A0%BC%E7%B3%BB%E7%BB%9F/
最强大的 CSS 布局 — Grid 布局 https://juejin.cn/post/6854573220306255880
CSS Grid 网格布局 https://juejin.cn/post/6992510224842096677#heading-1
未来布局之星 Gridhttps://juejin.cn/post/6844903497603760135
[CSS] 栅格化布局 https://juejin.cn/post/6886367930523746312#heading-0

34. grid 和 flex 区别是什么？适用什么场景?

Grid 布局与 Flex 布局有一定的相似性，都可以指定容器内部多个项目的位置。但是，它们也存在重大区别。

Flexbox 是一维布局系统，适合做局部布局，比如导航栏组件。

Grid 是二维布局系统，通常用于整个页面的规划。

二者从应用场景来说并不冲突。虽然 Flexbox 也可以用于大的页面布局，但是没有 Grid 强大和灵活。二者结合使用更加轻松。

四、定位 &&浮动

35. 为什么需要清除浮动？清除浮动的方式？

浮动的定义： 非 IE 浏览器下，容器不设高度且子元素浮动时，容器高度不能被内容撑开。 此时，内容会溢出到容器外面而影响布局。这种现象被称为浮动（溢出）。

浮动的工作原理：

浮动元素脱离文档流，不占据空间（引起“高度塌陷”现象）

浮动元素碰到包含它的边框或者其他浮动元素的边框停留

浮动元素可以左右移动，直到遇到另一个浮动元素或者遇到它外边缘的包含框。浮动框不属于文档流中的普通流，当元素浮动之后，不会影响块级元素的布局，只会影响内联元素布局。此时文档流中的普通流就会表现得该浮动框不存在一样的布局模式。当包含框的高度小于浮动框的时候，此时就会出现“高度塌陷”。

浮动元素引起的问题？

父元素的高度无法被撑开，影响与父元素同级的元素

与浮动元素同级的非浮动元素会跟随其后

若浮动的元素不是第一个元素，则该元素之前的元素也要浮动，否则会影响页面的显示结构

清除浮动的方式如下：

给父级 div 定义 height 属性

最后一个浮动元素之后添加一个空的 div 标签，并添加 clear:both 样式

包含浮动元素的父级标签添加 overflow:hidden 或者 overflow:auto

使用 :after 伪元素。由于 IE6-7 不支持 :after，使用 zoom:1 触发 hasLayout**

```
.clearfix:after{
    content: "\200B";
    display: table;
    height: 0;
    clear: both;
  }
  .clearfix{
    *zoom: 1;
  }
```

36. 使用 clear 属性清除浮动的原理？

使用 clear 属性清除浮动，其语法如下：

```
.clear:none|left|right|both
```

如果单看字面意思，clear:left 是“清除左浮动”，clear:right 是“清除右浮动”，实际上，这种解释是有问题的，因为浮动一直还在，并没有清除。

官方对 clear 属性解释：“元素盒子的边不能和前面的浮动元素相邻”，对元素设置 clear 属性是为了避免浮动元素对该元素的影响，而不是清除掉浮动。

还需要注意 clear 属性指的是元素盒子的边不能和前面的浮动元素相邻，注意这里“前面的”3 个字，也就是 clear 属性对“后面的”浮动元素是不闻不问的。考虑到 float 属性要么是 left，要么是 right，不可能同时存在，同时由于 clear 属性对“后面的”浮动元素不闻不问，因此，当 clear:left 有效的时候，clear:right 必定无效，也就是此时 clear:left 等同于设置 clear:both；同样地，clear:right 如果有效也是等同于设置 clear:both。由此可见，clear:left 和 clear:right 这两个声明就没有任何使用的价值，至少在 CSS 世界中是如此，直接使用 clear:both 吧。

一般使用伪元素的方式清除浮动：

```
.clear::after{  content:'';  display: block;   clear:both;}
```

clear 属性只有块级元素才有效的，而::after 等伪元素默认都是内联水平，这就是借助伪元素清除浮动影响时需要设置 display 属性值的原因。

37. 对 BFC 的理解，如何创建 BFC?

先来看两个相关的概念：

Box: Box 是 CSS 布局的对象和基本单位，一个页面是由很多个 Box 组成的，这个 Box 就是我们所说的盒模型。

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。

块格式化上下文（Block Formatting Context，BFC）是 Web 页面的可视化 CSS 渲染的一部分，是布局过程中生成块级盒子的区域，也是浮动元素与其他元素的交互限定区域。

通俗来讲：BFC 是一个独立的布局环境，可以理解为一个容器，在这个容器中按照一定规则进行物品摆放，并且不会影响其它环境中的物品。如果一个元素符合触发 BFC 的条件，则 BFC 中的元素布局不受外部影响。

创建 BFC 的条件：

根元素：body；

元素设置浮动：float 除 none 以外的值；

元素设置绝对定位：position (absolute、fixed)；

display 值为：inline-block、table-cell、table-caption、flex 等；

overflow 值为：hidden、auto、scroll；

BFC 的特点：

垂直方向上，自上而下排列，和文档流的排列方式一致。

在 BFC 中上下相邻的两个容器的 margin 会重叠

计算 BFC 的高度时，需要计算浮动元素的高度

BFC 区域不会与浮动的容器发生重叠

BFC 是独立的容器，容器内部元素不会影响外部元素

每个元素的左 margin 值和容器的左 border 相接触

BFC 的作用：

解决 margin 的重叠问题：由于 BFC 是一个独立的区域，内部的元素和外部的元素互不影响，将两个元素变为两个 BFC，就解决了 margin 重叠的问题。

解决高度塌陷的问题：在对子元素设置浮动后，父元素会发生高度塌陷，也就是父元素的高度变为 0。解决这个问题，只需要把父元素变成一个 BFC。常用的办法是给父元素设置 overflow:hidden。

创建自适应两栏布局：可以用来创建自适应两栏布局：左边的宽度固定，右边的宽度自适应。

```
.left{
     width: 100px;
     height: 200px;
     background: red;
     float: left;
 }
 .right{
     height: 300px;
     background: blue;
     overflow: hidden;
 }

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

左侧设置 float:left，右侧设置 overflow: hidden。这样右边就触发了 BFC，BFC 的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠，实现了自适应两栏布局。

38. 什么是 margin 重叠问题？如何解决？

问题描述：

两个块级元素的上外边距和下外边距可能会合并（折叠）为一个外边距，其大小会取其中外边距值大的那个，这种行为就是外边距折叠。需要注意的是，浮动的元素和绝对定位这种脱离文档流的元素的外边距不会折叠。重叠只会出现在垂直方向。

计算原则（折叠合并后外边距的计算原则）如下：

如果两者都是正数，那么就去最大者

如果是一正一负，就会正值减去负值的绝对值

两个都是负值时，用 0 减去两个中绝对值大的那个

解决办法：

对于折叠的情况，主要有两种：兄弟之间重叠和父子之间重叠

（1）兄弟之间重叠

底部元素变为行内盒子：display: inline-block

底部元素设置浮动：float

底部元素的 position 的值为 absolute/fixed

（2）父子之间重叠

父元素加入：overflow: hidden

父元素添加透明边框：border:1px solid transparent

子元素变为行内盒子：display: inline-block

子元素加入浮动属性或定位

39. 元素的层叠顺序?

（1）背景和边框：建立当前层叠上下文元素的背景和边框。

（2）负的 z-index：当前层叠上下文中，z-index 属性值为负的元素。

（3）块级盒：文档流内非行内级非定位后代元素。

（4）浮动盒：非定位浮动元素。

（5）行内盒：文档流内行内级非定位后代元素。

（6）z-index:0：层叠级数为 0 的定位元素。

（7）正 z-index：z-index 属性值为正的定位元素。

注意: 当定位元素 z-index:auto，生成盒在当前层叠上下文中的层级为 0，不会建立新的层叠上下文，除非是根元素。

40. position 的属性有哪些，区别是什么?

absolute 生成绝对定位的元素，相对于 static 定位以外的一个父元素进行定位。元素的位置通过 left、top、right、bottom 属性进行规定。

relative 生成相对定位的元素，相对于其原来的位置进行定位。元素的位置通过 left、top、right、bottom 属性进行规定。

fixed 生成绝对定位的元素，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，比如回到顶部的按钮一般都是用此定位方式。

static 默认值，没有定位，元素出现在正常的文档流中，会忽略 top, bottom, left, right 或者 z-index 声明，块级元素从上往下纵向排布，行级元素从左向右排列。

inherit 规定从父元素继承 position 属性的值。

41. display、float、position 的关系?

（1）首先判断 display 属性是否为 none，如果为 none，则 position 和 float 属性的值不影响元素最后的表现。

（2）然后判断 position 的值是否为 absolute 或者 fixed，如果是，则 float 属性失效，并且 display 的值应该被设置为 table 或者 block，具体转换需要看初始转换值。

（3）如果 position 的值不为 absolute 或者 fixed，则判断 float 属性的值是否为 none，如果不是，则 display 的值则按上面的规则转换。注意，如果 position 的值为 relative 并且 float 属性的值存在，则 relative 相对于浮动后的最终位置定位。

（4）如果 float 的值为 none，则判断元素是否为根元素，如果是根元素则 display 属性按照上面的规则转换，如果不是，则保持指定的 display 属性值不变。

总的来说，可以把它看作是一个类似优先级的机制，"position:absolute"和"position:fixed"优先级最高，有它存在的时候，浮动不起作用，'display'的值也需要调整；其次，元素的'float'特性的值不是"none"的时候或者它是根元素的时候，调整'display'的值；最后，非根元素，并且非浮动元素，并且非绝对定位的元素，'display'特性值同设置值。

42. absolute 与 fixed 共同点与不同点?

共同点：

改变行内元素的呈现方式，将 display 置为 inline-block

使元素脱离普通文档流，不再占据文档物理空间

覆盖非定位文档元素

不同点：

abuselute 与 fixed 的根元素不同，abuselute 的根元素可以设置，fixed 根元素是浏览器。

在有滚动条的页面中，absolute 会跟着父元素进行移动，fixed 固定在页面的具体位置。

43. 对 sticky 定位的理解?

sticky 英文字面意思是粘贴，所以可以把它称之为粘性定位。语法：position: sticky; 基于用户的滚动位置来定位。

粘性定位的元素是依赖于用户的滚动，在 position:relative 与 position:fixed 定位之间切换。它的行为就像 position:relative; 而当页面滚动超出目标区域时，它的表现就像 position:fixed;，它会固定在目标位置。元素定位表现为在跨越特定阈值前为相对定位，之后为固定定位。这个特定阈值指的是 top, right, bottom 或 left 之一，换言之，指定 top, right, bottom 或 left 四个阈值其中之一，才可使粘性定位生效。否则其行为与相对定位相同。

参考：https://www.zhangxinxu.com/wordpress/2018/12/css-position-sticky/

五、过渡、变形 && 动画

参考：

过渡 https://juejin.cn/post/6844903512581603335

https://juejin.cn/post/6844904020729921543

变形 https://juejin.cn/post/6844903615920898056

https://juejin.cn/post/6844903585809956871#heading-11

动画 https://juejin.cn/post/6889226357851553805

https://juejin.cn/post/6844903812977655821

44. animation 的属性?

下面各属性的简写（除了 animation-play-state）

animation-name 指定 @keyframes 动画的名称

animation-duration 动画完成一个周期的时间，默认为 0s

animation-timing-function 动画运行的'节奏'，默认是 ease

ease 缓慢开始，缓慢结束

ease-in 先慢后快

ease-out 先快后慢

ease-in-out 以慢速开始和结束的过渡效果

linear 平滑效果

step-start 步进，忽略第一帧

step-end 步进，忽略最后一帧

step-middle 步进，从第一帧到最后一帧

nimation-delay 动画开始播放的延迟时间，默认是 0

animation-iteration-count 动画播放的次数，默认是 1，不可以为负数，infinite 表示无限循环，可以为小数，比如 0.5 代表播放动画的一半即结束

animation-direction 规定动画是否在下一个周期逆向播放

normal: 每次从 @keyframes 0% 执行到 100%，一个周期结束后立即回到 0% 的位置

alternate: 假设 animation-iteration-count: infinite，从 @keyframes 0% 执行到 100%后，再从 100% 的位置 回到 0%，周而复始

reverse: 每次从 @keyframes 100% 执行到 0%，，一个周期结束后立即回到 100% 的位置

alternate-reverse: 假设 animation-iteration-count: infinite，从 @keyframes 100% 执行到 0%后，再从 0% 的位置 回到 100%，周而复始

animation-fill-mode 规定动画的填充模式

none 是默认值，表示动画播放完成后，恢复到初始的状态

forwards 表示动画播放完成后，保持 @keyframes 里最后一帧的样式

backwards 表示开始播放动画之前，元素的样式将设置为动画第一帧的样式

both 相当于同时配置了 forwards 和 backwards。也就是说，动画开始前，元素样式将设置为动画第一帧的样式；而在动画线束状态，元素样式将设置为动画最后一帧样式

animation-play-state 控制动画的运行或暂停，默认是 running

两个属性值，分别是 running 和 pause，当设置为 pause 时，动画会立即会停在当前位置，当取消暂停后会在停住的位置继续执行，而不会回到原点（或终点）重新执行。

六、CANVAS 绘板

45. 什么是 canvas 污染? 如何解决?

如果 canvas 中绘制了跨域请求到的图片，就被污染‘taint’了，这时不能再调用 toBlob(), toDataURL()和 getImageData()等方法, 否则会抛出安全错误(security error).也就是说通过 canvas 无法获取其它域图片的内部信息。

解决办法：把所有图片都重定向同一个域名下：

```
let count = 0;
let bgImg = document.creatElement('img');
let qrImg = document.creatElement('img');
bgImg.src = redirectUrl('x.png');
qrImg.src = redirectUrl('y.png');
[bgImg, qrImg].forEach((e) => {
    e.onload = () => {
        count ++;
        if (count === 2) {
            drawerImg(bgImg, qrImg);
        }
    }
```

```
})
function redirectUrl (url) {
    return 'https://xxx/view?fileUrl=' + encodeURIComponent(url);
}
function drawerImg (imgContent, qrContent, scaleBy = 2) {
    let {bgImgW, bgImgH} = {375, 800};
    let {qrx, qry, qrw, qrh} = {20, 700, 50, 50};
    let Canvas = document.createElement('canvas');
    let ctx = Canvas.getContext("2d");
    Canvas.width = bgImgW * scaleBy;
    Canvas.height= bgImgH * scaleBy;
    ctx.drawImage(imgContent, 0, 0, bgImgW * scaleBy, bgImgH * scaleBy);
    ctx.drawImage(qrContent, qrx * scaleBy, qry * scaleBy, qrw * scaleBy, qrh * scaleBy);
    let nodeI = document.createElement("img");
    nodeI.src = Canvas.toDataURL();
    document.body.appendChild(nodeI)
}
```

46. canvas 如何实现多张图片编辑的图片编辑器?

https://juejin.cn/post/6844904085158625287

参考 https://juejin.cn/post/6989003710030413838

第三章 JavaScript

一、数据类型 & 操作符

1. JavaScript 有哪些数据类型，它们的区别?

JavaScript 共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是 ES6 中新增的数据类型：

Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。

BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

这些数据可以分为原始数据类型和引用数据类型：

栈：原始数据类型（Undefined、Null、Boolean、Number、String）

堆：引用数据类型（对象、数组和函数）

两种类型的区别在于存储位置的不同：

原始数据类型直接存储在栈（stack）中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以

放入栈中存储；

引用数据类型存储在堆（heap）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

堆和栈的概念存在于数据结构和操作系统内存中，在数据结构中：

在数据结构中，栈中数据的存取方式为先进后出。

堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。

在操作系统中，内存被分为栈区和堆区：

栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆区内存一般由开发着分配释放，若开发者不释放，程序结束时可能由垃圾回收机制回收。

2. 数据类型检测的方式有哪些?

（1）typeof

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

其中数组、对象、null 都会被判断为 object，其他判断都正确。

（2）instanceof

instanceof 可以正确判断对象的类型，其内部运行机制是判断在其原型链中能否找到该类型的原型。

```
console.log(2 instanceof Number);                   // false
console.log(true instanceof Boolean);               // false
console.log('str' instanceof String);               // false
console.log([] instanceof Array);                   // true
console.log(function(){} instanceof Function);      // true
console.log({} instanceof Object);                  // true
```

可以看到，instanceof 只能正确判断引用数据类型，而不能判断基本数据类型。instanceof 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。

（3） constructor

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
```

constructor 有两个作用，一是判断数据的类型，二是对象实例通过 constrcutor 对象访问它的构造函数。需

要注意，如果创建一个对象来改变它的原型，constructor 就不能用来判断数据类型了：

```
function Fn(){};
Fn.prototype = new Array();
var f = new Fn();
console.log(f.constructor===Fn);       // false
console.log(f.constructor===Array); // true
```

（4）Object.prototype.toString.call()

Object.prototype.toString.call() 使用 Object 对象的原型方法 toString 来判断数据类型：

```
var a = Object.prototype.toString;
console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
```

同样是检测对象 obj 调用 toString 方法，obj.toString()的结果和 Object.prototype.toString.call(obj)的结果不一样，这是为什么？

这是因为 toString 是 Object 的原型方法，而 Array、function 等类型作为 Object 的实例，都重写了 toString 方法。不同的对象类型调用 toString 方法时，根据原型链的知识，调用的是对应的重写之后的 toString 方法（function 类型返回内容为函数体的字符串，Array 类型返回元素组成的字符串…），而不会去调用 Object 上原型 toString 方法（返回对象的具体类型），所以采用 obj.toString()不能得到其对象类型，只能将 obj 转换为字符串类型；因此，在想要得到对象的具体类型时，应该调用 Object 原型上的 toString 方法。

3. 判断数组的方式有哪些?

通过 Object.prototype.toString.call()做判断

```
Object.prototype.toString.call(obj).slice(8,-1) === 'Array';
```

通过原型链做判断

```
obj.__proto__ === Array.prototype;
```

通过 ES6 的 Array.isArray()做判断

```
Array.isArrray(obj);
```

通过 instanceof 做判断

```
obj instanceof Array
```

通过 Array.prototype.isPrototypeOf

```
Array.prototype.isPrototypeOf(obj)
```

4. null 和 undefined 区别?

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

undefined 代表的含义是未定义，null 代表的含义是空对象。一般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。

undefined 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断。我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回 “object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

5. undefined 与 undeclared 的区别?

已在作用域中声明但还没有赋值的变量是 undefined。相反，还没有在作用域中声明过的变量是 undeclared 。

对于 undeclared 变量的引用，浏览器会报引用错误，如 ReferenceError: b is not defined 。但是我们可以使用 typ

eof 的安全防范机制来避免报错，因为对于 undeclared（或者 not defined ）变量，typeof 会返回 "undefined"。

6. typeof null 的结果是什么，为什么?

typeof null 的结果是 Object。

在 JavaScript 第一个版本中，所有值都存储在 32 位的单元中，每个单元包含一个小的 类型标签(1-3 bits) 以及当前要存储值的真实数据。类型标签存储在每个单元的低位中，共有五种数据类型：

000: object - 当前存储的数据指向一个对象。

1: int - 当前存储的数据是一个 31 位的有符号整数。

010: double - 当前存储的数据指向一个双精度的浮点数。

100: string - 当前存储的数据指向一个字符串。

110: boolean - 当前存储的数据是布尔值。

如果最低位是 1，则类型标签标志位的长度只有一位；如果最低位是 0，则类型标签标志位的长度占三位，为存储其他四种数据类型提供了额外两个 bit 的长度。

有两种特殊数据类型：

undefined 的值是 (-2)30(一个超出整数范围的数字)；

null 的值是机器码 NULL 指针(null 指针的值全是 0)

那也就是说 null 的类型标签也是 000，和 Object 的类型标签一样，所以会被判定为 Object。

7. intanceof 操作符的实现原理及实现?

instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

```
function myInstanceof(left, right) {
  // 获取对象的原型
  let proto = Object.getPrototypeOf(left)
  // 获取构造函数的 prototype 对象
  let prototype = right.prototype;
  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
```

```
    if (proto === prototype) return true;
    // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf 方法用来获取指定对象的原型
    proto = Object.getPrototypeOf(proto);
  }
}
```

8. 如何获取安全的 undefined 值?

因为 undefined 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。表达式 void ___ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。因此可以用 void 0 来获得 undefined。

9. typeof NaN 的结果是什么?

NaN 指“不是一个数字”（not a number），NaN 是一个“警戒值”（sentinel value，有特殊用途的常规值），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。

typeof NaN; // "number"

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即 x === x 不成立）的值。而 NaN !== NaN 为 true。

10. isNaN 和 Number.isNaN 函数的区别?

函数 isNaN 接收参数后，会尝试将这个参数转换为数值，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会返回 true ，会影响 NaN 的判断。

函数 Number.isNaN 会首先判断传入参数是否为数字，如果是数字再继续判断是否为 NaN ，不会进行数据类型的转换，这种方法对于 NaN 的判断更为准确。

11. JavaScript 有哪些数据类型，它们的区别?

J== 操作符的强制类型转换规则？

对于 == 来说，如果对比双方的类型不一样，就会进行类型转换。假如对比 x 和 y 是否相同，就会进行如下判断流程：

首先会判断两者类型是否**相同，**相同的话就比较两者的大小；

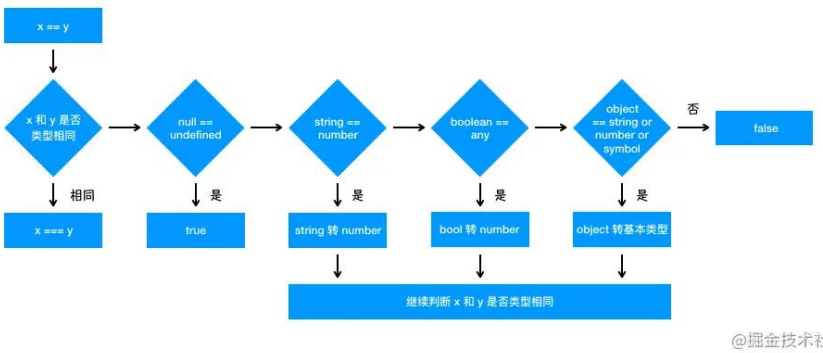
类型不相同的话，就会进行类型转换；

会先判断是否在对比 null 和 undefined，是的话就会返回 true

判断两者类型是否为 string 和 number，是的话就会将字符串转换为 number

判断其中一方是否为 boolean，是的话就会把 boolean 转为 number 再进行判断

判断其中一方是否为 object 且另一方为 string、number 或者 symbol，是的话就会把 object 转为原始类型再进行判断。



12. 其他值到字符串的转换规则?

Null 和 Undefined 类型 ，null 转换为 "null"，undefined 转换为 "undefined"，

Boolean 类型，true 转换为 "true"，false 转换为 "false"。

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。

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。

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。

13. 其他值到数字值的转换规则?

Undefined 类型的值转换为 NaN。

Null 类型的值转换为 0。

Boolean 类型的值，true 转换为 1，false 转换为 0。

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。

Symbol 类型的值不能转换为数字，会报错。

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。

14. 其他值到布尔类型的值的转换规则?

以下这些是假值： • undefined • null • false • +0、-0 和 NaN • ""

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

15. || 和 && 操作符的返回值?

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先强制转换为布尔类型，然后再执行条件判断。

对于 || 来说,如果条件判断结果为 true 就返回第一个操作数的值,如果为 false 就返回第二个操作数的值。

&& 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果。

16. Object.is() 与比较操作符 “===”、“==” 的区别?

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 是相等的。

17. 什么是 JavaScript 中的包装类型?

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象，如：

```
const a = "abc";
```

```
a.length; // 3
a.toUpperCase(); // "ABC"
```

在访问'abc'.length 时，JavaScript 将'abc'在后台转换成 String('abc')，然后再访问其 length 属性。

JavaScript 也可以使用 Object 函数显式地将基本类型转换为包装类型：

```
var a = 'abc'
Object(a) // String {"abc"}
```

也可以使用 valueOf 方法将包装类型倒转成基本类型：

```
var a = 'abc'
var b = Object(a)
var c = b.valueOf() // 'abc'
```

看看如下代码会打印出什么：

```
var a = new Boolean( false );
if (!a) {
console.log( "Oops" ); // never runs
}
```

答案是什么都不会打印，因为虽然包裹的基本类型是 false，但是 false 被包裹成包装类型后就成了对象，所以其非值为 false，所以循环体中的内容不会运行。

18. JavaScript 中如何进行隐式类型转换?

首先要介绍 ToPrimitive 方法，这是 JavaScript 中每个值隐含的自带的方法，用来将值 （无论是基本类型值还是对象）转换为基本类型值。如果值为基本类型，则直接返回值本身；如果值为对象，其看起来大概是这样：

```
/**
* @obj 需要转换的对象
* @type 期望的结果类型
*/
ToPrimitive(obj,type)
```

type 的值为 number 或者 string。

（1）当 type 为 number 时规则如下：

调用 obj 的 valueOf 方法，如果为原始值，则返回，否则下一步；

调用 obj 的 toString 方法，后续同上；

抛出 TypeError 异常。

（2）当 type 为 string 时规则如下：

调用 obj 的 toString 方法，如果为原始值，则返回，否则下一步；

调用 obj 的 valueOf 方法，后续同上；

抛出 TypeError 异常。

可以看出两者的主要区别在于调用 toString 和 valueOf 的先后顺序。默认情况下：

如果对象为 Date 对象，则 type 默认为 string；

其他情况下，type 默认为 number。

总结上面的规则，对于 Date 以外的对象，转换为基本类型的大概规则可以概括为一个函数：

```
var objToNumber = value => Number(value.valueOf().toString())
objToNumber([]) === 0
objToNumber({}) === NaN
```

而 JavaScript 中的隐式类型转换主要发生在+、-、*、/以及==、>、<这些运算符之间。而这些运算符只能操作基本类型值，所以在进行这些运算前的第一步就是将两边的值用 ToPrimitive 转换成基本类型，再进行操作。

以下是基本类型的值在不同操作符的情况下隐式转换的规则（对于对象，其会被 ToPrimitive 转换成基本类型，所以最终还是要应用基本类型转换规则）：

+操作符

+操作符的两边有至少一个 string 类型变量时，两边的变量都会被隐式转换为字符串；其他情况下两边的变量都会被转换为数字。

```
1 + '23' // '123'
 1 + false // 1
 1 + Symbol() // Uncaught TypeError: Cannot convert a Symbol value to a number
 '1' + false // '1false'
 false + true // 1
```

-、*、\操作符

NaN 也是一个数字

```
1 * '23' // 23
 1 * false // 0
 1 / 'aa' // NaN
```

对于==操作符

操作符两边的值都尽量转成 number：

```
3 == true // false, 3 转为 number 为 3，true 转为 number 为 1
'0' == false //true, '0'转为 number 为 0，false 转为 number 为 0
'0' == 0 // '0'转为 number 为 0
```

对于<和>比较符

如果两边都是字符串，则比较字母表顺序：

```
'ca' < 'bd' // false
'a' < 'b' // true
```

其他情况下，转换为数字再比较：

```
'12' < 13 // true
false > -1 // true
```

以上说的是基本类型的隐式转换，而对象会被 ToPrimitive 转换为基本类型再进行转换：

```
var a = {}
a > 2 // false
```

其对比过程如下：

a.valueOf() // {}, 上面提到过，ToPrimitive 默认 type 为 number，所以先 valueOf，结果还是个对象，下一步

a.toString() // "[object Object]"，现在是一个字符串了

Number(a.toString()) // NaN，根据上面 < 和 > 操作符的规则，要转换成数字

NaN > 2 //false，得出比较结果

又比如：

```
var a = {name:'Jack'}
var b = {age: 18}
a + b // "[object Object][object Object]"
```

运算过程如下：

a.valueOf() // {}，上面提到过，ToPrimitive 默认 type 为 number，所以先 valueOf，结果还是个对象，下一步

a.toString() // "[object Object]"

b.valueOf() // 同理

b.toString() // "[object Object]"

a + b // "[object Object][object Object]"

19. + 操作符什么时候用于字符串的拼接?

根据 ES5 规范，如果某个操作数是字符串或者能够通过以下步骤转换为字符串的话，+ 将进行拼接操作。如果其中一个操作数是对象（包括数组），则首先对其调用 ToPrimitive 抽象操作，该抽象操作再调用 [[DefaultValue]]，以数字作为上下文。如果不能转换为字符串，则会将其转换为数字类型来进行计算。

简单来说就是，如果 + 的其中一个操作数是字符串（或者通过以上步骤最终得到字符串），则执行字符串拼接，否则执行数字加法。

那么对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字。

20. object.assign 和扩展运算法是深拷贝还是浅拷贝，两者区别?

两者都是浅拷贝。

Object.assign()方法接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。它会修改了一个对象，因此会触发 ES6 setter。

扩展操作符（…）使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象中。它不复制继承的属性或类的属性，但是它会复制 ES6 的 symbols 属性。

21. 为什么会有 BigInt 的提案?

JavaScript 中 Number.MAX_SAFE_INTEGER 表示最大安全数字，计算结果是 9007199254740991，即在这个数范围内不会出现精度丢失（小数除外）。但是一旦超过这个范围，js 就会出现计算不准确的情况，这在大数计算的时候不得不依靠一些第三方库进行解决，因此官方提出了 BigInt 来解决此问题。

要创建一个 BigInt ，我们只需要在任意整型的字面量上加上一个 n 后缀即可。例如，把 123 写成 123n。这个全局的 BigInt(number) 可以用来将一个 Number 转换为一个 BigInt，言外之意就是说，BigInt(123) === 123n。

二、JS 基础

1. new 操作符的实现原理?

new 操作符的执行过程：

（1）首先创建了一个新的空对象

（2）设置原型，将对象的原型设置为函数的 prototype 对象。

（3）让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）

（4）判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，返回这个引用类型的对象。

2. JS 中有哪些内置函数?

Object、 Array、 Boolean、Number、 String、 Function、 Date、RegExp、 Error

3. map 和 Object 的区别?

	Map	Object
意外的键	Map默认情况不包含任何键，只包含显式插入的键。	Object 有一个原型，原型链上的键名有可能和自己在对象上的设置的键名产生冲突。
键的类型	Map的键可以是任意值，包括函数、对象或任意基本类型。	Object 的键必须是 String 或是Symbol。
键的顺序	Map 中的 key 是有序的。因此，当迭代的时候， Map 对象以插入的顺序返回键值。	Object 的键是无序的
Size	Map 的键值对个数可以轻易地通过size 属性获取	Object 的键值对个数只能手动计算
迭代	Map 是 iterable 的，所以可以直接被迭代。	迭代Object需要以某种方式获取它的键然后才能迭代。
性能	在频繁增删键值对的场景下表现更好。	在频繁添加和删除键值对的场景下未作出优化。

4. map 和 weakMap 的区别?

（1）Map

map 本质上就是键值对的集合，但是普通的 Object 中的键值对中的键只能是字符串。而 ES6 提供的 Map 数据结构类似于对象，但是它的键不限制范围，可以是任意类型，是一种更加完善的 Hash 结构。如果 Map 的键是一个原始数据类型，只要两个键严格相同，就视为是同一个键。

实际上 Map 是一个数组，它的每一个数据也都是一个数组，其形式如下：

```
const map = [
    ["name","张三"],
    ["age",18],
]
```

Map 数据结构有以下操作方法：

size： map.size 返回 Map 结构的成员总数。

set(key,value)：设置键名 key 对应的键值 value，然后返回整个 Map 结构，如果 key 已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前 Map 对象，所以可以链式调用）

get(key)：该方法读取 key 对应的键值，如果找不到 key，返回 undefined。

has(key)：该方法返回一个布尔值，表示某个键是否在当前 Map 对象中。

delete(key)：该方法删除某个键，返回 true，如果删除失败，返回 false。

clear()：map.clear()清除所有成员，没有返回值。

Map 结构原生提供是三个遍历器生成函数和一个遍历方法

keys()：返回键名的遍历器。

values()：返回键值的遍历器。

entries()：返回所有成员的遍历器。

forEach()：遍历 Map 的所有成员。

（2）WeakMap

WeakMap 对象也是一组键值对的集合，其中的键是弱引用的。其键必须是对象，原始数据类型不能作为 key 值，而值可以是任意的。

该对象也有以下几种方法：

set(key,value)：设置键名 key 对应的键值 value，然后返回整个 Map 结构，如果 key 已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前 Map 对象，所以可以链式调用）

get(key)：该方法读取 key 对应的键值，如果找不到 key，返回 undefined。

has(key)：该方法返回一个布尔值，表示某个键是否在当前 Map 对象中。

delete(key)：该方法删除某个键，返回 true，如果删除失败，返回 false。

其 clear()方法已经被弃用，所以可以通过创建一个空的 WeakMap 并替换原对象来实现清除。

WeakMap 的设计目的在于，有时想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用。一旦不再需要这两个对象，就必须手动删除这个引用，否则垃圾回收机制就不会释放对象占用的内存。

而 WeakMap 的键名所引用的对象都是弱引用，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，WeakMap 里面的键名对象和所对应的键值对会自动消失，不用手动删除引用。

总结：

Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括

对象）都可以当作键。

WeakMap 结构与 Map 结构类似，也是用于生成键值对的集合。但是 WeakMap 只接受对象作为键名（ null 除外），不接受其他类型的值作为键名。而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。

5. Set，WeakSet，Map 和 WeakMap?

https://juejin.cn/post/6844904191610060814

https://juejin.cn/post/6844903897409011720

6. JavaScript 有哪些内置对象?

全局的对象（ global objects ）或称标准内置对象，不要和 "全局对象（global object）" 混淆。这里说的全局的对象是说在

全局作用域里的对象。全局作用域中的其他对象可以由用户的脚本创建或由宿主程序提供。

标准内置对象的分类：

（1）值属性，这些全局属性返回一个简单值，这些值没有自己的属性和方法。例如 Infinity、NaN、undefined、null 字面量

（2）函数属性，全局函数可以直接调用，不需要在调用时指定所属对象，执行结束后会将结果直接返回给调用者。例如 eval()、parseFloat()、parseInt() 等

（3）基本对象，基本对象是定义或使用其他对象的基础。基本对象包括一般对象、函数对象和错误对象。例如 Object、Function、Boolean、Symbol、Error 等

（4）数字和日期对象，用来表示数字、日期和执行数学计算的对象。例如 Number、Math、Date

（5）字符串，用来表示和操作字符串的对象。例如 String、RegExp

（6）可索引的集合对象，这些对象表示按照索引值来排序的数据集合，包括数组和类型数组，以及类数组结构的对象。例如 Array

（7）使用键的集合对象，这些集合对象在存储数据时会使用到键，支持按照插入顺序来迭代元素。

例如 Map、Set、WeakMap、WeakSet

（8）矢量集合，SIMD 矢量集合中的数据会被组织为一个数据序列。

例如 SIMD 等

（9）结构化数据，这些对象用来表示和操作结构化的缓冲区数据，或使用 JSON 编码的数据。例如 JSON 等

（10）控制抽象对象

例如 Promise、Generator 等

（11）反射。例如 Reflect、Proxy

（12）国际化，为了支持多语言处理而加入 ECMAScript 的对象。例如 Intl、Intl.Collator 等

（13）WebAssembly

（14）其他。例如 arguments

总结：

js 中的内置对象主要指的是在程序执行前存在全局作用域里的由 js 定义的一些全局值属性、函数和用来实例化其他对象的构造函数对象。一般经常用到的如全局变量值 NaN、undefined，全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，还有提供数学计算的单体内置对象如 Math 对象。

7. 对 JSON 的理解?

JSON 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。

在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列

化为

JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，

JSON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。

JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问。

8. JavaScript 脚本延迟加载的方式有哪些?

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

defer 属性： 给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。

async 属性： 给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。

动态创建 DOM 方式： 动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。

使用 setTimeout 延迟方法： 设置一个定时器来延迟加载 js 脚本文件

让 JS 最后加载： 将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

9. JavaScript 类数组对象的定义？？

一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有 arguments 和 DOM 方法的返回结果，还有一个函数也可以被看作是类数组对象，因为它含有 length 属性值，代表可接收的参数个数。

常见的类数组转换为数组的方法有这样几种：

（1）通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

（2）通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

（3）通过 apply 调用数组的 concat 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

（4）通过 Array.from 方法来实现转换

```
Array.from(arrayLike);
```

10. 数组有哪些原生方法？？

数组和字符串的转换方法：toString()、toLocalString()、join() 其中 join() 方法可以指定转换为字符串时的分隔符。

数组尾部操作的方法 pop() 和 push()，push 方法可以传入多个参数。

数组首部操作的方法 shift() 和 unshift() 重排序的方法 reverse() 和 sort()，sort() 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。

数组连接的方法 concat() ，返回的是拼接好的数组，不影响原数组。

数组截取办法 slice()，用于截取数组中的一部分返回，不影响原数组。

数组插入方法 splice()，影响原数组查找特定项的索引的方法，indexOf() 和 lastIndexOf() 迭代方法 every()、some()、filter()、map() 和 forEach() 方法

数组归并方法 reduce() 和 reduceRight() 方法。

11. Unicode、UTF-8、UTF-16、UTF-32 的区别?

（1）Unicode

在说 Unicode 之前需要先了解一下 ASCII 码：ASCII 码（American Standard Code for Information Interchange）称为美国标准信息交换码。

它是基于拉丁字母的一套电脑编码系统。

它定义了一个用于代表常见字符的字典。

它包含了"A-Z"(包含大小写)，数据"0-9" 以及一些常见的符号。

它是专门为英语而设计的，有 128 个编码，对其他语言无能为力

ASCII 码可以表示的编码有限，要想表示其他语言的编码，还是要使用 Unicode 来表示，可以说 Unicode 是 ASCII 的超集。

Unicode 全称 Unicode Translation Format，又叫做统一码、万国码、单一码。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

Unicode 的实现方式（也就是编码方式）有很多种，常见的是 UTF-8、UTF-16、UTF-32 和 USC-2。

（2）UTF-8

UTF-8 是使用最广泛的 Unicode 编码方式，它是一种可变长的编码方式，可以是 1—4 个字节不等，它可以完全兼容 ASCII 码的 128 个字符。

注意： UTF-8 是一种编码方式，Unicode 是一个字符集合。

UTF-8 的编码规则：

对于单字节的符号，字节的第一位为 0，后面的 7 位为这个字符的 Unicode 编码，因此对于英文字母，它的 Unicode 编码和 ACSII 编码一样。

对于 n 字节的符号，第一个字节的前 n 位都是 1，第 n+1 位设为 0，后面字节的前两位一律设为 10，剩下的没有提及的二进制位，全部为这个符号的 Unicode 码 。

编码范围（编号对应的十进制数）二进制格式 0x00—0x7F （0-127）0xxxxxxx0x80—0x7FF （128-2047）110xxxxx 10xxxxxx0x800—0xFFFF （2048-65535）1110xxxx 10xxxxxx 10xxxxxx0x10000—0x10FFFF （65536 以上）11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

那该如何通过具体的 Unicode 编码，进行具体的 UTF-8 编码呢？步骤如下：

找到该 Unicode 编码的所在的编号范围，进而找到与之对应的二进制格式

将 Unicode 编码转换为二进制数（去掉最高位的 0）

将二进制数从右往左一次填入二进制格式的 X 中，如果有 X 未填，就设为 0

来看一个实际的例子：

“马” 字的 Unicode 编码是：0x9A6C，整数编号是 39532

首选确定了该字符在第三个范围内，它的格式是 1110xxxx 10xxxxxx 10xxxxxx

39532 对应的二进制数为 1001 1010 0110 1100

将二进制数填入 X 中，结果是：11101001 10101001 10101100

（3）UTF-16

1. 平面的概念

在了解 UTF-16 之前，先看一下平面的概念：

Unicode 编码中有很多很多的字符，它并不是一次性定义的，而是分区进行定义的，每个区存放 65536（216）个字符，这称为一个平面，目前总共有 17 个平面。

最前面的一个平面称为基本平面，它的码点从 0 — 216-1，写成 16 进制就是 U+0000 — U+FFFF，那剩下的 16 个平面就是辅助平面，码点范围是 U+10000—U+10FFFF。

2. UTF-16 概念：

UTF-16 也是 Unicode 编码集的一种编码形式，把 Unicode 字符集的抽象码位映射为 16 位长的整数（即码元）的序列，用于数据存储或传递。Unicode 字符的码位需要 1 个或者 2 个 16 位长的码元来表示，因此 UTF-16 也是用变长字节表示的。

3. UTF-16 编码规则：

编号在 U+0000—U+FFFF 的字符（常用字符集），直接用两个字节表示。

编号在 U+10000—U+10FFFF 之间的字符，需要用四个字节表示。

4. 编码识别

那么问题来了，当遇到两个字节时，怎么知道是把它当做一个字符还是和后面的两个字节一起当做一个字符呢？

UTF-16 编码肯定也考虑到了这个问题，在基本平面内，从 U+D800 — U+DFFF 是一个空段，也就是说这个区间的码点不对应任何的字符，因此这些空段就可以用来映射辅助平面的字符。

辅助平面共有 220 个字符位，因此表示这些字符至少需要 20 个二进制位。UTF-16 将这 20 个二进制位分成两半，前 10 位映射在 U+D800 — U+DBFF，称为高位（H），后 10 位映射在 U+DC00 — U+DFFF，称为低位（L）。这就相当于，将一个辅助平面的字符拆成了两个基本平面的字符来表示。

因此，当遇到两个字节时，发现它的码点在 U+D800 —U+DBFF 之间，就可以知道，它后面的两个字节的码点应该在 U+DC00 — U+DFFF 之间，这四个字节必须放在一起进行解读。

5. 举例说明

以 " " 字为例，它的 Unicode 码点为 0x21800，该码点超出了基本平面的范围，因此需要用四个字节来表示，步骤如下：

首先计算超出部分的结果：0x21800 - 0x10000

将上面的计算结果转为 20 位的二进制数，不足 20 位就在前面补 0，结果为：0001000110 0000000000

将得到的两个 10 位二进制数分别对应到两个区间中

U+D800 对应的二进制数为 1101100000000000， 将 0001000110 填充在它的后 10 个二进制位，得到 1101100001000110，转成 16 进制数为 0xD846。同理，低位为 0xDC00，所以这个字的 UTF-16 编码为 0xD846 0xDC00

（4） UTF-32

UTF-32 就是字符所对应编号的整数二进制形式，每个字符占四个字节，这个是直接进行转换的。该编码方式占用的储存空间较多，所以使用较少。

比如“马” 字的 Unicode 编号是：U+9A6C，整数编号是 39532，直接转化为二进制：1001 1010 0110 1100，这就是它的 UTF-32 编码。

（5）总结

Unicode、UTF-8、UTF-16、UTF-32 有什么区别？

Unicode 是编码字符集（字符集），而 UTF-8、UTF-16、UTF-32 是字符集编码（编码规则）；

UTF-16 使用变长码元序列的编码方式，相较于定长码元序列的 UTF-32 算法更复杂，甚至比同样是变长码元序列的 UTF-8 也更为复杂，因为其引入了独特的代理对这样的代理机制；

UTF-8 需要判断每个字节中的开头标志信息，所以如果某个字节在传送过程中出错了，就会导致后面的字节也会解析出错；而 UTF-16 不会判断开头标志，即使错也只会错一个字符，所以容错能力教强；

如果字符内容全部英文或英文与其他文字混合，但英文占绝大部分，那么用 UTF-8 就比 UTF-16 节省了很多空间；而如果字符内容全部是中文这样类似的字符或者混合字符中中文占绝大多数，那么 UTF-16 就占优势了，可以节省很多空间。

12. 常见的位运算符有哪些？其计算规则是什么？

现代计算机中数据都是以二进制的形式存储的，即 0、1 两种状态，计算机对二进制数据进行的运算加减乘除等都是叫位运算，即将符号位共同参与运算的运算。

常见的位运算有以下几种：

运算符描述运算规则&与两个位都为 1 时，结果才为 1``或^异或两个位相同为 0，相异为 1~取反 0 变 1，1 变 0<<左移各二进制位全部左移若干位，高位丢弃，低位补 0>>右移各二进制位全部右移若干位，正数左补 0，负数左补 1，右边丢弃

1. 按位与运算符（&）

定义： 参加运算的两个数据按二进制位进行“与”运算。

运算规则：

0 & 0 = 0

0 & 1 = 0

1 & 0 = 0

1 & 1 = 1

总结：两位同时为 1，结果才为 1，否则结果为 0。

例如：3&5 即：

0000 0011

0000 0101

= 0000 0001

因此 3&5 的值为 1。

注意：负数按补码形式参加按位与运算。

用途：

（1）判断奇偶

只要根据最未位是 0 还是 1 来决定，为 0 就是偶数，为 1 就是奇数。因此可以用 if ((i & 1) == 0)代替 if (i % 2 == 0)来判断 a 是不是偶数。

（2）清零

如果想将一个单元清零，即使其全部二进制位为 0，只要与一个各位都为零的数值相与，结果为零。

2. 按位或运算符（|）

定义： 参加运算的两个对象按二进制位进行“或”运算。

运算规则：

0 | 0 = 0

0 | 1 = 1

1 | 0 = 1

1 | 1 = 1

总结：参加运算的两个对象只要有一个为 1，其值为 1。

例如：3|5 即：

0000 0011

0000 0101

= 0000 0111

因此，3|5 的值为 7。

注意：负数按补码形式参加按位或运算。

3. 异或运算符（^）

定义： 参加运算的两个数据按二进制位进行“异或”运算。

运算规则：

0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0

总结：参加运算的两个对象，如果两个相应位相同为 0，相异为 1。

例如：3|5 即：

0000 0011

0000 0101

= 0000 0110

复制代码

因此，3^5 的值为 6。

异或运算的性质:

交换律：(a^b)^c == a^(b^c)

结合律：(a + b)^c == a^b + b^c

对于任何数 x，都有 x^x=0，x^0=x

自反性: a^b^b=a^0=a;

4. 取反运算符 (~)

定义： 参加运算的一个数据按二进制进行“取反”运算。

运算规则：

~ 1 = 0~ 0 = 1

复制代码

总结：对一个二进制数按位取反，即将 0 变 1，1 变 0。

例如：~6 即：

0000 0110= 1111 1001

在计算机中，正数用原码表示，负数使用补码存储，首先看最高位，最高位 1 表示负数，0 表示正数。此计算机二进制码为负数，最高位为符号位。

当发现按位取反为负数时，就直接取其补码，变为十进制：

0000 0110 = 1111 1001 反码：1000 0110 补码：1000 0111

因此，~6 的值为-7。

5. 左移运算符（<<）

定义： 将一个运算对象的各二进制位全部左移若干位，左边的二进制位丢弃，右边补 0。

设 a=1010 1110，a = a<< 2 将 a 的二进制位左移 2 位、右补 0，即得 a=1011 1000。

若左移时舍弃的高位不包含 1，则每左移一位，相当于该数乘以 2。

6. 右移运算符（>>）

定义： 将一个数的各二进制位全部右移若干位，正数左补 0，负数左补 1，右边丢弃。

例如：a=a>>2 将 a 的二进制位右移 2 位，左补 0 或者 左补 1 得看被移数是正还是负。

操作数每右移一位，相当于该数除以 2。

7. 原码、补码、反码

上面提到了补码、反码等知识，这里就补充一下。

计算机中的有符号数有三种表示方法，即原码、反码和补码。三种表示方法均有符号位和数值位两部分，符号位都是用 0 表示“正”，用 1 表示“负”，而数值位，三种表示方法各不相同。

（1）原码

原码就是一个数的二进制数。例如：10 的原码为 0000 1010

（2）反码

正数的反码与原码相同，如：10 反码为 0000 1010

负数的反码为除符号位，按位取反，即 0 变 1，1 变 0。

例如：-10

原码：1000 1010

反码：1111 0101

（3）补码

正数的补码与原码相同，如：10 补码为 0000 1010

负数的补码是原码除符号位外的所有位取反即 0 变 1，1 变 0，然后加 1，也就是反码加 1。

例如：-10

原码：1000 1010

反码：1111 0101

补码：1111 0110

13. 为什么函数的 arguments 参数是类数组而不是数组？如何遍历类数组？

arguments 是一个对象，它的属性是从 0 开始依次递增的数字，还有 callee 和 length 等属性，与数组相似；但是它却没有数组常见的方法属性，如 forEach, reduce 等，所以叫它们类数组。

要遍历类数组，有三个方法：

（1）将数组的方法应用到类数组上，这时候就可以使用 call 和 apply 方法，

（2）使用 Array.from 方法将类数组转化成数组：

（3）使用展开运算符将类数组转化成数组。

14. 什么是 DOM 和 BOM?

DOM 指的是文档对象模型，它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。

BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的法和接口。BOM 的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

15. 对类数组对象的理解，如何转化为数组?

一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有 arguments 和 DOM 方法的返回结果，函数参数也可以被看作是类数组对象，因为它含有 length 属性值，代表可接收的参数个数。

常见的类数组转换为数组的方法有这样几种：

通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

通过 apply 调用数组的 concat 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

通过 Array.from 方法来实现转换

```
Array.from(arrayLike);
```

16. escape、encodeURI、encodeURIComponent 的区别?

encodeURI 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。

encodeURIComponent 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。

escape 和 encodeURI 的作用相同，不过它们对于 unicode 编码为 0xff 之外字符的时候会有区别，escape 是直接在字符的 unicode 编码前加上 %u，而 encodeURI 首先会将字符转换为 UTF-8 的格式，再在每个字节前加上 %。

17. 对 AJAX 的理解，实现一个 AJAX 请求?

AJAX 是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

创建 AJAX 请求的步骤：

创建一个 XMLHttpRequest 对象。

在这个对象上使用 open 方法创建一个 HTTP 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。

在发起请求前，可以为这个对象添加一些信息和监听函数。比如说可以通过 setRequestHeader 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 onreadystatechange 事件，可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 response 中的数据来对页面进行更新了。

当对象的属性和监听函数设置完成后，最后调用 sent 方法来向服务器发起请求，可以传入参数作为发送的数据体。

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", url, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
```

```
    console.error(this.statusText);
};
// 设置请求头信息
xhr.responseType = "json";
xhr.setRequestHeader("Accept", "application/json");
// 发送 Http 请求
xhr.send(null);
```

使用 Promise 封装 AJAX：

```
// promise 封装实现：
function getJSON(url) {
    // 创建一个 promise 对象
    let promise = new Promise(function(resolve, reject) {
        let xhr = new XMLHttpRequest();
        // 新建一个 http 请求
        xhr.open("GET", url, true);
        // 设置状态的监听函数
        xhr.onreadystatechange = function() {
            if (this.readyState !== 4) return;
            // 当请求成功或失败时，改变 promise 的状态
            if (this.status === 200) {
                resolve(this.response);
            } else {
                reject(new Error(this.statusText));
            }
        };
        // 设置错误监听函数
        xhr.onerror = function() {
            reject(new Error(this.statusText));
        };
        // 设置响应的数据类型
        xhr.responseType = "json";
        // 设置请求头信息
        xhr.setRequestHeader("Accept", "application/json");
        // 发送 http 请求
        xhr.send(null);
    });
    return promise;
}
```

18. JavaScript 为什么要进行变量提升，它导致了什么问题?

变量提升的表现是，无论在函数中何处位置声明的变量，好像都被提升到了函数的首部，可以在变量声明前访问到而不会报错。

造成变量声明提升的本质原因是 js 引擎在代码执行前有一个解析的过程，创建了执行上下文，初始化了一些代码执行时需要用到的对象。当访问一个变量时，会到当前执行上下文中的作用域链中去查找，而作用域链的首端指向的是当前执行上下文的变量对象，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的。

首先要知道，JS 在拿到一个变量或者一个函数的时候，会有两步操作，即解析和执行。

在解析阶段，JS 会检查语法，并对函数进行预编译。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为 undefined，函数先声明好可使用。在一个函数执行之前，也会创建一个函数执行上下文环境，跟全局执行上下文类似，不过函数执行上下文会多出 this、arguments 和函数的参数。

全局上下文：变量定义，函数声明

函数上下文：变量定义，函数声明，this，arguments

在执行阶段，就是按照代码的顺序依次执行。

那为什么会进行变量提升呢？主要有以下两个原因：

提高性能

容错性更好

（1）提高性能

在 JS 代码执行之前，会进行语法检查和预编译，并且这一操作只进行一次。这么做就是为了提高性能，如果没有这一步，那么每次执行代码前都必须重新解析一遍该变量（函数），而这是没有必要的，因为变量（函数）的代码并不会改变，解析一遍就够了。

在解析的过程中，还会为函数生成预编译代码。在预编译时，会统计声明了哪些变量、创建了哪些函数，并对函数的代码进行压缩，去除注释、不必要的空白等。这样做的好处就是每次执行函数时都可以直接为该函数分配栈空间（不需要再解析一遍去获取代码中声明了哪些变量，创建了哪些函数），并且因为代码压缩的原因，代码执行也更快了。

（2）容错性更好

变量提升可以在一定程度上提高 JS 的容错性，看下面的代码：

```
a = 1;var a;console.log(a);
```

如果没有变量提升，这两行代码就会报错，但是因为有了变量提升，这段代码就可以正常执行。

虽然，在可以开发过程中，可以完全避免这样写，但是有时代码很复杂的时候。可能因为疏忽而先使用后定义了，这样也不会影响正常使用。由于变量提升的存在，而会正常运行。

总结：

解析和预编译过程中的声明提升可以提高性能，让函数可以在执行时预先为变量分配栈空间

声明提升还可以提高 JS 代码的容错性，使一些不规范的代码也可以正常执行

变量提升虽然有一些优点，但是他也会造成一定的问题，在 ES6 中提出了 let、const 来定义变量，它们就没有变量提升的机制。下面看一下变量提升可能会导致的问题：

```
var tmp = new Date();
function fn(){
```

```
console.log(tmp);
    if(false){
        var tmp = 'hello world';
    }
}
fn();   // undefined
```

在这个函数中，原本是要打印出外层的 tmp 变量，但是因为变量提升的问题，内层定义的 tmp 被提到函数内部的最顶部，相当于覆盖了外层的 tmp，所以打印结果为 undefined。

```
var tmp = 'hello world';
for (var i = 0; i < tmp.length; i++) {
console.log(tmp[i]);
}
console.log(i); // 11
```

由于遍历时定义的 i 会变量提升成为一个全局变量，在函数结束之后不会被销毁，所以打印出来 11。

19. 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。代码执行是基于执行栈的，所以当在一个函数里调用另一个函数时，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，所以这时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。但是 ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

20. ES6 模块与 CommonJS 模块有什么异同?

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，也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。

21. 常见的 DOM 操作有哪些?

1）DOM 节点的获取

DOM 节点的获取的 API 及使用：

getElementById // 按照 id 查询

getElementsByTagName // 按照标签名查询

getElementsByClassName // 按照类名查询

querySelectorAll // 按照 css 选择器查询

2）DOM 节点的创建

创建一个新节点，并把它添加到指定节点的后面。 已知的 HTML 结构如下：

```
<html>
  <head>
```

```
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
    </div>
  </body>
</html>
```

要求添加一个有内容的 span 节点到 id 为 title 的节点后面，做法就是：

```
// 首先获取父节点
var container = document.getElementById('container')
// 创建新节点
var targetSpan = document.createElement('span')
// 设置 span 节点的内容
targetSpan.innerHTML = 'hello world'
// 把新创建的元素塞进父节点里去
container.appendChild(targetSpan)
```

3）DOM 节点的删除

删除指定的 DOM 节点， 已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
    </div>
  </body>
</html>
```

需要删除 id 为 title 的元素，做法是：

```
// 获取目标元素的父元素
var container = document.getElementById('container')
// 获取目标元素
var targetNode = document.getElementById('title')
// 删除目标元素
container.removeChild(targetNode)
```

或者通过子节点数组来完成删除：

```
// 获取目标元素的父元素 var container = document.getElementById('container')// 获取目标元素 var targetNode = container.childNodes[1]// 删除目标元素 container.removeChild(targetNode)
```

4）修改 DOM 元素

修改 DOM 元素这个动作可以分很多维度，比如说移动 DOM 元素的位置，修改 DOM 元素的属性等。

将指定的两个 DOM 元素交换位置， 已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
      <p id="content">我是内容</p>
    </div>
  </body>
</html>
```

现在需要调换 title 和 content 的位置，可以考虑 insertBefore 或者 appendChild：

```
// 获取父元素
var container = document.getElementById('container')
// 获取两个需要被交换的元素
var title = document.getElementById('title')
var content = document.getElementById('content')
// 交换两个元素，把 content 置于 title 前面
container.insertBefore(content, title)
```

22. 元素尺寸?

23. use strict 是什么意思 ？使用它区别是什么?

use strict 是一种 ECMAscript5 添加的（严格模式）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如下：

消除 Javascript 语法的不合理、不严谨之处，减少怪异行为;

消除代码运行的不安全之处，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：

禁止使用 with 语句。

禁止 this 关键字指向全局对象。

对象不能有重名的属性。

24. 如何判断一个对象是否属于某个类?

第一种方式，使用 instanceof 运算符来判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

第二种方式，通过对象的 constructor 属性来判断，对象的 constructor 属性指向该对象的构造函数，但是这种方式不是很安全，因为 constructor 属性可以被改写。

第三种方式，如果需要判断的是某个内置的引用类型的话，可以使用 Object.prototype.toString() 方法来打印对象的[[Class]] 属性来进行判断。

25. 强类型语言和弱类型语言的区别?

强类型语言：强类型语言也称为强类型定义语言，是一种总是强制类型定义的语言，要求变量的使用要严格符合定义，所有变量都必须先定义后使用。Java 和 C++等语言都是强制类型定义的，也就是说，一旦一个变量被指定了某个数据类型，如果不经过强制转换，那么它就永远是这个数据类型了。例如你有一个整数，如果不显式地进行转换，你不能将其视为一个字符串。

弱类型语言：弱类型语言也称为弱类型定义语言，与强类型定义相反。JavaScript 语言就属于弱类型语言。简单理解就是一种变量类型可以被忽略的语言。比如 JavaScript 是弱类型定义的，在 JavaScript 中就可以将字符串'12'和整数 3 进行连接得到字符串'123'，在相加的时候会进行强制类型转换。

两者对比：强类型语言在速度上可能略逊色于弱类型语言，但是强类型语言带来的严谨性可以有效地帮助避免许多错误。

26. 解释性语言和编译型语言的区别?

（1）解释型语言

使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行。是代码在执行时才被解释器一行行动态翻译和执行，而不是在执行之前就完成翻译。解释型语言不需要事先编译，其直接将源代码解释成机器码并立即执行，所以只要某一平台提供了相应的解释器即可运行该程序。其特点总结如下

解释型语言每次运行都需要将源代码解释称机器码并执行，效率较低；

只要平台提供相应的解释器，就可以运行源代码，所以可以方便源程序移植；

JavaScript、Python 等属于解释型语言。

（2）编译型语言

使用专门的编译器，针对特定的平台，将高级语言源代码一次性的编译成可被该平台硬件执行的机器码，并

包装成该平台所能识别的可执行性程序的格式。在编译型语言写的程序执行之前，需要一个专门的编译过程，把源代码编译成机器语言的文件，如 exe 格式的文件，以后要再运行时，直接使用编译结果即可，如直接运行 exe 文件。因为只需编译一次，以后运行时不需要编译，所以编译型语言执行效率高。其特点总结如下：

一次性的编译成平台相关的机器语言文件，运行时脱离开发环境，运行效率高；

与特定平台相关，一般无法移植到其他平台；

C、C++等属于编译型语言。

两者主要区别在于： 前者源程序编译后即可在该平台运行，后者是在运行期间才编译。所以前者运行速度快，后者跨平台性好。

27. for...in 和 for...of 的区别?

for…of 是 ES6 新增的遍历方式，允许遍历一个含有 iterator 接口的数据结构（数组、对象等）并且返回各项的值，和 ES3 中的 for…in 的区别如下

for…of 遍历获取的是对象的键值，for…in 获取的是对象的键名；

for… in 会遍历对象的整个原型链，性能非常差不推荐使用，而 for … of 只遍历当前对象不会遍历原型链；

对于数组的遍历，for…in 会返回数组中所有可枚举的属性(包括原型链上可枚举的属性)，for…of 只返回数组的下标对应的属性值；

总结： for...in 循环主要是为了遍历对象而生，不适用于遍历数组；for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象。

28. ajax、axios、fetch 的区别?

（1）AJAX

Ajax 即“AsynchronousJavascriptAndXML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。它是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点如下：

本身是针对 MVC 编程，不符合前端 MVVM 的浪潮

基于原生 XHR 开发，XHR 本身的架构不清晰

不符合关注分离（Separation of Concerns）的原则

配置和调用方式非常混乱，而且基于事件的异步模型不友好。

（2）Fetch

fetch 号称是 AJAX 的替代品，是在 ES6 出现的，使用了 ES6 中的 promise 对象。Fetch 是基于 promise 设计的。Fetch 的代码结构比起 ajax 简单多。fetch 不是 ajax 的进一步封装，而是原生 js，没有使用 XMLHttpRequest 对象。

fetch 的优点：

语法简洁，更加语义化

基于标准 Promise 实现，支持 async/await

更加底层，提供的 API 丰富（request, response）

脱离了 XHR，是 ES 规范里新的实现方式

fetch 的缺点：

fetch 只对网络请求报错，对 400，500 都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。

fetch 默认不会带 cookie，需要添加配置项： fetch(url, {credentials: 'include'})

fetch 不支持 abort，不支持超时控制，使用 setTimeout 及 Promise.reject 的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费

fetch 没有办法原生监测请求的进度，而 XHR 可以

（3）Axios

Axios 是一种基于 Promise 封装的 HTTP 客户端，其特点如下：

浏览器端发起 XMLHttpRequests 请求

node 端发起 http 请求

支持 Promise API

监听请求和返回

对请求和返回进行转化

取消请求

自动转换 json 数据

客户端支持抵御 XSRF 攻击

29. forEach 和 map 方法有什么区别?

forEach()方法会针对每一个元素执行提供的函数，对数据的操作会改变原数组，该方法没有返回值；

map()方法不会改变原数组的值，返回一个新数组，新数组中的值为原数组调用函数处理之后的值。

30. {}和[]的 valueOf 和 toString 的结果是什么？？

{} 的 valueOf 结果为 {} ，toString 的结果为 "[object Object]"

[] 的 valueOf 结果为 [] ，toString 的结果为 ""。

31. 为什么函数被称为一等公民?

在 JavaScript 中，函数不仅拥有一切传统函数的使用方式（声明和调用），而且可以做到像简单值一样:

赋值（var func = function(){}）、

传参(function func(x,callback){callback();})、

返回(function(){return function(){}})，

这样的函数也称之为第一级函数（First-class Function）。不仅如此，JavaScript 中的函数还充当了类的构造函数的作用，同时又是一个 Function 类的实例(instance)。这样的多重身份让 JavaScript 的函数变得非常重要。

三、JS 原型与原型链

1. 对原型、原型链的理解?

在 JavaScript 中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说不应该能够获取到这个值的，但是现在浏览器中都实现了 proto 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

当访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，

这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。

特点： JavaScript 对象是通过引用来传递的，创建的每个新对象实体中并没有一份属于自己的原型副本。当修改原型时，与之相关的对象也会继承这一改变。

2. 原型的五条规则?

所有的引用类型都可以自定义添加属性

所有的引用类型都有自己的隐式原型（proto）

函数都有自己的显式原型（prototype）

所有的引用类型的隐式原型都指向对应构造函数的显示原型

使用引用类型的某个自定义属性时，如果没有这个属性，会去该引用类型的__proto__（也就是对应构造函数的 prototype）中去找。

3. js 获取原型的方法?

p.proto

p.constructor.prototype

Object.getPrototypeOf(p)

4. 描述 new 一个对象的过程?

创建一个新对象

this 指向这个新对象

执行代码给 this 赋值

return this

5. 原型链的终点是什么？如何打印出原型链的终点?

由于 Object 是构造函数，原型链终点是 Object.prototype.__proto__，而 Object.prototype.__proto__=== null // true，所以，原型链的终点是 null。原型链上的所有原型都是对象，所有的对象最终都是由 Object 构造的，而 Object.prototype 的下一级是 Object.prototype.__proto__。

四、JS 执行上下文/作用域链/闭包

1. 对闭包的理解?

闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包的两个场景

函数作为变量传递

函数作为返回值

闭包有两个常用的用途；

闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。

闭包的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

2. 对作用域、作用域链的理解?

1）全局作用域和函数作用域

（1）全局作用域

最外层函数和最外层函数外面定义的变量拥有全局作用域

所有未定义直接赋值的变量自动声明为全局作用域

所有 window 对象的属性拥有全局作用域

全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突。

（2）函数作用域

函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到

用域是分层的，内层作用域可以访问外层作用域，反之不行

2）块级作用域

使用 ES6 中新增的 let 和 const 指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由{}包裹的代码片段）

let 和 const 声明的变量不会有变量提升，也不可以重复声明

在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部。

作用域链：

自由变量

作用域链，及自由变量的查找，找不到逐级向上找

在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到 window 对象就被终止，这一层层的关系就是作用域链。

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当查找一个变量时，如果当前执行环境中没有找到，可以沿着作用域链向后查找。

3. 对执行上下文的理解?

执行上下文类型

（1）全局执行上下文

任何不在函数内部的都是全局执行上下文，它首先会创建一个全局的 window 对象，并且设置 this 的值等于这个全局对象，一个程序中只有一个全局执行上下文。

（2）函数执行上下文

当一个函数被调用时，就会为该函数创建一个新的执行上下文，函数的上下文可以有任意多个。

（3）eval 函数执行上下文

执行在 eval 函数中的代码会有属于他自己的执行上下文，不过 eval 函数不常使用，不做介绍。

2. 执行上下文栈

JavaScript 引擎使用执行上下文栈来管理执行上下文

当 JavaScript 执行代码时，首先遇到全局代码，会创建一个全局执行上下文并且压入执行栈中，每当遇到一个函数调用，就会为该函数创建一个新的执行上下文并压入栈顶，引擎会执行位于执行上下文栈顶的函数，当函数执行完成之后，执行上下文从栈中弹出，继续执行下一个上下文。当所有的代码都执行完毕之后，从栈中弹出全局执行上下文。

```
let a = 'Hello World!';
function first() {
   console.log('Inside first function');
   second();
   console.log('Again inside first function');
}
function second() {
   console.log('Inside second function');
}
first();
//执行顺序
//先执行 second(),在执行 first()
```

3. 创建执行上下文

创建执行上下文有两个阶段：创建阶段和执行阶段

1）创建阶段

（1）this 绑定

在全局执行上下文中，this 指向全局对象（window 对象）

在函数执行上下文中，this 指向取决于函数如何调用。如果它被一个引用对象调用，那么 this 会被设置成那个对象，否则 this 的值被设置为全局对象或者 undefined

（2）创建词法环境组件

词法环境是一种有标识符——变量映射的数据结构，标识符是指变量/函数名，变量是对实际对象或原始数据的引用。

词法环境的内部有两个组件：加粗样式：环境记录器:用来储存变量个函数声明的实际位置外部环境的引用：可以访问父级作用域

（3）创建变量环境组件

变量环境也是一个词法环境，其环境记录器持有变量声明语句在执行上下文中创建的绑定关系。

2）执行阶段

此阶段会完成对变量的分配，最后执行完代码。

简单来说执行上下文就是指：

在执行一点 JS 代码之前，需要先解析代码。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为 undefined，函数先声明好可使用。这一步执行完了，才开始正式的执行程序。

在一个函数执行之前，也会创建一个函数执行上下文环境，跟全局执行上下文类似，不过函数执行上下文会多出 this、arguments 和函数的参数。

全局上下文：变量定义，函数声明

函数上下文：变量定义，函数声明，this，arguments

4. 什么是执行栈?

执行栈，也就是在其它编程语言中所说的“调用栈”，是一种拥有 LIFO（后进先出）数据结构的栈，被用来存储代码运行时创建的所有执行上下文。

当 JavaScript 引擎第一次遇到你的脚本时，它会创建一个全局的执行上下文并且压入当前执行栈。每当引擎遇到一个函数调用，它会为该函数创建一个新的执行上下文并压入栈的顶部。

引擎会执行那些执行上下文位于栈顶的函数。当该函数执行结束时，执行上下文从栈中弹出，控制流程到达当前栈中的下一个上下文。

让我们通过下面的代码示例来理解：

```
let a = 'Hello World!';
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}

function second() {
  console.log('Inside second function');
}

first();
console.log('Inside Global Execution Context');
```

上述代码的执行上下文栈。

当上述代码在浏览器加载时，JavaScript 引擎创建了一个全局执行上下文并把它压入当前执行栈。当遇到 first() 函数调用时，JavaScript 引擎为该函数创建一个新的执行上下文并把它压入当前执行栈的顶部。

当从 first() 函数内部调用 second() 函数时，JavaScript 引擎为 second() 函数创建了一个新的执行上下文并把它压入当前执行栈的顶部。当 second() 函数执行完毕，它的执行上下文会从当前栈弹出，并且控制流程到达下一个执行上下文，即 first() 函数的执行上下文。

当 first() 执行完毕，它的执行上下文从栈弹出，控制流程到达全局执行上下文。一旦所有代码执行完毕，JavaScript 引擎从当前栈中移除全局执行上下文。

5. 参考

https://juejin.cn/post/6844903682283143181#heading-3

五、JS 事件

1. 三种事件模型是什么?

事件 是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型。

DOM0 级模型： ，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，

它可以在网页中直接定义监听函数，也可以通过 js 属性来指定监听函数。这种方式是所有浏览器都兼容的。

IE 事件模型： 在该事件模型中，一次事件共有两个过程，事件处理阶段，和事件冒泡阶段。事件处理阶段会首先执行目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。

DOM2 级事件模型： 在该事件模型中，一次事件共有三个过程，第一个过程是事件捕获阶段。捕获指的是事件从 document 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，事件绑定的函数是 addEventListener，其中第三个参数可以指定事件是否在捕获阶段执行。

2. 事件委托是什么?

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。

3. 什么是事件传播?

当事件发生在 DOM 元素上时，该事件并不完全发生在那个元素上。在“当事件发生在 DOM 元素上时，该事件并不完全发生在那个元素上。

事件传播有三个阶段：

捕获阶段 - 事件从 window 开始，然后向下到每个元素，直到到达目标元素事件或 event.target。

目标阶段 - 事件已达到目标元素。

冒泡阶段 - 事件从目标元素冒泡，然后上升到每个元素，直到到达 window。

4. 什么是事件捕获?

当事件发生在 DOM 元素上时，该事件并不完全发生在那个元素上。在捕获阶段，事件从 window 开始，一直到触发事件的元素。window----> document----> html----> body ---->目标元素

addEventListener 方法具有第三个可选参数 useCapture，其默认值为 false，事件将在冒泡阶段中发生，如果为 true，则事件将在捕获阶段中发生。如果单击 child 元素，它将分别在控制台上打印 window，document，html，grandparent 和 parent，这就是事件捕获。

5. 什么是事件冒泡?

事件冒泡刚好与事件捕获相反，当前元素---->body ----> html---->document ---->window。当事件发生在 DOM 元素上时，该事件并不完全发生在那个元素上。在冒泡阶段，事件冒泡，或者事件发生在它的父代，祖父母，祖父母的父代，直到到达 window 为止。

addEventListener 方法具有第三个可选参数 useCapture，其默认值为 false，事件将在冒泡阶段中发生，如果为 true，则事件将在捕获阶段中发生。如果单击 child 元素，它将分别在控制台上打印 child，parent，grandparent，html，document 和 window，这就是事件冒泡。

6. 一个 DOM 元素绑定多个事件时，先执行冒泡还是捕获?

https://blog.csdn.net/u013217071/article/details/77613706

六、JS 指针 this/call/apply/bind

1. 对 this 对象的理解?

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模式来判断。

第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。

第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。

第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

这四种方式，使用构造器调用模式的优先级最高，然后是 apply、call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

2. call() apply() 和 bind() 的区别?

https://juejin.cn/post/6844903906279964686

https://juejin.cn/post/6844903906279964686#heading-9

call/apply 的区别

它们的作用一模一样，区别仅在于传入参数的形式的不同。

apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。

call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

call/apply 与 bind 的区别

执行：

call/apply 改变了函数的 this 上下文后马上执行该函数

bind 则是返回改变了上下文后的函数,不执行该函数

返回值:

call/apply 返回 fun 的执行结果

bind 返回 fun 的拷贝，并指定了 fun 的 this 指向，保存了 fun 的参数。

3. 手写 call、apply 及 bind 函数?

call 函数的实现步骤：

1.判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。

2.判断传入上下文对象是否存在，如果不存在，则设置为 window 。

3.处理传入的参数，截取第一个参数后的所有参数。

4.将函数作为上下文对象的一个属性。

5.使用上下文对象来调用这个方法，并保存返回结果。

6.删除刚才新增的属性。

7.返回结果。

```
// 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 函数的实现步骤：

判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。

判断传入上下文对象是否存在，如果不存在，则设置为 window 。

将函数作为上下文对象的一个属性。

判断参数值是否传入

使用上下文对象来调用这个方法，并保存返回结果。

删除刚才新增的属性

返回结果

```
// 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 函数的实现步骤：

1.判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。

2.保存当前函数的引用，获取其余传入参数值。

3.创建一个函数返回

4.函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指定的上下文对象。

```
// 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)
        );
```

```
    };
};
```

七、JS 异步编程

1. 前端使用异步的场景?

同步会阻塞代码，但是异步不会

定时任务：setTimeout，setInterval

网络请求：ajax 请求，动态 img 加载

事件绑定

需要等待的情况下都需要异步，因为不会像同步一样阻塞

2. 异步编程的实现方式?

JavaScript 中的异步机制可以分为以下几种：

回调函数 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

generator 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

async 函数 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

3. 对 Promise 的理解?

Promise 是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息，他的出现大大改善了异步编程的困境，避免了地狱回调，它比传统的解决方案回调函数和事件更合理和更强大。

所谓 Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

（1）Promise 的实例有三个状态:

Pending（进行中）

Resolved（已完成）

Rejected（已拒绝）

当把一件事情交给 promise 时，它的状态就是 Pending，任务完成了状态就变成了 Resolved、没有完成失败了就变成了 Rejected。

（2）Promise 的实例有两个过程：

pending -> fulfilled : Resolved（已完成）

pending -> rejected：Rejected（已拒绝）

注意：一旦从进行状态变成为其他状态就永远不能更改状态了。

Promise 的特点：

对象的状态不受外界影响。promise 对象代表一个异步操作，有三种状态，pending（进行中）、fulfilled（已成功）、rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，这也是 promise 这个名字的由来——“承诺”；

一旦状态改变就不会再变，任何时候都可以得到这个结果。promise 对象的状态改变，只有两种可能：从 pending 变为 fulfilled，从 pending 变为 rejected。这时就称为 resolved（已定型）。如果改变已经发生了，你再对 promise 对象添加回调函数，也会立即得到这个结果。这与事件（event）完全不同，事件的特点是：如果你错过了它，再去监听是得不到结果的。

Promise 的缺点：

无法取消 Promise，一旦新建它就会立即执行，无法中途取消。

如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。

当处于 pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

总结：

Promise 对象是异步编程的一种解决方案，最早由社区提出。Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由 pending 转变 resolved 或者 rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。

状态的改变是通过 resolve() 和 reject() 函数来实现的，可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

注意： 在构造 Promise 的时候，构造函数内部的代码是立即执行的。

4. setTimeout、Promise、Async/Await 的区别?

（1）setTimeout

```
console.log('script start')//1. 打印 script start
setTimeout(function(){
    console.log('settimeout')      // 4. 打印 settimeout
})    // 2. 调用 setTimeout 函数，并定义其完成后执行的回调函数
console.log('script end') //3. 打印 script start
// 输出顺序：script start->script end->settimeout
```

（2）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，当主栈完成后，才会去调用 resolve/reject 中存放的方法执行，打印 p 的时候，是打印的返回结果，一个 Promise 实例。

```
console.log('script start')
```

```
let promise1 = new Promise(function (resolve) {
    console.log('promise1')
    resolve()
    console.log('promise1 end')
}).then(function () {
    console.log('promise2')
})
setTimeout(function(){
    console.log('settimeout')
})
console.log('script end')
// 输出顺序: script start->promise1->promise1 end->script end->promise2->settimeout
```

当 JS 主线程执行到 Promise 对象时：

promise1.then() 的回调就是一个 task

promise1 是 resolved 或 rejected: 那这个 task 就会放入当前事件循环回合的 microtask queue

promise1 是 pending:这个 task 就会放入事件循环的未来的某个(可能下一个)回合的 microtask queue 中

setTimeout 的回调也是个 task ，它会被放入 macrotask queue 即使是 0ms 的情况

（3）async/await

```
async function async1(){
    console.log('async1 start');
    await async2();
    console.log('async1 end')
}
async function async2(){
    console.log('async2')
}
console.log('script start');
async1();
console.log('script end')
// 输出顺序：script start->async1 start->async2->script end->async1 end
```

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。运行结果其实就是一个 Promise 对象。因此也可以使用 then 来处理后续逻辑。

await 的含义为等待，也就是 async 函数需要等待 await 后的函数执行完成并且有了返回结果（Promise 对象）之后，才能继续执行下面的代码。await 通过返回一个 Promise 对象来实现同步的效果。

5. Promise 的基本用法?

（1）创建 Promise 对象

Promise 对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。

Promise 构造函数接受一个函数作为参数，该函数的两个参数分别是 resolve 和 reject。

```
const promise = new Promise(function(resolve, reject) {
  // ... some code
  if (/*  异步操作成功  */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

一般情况下都会使用 new Promise()来创建 promise 对象，但是也可以使用 promise.resolve 和 promise.reject 这两个方法：

Promise.resolve

Promise.resolve(value)的返回值也是一个 promise 对象，可以对返回值进行.then 调用，代码如下：

```
Promise.resolve(11).then(function(value){
  console.log(value); //  打印出 11
});
```

resolve(11)代码中，会让 promise 对象进入确定(resolve 状态)，并将参数 11 传递给后面的 then 所指定的 onFulfilled 函数；

创建 promise 对象可以使用 new Promise 的形式创建对象，也可以使用 Promise.resolve(value)的形式创建 promise 对象；

Promise.reject

Promise.reject 也是 new Promise 的快捷形式，也创建一个 promise 对象。代码如下：

```
Promise.reject(new Error(“我错了，请原谅俺！！”));
```

就是下面的代码 new Promise 的简单形式：

```
new Promise(function(resolve,reject){
    reject(new Error("我错了！"));
});
```

下面是使用 resolve 方法和 reject 方法：

```
function testPromise(ready) {
  return new Promise(function(resolve,reject){
    if(ready) {
      resolve("hello world");
    }else {
      reject("No thanks");
    }
  });
};
// 方法调用
testPromise(true).then(function(msg){
```

```
    console.log(msg);
},function(error){
    console.log(error);
});
```

上面的代码的含义是给 testPromise 方法传递一个参数，返回一个 promise 对象，如果为 true 的话，那么调用 promise 对象中的 resolve()方法，并且把其中的参数传递给后面的 then 第一个函数内，因此打印出 “hello world”, 如果为 false 的话，会调用 promise 对象中的 reject()方法，则会进入 then 的第二个函数内，会打印 No thanks；

（2）Promise 方法

Promise 有五个常用的方法：then()、catch()、all()、race()、finally。下面就来看一下这些方法。

then()

当 Promise 执行的内容符合成功条件时，调用 resolve 函数，失败就调用 reject 函数。Promise 创建完了，那该如何调用呢？

```
promise.then(function(value) {
    // success
}, function(error) {
    // failure
});
```

then 方法可以接受两个回调函数作为参数。第一个回调函数是 Promise 对象的状态变为 resolved 时调用，第二个回调函数是 Promise 对象的状态变为 rejected 时调用。其中第二个参数可以省略。

then 方法返回的是一个新的 Promise 实例（不是原来那个 Promise 实例）。因此可以采用链式写法，即 then 方法后面再调用另一个 then 方法。

当要写有顺序的异步事件时，需要串行时，可以这样写：

```
let promise = new Promise((resolve,reject)=>{
    ajax('first').success(function(res){
        resolve(res);
    })
})
promise.then(res=>{
    return new Promise((resovle,reject)=>{
        ajax('second').success(function(res){
            resolve(res)
        })
    })
}).then(res=>{
    return new Promise((resovle,reject)=>{
        ajax('second').success(function(res){
            resolve(res)
        })
    })
```

```
}).then(res=>{


})
```

那当要写的事件没有顺序或者关系时，还如何写呢？可以使用 all 方法来解决。

2. catch()

Promise 对象除了有 then 方法，还有一个 catch 方法，该方法相当于 then 方法的第二个参数，指向 reject 的回调函数。不过 catch 方法还有一个作用，就是在执行 resolve 回调函数时，如果出现错误，抛出异常，不会停止运行，而是进入 catch 方法中。

```
p.then((data) => {
        console.log('resolved',data);
},(err) => {
        console.log('rejected',err);
        }
);
p.then((data) => {
      console.log('resolved',data);
}).catch((err) => {
      console.log('rejected',err);
});
```

3. all()

all 方法可以完成并行任务， 它接收一个数组，数组的每一项都是一个 promise 对象。当数组中所有的 promise 的状态都达到 resolved 的时候，all 方法的状态就会变成 resolved，如果有一个状态变成了 rejected，那么 all 方法的状态就会变成 rejected。

```
javascript
let promise1 = new Promise((resolve,reject)=>{
setTimeout(()=>{
            resolve(1);
},2000)
});
let promise2 = new Promise((resolve,reject)=>{
setTimeout(()=>{
            resolve(2);
},1000)
});
let promise3 = new Promise((resolve,reject)=>{
setTimeout(()=>{
            resolve(3);
},3000)
});
```

```
Promise.all([promise1,promise2,promise3]).then(res=>{
    console.log(res);
    //结果为：[1,2,3]
})
```

调用 all 方法时的结果成功的时候是回调函数的参数也是一个数组，这个数组按顺序保存着每一个 promise 对象 resolve 执行时的值。

（4）race()

race 方法和 all 一样，接受的参数是一个每项都是 promise 的数组，但是与 all 不同的是，当最先执行完的事件执行完之后，就直接返回该 promise 对象的值。如果第一个 promise 对象状态变成 resolved，那自身的状态变成了 resolved；反之第一个 promise 变成 rejected，那自身状态就会变成 rejected。

```
let promise1 = new Promise((resolve,reject)=>{
setTimeout(()=>{
        reject(1);
},2000)
});
let promise2 = new Promise((resolve,reject)=>{
setTimeout(()=>{
        resolve(2);
},1000)
});
let promise3 = new Promise((resolve,reject)=>{
setTimeout(()=>{
        resolve(3);
},3000)
});
Promise.race([promise1,promise2,promise3]).then(res=>{
console.log(res);
//结果：2
},rej=>{
    console.log(rej)};
)
```

那么 race 方法有什么实际作用呢？当要做一件事，超过多长时间就不做了，可以用这个方法来解决：

```
Promise.race([promise1,timeOutPromise(5000)]).then(res=>{})
```

5. finally()

finally 方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

```
promise
.then(result => {・・・})
.catch(error => {・・・})
.finally(() => {・・・});
```

上面代码中，不管 promise 最后的状态，在执行完 then 或 catch 指定的回调函数以后，都会执行 finally 方法指定的回调函数。

下面是一个例子，服务器使用 Promise 处理请求，然后使用 finally 方法关掉服务器。

```
server.listen(port)
  .then(function () {
    // ...
  })
  .finally(server.stop);
```

finally 方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是 fulfilled 还是 rejected。这表明，finally 方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。finally 本质上是 then 方法的特例：

```
promise
.finally(() => {
  // 语句
});
// 等同于
promise
.then(
  result => {
    // 语句
    return result;
  },
  error => {
    // 语句
    throw error;
  }
);
```

上面代码中，如果不使用 finally 方法，同样的语句需要为成功和失败两种情况各写一次。有了 finally 方法，则只需要写一次。

6. Promise 解决了什么问题?

在工作中经常会碰到这样一个需求，比如我使用 ajax 发一个 A 请求后，成功后拿到数据，需要把数据传给 B 请求；那么需要如下编写代码：

```
let fs = require('fs')
fs.readFile('./a.txt','utf8',function(err,data){
  fs.readFile(data,'utf8',function(err,data){
    fs.readFile(data,'utf8',function(err,data){
      console.log(data)
    })
  })
```

```
})
```

上面的代码有如下缺点：

后一个请求需要依赖于前一个请求成功后，将数据往下传递，会导致多个 ajax 请求嵌套的情况，代码不够直观。

如果前后两个请求不需要传递参数的情况下，那么后一个请求也需要前一个请求成功后再执行下一步操作，这种情况下，那么也需要如上编写代码，导致代码不够直观。

Promise 出现之后，代码变成这样：

```
let fs = require('fs')
function read(url){
   return new Promise((resolve,reject)=>{
      fs.readFile(url,'utf8',function(error,data){
         error && reject(error)
         resolve(data)
      })
   })
}
read('./a.txt').then(data=>{
   return read(data)
}).then(data=>{
   return read(data)
}).then(data=>{
   console.log(data)
})
```

这样代码看起了就简洁了很多，解决了地狱回调的问题。

7. Promise.all 和 Promise.race 的区别的使用场景?

（1）Promise.all

Promise.all 可以将多个 Promise 实例包装成一个新的 Promise 实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被 reject 失败状态的值。

Promise.all 中传入的是数组，返回的也是是数组，并且会将进行映射，传入的 promise 对象返回的值是按照顺序在数组中排列的，但是注意的是他们执行的顺序并不是按照顺序的，除非可迭代对象为空。

需要注意，Promise.all 获得的成功结果的数组里面的数据顺序和 Promise.all 接收到的数组顺序是一致的，这样当遇到发送多个请求并根据请求顺序获取和使用数据的场景，就可以使用 Promise.all 来解决。

（2）Promise.race

顾名思义，Promse.race 就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。当要做一件事，超过多长时间就不做了，可以用这个方法来解决：

```
Promise.race([promise1,timeOutPromise(5000)]).then(res=>{})
```

8. 对 async/await 的理解?

async/await 其实是 Generator 的语法糖，它能实现的效果都能用 then 链来实现，它是为优化 then 链而开发

出来的。从字面上来看，async 是“异步”的简写，await 则为等待，所以很好理解 async 用于申明一个 function 是异步的，而 await 用于等待一个异步方法执行完成。当然语法上强制规定 await 只能出现在 asnyc 函数中，先来看看 async 函数返回了什么：

```
async function testAsy(){
    return 'hello world';
}
let result = testAsy();
console.log(result)
```

所以，async 函数返回的是一个 Promise 对象。async 函数（包含函数语句、函数表达式、Lambda 表达式）会返回一个 Promise 对象，如果在函数中 return 一个直接量，async 会把这个直接量通过 Promise.resolve() 封装成 Promise 对象。

async 函数返回的是一个 Promise 对象，所以在最外层不能用 await 获取其返回值的情况下，当然应该用原来的方式：then() 链来处理这个 Promise 对象，就像这样：

```
async function testAsy(){
    return 'hello world'
}
let result = testAsy()
console.log(result)
result.then(v=>{
      console.log(v)     // hello world
})
```

那如果 async 函数没有返回值，又该如何？很容易想到，它会返回 Promise.resolve(undefined)。

联想一下 Promise 的特点——无等待，所以在没有 await 的情况下执行 async 函数，它会立即执行，返回一个 Promise 对象，并且，绝不会阻塞后面的语句。这和普通返回 Promise 对象的函数并无二致。

注意：Promise.resolve(x) 可以看作是 new Promise(resolve => resolve(x)) 的简写，可以用于快速封装字面量对象或其他对象，将其封装成 Promise 实例。

9. await 到底在等啥?

await 在等待什么呢？ 一般来说，都认为 await 是在等待一个 async 函数完成。不过按语法说明，await 等待的是一个表达式，这个表达式的计算结果是 Promise 对象或者其它值（换句话说，就是没有特殊限定）。

因为 async 函数返回一个 Promise 对象，所以 await 可以用于等待一个 async 函数的返回值——这也可以说是 await 在等 async 函数，但要清楚，它等的实际是一个返回值。注意到 await 不仅仅用于等 Promise 对象，它可以等任意表达式的结果，所以，await 后面实际是可以接普通函数调用或者直接量的。所以下面这个示例完全可以正确运行：

```
function getSomething() {
      return "something";
}
async function testAsync() {
      return Promise.resolve("hello async");
}
```

```
async function test() {
    const v1 = await getSomething();
    const v2 = await testAsync();
    console.log(v1, v2);
}
test();
```

await 表达式的运算结果取决于它等的是什么。

如果它等到的不是一个 Promise 对象，那 await 表达式的运算结果就是它等到的东西。

如果它等到的是一个 Promise 对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果。

来看一个例子：

```
function testAsy(x){
    return new Promise(resolve=>{setTimeout(() => {
        resolve(x);
      }, 3000)
     }
    )
}
async function testAwt(){
   let result =   await testAsy('hello world');
   console.log(result);       // 3 秒钟之后出现 hello world
   console.log('cuger')     // 3 秒钟之后出现 cug
}
testAwt();
console.log('cug')   //立即输出 cug
```

这就是 await 必须用在 async 函数中的原因。async 函数调用不会造成阻塞，它内部所有的阻塞都被封装在一个 Promise 对象中异步执行。await 暂停当前 async 的执行，所以'cug''最先输出，hello world'和‘cuger’是 3 秒钟后同时出现的。

10. async/await 的优势?

单一的 Promise 链并不能发现 async/await 的优势，但是，如果需要处理由多个 Promise 组成的 then 链的时候，优势就能体现出来了（很有意思，Promise 通过 then 链来解决多层回调的问题，现在又用 async/await 来进一步优化它）。

假设一个业务，分多个步骤完成，每个步骤都是异步的，而且依赖于上一个步骤的结果。仍然用 setTimeout 来模拟异步操作：

```
/**
 * 传入参数 n，表示这个函数执行的时间（毫秒）
 * 执行的结果是 n + 200，这个值将用于下一步骤
 */
function takeLongTime(n) {
```

```
    return new Promise(resolve => {
        setTimeout(() => resolve(n + 200), n);
    });
}
function step1(n) {
    console.log(`step1 with ${n}`);
    return takeLongTime(n);
}
function step2(n) {
    console.log(`step2 with ${n}`);
    return takeLongTime(n);
}
function step3(n) {
    console.log(`step3 with ${n}`);
    return takeLongTime(n);
}
```

现在用 Promise 方式来实现这三个步骤的处理：

```
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
            console.timeEnd("doIt");
        });
}
doIt();
// c:\var\test>node --harmony_async_await .
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
// doIt: 1507.251ms
```

输出结果 result 是 step3() 的参数 700 + 200 = 900。doIt() 顺序执行了三个步骤，一共用了 300 + 500 + 700 = 1500 毫秒，和 console.time()/console.timeEnd() 计算的结果一致。

如果用 async/await 来实现呢，会是这样：

```
async function doIt() {
```

```
    console.time("doIt");
    const time1 = 300;
    const time2 = await step1(time1);
    const time3 = await step2(time2);
    const result = await step3(time3);
    console.log(`result is ${result}`);
    console.timeEnd("doIt");
}
doIt();
```

结果和之前的 Promise 实现是一样的，但是这个代码看起来是不是清晰得多，几乎跟同步代码一样。

11. async/await 对比 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. async/await 如何捕获异常?

```
async function fn(){
    try{
        let a = await Promise.reject('error')
    }catch(error){
        console.log(error)
    }
}
```

13. 并发与并行的区别?

并发是宏观概念，我分别有任务 A 和任务 B，在一段时间内通过任务间的切换完成了这两个任务，这种情况就可以称之为并发。

并行是微观概念，假设 CPU 中存在两个核心，那么我就可以同时完成任务 A、B。同时完成多个任务的情况就可以称之为并行。

14. 什么是回调函数？回调函数有什么缺点？如何解决回调地狱问题?

以下代码就是一个回调函数的例子：

```
ajax(url, () => {
    // 处理逻辑
})
```

回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell）。假设多个请求存在依赖性，可能会有如下代码：

```
ajax(url, () => {
    // 处理逻辑
```

```
    ajax(url1, () => {
        // 处理逻辑
        ajax(url2, () => {
            // 处理逻辑
        })
    })
})
```

以上代码看起来不利于阅读和维护，当然，也可以把函数分开来写：

```
function firstAjax() {
  ajax(url1, () => {
    // 处理逻辑
    secondAjax()
  })
}
function secondAjax() {
  ajax(url2, () => {
    // 处理逻辑
  })
}
ajax(url, () => {
  // 处理逻辑
  firstAjax()
})
```

以上的代码虽然看上去利于阅读了，但是还是没有解决根本问题。回调地狱的根本问题就是：

嵌套函数存在耦合性，一旦有所改动，就会牵一发而动全身

嵌套函数一多，就很难处理错误

当然，回调函数还存在着别的几个缺点，比如不能使用 try catch 捕获错误，不能直接 return。

15. setTimeout、setInterval、requestAnimationFrame 各有什么特点?

异步编程当然少不了定时器了，常见的定时器函数有 setTimeout、setInterval、requestAnimationFrame。最常用的是 setTimeout，很多人认为 setTimeout 是延时多久，那就应该是多久后执行。

其实这个观点是错误的，因为 JS 是单线程执行的，如果前面的代码影响了性能，就会导致 setTimeout 不会按期执行。当然了，可以通过代码去修正 setTimeout，从而使定时器相对准确：

```
let period = 60 * 1000 * 60 * 2
let startTime = new Date().getTime()
let count = 0
let end = new Date().getTime() + period
let interval = 1000
let currentInterval = interval
function loop() {
```

```
    count++
    // 代码执行所消耗的时间
    let offset = new Date().getTime() - (startTime + count * interval);
    let diff = end - new Date().getTime()
    let h = Math.floor(diff / (60 * 1000 * 60))
    let hdiff = diff % (60 * 1000 * 60)
    let m = Math.floor(hdiff / (60 * 1000))
    let mdiff = hdiff % (60 * 1000)
    let s = mdiff / (1000)
    let sCeil = Math.ceil(s)
    let sFloor = Math.floor(s)
    // 得到下一次循环所消耗的时间
    currentInterval = interval - offset
    console.log('时：'+h, '分：'+m, '毫秒：'+s, '秒向上取整：'+sCeil, '代码执行时间：'+offset, '下次循环间隔'+currentInterval)
    setTimeout(loop, currentInterval)
  }
  setTimeout(loop, currentInterval)
```

接下来看 setInterval，其实这个函数作用和 setTimeout 基本一致，只是该函数是每隔一段时间执行一次回调函数。

通常来说不建议使用 setInterval。第一，它和 setTimeout 一样，不能保证在预期的时间执行任务。第二，它存在执行累积的问题，请看以下伪代码

```
function demo() {
  setInterval(function(){
    console.log(2)
  },1000)
  sleep(2000)
}
demo()
```

以上代码在浏览器环境中，如果定时器执行过程中出现了耗时操作，多个回调函数会在耗时操作结束以后同时执行，这样可能就会带来性能上的问题。

如果有循环定时器的需求，其实完全可以通过 requestAnimationFrame 来实现：

```
function setInterval(callback, interval) {
  let timer
  const now = Date.now
  let startTime = now()
  let endTime = startTime
  const loop = () => {
    timer = window.requestAnimationFrame(loop)
```

```
        endTime = now()
        if (endTime - startTime >= interval) {
            startTime = endTime = now()
            callback(timer)
        }
    }
    timer = window.requestAnimationFrame(loop)
    return timer
}
let a = 0
setInterval(timer => {
    console.log(1)
    a++
    if (a === 3) cancelAnimationFrame(timer)
}, 1000)
```

首先 requestAnimationFrame 自带函数节流功能，基本可以保证在 16.6 毫秒内只执行一次（不掉帧的情况下），并且该函数的延时效果是精确的，没有其他定时器时间不准的问题，当然你也可以通过该函数来实现 setTimeout。

16. 对 Promise 的理解?

Promise 是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息，他的出现大大改善了异步编程的困境，避免了地狱回调，它比传统的解决方案回调函数和事件更合理和更强大。

八、JS 面向对象

1. 对象创建的方式有哪些?

一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js 和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是可以使用函数来进行模拟，从而产生出可复用的对象创建方式，常见的有以下几种：

（1）第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

（2）第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

（3）第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了

通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

（4）第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

（5）第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

（6）第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

2. 对象继承的方式有哪些?

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

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

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

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

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

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

九、JS 垃圾回收与内存泄漏

3. 浏览器的垃圾回收机制?

（1）垃圾回收的概念

垃圾回收：JavaScript 代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

回收机制：

Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。

JavaScript 中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。

不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

（2）垃圾回收的方式

浏览器通常使用的垃圾回收方法有两种：标记清除，引用计数。

1）标记清除

标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，因为他们正在被使用。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。

垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

2）引用计数

另外一种垃圾回收机制就是引用计数，这个用的相对较少。引用计数就是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是 1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减 1。当这个引用次数变为 0 时，说明这个变量已经没有价值，因此，在在机回收期下次再运行时，这个变量所占有的内存空间就会被释放出来。

这种方法会引起循环引用的问题：例如： obj1 和 obj2 通过属性进行相互引用，两个对象的引用次数都是 2。当使用循环计数时，由于函数执行完后，两个对象都离开作用域，函数执行结束，obj1 和 obj2 还将会继续存在，因此它们的引用次数永远不会是 0，就会引起循环引用。

```
function fun() {
    let obj1 = {};
    let obj2 = {};
    obj1.a = obj2; // obj1  引用  obj2
    obj2.a = obj1; // obj2  引用  obj1
}
```

这种情况下，就要手动释放变量占用的内存：

```
    obj1.a =    null
    obj2.a =    null
```

（3）减少垃圾回收

虽然浏览器可以进行垃圾自动回收，但是当代码比较复杂时，垃圾回收所带来的代价比较大，所以应该尽量减少垃圾回收。

对数组进行优化： 在清空一个数组时，最简单的方法就是给其赋值为[ ]，但是与此同时会创建一个新的空对象，可以将数组的长度设置为 0，以此来达到清空数组的目的。

对 object 进行优化： 对象尽量复用，对于不再使用的对象，就将其设置为 null，尽快被回收。

对函数进行优化： 在循环中的函数表达式，如果可以复用，尽量放在函数的外面。

4. 哪些情况会导致内存泄漏?

以下四种情况会造成内存的泄漏：

意外的全局变量： 由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。

被遗忘的计时器或回调函数： 设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。

脱离 DOM 的引用： 获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。

闭包： 不合理的使用闭包，从而导致某些变量一直被留在内存当中。

5. 参考

https://juejin.cn/post/6844903917986267143#heading-7

十、ES6 & ES6+

6. 参考

https://juejin.cn/post/6995334897065787422#heading-34

第四章 框架知识

1. MVVM、MVC、MVP 有什么区别?

https://blog.csdn.net/xx326664162/article/details/50478335

https://blog.csdn.net/victoryzn/article/details/78392128

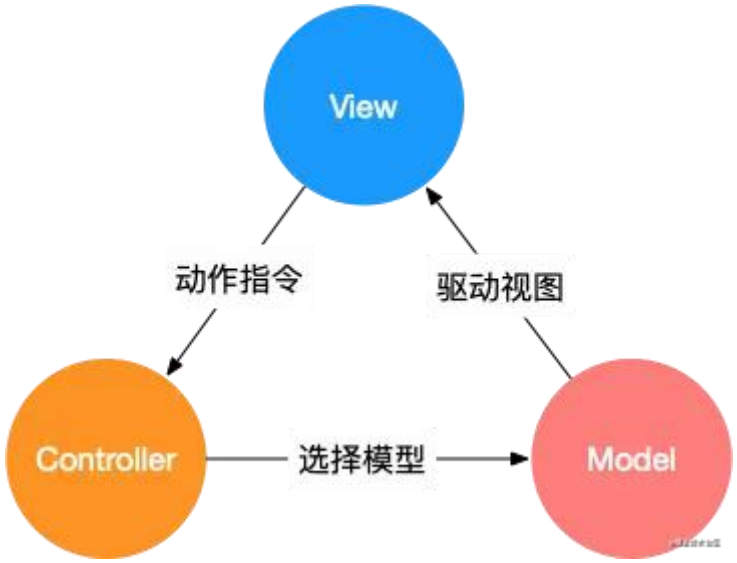
https://www.jianshu.com/p/ff6de219f988

MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化开发效率。

在开发单页面应用时，往往一个路由页面对应了一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，如果项目变得复杂，那么整个文件就会变得冗长、混乱，这样对项目开发和后期的项目维护是非常不利的。

2. MVC 是什么?

MVC 通过分离 Model、View 和 Controller 的方式来组织代码结构。其中 View 负责页面的显示逻辑，Model 负责存储页面的业务数据，以及对相应数据的操作。并且 View 和 Model 应用了观察者模式，当 Model 层发生改变的时候它会通知有关 View 层更新页面。Controller 层是 View 层和 Model 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，Controller 中的事件触发器就开始工作了，通过调用 Model 层，来完成对 Model 的修改，然后 Model 层再去通知 View 层更新。



MVC 优点：

耦合性低，视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码。

重用性高

生命周期成本低

MVC 使开发和维护用户接口的技术含量降低

可维护性高，分离视图层和业务逻辑层也使得 WEB 应用更易于维护和修改

部署快

MVC 缺点：

不适合小型，中等规模的应用程序，花费大量时间将 MVC 应用到规模并不是很大的应用程序通常会得不偿失。

视图与控制器间过于紧密连接，视图与控制器是相互分离，但却是联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。

视图对模型数据的低效率访问，依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

MVC 应用：

在 web app 流行之初， MVC 就应用在了 java（struts2）和 C#（ASP.NET）服务端应用中，后来在客户端应用程序中，基于 MVC 模式，AngularJS 应运而生。

3. MVVM 是什么?

MVVM 分为 Model、View、ViewModel：

Model 代表数据模型，数据和业务逻辑都在 Model 层中定义；

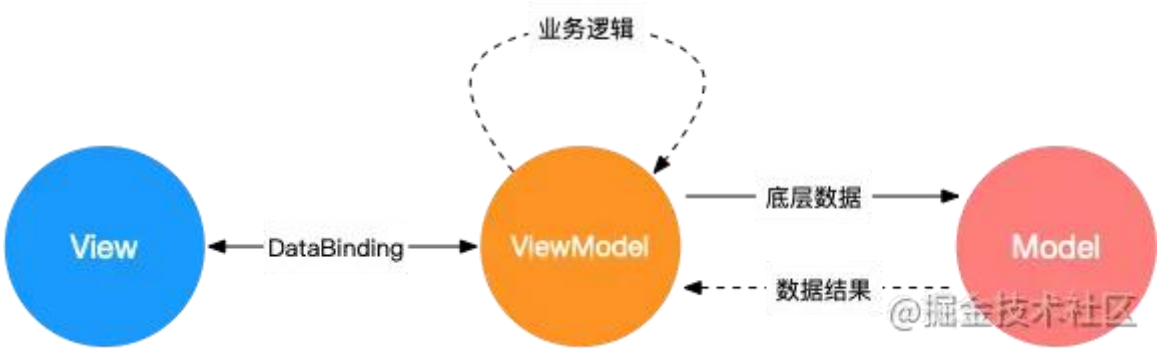
View 代表 UI 视图，负责数据的展示；

ViewModel 负责监听 Model 中数据的改变并且控制视图的更新，处理用户交互操作；

Model 和 View 并无直接关联，而是通过 ViewModel 来进行联系的，Model 和 ViewModel 之间有着双向数据绑

定的联系。因此当 Model 中的数据改变时会触发 View 层的刷新，View 中由于用户交互操作而改变的数据也会在 Model 中同步。

这种模式实现了 Model 和 View 的数据自动同步，因此开发者只需要专注于数据的维护操作即可，而不需要自己操作 DOM。



MVVM 优点：

MVVM 模式和 MVC 模式类似，主要目的是分离视图（View）和模型（Model），有几大优点：

低耦合，视图（View）可以独立于 Model 变化和修改，一个 ViewModel 可以绑定到不同的”View”上，当 View 变化的时候 Model 可以不变，当 Model 变化的时候 View 也可以不变。

可重用性，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。

独立开发，开发人员可以专注于业务逻辑和数据的开发（ViewModel），设计人员可以专注于页面设计，使用 Expression Blend 可以很容易设计界面并生成 xml 代码。

可测试，界面向来是比较难于测试的，而现在测试可以针对 ViewModel 来写。

4. MVP 是什么?

MVP 模式与 MVC 唯一不同的在于 Presenter 和 Controller。在 MVC 模式中使用观察者模式，来实现当 Model 层数据发生变化的时候，通知 View 层的更新。这样 View 层和 Model 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。MVP 的模式通过使用 Presenter 来实现对 View 层和 Model 层的解耦。MVC 中的 Controller 只知道 Model 的接口，因此它没有办法控制 View 层的更新，MVP 模式中，View 层的接口暴露给了 Presenter 因此可以在 Presenter 中将 Model 的变化和 View 的变化绑定在一起，以此来实现 View 和 Model 的同步更新。这样就实现了对 View 和 Model 的解耦，Presenter 还包含了其他的响应逻辑。

MVP 优点：

模型与视图完全分离，我们可以修改视图而不影响模型；

可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter 内部；

我们可以将一个 Presenter 用于多个视图，而不需要改变 Presenter 的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁；

如果我们把逻辑放在 Presenter 中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）。

MVP 缺点：

视图和 Presenter 的交互会过于频繁，使得他们的联系过于紧密。也就是说，一旦视图变更了，presenter 也要变更。

MVP 应用：

可应用与 Android 开发

5. React 和 Vue 的比较?

React

React 起源于 Facebook 的内部项目，用来架设 Instagram 的网站， 并于 2013 年 5 月开源。React 拥有较高的性能，代码逻辑非常简单，越来越多的人已开始关注和使用它。它有以下的特性：

1.声明式设计：React 采用声明范式，可以轻松描述应用。

2.高效：React 通过对 DOM 的模拟，最大限度地减少与 DOM 的交互。

3.灵活：React 可以与已知的库或框架很好地配合。

优点：

1. 速度快：在 UI 渲染过程中，React 通过在虚拟 DOM 中的微操作来实现对实际 DOM 的局部更新。

2. 跨浏览器兼容：虚拟 DOM 帮助我们解决了跨浏览器问题，它为我们提供了标准化的 API，甚至在 IE8 中都是没问题的。

3. 模块化：为你程序编写独立的模块化 UI 组件，这样当某个或某些组件出现问题是，可以方便地进行隔离。

4. 单向数据流：Flux 是一个用于在 JavaScript 应用中创建单向数据层的架构，它随着 React 视图库的开发而被 Facebook 概念化。

5. 同构、纯粹的 javascript：因为搜索引擎的爬虫程序依赖的是服务端响应而不是 JavaScript 的执行，预渲染你的应用有助于搜索引擎优化。

6.兼容性好：比如使用 RequireJS 来加载和打包，而 Browserify 和 Webpack 适用于构建大型应用。它们使得那些艰难的任务不再让人望而生畏。

缺点：

React 本身只是一个 V 而已，并不是一个完整的框架，所以如果是大型项目想要一套完整的框架的话，基本都需要加上 ReactRouter 和 Flux 才能写大型应用。

Vue

Vue 是尤雨溪编写的一个构建数据驱动的 Web 界面的库，准确来说不是一个框架，它聚焦在 V（view）视图层。

它有以下的特性：

1.轻量级的框架

2.双向数据绑定

3.指令

4.插件化

优点：

1. 简单：官方文档很清晰，比 Angular 简单易学。

2. 快速：异步批处理方式更新 DOM。

3. 组合：用解耦的、可复用的组件组合你的应用程序。

4. 紧凑：~18kb min+gzip，且无依赖。

5. 强大：表达式 无需声明依赖的可推导属性 (computed properties)。

6. 对模块友好：可以通过 NPM、Bower 或 Duo 安装，不强迫你所有的代码都遵循 Angular 的各种规定，使用场景更加灵活。

缺点：

1. 新生儿：Vue.js 是一个新的项目，没有 angular 那么成熟。

2. 影响度不是很大：google 了一下，有关于 Vue.js 多样性或者说丰富性少于其他一些有名的库。

3. 不支持 IE8

Angular

Angular 是一款优秀的前端 JS 框架，已经被用于 Google 的多款产品当中。

它有以下的特性：

1.良好的应用程序结构

2.双向数据绑定

3.指令

4.HTML 模板

5.可嵌入、注入和测试

优点：

1. 模板功能强大丰富，自带了极其丰富的 angular 指令。

2. 是一个比较完善的前端框架，包含服务，模板，数据双向绑定，模块化，路由，过滤器，依赖注入等所有功能；

3. 自定义指令，自定义指令后可以在项目中多次使用。

4. ng 模块化比较大胆的引入了 Java 的一些东西（依赖注入），能够很容易的写出可复用的代码，对于敏捷开发的团队来说非常有帮助。

5. angularjs 是互联网巨人谷歌开发，这也意味着他有一个坚实的基础和社区支持。

缺点：

1. angular 入门很容易 但深入后概念很多, 学习中较难理解.

2. 文档例子非常少, 官方的文档基本只写了 api, 一个例子都没有, 很多时候具体怎么用都是 google 来的, 或直接问 misko,angular 的作者.

3. 对 IE6/7 兼容不算特别好, 就是可以用 jQuery 自己手写代码解决一些.

4. 指令的应用的最佳实践教程少, angular 其实很灵活, 如果不看一些作者的使用原则,很容易写出四不像的代码, 例如 js 中还是像 jQuery 的思想有很多 dom 操作.

相似之处：

都将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库；

都有自己的构建工具，能让你得到一个根据最佳实践设置的项目模板；

都使用了 Virtual DOM（虚拟 DOM）提高重绘性能；

都有 props 的概念，允许组件间的数据传递；

都鼓励组件化应用，将应用分拆成一个个功能明确的模块，提高复用性。

不同之处 ：

1）数据流

Vue 默认支持数据双向绑定，而 React 一直提倡单向数据流

2）虚拟 DOM

Vue2.x 开始引入"Virtual DOM"，消除了和 React 在这方面的差异，但是在具体的细节还是有各自的特点。

Vue 宣称可以更快地计算出 Virtual DOM 的差异，这是由于它在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。

对于 React 而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过 PureComponent/shouldComponentUpdate 这个生命周期方法来进行控制，但 Vue 将此视为默认的优化。

3）组件化

React 与 Vue 最大的不同是模板的编写。

Vue 鼓励写近似常规 HTML 的模板。写起来很接近标准 HTML 元素，只是多了一些属性。

React 推荐你所有的模板通用 JavaScript 的语法扩展——JSX 书写。

具体来讲：React 中 render 函数是支持闭包特性的，所以 import 的组件在 render 中可以直接调用。但是在 Vue 中，由于模板中使用的数据都必须挂在 this 上进行一次中转，所以 import 一个组件完了之后，还需要在 components 中再声明下。

4）监听数据变化的实现原理不同

Vue 通过 getter/setter 以及一些函数的劫持，能精确知道数据变化，不需要特别的优化就能达到很好的性能

React 默认是通过比较引用的方式进行的，如果不优化（PureComponent/shouldComponentUpdate）可能导致大量不必要的 vDOM 的重新渲染。这是因为 Vue 使用的是可变数据，而 React 更强调数据的不可变。

5）高阶组件

react 可以通过高阶组件（HOC）来扩展，而 Vue 需要通过 mixins 来扩展。

高阶组件就是高阶函数，而 React 的组件本身就是纯粹的函数，所以高阶函数对 React 来说易如反掌。相反 Vue.js 使用 HTML 模板创建视图组件，这时模板无法有效的编译，因此 Vue 不能采用 HOC 来实现。

6）构建工具

两者都有自己的构建工具：

React ==> Create React APP

Vue ==> vue-cli

7）跨平台

React ==> React Native

Vue ==> Weex

参考：

https://juejin.cn/post/6844904158093377549

https://juejin.cn/post/6844903733659189255

https://juejin.cn/post/6844903668446134286

https://juejin.cn/post/6847009771355127822

https://juejin.cn/post/6844904136580792334

https://juejin.cn/post/6943068367125381127

6. 单页应用与多页应用的区别?

概念：

SPA 单页面应用（SinglePage Web Application），指只有一个主页面的应用，一开始只需要加载一次 js、css 等相关资源。所有内容都包含在主页面，对每一个功能模块组件化。单页应用跳转，就是切换相关组件，仅仅刷新

局部资源。

MPA 多页面应用 （MultiPage Application），指有多个独立页面的应用，每个页面必须重复加载 js、css 等相关资源。多页应用跳转，需要整页资源刷新。

对比项 \ 模式	SPA	MPA
结构	一个主页面 + 许多模块的组件	许多完整的页面
体验	页面切换快，体验佳；当初次加载文件过多时，需要做相关的调优。	页面切换慢，网速慢的时候，体验尤其不好
资源文件	组件公用的资源只需要加载一次	每个页面都要自己加载公用的资源
适用场景	对体验度和流畅度有较高要求的应用，不利于 SEO（可借助 SSR 优化 SEO）	适用于对 SEO 要求较高的应用
过渡动画	Vue 提供了 transition 的封装组件，容易实现	很难实现
内容更新	相关组件的切换，即局部更新	整体 HTML 的切换，费钱（重复 HTTP 请求）
路由模式	可以使用 hash ，也可以使用 history	普通链接跳转
数据传递	因为单页面，使用全局变量就好（Vuex）	cookie 、localStorage 等缓存方案，URL 参数，调用接口保存等
相关成本	前期开发成本较高，后期维护较为容易	前期开发成本低，后期维护就比较麻烦，因为可能一个功能需要改很多地方

7. 为什么选择使用框架而不是原生?

框架的好处:

组件化: 其中以 React 的组件化最为彻底,甚至可以到函数级别的原子组件,高度的组件化可以是我们的工程易于维护、易于组合拓展。

天然分层: JQuery 时代的代码大部分情况下是面条代码,耦合严重,现代框架不管是 MVC、MVP 还是 MVVM 模式都能帮助我们进行分层，代码解耦更易于读写。

生态: 现在主流前端框架都自带生态,不管是数据流管理架构还是 UI 库都有成熟的解决方案。

开发效率: 现代前端框架都默认自动更新 DOM,而非我们手动操作,解放了开发者的手动 DOM 成本,提高开发效率,从根本上解决了 UI 与状态同步问题。

第五章 Vue

https://www.w3cways.com/tag/vue

https://www.cnblogs.com/ziyue7575/p/42e7ef2211f3ed5d53d796f879184064.html

https://juejin.cn/post/6919373017218809864#heading-10

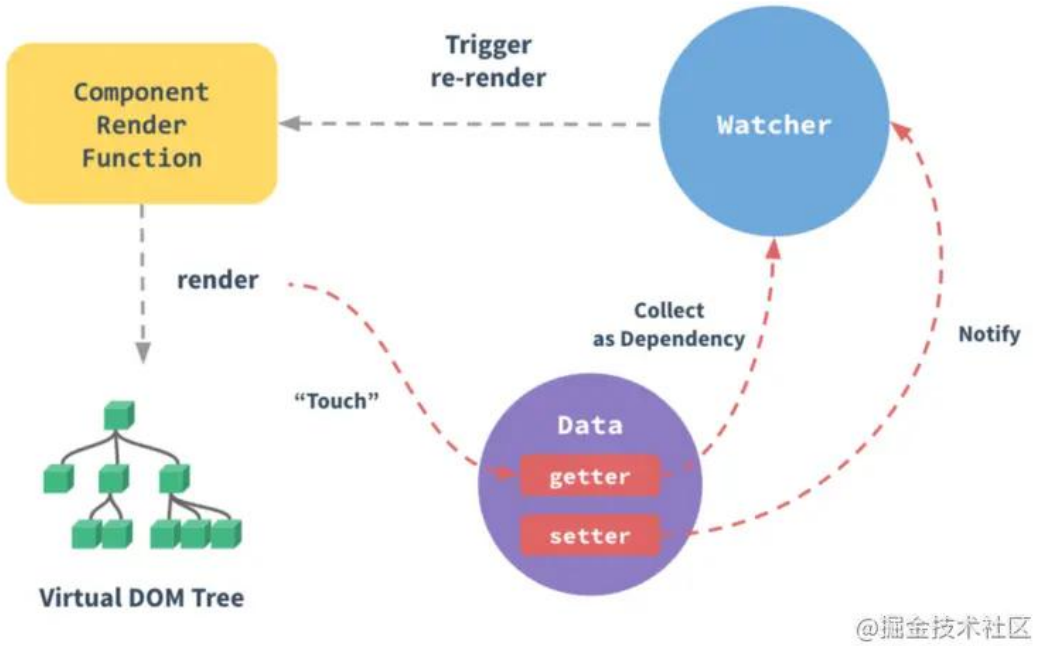
https://juejin.cn/post/6964779204462247950/

一、基础

1.Vue 的基本原理?

当一个 Vue 实例创建时，Vue 会遍历 data 中的属性，用 Object.defineProperty（vue3.0 使用 proxy ）将它们转为 getter/setter，并且在内部追踪相关依赖，在属性被访问和修改时通知变化。 每个组件实例都有相应的 watcher 程序实例，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

https://www.jianshu.com/p/7f01a71d18a6



响应式系统简述:

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。

vue 的数据驱动主要实现建立在三个对象上 Dep、Watcher、Compiler,

Dep 主要负责依赖的收集

Watcher 主要负责 Dep 和 Compiler 之间的联系

Compiler 可以理解为 virtual dom + patch 也就是负责视图层的渲染

2. 双向数据绑定的原理?

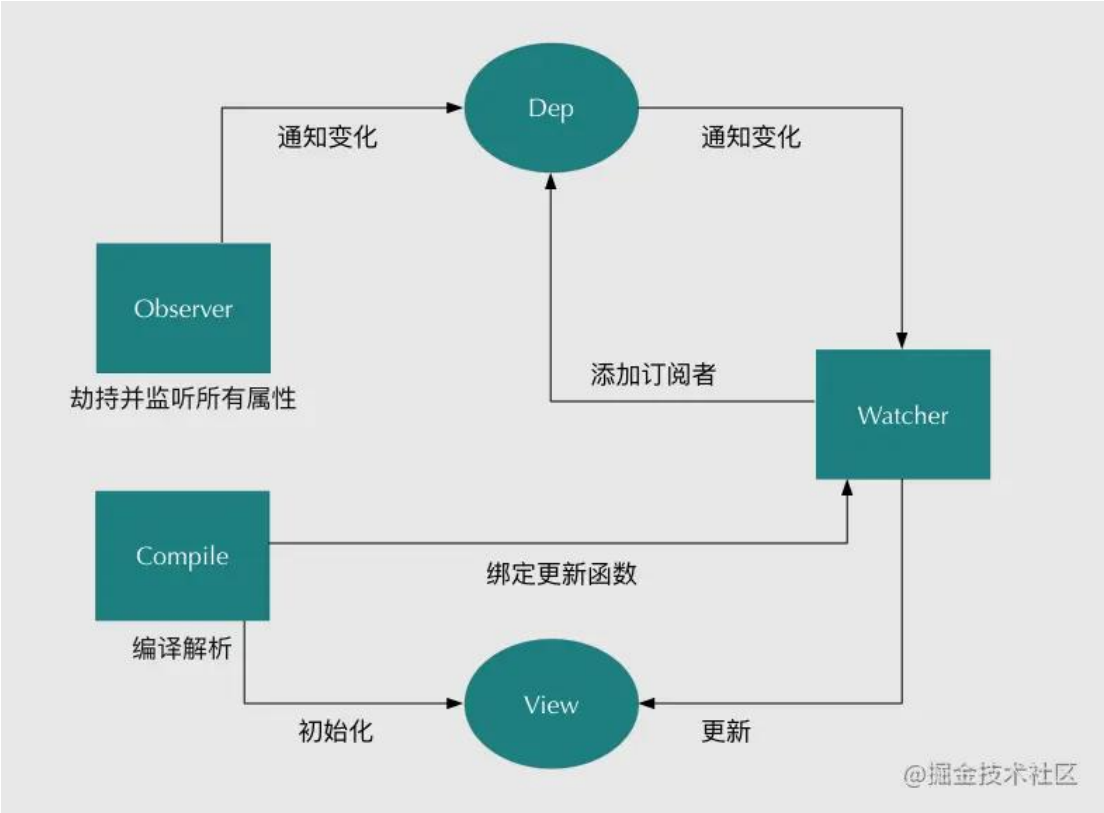
Vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty()来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。主要分为以下几个步骤：

需要 observe 的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter 和 getter 这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化

compile 解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要做的事情是: ①在自身实例化时往属性订阅器(dep)里面添加自己 ②自身必须有一个 update()方法 ③待属性变动 dep.notice()通知时，能调用自身的 update()方法，并触发 Compile 中绑定的回调，则功成身退。

MVVM 作为数据绑定的入口，整合 Observer、Compile 和 Watcher 三者，通过 Observer 来监听自己的 model 数据变化，通过 Compile 来解析编译模板指令，最终利用 Watcher 搭起 Observer 和 Compile 之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据 model 变更的双向绑定效果。



3. 使用 Object.defineProperty() 来进行数据劫持有什么缺点?

在对一些属性进行操作时，使用这种方法无法拦截，比如通过下标方式修改数组数据或者给对象新增属性，

这都不能触发组件的重新渲染，因为 Object.defineProperty 不能拦截到这些操作。更精确的来说，对于数组而言，大部分操作都是拦截不到的，只是 Vue 内部通过重写函数的方式解决了这个问题。

在 Vue3.0 中已经不使用这种方式了，而是通过使用 Proxy 对对象进行代理，从而实现数据劫持。使用 Proxy 的好处是它可以完美的监听到任何方式的数据改变，唯一的缺点是兼容性的问题，因为 Proxy 是 ES6 的语法。

4. 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。

5. Vue 如何检测数组变化?

https://zhuanlan.zhihu.com/p/173963853

https://juejin.cn/post/6935344605424517128#heading-4

Vue 不能检测以下数组的变动，举个例子：

```
var vm = new Vue({
  data: {
    items: ['a', 'b', 'c']
  }
})
vm.items[1] = 'x' // 不是响应性的
vm.items.length = 2 // 不是响应性的
```

当你利用索引直接设置一个数组项时，例如：vm.items[indexOfItem] = newValue

当你修改数组的长度时，例如：vm.items.length = newLength

为了解决第一类问题，以下两种方式都可以实现相同的效果，同时也将在响应式系统内触发状态更新：

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)
// Array.prototype.splice
vm.items.splice(indexOfItem, 1, newValue)
```

你也可以使用 vm.$set 实例方法，该方法是全局方法 Vue.set 的一个别名：

```
vm.$set(vm.items, indexOfItem, newValue)
```

为了解决第二类问题，你可以使用 splice：

```
vm.items.splice(newLength)
```

6. Vue 怎么用 vm.$set() 解决对象新增属性不能响应的问题 ？

受现代 JavaScript 的限制 ，Vue 无法检测到对象属性的添加或删除。由于 Vue 会在初始化实例时对属性执行 getter/setter 转化，所以属性必须在 data 对象上存在才能让 Vue 将它转换为响应式的。但是 Vue 提供了 Vue.set (object, propertyName, value) / vm.$set (object, propertyName, value) 来实现为对象添加响应式属性。

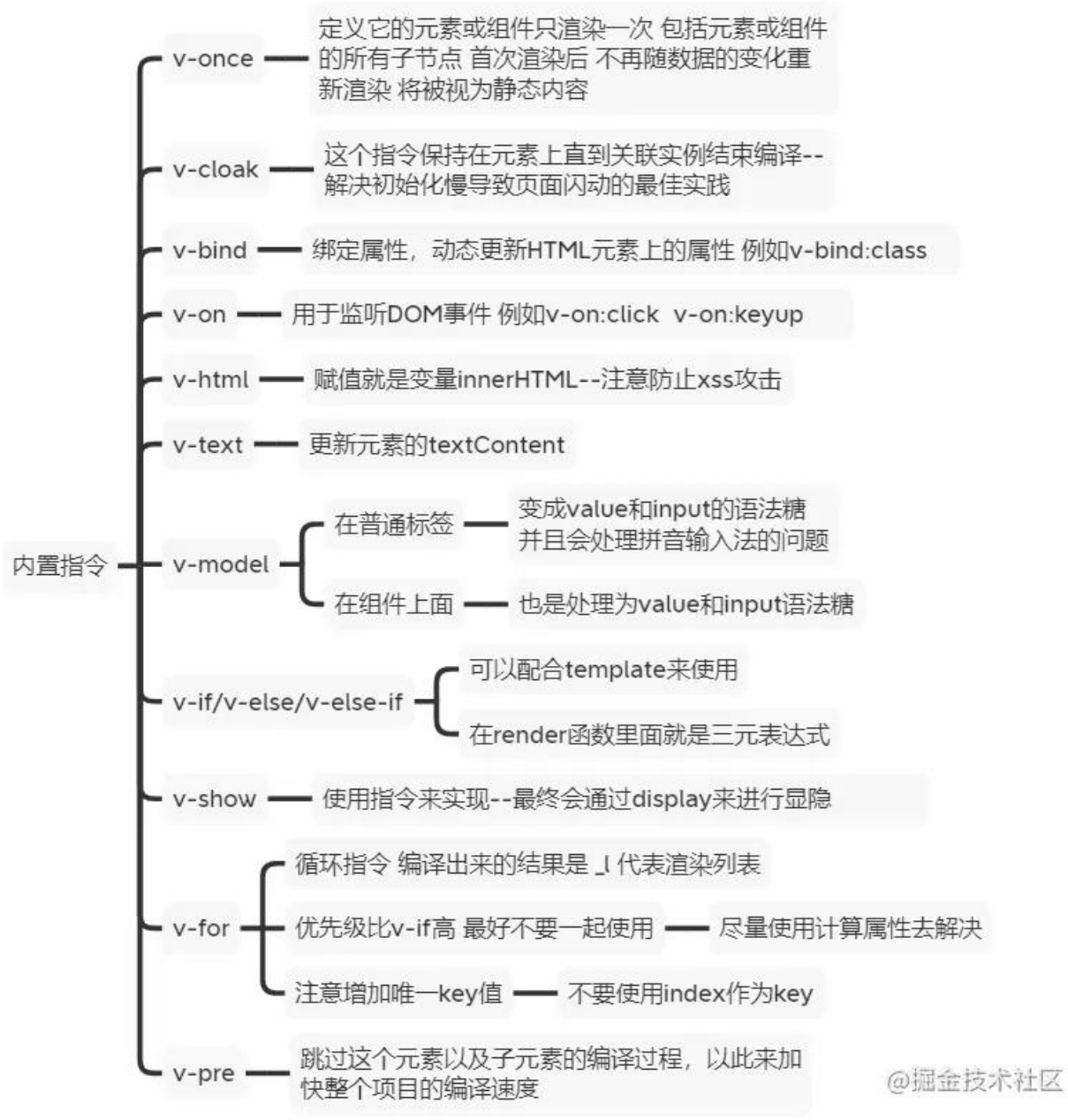
7. 怎样理解 Vue 的单向数据流?

数据总是从父组件传到子组件，子组件没有权利修改父组件传过来的数据，只能请求父组件对原始数据进行修改。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

注意：在子组件直接用 v-model 绑定父组件传过来的 prop 这样是不规范的写法 开发环境会报警告

如果实在要改变父组件的 prop 值 可以再 data 里面定义一个变量 并用 prop 的值初始化它 之后用$emit 通知父组件去修改。

8. Vue 的指令有哪些?



9. v-if、v-show、v-html 的原理?

v-if 会调用 addIfCondition 方法，生成 vnode 的时候会忽略对应节点，render 的时候就不会渲染；

v-show 会生成 vnode，render 的时候也会渲染成真实节点，只是在 render 过程中会在节点的属性中修改 show 属性值，也就是常说的 display；

v-html 会先移除节点下的所有节点，调用 html 方法，通过 addProp 添加 innerHTML 属性，归根结底还是设置 innerHTML 为 v-html 的值。

10. v-if 和 v-show 的区别?

手段：v-if 是动态的向 DOM 树内添加或者删除 DOM 元素；v-show 是通过设置 DOM 元素的 display 样式属性控制显隐；

编译过程：v-if 切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；v-show 只是简单的基于 css 切换；

编译条件：v-if 是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译; v-show 是在任何条件下，无论首次条件是否为真，都被编译，然后被缓存，而且 DOM 元素保留；

性能消耗：v-if 有更高的切换消耗；v-show 有更高的初始渲染消耗；

使用场景：v-if 适合运营条件不大可能改变；v-show 适合频繁切换。

11. vue 中 v-html 会导致哪些问题?

可能会导致 xss 攻击

V-html 更新的是元素的 innerHTML 。内容按普通 HTML 插入， 不会作为 Vue 模板进行编译 。

但是有的时候我们需要渲染的 html 片段中有插值表达式，或者按照 Vue 模板语法给 dom 元素绑定了事件。

在单文件组件里，scoped 的样式不会应用在 v-html 内部，因为那部分 HTML 没有被 Vue 的模板编译器处理。如果你希望针对 v-html 的内容设置带作用域的 CSS，你可以替换为 CSS Modules 或用一个额外的全局 <style>元素手动设置类似 BEM 的作用域策略。

第一种解决方案，照样使用 scoped，但是我们可以使用深度选择器（>>>），示例如下：

```
<style scoped>.a >>> .b{/* ... */}</style>
```

以上代码最终会被编译为：

```
.a[data-v-f3f3eg9] .b{/* ... */}
```

但是这里需要注意，当你的 vue 项目使用 less 或者 sass 的时候，>>>这个玩意可能会失效，我们用/deep/来代替，代码如下：

```
.a {
/deep/ .b{/* ... */}
}
```

第二种解决方案，单文件组件的 style 标签可以使用多次，可以一个 stlye 标签带 scoped 属性针对当前组件，另外一个 style 标签针对全局生效，但是内部我们采用特殊的命名规则即可，例如 BEM 规则。

后台返回的 html 片段，以及 css 样式和 js，但是返回的 js 是不执行的，因为浏览器在渲染的时候并没有将 js 渲染，这时要在$nextTick 中动态创建 script 标签并插入。

12. v-model 是如何实现的，语法糖实际是什么?

（1）作用在表单元素上

动态绑定了 input 的 value 指向了 messgae 变量，并且在触发 input 事件的时候去动态把 message 设置为目标值：

```
<input v-model="sth" />
//  等同于
<input
    v-bind:value="message"
    v-on:input="message=$event.target.value"
>
//$event 指代当前触发的事件对象;
//$event.target 指代当前触发的事件对象的 dom;
//$event.target.value 就是当前 dom 的 value 值;
//在@input 方法中，value => sth;
//在:value 中,sth => value;
```

（2）作用在组件上

在自定义组件中，v-model 默认会利用名为 value 的 prop 和名为 input 的事件

本质是一个父子组件通信的语法糖，通过 prop 和$.emit 实现。 因此父组件 v-model 语法糖本质上可以修改为：

```
<child :value="message"    @input="function(e){message = e}"></child>
```

在组件的实现中，可以通过 v-model 属性来配置子组件接收的 prop 名称，以及派发的事件名称。例子：

```
// 父组件
<aa-input v-model="aa"></aa-input>
// 等价于
<aa-input v-bind:value="aa" v-on:input="aa=$event.target.value"></aa-input>
// 子组件：
<input v-bind:value="aa" v-on:input="onmessage"></aa-input>

props:{value:aa,}
methods:{
    onmessage(e){
        $emit('input',e.target.value)
    }
}
```

默认情况下，一个组件上的 v-model 会把 value 用作 prop 且把 input 用作 event。但是一些输入类型比如单选框和复选框按钮可能想使用 value prop 来达到不同的目的。使用 model 选项可以回避这些情况产生的冲突。js 监听 input 输入框输入数据改变，用 oninput，数据改变以后就会立刻出发这个事件。通过 input 事件把数据$emit 出去，在父组件接受。父组件设置 v-model 的值为 input $emit 过来的值。

13. v-model 可以被用在自定义组件上吗？如果可以，如何使用？

可以。v-model 实际上是一个语法糖，如：

```
<input v-model="searchText">
```

实际上相当于：

```
<input
  v-bind:value="searchText"
  v-on:input="searchText = $event.target.value"
>
```

用在自定义组件上也是同理：

```
<custom-input v-model="searchText">
```

相当于：

```
<custom-input
  v-bind:value="searchText"
  v-on:input="searchText = $event"
></custom-input>
```

显然，custom-input 与父组件的交互如下：

父组件将 searchText 变量传入 custom-input 组件，使用的 prop 名为 value；
custom-input 组件向父组件传出名为 input 的事件，父组件将接收到的值赋值给 searchText；
所以，custom-input 组件的实现应该类似于这样：

```
Vue.component('custom-input', {
  props: ['value'],
  template: `
    <input
      v-bind:value="value"
      v-on:input="$emit('input', $event.target.value)"
    >
  `
})
```

14. v-for 为什么要加 key?

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

如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试就地修改/复用相同类型元素的算法。key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快

相关代码如下

```
// 判断两个 vnode 的标签和 key 是否相同 如果相同 就可以认为是同一节点就地复用
function isSameVnode(oldVnode, newVnode) {
  return oldVnode.tag === newVnode.tag && oldVnode.key === newVnode.key;
}

// 根据 key 来创建老的儿子的 index 映射表 类似 {'a':0,'b':1} 代表 key 为'a'的节点在第一个位置 key 为'b'的节点在第二个位置
function makeIndexByKey(children) {
  let map = {};
  children.forEach((item, index) => {
    map[item.key] = index;
  });
  return map;
}
// 生成的映射表
let map = makeIndexByKey(oldCh);
```

15. diff 算法原理

https://juejin.cn/post/6953433215218483236#heading-2

16. 虚拟 DOM 实现原理?

https://p1-jj.byteimg.com/tos-cn-i-t2oaga2asx/gold-user-assets/2019/8/1/16c49afec13e0416~tplv-t2oaga2asx-image.image

主要包括以下 3 部分：

虚拟 DOM 本质上是 JavaScript 对象,用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象；

记录新树和旧树的差异 diff 算法 — 比较两棵虚拟 DOM 树的差异；状态变更时，

最后把差异更新到真正的 dom 中 pach 算法 — 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树。

17. 虚拟 DOM 的优缺点?

优点:

保证性能下限: 虚拟 DOM 可以经过 diff 找出最小差异,然后批量进行 patch,这种操作虽然比不上手动优化,但是比起粗暴的 DOM 操作性能要好很多,因此虚拟 DOM 可以保证性能下限。

无需手动操作 DOM: 虚拟 DOM 的 diff 和 patch 都是在一次更新中自动进行的,我们无需手动操作 DOM,极大提高开发效率。

跨平台: 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关,相比之下虚拟 DOM 可以进行更方便地跨平台操作,例如服务器渲染、移动端开发等等。

缺点:

无法进行极致优化: 在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化,比如 VScode 采用直接手动操作 DOM 的方式进行极端的性能优化。

18. 常见的事件修饰符及其作用?

https://juejin.cn/post/6981628129089421326

.lazy，改变输入框的值时 value 不会改变，当光标离开输入框时，v-model 绑定的值 value 才会改变

<input type="text" v-model.lazy="value">

.trim，类似于 JavaScript 中的 trim()方法，作用是把 v-model 绑定的值的首尾空格给过滤掉

.number，将值转成数字，但是先输入字符串和先输入数字，是两种情况

先输入数字的话，只取前面数字部分

先输入字母的话，number 修饰符无效

.stop：等同于 JavaScript 中的 event.stopPropagation() ，防止事件冒泡；

<button @click.stop="clickEvent(1)">点击</button>

.prevent ：等同于 JavaScript 中的 event.preventDefault() ，防止执行预设的行为（如果事件可取消，则取消该事件，而不停止事件的进一步传播）；

.capture：与事件冒泡的方向相反，事件捕获由外到内；

.self：只会触发自己范围内的事件，不包含子元素；

.once：只会触发一次。

.native 修饰符是加在自定义组件的事件上，保证事件能执行

执行不了

<My-component @click="shout(3)"></My-component>

可以执行

<My-component @click.native="shout(3)"></My-component>

.left，right，middle 这三个修饰符是鼠标的左中右按键触发的事件

.passive：当我们在监听元素滚动事件的时候，会一直触发 onscroll 事件，在 pc 端是没啥问题的，但是在移动端，会让我们的网页变卡，因此我们使用这个修饰符的时候，相当于给 onscroll 事件整了一个.lazy 修饰符

<div @scroll.passive="onScroll">...

.keyCode：

当我们这么写事件<input type="text" @keyup="shout(4)">的时候，无论按什么按钮都会触发事件

那么想要限制成某个按键触发怎么办？这时候 keyCode 修饰符就派上用场了

<input type="text" @keyup.keyCode="shout(4)">

Vue 提供的 keyCode：

//普通键

.enter .tab .space .esc .up .down .left .right .delete (捕获“删除”和“退格”键)

//系统修饰键

.ctrl .alt .meta .shift

例如（具体的键码请看键码对应表）

按 ctrl 才会触发 <input type="text" @keyup.ctrl="shout(4)">

也可以鼠标事件+按键 <input type="text" @mousedown.ctrl.="shout(4)">

可以多按键触发 例如 ctrl + 67 <input type="text" @keyup.ctrl.67="shout(4)">

19. computed 和 watch 有什么区别?

computed:

是计算属性,也就是计算值,它更多用于计算值的场景

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

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

20. Computed 和 Methods 的区别?

可以将同一函数定义为一个 method 或者一个计算属性。对于最终的结果，两种方式是相同的

不同点：

computed: 计算属性是基于它们的依赖进行缓存的，只有在它的相关依赖发生改变时才会重新求值；

method 调用总会执行该函数。

21. vue 中使用了哪些设计模式?

1.工厂模式 - 传入参数即可创建实例

虚拟 DOM 根据参数的不同返回基础标签的 Vnode 和组件 Vnode

2.单例模式 - 整个程序有且仅有一个实例

vuex 和 vue-router 的插件注册方法 install 判断如果系统存在实例就直接返回掉

3.发布-订阅模式 (vue 事件机制)

4.观察者模式 (响应式数据原理)

5.装饰模式: (@装饰器的用法)

6.策略模式 策略模式指对象有某个行为,但是在不同的场景中,该行为有不同的实现方案-比如选项的合并

22. 使用过 Vue SSR 吗? 说说 SSR?

https://juejin.cn/post/6844903824956588040

SSR 也就是服务端渲染，也就是将 Vue 在客户端把标签渲染成 HTML 的工作放在服务端完成，然后再把 html 直接返回给客户端。

1）服务端渲染的优点：

更好的 SEO： 因为 SPA 页面的内容是通过 Ajax 获取，而搜索引擎爬取工具并不会等待 Ajax 异步完成后再抓取页面内容，所以在 SPA 中是抓取不到页面通过 Ajax 获取到的内容；而 SSR 是直接由服务端返回已经渲染好的页面（数据已经包含在页面中），所以搜索引擎爬取工具可以抓取渲染好的页面；

更快的内容到达时间（首屏加载更快）： SPA 会等待所有 Vue 编译后的 js 文件都下载完成后，才开始进行页面的渲染，文件下载等需要一定的时间等，所以首屏渲染需要一定的时间；SSR 直接由服务端渲染好页面直接返回显示，无需等待下载 js 文件及再去渲染等，所以 SSR 有更快的内容到达时间；

2) 服务端渲染的缺点：

更多的开发条件限制： 例如服务端渲染只支持 beforCreate 和 created 两个钩子函数，这会导致一些外部扩展库需要特殊处理，才能在服务端渲染应用程序中运行；并且与可以部署在任何静态文件服务器上的完全静态单页面应用程序 SPA 不同，服务端渲染应用程序，需要处于 Node.js server 运行环境；

更多的服务器负载：在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用 CPU 资源 (CPU-intensive - CPU 密集)，因此如果你预料在高流量环境 (high traffic) 下使用，请准备相应的服务器负载，并明智地采用缓存策略。

23. Vue 事件绑定原理?

原生事件绑定是通过 addEventListener 绑定给真实元素的，组件事件绑定是通过 Vue 自定义的$on 实现的。如果要在组件上使用原生事件，需要加.native 修饰符，这样就相当于在父组件中把子组件当做普通 html 标签，然后加上原生事件。

$on、$emit 是基于发布订阅模式的，维护一个事件中心，on 的时候将事件按名称存在事件中心里，称之为订阅者，然后 emit 将对应的事件进行发布，去执行事件中心里的对应的监听器。

24. template 和 jsx 的有什么分别?

对于 runtime 来说，只需要保证组件存在 render 函数即可，而有了预编译之后，只需要保证构建过程中生成 render 函数就可以。在 webpack 中，使用 vue-loader 编译.vue 文件，内部依赖的 vue-template-compiler 模块，在 webpack 构建过程中，将 template 预编译成 render 函数。与 react 类似，在添加了 jsx 的语法糖解析器 babel-plugin-transform-vue-jsx 之后，就可以直接手写 render 函数。

所以，template 和 jsx 的都是 render 的一种表现形式，不同的是：JSX 相对于 template 而言，具有更高的灵活性，在复杂的组件中，更具有优势，而 template 虽然显得有些呆滞。但是 template 在代码结构上更符合视图与逻辑分离的习惯，更简单、更直观、更好维护。

25. assets 和 static 的区别?

相同点： assets 和 static 两个都是存放静态资源文件。项目中所需要的资源文件图片，字体图标，样式文件等都可以放在这两个文件下，这是相同点

不相同点：assets 中存放的静态资源文件在项目打包时，也就是运行 npm run build 时会将 assets 中放置的静态资源文件进行打包上传，所谓打包简单点可以理解为压缩体积，代码格式化。而压缩后的静态资源文件最终也都会放置在 static 文件中跟着 index.html 一同上传至服务器。static 中放置的静态资源文件就不会要走打包压缩格式化等流程，而是直接进入打包好的目录，直接上传至服务器。因为避免了压缩直接进行上传，在打包时会提高一定的效率，但是 static 中的资源文件由于没有进行压缩等操作，所以文件的体积也就相对于 assets 中打包后的文件提交较大点。在服务器中就会占据更大的空间。

建议： 将项目中 template 需要的样式文件 js 文件等都可以放置在 assets 中，走打包这一流程。减少体积。而项目中引入的第三方的资源文件如 iconfoont.css 等文件可以放置在 static 中，因为这些引入的第三方文件已经经过处理，不再需要处理，直接上传。

26. delete 和 Vue.delete 删除数组的区别?

delete 只是被删除的元素变成了 empty/undefined 其他的元素的键值还是不变。

Vue.delete 直接删除了数组 改变了数组的键值。

27. 你都做过哪些 Vue 的性能优化?

对象层级不要过深，否则性能就会差

不需要响应式的数据不要放到 data 中（可以用 Object.freeze() 冻结数据）

v-if 和 v-show 区分使用场景

computed 和 watch 区分使用场景

v-for 遍历必须加 key，key 最好是 id 值（唯一值），且避免同时使用 v-if

大数据列表和表格性能优化-虚拟列表/虚拟表格

防止内部泄漏，组件销毁后把全局变量和事件销毁

图片懒加载

路由懒加载

第三方插件的按需引入

适当采用 keep-alive 缓存组件

防抖、节流运用

服务端渲染 SSR or 预渲染

二、生命周期 && keep-alive

1. Vue 的生命周期?

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，称这是 Vue 的生命周期。

beforeCreate（创建前）：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不能访问到 data、computed、watch、methods 上的方法和数据。

created（创建后）：实例创建完成，实例上配置的 options 包括 data、computed、watch、methods 等都配置完成，但是此时渲染得节点还未挂载到 DOM，所以不能访问到 $el 属性。

beforeMount（挂载前）：在挂载开始之前被调用，相关的 render 函数首次被调用。实例已完成以下的配置：编译模板，把 data 里面的数据和模板生成 html。此时还没有挂载 html 到页面上。

mounted（挂载后）：在 el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的 html 内容替换 el 属性指向的 DOM 对象。完成模板中的 html 渲染到 html 页面中。此过程中进行 ajax 交互。

beforeUpdate（更新前）：响应式数据更新时调用，此时虽然响应式数据更新了，但是对应的真实 DOM 还没有被渲染。

updated（更新后）：在由于数据更改导致的虚拟 DOM 重新渲染和打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调用时，组件 DOM 已经更新，所以可以执行依赖于 DOM 的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。

beforeDestroy（销毁前）：实例销毁之前调用。这一步，实例仍然完全可用，this 仍能获取到实例。

destroyed（销毁后）：实例销毁后调用，调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务端渲染期间不被调用。

另外还有 keep-alive 独有的生命周期，分别为 activated 和 deactivated 。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 activated 钩子函数。

2. Vue 子组件和父组件执行顺序?

加载渲染过程：

父组件 beforeCreate

父组件 created

父组件 beforeMount

子组件 beforeCreate

子组件 created

子组件 beforeMount

子组件 mounted

父组件 mounted

更新过程：

父组件 beforeUpdate

子组件 beforeUpdate

子组件 updated

父组件 updated

销毁过程：

父组件 beforeDestroy

子组件 beforeDestroy

子组件 destroyed

父组件 destoryed

3. 父组件可以监听到子组件的生命周期吗?

比如有父组件 Parent 和子组件 Child，如果父组件监听到子组件挂载 mounted 就做一些逻辑处理，可以通过以下写法实现：

```
// Parent.vue
<Child @mounted="doSomething"/>


// Child.vue
mounted() {
  this.$emit("mounted");
}
```

以上需要手动通过 $emit 触发父组件的事件，更简单的方式可以在父组件引用子组件时通过 @hook 来监听即可，如下所示：

```
//  Parent.vue
<Child @hook:mounted="doSomething" ></Child>
doSomething() {
   console.log('父组件监听到 mounted 钩子函数 ...');
},


//  Child.vue
mounted(){
   console.log('子组件触发 mounted 钩子函数 ...');
},
// 以上输出顺序为：
// 子组件触发 mounted 钩子函数 ...
```

// 父组件监听到 mounted 钩子函数 ...

当然 **@hook 方法不仅仅是可以监听 mounted，其它的生命周期事件，例如：created，updated 等都可以监听。**

4. created 和 mounted 的区别?

created:在模板渲染成 html 前调用，即通常初始化某些属性值，然后再渲染成视图。

mounted:在模板渲染成 html 后调用，通常是初始化页面完成后，再对 html 的 dom 节点进行一些需要的操作。

5. 一般在哪个生命周期请求异步数据?

我们可以在钩子函数 created、beforeMount、mounted 中进行调用，因为在这三个钩子函数中，data 已经创建，可以将服务端端返回的数据进行赋值。

官方实例的异步请求是在 mounted 生命周期中调用的，而实际上推荐在 created 钩子函数中调用异步请求，因为在 created 钩子函数中调用异步请求有以下优点：

能更快获取到服务端数据，减少页面加载时间，用户体验更好；

SSR 不支持 beforeMount 、mounted 钩子函数，放在 created 中有助于一致性。

6. 谈谈你对 keep-alive 的了解? keep-alive 中的生命周期哪些?

keep-alive 是 Vue 提供的一个内置组件，用来对组件进行缓存——在组件切换过程中将状态保留在内存中，防止重复渲染 DOM。

如果为一个组件包裹了 keep-alive，那么它会多出两个生命周期：deactivated、activated。同时，beforeDestroy 和 destroyed 就不会再被触发了，因为组件不会被真正销毁。

当组件被换掉时，会被缓存到内存中、触发 deactivated 生命周期；当组件被切回来时，再去缓存里找这个组件、触发 activated 钩子函数。

7. 谈谈你对 keep-alive 的了解?

如果未使用 keep-alive 组件，则在页面回退时仍然会重新渲染页面，触发 created 钩子，使用体验不好。

在以下场景中使用 keep-alive 组件会显著提高用户体验：

商品列表页点击商品跳转到商品详情，返回后仍显示原有信息

订单列表跳转到订单详情，返回，等等场景。

在动态组件中的应用

```
<keep-alive :include="whiteList" :exclude="blackList" :max="amount">
  <component :is="currentComponent"></component>
</keep-alive>
```

在 vue-router 中的应用

```
<keep-alive :include="whiteList" :exclude="blackList" :max="amount">
  <router-view></router-view>
</keep-alive>
```

include 定义缓存白名单，keep-alive 会缓存命中的组件；

exclude 定义缓存黑名单，被命中的组件将不会被缓存；

max 定义缓存组件上限，超出上限使用 LRU 的策略置换缓存数据。

实际应用场景：

实现前进刷新，后退不刷新 https://juejin.cn/post/6844903555657269261

记录页面滚动位置：

keep-alive 并不会记录页面的滚动位置，所以我们在跳转时需要记录当前的滚动位置，在触发 activated 钩子时重新定位到原有位置。

具体设计思路：

在 deactivated 钩子中记录当前滚动位置，在这里我使用的是 localStorage：

```
deactivated () {
    window.localStorage.setItem(this.key, JSON.stringify({
      listScrollTop: this.scrollTop
    }))
  }
```

在 activated 钩子中滚动：

```
this.cacheData = window.localStorage.getItem(this.key) ? JSON.parse(window.localStorage.getItem(this.key)) : null
$('.sidebar-item').scrollTop(this.cacheData.listScrollTop)
```

8. keep-alive 组件的渲染?

Vue 在初始化生命周期的时候，为组件实例建立父子关系会根据 abstract 属性决定是否忽略某个组件。在 keep-alive 中，设置了 abstract: true，那 Vue 就会跳过该组件实例。

最后构建的组件树中就不会包含 keep-alive 组件，那么由组件树渲染成的 DOM 树自然也不会有 keep-alive 相关的节点了。

在首次加载被包裹组件时，由 keep-alive.js 中的 render 函数可知，vnode.componentInstance 的值是 undefined，keepAlive 的值是 true，因为 keep-alive 组件作为父组件，它的 render 函数会先于被包裹组件执行；那么就只执行到 i(vnode, false /* hydrating */)，后面的逻辑不再执行；

再次访问被包裹组件时，vnode.componentInstance 的值就是已经缓存的组件实例，那么会执行 insert(parentElm, vnode.elm, refElm)逻辑，这样就直接把上一次的 DOM 插入到了父元素中

参考：https://juejin.cn/post/6844903846901186574#heading-5

三、组件通信 && 组件相关问题

9. 组件中的 data 为什么是函数?

避免命名污染

JavaScript 中的对象是引用类型的数据，当多个实例引用同一个对象时，只要一个实例对这个对象进行操作，其他实例中的数据也会发生变化。

而在 Vue 中，更多的是想要复用组件，那就需要每个组件都有自己的数据，这样组件之间才不会相互干扰。

所以组件的数据不能写成对象的形式，而是要写成函数的形式。数据以函数返回值的形式定义，这样当每次复用组件的时候，就会返回一个新的 data，也就是说每个组件都有自己的私有数据空间，它们各自维护自己的数据，不会干扰其他组件的正常运行。

10. 子组件可以直接改变父组件的数据吗?

子组件不可以直接改变父组件的数据。这样做主要是为了维护父子组件的单向数据流。每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。如果这样做了，Vue 会在浏览器的控制台中发出警告。

Vue 提倡单向数据流，即父级 props 的更新会流向子组件，但是反过来则不行。这是为了防止意外的改变父

组件状态，使得应用的数据流变得难以理解，导致数据流混乱。如果破坏了单向数据流，当应用复杂时，debug 的成本会非常高。

只能通过 $emit 派发一个自定义事件，父组件接收到后，由父组件修改。

11. Vue 组件通讯有哪几种方式?

（1） props / $emit

父组件通过 props 向子组件传递数据，子组件通过$emit 和父组件通信

父组件向子组件传值

props 只能是父组件向子组件进行传值，props 使得父子组件之间形成了一个单向下行绑定。子组件的数据会随着父组件不断更新。 可以显示定义一个或一个以上的数据，对于接收的数据，可以是各种数据类型，同样也可以传递一个函数。props 属性名规则：若在 props 中使用驼峰形式，模板中需要使用短横线的形式。

子组件向父组件传值

$emit 绑定一个自定义事件，当这个事件被执行的时就会将参数传递给父组件，而父组件通过 v-on 监听并接收参数。

（2）eventBus 事件总线（$emit / $on）

（3）依赖注入（project / inject）

这种方式就是 Vue 中的依赖注入，该方法用于父子组件之间的通信。当然这里所说的父子不一定是真正的父子，也可以是祖孙组件，在层数很深的情况下，可以使用这种方法来进行传值。就不用一层一层的传递了。

project / inject 是 Vue 提供的两个钩子，和 data、methods 是同级的。并且 project 的书写形式和 data 一样。

project 钩子用来发送数据或方法

inject 钩子用来接收数据或方法

官方不推荐在实际业务中使用，但是写组件库时很常用

（4）ref / $refs

这种方式也是实现父子组件之间的通信。

ref： 这个属性用在子组件上，它的引用就指向了子组件的实例。可以通过实例来访问组件的数据和方法。

在子组件中：

```
export default {
  data () {
    return {
      name: 'JavaScript'
    }
  },
  methods: {
    sayHello () {
      console.log('hello')
    }
  }
}
```

在父组件中：

```
<template>
  <child ref="child"></component-a>
</template>
<script>
  import child from './child.vue'
  export default {
    components: { child },
    mounted () {
      console.log(this.$refs.child.name);   // JavaScript
      this.$refs.child.sayHello();   // hello
    }
  }
</script>
```

（5）$parent / $children

使用$parent 可以让组件访问父组件的实例（访问的是上一级父组件的属性和方法）

使用$children 可以让组件访问子组件的实例，但是，$children 并不能保证顺序，并且访问的数据也不是响应式的。

通过$parent 访问到的是上一级父组件的实例，可以使用$root 来访问根组件的实例

在组件中使用$children 拿到的是所有的子组件的实例，它是一个数组，并且是无序的

在根组件#app 上拿$parent 得到的是 new Vue()的实例，在这实例上再拿$parent 得到的是 undefined，而在最底层的子组件拿$children 是个空数组

$children 的值是数组，而$parent 是个对象

（6）$attrs / $listeners

考虑一种场景，如果 A 是 B 组件的父组件，B 是 C 组件的父组件。如果想要组件 A 给组件 C 传递数据，这种隔代的数据，该使用哪种方式呢？

如果是用 props/$emit 来一级一级的传递，确实可以完成，但是比较复杂；如果使用事件总线，在多人开发或者项目较大的时候，维护起来很麻烦；如果使用 Vuex，的确也可以，但是如果仅仅是传递数据，那可能就有点浪费了。

针对上述情况，Vue 引入了$attrs / $listeners，实现组件之间的跨代通信。

inheritAttrs，它的默认值 true，继承所有的父组件属性除 props 之外的所有属性；inheritAttrs：false 只继承 class 属性 。

$attrs：继承所有的父组件属性（除了 prop 传递的属性、class 和 style ），一般用在子组件的子元素上

$listeners：该属性是一个对象，里面包含了作用在这个组件上的所有监听器，可以配合 v-on="$listeners" 将所有的事件监听器指向这个组件的某个特定的子元素。（相当于子组件继承父组件的事件）

（7）vueX

四、vue-router

12. 对前端路由的理解?

在前端技术早期，一个 url 对应一个页面，如果要从 A 页面切换到 B 页面，那么必然伴随着页面的刷新。这个体验并不好，不过在最初也是无奈之举——用户只有在刷新页面的情况下，才可以重新去请求数据。

后来，改变发生了——Ajax 出现了，它允许人们在不刷新页面的情况下发起请求；与之共生的，还有“不刷新页面即可更新页面内容”这种需求。在这样的背景下，出现了 SPA（单页面应用）。

SPA 极大地提升了用户体验，它允许页面在不刷新的情况下更新页面内容，使内容的切换更加流畅。但是在 SPA 诞生之初，人们并没有考虑到“定位”这个问题——在内容切换前后，页面的 URL 都是一样的，这就带来了两个问题：

SPA 其实并不知道当前的页面“进展到了哪一步”。可能在一个站点下经过了反复的“前进”才终于唤出了某一块内容，但是此时只要刷新一下页面，一切就会被清零，必须重复之前的操作、才可以重新对内容进行定位——SPA 并不会“记住”你的操作。

由于有且仅有一个 URL 给页面做映射，这对 SEO 也不够友好，搜索引擎无法收集全面的信息

为了解决这个问题，前端路由出现了。

前端路由可以帮助我们在仅有一个页面的情况下，“记住”用户当前走到了哪一步——为 SPA 中的各个视图匹配一个唯一标识。这意味着用户前进、后退触发的新内容，都会映射到不同的 URL 上去。此时即便他刷新页面，因为当前的 URL 可以标识出他所处的位置，因此内容也不会丢失。

那么如何实现这个目的呢？首先要解决两个问题：

当用户刷新页面时，浏览器会默认根据当前 URL 对资源进行重新定位（发送请求）。这个动作对 SPA 是不必要的，因为我们的 SPA 作为单页面，无论如何也只会有一个资源与之对应。此时若走正常的请求-刷新流程，反而会使用户的前进后退操作无法被记录。

单页面应用对服务端来说，就是一个 URL、一套资源，那么如何做到用“不同的 URL”来映射不同的视图内容呢？

从这两个问题来看，服务端已经完全救不了这个场景了。所以要靠咱们前端自力更生，不然怎么叫“前端路由”呢？作为前端，可以提供这样的解决思路：

拦截用户的刷新操作，避免服务端盲目响应、返回不符合预期的资源内容。把刷新这个动作完全放到前端逻辑里消化掉。

感知 URL 的变化。这里不是说要改造 URL、凭空制造出 N 个 URL 来。而是说 URL 还是那个 URL，只不过我们可以给它做一些微小的处理——这些处理并不会影响 URL 本身的性质，不会影响服务器对它的识别，只有我们前端感知的到。一旦我们感知到了，我们就根据这些变化、用 JS 去给它生成不同的内容。

13. Vue-Router 的懒加载如何实现?

（1）方案一(常用)：使用箭头函数+import 动态加载

```
const List = () => import('@/components/list.vue')
const router = new VueRouter({
  routes: [
    { path: '/list', component: List }
  ]
})
```

（2）方案二：使用箭头函数+require 动态加载

```
const router = new Router({
```

```
    routes: [
     {
        path: '/list',
        component: resolve => require(['@/components/list'], resolve)
     }
    ]
})
```

（3）方案三：使用 webpack 的 require.ensure 技术，也可以实现按需加载。 这种情况下，多个路由指定相同的 chunkName，会合并打包成一个 js 文件。

```
const List = resolve => require.ensure([], () => resolve(require('@/components/list')), 'list');
// 路由也是正常的写法  这种是官方推荐的写的 按模块划分懒加载
const router = new Router({
    routes: [
    {
        path: '/list',
        component: List,
        name: 'list'
    }
 ]
})
```

14. vue-router 路由模式有几种?

3 种路由模式：

hash: 使用 URL hash 值来作路由。支持所有浏览器，包括不支持 HTML5 History Api 的浏览器；

history : 依赖 HTML5 History API 和服务器配置。具体可以查看 HTML5 History 模式；

abstract : 支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式.。

15. 路由的 hash 和 history 模式的区别?

1. hash 模式

简介： hash 模式是开发中默认的模式，它的 URL 带着一个#，例如：www.abc.com/#/vue，它的 hash 值就是#/vue。

特点：hash 值会出现在 URL 里面，但是不会出现在 HTTP 请求中，对后端完全没有影响。所以改变 hash 值，不会重新加载页面。这种模式的浏览器支持度很好，低版本的 IE 浏览器也支持这种模式。hash 路由被称为是前端路由，已经成为 SPA（单页面应用）的标配。

原理： hash 模式的主要原理就是 onhashchange()事件：

使用 onhashchange()事件的好处就是，在页面的 hash 值发生变化时，无需向后端发起请求，window 就可以监听事件的改变，并按规则加载相应的代码。除此之外，hash 值变化对应的 URL 都会被浏览器记录下来，这样浏览器就能实现页面的前进和后退。虽然是没有请求后端服务器，但是页面的 hash 值和对应的 URL 关联起来了。

2. history 模式

简介： history 模式的 URL 中没有#，它使用的是传统的路由分发模式，即用户在输入一个 URL 时，服务器会接收这个请求，并解析这个 URL，然后做出相应的逻辑处理。

特点： 当使用 history 模式时，URL 就像这样：abc.com/user/id。相比 hash 模式更加好看。但是，history 模式需要后台配置支持。如果后台没有正确配置，访问时会返回 404。

API： history api 可以分为两大部分，切换历史状态和修改历史状态：

修改历史状态：包括了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法，这两个方法应用于浏览器的历史记录栈，提供了对历史记录进行修改的功能。只是当他们进行修改时，虽然修改了 url，但浏览器不会立即向后端发送请求。如果要做到改变 url 但又不刷新页面的效果，就需要前端用上这两个 API。

切换历史状态： 包括 forward()、back()、go()三个方法，对应浏览器的前进，后退，跳转操作。

虽然 history 模式丢弃了丑陋的#。但是，它也有自己的缺点，就是在刷新页面的时候，如果没有相应的路由或资源，就会刷出 404 来。

3. 两种模式对比

调用 history.pushState() 相比于直接修改 hash，存在以下优势:

pushState() 设置的新 URL 可以是与当前 URL 同源的任意 URL；而 hash 只可修改 # 后面的部分，因此只能设置与当前 URL 同文档的 URL；

pushState() 设置的新 URL 可以与当前 URL 一模一样，这样也会把记录添加到栈中；而 hash 设置的新值必须与原来不一样才会触发动作将记录添加到栈中；

pushState() 通过 stateObject 参数可以添加任意类型的数据到记录中；而 hash 只可添加短字符串；

pushState() 可额外设置 title 属性供后续使用。

hash 模式下，仅 hash 符号之前的 url 会被包含在请求中，后端如果没有做到对路由的全覆盖，也不会返回 404 错误；history 模式下，前端的 url 必须和实际向后端发起请求的 url 一致，如果没有对用的路由处理，将返回 404 错误。

hash 模式和 history 模式都有各自的优势和缺陷，还是要根据实际情况选择性的使用。

16. 如何获取页面的 hash 变化?

（1）监听$route 的变化

```
// 监听,当路由发生变化的时候执行
watch: {
  $route: {
    handler: function(val, oldVal){
      console.log(val);
    },
    // 深度观察监听
    deep: true
  }
},
```

（2）window.location.hash 读取#值

window.location.hash 的值可读可写，读取来判断状态是否改变，写入时可以在不重载网页的前提下，添加一条历史访问记录。

17. $route 和$router 的区别?

$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。

18. Vue-router 跳转和 location.href 有什么区别?

使用 location.href= /url 来跳转，简单方便，但是刷新了页面；

使用 history.pushState(/url) ，无刷新页面，静态跳转；

引进 router ，然后使用 router.push(/url) 来跳转，使用了 diff 算法，实现了按需加载，减少了 dom 的消耗。其实使用 router 跳转和使用 history.pushState() 没什么差别的，因为 vue-router 就是用了 history.pushState() ，尤其是在 history 模式下。

19. params 和 query 的区别?

用法：query 要用 path 来引入，params 要用 name 来引入，接收参数都是类似的，分别是 this.$route.query.name 和 this.$route.params.name 。

url 地址显示：query 更加类似于 ajax 中 get 传参，params 则类似于 post，说的再简单一点，前者在浏览器地址栏中显示参数，后者则不显示

注意：query 刷新不会丢失 query 里面的数据 params 刷新会丢失 params 里面的数据。

20. Vue-router 导航守卫有哪些?

全局前置/钩子：beforeEach、beforeResolve、afterEach

路由独享的守卫：beforeEnter

组件内的守卫：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave

21. Vue-router 路由钩子在生命周期的体现?

一、Vue-Router 导航守卫

有的时候，需要通过路由来进行一些操作，比如最常见的登录权限验证，当用户满足条件时，才让其进入导航，否则就取消跳转，并跳到登录页面让其登录。

为此有很多种方法可以植入路由的导航过程：全局的，单个路由独享的，或者组件级的

全局路由钩子

vue-router 全局有三个路由钩子;

router.beforeEach 全局前置守卫 进入路由之前

router.beforeResolve 全局解析守卫（2.5.0+）在 beforeRouteEnter 调用之后调用

router.afterEach 全局后置钩子 进入路由之后

具体使用：

beforeEach（判断是否登录了，没登录就跳转到登录页）

```
router.beforeEach((to, from, next) => {
    let ifInfo = Vue.prototype.$common.getSession('userData');  // 判断是否登录的存储信息
    if (!ifInfo) {
        // sessionStorage 里没有储存 user 信息
        if (to.path == '/') {
            //如果是登录页面路径，就直接 next()
```

```
            next();
        } else {
            //不然就跳转到登录
            Message.warning("请重新登录！");
            window.location.href = Vue.prototype.$loginUrl;
        }
    } else {
        return next();
    }
})
```

afterEach （跳转之后滚动条回到顶部）

```
router.afterEach((to, from) => {
    // 跳转之后滚动条回到顶部
    window.scrollTo(0,0);
});
```

单个路由独享钩子

beforeEnter

如果不想全局配置守卫的话，可以为某些路由单独配置守卫，有三个参数： to、from、next

```
export default [
    {
        path: '/',
        name: 'login',
        component: login,
        beforeEnter: (to, from, next) => {
            console.log('即将进入登录页面')
            next()
        }
    }
]
```

组件内钩子

beforeRouteUpdate、beforeRouteEnter、beforeRouteLeave

beforeRouteEnter： 进入组件前触发

beforeRouteUpdate： 当前地址改变并且改组件被复用时触发，举例来说，带有动态参数的路径 foo/：id，在 /foo/1 和 /foo/2 之间跳转的时候，由于会渲染同样的 foa 组件，这个钩子在这种情况下就会被调用

beforeRouteLeave： 离开组件被调用

注意，beforeRouteEnter 组件内还访问不到 this，因为该守卫执行前组件实例还没有被创建，需要传一个回调给 next 来访问，例如：

```
beforeRouteEnter(to, from, next) {
    next(target => {
        if (from.path == '/classProcess') {
            target.isFromProcess = true
        }
    })
}
```

二、Vue 路由钩子在生命周期函数的体现

完整的路由导航解析流程（不包括其他生命周期）

触发进入其他路由。

调用要离开路由的组件守卫 beforeRouteLeave

调用局前置守卫： beforeEach

在重用的组件里调用 beforeRouteUpdate

调用路由独享守卫 beforeEnter。

解析异步路由组件。

在将要进入的路由组件中调用 beforeRouteEnter

调用全局解析守卫 beforeResolve

导航被确认。

调用全局后置钩子的 afterEach 钩子。

触发 DOM 更新（mounted）。

执行 beforeRouteEnter 守卫中传给 next 的回调函数

触发钩子的完整顺序（路由导航、keep-alive、和组件生命周期钩子结合起来的，触发顺序，假设是从 a 组件离开，第一次进入 b 组件）：

beforeRouteLeave：路由组件的组件离开路由前钩子，可取消路由离开。

beforeEach：路由全局前置守卫，可用于登录验证、全局路由 loading 等。

beforeEnter：路由独享守卫

beforeRouteEnter：路由组件的组件进入路由前钩子。

beforeResolve：路由全局解析守卫

afterEach：路由全局后置钩子

beforeCreate：组件生命周期，不能访问 tAis。

created;组件生命周期，可以访问 tAis，不能访问 dom。

beforeMount：组件生命周期

deactivated：离开缓存组件 a，或者触发 a 的 beforeDestroy 和 destroyed 组件销毁钩子。

mounted：访问/操作 dom。

activated：进入缓存组件，进入 a 的嵌套子组件（如果有的话）。

执行 beforeRouteEnter 回调函数 next。

导航行为被触发到导航完成的整个过程

导航行为被触发，此时导航未被确认。

在失活的组件里调用离开守卫 beforeRouteLeave。

调用全局的 beforeEach 守卫。

在重用的组件里调用 beforeRouteUpdate 守卫(2.2+)。

在路由配置里调用 beforeEnteY。

解析异步路由组件（如果有）。

在被激活的组件里调用 beforeRouteEnter。

调用全局的 beforeResolve 守卫（2.5+），标示解析阶段完成。

导航被确认。

调用全局的 afterEach 钩子。

非重用组件，开始组件实例的生命周期：beforeCreate&created、beforeMount&mounted

触发 DOM 更新。

用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

导航完成

五、vueX

22. Vuex 的原理?

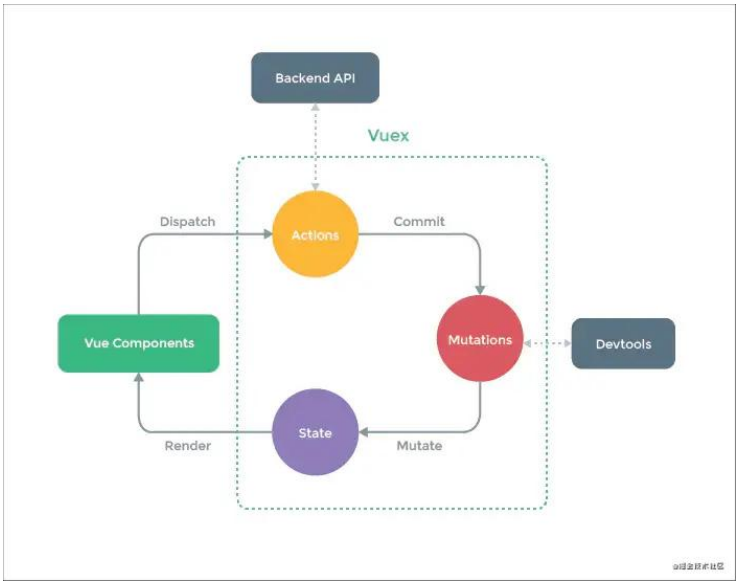
Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。

Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样可以方便地跟踪每一个状态的变化。

Vuex 为 Vue Components 建立起了一个完整的生态圈，包括开发中的 API 调用一环。

23. 核心流程中的主要功能?



Vue Components 是 vue 组件，组件会触发（dispatch）一些事件或动作，也就是图中的 Actions;

在组件中发出的动作，肯定是想获取或者改变数据的，但是在 vuex 中，数据是集中管理的，不能直接去更改数据，所以会把这个动作提交（Commit）到 Mutations 中;

然后 Mutations 就去改变（Mutate）State 中的数据;

当 State 中的数据被改变之后，就会重新渲染（Render）到 Vue Components 中去，组件展示更新后的数据，完成一个流程。

24. 各模块在核心流程中的主要功能?

Vue Components：Vue 组件。HTML 页面上，负责接收用户操作等交互行为，执行 dispatch 方法触发对应 action 进行回应。

dispatch：操作行为触发方法，是唯一能执行 action 的方法。

actions： 操作行为处理模块。负责处理 Vue Components 接收到的所有交互行为。包含同步/异步操作，支持多个同名方法，按照注册的顺序依次触发。向后台 API 请求的操作就在这个模块中进行，包括触发其他 action 以及提交 mutation 的操作。该模块提供了 Promise 的封装，以支持 action 的链式触发。

commit：状态改变提交操作方法。对 mutation 进行提交，是唯一能执行 mutation 的方法。

mutations：状态改变操作方法。是 Vuex 修改 state 的唯一推荐方法，其他修改方式在严格模式下将会报错。该方法只能进行同步操作，且方法名只能全局唯一。操作之中会有一些 hook 暴露出来，以进行 state 的监控等。

state： 页面状态管理容器对象。集中存储 Vuecomponents 中 data 对象的零散数据，全局唯一，以进行统一的状态管理。页面显示所需的数据从该对象中进行读取，利用 Vue 的细粒度数据响应机制来进行高效的状态更新。

getters： state 对象读取方法。图中没有单独列出该模块，应该被包含在了 render 中，Vue Components 通过该方法读取全局 state 对象。

25. Vuex 有哪几种属性?

state => 基本数据(数据源存放地)

getters => 从基本数据派生出来的数据

mutations => 提交更改数据的方法，同步

actions => 像一个装饰器，包裹 mutations，使之可以异步

modules => 模块化 Vuex

26. Vuex 和单纯的全局对象有什么区别?

Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样可以方便地跟踪每一个状态的变化，从而能够实现一些工具帮助更好地了解我们的应用。

27. 为什么 Vuex 的 mutation 中不能做异步操作?

Vuex 中所有的状态更新的唯一途径都是 mutation，异步操作通过 Action 来提交 mutation 实现，这样可以方便地跟踪每一个状态的变化，从而能够实现一些工具帮助更好地了解我们的应用。

每个 mutation 执行完成后都会对应到一个新的状态变更，这样 devtools 就可以打个快照存下来，然后就可以实现 time-travel 了。如果 mutation 支持异步操作，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难。

28. Vuex 的严格模式是什么,有什么作用，如何开启?

在严格模式下，无论何时发生了状态变更且不是由 mutation 函数引起的，将会抛出错误。这能保证所有的状

态变更都能被调试工具跟踪到。

在 Vuex.Store 构造器选项中开启,如下

```
const store = new Vuex.Store({
    strict:true,
})
```

29. Redux 和 Vuex 有什么区别，它们的共同思想?

（1）Redux 和 Vuex 区别

Vuex 改进了 Redux 中的 Action 和 Reducer 函数，以 mutations 变化函数取代 Reducer，无需 switch，只需在对应的 mutation 函数里改变 state 值即可

Vuex 由于 Vue 自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的 State 即可

Vuex 数据流的顺序是 :View 调用 store.commit 提交对应的请求到 Store 中对应的 mutation 函数->store 改变（vue 检测到数据变化自动渲染）

通俗点理解就是，vuex 弱化 dispatch，通过 commit 进行 store 状态的一次更变;取消了 action 概念，不必传入特定的 action 形式进行指定变更;弱化 reducer，基于 commit 参数直接对数据进行转变，使得框架更加简易;

（2）共同思想

单一的数据源

变化可以预测

本质上：redux 与 vuex 都是对 mvvm 思想的服务，将数据从视图中抽离的一种方案;

形式上：vuex 借鉴了 redux，将 store 作为全局的数据中心，进行 mode 管理;

30. 为什么要用 Vuex 或者 Redux?

由于传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致代码无法维护。

所以需要把组件的共享状态抽取出来，以一个全局单例模式管理。在这种模式下，组件树构成了一个巨大的"视图"，不管在树的哪个位置，任何组件都能获取状态或者触发行为。

另外，通过定义和隔离状态管理中的各种概念并强制遵守一定的规则，代码将会变得更结构化且易维护。

六、其他

31. Proxy 与 Object.defineProperty 优劣对比?

当一个 Vue 实例创建时，Vue 会遍历 data 中的属性，用 Object.defineProperty（vue3.0 使用 proxy ）将它们转为 getter/setter，并且在内部追踪相关依赖，在属性被访问和修改时通知变化。 每个组件实例都有相应的 watcher 程序实例，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

Vue 在实例初始化时遍历 data 中的所有属性，并使用 Object.defineProperty 把这些属性全部转为 getter/setter。这样当追踪数据发生变化时，setter 会被自动调用。

Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

但是这样做有以下问题：

添加或删除对象的属性时，Vue 检测不到。因为添加或删除的对象没有在初始化进行响应式处理，只能通过 $set 来调用 Object.defineProperty()处理。

无法监控到数组下标和长度的变化。

Vue3 使用 Proxy 来监控数据的变化。Proxy 是 ES6 中提供的功能，其作用为：用于定义基本操作的自定义行为（如属性查找，赋值，枚举，函数调用等）。相对于 Object.defineProperty()，其有以下特点：

Proxy 直接代理整个对象而非对象属性，这样只需做一层代理就可以监听同级结构下的所有属性变化，包括新增属性和删除属性。

Proxy 可以监听数组的变化。

32. 函数式组件使用场景和原理?

函数式组件与普通组件的区别

1.函数式组件需要在声明组件是指定 functional:true

2.不需要实例化，所以没有 this,this 通过 render 函数的第二个参数 context 来代替

3.没有生命周期钩子函数，不能使用计算属性，watch

4.不能通过$emit 对外暴露事件，调用事件只能通过 context.listeners.click 的方式调用外部传入的事件

5.因为函数式组件是没有实例化的，所以在外部通过 ref 去引用组件时，实际引用的是 HTMLElement

6.函数式组件的 props 可以不用显示声明，所以没有在 props 里面声明的属性都会被自动隐式解析为 prop,而普通组件所有未声明的属性都解析到$attrs 里面，并自动挂载到组件根元素上面(可以通过 inheritAttrs 属性禁止)

优点

1. 由于函数式组件不需要实例化，无状态，没有生命周期，所以渲染性能要好于普通组件
2. 2.函数式组件结构比较简单，代码结构更清晰

使用场景：

一个简单的展示组件，作为容器组件使用 比如 router-view 就是一个函数式组件

“高阶组件”——用于接收一个组件作为参数，返回一个被包装过的组件。

33. 写过自定义指令吗 原理是什么?

指令本质上是装饰器，是 vue 对 HTML 元素的扩展，给 HTML 元素增加自定义功能。vue 编译 DOM 时，会找到指令对象，执行指令的相关方法。

自定义指令有五个生命周期（也叫钩子函数），分别是 bind、inserted、update、componentUpdated、unbind

1. bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。

2. inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。

3. update：被绑定于元素所在的模板更新时调用，而无论绑定值是否变化。通过比较更新前后的绑定值，可以忽略不必要的模板更新。

4. componentUpdated：被绑定元素所在模板完成一次更新周期时调用。

5. unbind：只调用一次，指令与元素解绑时调用。

原理

1.在生成 ast 语法树时，遇到指令会给当前元素添加 directives 属性

2.通过 genDirectives 生成指令代码

3.在 patch 前将指令的钩子提取到 cbs 中,在 patch 过程中调用对应的钩子

4.当执行指令对应钩子函数时，调用对应指令定义的方法

34. Vue3.0 有什么更新?

（1）监测机制的改变

3.0 将带来基于代理 Proxy 的 observer 实现，提供全语言覆盖的反应性跟踪。

消除了 Vue 2 当中基于 Object.defineProperty 的实现所存在的很多限制：

（2）只能监测属性，不能监测对象

检测属性的添加和删除；

检测数组索引和长度的变更；

支持 Map、Set、WeakMap 和 WeakSet。

（3）模板

作用域插槽，2.x 的机制导致作用域插槽变了，父组件会重新渲染，而 3.0 把作用域插槽改成了函数的方式，这样只会影响子组件的重新渲染，提升了渲染的性能。

同时，对于 render 函数的方面，vue3.0 也会进行一系列更改来方便习惯直接使用 api 来生成 vdom 。

（4）对象式的组件声明方式

vue2.x 中的组件是通过声明的方式传入一系列 option，和 TypeScript 的结合需要通过一些装饰器的方式来做，虽然能实现功能，但是比较麻烦。

3.0 修改了组件的声明方式，改成了类式的写法，这样使得和 TypeScript 的结合变得很容易

（5）其它方面的更改

支持自定义渲染器，从而使得 weex 可以通过自定义渲染器的方式来扩展，而不是直接 fork 源码来改的方式。

支持 Fragment（多个根节点）和 Protal（在 dom 其他部分渲染组建内容）组件，针对一些特殊的场景做了处理。

基于 tree shaking 优化，提供了更多的内置功能。

35. Vue3.0 为什么要用 proxy?

在 Vue2 中， 0bject.defineProperty 会改变原始数据，而 Proxy 是创建对象的虚拟表示，并提供 set 、get 和 deleteProperty 等处理器，这些处理器可在访问或修改原始对象上的属性时进行拦截，有以下特点：

不需用使用 Vue.$set 或 Vue.$delete 触发响应式。

全方位的数组变化检测，消除了 Vue2 无效的边界情况。

支持 Map，Set，WeakMap 和 WeakSet。

Proxy 实现的响应式原理与 Vue2 的实现原理相同，实现方式大同小异：

get 收集依赖

Set、delete 等触发依赖

对于集合类型，就是对集合对象的方法做一层包装：原方法执行后执行依赖相关的收集或触发逻辑。

36. Composition API 与 React Hook 很像，区别是什么?

从 React Hook 的实现角度看，React Hook 是根据 useState 调用的顺序来确定下一次重渲染时的 state 是来源于哪个 useState，所以出现了以下限制

不能在循环、条件、嵌套函数中调用 Hook

必须确保总是在你的 React 函数的顶层调用 Hook

useEffect、useMemo 等函数必须手动确定依赖关系

而 Composition API 是基于 Vue 的响应式系统实现的，与 React Hook 的相比

声明在 setup 函数内，一次组件实例化只调用一次 setup，而 React Hook 每次重渲染都需要调用 Hook，使得 React 的 GC 比 Vue 更有压力，性能也相对于 Vue 来说也较慢

Compositon API 的调用不需要顾虑调用顺序，也可以在循环、条件、嵌套函数中使用

响应式系统自动实现了依赖收集，进而组件的部分的性能优化由 Vue 内部自己完成，而 React Hook 需要手动传入依赖，而且必须必须保证依赖的顺序，让 useEffect、useMemo 等函数正确的捕获依赖变量，否则会由于依赖不正确使得组件性能下降。

虽然 Compositon API 看起来比 React Hook 好用，但是其设计思想也是借鉴 React Hook 的。。

37. 参考

https://juejin.cn/post/6844903709055401991#heading-10

第六章 React

一、基础

1. 什么是 React?

React 是 Facebook 在 2011 年开发的前端 JavaScript 库。

它遵循基于组件的方法，有助于构建可重用的 UI 组件。

它用于开发复杂和交互式的 Web 和移动 UI。

尽管它仅在 2015 年开源，但有一个很大的支持社区。

2. React 与 Angular 有何不同?

Angular 是一个成熟的 MVC 框架，带有很多特定的特性，比如服务、指令、模板、模块、解析器等等。React 是一个非常轻量级的库，它只关注 MVC 的视图部分。

Angular 遵循两个方向的数据流，而 React 遵循从上到下的单向数据流。React 在开发特性时给了开发人员很大的自由，例如，调用 API 的方式、路由等等。我们不需要包括路由器库，除非我们需要它在我们的项目。

3. React 有什么特点?

它使用**虚拟 DOM **而不是真正的 DOM。

它可以用服务器端渲染。

它遵循单向数据流或数据绑定。

4. 列出 React 的一些主要优点?

它提高了应用的性能

可以方便地在客户端和服务器端使用

由于 JSX，代码的可读性很好

React 很容易与 Meteor，Angular 等其他框架集成

使用 React，编写 UI 测试用例变得非常容易。

5. React 有哪些限制?

React 只是一个库，而不是一个完整的框架

它的库非常庞大，需要时间来理解

新手程序员可能很难理解

编码变得复杂，因为它使用内联模板和 JSX。

6. 什么是 JSX?

JSX 即 JavaScript XML。一种在 React 组件内部构建标签的类 XML 语法。JSX 为 react.js 开发的一套语法糖，也是 react.js 的使用基础。

7. 浏览器可以读取 JSX 吗？为什么?

不可以。

浏览器只能处理 JavaScript 对象，而不能读取常规 JavaScript 对象中的 JSX。所以为了使浏览器能够读取 JSX，首先，需要用像 Babel 这样的 JSX 转换器将 JSX 文件转换为 JavaScript 对象，然后再将其传给浏览器。

8. 使用 JSX 的好处?

1.更加熟悉：许多团队都包括了非开发人员，例如熟悉 HTML 的 UI 以及 ux 设计师和负责完整测试产品的质量保证人员。使用 JSX 之后，这些团队成员都可以更轻松的阅读和贡献代码。任何熟悉基于 XML 语言的人都可以轻松的掌握 JSX。此外，由于 React 组件囊括了所有可能的 DOM 表现形式（后续详细解释），因此 JSX 能巧妙地用简单明了的方式来展现这种结构。

2.更加语义化：提供更加语义化且易懂的标签(将传统的HTML标签封装成React组件)，我们就可以像使用HTML标签一样使用这个组件。

3.更加直观：JSX 让小组件更加简单、明了、直观。在有上百个组件及更深层标签树的大项目中，这种好处会成倍地放大。在函数作用域内，使用 jsx 语法的版本与使用原生 JavaScript 相比，其标签的意图变得更加直观，可读性也更高。

4.抽象化：对于使用 jsx 的人来说，从 React0.11 升级到 React0.12 是无痛的——不需要修改任何代码。虽然不是灵丹妙药，但是 jsx 提供的抽象能力确实能够减少代码在项目开发过程中的改动。

5.关注点分离：将 HTML 标签以及生成这些标签的代码内在地紧密联系在一起。在 React 内，你不需要把整个程序甚至单个组件的关注点分离成视图和模板文件。相反，React 鼓励你为每一个关注点创造一个独立的组件，并把所有的逻辑和标签封装在其中。

JSX 以干净切简洁的方式保证了组件中的标签与所有业务逻辑的相互分离。它不仅提供了一个清晰、直观的方式来描述组件树，同时还让你的应用程序更加符合逻辑。

9. 你了解 Virtual DOM 吗？解释一下它的工作原理?

Virtual DOM 是一个轻量级的 JavaScript 对象，它最初只是 real DOM 的副本。它是一个节点树，它将元素、它们的属性和内容作为对象及其属性。 React 的渲染函数从 React 组件中创建一个节点树。然后它响应数据模型中的变化来更新该树，该变化是由用户或系统完成的各种动作引起的。

Virtual DOM 工作过程有三个简单的步骤。

每当底层数据发生改变时，整个 UI 都将在 Virtual DOM 描述中重新渲染。

然后计算之前 DOM 表示与新表示的之间的差异。

完成计算后，将只用实际更改的内容更新 real DOM。

10. 区分 Real DOM 和 Virtual DOM?

Real DOM	Virtual DOM
1. 更新缓慢。	1. 更新更快。
2. 可以直接更新 HTML。	2. 无法直接更新 HTML。
3. 如果元素更新，则创建新 DOM。	3. 如果元素更新，则更新 JSX 。
4. DOM 操作代价很高。	4. DOM 操作非常简单。

5. 消耗的内存较多。

5. 很少的内存消耗。

11. 什么是声明式编程?

声明式编程是一种编程范式，它关注的是你要做什么，而不是如何做。它表达逻辑而不显式地定义步骤。这意味着我们需要根据逻辑的计算来声明要显示的组件。它没有描述控制流步骤。

12. 声明式编程 vs 命令式编程?

声明式编程的编写方式描述了应该做什么，而命令式编程描述了如何做。在声明式编程中，让编译器决定如何做事情。声明性程序很容易推理，因为代码本身描述了它在做什么。

13. 什么是函数式编程?

函数式编程是声明式编程的一部分。javascript 中的函数是第一类公民，这意味着函数是数据，你可以像保存变量一样在应用程序中保存、检索和传递这些函数。

函数式编程有些核心的概念，如下：

不可变性(Immutability)

纯函数(Pure Functions)

数据转换(Data Transformations)

高阶函数 (Higher-Order Functions)

递归

组合

14. React 中元素与组件的区别?

React 元素（React element）

它是 React 中最小基本单位，我们可以使用 JSX 语法轻松地创建一个 React 元素，React 元素不是真实的 DOM 元素，它仅仅是 js 的普通对象（plain objects），所以也没办法直接调用 DOM 原生的 API。

除了使用 JSX 语法，我们还可以使用 React.createElement() 和 React.cloneElement() 来构建 React 元素。

React 组件

React 中有三种构建组件的方式。React.createClass()、ES6 class 和无状态函数。

1、React.createClass()

```
var Greeting = React.createClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

2、ES6 class

```
class Greeting extends React.Component{
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
};
```

3、函数(无状态)函数

无状态函数是使用函数构建的无状态组件，无状态组件传入 props 和 context 两个参数，它没有 state，除了 render()，没有其它生命周期方法。

```
function Greeting (props) {
  return <h1>Hello, {props.name}</h1>;
}
```

4、PureComponent

除了为你提供了一个具有浅比较的 shouldComponentUpdate 方法，PureComponent 和 Component 基本上完全相同。

元素与组件的区别

组件是由元素构成的。元素数据结构是普通对象，而组件数据结构是类或纯函数。

15. 你怎么理解“在 React 中，一切都是组件”这句话?

组件是 React 应用 UI 的构建块。这些组件将整个 UI 分成小的独立并可重用的部分。每个组件彼此独立，而不会影响 UI 的其余部分。

16. Component, Element, Instance 之间有什么区别和联系?

元素： 一个元素 element 是一个普通对象(plain object)，描述了对于一个 DOM 节点或者其他组件 component，你想让它在屏幕上呈现成什么样子。元素 element 可以在它的属性 props 中包含其他元素(译注:用于形成元素树)。创建一个 React 元素 element 成本很低。元素 element 创建之后是不可变的。

组件： 一个组件 component 可以通过多种方式声明。可以是带有一个 render()方法的类，简单点也可以定义为一个函数。这两种情况下，它都把属性 props 作为输入，把返回的一棵元素树作为输出。

实例： 一个实例 instance 是你在所写的组件类 component class 中使用关键字 this 所指向的东西(译注:组件实例)。它用来存储本地状态和响应生命周期事件很有用。

函数式组件(Functional component)根本没有实例 instance。类组件(Class component)有实例 instance。

17. React 组件分类？有什么不同类型?

React 中一切都是组件。 我们通常将应用程序的整个逻辑分解为小的单个部分。 我们将每个单独的部分称为组件。 通常，组件是一个 javascript 函数，它接受输入，处理它并返回在 UI 中呈现的 React 元素。

在 React 中有不同类型的组件。让我们详细看看。

函数/无状态/展示组件

函数或无状态组件是一个纯函数，它可接受接受参数，并返回 react 元素。这些都是没有任何副作用的纯函数。这些组件没有状态或生命周期方法，这里有一个例子。

类/有状态组件

类或有状态组件具有状态和生命周期方可能通过 setState()方法更改组件的状态。类组件是通过扩展 React 创建的。它在构造函数中初始化，也可能有子组件,这里有一个例子。

受控组件

受控组件是在 React 中处理输入表单的一种技术。表单元素通常维护它们自己的状态，而 react 则在组件的状态属性中维护状态。我们可以将两者结合起来控制输入表单。这称为受控组件。因此，在受控组件表单中，数据由 React 组件处理。

非受控组件

大多数情况下，建议使用受控组件。有一种称为非受控组件的方法可以通过使用 Ref 来处理表单数据。在非受控组件中，Ref 用于直接从 DOM 访问表单值，而不是事件处理程序。

我们使用 Ref 构建了相同的表单，而不是使用 React 状态。 我们使用 React.createRef() 定义 Ref 并传递该输

入表单并直接从 handleSubmit 方法中的 DOM 访问表单值。

容器组件

容器组件是处理获取数据、订阅 redux 存储等的组件。它们包含展示组件和其他容器组件，但是里面从来没有 html。

高阶组件

高阶组件是将组件作为参数并生成另一个组件的组件。 Redux connect 是高阶组件的示例。 这是一种用于生成可重用组件的强大技术。

18. React 声明组件有哪几种方法，有什么不同?

React 声明组件的三种方式：

函数式定义的无状态组件

ES5 原生方式 React.createClass 定义的组件

ES6 形式的 extends React.Component 定义的组件

（1）无状态函数式组件

它是为了创建纯展示组件，这种组件只负责根据传入的 props 来展示，不涉及到 state 状态的操作

组件不会被实例化，整体渲染性能得到提升，不能访问 this 对象，不能访问生命周期的方法

（2）ES5 原生方式 React.createClass // RFC

React.createClass 会自绑定函数方法，导致不必要的性能开销，增加代码过时的可能性。

（3）E6 继承形式 React.Component // RCC

目前极为推荐的创建有状态组件的方式，最终会取代 React.createClass 形式；相对于 React.createClass 可以更好实现代码复用。

无状态组件相对于于后者的区别：

与无状态组件相比，React.createClass 和 React.Component 都是创建有状态的组件，这些组件是要被实例化的，并且可以访问组件的生命周期方法。

React.createClass 与 React.Component 区别：

① 函数 this 自绑定

React.createClass 创建的组件，其每一个成员函数的 this 都有 React 自动绑定，函数中的 this 会被正确设置。

React.Component 创建的组件，其成员函数不会自动绑定 this，需要开发者手动绑定，否则 this 不能获取当前组件实例对象。

② 组件属性类型 propTypes 及其默认 props 属性 defaultProps 配置不同

React.createClass 在创建组件时，有关组件 props 的属性类型及组件默认的属性会作为组件实例的属性来配置，其中 defaultProps 是使用 getDefaultProps 的方法来获取默认组件属性的

React.Component 在创建组件时配置这两个对应信息时，他们是作为组件类的属性，不是组件实例的属性，也就是所谓的类的静态属性来配置的。

③ 组件初始状态 state 的配置不同

React.createClass 创建的组件，其状态 state 是通过 getInitialState 方法来配置组件相关的状态；

React.Component 创建的组件，其状态 state 是在 constructor 中像初始化组件属性一样声明的。

19. 对有状态组件和无状态组件的理解及使用场景?

（1）有状态组件

特点：

是类组件

有继承

可以使用 this

可以使用 react 的生命周期

使用较多，容易频繁触发生命周期钩子函数，影响性能

内部使用 state，维护自身状态的变化，有状态组件根据外部组件传入的 props 和自身的 state 进行渲染。

使用场景：

需要使用到状态的。

需要使用状态操作组件的（无状态组件的也可以实现新版本 react hooks 也可实现）

总结：

类组件可以维护自身的状态变量，即组件的 state ，类组件还有不同的生命周期方法，可以让开发者能够在组件的不同阶段（挂载、更新、卸载），对组件做更多的控制。类组件则既可以充当无状态组件，也可以充当有状态组件。当一个类组件不需要管理自身状态时，也可称为无状态组件。

（2）无状态组件

特点：

不依赖自身的状态 state

可以是类组件或者函数组件。

可以完全避免使用 this 关键字。（由于使用的是箭头函数事件无需绑定）

有更高的性能。当不需要使用生命周期钩子时，应该首先使用无状态函数组件

组件内部不维护 state ，只根据外部组件传入的 props 进行渲染的组件，当 props 改变时，组件重新渲染。

使用场景：

组件不需要管理 state，纯展示

优点：

简化代码、专注于 render

组件不需要被实例化，无生命周期，提升性能。 输出（渲染）只取决于输入（属性），无副作用

视图和数据的解耦分离

缺点：

无法使用 ref

无生命周期方法

无法控制组件的重渲染，因为无法使用 shouldComponentUpdate 方法，当组件接受到新的属性时则会重渲染。

总结：

组件内部状态且与外部无关的组件，可以考虑用状态组件，这样状态树就不会过于复杂，易于理解和管理。当一个组件不需要管理自身状态时，也就是无状态组件，应该优先设计为函数组件。比如自定义的 <Button/>、 <Input /> 等组件。

20. React 事件机制?

React 并不是将 click 事件绑定到了 div 的真实 DOM 上，而是在 document 处监听了所有的事件，当事件发生并且冒泡到 document 处的时候，React 将事件内容封装并交由真正的处理函数运行。这样的方式不仅仅减少了内

存的消耗，还能在组件挂在销毁时统一订阅和移除事件。

除此之外，冒泡到 document 上的事件也不是原生的浏览器事件，而是由 react 自己实现的合成事件（SyntheticEvent）。因此如果不想要是事件冒泡的话应该调用 event.preventDefault()方法，而不是调用 event.stopProppagation()方法。

JSX 上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 document 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。

另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 event.stopPropagation 是无效的，而应该调用 event.preventDefault。

实现合成事件的目的如下：

合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力；

对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

21. React 的事件和普通的 HTML 事件有什么不同?

区别：

对于事件名称命名方式，原生事件为全小写，react 事件采用小驼峰；

对于事件函数处理语法，原生事件为字符串，react 事件为函数；

react 事件不能采用 return false 的方式来阻止浏览器的默认行为，而必须要地明确地调用 preventDefault()来阻止默认行为。

合成事件是 react 模拟原生 DOM 事件所有能力的一个事件对象，其优点如下：

兼容所有浏览器，更好的跨平台；

将事件统一存放在一个数组，避免频繁的新增与删除（垃圾回收）。

方便 react 统一管理和事务机制。

事件的执行顺序为原生事件先执行，合成事件后执行，合成事件会冒泡绑定到 document 上，所以尽量避免原生事件与合成事件混用，如果原生事件阻止冒泡，可能会导致合成事件不执行，因为需要冒泡到 document 上合成事件才会执行。

22. React 组件中怎么做事件代理？它的原理是什么?

React 基于 Virtual DOM 实现了一个 SyntheticEvent 层（合成事件层），定义的事件处理器会接收到一个合成事件对象的实例，它符合 W3C 标准，且与原生的浏览器事件拥有同样的接口，支持冒泡机制，所有的事件都自动绑定在最外层上。

在 React 底层，主要对合成事件做了两件事：

事件委派： React 会把所有的事件绑定到结构的最外层，使用统一的事件监听器，这个事件监听器上维持了一个映射来保存所有组件内部事件监听和处理函数。

自动绑定： React 组件中，每个方法的上下文都会指向该组件的实例，即自动绑定 this 为当前组件。

23. React 中什么是受控组件和非控组件?

（1）受控组件

在使用表单来收集用户输入时，例如<input><select><textearea>等元素都要绑定一个 change 事件，当表单的

状态发生变化，就会触发 onChange 事件，更新组件的 state。这种组件在 React 中被称为受控组件，在受控组件中，组件渲染出的状态与它的 value 或 checked 属性相对应，react 通过这种方式消除了组件的局部状态，使整个状态可控。react 官方推荐使用受控表单组件。

受控组件更新 state 的流程：

可以通过初始 state 中设置表单的默认值

每当表单的值发生变化时，调用 onChange 事件处理器

事件处理器通过事件对象 e 拿到改变后的状态，并更新组件的 state

一旦通过 setState 方法更新 state，就会触发视图的重新渲染，完成表单组件的更新

受控组件缺陷：

表单元素的值都是由 React 组件进行管理，当有多个输入框，或者多个这种组件时，如果想同时获取到全部的值就必须每个都要编写事件处理函数，这会让代码看着很臃肿，所以为了解决这种情况，出现了非受控组件。

（2）非受控组件

如果一个表单组件没有 value props（单选和复选按钮对应的是 checked props）时，就可以称为非受控组件。在非受控组件中，可以使用一个 ref 来从 DOM 获得表单值。而不是为每个状态更新编写一个事件处理程序。

24. React 高阶组件是什么，和普通组件有什么区别，适用什么场景?

官方解释：

高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

高阶组件不是组件，是 增强函数，可以输入一个元组件，返回出一个新的增强组件

属性代理 (Props Proxy)

在我看来属性代理就是提取公共的数据和方法到父组件，子组件只负责渲染数据，相当于设计模式里的模板模式，这样组件的重用性就更高了

```
function proxyHoc(WrappedComponent) {
return class extends React.Component {
    render() {
        const newProps = {
            count: 1
        }
        return <WrappedComponent {...this.props} {...newProps} />
    }
}
}
```

反向继承

```
const MyContainer = (WrappedComponent)=>{
    return class extends WrappedComponent {
        render(){
```

```
                return super.render();
            }
        }
    }
```

高阶组件（HOC）就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件，它只是一种组件的设计模式，这种设计模式是由 react 自身的组合性质必然产生的。我们将它们称为纯组件，因为它们可以接受任何动态提供的子组件，但它们不会修改或复制其输入组件中的任何行为。

```
// hoc 的定义
function withSubscription(WrappedComponent, selectData) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
    // 一些通用的逻辑处理
    render() {
      // … 并使用新数据渲染被包装的组件!
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };

// 使用
const BlogPostWithSubscription = withSubscription(BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id));
```

1）HOC 的优缺点

优点： 逻辑服用、不影响被包裹组件的内部逻辑。

缺点：hoc 传递给被包裹组件的 props 容易和被包裹后的组件重名，进而被覆盖

2）适用场景

代码复用，逻辑抽象

渲染劫持

State 抽象和更改

Props 更改

3）具体应用例子

权限控制： 利用高阶组件的 条件渲染 特性可以对页面进行权限控制，权限控制一般分为两个维度：页面级别和 页面元素级别

```
// HOC.js
function withAdminAuth(WrappedComponent) {
    return class extends React.Component {
        state = {
            isAdmin: false,
        }
        async UNSAFE_componentWillMount() {
            const currentRole = await getCurrentUserRole();
            this.setState({
                isAdmin: currentRole === 'Admin',
            });
        }
        render() {
            if (this.state.isAdmin) {
                return <WrappedComponent {...this.props} />;
            } else {
                return (<div>您没有权限查看该页面，请联系管理员！</div>);
            }
        }
    };
}

// pages/page-a.js
class PageA extends React.Component {
    constructor(props) {
        super(props);
        // something here...
    }
    UNSAFE_componentWillMount() {
        // fetching data
    }
    render() {
        // render page with data
    }
}
export default withAdminAuth(PageA);

// pages/page-b.js
```

```
class PageB extends React.Component {
    constructor(props) {
        super(props);
    // something here...
        }
    UNSAFE_componentWillMount() {
    // fetching data
    }
    render() {
    // render page with data
    }
}
export default withAdminAuth(PageB);
```

组件渲染性能追踪： 借助父组件子组件生命周期规则捕获子组件的生命周期，可以方便的对某个组件的渲染时间进行记录：

```
class Home extends React.Component {
        render() {
            return (<h1>Hello World.</h1>);
        }
    }
    function withTiming(WrappedComponent) {
        return class extends WrappedComponent {
            constructor(props) {
                super(props);
                this.start = 0;
                this.end = 0;
            }
            UNSAFE_componentWillMount() {
                super.componentWillMount && super.componentWillMount();
                this.start = Date.now();
            }
            componentDidMount() {
                super.componentDidMount && super.componentDidMount();
                this.end = Date.now();
                console.log(`${WrappedComponent.name} 组件渲染时间为 ${this.end - this.start} ms`);
            }
            render() {
                return super.render();
```

```
            }
        };
    }

    export default withTiming(Home);
```

注意：withTiming 是利用 反向继承 实现的一个高阶组件，功能是计算被包裹组件（这里是 Home 组件）的渲染时间。

页面复用

```
const withFetching = fetching => WrappedComponent => {
    return class extends React.Component {
        state = {
            data: [],
        }
        async UNSAFE_componentWillMount() {
            const data = await fetching();
            this.setState({
                data,
            });
        }
        render() {
            return <WrappedComponent data={this.state.data} {...this.props} />;
        }
    }
}

// pages/page-a.js
export default withFetching(fetching('science-fiction'))(MovieList);
// pages/page-b.js
export default withFetching(fetching('action'))(MovieList);
// pages/page-other.js
export default withFetching(fetching('some-other-type'))(MovieList);
```

25. 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）

26. 什么是 Props?

Props 是 React 中属性的简写。它们是只读组件，必须保持纯，即不可变。它们总是在整个应用中从父组件传递到子组件。子组件永远不能将 prop 送回父组件。这有助于维护单向数据流，通常用于呈现动态生成的数据。

27. React 中的状态是什么？它是如何使用的?

状态是 React 组件的核心，是数据的来源，必须尽可能简单。基本上状态是确定组件呈现和行为的对象。与 props 不同，它们是可变的，并创建动态和交互式组件。可以通过 this.state() 访问它们。。

28. Props 和 State 的区别是什么?

条件	State	Props
1. 从父组件中接收初始值	Yes	Yes
2. 父组件可以改变值	No	Yes
3. 在组件中设置默认值	Yes	Yes
4. 在组件的内部变化	Yes	No
5. 设置子组件的初始值	Yes	Yes
6. 在子组件的内部更改	No	Yes

29. 如何更新状态以及如何不更新?

this.setState()

你不应该直接修改状态。可以在构造函数中定义状态值。直接使用状态不会触发重新渲染。React 使用 this.setState()时合并状态。

```
//  错误方式
this.state.name = "some name"
//  正确方式
this.setState({name:"some name"})
```

使用 this.setState()的第二种形式总是更安全的，因为更新的 props 和状态是异步的。这里，我们根据这些 props 更新状态。

```
// 错误方式
this.setState({
    timesVisited: this.state.timesVisited + this.props.count
})
// 正确方式
this.setState((state, props) => {
    timesVisited: state.timesVisited + props.count
});
```

30. setState 是同步还是异步的?

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，形式了所谓的“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。

31. 哪些方法会触发 React 重新渲染？重新渲染 render 会做些什么？

（1）哪些方法会触发 react 重新渲染?

setState（）方法被调用、父组件重新渲染

setState 是 React 中最常用的命令，通常情况下，执行 setState 会触发 render。但是这里有个点值得关注，执行 setState 的时候不一定会重新渲染。当 setState 传入 null 时，并不会触发 render。

```
class App extends React.Component {
  state = {
    a: 1
  };
  render() {
    console.log("render");
    return (
      <React.Fragement>
        <p>{this.state.a}</p>
        <button
          onClick={() => {
            this.setState({ a: 1 }); // 这里并没有改变 a 的值
          }}
        >
          Click me
        </button>
        <button onClick={() => this.setState(null)}>setState null</button>
        <Child />
      </React.Fragement>
    );
  }
}
```

父组件重新渲染

只要父组件重新渲染了，即使传入子组件的 props 未发生变化，那么子组件也会重新渲染，进而触发 render

（2）重新渲染 render 会做些什么?

会对新旧 VNode 进行对比，也就是我们所说的 Diff 算法。

对新旧两棵树进行一个深度优先遍历，这样每一个节点都会一个标记，在到深度遍历的时候，每遍历到一和个节点，就把该节点和新的节点树进行对比，如果有差异就放到一个对象里面

遍历差异对象，根据差异的类型，根据对应对规则更新 VNode

React 的处理 render 的基本思维模式是每次一有变动就会去重新渲染整个应用。在 Virtual DOM 没有出现

之前，最简单的方法就是直接调用 innerHTML。Virtual DOM 厉害的地方并不是说它比直接操作 DOM 快，而是说不管数据怎么变，都会尽量以最小的代价去更新 DOM。React 将 render 函数返回的虚拟 DOM 树与老的进行比较，从而确定 DOM 要不要更新、怎么更新。当 DOM 树很大时，遍历两棵树进行各种比对还是相当耗性能的，特别是在顶层 setState 一个微小的修改，默认会去遍历整棵树。尽管 React 使用高度优化的 Diff 算法，但是这个过程仍然会损耗性能。

32. React 如何判断什么时候重新渲染组件?

组件状态的改变可以因为 props 的改变，或者直接通过 setState 方法改变。组件获得新的状态，然后 React 决定是否应该重新渲染组件。只要组件的 state 发生变化，React 就会对组件进行重新渲染。这是因为 React 中的 shouldComponentUpdate 方法默认返回 true，这就是导致每次更新都重新渲染的原因。

当 React 将要渲染组件时会执行 shouldComponentUpdate 方法来看它是否返回 true（组件应该更新，也就是重新渲染）。所以需要重写 shouldComponentUpdate 方法让它根据情况返回 true 或者 false 来告诉 React 什么时候重新渲染什么时候跳过重新渲染。

33. 对 React 中 Fragment 的理解，它的使用场景是什么?

简写<></>

在 React 中，组件返回的元素只能有一个根元素。为了不添加多余的 DOM 节点，我们可以使用 Fragment 标签来包裹所有的元素，Fragment 标签不会渲染出任何元素。React 官方对 Fragment 的解释：

React 中的一个常见模式是一个组件返回多个元素。Fragments 允许你将子列表分组，而无需向 DOM 添加额外节点。

34. React 如何获取组件对应的 DOM 元素?

可以用 ref 来获取某个子节点的实例，然后通过当前 class 组件实例的一些特定属性来直接获取子节点实例。ref 有三种实现方法:

字符串格式：字符串格式，这是 React16 版本之前用得最多的，例如： span

函数格式：ref 对应一个方法，该方法有一个参数，也就是对应的节点实例，例如：<p ref={ele => this.info = ele}>

createRef 方法：React 16 提供的一个 API，使用 React.createRef()来实现。

35. 你对 React 的 refs 有什么了解? 列出一些应该使用 Refs 的情况?

Refs 是 React 中引用的简写。它是一个有助于存储对特定的 React 元素或组件的引用的属性，它将由组件渲染配置函数返回。用于对 render() 返回的特定元素或组件的引用。当需要进行 DOM 测量或向组件添加方法时，它们会派上用场。

以下是应该使用 refs 的情况：

需要管理焦点、选择文本或媒体播放时

触发式动画

与第三方 DOM 库集成

不应该过度的使用 Refs

ref 的返回值取决于节点的类型：

当 ref 属性被用于一个普通的 HTML 元素时，React.createRef() 将接收底层 DOM 元素作为他的 current 属性以创建 ref。

当 ref 属性被用于一个自定义的类组件时，ref 对象将接收该组件已挂载的实例作为他的 current。

当在父组件中需要访问子组件中的 ref 时可使用传递 Refs 或回调 Refs。

36. React 中可以在 render 访问 refs 吗？为什么?

render 阶段 DOM 还没有生成，无法获取 DOM。DOM 的获取需要在 pre-commit 阶段和 commit 阶段。

37. 对 React 的插槽(Portals)的理解，如何使用，有哪些使用场景?

React 官方对 Portals 的定义：

Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀的方案

Portals 是 React 16 提供的官方解决方案，使得组件可以脱离父组件层级挂载在 DOM 树的任何位置。通俗来讲，就是我们 render 一个组件，但这个组件的 DOM 结构并不在本组件内。

Portals 语法如下：

ReactDOM.createPortal(child, container);

第一个参数 child 是可渲染的 React 子项，比如元素，字符串或者片段等;

第二个参数 container 是一个 DOM 元素。

一般情况下，组件的 render 函数返回的元素会被挂载在它的父级组件上：

```
import DemoComponent from './DemoComponent';
render() {
  // DemoComponent 元素会被挂载在 id 为 parent 的 div 的元素上
  return (
    <div id="parent">
        <DemoComponent />
    </div>
  );
}
```

然而，有些元素需要被挂载在更高层级的位置。最典型的应用场景：当父组件具有 overflow: hidden 或者 z-index 的样式设置时，组件有可能被其他元素遮挡，这时就可以考虑要不要使用 Portal 使组件的挂载脱离父组件。例如：对话框，模态窗。

```
import DemoComponent from './DemoComponent';
render() {
  // DemoComponent 元素会被挂载在 id 为 parent 的 div 的元素上
  return (
    <div id="parent">
        <DemoComponent />
    </div>
  );
}
```

38. 在 React 中如何避免不必要的 render?

React 基于虚拟 DOM 和高效 Diff 算法的完美配合，实现了对 DOM 最小粒度的更新。大多数情况下，React 对 DOM 的渲染效率足以业务日常。但在个别复杂业务场景下，性能问题依然会困扰我们。此时需要采取一些措施来提升运行性能，其很重要的一个方向，就是避免不必要的渲染（Render）。这里提下优化的点：

shouldComponentUpdate 和 PureComponent

在 React 类组件中，可以利用 shouldComponentUpdate 或者 PureComponent 来减少因父组件更新而触发子组件的 render，从而达到目的。shouldComponentUpdate 来决定是否组件是否重新渲染，如果不希望组件重新渲染，返回 false 即可。

利用高阶组件

在函数组件中，并没有 shouldComponentUpdate 这个生命周期，可以利用高阶组件，封装一个类似 PureComponet 的功能

使用 React.memo

React.memo 是 React 16.6 新的一个 API，用来缓存组件的渲染，避免不必要的更新，其实也是一个高阶组件，与 PureComponent 十分类似，但不同的是， React.memo 只能用于函数组件。

39. 对 React-Intl 的理解，它的工作原理?

React-intl 是雅虎的语言国际化开源项目 FormatJS 的一部分，通过其提供的组件和 API 可以与 ReactJS 绑定。

React-intl 提供了两种使用方法，一种是引用 React 组件，另一种是直接调取 API，官方更加推荐在 React 项目中使用前者，只有在无法使用 React 组件的地方，才应该调用框架提供的 API。它提供了一系列的 React 组件，包括数字格式化、字符串格式化、日期格式化等。

在 React-intl 中，可以配置不同的语言包，他的工作原理就是根据需要，在语言包之间进行切换。

40. 对 React context 的理解?

在 React 中，数据传递一般使用 props 传递数据，维持单向数据流，这样可以让组件之间的关系变得简单且可预测，但是单项数据流在某些场景中并不适用。单纯一对的父子组件传递并无问题，但要是组件之间层层依赖深入，props 就需要层层传递显然，这样做太繁琐了。

Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

可以把 context 当做是特定一个组件树内共享的 store，用来做数据传递。简单说就是，当你不想在组件树中通过逐层传递 props 或者 state 的方式来传递数据时，可以使用 Context 来实现跨层级的组件数据传递。

JS 的代码块在执行期间，会创建一个相应的作用域链，这个作用域链记录着运行时 JS 代码块执行期间所能访问的活动对象，包括变量和函数，JS 程序通过作用域链访问到代码块内部或者外部的变量和函数。

假如以 JS 的作用域链作为类比，React 组件提供的 Context 对象其实就好比一个提供给子组件访问的作用域，而 Context 对象的属性可以看成作用域上的活动对象。由于组件 的 Context 由其父节点链上所有组件通 过 getChildContext（）返回的 Context 对象组合而成，所以，组件通过 Context 是可以访问到其父组件链上所有节点组件提供的 Context 的属性。

41. 为什么 React 并不推荐优先考虑使用 Context?

Context 目前还处于实验阶段，可能会在后面的发行版本中有很大的变化，事实上这种情况已经发生了，所以为了避免给今后升级带来大的影响和麻烦，不建议在 app 中使用 context。

尽管不建议在 app 中使用 context，但是独有组件而言，由于影响范围小于 app，如果可以做到高内聚，不破坏组件树之间的依赖关系，可以考虑使用 context

对于组件之间的数据通信或者状态管理，有效使用 props 或者 state 解决，然后再考虑使用第三方的成熟库进行解决，以上的方法都不是最佳的方案的时候，在考虑 context。

context 的更新需要通过 setState()触发，但是这并不是很可靠的，Context 支持跨组件的访问，但是如果中间的子组件通过一些方法不影响更新，比如 shouldComponentUpdate() 返回 false 那么不能保证 Context 的更新一定可以使用 Context 的子组件，因此，Context 的可靠性需要关注。

42. React 组件的构造函数有什么作用？它是必须的吗?

构造函数主要用于两个目的：

通过将对象分配给 this.state 来初始化本地状态

将事件处理程序方法绑定到实例上

所以，当在 React class 中需要设置 state 的初始值或者绑定事件时，需要加上构造函数。

构造函数用来新建父类的 this 对象；子类必须在 constructor 方法中调用 super 方法；否则新建实例时会报错；因为子类没有自己的 this 对象，而是继承父类的 this 对象，然后对其进行加工。如果不调用 super 方法；子类就得不到 this 对象。

注意：

constructor () 必须配上 super(), 如果要在 constructor 内部使用 this.props 就要 传入 props，否则不用

JavaScript 中的 bind 每次都会返回一个新的函数, 为了性能等考虑, 尽量在 constructor 中绑定事件。

43. React.forwardRef 是什么？它有什么作用?

React.forwardRef 会创建一个 React 组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中。这种技术并不常见，但在以下两种场景中特别有用：

转发 refs 到 DOM 组件

在高阶组件中转发 refs。

44. 如何在 React 中使用 innerHTML?

增加 dangerouslySetInnerHTML 属性，并且传入对象的属性名叫_html

```
function Component(props){
    return <div dangerouslySetInnerHTML={{_html:'<span>你好</span>'}}></div>
}
```

二、React-Fiber

45. 对 React-Fiber 的理解，它解决了什么问题?

React V15 在渲染时，会递归比对 VirtualDOM 树，找出需要变动的节点，然后同步更新它们， 一气呵成。这个过程期间， React 会占据浏览器资源，这会导致用户触发的事件得不到响应，并且会导致掉帧，导致用户感觉到卡顿。

为了给用户制造一种应用很快的“假象”，不能让一个任务长期霸占着资源。 可以将浏览器的渲染、布局、绘制、资源加载(例如 HTML 解析)、事件响应、脚本执行视作操作系统的“进程”，需要通过某些调度策略合理地分配 CPU 资源，从而提高浏览器的用户响应速率, 同时兼顾任务执行效率。

所以 React 通过 Fiber 架构，让这个执行过程变成可被中断。“适时”地让出 CPU 执行权，除了可以让浏览器及时地响应用户的交互，还有其他好处:

分批延时对 DOM 进行操作，避免一次性操作大量 DOM 节点，可以得到更好的用户体验；

给浏览器一点喘息的机会，它会对代码进行编译优化（JIT）及进行热代码优化，或者对 reflow 进行修正。

核心思想: Fiber 也称协程或者纤程。它和线程并不一样，协程本身是没有并发或者并行能力的（需要配合线程），它只是一种控制流程的让出机制。让出 CPU 的执行权，让 CPU 能在这段时间执行其他的操作。渲染的过程可以被中断，可以将控制权交回浏览器，让位给高优先级的任务，浏览器空闲后再恢复渲染。

三、数据管理 state、setState、props

46. React setState 调用的原理?

首先调用了 setState 入口函数，入口函数在这里就是充当一个分发器的角色，根据入参的不同，将其分发到不同的功能函数中去；

```
ReactComponent.prototype.setState = function (partialState, callback) {
  this.updater.enqueueSetState(this, partialState);
  if (callback) {
    this.updater.enqueueCallback(this, callback, 'setState');
  }
};
```

enqueueSetState 方法将新的 state 放进组件的状态队列里，并调用 enqueueUpdate 来处理将要更新的实例对象；

```
enqueueSetState: function (publicInstance, partialState) {
  // 根据 this 拿到对应的组件实例
  var internalInstance = getInternalInstanceReadyForUpdate(publicInstance, 'setState');
  // 这个 queue 对应的就是一个组件实例的 state 数组
  var queue = internalInstance._pendingStateQueue || (internalInstance._pendingStateQueue = []);
  queue.push(partialState);
  //  enqueueUpdate 用来处理当前的组件实例
  enqueueUpdate(internalInstance);
}
```

在 enqueueUpdate 方法中引出了一个关键的对象——batchingStrategy，该对象所具备的 isBatchingUpdates 属性直接决定了当下是要走更新流程，还是应该排队等待；如果轮到执行，就调用 batchedUpdates 方法来直接发起更新流程。由此可以推测，batchingStrategy 或许正是 React 内部专门用于管控批量更新的对象。

```
function enqueueUpdate(component) {
  ensureInjected();
  // 注意这一句是问题的关键，isBatchingUpdates 标识着当前是否处于批量创建/更新组件的阶段
  if (!batchingStrategy.isBatchingUpdates) {
    // 若当前没有处于批量创建/更新组件的阶段，则立即更新组件
    batchingStrategy.batchedUpdates(enqueueUpdate, component);
    return;
  }
  // 否则，先把组件塞入 dirtyComponents 队列里，让它“再等等”
  dirtyComponents.push(component);
  if (component._updateBatchNumber == null) {
    component._updateBatchNumber = updateBatchNumber + 1;
  }
}
```

注意：batchingStrategy 对象可以理解为“锁管理器”。这里的“锁”，是指 React 全局唯一的 isBatchingUpdates 变量，isBatchingUpdates 的初始值是 false，意味着“当前并未进行任何批量更新操作”。每当 React 调用 batchedUpdate 去执行更新动作时，会先把这个锁给“锁上”（置为 true），表明“现在正处于批量更新过程中”。当锁被“锁上”的时候，任何需要更新的组件都只能暂时进入 dirtyComponents 里排队等候下一次的批量更新，而不能随意“插队”。此处体现的“任务锁”的思想，是 React 面对大量状态仍然能够实现有序分批处理的基石。

47. React setState 调用之后发生了什么？是同步还是异步？

（1）React 中 setState 后发生了什么

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。

如果在短时间内频繁 setState。React 会将 state 的改变压入栈中，在合适的时机，批量更新 state 和视图，达到提高性能的效果。

（2）setState 是同步还是异步的

假如所有 setState 是同步的，意味着每执行一次 setState 时（有可能一个同步代码中，多次 setState），都重新 vnode diff + dom 修改，这对性能来说是极为不好的。如果是异步，则可以把一个同步代码中的多个 setState 合并成一次组件更新。所以默认是异步的，但是在一些情况下是同步的。

setState 并不是单纯同步/异步的，它的表现会因调用场景的不同而不同。在源码中，通过 isBatchingUpdates 来判断 setState 是先存进 state 队列还是直接更新，如果值为 true 则执行异步操作，为 false 则直接更新。

异步： 在 React 可以控制的地方，就为 true，比如在 React 生命周期事件和合成事件中，都会走合并操作，延迟更新的策略。

同步： 在 React 无法控制的地方，比如原生事件，具体就是在 addEventListener 、setTimeout、setInterval 等事件中，就只能同步更新。

一般认为，做异步设计是为了性能优化、减少渲染次数：

setState 设计为异步，可以显著的提升性能。如果每次调用 setState 都进行一次更新，那么意味着 render 函数会被频繁调用，界面重新渲染，这样效率是很低的；最好的办法应该是获取到多个更新，之后进行批量更新；

如果同步更新了 state，但是还没有执行 render 函数，那么 state 和 props 不能保持同步。state 和 props 不能保持一致性，会在开发中产生很多的问题。

48. React 中的 setState 批量更新的过程是什么？

调用 setState 时，组件的 state 并不会立即改变， setState 只是把要修改的 state 放入一个队列， React 会优化真正的执行时机，并出于性能原因，会将 React 事件处理程序中的多次 React 事件处理程序中的多次 setState 的状态修改合并成一次状态修改。 最终更新只产生一次组件及其子组件的重新渲染，这对于大型应用程序中的性能提升至关重要。

需要注意的是，只要同步代码还在执行，“攒起来”这个动作就不会停止。（注：这里之所以多次 +1 最终只有一次生效，是因为在同一个方法中多次 setState 的合并动作不是单纯地将更新累加。比如这里对于相同属性的设置，React 只会为其保留最后一次的更新）。

49. React 中有使用过 getDefaultProps 吗？它有什么作用?

通过实现组件的 getDefaultProps，对属性设置默认值（ES5 的写法）：

```
var ShowTitle = React.createClass({
  getDefaultProps:function(){
    return{
      title : "React"
    }
  },
  render : function(){
    return <h1>{this.props.title}</h1>
  }
});
```

50. React 中 setState 的第二个参数作用是什么?

setState 的第二个参数是一个可选的回调函数。这个回调函数将在组件重新渲染后执行。等价于在 componentDidUpdate 生命周期内执行。通常建议使用 componentDidUpdate 来代替此方式。在这个回调函数中你可以拿到更新后 state 的值。

51. React 中的 setState 和 replaceState 的区别是什么?

（1）setState()

setState()用于设置状态对象，其语法如下：

setState(object nextState[, function callback])

nextState，将要设置的新状态，该状态会和当前的 state 合并

callback，可选参数，回调函数。该函数会在 setState 设置成功，且组件重新渲染后调用。

合并 nextState 和当前 state，并重新渲染组件。setState 是 React 事件处理函数中和请求回调函数中触发 UI 更新的主要方法。

（2）replaceState()

replaceState()方法与 setState()类似，但是方法只会保留 nextState 中状态，原 state 不在 nextState 中的状态都会被删除。其语法如下：

replaceState(object nextState[, function callback])

nextState，将要设置的新状态，该状态会替换当前的 state。

callback，可选参数，回调函数。该函数会在 replaceState 设置成功，且组件重新渲染后调用。

总结：setState 是修改其中的部分状态，相当于 Object.assign，只是覆盖，不会减少原来的状态。而 replaceState 是完全替换原来的状态，相当于赋值，将原来的 state 替换为另一个对象，如果新状态属性减少，那么 state 中就没有这个状态了。

52. 在 React 中组件的 this.state 和 setState 有什么区别？？

this.state 通常是用来初始化 state 的，this.setState 是用来修改 state 值的。如果初始化了 state 之后再使用 this.state，之前的 state 会被覆盖掉，如果使用 this.setState，只会替换掉相应的 state 值。所以，如果想要修改 state 的值，就需要使用 setState，而不能直接修改 state，直接修改 state 之后页面是不会更新的。

53. state 是怎么注入到组件的，从 reducer 到组件经历了什么样的过程?

通过 connect 和 mapStateToProps 将 state 注入到组件中：

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '@/reducers/Todo/actions'
import Link from '@/containers/Todo/components/Link'

const mapStateToProps = (state, ownProps) => ({
    active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
    setFilter: () => {
        dispatch(setVisibilityFilter(ownProps.filter))
    }
})

export default connect(
    mapStateToProps,
    mapDispatchToProps
)(Link)
```

上面代码中，active 就是注入到 Link 组件中的状态。 mapStateToProps（state，ownProps）中带有两个参数，含义是：

state-store 管理的全局状态对象，所有都组件状态数据都存储在该对象中。

ownProps 组件通过 props 传入的参数。

reducer 到组件经历的过程：

reducer 对 action 对象处理，更新组件状态，并将新的状态值返回 store。

通过 connect（mapStateToProps，mapDispatchToProps）（Component）对组件 Component 进行升级，此时将状态值从 store 取出并作为 props 参数传递到组件。

高阶组件实现源码：

```
import React from 'react'
import PropTypes from 'prop-types'

// 高阶组件 contect
export const connect = (mapStateToProps, mapDispatchToProps) => (WrappedComponent) => {
    class Connect extends React.Component {
        // 通过对 context 调用获取 store
        static contextTypes = {
            store: PropTypes.object
        }
```

```
        constructor() {
            super()
            this.state = {
                allProps: {}
            }
        }

        // 第一遍需初始化所有组件初始状态
        componentWillMount() {
            const store = this.context.store
            this._updateProps()
            store.subscribe(() => this._updateProps()); // 加入_updateProps()至 store 里的监听事件列表
        }

        // 执行 action 后更新 props，使组件可以更新至最新状态（类似于 setState）
        _updateProps() {
            const store = this.context.store;
            let stateProps = mapStateToProps ?
                mapStateToProps(store.getState(), this.props) : {} // 防止 mapStateToProps 没有传入
            let dispatchProps = mapDispatchToProps ?
                mapDispatchToProps(store.dispatch, this.props) : {
                                        dispatch: store.dispatch
                                    } // 防止 mapDispatchToProps 没有传入
            this.setState({
                allProps: {
                    ...stateProps,
                    ...dispatchProps,
                    ...this.props
                }
            })
        }

        render() {
            return <WrappedComponent {...this.state.allProps} />
        }
    }
    return Connect
}
```

54. React 组件的 state 和 props 有什么区别?

（1）props

props 是一个从外部传进组件的参数，主要作为就是从父组件向子组件传递数据，它具有可读性和不变性，只能通过外部组件主动传入新的 props 来重新渲染子组件，否则子组件的 props 以及展现形式不会改变。

（2）state

state 的主要作用是用于组件保存、控制以及修改自己的状态，它只能在 constructor 中初始化，它算是组件的私有属性，不可通过外部访问和修改，只能通过组件内部的 this.setState 来修改，修改 state 属性会导致组件的重新渲染。

（3）区别

props 是传递给组件的（类似于函数的形参），而 state 是在组件内被组件自己管理的（类似于在一个函数内声明的变量）。

props 是不可修改的，所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。

state 是在组件中创建的，一般在 constructor 中初始化 state。state 是多变的、可以修改，每次 setState 都异步更新的。

55. React 中的 props 为什么是只读的?

this.props 是组件之间沟通的一个接口，原则上来讲，它只能从父组件流向子组件。React 具有浓重的函数式编程的思想。

提到函数式编程就要提一个概念：纯函数。它有几个特点：

给定相同的输入，总是返回相同的输出。

过程没有副作用。

不依赖外部状态。

this.props 就是汲取了纯函数的思想。props 的不可以变性就保证的相同的输入，页面显示的内容是一样的，并且不会产生副作用。

56. 在 React 中组件的 props 改变时更新组件的有哪些方法?

在一个组件传入的 props 更新时重新渲染该组件常用的方法是在 componentWillReceiveProps 中将新的 props 更新到组件的 state 中（这种 state 被成为派生状态（Derived State）），从而实现重新渲染。React 16.3 中还引入了一个新的钩子函数 getDerivedStateFromProps 来专门实现这一需求。

（1）componentWillReceiveProps（已废弃）

在 react 的 componentWillReceiveProps(nextProps)生命周期中，可以在子组件的 render 函数执行前，通过 this.props 获取旧的属性，通过 nextProps 获取新的 props，对比两次 props 是否相同，从而更新子组件自己的 state。

这样的好处是，可以将数据请求放在这里进行执行，需要传的参数则从 componentWillReceiveProps(nextProps)中获取。而不必将所有的请求都放在父组件中。于是该请求只会在该组件渲染时才会发出，从而减轻请求负担。

（2）getDerivedStateFromProps（16.3 引入）

这个生命周期函数是为了替代 componentWillReceiveProps 存在的，所以在需要使用 componentWillReceiveProps 时，就可以考虑使用 getDerivedStateFromProps 来进行替代。

两者的参数是不相同的，而 getDerivedStateFromProps 是一个静态函数，也就是这个函数不能通过 this 访问到 class 的属性，也并不推荐直接访问属性。而是应该通过参数提供的 nextProps 以及 prevState 来进行判断，根据新传入的 props 来映射到 state。

需要注意的是，如果 props 传入的内容不需要影响到你的 state，那么就需要返回一个 null，这个返回值是必

须的，所以尽量将其写到函数的末尾：

```
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的 type 发生变化的时候，更新 state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于 state 不进行任何操作
    return null;
}
```

57. React 中怎么检验 props？验证 props 的目的是什么?

使用 PropTypes

React 为我们提供了 PropTypes 以供验证使用。当我们向 Props 传入的数据无效（向 Props 传入的数据类型和验证的数据类型不符）就会在控制台发出警告信息。它可以避免随着应用越来越复杂从而出现的问题。并且，它还可以让程序变得更易读。

```
import PropTypes from 'prop-types';
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}
Greeting.propTypes = {
  name: PropTypes.string
};
```

当然，如果项目汇中使用了 TypeScript，那么就可以不用 PropTypes 来校验，而使用 TypeScript 定义接口来校验 props。

四、生命周期

1. React 的生命周期怎么划分?

React 通常将组件生命周期分为三个阶段：

装载阶段（Mount），组件第一次在 DOM 树中被渲染的过程；

更新过程（Update），组件状态发生变化，重新更新渲染的过程；

卸载过程（Unmount），组件从 DOM 树中被移除的过程；

React 常见生命周期的过程大致如下：

挂载阶段，首先执行 constructor 构造方法，来创建组件

创建完成之后，就会执行 render 方法，该方法会返回需要渲染的内容

随后，React 会将需要渲染的内容挂载到 DOM 树上

挂载完成之后就会执行 componentDidMount 生命周期函数

如果我们给组件创建一个 props（用于组件通信）、调用 setState（更改 state 中的数据）、调用 forceUpdate（强制更新组件）时，都会重新调用 render 函数

render 函数重新执行之后，就会重新进行 DOM 树的挂载

挂载完成之后就会执行 componentDidUpdate 生命周期函数

当移除组件时，就会执行 componentWillUnmount 生命周期函数

2. React >=16.4 的生命周期?

参考：https://juejin.cn/post/6976593900521816072#heading-1

https://juejin.cn/post/6892604247893147656#heading-5

React 组件挂载阶段

- Constructor
- getDerivedStateFromProps
- componentDidMount
- Render
- componentDidMount

React 组件更新阶段

- getDerivedStateFromProps
- shouldComponentUpdate
- Render
- getSnapshotBeforeUpdate
- componentDidUpdate

React 组件卸载阶段

- componentWillUnmount

3. React 废弃了哪些生命周期？为什么?

被废弃的三个函数都是在 render 之前，因为 fber 的出现，很可能因为高优先级任务的出现而打断现有任务导致它们会被执行多次。另外的一个原因则是，React 想约束使用者，好的框架能够让人不得已写出容易维护和扩展的代码，这一点又是从何谈起，可以从新增加以及即将废弃的生命周期分析入手

1) componentWillMount

首先这个函数的功能完全可以使用 componentDidMount 和 constructor 来代替，异步获取的数据的情况上面已经说明了，而如果抛去异步获取数据，其余的即是初始化而已，这些功能都可以在 constructor 中执行，除此之外，如果在 willMount 中订阅事件，但在服务端这并不会执行 willUnMount 事件，也就是说服务端会导致内存泄漏所以 componentWillMount 完全可以不使用，但使用者有时候难免因为各 种各样的情况在 componentWilMount 中做一些操作，那么 React 为了约束开发者，干脆就抛掉了这个 API

2) componentWillReceiveProps

在老版本的 React 中，如果组件自身的某个 state 跟其 props 密切相关的话，一直都没有一种很优雅的处理方式去更新 state，而是需要在 componentWilReceiveProps 中判断前后两个 props 是否相同，如果不同再将新的

props 更新到相应的 state 上去。这样做一来会破坏 state 数据的单一数据源，导致组件状态变得不可预测，另一方面也会增加组件的重绘次数。类似的业务需求也有很多，如一个可以横向滑动的列表，当前高亮的 Tab 显然隶属于列表自身的时，根据传入的某个值，直接定位到某个 Tab。为了解决这些问题，React 引入了第一个新的生命周期：getDerivedStateFromProps。它有以下的优点：

getDSFP 是静态方法，在这里不能使用 this，也就是一个纯函数，开发者不能写出副作用的代码

开发者只能通过 prevState 而不是 prevProps 来做对比，保证了 state 和 props 之间的简单关系以及不需要处理第一次渲染时 prevProps 为空的情况

基于第一点，将状态变化（setState）和昂贵操作（tabChange）区分开，更加便于 render 和 commit 阶段操作或者说优化。

3) componentWillUpdate

与 componentWillReceiveProps 类似，许多开发者也会在 componentWillUpdate 中根据 props 的变化去触发一些回调 。 但不论是 componentWilReceiveProps 还 是 componentWilUpdate，都有可能在一次更新中被调用多次，也就是说写在这里的回调函数也有可能会被调用多次，这显然是不可取的。与 componentDidMount 类 似，componentDidUpdate 也不存在这样的问题，一次更新中 componentDidUpdate 只会被调用一次，所以将原先写在 componentWillUpdate 中 的 回 调 迁 移 至 componentDidUpdate 就可以解决这个问题。

另外一种情况则是需要获取 DOM 元素状态，但是由于在 fber 中，render 可打断，可能在 wilMount 中获取到的元素状态很可能与实际需要的不同，这个通常可以使用第二个新增的生命函数的解决 getSnapshotBeforeUpdate(prevProps, prevState)

4) getSnapshotBeforeUpdate(prevProps, prevState)

返回的值作为 componentDidUpdate 的第三个参数。与 willMount 不同的是，getSnapshotBeforeUpdate 会在最终确定的 render 执行之前执行，也就是能保证其获取到的元素状态与 didUpdate 中获取到的元素状态相同。

4. React 父子组件各生命周期执行顺序?

首次渲染

Root constructor

Root getDerivedStateFromProps

Root render

father constructor

father getDerivedStateFromProps

father render

children constructor

children getDerivedStateFromProps

children render

children componentDidMount

father componentDidMount

Root componentDidMount

父组件数据修改触发重渲染

Root constructor

Root getDerivedStateFromProps

Root render

father getDerivedStateFromProps

father shouldComponentUpdate

father render

children getDerivedStateFromProps

children shouldComponentUpdate

children render

children getSnapshotBeforeUpdate

father getSnapshotBeforeUpdate

Root getSnapshotBeforeUpdate

children componentDidUpdate, snapshot: 1

father componentDidUpdate, snapshot: 1

Root componentDidUpdate, snapshot: 1

父组件调用 forceUpdate

father getDerivedStateFromProps

father render

children getDerivedStateFromProps

children shouldComponentUpdate

children render

children getSnapshotBeforeUpdate

father getSnapshotBeforeUpdate

children componentDidUpdate, snapshot: 1

father componentDidUpdate, snapshot: 1

销毁、卸载阶段：

React 父组件卸载后。其内部嵌套最深一层组件先调用 componentWillUnmount 钩子函数，然后依次往外调用各组件的 componentWillUnmount 构造函数。

5. 错误处理阶段?

componentDidCatch(error, info)，此生命周期在后代组件抛出错误后被调用。 它接收两个参数：

error：抛出的错误。

info：带有 componentStack key 的对象，其中包含有关组件引发错误的栈信息。

6. React 16.X 中 props 改变后在哪个生命周期中处理?

在 getDerivedStateFromProps 中进行处理。

这个生命周期函数是为了替代 componentWillReceiveProps 存在的，所以在需要使用 componentWillReceiveProps 时，就可以考虑使用 getDerivedStateFromProps 来进行替代。

两者的参数是不相同的，而 getDerivedStateFromProps 是一个静态函数，也就是这个函数不能通过 this 访问到 class 的属性，也并不推荐直接访问属性。而是应该通过参数提供的 nextProps 以及 prevState 来进行判断，根据新传入的 props 来映射到 state。

需要注意的是，如果 props 传入的内容不需要影响到你的 state，那么就需要返回一个 null，这个返回值是必须的，所以尽量将其写到函数的末尾：

```
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的 type 发生变化的时候，更新 state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于 state 不进行任何操作
    return null;
}
```

7. React 性能优化在哪个生命周期？它优化的原理是什么？

react 的父级组件的 render 函数重新渲染会引起子组件的 render 方法的重新渲染。但是，有的时候子组件的接受父组件的数据没有变动。子组件 render 的执行会影响性能，这时就可以使用 shouldComponentUpdate 来解决这个问题。

使用方法如下：

```
shouldComponentUpdate(nexrProps) {
    if (this.props.num === nexrProps.num) {
        return false
    }
    return true;
}
```

shouldComponentUpdate 提供了两个参数 nextProps 和 nextState，表示下一次 props 和一次 state 的值，当函数返回 false 时候，render()方法不执行，组件也就不会渲染，返回 true 时，组件照常重渲染。此方法就是拿当前 props 中值和下一次 props 中的值进行对比，数据相等时，返回 false，反之返回 true。

需要注意，在进行新旧对比的时候，是**浅对比，**也就是说如果比较的数据时引用数据类型，只要数据的引用的地址没变，即使内容变了，也会被判定为 true。

面对这个问题，可以使用如下方法进行解决：

（1）使用 setState 改变数据之前，先采用 ES6 中 assgin 进行拷贝，但是 assgin 只深拷贝的数据的第一层，所以说不是最完美的解决办法：

```
const o2 = Object.assign({},this.state.obj)
    o2.student.count = '00000';
    this.setState({
        obj: o2,
    })
```

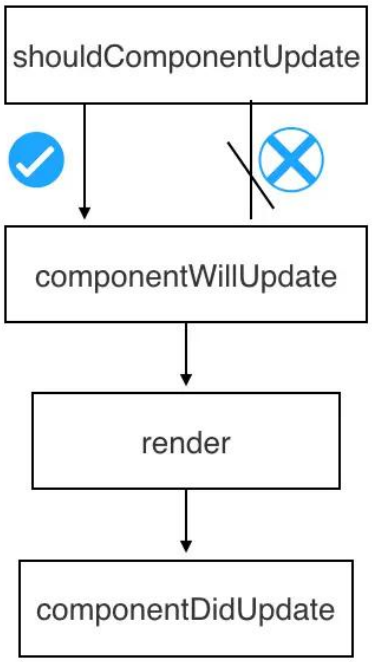
（2）使用 JSON.parse(JSON.stringfy())进行深拷贝，但是遇到数据为 undefined 和函数时就会错。

```
const o2 = JSON.parse(JSON.stringify(this.state.obj))
```

```
o2.student.count = '00000';
this.setState({
    obj: o2,
})
```

8. state 和 props 触发更新的生命周期分别有什么区别?

state 更新流程：



这个过程当中涉及的函数：

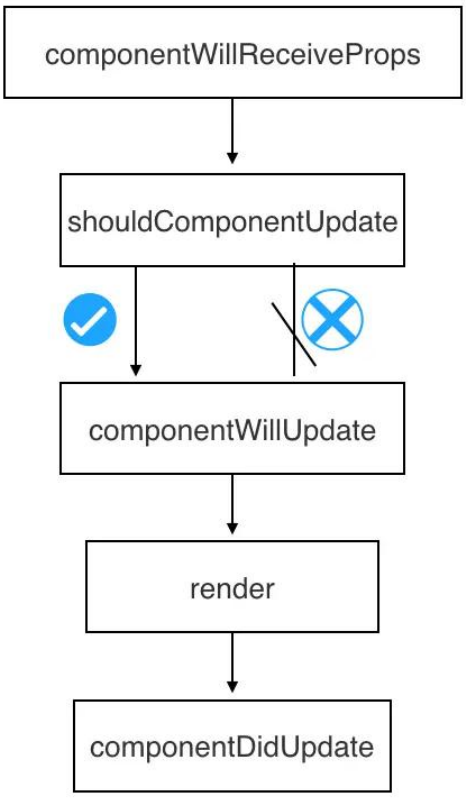
shouldComponentUpdate: 当组件的 state 或 props 发生改变时，都会首先触发这个生命周期函数。它会接收两个参数：nextProps, nextState——它们分别代表传入的新 props 和新的 state 值。拿到这两个值之后，我们就可以通过一些对比逻辑来决定是否有 re-render（重渲染）的必要了。如果该函数的返回值为 false，则生命周期终止，反之继续；

注意：此方法仅作为性能优化的方式而存在。不要企图依靠此方法来“阻止”渲染，因为这可能会产生 bug。应该考虑使用内置的 PureComponent 组件，而不是手动编写 shouldComponentUpdate()

componentWillUpdate：当组件的 state 或 props 发生改变时，会在渲染之前调用 componentWillUpdate。componentWillUpdate 是 React16 废弃的三个生命周期之一。过去，我们可能希望能在这个阶段去收集一些必要的信息（比如更新前的 DOM 信息等等），现在我们完全可以在 React16 的 getSnapshotBeforeUpdate 中去做这些事；

componentDidUpdate：componentDidUpdate() 会在 UI 更新后会被立即调用。它接收 prevProps（上一次的 props 值）作为入参，也就是说在此处我们仍然可以进行 props 值对比（再次说明 componentWillUpdate 确实鸡肋哈）。

props 更新流程：



相对于 state 更新，props 更新后唯一的区别是增加了对 componentWillReceiveProps 的调用。关于 componentWillReceiveProps，需要知道这些事情：

componentWillReceiveProps：它在 Component 接受到新的 props 时被触发。componentWillReceiveProps 会接收一个名为 nextProps 的参数（对应新的 props 值）。该生命周期是 React16 废弃掉的三个生命周期之一。在它被废弃前，可以用它来比较 this.props 和 nextProps 来重新 setState。在 React16 中，用一个类似的新生命周期 getDerivedStateFromProps 来代替它。

9. React 中发起网络请求应该在哪个生命周期中进行？为什么？

对于异步请求，最好放在 componentDidMount 中去操作，对于同步的状态改变，可以放在 componentWillMount 中，一般用的比较少。

如果认为在 componentWillMount 里发起请求能提早获得结果，这种想法其实是错误的，通常 componentWillMount 比 componentDidMount 早不了多少微秒，网络上任何一点延迟，这一点差异都可忽略不计。

react 的生命周期： constructor() -> componentWillMount() -> render() -> componentDidMount()

上面这些方法的调用是有次序的，由上而下依次调用。

constructor 被调用是在组件准备要挂载的最开始，此时组件尚未挂载到网页上。

componentWillMount 方法的调用在 constructor 之后，在 render 之前，在这方法里的代码调用 setState 方法不会触发重新 render，所以它一般不会用来作加载数据之用。

componentDidMount 方法中的代码，是在组件已经完全挂载到网页上才会调用被执行，所以可以保证数据的加载。此外，在这方法中调用 setState 方法，会触发重新渲染。所以，官方设计这个方法就是用来加载外部数据用的，或处理其他的副作用代码。与组件上的数据无关的加载，也可以在 constructor 里做，但 constructor 是做组

件 state 初始化工作，并不是做加载数据这工作的，constructor 里也不能 setState，还有加载的时间太长或者出错，页面就无法加载出来。所以有副作用的代码都会集中在 componentDidMount 方法里。

总结：

跟服务器端渲染（同构）有关系，如果在 componentWillMount 里面获取数据，fetch data 会执行两次，一次在服务器端一次在客户端。在 componentDidMount 中可以解决这个问题，componentWillMount 同样也会 render 两次。

在 componentWillMount 中 fetch data，数据一定在 render 后才能到达，如果忘记了设置初始状态，用户体验不好。

react16.0 以后，componentWillMount 可能会被执行多次。

10. React<16 组件生命周期方法?

组件在进入和离开 DOM 时要经历一系列生命周期方法，下面是这些生命周期方法。

componentWillMount()

在渲染前调用,在客户端也在服务端，它只发生一次。

componentDidMount()

在第一次渲染后调用，只在客户端。之后组件已经生成了对应的 DOM 结构，可以通过 this.getDOMNode()来进行访问。如果你想和其他 JavaScript 框架一起使用，可以在这个方法中调用 setTimeout, setInterval 或者发送 AJAX 请求等操作(防止异部操作阻塞 UI)。

componentWillReceiveProps()

在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化 render 时不会被调用。

shouldComponentUpdate()

返回一个布尔值。在组件接收到新的 props 或者 state 时被调用。在初始化时或者使用 forceUpdate 时不被调用。 可以在你确认不需要更新组件时使用。

componentWillUpdate()

在组件接收到新的 props 或者 state 但还没有 render 时被调用。在初始化时不会被调用。

componentDidUpdate()

在组件完成更新后立即调用。在初始化时不会被调用。

componentWillUnMount()

组件从 DOM 中移除的时候立刻被调用。

getDerivedStateFromError()

这个生命周期方法在 ErrorBoundary 类中使用。实际上，如果使用这个生命周期方法，任何类都会变成 ErrorBoundary。这用于在组件树中出现错误时呈现回退 UI，而不是在屏幕上显示一些奇怪的错误。

componentDidCatch()

这个生命周期方法在 ErrorBoundary 类中使用。实际上，如果使用这个生命周期方法，任何类都会变成 ErrorBoundary。这用于在组件树中出现错误时记录错误。

五、组件通信

1. React 组件间通信常见的几种情况?

父组件向子组件通信

子组件向父组件通信

跨级组件通信

非嵌套关系的组件通信。

2. 组件通信的方式有哪些?

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。

3. 父子组件的通信方式?

父组件向子组件通信：父组件通过 props 向子组件传递需要的信息。

```
// 子组件: Child
const Child = props =>{
   return <p>{props.name}</p>
}
// 父组件 Parent
const Parent = ()=>{
      return <Child name="react"></Child>
}
```

子组件向父组件通信：: props+回调的方式。

```
// 子组件: Child
const Child = props =>{
   const cb = msg =>{
         return ()=>{
               props.callback(msg)
         }
   }
   return (
         <button onClick={cb("你好!")}>你好</button>
   )
}
// 父组件 Parent
class Parent extends Component {
      callback(msg){
```

```
        console.log(msg)
    }
    render(){
        return <Child callback={this.callback.bind(this)}></Child>
    }
}
```

4. 跨级组件的通信方式?

父组件向子组件的子组件通信，向更深层子组件通信：

使用 props，利用中间组件层层传递,但是如果父组件结构较深，那么中间每一层组件都要去传递 props，增加了复杂度，并且这些 props 并不是中间组件自己需要的。

使用 context，context 相当于一个大容器，可以把要通信的内容放在这个容器中，这样不管嵌套多深，都可以随意取用，对于跨越多层的全局数据可以使用 context 实现。

```
// context 方式实现跨级组件通信
// 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
const BatteryContext = createContext();
//   子组件的子组件
class GrandChild extends Component {
    render(){
        return (
            <BatteryContext.Consumer>
                {
                    color => <h1 style={{"color":color}}>我是红色的:{color}</h1>
                }
            </BatteryContext.Consumer>
        )
    }
}
//   子组件
const Child = () =>{
    return (
        <GrandChild/>
    )
}
// 父组件
class Parent extends Component {
      state = {
          color:"red"
      }
      render(){
```

```
            const {color} = this.state
            return (
            <BatteryContext.Provider value={color}>
                <Child></Child>
            </BatteryContext.Provider>
            )
        }
    }
```

5. 什么是上下文 Context?

Context 通过组件树提供了一个传递数据的方法，从而避免了在每一个层级手动的传递 props 属性。

用法：在父组件上定义 getChildContext 方法，返回一个对象，然后它的子组件就可以通过 this.context 属性来获取。

6. 非嵌套关系组件的通信方式?

没有任何包含关系的组件，包括兄弟组件以及不在同一个父级中的非兄弟组件。

可以使用自定义事件通信（发布订阅模式）

可以通过 redux 等进行全局状态管理

如果是兄弟组件通信，可以找到这两个兄弟节点共同的父节点, 结合父子间通信方式进行通信。

7. 如何解决 props 层级过深的问题?

使用 Context API：提供一种组件之间的状态共享，而不必通过显式组件树逐层传递 props；

使用 Redux 等状态库。

8. 如何使用 Context API?

首先要引入 React 内置的 React Context API

然后创建 provider

最后创建 consumer。

六、路由 react-router

1. 什么是 React 路由? 什么是 React Router Dom?

React 路由是一个构建在 React 之上的强大的路由库，它有助于向应用程序添加新的屏幕和流。这使 URL 与网页上显示的数据保持同步。它负责维护标准化的结构和行为，并用于开发单页 Web 应用。 React 路由有一个简单的 API。

react-router-dom 是应用程序中路由的库。 React 库中没有路由功能，需要单独安装 react-router-dom。 提供两个路由器 BrowserRouter 和 HashRoauter。前者基于 url 的 pathname 段，后者基于 hash 段。

前者：http://127.0.0.1:3000/article/num1

后者：http://127.0.0.1:3000/#/article/num1（不一定是这样，但#是少不了的）

react-router-dom 组件

BrowserRouter 和 HashRouter 是路由器。

Route 用于路由匹配。

Link 组件用于在应用程序中创建链接。 它将在 HTML 中渲染为锚标记。

NavLink 是突出显示当前活动链接的特殊链接。

Switch 不是必需的，但在组合路由时很有用。

Redirect 用于强制路由重定向。

2. react 路由与 vue 路由对比?

个人认为最大的区别在于匹配模式

React 包容性路由：

如果路由有/food 和 /food/1 那么在匹配 /food 的时候两个都能匹配到

react 就是典型的包容性路由

所以 react 需要引入 Switch 标签，把路由变成排他性的

Vue 排他性路由：

只要匹配成功一个就不会往下面进行匹配了

vue 是排他性路由

匹配从上到下，匹配到一个即止。

3. 为什么 React Router v4 中使用 switch 关键字?

虽然 ** 用于封装 Router 中的多个路由，当你想要仅显示要在多个定义的路线中呈现的单个路线时，可以使用 “switch” 关键字。使用时，<switch>** 标记会按顺序将已定义的 URL 与已定义的路由进行匹配。找到第一个匹配项后，它将渲染指定的路径。从而绕过其它路线。

Switch 通常被用来包裹 Route，用于渲染与路径匹配的第一个子 <Route> 或 <Redirect>，它里面不能放其他元素。

假如不加 <Switch> ：

```
import { Route } from 'react-router-dom'
<Route path="/" component={Home}></Route>
<Route path="/login" component={Login}></Route>
```

Route 组件的 path 属性用于匹配路径，因为需要匹配 / 到 Home，匹配 /login 到 Login，所以需要两个 Route，但是不能这么写。这样写的话，当 URL 的 path 为 “/login” 时，<Route path="/" />和<Route path="/login" /> 都会被匹配，因此页面会展示 Home 和 Login 两个组件。这时就需要借助 <Switch> 来做到只显示一个匹配组件：

```
import { Switch, Route} from 'react-router-dom'
<Switch>
    <Route path="/" component={Home}></Route>
    <Route path="/login" component={Login}></Route>
</Switch>
```

此时，再访问 “/login” 路径时，却只显示了 Home 组件。这是就用到了 exact 属性，它的作用就是精确匹配路径，经常与<Switch> 联合使用。只有当 URL 和该 <Route> 的 path 属性完全一致的情况下才能匹配上：

```
import { Switch, Route} from 'react-router-dom'
<Switch>
   <Route exact path="/" component={Home}></Route>
   <Route exact path="/login" component={Login}></Route>
</Switch>
```

4. 列出 React Router 的优点?

就像 React 基于组件一样，在 React Router v4 中，API 是 'All About Components'。可以将 Router 可视化为单个根组件（<BrowserRouter>），其中我们将特定的子路由（<route>）包起来。

无需手动设置历史值：在 React Router v4 中，我们要做的就是将路由包装在 <BrowserRouter> 组件中。

包是分开的：共有三个包，分别用于 Web、Native 和 Core。这使我们应用更加紧凑。基于类似的编码风格很容易进行切换。

5. React Router 与常规路由有何不同?

	常规路由	React 路由
参与的页面	每个视图对应一个新文件	只涉及单个 HTML 页面
URL 更改	HTTP 请求被发送到服务器并且接收相应的 HTML 页面	仅更改历史记录属性
体验	用户实际在每个视图的不同页面切换	用户认为自己正在不同的页面间切换

6. 如何配置 React-Router 实现路由切换?

（1）使用<Route> 组件

路由匹配是通过比较 <Route> 的 path 属性和当前地址的 pathname 来实现的。当一个 <Route> 匹配成功时，它将渲染其内容，当它不匹配时就会渲染 null。没有路径的 <Route> 将始终被匹配。

```
// when location = { pathname: '/about' }
<Route path='/about' component={About}/> // renders <About/>
<Route path='/contact' component={Contact}/> // renders null
<Route component={Always}/> // renders <Always/>
```

（2）结合使用 <Switch> 组件和 <Route> 组件

<Switch> 用于将 <Route> 分组。

```
<Switch>
    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
</Switch>
```

<Switch> 不是分组 <Route> 所必须的，但他通常很有用。 一个 <Switch> 会遍历其所有的子 <Route>元素，并仅渲染与当前地址匹配的第一个元素。

（3）使用 <Link>、 <NavLink>、<Redirect> 组件

<Link> 组件来在你的应用程序中创建链接。无论你在何处渲染一个<Link> ，都会在应用程序的 HTML 中渲染锚（<a>）。

```
<Link to="/">Home</Link>
// <a href='/'>Home</a>
```

是一种特殊类型的 当它的 to 属性与当前地址匹配时，可以将其定义为"活跃的"。

```
// location = { pathname: '/react' }
<NavLink to="/react" activeClassName="hurray">
    React
</NavLink>
// <a href='/react' className='hurray'>React</a>
```

当我们想强制导航时，可以渲染一个<Redirect>，当一个<Redirect>渲染时，它将使用它的 to 属性进行定向。

7. React-Router 怎么设置重定向?

使用<Redirect>组件实现路由的重定向：

```
<Switch>
  <Redirect from='/users/:id' to='/users/profile/:id'/>
  <Route path='/users/profile/:id' component={Profile}/>
</Switch>
```

当请求 /users/:id 被重定向去 '/users/profile/:id'：

属性 from: string：需要匹配的将要被重定向路径。

属性 to: string：重定向的 URL 字符串

属性 to: object：重定向的 location 对象

属性 push: bool：若为真，重定向操作将会把新地址加入到访问历史记录里面，并且无法回退到前面的页面。

8. react-router 里的 Link 标签和 a 标签的区别?

从最终渲染的 DOM 来看，这两者都是链接，都是 标签，区别是：

<Link>是 react-router 里实现路由跳转的链接，一般配合<Route> 使用，react-router 接管了其默认的链接跳转行为，区别于传统的页面跳转，<Link> 的“跳转”行为只会触发相匹配的<Route>对应的页面内容更新，而不会刷新整个页面。

<Link>做了 3 件事情:

有 onclick 那就执行 onclick

click 的时候阻止 a 标签默认事件

根据跳转 href(即是 to)，用 history (web 前端路由两种方式之一，history & hash)跳转，此时只是链接变了，并没有刷新页面而<a>标签就是普通的超链接了，用于从当前页面跳转到 href 指向的另一 个页面(非锚点情况)。

9. a 标签默认事件禁掉之后做了什么才实现了跳转?

```
let domArr = document.getElementsByTagName('a')
[...domArr].forEach(item=>{
    item.addEventListener('click',function () {
        location.href = this.href
    })
})
```

10. React-Router 如何获取 URL 的参数和历史对象?

（1）获取 URL 的参数

get 传值

路由配置还是普通的配置，如：'admin'，传参方式如：'admin?id='1111''。通过 this.props.location.search 获取 url 获取到一个字符串'?id='1111'

可以用 url，qs，querystring，浏览器提供的 api URLSearchParams 对象或者自己封装的方法去解析出 id 的值。

动态路由传值

路由需要配置成动态路由：如 path='/admin/:id'，传参方式，如'admin/111'。通过 this.props.match.params.id 取得 url 中的动态路由 id 部分的值，除此之外还可以通过 useParams（Hooks）来获取

通过 query 或 state 传值

传参方式如：在 Link 组件的 to 属性中可以传递对象{pathname:'/admin',query:'111',state:'111'};。通过 this.props.location.state 或 this.props.location.query 来获取即可，传递的参数可以是对象、数组等，但是存在缺点就是只要刷新页面，参数就会丢失。

（2）获取历史对象

如果 React >= 16.8 时可以使用 React Router 中提供的 Hooks

```
import { useHistory } from "react-router-dom";
let history = useHistory();
```

2.使用 this.props.history 获取历史对象

```
let history = this.props.history;
```

11. React-Router 4 怎样在路由变化时重新渲染同一个组件?

当路由变化时，即组件的 props 发生了变化，会调用 componentWillReceiveProps 等生命周期钩子。那需要做的只是： 当路由改变时，根据路由，也去请求数据：

```
class NewsList extends Component {
  componentDidMount () {
      this.fetchData(this.props.location);
  }

  fetchData(location) {
    const type = location.pathname.replace('/', '') || 'top'
    this.props.dispatch(fetchListData(type))
  }
  componentWillReceiveProps(nextProps) {
      if (nextProps.location.pathname != this.props.location.pathname) {
          this.fetchData(nextProps.location);
      }
  }
  render () {
    ...
  }
}
```

利用生命周期 componentWillReceiveProps，进行重新 render 的预处理操作。

12. React-Router 的路由有几种模式?

React-Router 支持使用 hash（对应 HashRouter）和 browser（对应 BrowserRouter） 两种路由规则，react-router-dom 提供了 BrowserRouter 和 HashRouter 两个组件来实现应用的 UI 和 URL 同步：

BrowserRouter 创建的 URL 格式：xxx.com/path

HashRouter 创建的 URL 格式：xxx.com/#/path

（1）BrowserRouter

它使用 HTML5 提供的 history API（pushState、replaceState 和 popstate 事件）来保持 UI 和 URL 的同步。由此可以看出，BrowserRouter 是使用 HTML 5 的 history API 来控制路由跳转的：

```
<BrowserRouter
    basename={string}
    forceRefresh={bool}
    getUserConfirmation={func}
    keyLength={number}
/>
```

其中的属性如下：

basename 所有路由的基准 URL。basename 的正确格式是前面有一个前导斜杠，但不能有尾部斜杠；

```
<BrowserRouter basename="/calendar">
    <Link to="/today" />
</BrowserRouter>
```

等同于

```
<a href="/calendar/today" />
```

forceRefresh 如果为 true，在导航的过程中整个页面将会刷新。一般情况下，只有在不支持 HTML5 history API 的浏览器中使用此功能；

getUserConfirmation 用于确认导航的函数，默认使用 window.confirm。例如，当从 /a 导航至 /b 时，会使用默认的 confirm 函数弹出一个提示，用户点击确定后才进行导航，否则不做任何处理；

```
// 这是默认的确认函数
const getConfirmation = (message, callback) => {
  const allowTransition = window.confirm(message);
  callback(allowTransition);
}
<BrowserRouter getUserConfirmation={getConfirmation} />
```

需要配合<Prompt> 一起使用。

KeyLength 用来设置 Location.Key 的长度。

（2）HashRouter

使用 URL 的 hash 部分（即 window.location.hash）来保持 UI 和 URL 的同步。由此可以看出，HashRouter 是通过 URL 的 hash 属性来控制路由跳转的：

```
<HashRouter
    basename={string}
    getUserConfirmation={func}
    hashType={string}
/>
```

其参数如下：

basename, getUserConfirmation 和 BrowserRouter 功能一样；

hashType window.location.hash 使用的 hash 类型，有如下几种：

slash - 后面跟一个斜杠，例如 #/ 和 #/sunshine/lollipops；

noslash - 后面没有斜杠，例如 # 和 #sunshine/lollipops；

hashbang - Google 风格的 ajax crawlable，例如 #!/ 和 #!/sunshine/lollipops。

13. React 路由鉴权?

https://juejin.cn/post/6844903924441284615#heading-22

七、Redux && react-redux

1. 什么是 Redux? 与 react-redux 有什么区别?

Redux 是当今最热门的前端开发库之一。它是 JavaScript 程序的可预测状态容器，用于整个应用的状态管理。使用 Redux 开发的应用易于测试，可以在不同环境中运行，并显示一致的行为。

React 是视图层框架。Redux 是一个用来管理数据状态和 UI 状态的 JavaScript 应用工具。随着 JavaScript 单页应用（SPA）开发日趋复杂， JavaScript 需要管理比任何时候都要多的 state（状态）， Redux 就是降低管理难度的。（Redux 支持 React、Angular、jQuery 甚至纯 JavaScript）。

在 React 中，UI 以组件的形式来搭建，组件之间可以嵌套组合。但 React 中组件间通信的数据流是单向的，顶层组件可以通过 props 属性向下层组件传递数据，而下层组件不能向上层组件传递数据，兄弟组件之间同样不能。这样简单的单向数据流支撑起了 React 中的数据可控性。

当项目越来越大的时候，管理数据的事件或回调函数将越来越多，也将越来越不好管理。管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等。state 的管理在大项目中相当复杂。

Redux 提供了一个叫 store 的统一仓储库，组件通过 dispatch 将 state 直接传入 store，不用通过其他的组件。并且组件通过 subscribe 从 store 获取到 state 的改变。使用了 Redux，所有的组件都可以从 store 中获取到所需的 state，他们也能从 store 获取到 state 的改变。这比组件之间互相传递数据清晰明朗的多。

主要解决的问题：

单纯的 Redux 只是一个状态机，是没有 UI 呈现的，react- redux 作用是将 Redux 的状态机和 React 的 UI 呈现绑定在一起，当你 dispatch action 改变 state 的时候，会自动更新页面。

2. Redux 遵循的三个原则是什么?

单一事实来源：整个应用的状态存储在单个 store 中的对象/状态树里。单一状态树可以更容易地跟踪随时间的变化，并调试或检查应用程序。

状态是只读的：改变状态的唯一方法是去触发一个动作。动作是描述变化的普通 JS 对象。就像 state 是数据的最小表示一样，该操作是对数据更改的最小表示。

使用纯函数进行更改：为了指定状态树如何通过操作进行转换，你需要纯函数。纯函数是那些返回值仅取决于其参数值的函数。

3. 你对“单一事实来源”有什么理解?

Redux 使用 “Store” 将程序的整个状态存储在同一个地方。因此所有组件的状态都存储在 Store 中，并且它们从 Store 本身接收更新。单一状态树可以更容易地跟踪随时间的变化，并调试或检查程序。

4. 列出 Redux 的组件?

Action – 这是一个用来描述发生了什么事情的对象。

Reducer – 这是一个确定状态将如何变化的地方。

Store – 整个程序的状态/对象树保存在 Store 中。

View – 只显示 Store 提供的数据。

5. redux 的工作流程?

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。

6. Redux 有哪些优点?

结果的可预测性 - 由于总是存在一个真实来源，即 store ，因此不存在如何将当前状态与动作和应用的其他部分同步的问题。

可维护性 - 代码变得更容易维护，具有可预测的结果和严格的结构。

服务器端渲染 - 你只需将服务器上创建的 store 传到客户端即可。这对初始渲染非常有用，并且可以优化应用性能，从而提供更好的用户体验。

开发人员工具 - 从操作到状态更改，开发人员可以实时跟踪应用中发生的所有事情。

社区和生态系统 - Redux 背后有一个巨大的社区，这使得它更加迷人。一个由才华横溢的人组成的大型社区为库的改进做出了贡献，并开发了各种应用。

易于测试 - Redux 的代码主要是小巧、纯粹和独立的功能。这使代码可测试且独立。

组织 - Redux 准确地说明了代码的组织方式，这使得代码在团队使用时更加一致和简单。

7. 什么是 Flux?

Flux 是一种强制单向数据流的架构模式。它控制派生数据，并使用具有所有数据权限的中心 store 实现多个组件之间的通信。整个应用中的数据更新必须只能在此处进行。 Flux 为应用提供稳定性并减少运行时的错误。

8. Redux 与 Flux 有何不同?

Flux	Redux
Store 包含状态和更改逻辑	Store 和更改逻辑是分开的
有多个 Store	只有一个 Store

所有 Store 都互不影响且是平级的	带有分层 reducer 的单一 Store
有单一调度器	没有调度器的概念
React 组件订阅 store	容器组件是有联系的
状态是可变的	状态是不可改变的

9. Redux 原理及工作流程?

（1）原理

Redux 源码主要分为以下几个模块文件

compose.js 提供从右到左进行函数式编程

createStore.js 提供作为生成唯一 store 的函数

combineReducers.js 提供合并多个 reducer 的函数，保证 store 的唯一性

bindActionCreators.js 可以让开发者在不直接接触 dispacth 的前提下进行更改 state 的操作

applyMiddleware.js 这个方法通过中间件来增强 dispatch 的功能

（2）工作流程

const store= createStore（fn）生成数据;

action: {type: Symble('action01), payload:'payload' }定义行为;

dispatch 发起 action：store.dispatch(doSomething('action001'));

reducer：处理 action，返回新的 state;

通俗点解释：

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

以 store 为核心，可以把它看成数据存储中心，但是他要更改数据的时候不能直接修改，数据修改更新的角色由 Reducers 来担任，store 只做存储，中间人，当 Reducers 的更新完成以后会通过 store 的订阅来通知 react component，组件把新的状态重新获取渲染，组件中也能主动发送 action，创建 action 后这个动作是不会执行的，所以要 dispatch 这个 action，让 store 通过 reducers 去做更新 React Component 就是 react 的每个组件。

10. 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。

11. 如何在 Redux 中定义 Action?

React 中的 Action 必须具有 type 属性，该属性指示正在执行的 ACTION 的类型。必须将它们定义为字符串常量，并且还可以向其添加更多的属性。在 Redux 中，action 被名为 Action Creators 的函数所创建。以下是 Action 和 Action Creator 的示例：

```
function addTodo(text) {
```

```
        return {
                type: ADD_TODO,
                 text
        }
    }
```

12. 解释 Reducer 的作用?

Reducers 是纯函数，它规定应用程序的状态怎样因响应 ACTION 而改变。Reducers 通过接受先前的状态和 action 来工作，然后它返回一个新的状态。它根据操作的类型确定需要执行哪种更新，然后返回新的值。如果不需要完成任务，它会返回原来的状态。

13. Store 在 Redux 中的意义是什么?

Store 是一个 JavaScript 对象，它可以保存程序的状态，并提供一些方法来访问状态、调度操作和注册侦听器。应用程序的整个状态/对象树保存在单一存储中。因此，Redux 非常简单且是可预测的。我们可以将中间件传递到 store 来处理数据，并记录改变存储状态的各种操作。所有操作都通过 reducer 返回一个新状态。

14. redux 中如何进行异步操作?异步的请求怎么处理?

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，当下主流的异步中间件有两种 redux-thunk、redux-saga。

（1）使用 react-thunk 中间件

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

使用步骤：

配置中间件，在 store 的创建中配置

```
import {createStore, applyMiddleware, compose} from 'redux';
import reducer from './reducer';
import thunk from 'redux-thunk'
// 设置调试工具
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__({}) : compose;
// 设置中间件
const enhancer = composeEnhancers(
  applyMiddleware(thunk)
);
const store = createStore(reducer, enhancer);
export default store;
```

添加一个返回函数的 actionCreator，将异步请求逻辑放在里面

```
/**
   发送 get 请求，并生成相应 action，更新 store 的函数
   @param url {string} 请求地址
   @param func {function} 真正需要生成的 action 对应的 actionCreator
   @return {function}
*/
// dispatch 为自动接收的 store.dispatch 函数
export const getHttpAction = (url, func) => (dispatch) => {
      axios.get(url).then(function(res){
            const action = func(res.data)
            dispatch(action)
      })
}
```

生成 action，并发送 action

```
componentDidMount(){
      var action = getHttpAction('/getData', getInitTodoItemAction)
      // 发送函数类型的 action 时，该 action 的函数体会自动执行
      store.dispatch(action)
}
```

（2）使用 redux-saga 中间件

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-saga 可以捕获 action，然后执行一个函数，那么可以把异步代码放在这个函数中，使用步骤如下：

配置中间件

```
import {createStore, applyMiddleware, compose} from 'redux';
import reducer from './reducer';
import createSagaMiddleware from 'redux-saga'
import TodoListSaga from './sagas'
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__({}) : compose;
const sagaMiddleware = createSagaMiddleware()
const enhancer = composeEnhancers(
  applyMiddleware(sagaMiddleware)
);
const store = createStore(reducer, enhancer);
sagaMiddleware.run(TodoListSaga)

export default store;
```

将异步请求放在 sagas.js 中

```
import {takeEvery, put} from 'redux-saga/effects'
import {initTodoList} from './actionCreator'
import {GET_INIT_ITEM} from './actionTypes'
import axios from 'axios'
function* func(){
    try{
        // 可以获取异步返回数据
        const res = yield axios.get('/getData')
        const action = initTodoList(res.data)
        // 将 action 发送到 reducer
        yield put(action)
    }catch(e){
        console.log('网络请求失败')
    }
}
function* mySaga(){
```

```
    // 自动捕获 GET_INIT_ITEM 类型的 action，并执行 func
    yield takeEvery(GET_INIT_ITEM, func)
}
export default mySaga
```

发送 action

```
componentDidMount(){
  const action = getInitTodoItemAction()
  store.dispatch(action)
}
```

15. 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。

16. Redux 中间件是什么？接受几个参数？柯里化函数两端的参数具体是什么？

Redux 的中间件提供的是位于 action 被发起之后，到达 reducer 之前的扩展点，换而言之，原本 view -→> action -> reducer -> store 的数据流加上中间件后变成了 view -> action -> middleware -> reducer -> store ，在这一环节可以做一些"副作用"的操作，如异步请求、打印日志等。

从 applyMiddleware 源码中可以看出：

redux 中间件接受一个对象作为参数，对象的参数上有两个字段 dispatch 和 getState，分别代表着 Redux Store 上的两个同名函数。

柯里化函数两端一个是 middewares，一个是 store.dispatch。

17. Redux 请求中间件如何处理并发?

使用 redux-Saga

redux-saga 是一个管理 redux 应用异步操作的中间件，用于代替 redux-thunk 的。它通过创建 Sagas 将所有异步操作逻辑存放在一个地方进行集中处理，以此将 react 中的同步操作与异步操作区分开来，以便于后期的管理与维护。 redux-saga 如何处理并发：

takeEvery

可以让多个 saga 任务并行被 fork 执行。

```
import {
    fork,
    take
} from "redux-saga/effects"


const takeEvery = (pattern, saga, ...args) => fork(function*() {
    while (true) {
        const action = yield take(pattern)
        yield fork(saga, ...args.concat(action))
    }
})
```

takeLatest

takeLatest 不允许多个 saga 任务并行地执行。一旦接收到新的发起的 action，它就会取消前面所有 fork 过的任务（如果这些任务还在执行的话）。

在处理 AJAX 请求的时候，如果只希望获取最后那个请求的响应， takeLatest 就会非常有用。

```
import {
```

```
    cancel,
    fork,
    take
} from "redux-saga/effects"
const takeLatest = (pattern, saga, ...args) => fork(function*() {
    let lastTask
    while (true) {
        const action = yield take(pattern)
        if (lastTask) {
            yield cancel(lastTask) // 如果任务已经结束，则 cancel 为空操作
        }
        lastTask = yield fork(saga, ...args.concat(action))
    }
})
```

18. Redux 怎么实现属性传递，介绍下原理?

react-redux 数据传输： view-->action-->reducer-->store-->view。看下点击事件的数据是如何通过 redux 传到 view 上：

view 上的 AddClick 事件通过 mapDispatchToProps 把数据传到 action ---> click:()=>dispatch(ADD)

action 的 ADD 传到 reducer 上

reducer 传到 store 上 const store = createStore(reducer);

store 再通过 mapStateToProps 映射穿到 view 上 text:State.text

19. Redux 中间件是怎么拿到 store 和 action? 然后怎么处理?

redux 中间件本质就是一个函数柯里化。redux applyMiddleware Api 源码中每个 middleware 接受 2 个参数，Store 的 getState 函数和 dispatch 函数，分别获得 store 和 action，最终返回一个函数。该函数会被传入 next 的下一个 middleware 的 dispatch 方法，并返回一个接收 action 的新函数，这个函数可以直接调用 next（action），或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store 的 dispatch 方法作为 next 参数，并借此结束调用链。所以，middleware 的函数签名是（{ getState，dispatch })=> next => action。

20. Redux 状态管理器和变量挂载到 window 中有什么区别?

两者都是存储数据以供后期使用。但是 Redux 状态更改可回溯——Time travel，数据多了的时候可以很清晰的知道改动在哪里发生，完整的提供了一套状态管理模式。

随着 JavaScript 单页应用开发日趋复杂，JavaScript 需要管理比任何时候都要多的 state（状态）。这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI 状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。

如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据

等等。前端开发者正在经受前所未有的复杂性，难道就这么放弃了吗?当然不是。

这里的复杂性很大程度上来自于：我们总是将两个难以理清的概念混淆在一起：变化和异步。 可以称它们为曼妥思和可乐。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 React 视图在视图层禁止异步和直接操作 DOM 来解决这个问题。美中不足的是，React 依旧把处理 state 中数据的问题留给了你。Redux 就是为了帮你解决这个问题。

21. mobox 和 redux 有什么区别?

（1）共同点

为了解决状态管理混乱，无法有效同步的问题统一维护管理应用状态;

某一状态只有一个可信数据来源（通常命名为 store，指状态容器）;

操作更新状态方式统一，并且可控（通常以 action 方式提供更新状态的途径）;

支持将 store 与 React 组件连接，如 react-redux，mobx- react;

（2）区别

Redux 更多的是遵循 Flux 模式的一种实现，是一个 JavaScript 库，它关注点主要是以下几方面：

Action： 一个 JavaScript 对象，描述动作相关信息，主要包含 type 属性和 payload 属性：

o type： action 类型; o payload： 负载数据;

Reducer： 定义应用状态如何响应不同动作（action），如何更新状态;

Store： 管理 action 和 reducer 及其关系的对象，主要提供以下功能：

o 维护应用状态并支持访问状态(getState());

o 支持监听 action 的分发，更新状态(dispatch(action));

o 支持订阅 store 的变更(subscribe(listener));

异步流： 由于 Redux 所有对 store 状态的变更，都应该通过 action 触发，异步任务（通常都是业务或获取数据任务）也不例外，而为了不将业务或数据相关的任务混入 React 组件中，就需要使用其他框架配合管理异步任务流程，如 redux-thunk，redux-saga 等;

Mobx 是一个透明函数响应式编程的状态管理库，它使得状态管理简单可伸缩：

Action：定义改变状态的动作函数，包括如何变更状态;

Store： 集中管理模块状态（State）和动作(action)

Derivation（衍生）： 从应用状态中派生而出，且没有任何其他影响的数据

对比总结：

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。

22. Redux 和 Vuex 有什么区别，它们的共同思想?

（1）Redux 和 Vuex 区别

Vuex 改进了 Redux 中的 Action 和 Reducer 函数，以 mutations 变化函数取代 Reducer，无需 switch，只需在对应的 mutation 函数里改变 state 值即可

Vuex 由于 Vue 自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的 State 即可

Vuex 数据流的顺序是：View 调用 store.commit 提交对应的请求到 Store 中对应的 mutation 函数->store 改变（vue 检测到数据变化自动渲染）

通俗点理解就是，vuex 弱化 dispatch，通过 commit 进行 store 状态的一次更变；取消了 action 概念，不必传入特定的 action 形式进行指定变更；弱化 reducer，基于 commit 参数直接对数据进行转变，使得框架更加简易；

（2）共同思想

单一的数据源

变化可以预测

本质上： redux 与 vuex 都是对 mvvm 思想的服务，将数据从视图中抽离的一种方案。。

23. Redux 中的 connect 有什么作用?

connect 负责连接 React 和 Redux

（1）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

（2）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

（3）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。

八、Hooks

1. 对 React Hook 的理解，它的实现原理是什么?

React-Hooks 是 React 团队在 React 组件开发实践中，逐渐认知到的一个改进点，这背后其实涉及对类组件和函数组件两种组件形式的思考和侧重。

（1）类组件： 所谓类组件，就是基于 ES6 Class 这种写法，通过继承 React.Component 得来的 React 组件。以下是一个类组件：

```
class DemoClass extends React.Component {
  state = {
    text: ""
  };
  componentDidMount() {
    //...
  }
  changeText = (newText) => {
```

```
      this.setState({
        text: newText
      });
    };

    render() {
      return (
        <div className="demoClass">
          <p>{this.state.text}</p>
          <button onClick={this.changeText}>修改</button>
        </div>
      );
    }
  }
```

可以看出，React 类组件内部预置了相当多的“现成的东西”等着我们去调度/定制，state 和生命周期就是这些“现成东西”中的典型。要想得到这些东西，难度也不大，只需要继承一个 React.Component 即可。

当然，这也是类组件的一个不便，它太繁杂了，对于解决许多问题来说，编写一个类组件实在是一个过于复杂的姿势。复杂的姿势必然带来高昂的理解成本，这也是我们所不想看到的。除此之外，由于开发者编写的逻辑在封装后是和组件粘在一起的，这就使得类组件内部的逻辑难以实现拆分和复用。

（2）函数组件：函数组件就是以函数的形态存在的 React 组件。早期并没有 React-Hooks，函数组件内部无法定义和维护 state，因此它还有一个别名叫“无状态组件”。以下是一个函数组件：

```
function DemoFunction(props) {
  const { text } = props
  return (
    <div className="demoFunction">
      <p>{`函数组件接收的内容：[${text}]`}</p>
    </div>
  );
}
```

相比于类组件，函数组件肉眼可见的特质自然包括轻量、灵活、易于组织和维护、较低的学习成本等。

通过对比，从形态上可以对两种组件做区分，它们之间的区别如下：

类组件需要继承 class，函数组件不需要；

类组件可以访问生命周期方法，函数组件不能；

类组件中可以获取到实例化后的 this，并基于这个 this 做各种各样的事情，而函数组件不可以；

类组件中可以定义并维护 state（状态），而函数组件不可以；

除此之外，还有一些其他的不同。通过上面的区别，我们不能说谁好谁坏，它们各有自己的优势。在 React-Hooks 出现之前，类组件的能力边界明显强于函数组件。

实际上，类组件和函数组件之间，是面向对象和函数式编程这两套不同的设计思想之间的差异。而函数组件更加契合 React 框架的设计理念：

React 组件本身的定位就是函数，一个输入数据、输出 UI 的函数。作为开发者，我们编写的是声明式的代码，而 React 框架的主要工作，就是及时地把声明式的代码转换为命令式的 DOM 操作，把数据层面的描述映射到用户可见的 UI 变化中去。这就意味着从原则上来讲，React 的数据应该总是紧紧地和渲染绑定在一起的，而类组件做不到这一点。函数组件就真正地将数据和渲染绑定到了一起。函数组件是一个更加匹配其设计理念、也更有利于逻辑拆分与重用的组件表达形式。

为了能让开发者更好的的去编写函数式组件。于是，React-Hooks 便应运而生。

React-Hooks 是一套能够使函数组件更强大、更灵活的“钩子”。

函数组件比起类组件少了很多东西，比如生命周期、对 state 的管理等。这就给函数组件的使用带来了非常多的局限性，导致我们并不能使用函数这种形式，写出一个真正的全功能的组件。而 React-Hooks 的出现，就是为了帮助函数组件补齐这些（相对于类组件来说）缺失的能力。

如果说函数组件是一台轻巧的快艇，那么 React-Hooks 就是一个内容丰富的零部件箱。“重装战舰”所预置的那些设备，这个箱子里基本全都有，同时它还不强制你全都要，而是允许你自由地选择和使用你需要的那些能力，然后将这些能力以 Hook（钩子）的形式“钩”进你的组件里，从而定制出一个最适合你的“专属战舰”。

2. React Hooks 解决了哪些问题?

（1）在组件之间复用状态逻辑很难

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）解决此类问题可以使用 render props 和 高阶组件。但是这类方案需要重新组织组件结构，这可能会很麻烦，并且会使代码难以理解。由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。尽管可以在 DevTools 过滤掉它们，但这说明了一个更深层次的问题：React 需要为共享状态逻辑提供更好的原生途径。

可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。Hook 使我们在无需修改组件结构的情况下复用状态逻辑。 这使得在组件间或社区内共享 Hook 变得更便捷。

（2）复杂组件变得难以理解

在组件中，每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 componentDidMount 和 componentDidUpdate 中获取数据。但是，同一个 componentDidMount 中可能也包含很多其它的逻辑，如设置事件监听，而之后需在 componentWillUnmount 中清除。相互关联且需要对照修改的代码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致逻辑不一致。

在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往会引入了很多抽象概念，需要你在不同的文件之间来回切换，使得复用变得更加困难。

为了解决这个问题，Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据），而并非强制按照生命周期划分。你还可以使用 reducer 来管理组件的内部状态，使其更加可预测。

（3）难以理解的 class

除了代码复用和代码管理会遇到困难外，class 是学习 React 的一大屏障。我们必须去理解 JavaScript 中 this 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。没有稳定的语法提案，这些代码非常冗余。大家可以很好地理解 props，state 和自顶向下的数据流，但对 class 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 class 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

为了解决这些问题，Hook 使你在非 class 的情况下可以使用更多的 React 特性。 从概念上讲，React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也没有牺牲 React 的精神原则。Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术。

3. React Hook 的使用限制有哪些?

React Hooks 的限制主要有两条：

不要在循环、条件或嵌套函数中调用 Hook；

在 React 的函数组件中调用 Hook。

那为什么会有这样的限制呢？Hooks 的设计初衷是为了改进 React 组件的开发模式。在旧有的开发模式下遇到了三个问题。

组件之间难以复用状态逻辑。过去常见的解决方案是高阶组件、render props 及状态管理框架。

复杂的组件变得难以理解。生命周期函数与业务逻辑耦合太深，导致关联部分难以拆分。

人和机器都很容易混淆类。常见的有 this 的问题，但在 React 团队中还有类难以优化的问题，希望在编译优化层面做出一些改进。

这三个问题在一定程度上阻碍了 React 的后续发展，所以为了解决这三个问题，Hooks 基于函数组件开始设计。然而第三个问题决定了 Hooks 只支持函数组件。

那为什么不要在循环、条件或嵌套函数中调用 Hook 呢？因为 Hooks 的设计是基于数组实现。在调用时按顺序加入数组中，如果使用循环、条件或嵌套函数很有可能导致数组取值错位，执行错误的 Hook。当然，实质上 React 的源码里不是数组，是链表。

这些限制会在编码上造成一定程度的心智负担，新手可能会写错，为了避免这样的情况，可以引入 ESLint 的 Hooks 检查插件进行预防。

4. React Hooks 在平时开发中需要注意的问题和原因?

（1）不要在循环，条件或嵌套函数中调用 Hook，必须始终在 React 函数的顶层使用 Hook

这是因为 React 需要利用调用顺序来正确更新相应的状态，以及调用相应的钩子函数。一旦在循环或条件分支语句中调用 Hook，就容易导致调用顺序的不一致性，从而产生难以预料到的后果。

（2）使用 useState 时候，使用 push，pop，splice 等直接更改数组对象的坑

使用 push 直接更改数组无法获取到新值，应该采用析构方式，但是在 class 里面不会有这个问题。代码示例：

```
function Indicatorfilter() {
  let [num,setNums] = useState([0,1,2,3])
  const test = () => {
    // 这里坑是直接采用 push 去更新 num
    // setNums(num)是无法更新 num 的
    // 必须使用 num = [...num ,1]
    num.push(1)
    // num = [...num ,1]
    setNums(num)
  }
return (
    <div className='filter'>
      <div onClick={test}>测试</div>
        <div>
          {num.map((item,index) => (
              <div key={index}>{item}</div>
```

```
                ))}
            </div>
        </div>
    )
}

class Indicatorfilter extends React.Component<any,any>{
    constructor(props:any){
        super(props)
        this.state = {
            nums:[1,2,3]
        }
        this.test = this.test.bind(this)
    }

    test(){
        // class 采用同样的方式是没有问题的
        this.state.nums.push(1)
        this.setState({
            nums: this.state.nums
        })
    }

    render(){
        let {nums} = this.state
        return(
            <div>
                <div onClick={this.test}>测试</div>
                    <div>
                        {nums.map((item:any,index:number) => (
                            <div key={index}>{item}</div>
                        ))}
                    </div>
            </div>

        )
    }
}
```

（3）useState 设置状态的时候，只有第一次生效，后期需要更新状态，必须通过 useEffect

TableDeail 是一个公共组件，在调用它的父组件里面，我们通过 set 改变 columns 的值，以为传递给 TableDeail 的 columns 是最新的值，所以 tabColumn 每次也是最新的值，但是实际 tabColumn 是最开始的值，不会随着 columns 的更新而更新：

```
const TableDeail = ({
    columns,
}:TableData) => {
    const [tabColumn, setTabColumn] = useState(columns)
}

// 正确的做法是通过 useEffect 改变这个值
const TableDeail = ({
    columns,
}:TableData) => {
    const [tabColumn, setTabColumn] = useState(columns)
    useEffect(() =>{setTabColumn(columns)},[columns])
}
```

（4）善用 useCallback

父组件传递给子组件事件句柄时，如果我们没有任何参数变动可能会选用 useMemo。但是每一次父组件渲染子组件即使没变化也会跟着渲染一次。

（5）不要滥用 useContext

可以使用基于 useContext 封装的状态管理工具。

5. React Hooks 和生命周期的关系?

函数组件 的本质是函数，没有 state 的概念的，因此不存在生命周期一说，仅仅是一个 render 函数而已。

但是引入 Hooks 之后就变得不同了，它能让组件在不使用 class 的情况下拥有 state，所以就有了生命周期的概念，所谓的生命周期其实就是 useState、 useEffect() 和 useLayoutEffect() 。

即：Hooks 组件（使用了 Hooks 的函数组件）有生命周期，而函数组件（未使用 Hooks 的函数组件）是没有生命周期的。

下面是具体的 class 与 Hooks 的生命周期对应关系：

class 组件	Hooks 组件
Constructor	useState
getDerivedStateFromProps	useState 里面 update 函数
shouldComponentUpdate	useMemo
Render	函数本身
componentDidMount	useEffect
componentDidUpdate	useEffect
componentWillUnmount	useEffect 里面返回的函数
componentDidCatch	无

getDerivedStateFromError 无

constructor：函数组件不需要构造函数，可以通过调用 **useState 来初始化 state**。如果计算的代价比较昂贵，也可以传一个函数给 useState。

```
const [num, UpdateNum] = useState(0)
```

getDerivedStateFromProps：一般情况下，我们不需要使用它，可以在渲染过程中更新 state，以达到实现 getDerivedStateFromProps 的目的。

```
function ScrollView({row}) {
  let [isScrollingDown, setIsScrollingDown] = useState(false);
  let [prevRow, setPrevRow] = useState(null);
  if (row !== prevRow) {
    // Row 自上次渲染以来发生过改变。更新 isScrollingDown。
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }
  return `Scrolling down: ${isScrollingDown}`;
}
```

React 会立即退出第一次渲染并用更新后的 state 重新运行组件以避免耗费太多性能。

shouldComponentUpdate：可以用 **React.memo** 包裹一个组件来对它的 props 进行浅比较

```
const Button = React.memo((props) => {   // 具体的组件});
```

注意：**React.memo 等效于 **``**PureComponent**，它只浅比较 props。这里也可以使用 useMemo 优化每一个节点。

render：这是函数组件体本身。

componentDidMount, componentDidUpdate： useLayoutEffect 与它们两的调用阶段是一样的。但是，我们推荐你一开始先用 useEffect，只有当它出问题的时候再尝试使用 useLayoutEffect。useEffect 可以表达所有这些的组合。

```
// componentDidMount
useEffect(()=>{
  // 需要在 componentDidMount 执行的内容
}, [])
useEffect(() => {
```

```
  // 在 componentDidMount，以及 count 更改时 componentDidUpdate 执行的内容
  document.title = `You clicked ${count} times`;
  return () => {
    // 需要在 count 更改时 componentDidUpdate（先于 document.title = ... 执行，遵守先清理后更新）
    // 以及 componentWillUnmount 执行的内容
  } // 当函数中 Cleanup 函数会按照在代码中定义的顺序先后执行，与函数本身的特性无关
}, [count]); // 仅在 count 更改时更新
```

请记得 React 会等待浏览器完成画面渲染之后才会延迟调用 ，因此会使得额外操作很方便

componentWillUnmount：相当于 useEffect 里面返回的 cleanup 函数

```
// componentDidMount/componentWillUnmount
useEffect(()=>{
  // 需要在 componentDidMount 执行的内容
  return function cleanup() {
    // 需要在 componentWillUnmount 执行的内容
  }
}, [])
```

componentDidCatch and getDerivedStateFromError：目前还没有这些方法的 Hook 等价写法，但很快会加上。

6. 为什么 useState 要使用数组而不是对象?

useState 的用法：const [count, setCount] = useState(0)

可以看到 useState 返回的是一个数组。

果 useState 返回的是数组，那么使用者可以对数组中的元素命名，代码看起来也比较干净

如果 useState 返回的是对象，在解构对象的时候必须要和 useState 内部实现返回的对象同名，想要使用多次的话，必须得设置别名才能使用返回值

下面来看看如果 useState 返回对象的情况：

```
// 第一次使用
const { state, setState } = useState(false);
// 第二次使用
const { state: counter, setState: setCounter } = useState(0)
```

这里可以看到，返回对象的使用方式还是挺麻烦的，更何况实际项目中会使用的更频繁。

总结：useState 返回的是 array 而不是 object 的原因就是为了降低使用的复杂度，返回数组的话可以直接根据顺序解构，而返回对象的话要想使用多次就需要定义别名了。

7. useEffect 与 useLayoutEffect 的区别?

（1）共同点

运用效果： useEffect 与 useLayoutEffect 两者都是用于处理副作用，这些副作用包括改变 DOM、设置订阅、操作定时器等。在函数组件内部操作副作用是不被允许的，所以需要使用这两个函数去处理。

使用方式： useEffect 与 useLayoutEffect 两者底层的函数签名是完全一致的，都是调用的 mountEffectImpl 方法，在使用上也没什么差异，基本可以直接替换。

（2）不同点

使用场景： useEffect 在 React 的渲染过程中是被异步调用的，用于绝大多数场景；而 useLayoutEffect 会在所有的 DOM 变更之后同步调用，主要用于处理 DOM 操作、调整样式、避免页面闪烁等问题。也正因为是同步处理，所以需要避免在 useLayoutEffect 做计算量较大的耗时任务从而造成阻塞。

使用效果： useEffect 是按照顺序执行代码的，改变屏幕像素之后执行（先渲染，后改变 DOM），当改变屏幕内容时可能会产生闪烁；useLayoutEffect 是改变屏幕像素之前就执行了（会推迟页面显示的事件，先改变 DOM 后渲染），不会产生闪烁。useLayoutEffect 总是比 useEffect 先执行。

在未来的趋势上，两个 API 是会长期共存的，暂时没有删减合并的计划，需要开发者根据场景去自行选择。React 团队的建议非常实用，如果实在分不清，先用 useEffect，一般问题不大；如果页面有异常，再直接替换为 useLayoutEffect 即可。

九、Hoc 高阶组件

https://juejin.cn/post/6844903477798256647#heading-1

https://zhuanlan.zhihu.com/p/24776678

https://juejin.cn/post/6844904085083127821

8. 什么是高阶组件（HOC）?什么是纯组件?有什么区别，适用什么场景?

高阶组件是重用组件逻辑的高级方法，是一种源于 React 的组件模式。 HOC 是自定义组件，在它之内包含另一个组件。它们可以接受子组件提供的任何动态，但不会修改或复制其输入组件中的任何行为。你可以认为 HOC 是“纯（Pure）”组件。

纯（Pure） 组件是可以编写的最简单、最快的组件。它们可以替换任何只有 render() 的组件。这些组件增强了代码的简单性和应用的性能。

1）HOC 的优缺点

优点： 逻辑服用、不影响被包裹组件的内部逻辑。

缺点：hoc 传递给被包裹组件的 props 容易和被包裹后的组件重名，进而被覆盖

2）适用场景

代码复用，逻辑抽象

渲染劫持

State 抽象和更改

Props 更改

3）具体应用例子

权限控制、组件渲染性能追踪、页面复用

权限控制： 利用高阶组件的 条件渲染 特性可以对页面进行权限控制，权限控制一般分为两个维度：页面级

别和 页面元素级别

```
// HOC.js
function withAdminAuth(WrappedComponent) {
    return class extends React.Component {
        state = {
            isAdmin: false,
        }
        async UNSAFE_componentWillMount() {
            const currentRole = await getCurrentUserRole();
            this.setState({
                isAdmin: currentRole === 'Admin',
            });
        }
        render() {
            if (this.state.isAdmin) {
                return <WrappedComponent {...this.props} />;
            } else {
                return (<div>您没有权限查看该页面，请联系管理员！</div>);
            }
        }
    };
}

// pages/page-a.js
class PageA extends React.Component {
    constructor(props) {
        super(props);
        // something here...
    }
    UNSAFE_componentWillMount() {
        // fetching data
    }
    render() {
        // render page with data
    }
}
export default withAdminAuth(PageA);

// pages/page-b.js
```

```
class PageB extends React.Component {
    constructor(props) {
        super(props);
    // something here...
        }
    UNSAFE_componentWillMount() {
    // fetching data
    }
    render() {
    // render page with data
    }
}
export default withAdminAuth(PageB);
```

组件渲染性能追踪： 借助父组件子组件生命周期规则捕获子组件的生命周期，可以方便的对某个组件的渲染时间进行记录：

```
class Home extends React.Component {
        render() {
            return (<h1>Hello World.</h1>);
        }
    }
    function withTiming(WrappedComponent) {
        return class extends WrappedComponent {
            constructor(props) {
                super(props);
                this.start = 0;
                this.end = 0;
            }
            UNSAFE_componentWillMount() {
                super.componentWillMount && super.componentWillMount();
                this.start = Date.now();
            }
            componentDidMount() {
                super.componentDidMount && super.componentDidMount();
                this.end = Date.now();
                console.log(`${WrappedComponent.name} 组件渲染时间为 ${this.end - this.start} ms`);
            }
            render() {
                return super.render();
```

```
        }
      };
    }

    export default withTiming(Home);
```

注意：withTiming 是利用 反向继承 实现的一个高阶组件，功能是计算被包裹组件（这里是 Home 组件）的渲染时间。

页面复用

```
const withFetching = fetching => WrappedComponent => {
    return class extends React.Component {
        state = {
            data: [],
        }
        async UNSAFE_componentWillMount() {
            const data = await fetching();
            this.setState({
                data,
            });
        }
        render() {
            return <WrappedComponent data={this.state.data} {...this.props} />;
        }
    }
}

// pages/page-a.js
export default withFetching(fetching('science-fiction'))(MovieList);
// pages/page-b.js
export default withFetching(fetching('action'))(MovieList);
// pages/page-other.js
export default withFetching(fetching('some-other-type'))(MovieList);
```

9. React 中的高阶组件运用了什么设计模式?

使用了装饰模式，高阶组件的运用：

```
function withWindowWidth(BaseComponent) {
  class DerivedClass extends React.Component {
    state = {
      windowWidth: window.innerWidth,
    }
```

```
        onResize = () => {
            this.setState({
                windowWidth: window.innerWidth,
            })
        }
        componentDidMount() {
            window.addEventListener('resize', this.onResize)
        }
        componentWillUnmount() {
            window.removeEventListener('resize', this.onResize);
        }
        render() {
            return <BaseComponent {...this.props} {...this.state}/>
        }
    }
    return DerivedClass;
}
const MyComponent = (props) => {
    return <div>Window width is: {props.windowWidth}</div>
};
export default withWindowWidth(MyComponent);
```

装饰模式的特点是不需要改变 被装饰对象 本身，而只是在外面套一个外壳接口。JavaScript 目前已经有了原生装饰器的提案，其用法如下：

```
@testable
    class MyTestableClass {
}
```

10. HOC 相比 mixins 有什么优点?

HOC 和 Vue 中的 mixins 作用是一致的，并且在早期 React 也是使用 mixins 的方式。但是在使用 class 的方式创建组件以后，mixins 的方式就不能使用了，并且其实 mixins 也是存在一些问题的，比如：

隐含了一些依赖，比如我在组件中写了某个 state 并且在 mixin 中使用了，就这存在了一个依赖关系。万一下次别人要移除它，就得去 mixin 中查找依赖

多个 mixin 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。

雪球效应，虽然我一个组件还是使用着同一个 mixin，但是一个 mixin 会被多个组件使用，可能会存在需求使得 mixin 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本。

十、其他

1. react16 之后解决了什么问题，增加了哪些东西?

React 16.x 的三大新特性 Time Slicing、Suspense、 hooks

Time Slicing（解决 CPU 速度问题）使得在执行任务的期间可以随时暂停，跑去干别的事情，这个特性使得 react 能在性能极其差的机器跑时，仍然保持有良好的性能

Suspense （解决网络 IO 问题） 和 lazy 配合，实现异步加载组件。 能暂停当前组件的渲染， 当完成某件事以后再继续渲染，解决从 react 出生到现在都存在的「异步副作用」的问题，而且解决得非的优雅，使用的是 T 异步但是同步的写法，这是最好的解决异步问题的方式

提供了一个内置函数 componentDidCatch，当有错误发生时，可以友好地展示 fallback 组件; 可以捕捉到它的子元素（包括嵌套子元素）抛出的异常; 可以复用错误组件。

（1）React16.8

加入 hooks，让 React 函数式组件更加灵活，hooks 之前，React 存在很多问题：

在组件间复用状态逻辑很难

复杂组件变得难以理解，高阶组件和函数组件的嵌套过深。

class 组件的 this 指向问题

难以记忆的生命周期

hooks 很好的解决了上述问题，hooks 提供了很多方法

useState 返回有状态值，以及更新这个状态值的函数

useEffect 接受包含命令式，可能有副作用代码的函数。

useContext 接受上下文对象（从 React.createContext 返回的值）并返回当前上下文值，

useReducer useState 的替代方案。接受类型为 （state，action）=> newState 的 reducer，并返回与 dispatch 方法配对的当前状态。

useCalLback 返回一个回忆的 memoized 版本，该版本仅在其中一个输入发生更改时才会更改。纯函数的输入输出确定性 o useMemo 纯的一个记忆函数 o useRef 返回一个可变的 ref 对象，其 Current 属性被初始化为传递的参数，返回的 ref 对象在组件的整个生命周期内保持不变。

useImperativeMethods 自定义使用 ref 时公开给父组件的实例值

useMutationEffect 更新兄弟组件之前，它在 React 执行其 DOM 改变的同一阶段同步触发

useLayoutEffect DOM 改变后同步触发。使用它来从 DOM 读取布局并同步重新渲染

（2）React16.9

重命名 Unsafe 的生命周期方法。新的 UNSAFE_前缀将有助于在代码 review 和 debug 期间，使这些有问题的字样更突出

废弃 javascrip:形式的 URL。以 javascript:开头的 URL 非常容易遭受攻击，造成安全漏洞。

废弃"Factory"组件。 工厂组件会导致 React 变大且变慢。

act（）也支持异步函数，并且你可以在调用它时使用 await。

使用 <React.ProfiLer> 进行性能评估。在较大的应用中追踪性能回归可能会很方便

（3）React16.13.0

支持在渲染期间调用 setState，但仅适用于同一组件

可检测冲突的样式规则并记录警告

废弃 unstable_createPortal，使用 CreatePortal

将组件堆栈添加到其开发警告中，使开发人员能够隔离 bug 并调试其程序，这可以清楚地说明问题所在，并更快地定位和修复错误。

commit 阶段是对上一阶段获取到的变化部分应用到真实的 DOM 树中，是一系列的 DOM 操作。不仅要维护更复杂的 DOM 状态，而且中断后再继续，会对用户体验造成影响。在普遍的应用场景下，此阶段的耗时比 diff 计算等耗时相对短。

2. react16 版本的 reconciliation 阶段和 commit 阶段是什么?

reconciliation 阶段包含的主要工作是对 current tree 和 new tree 做 diff 计算，找出变化部分。进行遍历、对比等是可以中断，歇一会儿接着再来。

commit 阶段是对上一阶段获取到的变化部分应用到真实的 DOM 树中，是一系列的 DOM 操作。不仅要维护更复杂的 DOM 状态，而且中断后再继续，会对用户体验造成影响。在普遍的应用场景下，此阶段的耗时比 diff 计算等耗时相对短。

3. Fiber 架构是什么?

React16 启用了全新的架构，叫做 Fiber，其最大的使命是解决大型 React 项目的性能问题，再顺手解决之前的一些痛点。

主要有如下几个：

组件不能返回数组，最见的场合是 UL 元素下只能使用 LI，TR 元素下只能使用 TD 或 TH，这时这里有一个组件循环生成 LI 或 TD 列表时，我们并不想再放一个 DIV，这会破坏 HTML 的语义。

弹窗问题，之前一直使用不稳定的 unstable_renderSubtreeIntoContainer。弹窗是依赖原来 DOM 树的上下文，因此这个 API 第一个参数是组件实例，通过它得到对应虚拟 DOM，然后一级级往上找，得到上下文。它的其他参数也很好用，但这个方法一直没有转正。。。

异常处理，我们想知道哪个组件出错，虽然有了 React DevTool，但是太深的组件树查找起来还是很吃力。希望有个方法告诉我出错位置，并且出错时能让我有机会进行一些修复工作

HOC 的流行带来两个问题，毕竟是社区兴起的方案，没有考虑到 ref 与 context 的向下传递。

组件的性能优化全凭人肉，并且主要集中在 SCU，希望框架能干些事情，即使不用 SCU，性能也能上去。

解决进度

16.0 让组件支持返回任何数组类型，从而解决数组问题；推出 createPortal API ,解决弹窗问题；推出 componentDidCatch 新钩子， 划分出错误组件与边界组件， 每个边界组件能修复下方组件错误一次， 再次出错，转交更上层的边界组件来处理，解决异常处理问题。

16.2 推出 Fragment 组件，可以看作是数组的一种语法糖。

16.3 推出 createRef 与 forwardRef 解决 Ref 在 HOC 中的传递问题，推出 new Context API，解决 HOC 的 context 传递问题（主要是 SCU 作祟）

而性能问题，从 16.0 开始一直由一些内部机制来保证，涉及到批量更新及基于时间分片的限量更新。

4. Fiber 架构相对于以前的递归更新组件有有什么优势?

原因是递归更新组件会让 JS 调用栈占用很长时间。

因为浏览器是单线程的，它将 GUI 渲染，事件处理,js 执行等等放在了一起，只有将它做完才能做下一件事，如果有足够的时间，浏览器是会对我们的代码进行编译优化（JIT）及进行热代码优化。

Fiber 架构正是利用这个原理将组件渲染分段执行，提高这样浏览器就有时间优化 JS 代码与修正 reflow！

5. Fiber 是将组件分段渲染，那第一段渲染之后，怎么知道下一段从哪个组件开始渲染呢?

Fiber 节点拥有 return, child, sibling 三个属性，分别对应父节点， 第一个孩子， 它右边的兄弟， 有了它们就足够将一棵树变成一个链表， 实现深度优化遍历。

6. Fiber 怎么决定每次更新的数量?

React16 则是需要将虚拟 DOM 转换为 Fiber 节点，首先它规定一个时间段内，然后在这个时间段能转换多少个 FiberNode，就更新多少个。

因此我们需要将我们的更新逻辑分成两个阶段，第一个阶段是将虚拟 DOM 转换成 Fiber, Fiber 转换成组件实例或真实 DOM（不插入 DOM 树，插入 DOM 树会 reflow）。Fiber 转换成后两者明显会耗时，需要计算还剩下多少时间。

比如，可以记录开始更新视图的时间 var now = new Date - 0，假如我们更新试图自定义需要 100 毫秒，那么定义结束时间是 var deadline = new Date + 100 ,所以每次更新一部分视图，就去拿当前时间 new Date<deadline 做判断，如果没有超过 deadline 就更新视图，超过了，就进入下一个更新阶段。

7. 如何调度时间才能保证流畅?

使用浏览器自带的 api - requestIdleCallback，

它的第一个参数是一个回调，回调有一个参数对象，对象有一个 timeRemaining 方法，就相当于 new Date - deadline，并且它是一个高精度数据， 比毫秒更准确

这个因为浏览器兼容性问题，react 团队自己实现了 requestIdleCallback。

8. fiber 带来的新的生命周期是什么?

创建时

constructor ->

getDerivedStateFromProps(参数 nextProps, prevState,注意里面 this 不指向组件的实例)->

render ->

componentDidMount

更新时

getDerivedStateFromProps(这个时 props 更新才调用，setState 时不调用这个生命周期， 参数 nextProps, prevState) ->

shouldComponentUpdate(setState 时调用参数 nextProps, nextState)->

render->

getSnapsshotBeforeUpdate(替换 componentWillUpdate)

componentDidUpdate(参数 prevProps, prevState, snapshot)

9. 什么是状态提升?

使用 react 经常会遇到几个组件需要共用状态数据的情况。这种情况下，我们最好将这部分共享的状态提升至他们最近的父组件当中进行管理。我们来看一下具体如何操作吧。

```
import React from 'react'
class Child_1 extends React.Component{
```

```
        constructor(props){
            super(props)
        }
        render(){
            return (
                <div>
                    <h1>{this.props.value+2}</h1>
                </div>
            )
        }
    }
    class Child_2 extends React.Component{
        constructor(props){
            super(props)
        }
        render(){
            return (
                <div>
                    <h1>{this.props.value+1}</h1>
                </div>
            )
        }
    }
    class Three extends React.Component {
        constructor(props){
            super(props)
            this.state = {
                txt:"牛逼"
            }
            this.handleChange = this.handleChange.bind(this)
        }
        handleChange(e){
            this.setState({
                txt:e.target.value
            })
        }
        render(){
            return (
                <div>
```

```
                    <input type="text" value={this.state.txt} onChange={this.handleChange}/>
                    <p>{this.state.txt}</p>
                    <Child_1 value={this.state.txt}/>
                    <Child_2 value={this.state.txt}/>
                </div>
            )
        }
    }
    export default Three
```

10. 请说一说 Forwarding Refs 有什么用?

是父组件用来获取子组件的 dom 元素的，为什么有这个 API，原因如下

```
// 例如有一个子组件和父组件，代码如下
子组件为：
class Child extends React.Component{
      constructor(props){
         super(props);
      }
      render(){
         return <input />
      }
 }

// 父组件中，ref：
class Father extends React.Component{
   constructor(props){
      super(props);
      this.myRef=React.createRef();
   }
   componentDidMount(){
      console.log(this.myRef.current);
   }
   render(){
      return <Child ref={this.myRef}/>
   }
}
```

此时父组件的 this.myRef.current 的值是 Child 组件，也就是一个对象，如果用了 React.forwardRef，也就是如下

```
// 子组件
const Child = React.forwardRef((props, ref) => (
```

```
    <input ref={ref} />
));
// 父组件
class Father extends React.Component {
    constructor(props) {
        super(props);
        this.myRef = React.createRef();
    }
    componentDidMount() {
        console.log(this.myRef.current);
    }
    render() {
        return <Child ref={this.myRef} />
    }
}
```

此时父组件的 this.myRef.current 的值是 input 这个 DOM 元素。

11. Fiber 是怎样处理渲染的?

Fiber 会将一个大的更新任务拆解为许多个小任务。每当执行完一个小任务时，渲染线程都会把主线程交回去，看看有没有优先级更高的工作要处理，确保不会出现其他任务被“饿死”的情况，进而避免同步渲染带来的卡顿。在这个过程中，渲染线程不再“一去不回头”，而是可以被打断的，这就是所谓的“异步渲染”。

12. 废除的生命周期跟 Fiber 之间的联系?

在 Fiber 机制下，render 阶段是允许暂停、终止和重启的。当一个任务执行到一半被打断后，下一次渲染线程抢回主动权时，这个任务被重启的形式是“重复执行一遍整个任务”而非“接着上次执行到的那行代码往下走”。这就导致 render 阶段的生命周期都是有可能被重复执行的。

带着这个结论，我们再来看看 React 16 打算废弃的是哪些生命周期：

componentWillMount；

componentWillUpdate；

componentWillReceiveProps。

这些生命周期的共性，就是它们都处于 render 阶段，都可能重复被执行，而且由于这些 API 常年被滥用，它们在重复执行的过程中都存在着不可小觑的风险。。

13. React 历史算法有什么风险?

在 React16 之前，每一次组件的更新都会触发 React 去构建一颗新的虚拟 DOM 树，通过与上一次虚拟 DOM 树的 diff 算法对比，实现对 DOM 的定向更新。这个过程是一个递归的过程，调用栈非常深，只有最底层的返回了，整个渲染过程才回开始逐层返回。而这个漫长且不可被打断的过程，将会给用户的体验带来巨大的风险：同步渲染一旦开始，便会牢牢抓住线程，直到递归完成，这个国恒浏览器不会处理任何渲染之外的事情，会进入一种无法处理用户交互的状态，页面可能会卡死。

14. 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）。

15. 废除的生命周期跟 Fiber 之间的联系?

在 Fiber 机制下，render 阶段是允许暂停、终止和重启的。当一个任务执行到一半被打断后，下一次渲染线程抢回主动权时，这个任务被重启的形式是“重复执行一遍整个任务”而非“接着上次执行到的那行代码往下走”。

16. React 数据持久化有什么实践?

封装数据持久化组件：

```
let storage={
    // 增加
    set(key, value){
        localStorage.setItem(key, JSON.stringify(value));
    },
    // 获取
    get(key){
        return JSON.parse(localStorage.getItem(key));
    },
    // 删除
    remove(key){
        localStorage.removeItem(key);
    }
};
export default Storage;
```

在 React 项目中，通过 redux 存储全局数据时，会有一个问题，如果用户刷新了网页，那么通过 redux 存储的全局数据就会被全部清空，比如登录信息等。这时就会有全局数据持久化存储的需求。首先想到的就是 localStorage，localStorage 是没有时间限制的数据存储，可以通过它来实现数据的持久化存储。

但是在已经使用 redux 来管理和存储全局数据的基础上，再去使用 localStorage 来读写数据，这样不仅是工作量巨大，还容易出错。那么有没有结合 redux 来达到持久数据存储功能的框架呢？当然，它就是 redux-persist。redux-persist 会将 redux 的 store 中的数据缓存到浏览器的 localStorage 中。其使用步骤如下：

（1）首先要安装 redux-persist：

```
npm i redux-persist
```

（2）对于 reducer 和 action 的处理不变，只需修改 store 的生成代码，修改如下：

```
import {createStore} from 'redux'
import reducers from '../reducers/index'
import {persistStore, persistReducer} from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import autoMergeLevel2 from 'redux-persist/lib/stateReconciler/autoMergeLevel2';
```

```
const persistConfig = {
    key: 'root',
    storage: storage,
    stateReconciler: autoMergeLevel2 // 查看 'Merge Process' 部分的具体情况
};
const myPersistReducer = persistReducer(persistConfig, reducers)
const store = createStore(myPersistReducer)
export const persistor = persistStore(store)
export default store
```

（3）在 index.js 中，将 PersistGate 标签作为网页内容的父标签：

```
import React from 'react';
import ReactDOM from 'react-dom';
import {Provider} from 'react-redux'
import store from './redux/store/store'
import {persistor} from './redux/store/store'
import {PersistGate} from 'redux-persist/lib/integration/react';
ReactDOM.render(<Provider store={store}>
                <PersistGate loading={null} persistor={persistor}>
                    {/*网页内容*/}
                </PersistGate>
            </Provider>, document.getElementById('root'));
```

这就完成了通过 redux-persist 实现 React 持久化本地数据存储的简单应用。

17. React 中 props.children 和 React.Children 的区别?

在 React 中，当涉及组件嵌套，在父组件中使用 props.children 把所有子组件显示出来。如下：

```
function ParentComponent(props){
return (
    <div>
        {props.children}
    </div>
)
}
```

如果想把父组件中的属性传给所有的子组件，需要使用 React.Children 方法。

比如，把几个 Radio 组合起来，合成一个 RadioGroup，这就要求所有的 Radio 具有同样的 name 属性值。可以这样：把 Radio 看做子组件，RadioGroup 看做父组件，name 的属性值在 RadioGroup 这个父组件中设置。

首先是子组件：

```
//子组件
```

```
function RadioOption(props) {
  return (
    <label>
      <input type="radio" value={props.value} name={props.name} />
      {props.label}
    </label>
  )
}
```

然后是父组件，不仅需要把它所有的子组件显示出来，还需要为每个子组件赋上 name 属性和值：

```
//父组件用,props 是指父组件的 props
function renderChildren(props) {

  //遍历所有子组件
  return React.Children.map(props.children, child => {
    if (child.type === RadioOption)
      return React.cloneElement(child, {
        //把父组件的 props.name 赋值给每个子组件
        name: props.name
      })
    else
      return child
  })
}
//父组件
function RadioGroup(props) {
  return (
    <div>
      {renderChildren(props)}
    </div>
  )
}
function App() {
  return (
    <RadioGroup name="hello">
      <RadioOption label="选项一" value="1" />
      <RadioOption label="选项二" value="2" />
      <RadioOption label="选项三" value="3" />
    </RadioGroup>
```

```
    )
  }
  export default App;
```

以上，React.Children.map 让我们对父组件的所有子组件又更灵活的控制。

18. React 中 constructor 和 getInitialState 的区别?

两者都是用来初始化 state 的。前者是 ES6 中的语法，后者是 ES5 中的语法，新版本的 React 中已经废弃了该方法。

getInitialState 是 ES5 中的方法，如果使用 createClass 方法创建一个 Component 组件，可以自动调用它的 getInitialState 方法来获取初始化的 State 对象，

```
var APP = React.creatClass ({
  getInitialState() {
    return {
        userName: 'hi',
        userId: 0
     };
  }
})
```

React 在 ES6 的实现中去掉了 getInitialState 这个 hook 函数，规定 state 在 constructor 中实现，如下：

```
Class App extends React.Component{
    constructor(props){
      super(props);
      this.state={};
    }
  }
```

19. 同时引用这三个库 react.js、react-dom.js 和 babel.js 它们都有什么作用?

react：包含 react 所必须的核心代码

react-dom：react 渲染在不同平台所需要的核心代码

babel：将 jsx 转换成 React 代码的工具

20. 在 React 中怎么使用 async/await?

async/await 是 ES7 标准中的新特性。如果是使用 React 官方的脚手架创建的项目，就可以直接使用。如果是在自己搭建的 webpack 配置的项目中使用，可能会遇到 regeneratorRuntime is not defined 的异常错误。那么我们就需要引入 babel，并在 babel 中配置使用 async/await。可以利用 babel 的 transform-async-to-module-method 插件来转换其成为浏览器支持的语法，虽然没有性能的提升，但对于代码编写体验要更好。。

21. React.Children.map 和 js 的 map 有什么区别?

JavaScript 中的 map 不会对为 null 或者 undefined 的数据进行处理，而 React.Children.map 中的 map 可以处理 React.Children 为 null 或者 undefined 的情况。

22. 对 React SSR 的理解?

服务端渲染是数据与模版组成的 html，即 HTML = 数据 ＋ 模版。将组件或页面通过服务器生成 html 字符串，再发送到浏览器，最后将静态标记"混合"为客户端上完全交互的应用程序。页面没使用服务渲染，当请求页面时，返回的 body 里为空，之后执行 js 将 html 结构注入到 body 里，结合 css 显示出来;

SSR 的优势：

对 SEO 友好

所有的模版、图片等资源都存在服务器端

一个 html 返回所有数据

减少 HTTP 请求

响应快、用户体验好、首屏渲染快

1）更利于 SEO

不同爬虫工作原理类似，只会爬取源码，不会执行网站的任何脚本使用了 React 或者其它 MVVM 框架之后，页面大多数 DOM 元素都是在客户端根据 js 动态生成，可供爬虫抓取分析的内容大大减少。另外，浏览器爬虫不会等待我们的数据完成之后再去抓取页面数据。服务端渲染返回给客户端的是已经获取了异步数据并执行 JavaScript 脚本的最终 HTML，网络爬中就可以抓取到完整页面的信息。

2）更利于首屏渲染

首屏的渲染是 node 发送过来的 html 字符串，并不依赖于 js 文件了，这就会使用户更快的看到页面的内容。尤其是针对大型单页应用，打包后文件体积比较大，普通客户端渲染加载所有所需文件时间较长，首页就会有一个很长的白屏等待时间。

SSR 的局限：

1）服务端压力较大

本来是通过客户端完成渲染，现在统一到服务端 node 服务去做。尤其是高并发访问的情况，会大量占用服务端 CPU 资源;

2）开发条件受限

在服务端渲染中，只会执行到 componentDidMount 之前的生命周期钩子，因此项目引用的第三方的库也不可用其它生命周期钩子，这对引用库的选择产生了很大的限制;

3）学习成本相对较高

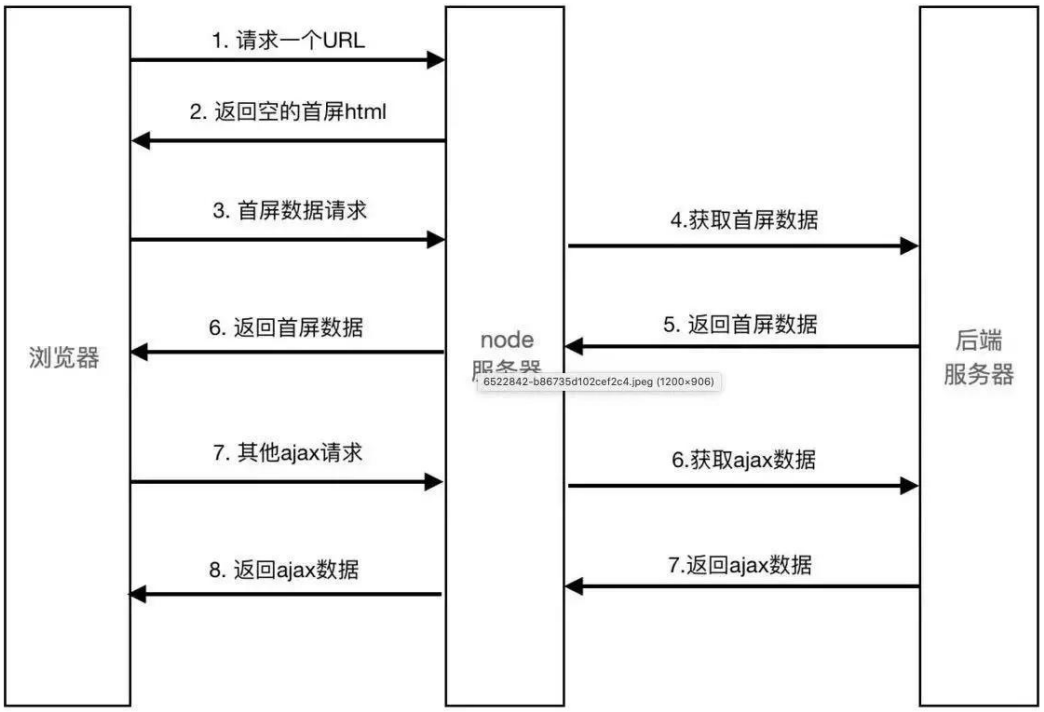
除了对 webpack、MVVM 框架要熟悉，还需要掌握 node、 Koa2 等相关技术。相对于客户端渲染，项目构建、部署过程更加复杂。

时间耗时比较：

1）数据请求

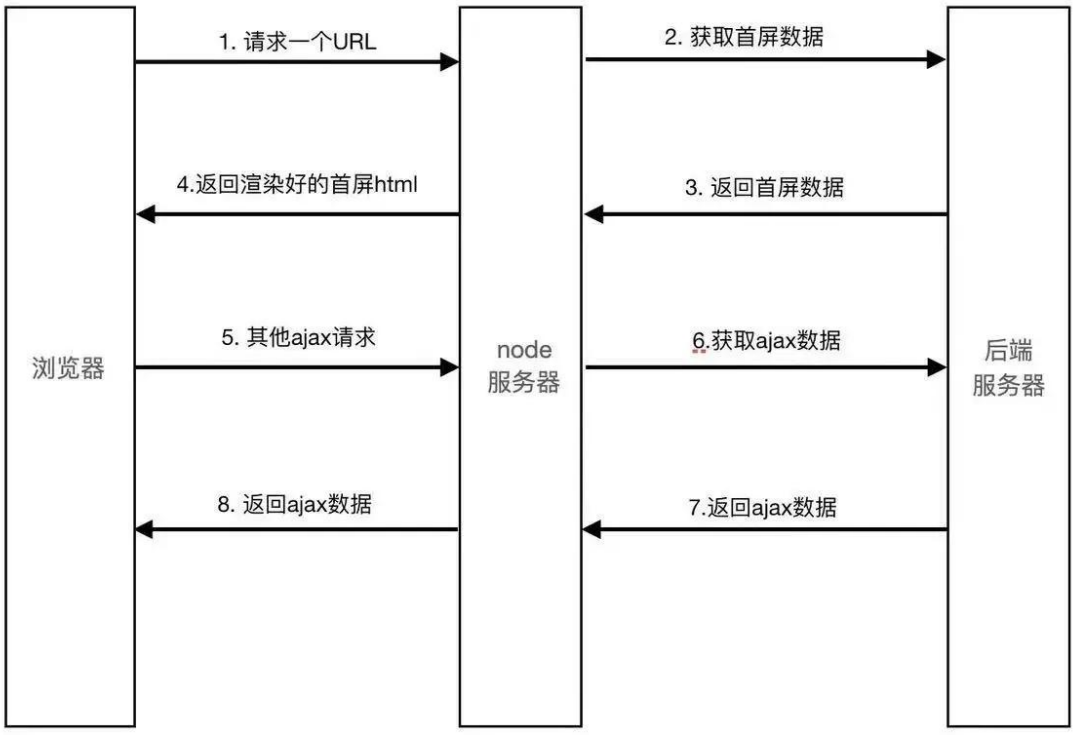
由服务端请求首屏数据，而不是客户端请求首屏数据，这是"快"的一个主要原因。服务端在内网进行请求，数据响应速度快。客户端在不同网络环境进行数据请求，且外网 http 请求开销大，导致时间差

客户端数据请求



客户端渲染路线

服务端数据请求

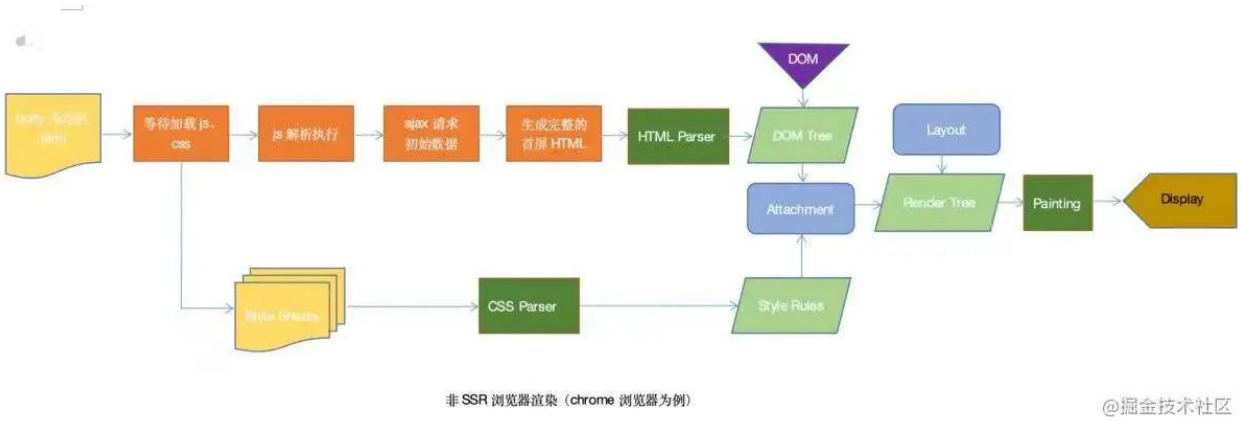


服务端渲染路线

2）html 渲染

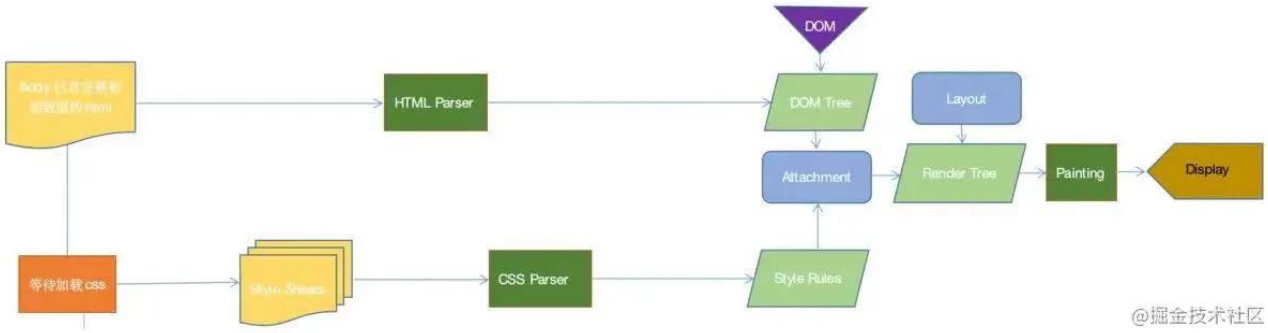
服务端渲染是先向后端服务器请求数据，然后生成完整首屏 html 返回给浏览器；而客户端渲染是等 js 代码下载、加载、解析完成后再请求数据渲染，等待的过程页面是什么都没有的，就是用户看到的白屏。就是服务端渲染不需要等待 js 代码下载完成并请求数据，就可以返回一个已有完整数据的首屏页面。

非 ssr html 渲染



非 SSR 浏览器渲染（chrome 浏览器为例）

ssr html 渲染



23. 在 React 中页面重新加载时怎样保留数据?

这个问题就设计到了数据持久化， 主要的实现方式有以下几种：

Redux： 将页面的数据存储在 redux 中，在重新加载页面时，获取 Redux 中的数据；

data.js： 使用 webpack 构建的项目，可以建一个文件，data.js，将数据保存 data.js 中，跳转页面后获取；

sessionStorge： 在进入选择地址页面之前，componentWillUnMount 的时候，将数据存储到 sessionStorage 中，每次进入页面判断 sessionStorage 中有没有存储的那个值，有，则读取渲染数据；没有，则说明数据是初始化的状态。返回或进入除了选择地址以外的页面，清掉存储的 sessionStorage，保证下次进入是初始化的数据

history API： History API 的 pushState 函数可以给历史记录关联一个任意的可序列化 state，所以可以在路由 push 的时候将当前页面的一些信息存到 state 中，下次返回到这个页面的时候就能从 state 里面取出离开前的数据重新渲染。react-router 直接可以支持。这个方法适合一些需要临时存储的场景。

24. 可以使用 TypeScript 写 React 应用吗？怎么操作?

（1）如果还未创建 Create React App 项目

直接创建一个具有 typescript 的 Create React App 项目：

```
npx create-react-app demo --typescript
```

（2）如果已经创建了 Create React App 项目，需要将 typescript 引入到已有项目中

通过命令将 typescript 引入项目：

npm install --save typescript @types/node @types/react @types/react-dom @types/jest

将项目中任何 后缀名为 ‘.js’ 的 JavaScript 文件重命名为 TypeScript 文件即后缀名为 ‘.tsx’（例如 src/index.js 重命名为 src/index.tsx ）。

25. React key 是干嘛用的 为什么要加？key 主要是解决哪一类问题的?

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。

在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系。

注意事项：

key 值一定要和具体的元素一一对应；

尽量不要用数组的 index 去作为 key；

不要在 render 的时候用随机数或者其他操作给元素加上不稳定的 key，这样造成的性能开销比不加 key 的情况下更糟糕。

第七章 React vs Vue

参考:

https://juejin.cn/post/6844904040564785159

https://juejin.cn/post/6844904052812169229

https://www.cnblogs.com/yangyangxxb/p/10105856.html

https://www.jb51.net/article/195483.htm

https://www.jb51.net/article/202405.htm

第八章 Webpack

一、参考

https://juejin.cn/post/6844904031240863758#heading-4

https://juejin.cn/post/6844903953092591630#heading-3

https://juejin.cn/post/6844904094281236487#heading-15

https://juejin.cn/post/6844904079219490830

https://juejin.cn/post/6844904084927938567

https://juejin.cn/post/6844904093463347208

https://webpack.wuhaolin.cn/

https://gitee.com/jianghuisheng/webpack5?_from=gitee_search

https://juejin.cn/post/6844903842346172430

二、基础

1. webpack 的作用是什么吗?

从官网上的描述我们其实不难理解，webpack 的作用其实有以下几点：

模块打包。可以将不同模块的文件打包整合在一起，并且保证它们之间的引用正确，执行有序。利用打包我们就可以在开发的时候根据我们自己的业务自由划分文件模块，保证项目结构的清晰和可读性。

编译兼容。在前端的“上古时期”，手写一堆浏览器兼容代码一直是令前端工程师头皮发麻的事情，而在今天这个问题被大大的弱化了，通过 webpack 的 Loader 机制，不仅仅可以帮助我们对代码做 polyfill，还可以编译转换诸如.less, .vue, .jsx 这类在浏览器无法识别的格式文件，让我们在开发的时候可以使用新特性和新语法做开发，提高开发效率。

能力扩展。通过 webpack 的 Plugin 机制，我们在实现模块化打包和编译兼容的基础上，可以进一步实现诸如按需加载，代码压缩等一系列功能，帮助我们进一步提高自动化程度，工程效率以及打包输出的质量。

2. webpack 的核心概念?

entry: 入口

output: 输出

loader: 模块转换器，用于把模块原内容按照需求转换成新内容

插件(plugins): 扩展插件，在 webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要做的事情。

Entry

入口起点(entry point)指示 webpack 应该使用哪个模块,来作为构建其内部依赖图的开始。

进入入口起点后,webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

每个依赖项随即被处理,最后输出到称之为 bundles 的文件中。

Output

output 属性告诉 webpack 在哪里输出它所创建的 bundles,以及如何命名这些文件,默认值为 ./dist。

基本上,整个应用程序结构,都会被编译到你指定的输出路径的文件夹中。

Module

模块,在 Webpack 里一切皆模块,一个模块对应着一个文件。Webpack 会从配置的 Entry 开始递归找出所有依赖的模块。

Chunk

代码块,一个 Chunk 由多个模块组合而成,用于代码合并与分割。

Loader

loader 让 webpack 能够去处理那些非 JavaScript 文件（webpack 自身只理解 JavaScript）。

loader 可以将所有类型的文件转换为 webpack 能够处理的有效模块,然后你就可以利用 webpack 的打包能力,对它们进行处理。

本质上,webpack loader 将所有类型的文件,转换为应用程序的依赖图（和最终的 bundle）可以直接引用的模块。

Plugin

loader 被用于转换某些类型的模块,而插件则可以用于执行范围更广的任务。

插件的范围包括,从打包优化和压缩,一直到重新定义环境中的变量。插件接口功能极其强大,可以用来处理各种各样的任务。

3. 说一下模块打包运行原理?

webpack 的整个打包流程：

1、读取 webpack 的配置参数；

2、启动 webpack，创建 Compiler 对象并开始解析项目；

3、从入口文件（entry）开始解析，并且找到其导入的依赖模块，递归遍历分析，形成依赖关系树；

4、对不同文件类型的依赖模块文件使用对应的 Loader 进行编译，最终转为 Javascript 文件；

5、整个过程中 webpack 会通过发布订阅模式，向外抛出一些 hooks，而 webpack 的插件即可通过监听这些关键的事件节点，执行插件任务进而达到干预输出结果的目的。

其中文件的解析与构建是一个比较复杂的过程，在 webpack 源码中主要依赖于 compiler 和 compilation 两个核心对象实现。

compiler 对象是一个全局单例，他负责把控整个 webpack 打包的构建流程。

compilation 对象是每一次构建的上下文对象，它包含了当次构建所需要的所有信息，每次热更新和重新构建，compiler 都会重新生成一个新的 compilation 对象，负责此次更新的构建过程。

而每个模块间的依赖关系，则依赖于 AST 语法树。每个模块文件在通过 Loader 解析完成之后，会通过 acorn 库生成模块代码的 AST 语法树，通过语法树就可以分析这个模块是否还有依赖的模块，进而继续循环执行下一个模块的编译解析。

最终 Webpack 打包出来的 bundle 文件是一个 IIFE 的立即执行函数。

和 webpack4 相比，webpack5 打包出来的 bundle 做了相当的精简。在上面的打包 demo 中，整个立即执行函数里边只有三个变量和一个函数方法，__webpack_modules__存放了编译后的各个文件模块的 JS 内容，__webpack_module_cache__ 用来做模块缓存，__webpack_require__是 Webpack 内部实现的一套依赖引入函数。最后一句则是代码运行的起点，从入口文件开始，启动整个项目。

其中值得一提的是__webpack_require__模块引入函数，我们在模块化开发的时候，通常会使用 ES Module 或者 CommonJS 规范导出/引入依赖模块，webpack 打包编译的时候，会统一替换成自己的__webpack_require__来实现模块的引入和导出，从而实现模块缓存机制，以及抹平不同模块规范之间的一些差异性。

4. Webpack 构建流程简单说一下?

Webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数

开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译

确定入口：根据配置中的 entry 找出所有的入口文件

编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理

完成模块编译：在经过第 4 步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系

输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转

换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会

输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果。

简单说

初始化：启动构建，读取与合并配置参数，加载 Plugin，实例化 Compiler

编译：从 Entry 出发，针对每个 Module 串行调用对应的 Loader 去翻译文件的内容，再找到该 Module 依赖的 Module，递归地进行编译处理

输出：将编译后的 Module 组合成 Chunk，将 Chunk 转换成文件，输出到文件系统中。

5. 有哪些常见的 Loader？你用过哪些 Loader?

raw-loader：加载文件原始内容（utf-8）

file-loader：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件 (处理图片和字体)

url-loader：与 file-loader 类似，区别是用户可以设置一个阈值，大于阈值会交给 file-loader 处理，小于阈值时返回文件 base64 形式编码 (处理图片和字体)

source-map-loader：加载额外的 Source Map 文件，以方便断点调试

svg-inline-loader：将压缩后的 SVG 内容注入代码中

image-loader：加载并且压缩图片文件

json-loader 加载 JSON 文件（默认包含）

handlebars-loader: 将 Handlebars 模版编译成函数并返回

babel-loader：把 ES6 转换成 ES5

ts-loader: 将 TypeScript 转换成 JavaScript

awesome-typescript-loader：将 TypeScript 转换成 JavaScript，性能优于 ts-loader

sass-loader：将 SCSS/SASS 代码转换成 CSS

css-loader：加载 CSS，支持模块化、压缩、文件导入等特性

style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS

postcss-loader：扩展 CSS 语法，使用下一代 CSS，可以配合 autoprefixer 插件自动补齐 CSS3 前缀

eslint-loader：通过 ESLint 检查 JavaScript 代码

tslint-loader：通过 TSLint 检查 TypeScript 代码

mocha-loader：加载 Mocha 测试用例的代码

coverjs-loader：计算测试的覆盖率

vue-loader：加载 Vue.js 单文件组件

i18n-loader: 国际化

cache-loader: 可以在一些性能开销较大的 Loader 之前添加，目的是将结果缓存到磁盘里。

6. 有哪些常见的 Plugin？你用过哪些 Plugin?

define-plugin：定义环境变量 (Webpack4 之后指定 mode 会自动配置)

ignore-plugin：忽略部分文件

html-webpack-plugin：简化 HTML 文件创建 (依赖于 html-loader)

web-webpack-plugin：可方便地为单页应用输出 HTML，比 html-webpack-plugin 好用

uglifyjs-webpack-plugin：不支持 ES6 压缩 (Webpack4 以前)

terser-webpack-plugin: 支持压缩 ES6 (Webpack4)

webpack-parallel-uglify-plugin: 多进程执行代码压缩，提升构建速度

mini-css-extract-plugin: 分离样式文件，CSS 提取为独立文件，支持按需加载 (替代 extract-text-webpack-plugin)

serviceworker-webpack-plugin：为网页应用增加离线缓存功能

clean-webpack-plugin: 目录清理

ModuleConcatenationPlugin: 开启 Scope Hoisting

speed-measure-webpack-plugin: 可以看到每个 Loader 和 Plugin 执行耗时 (整个打包耗时、每个 Plugin 和 Loader 耗时)

webpack-bundle-analyzer: 可视化 Webpack 输出文件的体积 (业务组件、依赖第三方模块)。

7. Loader 和 Plugin 的区别?

Loader 本质就是一个函数，在该函数中对接收到的内容进行转换，返回转换后的结果。

因为 Webpack 只认识 JavaScript，所以 Loader 就成了翻译官，对其他类型的资源进行转译的预处理工作。

Plugin 就是插件，基于事件流框架 Tapable，插件可以扩展 Webpack 的功能，在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

Loader 在 module.rules 中配置，作为模块的解析规则，类型为数组。每一项都是一个 Object，内部包含了 test(类型文件)、loader、options (参数)等属性。

Plugin 在 plugins 中单独配置，类型为数组，每一项是一个 Plugin 的实例，参数都通过构造函数传入。

8. source map 是什么？生产环境怎么用?

source map 是将编译、打包、压缩后的代码映射回源代码的过程。打包压缩后的代码不具备良好的可读性，想要调试源码就需要 soucre map。

map 文件只要不打开开发者工具，浏览器是不会加载的。

线上环境一般有三种处理方案：

hidden-source-map：借助第三方错误监控平台 Sentry 使用

nosources-source-map：只会显示具体行数以及查看源代码的错误栈。安全性比 sourcemap 高

sourcemap：通过 nginx 设置将 .map 文件只对白名单开放(公司内网)

注意：避免在生产中使用 inline- 和 eval-，因为它们会增加 bundle 体积大小，并降低整体性能。

9. 文件监听原理呢?

在发现源码发生变化时，自动重新构建出新的输出文件。

Webpack 开启监听模式，有两种方式：

启动 webpack 命令时，带上 --watch 参数在配置 webpack.config.js 中设置 watch:true

缺点：每次需要手动刷新浏览器

原理：轮询判断文件的最后编辑时间是否变化，如果某个文件发生了变化，并不会立刻告诉监听者，而是先缓存起来，等 aggregateTimeout 后再执行。

```
module.export = {
    // 默认 false,也就是不开启
    watch: true,
    // 只有开启监听模式时，watchOptions 才有意义
```

```
    watchOptions: {
        // 默认为空，不监听的文件或者文件夹，支持正则匹配
        ignored: /node_modules/,
        // 监听到变化发生后会等 300ms 再去执行，默认 300ms
        aggregateTimeout:300,
        // 判断文件是否发生变化是通过不停询问系统指定文件有没有变化实现的，默认每秒问 1000 次
        poll:1000
    }
}。
```

10. 说一下 Webpack 的热更新原理吧?

Webpack 的热更新又称热替换（Hot Module Replacement），缩写为 HMR。 这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

HMR 的核心就是客户端从服务端拉去更新后的文件，准确的说是 chunk diff (chunk 需要更新的部分)，实际上 WDS 与浏览器之间维护了一个 Websocket，当本地资源发生变化时，WDS 会向浏览器推送更新，并带上构建时的 hash，让客户端与上一次资源进行对比。客户端对比出差异后会向 WDS 发起 Ajax 请求来获取更改内容(文件列表、hash)，这样客户端就可以再借助这些信息继续向 WDS 发起 jsonp 请求获取该 chunk 的增量更新。

后续的部分(拿到增量更新之后如何处理？哪些状态该保留？哪些又需要更新？)由 HotModulePlugin 来完成，提供了相关 API 以供开发者针对自身场景进行处理，像 react-hot-loader 和 vue-loader 都是借助这些 API 实现 HMR。

11. 在实际工程中，配置文件上百行乃是常事，如何保证各个 loader 按照预想方式工作?

可以使用 enforce 强制执行 loader 的作用顺序，pre 代表在所有正常 loader 之前执行，post 是所有 loader 之后执行。

12. 如何优化 Webpack 的构建速度?

使用高版本的 Webpack 和 Node.js

多进程/多实例构建：HappyPack(不维护了)、thread-loader

压缩代码

多进程并行压缩

webpack-paralle-uglify-pluginuglifyjs-webpack-plugin 开启 parallel 参数 (不支持 ES6)terser-webpack-plugin 开启 parallel 参数

通过 mini-css-extract-plugin 提取 Chunk 中的 CSS 代码到单独文件，通过 css-loader 的 minimize 选项开启 cssnano 压缩 CSS。

图片压缩

使用基于 Node 库的 imagemin (很多定制选项、可以处理多种图片格式)配置 image-webpack-loader

缩小打包作用域：

exclude/include (确定 loader 规则范围)resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)resolve.mainFields 只采用 main 字段作为入口文件描述字段 (减少搜索步骤，需要考虑到所有运行时依赖的第三方模块的入口文件描述字段)resolve.extensions 尽可能减少后缀尝试的可能性 noParse 对完全不需要解析的库

进行忽略 (不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)IgnorePlugin (完全排除模块)合理使用 alias

提取页面公共资源：

基础包分离：

使用 html-webpack-externals-plugin，将基础包通过 CDN 引入，不打入 bundle 中使用 SplitChunksPlugin 进行(公共脚本、基础包、页面公共文件)分离(Webpack4 内置) ，替代了 CommonsChunkPlugin 插件

DLL：

使用 DllPlugin 进行分包，使用 DllReferencePlugin(索引链接) 对 manifest.json 引用，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间。HashedModuleIdsPlugin 可以解决模块数字 id 问题

充分利用缓存提升二次构建速度：

babel-loader 开启缓存 terser-webpack-plugin 开启缓存使用 cache-loader 或者 hard-source-webpack-plugin

Tree shaking

打包过程中检测工程中没有引用过的模块并进行标记，在资源压缩时将它们从最终的 bundle 中去掉(只能对 ES6 Modlue 生效) 开发中尽可能使用 ES6 Module 的模块，提高 tree shaking 效率禁用 babel-loader 的模块依赖解析，否则 Webpack 接收到的就都是转换过的 CommonJS 形式的模块，无法进行 tree-shaking 使用 PurifyCSS(不在维护) 或者 uncss 去除无用 CSS 代码

purgecss-webpack-plugin 和 mini-css-extract-plugin 配合使用(建议)

Scope hoisting

构建后的代码会存在大量闭包，造成体积增大，运行代码时创建的函数作用域变多，内存开销变大。Scope hoisting 将所有模块的代码按照引用顺序放在一个函数作用域里，然后适当的重命名一些变量以防止变量名冲突必须是 ES6 的语法，因为有很多第三方库仍采用 CommonJS 语法，为了充分发挥 Scope hoisting 的作用，需要配置 mainFields 对第三方模块优先采用 jsnext:main 中指向的 ES6 模块化语法

动态 Polyfill

建议采用 polyfill-service 只给用户返回需要的 polyfill，社区维护。 (部分国内奇葩浏览器 UA 可能无法识别，但可以降级返回所需全部 polyfill)。

13. 文件指纹是什么？怎么用?

文件指纹是打包后输出的文件名的后缀。

Hash：和整个项目的构建相关，只要项目文件有修改，整个项目构建的 hash 值就会更改

Chunkhash：和 Webpack 打包的 chunk 有关，不同的 entry 会生出不同的 chunkhash

Contenthash：根据文件内容来定义 hash，文件内容不变，则 contenthash 不变

JS 的文件指纹设置

设置 output 的 filename，用 chunkhash。

```
module.exports = {
    entry: {
        app: './scr/app.js',
        search: './src/search.js'
    },
    output: {
        filename: '[name][chunkhash:8].js',
        path:__dirname + '/dist'
    }
}
```

CSS 的文件指纹设置

设置 MiniCssExtractPlugin 的 filename，使用 contenthash。

```
module.exports = {
    entry: {
        app: './scr/app.js',
        search: './src/search.js'
    },
    output: {
        filename: '[name][chunkhash:8].js',
        path:__dirname + '/dist'
    },
    plugins:[
        new MiniCssExtractPlugin({
            filename: `[name][contenthash:8].css`
        })
    ]
}
```

图片的文件指纹设置

设置 file-loader 的 name，使用 hash。

占位符名称及含义

ext 资源后缀名

name 文件名称

path 文件的相对路径

folder 文件所在的文件夹

contenthash 文件的内容

hash，默认是 md5 生成 hash

文件内容的 hash，默认是 md5 生成 emoji

一个随机的指代文件内容的 emoj

```
const path = require('path');
module.exports = {
    entry: './src/index.js',
    output: {
        filename:'bundle.js',
        path:path.resolve(__dirname, 'dist')
    },
    module:{
        rules:[{
            test:/\.(png|svg|jpg|gif)$/,
            use:[{
                loader:'file-loader',
                options:{
                    name:'img/[name][hash:8].[ext]'
                }
            }]
        }]
    }}
```

14. 说一下 webpack 热更新?

https://juejin.cn/post/6844904008432222215

15. 控制代码分割?

https://juejin.cn/post/6844904103848443912

16. 关于 webpack 的性能优化?

https://juejin.cn/post/6951297954770583565#heading-6

17. 动态 CDN 资源配置?

在参考别人的配置过程中，发现一个可以在 webpack 中自定义配置 CDN 的方式，主要是利用 html-webpack-plugin 插件的能力，可以新增自定义属性，将 CDN 资源链接，配置至此自定义属性中。通过在 index.html 模板中遍历属性来自动生成 CDN 资源引入。

如下：

webpack.dev.conf.js、webpack.prod.conf.js

```
  // 插件选项
  plugins: [
    // html 模板、以及相关配置
    new HtmlWebpackPlugin({
      title: 'Lesson-06',
      template: resolve('../public/index.html'),
      // cdn（自定义属性）加载的资源，不需要手动添加至 index.html 中,
```

```
        // 顺序按数组索引加载
        cdn: {
          css:['https://cdn.bootcss.com/element-ui/2.8.2/theme-chalk/index.css'],
          js: [
            'https://cdn.bootcss.com/vue/2.6.10/vue.min.js',
            'https://cdn.bootcss.com/element-ui/2.8.2/index.js'
          ]
        }
      })
  ]
```

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title><%= htmlWebpackPlugin.options.title %></title>
    <!-- import cdn css -->
    <% if(htmlWebpackPlugin.options.cdn) {%>
        <% for(var css of htmlWebpackPlugin.options.cdn.css) { %>
            <link rel="stylesheet" href="<%=css%>">
        <% } %>
    <% } %>
</head>
<body>
    <div id="box"></div>
    <!-- import cdn js -->
    <% if(htmlWebpackPlugin.options.cdn) {%>
        <% for(var js of htmlWebpackPlugin.options.cdn.js) { %>
            <script src="<%=js%>"></script>
        <% } %>
    <% } %>
</body>
</html>
```

18. webpack5 和 webpack4 的区别有哪些?

https://juejin.cn/post/6990869970385109005

19. webpack 最佳实践?

https://juejin.cn/post/6982361231071903781

第九章 计算机网络篇

一、Url -> 页面

1. 当在浏览器中输入 Google.com 并且按下回车之后发生了什么?

(1)解析 URL： 首先会对 URL 进行解析，分析所需要使用的传输协议和请求的资源的路径。如果输入的 URL 中的协议或者主机名不合法，将会把地址栏中输入的内容传递给搜索引擎。如果没有问题，浏览器会检查 URL 中是否出现了非法字符，如果存在非法字符，则对非法字符进行转义后再进行下一过程。

(2) 缓存判断： 浏览器会判断所请求的资源是否在缓存里，如果请求的资源在缓存里并且没有失效，那么就直接使用，否则向服务器发起新的请求。

(3) DNS 解析： 下一步首先需要获取的是输入的 URL 中的域名的 IP 地址，首先会判断本地是否有该域名的 IP 地址的缓存，如果有则使用，如果没有则向本地 DNS 服务器发起请求。本地 DNS 服务器也会先检查是否存在缓存，如果没有就会先向根域名服务器发起请求，获得负责的顶级域名服务器的地址后，再向顶级域名服务器请求，然后获得负责的权威域名服务器的地址后，再向权威域名服务器发起请求，最终获得域名的 IP 地址后，本地 DNS 服务器再将这个 IP 地址返回给请求的用户。用户向本地 DNS 服务器发起请求属于递归请求，本地 DNS 服务器向各级域名服务器发起请求属于迭代请求。

(4) 获取 MAC 地址： 当浏览器得到 IP 地址后，数据传输还需要知道目的主机 MAC 地址，因为应用层下发数据给传输层，TCP 协议会指定源端口号和目的端口号，然后下发给网络层。网络层会将本机地址作为源地址，获取的 IP 地址作为目的地址。然后将下发给数据链路层，数据链路层的发送需要加入通信双方的 MAC 地址，本机的 MAC 地址作为源 MAC 地址，目的 MAC 地址需要分情况处理。通过将 IP 地址与本机的子网掩码相与，可以判断是否与请求主机在同一个子网里，如果在同一个子网里，可以使用 APR 协议获取到目的主机的 MAC 地址，如果不在一个子网里，那么请求应该转发给网关，由它代为转发，此时同样可以通过 ARP 协议来获取网关的 MAC 地址，此时目的主机的 MAC 地址应该为网关的地址。

(5) TCP 三次握手： 下面是 TCP 建立连接的三次握手的过程，首先客户端向服务器发送一个 SYN 连接请求报文段和一个随机序号，服务端接收到请求后向服务器端发送一个 SYN ACK 报文段，确认连接请求，并且也向客户端发送一个随机序号。客户端接收服务器的确认应答后，进入连接建立的状态，同时向服务器也发送一个 ACK 确认报文段，服务器端接收到确认后，也进入连接建立状态，此时双方的连接就建立起来了。

(6) HTTPS 握手： 如果使用的是 HTTPS 协议，在通信前还存在 TLS 的一个四次握手的过程。首先由客户端向服务器端发送使用的协议的版本号、一个随机数和可以使用的加密方法。服务器端收到后，确认加密的方法，也向客户端发送一个随机数和自己的数字证书。客户端收到后，首先检查数字证书是否有效，如果有效，则再生成一个随机数，并使用证书中的公钥对随机数加密，然后发送给服务器端，并且还会提供一个前面所有内容的 hash 值供服务器端检验。服务器端接收后，使用自己的私钥对数据解密，同时向客户端发送一个前面所有内容的 hash 值供客户端检验。这个时候双方都有了三个随机数，按照之前所约定的加密方法，使用这三个随机数生成一把秘钥，以后双方通信前，就使用这个秘钥对数据进行加密后再传输。

(7) 返回数据： 当页面请求发送到服务器端后，服务器端会返回一个 html 文件作为响应，浏览器接收到响应后，开始对 html 文件进行解析，开始页面的渲染过程。

(8) 页面渲染： 浏览器首先会根据 html 文件构建 DOM 树，根据解析到的 css 文件构建 CSSOM 树，如果遇到 script 标签，则判端是否含有 defer 或者 async 属性，要不然 script 的加载和执行会造成页面的渲染

的阻塞。当 DOM 树和 CSSOM 树建立好后，根据它们来构建渲染树。渲染树构建好后，会根据渲染树来进行布局。布局完成后，最后使用浏览器的 UI 接口对页面进行绘制。这个时候整个页面就显示出来了。

（9）TCP 四次挥手： 最后一步是 TCP 断开连接的四次挥手过程。若客户端认为数据发送完成，则它需要向服务端发送连接释放请求。服务端收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入 CLOSE_WAIT 状态，此时表明客户端到服务端的连接已经释放，不再接收客户端发的数据了。但是因为 TCP 连接是双向的，所以服务端仍旧可以发送数据给客户端。服务端如果此时还有没发完的数据会继续发送，完毕后会向客户端发送连接释放请求，然后服务端便进入 LAST-ACK 状态。客户端收到释放请求后，向服务端发送确认应答，此时客户端进入 TIME-WAIT 状态。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有服务端的重发请求的话，就进入 CLOSED 状态。当服务端收到确认应答后，也便进入 CLOSED 状态。

2. URL 有哪些组成部分?

以下面的 URL 为例：www.aspxfans.com:8080/news/index.…

从上面的 URL 可以看出，一个完整的 URL 包括以下几部分：

协议部分：该 URL 的协议部分为“http：”，这代表网页使用的是 HTTP 协议。在 Internet 中可以使用多种协议，如 HTTP，FTP 等等本例中使用的是 HTTP 协议。在"HTTP"后面的“//”为分隔符；

域名部分：该 URL 的域名部分为“www.aspxfans.com”。一个 URL 中，也可以使用 IP 地址作为域名使用

端口部分：跟在域名后面的是端口，域名和端口之间使用“:”作为分隔符。端口不是一个 URL 必须的部分，如果省略端口部分，将采用默认端口（HTTP 协议默认端口是 80，HTTPS 协议默认端口是 443）；

虚拟目录部分：从域名后的第一个“/”开始到最后一个“/”为止，是虚拟目录部分。虚拟目录也不是一个 URL 必须的部分。本例中的虚拟目录是“/news/”；

文件名部分：从域名后的最后一个“/”开始到“？”为止，是文件名部分，如果没有“?”,则是从域名后的最后一个“/”开始到“#”为止，是文件部分，如果没有“？”和“#”，那么从域名后的最后一个“/”开始到结束，都是文件名部分。本例中的文件名是“index.asp”。文件名部分也不是一个 URL 必须的部分，如果省略该部分，则使用默认的文件名；

锚部分：从“#”开始到最后，都是锚部分。本例中的锚部分是“name”。锚部分也不是一个 URL 必须的部分；

参数部分：从“？”开始到“#”为止之间的部分为参数部分，又称搜索部分、查询部分。本例中的参数部分为“boardID=5&ID=24618&page=1”。参数可以允许有多个参数，参数与参数之间用“&”作为分隔符。

二、DNS

3. DNS 协议是什么?

概念： DNS 是域名系统 (Domain Name System) 的缩写，提供的是一种主机名到 IP 地址的转换服务，就是我们常说的域名系统。它是一个由分层的 DNS 服务器组成的分布式数据库，是定义了主机如何查询这个分布式数据库的方式的应用层协议。能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的 IP 数串。

作用： 将域名解析为 IP 地址，客户端向 DNS 服务器（DNS 服务器有自己的 IP 地址）发送域名查询请求，DNS 服务器告知客户机 Web 服务器的 IP 地址。

4. DNS 同时使用 TCP 和 UDP 协议?

DNS 占用 53 号端口，同时使用 TCP 和 UDP 协议。

（1）在区域传输的时候使用 TCP 协议

辅域名服务器会定时（一般 3 小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，会执行一次区域传送，进行数据同步。区域传送使用 TCP 而不是 UDP，因为数据同步传送的数据量比一个请求应答的数据量要多得多。

TCP 是一种可靠连接，保证了数据的准确性。

（2）在域名解析的时候使用 UDP 协议

客户端向 DNS 服务器查询域名，一般返回的内容都不超过 512 字节，用 UDP 传输即可。不用经过三次握手，这样 DNS 服务器负载更低，响应更快。理论上说，客户端也可以指定向 DNS 服务器查询时用 TCP，但事实上，很多 DNS 服务器进行配置的时候，仅支持 UDP 查询包。

5. DNS 完整的查询过程?

首先会在浏览器的缓存中查找对应的 IP 地址，如果查找到直接返回，若找不到继续下一步

将请求发送给本地 DNS 服务器，在本地域名服务器缓存中查询，如果查找到，就直接将查找结果返回，若找不到继续下一步

本地 DNS 服务器向根域名服务器发送请求，根域名服务器会返回一个所查询域的顶级域名服务器地址

本地 DNS 服务器向顶级域名服务器发送请求，接受请求的服务器查询自己的缓存，如果有记录，就返回查询结果，如果没有就返回相关的下一级的权威域名服务器的地址

本地 DNS 服务器向权威域名服务器发送请求，域名服务器返回对应的结果

本地 DNS 服务器将返回结果保存在缓存中，便于下次使用

本地 DNS 服务器将返回结果返回给浏览器

比如要查询 www.baidu.com 的 IP 地址，首先会在浏览器的缓存中查找是否有该域名的缓存，如果不存在就将请求发送到本地的 DNS 服务器中，本地 DNS 服务器会判断是否存在该域名的缓存，如果不存在，则向根域名服务器发送一个请求，根域名服务器返回负责 .com 的顶级域名服务器的 IP 地址的列表。然后本地 DNS 服务器再向其中一个负责 .com 的顶级域名服务器发送一个请求，负责 .com 的顶级域名服务器返回负责 .baidu 的权威域名服务器的 IP 地址列表。然后本地 DNS 服务器再向其中一个权威域名服务器发送一个请求，最后权威域名服务器返回一个对应的主机名的 IP 地址列表。

6. URL 有哪些组成部分?

迭代查询与递归查询

实际上，DNS 解析是一个包含迭代查询和递归查询的过程。

递归查询指的是查询请求发出后，域名服务器代为向下一级域名服务器发出请求，最后向用户返回查询的最终结果。使用递归 查询，用户只需要发出一次查询请求。

迭代查询指的是查询请求后，域名服务器返回单次查询的结果。下一级的查询由用户自己请求。使用迭代查询，用户需要发出 多次的查询请求。

一般我们向本地 DNS 服务器发送请求的方式就是递归查询，因为我们只需要发出一次请求，然后本地 DNS 服务器返回给我 们最终的请求结果。而本地 DNS 服务器向其他域名服务器请求的过程是迭代查询的过程，因为每一次域名服务器只返回单次 查询的结果，下一级的查询由本地 DNS 服务器自己进行。

7. DNS 记录和报文?

DNS 服务器中以资源记录的形式存储信息，每一个 DNS 响应报文一般包含多条资源记录。一条资源记录的具体的格式为（Name，Value，Type，TTL）

其中 TTL 是资源记录的生存时间，它定义了资源记录能够被其他的 DNS 服务器缓存多长时间。

常用的一共有四种 Type 的值，分别是 A、NS、CNAME 和 MX ，不同 Type 的值，对应资源记录代表的意义不同：

如果 Type = A，则 Name 是主机名，Value 是主机名对应的 IP 地址。因此一条记录为 A 的资源记录，提供了标 准的主机名到 IP 地址的映射。

如果 Type = NS，则 Name 是个域名，Value 是负责该域名的 DNS 服务器的主机名。这个记录主要用于 DNS 链式 查询时，返回下一级需要查询的 DNS 服务器的信息。

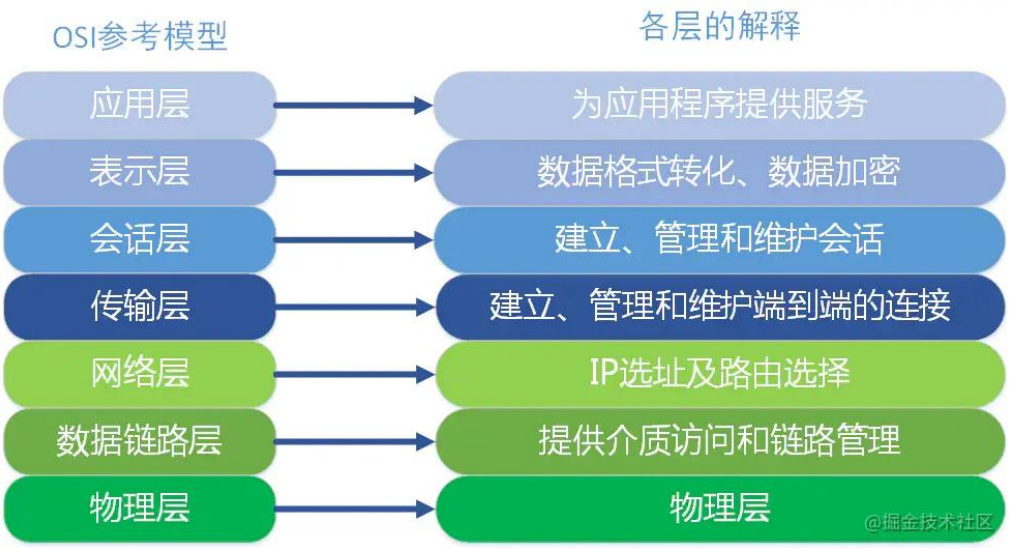
如果 Type = CNAME，则 Name 为别名，Value 为该主机的规范主机名。该条记录用于向查询的主机返回一个主机名 对应的规范主机名，从而告诉查询主机去查询这个主机名的 IP 地址。主机别名主要是为了通过给一些复杂的主机名提供 一个便于记忆的简单的别名。

如果 Type = MX，则 Name 为一个邮件服务器的别名，Value 为邮件服务器的规范主机名。它的作用和 CNAME 是一 样的，都是为了解决规范主机名不利于记忆的缺点。

三、网络模型

8. OSI 七层模型?

ISO 为了更好的使网络应用更为普及，推出了 OSI 参考模型。



（1）应用层

OSI 参考模型中最靠近用户的一层，是为计算机用户提供应用接口，也为用户直接提供各种网络服务。我们常见应用层的网络服务协议有：HTTP，HTTPS，FTP，POP3、SMTP 等。

在客户端与服务器中经常会有数据的请求，这个时候就是会用到 http(hyper text transfer protocol)(超文本传输协议)或者 https.在后端设计数据接口时，我们常常使用到这个协议。

FTP 是文件传输协议，在开发过程中，个人并没有涉及到，但是我想，在一些资源网站，比如百度网盘``迅雷应该是基于此协议的。

SMTP 是 simple mail transfer protocol（简单邮件传输协议）。在一个项目中，在用户邮箱验证码登录的功能时，使用到了这个协议。

（2）表示层

表示层提供各种用于应用层数据的编码和转换功能,确保一个系统的应用层发送的数据能被另一个系统的应

用层识别。如果必要，该层可提供一种标准表示形式，用于将计算机内部的多种数据格式转换成通信中采用的标准表示形式。数据压缩和加密也是表示层可提供的转换功能之一。

在项目开发中，为了方便数据传输，可以使用 base64 对数据进行编解码。如果按功能来划分，base64 应该是工作在表示层。

（3）会话层

会话层就是负责建立、管理和终止表示层实体之间的通信会话。该层的通信由不同设备中的应用程序之间的服务请求和响应组成。

（4）传输层

传输层建立了主机端到端的链接，传输层的作用是为上层协议提供端到端的可靠和透明的数据传输服务，包括处理差错控制和流量控制等问题。该层向高层屏蔽了下层数据通信的细节，使高层用户看到的只是在两个传输实体间的一条主机到主机的、可由用户控制和设定的、可靠的数据通路。我们通常说的，TCP UDP 就是在这一层。端口号既是这里的“端”。

（5）网络层

本层通过 IP 寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点，正确无误地按照地址传送给目的端的运输层。就是通常说的 IP 层。这一层就是我们经常说的 IP 协议层。IP 协议是 Internet 的基础。我们可以这样理解，网络层规定了数据包的传输路线，而传输层则规定了数据包的传输方式。

（6）数据链路层

将比特组合成字节,再将字节组合成帧,使用链路层地址 (以太网使用 MAC 地址)来访问介质,并进行差错检测。

网络层与数据链路层的对比，通过上面的描述，我们或许可以这样理解，网络层是规划了数据包的传输路线，而数据链路层就是传输路线。不过，在数据链路层上还增加了差错控制的功能。

（7）物理层

实际最终信号的传输是通过物理层实现的。通过物理介质传输比特流。规定了电平、速度和电缆针脚。常用设备有（各种物理设备）集线器、中继器、调制解调器、网线、双绞线、同轴电缆。这些都是物理层的传输介质。

OSI 七层模型通信特点：对等通信

对等通信，为了使数据分组从源传送到目的地，源端 OSI 模型的每一层都必须与目的端的对等层进行通信，这种通信方式称为对等层通信。在每一层通信过程中，使用本层自己协议进行通信。

9. TCP/IP 五层协议?

应用层 (application layer)：直接为应用进程提供服务。应用层协议定义的是应用进程间通讯和交互的规则，不同的应用有着不同的应用层协议，如 HTTP 协议（万维网服务）、FTP 协议（文件传输）、SMTP 协议（电子邮件）、DNS（域名查询）等。

传输层 (transport layer)：有时也译为运输层，它负责为两台主机中的进程提供通信服务。该层主要有以下两种协议：

传输控制协议 (Transmission Control Protocol，TCP)：提供面向连接的、可靠的数据传输服务，数据传输的基本单位是报文段（segment）；

用户数据报协议 (User Datagram Protocol，UDP)：提供无连接的、尽最大努力的数据传输服务，但不保证数据传输的可靠性，数据传输的基本单位是用户数据报。

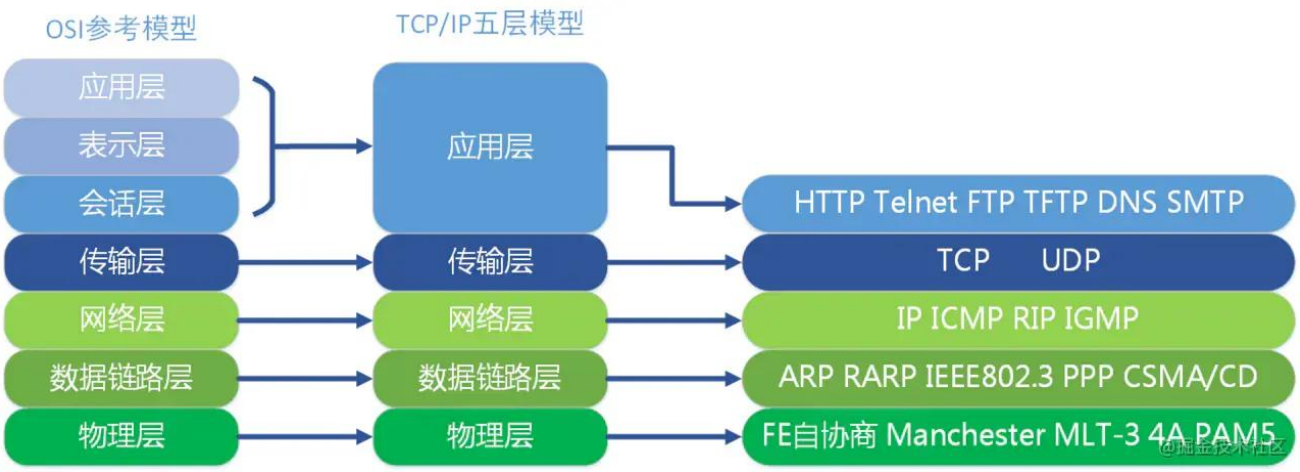
网络层 (internet layer)：有时也译为网际层，它负责为两台主机提供通信服务，并通过选择合适的路由将数据传递到目标主机。

数据链路层 (data link layer)：负责将网络层交下来的 IP 数据报封装成帧，并在链路的两个相邻节点间传送

帧，每一帧都包含数据和必要的控制信息（如同步信息、地址信息、差错控制等）。

物理层 (physical Layer)：确保数据可以在各种物理媒介上进行传输，为数据的传输提供可靠的环境。

TCP/IP 五层协议和 OSI 的七层协议对应关系如下



从上图中可以看出，TCP/IP 模型比 OSI 模型更加简洁，它把应用层/表示层/会话层全部整合为了应用层。

在每一层都工作着不同的设备，比如我们常用的交换机就工作在数据链路层的，一般的路由器是工作在网络层。

四、TCP 与 UDP

1. TCP 和 UDP 的概念及特点?

TCP 和 UDP 都是传输层协议，他们都属于 TCP/IP 协议族：

（1）UDP

UDP 的全称是用户数据报协议，在网络中它与 TCP 协议一样用于处理数据包，是一种无连接的协议。在 OSI 模型中，在传输层，处于 IP 协议的上一层。UDP 有不提供数据包分组、组装和不能对数据包进行排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。

它的特点如下：

1）面向无连接

首先 UDP 是不需要和 TCP 一样在发送数据前进行三次握手建立连接的，想发数据就可以开始发送了。并且也只是数据报文的搬运工，不会对数据报文进行任何拆分和拼接操作。

具体来说就是：

在发送端，应用层将数据传递给传输层的 UDP 协议，UDP 只会给数据增加一个 UDP 头标识下是 UDP 协议，然后就传递给网络层了

在接收端，网络层将数据传递给传输层，UDP 只去除 IP 报文头就传递给应用层，不会任何拼接操作

2）有单播，多播，广播的功能

UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式，也就是说 UDP 提供了单播，多播，广播的功能。

3）面向报文

发送方的 UDP 对应用程序交下来的报文，在添加首部后就向下交付 IP 层。UDP 对应用层交下来的报文，

既不合并，也不拆分，而是保留这些报文的边界。因此，应用程序必须选择合适大小的报文

4）不可靠性

首先不可靠性体现在无连接上，通信都不需要建立连接，想发就发，这样的情况肯定不可靠。

并且收到什么数据就传递什么数据，并且也不会备份数据，发送数据也不会关心对方是否已经正确接收到数据了。

再者网络环境时好时坏，但是 UDP 因为没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP。

5）头部开销小，传输数据报文时是很高效的。

UDP 头部包含了以下几个数据：

两个十六位的端口号，分别为源端口（可选字段）和目标端口

整个数据报文的长度

整个数据报文的检验和（IPv4 可选字段），该字段用于发现头部信息和数据中的错误

因此 UDP 的头部开销小，只有 8 字节，相比 TCP 的至少 20 字节要少得多，在传输数据报文时是很高效的。

（2）TCP

TCP 的全称是传输控制协议是一种面向连接的、可靠的、基于字节流的传输层通信协议。TCP 是面向连接的、可靠的流协议（流就是指不间断的数据结构）。

它有以下几个特点：

1）面向连接

面向连接，是指发送数据之前必须在两端建立连接。建立连接的方法是“三次握手”，这样能建立可靠的连接。建立连接，是为数据的可靠传输打下了基础。

2）仅支持单播传输

每条 TCP 传输连接只能有两个端点，只能进行点对点的数据传输，不支持多播和广播传输方式。

3）面向字节流

TCP 不像 UDP 一样那样一个个报文独立地传输，而是在不保留报文边界的情况下以字节流方式进行传输。

4）可靠传输

对于可靠传输，判断丢包、误码靠的是 TCP 的段编号以及确认号。TCP 为了保证报文传输的可靠，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的字节发回一个相应的确认(ACK)；如果发送端实体在合理的往返时延(RTT)内未收到确认，那么对应的数据（假设丢失了）将会被重传。

5）提供拥塞控制

当网络出现拥塞的时候，TCP 能够减小向网络注入数据的速率和数量，缓解拥塞。

6）提供全双工通信

TCP 允许通信双方的应用程序在任何时候都能发送数据，因为 TCP 连接的两端都设有缓存，用来临时存放双向通信的数据。当然，TCP 可以立即发送一个数据段，也可以缓存一段时间以便一次发送更多的数据段（最大的数据段大小取决于 MSS）。

2. TCP 和 UDP 的区别?

	UDP	TCP
是否连接	无连接	面向连接
是否可靠	不可靠传输，不使用流量控制和拥塞控制	可靠传输（数据顺序和正确性），使用流量控制和拥塞控制
连接对象个数	支持一对一,一对多,多对一,多对多交互通信	只能是一对一通信
传输方式	面向报文	面向字节流
首部开销	首部开销小，仅 8 字节	首部最小 20 字节，最大 60 字节
适用场景	适用于实时应用，例如视频会议、直播	适用于要求可靠传输的应用，例如文件传输。

3. TCP 和 UDP 的使用场景?

TCP 应用场景： 效率要求相对低，但对准确性要求相对高的场景。因为传输中需要对数据确认、重发、排序等操作，相比之下效率没有 UDP 高。例如：文件传输（准确高要求高、但是速度可以相对慢）、接受邮件、远程登录。

UDP 应用场景： 效率要求相对高，对准确性要求相对低的场景。例如：QQ 聊天、在线视频、网络语音电话（即时通讯，速度要求高，但是出现偶尔断续不是太大问题，并且此处完全不可以使用重发机制）、广播通信（广播、多播）。

4. UDP 协议为什么不可靠?

UDP 在传输数据之前不需要先建立连接，远地主机的运输层在接收到 UDP 报文后，不需要确认，提供不可靠交付。总结就以下四点：

不保证消息交付：不确认，不重传，无超时

不保证交付顺序：不设置包序号，不重排，不会发生队首阻塞

不跟踪连接状态：不必建立连接或重启状态机

不进行拥塞控制：不内置客户端或网络反馈机制

5. TCP 的重传机制?

由于 TCP 的下层网络（网络层）可能出现丢失、重复或失序的情况，TCP 协议提供可靠数据传输服务。为保证数据传输的正确性，TCP 会重传其认为已丢失（包括报文中的比特错误）的包。TCP 使用两套独立的机制来完成重传，一是基于时间，二是基于确认信息。

TCP 在发送一个数据之后，就开启一个定时器，若是在这个时间内没有收到发送数据的 ACK 确认报文，则对该报文进行重传，在达到一定次数还没有成功时放弃并发送一个复位信号。

6. TCP 的拥塞控制机制?

TCP 的拥塞控制机制主要是以下四种机制：

慢启动（慢开始）

拥塞避免

快速重传

快速恢复

（1）慢启动（慢开始）

在开始发送的时候设置 cwnd = 1（cwnd 指的是拥塞窗口）

思路：开始的时候不要发送大量数据，而是先测试一下网络的拥塞程度，由小到大增加拥塞窗口的大小。

为了防止 cwnd 增长过大引起网络拥塞，设置一个慢开始门限(ssthresh 状态变量)

当 cnwd < ssthresh，使用慢开始算法

当 cnwd = ssthresh，既可使用慢开始算法，也可以使用拥塞避免算法

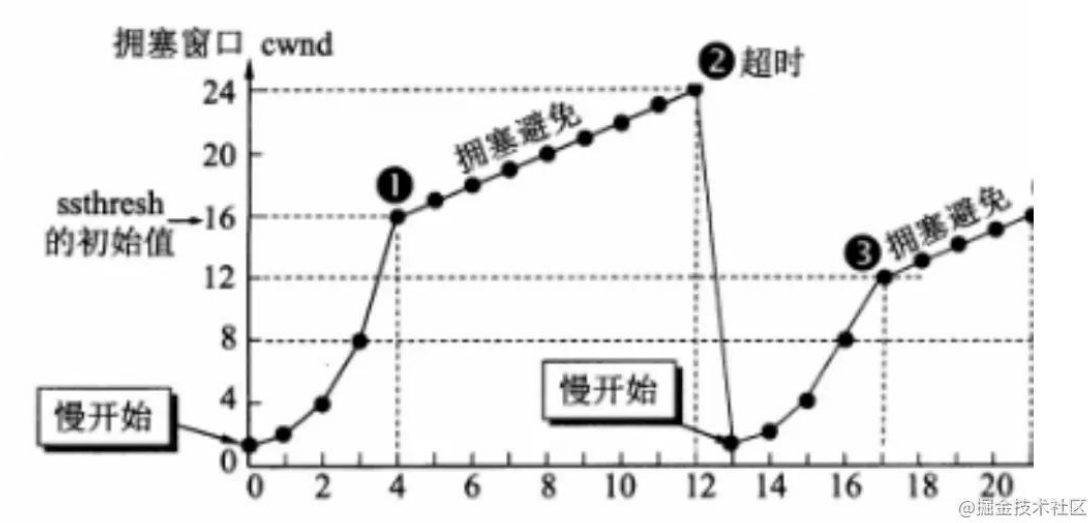
当 cnwd > ssthresh，使用拥塞避免算法

（2）拥塞避免

拥塞避免未必能够完全避免拥塞，是说在拥塞避免阶段将拥塞窗口控制为按线性增长，使网络不容易出现阻塞。

思路： 让拥塞窗口 cwnd 缓慢的增大，即每经过一个返回时间 RTT 就把发送方的拥塞控制窗口加一

无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞，就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为 1，执行慢开始算法。如图所示:



其中，判断网络出现拥塞的根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理。

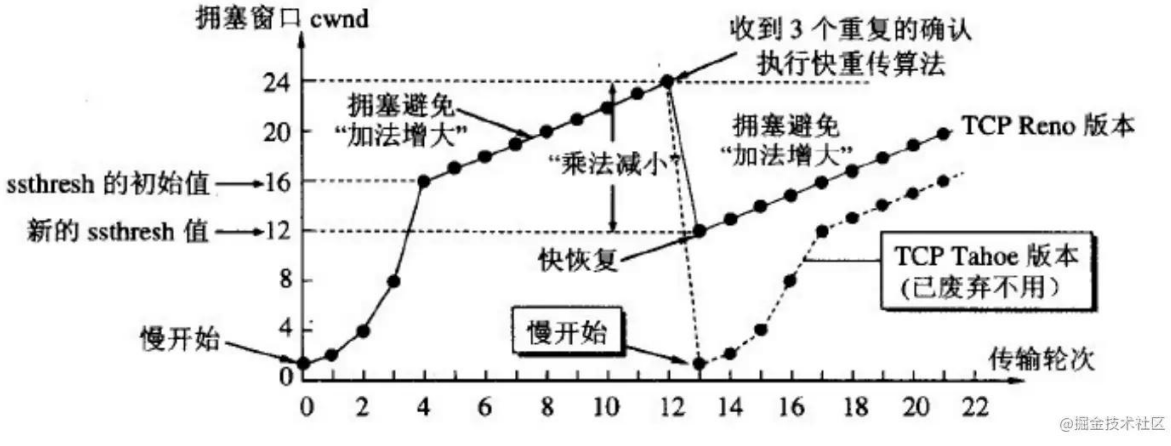
（3）快速重传

快重传要求接收方在收到一个失序的报文段后就立即发出重复确认(为的是使发送方及早知道有报文段没有到达对方)。发送方只要连续收到三个重复确认就立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

由于不需要等待设置的重传计时器到期，能尽早重传未被确认的报文段，能提高整个网络的吞吐量

（4）快速恢复

当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半。但是接下去并不执行慢开始算法。



考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。

所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。

7. TCP 的流量控制机制?

一般来说，流量控制就是为了让发送方发送数据的速度不要太快，要让接收方来得及接收。TCP 采用大小可变的滑动窗口进行流量控制，窗口大小的单位是字节。这里说的窗口大小其实就是每次传输的数据大小。

当一个连接建立时，连接的每一端分配一个缓冲区来保存输入的数据，并将缓冲区的大小发送给另一端。

当数据到达时，接收方发送确认，其中包含了自己剩余的缓冲区大小。（剩余的缓冲区空间的大小被称为窗口，指出窗口大小的通知称为窗口通告 。接收方在发送的每一确认中都含有一个窗口通告。）

如果接收方应用程序读数据的速度能够与数据到达的速度一样快，接收方将在每一确认中发送一个正的窗口通告。

如果发送方操作的速度快于接收方，接收到的数据最终将充满接收方的缓冲区，导致接收方通告一个零窗口 。发送方收到一个零窗口通告时，必须停止发送，直到接收方重新通告一个正的窗口。

8. TCP 的可靠传输机制?

TCP 的可靠传输机制是基于连续 ARQ 协议和滑动窗口协议的。

TCP 协议在发送方维持了一个发送窗口，发送窗口以前的报文段是已经发送并确认了的报文段，发送窗口中包含了已经发送但 未确认的报文段和允许发送但还未发送的报文段，发送窗口以后的报文段是缓存中还不允许发送的报文段。当发送方向接收方发 送报文时，会依次发送窗口内的所有报文段，并且设置一个定时器，这个定时器可以理解为是最早发送但未收到确认的报文段。 如果在定时器的时间内收到某一个报文段的确认回答，则滑动窗口，将窗口的首部向后滑动到确认报文段的后一个位置，此时如 果还有已发送但没有确认的报文段，则重新设置定时器，如果没有了则关闭定时器。如果定时器超时，则重新发送所有已经发送 但还未收到确认的报文段，并将超时的间隔设置为以前的两倍。当发送方收到接收方的三个冗余的确认应答后，这是一种指示， 说明该报文段以后的报文段很有可能发生丢失了，那么发送方会启用快速重传的机制，就是当前定时器结束前，发送所有的已发 送但确认的报文段。

接收方使用的是累计确认的机制，对于所有按序到达的报文段，接收方返回一个报文段的肯定回答。如果收到了一个乱序的报文 段，那么接方会直接丢弃，并返回一个最近的按序到达的报文段的肯定回答。使用累计确认保证了返回的确认号之前的报文段都 已经按序到达了，所以发送窗口可以移动到已确认报文段的后面。

发送窗口的大小是变化的，它是由接收窗口剩余大小和网络中拥塞程度来决定的，TCP 就是通过控制发送窗口的长度来控制报文 段的发送速率。

但是 TCP 协议并不完全和滑动窗口协议相同，因为许多的 TCP 实现会将失序的报文段给缓存起来，并且发生重传时，只会重 传一个报文段，因此 TCP 协议的可靠传输机制更像是窗口滑动协议和选择重传协议的一个混合体。

9. TCP 粘包是怎么回事，如何处理?

默认情况下, TCP 连接会启用延迟传送算法 (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 (缓冲大小见 socket.bufferSize), 这样可以减少 IO 消耗提高性能.

如果是传输文件的话, 那么根本不用处理粘包的问题, 来一个包拼一个包就好了。但是如果是多条消息, 或者是别的用途的数据那么就需要处理粘包.

下面看一个例子, 连续调用两次 send 分别发送两段数据 data1 和 data2, 在接收端有以下几种常见的情况:

A. 先接收到 data1, 然后接收到 data2 .

B. 先接收到 data1 的部分数据, 然后接收到 data1 余下的部分以及 data2 的全部.

C. 先接收到了 data1 的全部数据和 data2 的部分数据, 然后接收到了 data2 的余下的数据.

D. 一次性接收到了 data1 和 data2 的全部数据.

其中的 BCD 就是我们常见的粘包的情况. 而对于处理粘包的问题, 常见的解决方案有:

多次发送之前间隔一个等待时间：只需要等上一段时间再进行下一次 send 就好, 适用于交互频率特别低的场景. 缺点也很明显, 对于比较频繁的场景而言传输效率实在太低，不过几乎不用做什么处理.

关闭 Nagle 算法：关闭 Nagle 算法, 在 Node.js 中你可以通过 socket.setNoDelay() 方法来关闭 Nagle 算法, 让每一次 send 都不缓冲直接发送。该方法比较适用于每次发送的数据都比较大 (但不是文件那么大), 并且频率不是特别高的场景。如果是每次发送的数据量比较小, 并且频率特别高的, 关闭 Nagle 纯属自废武功。另外, 该方法不适用于网络较差的情况, 因为 Nagle 算法是在服务端进行的包合并情况, 但是如果短时间内客户端的网络情况不好, 或者应用层由于某些原因不能及时将 TCP 的数据 recv, 就会造成多个包在客户端缓冲从而粘包的情况。 (如果是在稳定的机房内部通信那么这个概率是比较小可以选择忽略的)

进行封包/拆包： 封包/拆包是目前业内常见的解决方案了。即给每个数据包在发送之前, 于其前/后放一些有特征的数据, 然后收到数据的时 候根据特征数据分割出来各个数据包。

10. 为什么 udp 不会粘包?

TCP 协议是面向流的协议，UDP 是面向消息的协议。UDP 段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据

UDP 具有保护消息边界，在每个 UDP 包中就有了消息头（消息来源地址，端口等信息），这样对于接收端来说就容易进行区分处理了。传输协议把数据当作一条独立的消息在网上传输，接收端只能接收独立的消息。接收端一次只能接收发送端发出的一个数据包,如果一次接受数据的大小小于发送端一次发送的数据大小，就会丢失一部分数据，即使丢失，接受端也不会分两次去接收。

11. TCP 的三次握手?

三次握手（Three-way Handshake）其实就是指建立一个 TCP 连接时，需要客户端和服务器总共发送 3 个包。进行三次握手的主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备。实质上其实就是连接服务器指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号，交换 TCP 窗口大小信息。

刚开始客户端处于 Closed 的状态，服务端处于 Listen 状态。

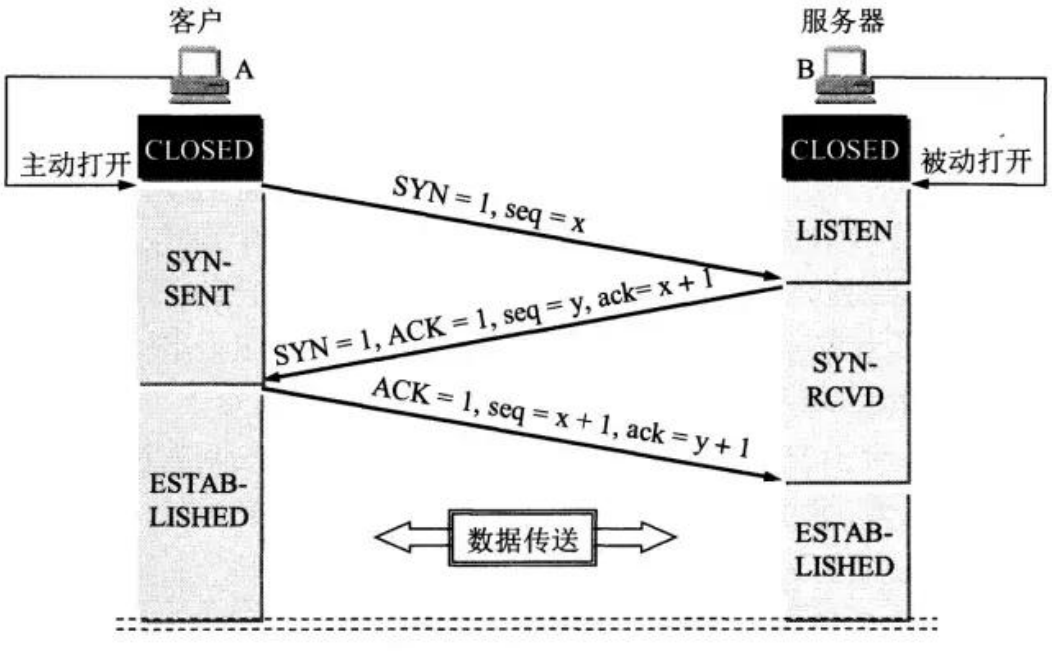


图 5-28 用三报文握手建立 TCP 连接

第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN，此时客户端处于 SYN_SEND 状态。首部的同步位 SYN=1，初始序号 seq=x，SYN=1 的报文段不能携带数据，但要消耗掉一个序号。

第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN。同时会把客户端的 ISN + 1 作为 ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 SYN_REVD 的状态。在确认报文段中 SYN=1，ACK=1，确认号 ack=x+1，初始序号 seq=y

第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立起了连接。

确认报文段 ACK=1，确认号 ack=y+1，序号 seq=x+1（初始为 seq=x，第二个报文段所以要+1），ACK 报文段可以携带数据，不携带数据则不消耗序号。

那为什么要三次握手呢？两次不行吗？

为了确认双方的接收能力和发送能力都正常

如果是用两次握手，则会出现下面这种情况：

如客户端发出连接请求，但因连接请求报文丢失而未收到确认，于是客户端再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接，客户端共发出了两个连接请求报文段，其中第一个丢失，第二个到达了服务端，但是第一个丢失的报文段只是在某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达服务端，此时服务端误认为客户端又发出一次新的连接请求，于是就向客户端发出确认报文段，同意建立连接，不采用三次握手，只要服务端发出确认，就建立新的连接了，此时客户端忽略服务端发来的确认，也不发送数据，则服务端一致等待客户端发送数据，浪费资源。

简单来说就是以下三步：

第一次握手： 客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 SYN-SENT 状态。

第二次握手： 服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 SYN-RECEIVED 状态。

第三次握手： 当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。

TCP 三次握手的建立连接的过程就是相互确认初始序号的过程，告诉对方，什么样序号的报文段能够被正确接收。 第三次握手的作用是客户端对服务器端的初始序号的确认。如果只使用两次握手，那么服务器就没有办法知道自己的序号是否 已被确认。同时这样也是为了防止失效的请求报文段被服务器接收，而出现错误的情况。

12. TCP 的四次挥手?

刚开始双方都处于 ESTABLISHED 状态，假如是客户端先发起关闭请求。四次挥手的过程如下：

第一次挥手： 客户端会发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于 FIN_WAIT1 状态。即发出连接释放报文段（FIN=1，序号 seq=u），并停止再发送数据，主动关闭 TCP 连接，进入 FIN_WAIT1（终止等待 1）状态，等待服务端的确认。

第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 CLOSE_WAIT 状态。即服务端收到连接释放报文段后即发出确认报文段（ACK=1，确认号 ack=u+1，序号 seq=v），服务端进入 CLOSE_WAIT（关闭等待）状态，此时的 TCP

处于半关闭状态，客户端到服务端的连接释放。客户端收到服务端的确认后，进入 FIN_WAIT2（终止等待 2）状态，等待服务端发出的连接释放报文段。

第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 LAST_ACK 的状态。即服务端没有要向客户端发出的数据，服务端发出连接释放报文段（FIN=1，ACK=1，序号 seq=w，确认号 ack=u+1），服务端进入 LAST_ACK（最后确认）状态，等待客户端的确认。

第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1 作为自己 ACK 报文的序列号值，此时客户端处于 TIME_WAIT 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。

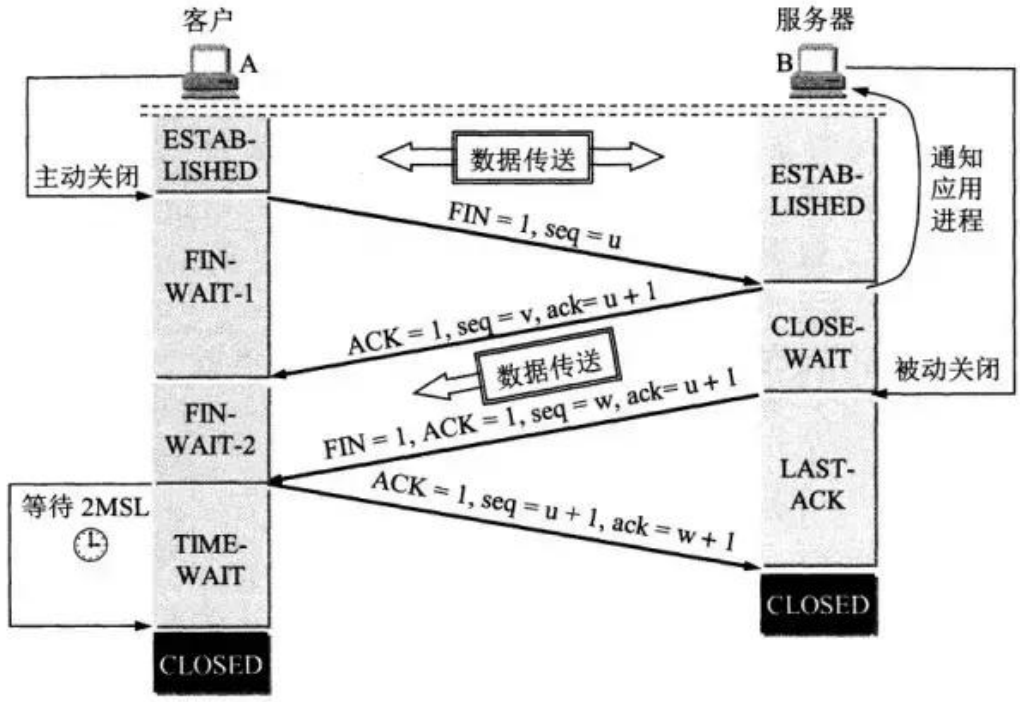


图 5-29　TCP 连接释放的过程　　@掘金技术社区

即客户端收到服务端的连接释放报文段后，对此发出确认报文段（ACK=1，seq=u+1，ack=w+1），客户端进入 TIME_WAIT（时间等待）状态。此时 TCP 未释放掉，需要经过时间等待计时器设置的时间 2MSL 后，客户端才进入 CLOSED 状态。

那为什么需要四次挥手呢？

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

简单来说就是以下四步：

第一次挥手： 若客户端认为数据发送完成，则它需要向服务端发送连接释放请求。

第二次挥手：服务端收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进

入 CLOSE_WAIT 状态，此时表明客户端到服务端的连接已经释放，不再接收客户端发的数据了。但是因为 TCP 连接是双向的，所以服务端仍旧可以发送数据给客户端。

第三次挥手：服务端如果此时还有没发完的数据会继续发送，完毕后会向客户端发送连接释放请求，然后服务端便进入 LAST-ACK 状态。

第四次挥手： 客户端收到释放请求后，向服务端发送确认应答，此时客户端进入 TIME-WAIT 状态。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有服务端的重发请求的话，就进入 CLOSED 状态。当服务端收到确认应答后，也便进入 CLOSED 状态。

TCP 使用四次挥手的原因是因为 TCP 的连接是全双工的，所以需要双方分别释放到对方的连接，单独一方的连接释放，只代 表不能再向对方发送数据，连接处于的是半释放的状态。

最后一次挥手中，客户端会等待一段时间再关闭的原因，是为了防止发送给服务器的确认报文段丢失或者出错，从而导致服务器 端不能正常关闭。

五、HTTP

1. HTTP 和 HTTPS 的基本概念?

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从 WWW 服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。

HTTPS 协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性

2. HTTP 与 HTTPS 有什么区别?

HTTP 协议传输的数据都是未加密的，也就是明文的，因此使用 HTTP 协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了 SSL（Secure Sockets Layer）协议用于对 HTTP 协议传输的数据进行加密，从而就诞生了 HTTPS。HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

HTTPS 和 HTTP 的区别主要如下：

1、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。

2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

3、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

3. 常见的 HTTP 请求方法?

GET: 向服务器获取数据；

POST：将实体提交到指定的资源，通常会造成服务器资源的修改；

PUT：上传文件，更新数据；

DELETE：删除服务器上的对象；

HEAD：获取报文首部，与 GET 相比，不返回报文主体部分；

OPTIONS：询问支持的请求方法，用来跨域请求；

CONNECT：要求在与代理服务器通信时建立隧道，使用隧道进行 TCP 通信；

TRACE：回显服务器收到的请求，主要用于测试或诊断。

4. GET 和 POST 的请求的区别?

Post 和 Get 是 HTTP 请求的两种方法，其区别如下：

应用场景： GET 请求是一个幂等的请求，一般 Get 请求用于对服务器资源不会产生影响的场景，比如说请求一个网页的资源。而 Post 不是一个幂等的请求，一般用于对服务器资源会产生影响的情景，比如注册用户这一类的操作。

是否缓存： 因为两者应用场景不同，浏览器一般会对 Get 请求缓存，但很少对 Post 请求缓存。

发送的报文格式： Get 请求的报文中实体部分为空，Post 请求的报文中实体部分一般为向服务器发送的数据。

安全性： Get 请求可以将请求的参数放入 url 中向服务器发送，这样的做法相对于 Post 请求来说是不太安全的，因为请求的 url 会被保留在历史记录中。

请求长度： 浏览器由于对 url 长度的限制，所以会影响 get 请求发送数据时的长度。这个限制是浏览器规定的，并不是 RFC 规定的。

参数类型： post 的参数传递支持更多的数据类型。

5. POST 和 PUT 请求的区别?

PUT 请求是向服务器端发送数据，从而修改数据的内容，但是不会增加数据的种类等，也就是说无论进行多少次 PUT 操作，其结果并没有不同。（可以理解为时更新数据）

POST 请求是向服务器端发送数据，该请求会改变数据的种类等资源，它会创建新的内容。（可以理解为是创建数据）。

6. OPTIONS 请求方法及使用场景?

OPTIONS 是除了 GET 和 POST 之外的其中一种 HTTP 请求方法。

OPTIONS 方法是用于请求获得由 Request-URI 标识的资源在请求/响应的通信过程中可以使用的功能选项。通过这个方法，客户端可以在采取具体资源请求之前，决定对该资源采取何种必要措施，或者了解服务器的性能。该请求方法的响应不能缓存。

OPTIONS 请求方法的主要用途有两个：

获取服务器支持的所有 HTTP 请求方法；

用来检查访问权限。例如：在进行 CORS 跨域资源共享时，对于复杂请求，就是使用 OPTIONS 方法发送嗅探请求，以判断是否有对指定资源的访问权限。

7. 常见的 HTTP 请求头和响应头有哪些?

HTTP Request Header 常见的请求头：

Accept:浏览器能够处理的内容类型

Accept-Charset:浏览器能够显示的字符集

Accept-Encoding：浏览器能够处理的压缩编码

Accept-Language：浏览器当前设置的语言

Connection：浏览器与服务器之间连接的类型

Cookie：当前页面设置的任何 Cookie

Host：发出请求的页面所在的域

Referer：发出请求的页面的 URL

User-Agent：浏览器的用户代理字符串

HTTP Responses Header 常见的响应头：

Date：表示消息发送的时间，时间的描述格式由 rfc822 定义

server:服务器名称

Connection：浏览器与服务器之间连接的类型

Cache-Control：控制 HTTP 缓存

content-type:表示后面的文档属于什么 MIME 类型

常见的 Content-Type 属性值有以下四种：

（1）application/x-www-form-urlencoded：浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。该种方式提交的数据放在 body 里面，数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL 转码。

（2）multipart/form-data：该种方式也是一个常见的 POST 提交方式，通常表单上传文件时使用该种方式。

（3）application/json：服务器消息主体是序列化后的 JSON 字符串。

（4）text/xml：该种方式主要用来提交 XML 格式的数据。

8. HTTP 1.0 和 HTTP 1.1 之间有哪些区别?

HTTP 1.0 和 HTTP 1.1 有以下区别：

连接方面，http1.0 默认使用非持久连接，而 http1.1 默认使用持久连接。http1.1 通过使用持久连接来使多个 http 请求复用同一个 TCP 连接，以此来避免使用非持久连接时每次需要建立连接的时延。

资源请求方面，在 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，并且不支持断点续传功能，http1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（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。

缓存方面，在 http1.0 中主要使用 header 里的 If-Modified-Since、Expires 来做为缓存判断的标准，http1.1 则引入了更多的缓存控制策略，例如 Etag、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。

http1.1 中新增了 host 字段，用来指定服务器的域名。http1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名（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，并且它们共享一个 IP 地址。因此有了 host 字段，这样就可以将请求发往到同一台服务器上的不同网站。

http1.1 相对于 http1.0 还新增了很多请求方法，如 PUT、HEAD、OPTIONS 等。

9. HTTP 1.1 和 HTTP 2.0 的区别?

二进制协议：HTTP/2 是一个二进制协议。在 HTTP/1.1 版中，报文的头信息必须是文本（ASCII 编码），数据体可以是文本，也可以是二进制。HTTP/2 则是一个彻底的二进制协议，头信息和数据体都是二进制，并且统称为"帧"，可以分为头信息帧和数据帧。 帧的概念是它实现多路复用的基础。

多路复用： HTTP/2 实现了多路复用，HTTP/2 仍然复用 TCP 连接，但是在一个连接里，客户端和服务器都可以同时发送多个请求或回应，而且不用按照顺序一一发送，这样就避免了"队头堵塞"【1】的问题。

数据流： HTTP/2 使用了数据流的概念，因为 HTTP/2 的数据包是不按顺序发送的，同一个连接里面连续的数据包，可能属于不同的请求。因此，必须要对数据包做标记，指出它属于哪个请求。HTTP/2 将每个请求或回应的所有数据包，称为一个数据流。每个数据流都有一个独一无二的编号。数据包发送时，都必须标记数据流 ID ，

用来区分它属于哪个数据流。

头信息压缩： HTTP/2 实现了头信息压缩，由于 HTTP 1.1 协议不带状态，每次请求都必须附上所有信息。所以，请求的很多字段都是重复的，比如 Cookie 和 User Agent ，一模一样的内容，每次请求都必须附带，这会浪费很多带宽，也影响速度。HTTP/2 对这一点做了优化，引入了头信息压缩机制。一方面，头信息使用 gzip 或 compress 压缩后再发送；另一方面，客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就能提高速度了。

服务器推送： HTTP/2 允许服务器未经请求，主动向客户端发送资源，这叫做服务器推送。使用服务器推送提前给客户端推送必要的资源，这样就可以相对减少一些延迟时间。这里需要注意的是 http2 下服务器主动推送的是静态资源，和 WebSocket 以及使用 SSE 等方式向客户端发送即时数据的推送是不同的。

【1】队头堵塞：

队头阻塞是由 HTTP 基本的“请求 - 应答”模型所导致的。HTTP 规定报文必须是“一发一收”，这就形成了一个先进先出的“串行”队列。队列里的请求是没有优先级的，只有入队的先后顺序，排在最前面的请求会被最优先处理。如果队首的请求因为处理的太慢耽误了时间，那么队列里后面的所有请求也不得不跟着一起等待，结果就是其他的请求承担了不应有的时间成本，造成了队头堵塞的现象。

10. GET 方法 URL 长度限制的原因?

实际上 HTTP 协议规范并没有对 get 方法请求的 url 长度进行限制，这个限制是特定的浏览器及服务器对它的限制。

IE 对 URL 长度的限制是 2083 字节(2K+35)。由于 IE 浏览器对 URL 长度的允许值是最小的，所以开发过程中，只要 URL 不超过 2083 字节，那么在所有浏览器中工作都不会有问题。

GET 的长度值 = URL（2083）- （你的 Domain+Path）-2（2 是 get 请求中?=两个字符的长度）

主流浏览器对 get 方法中 url 的长度限制范围：

Microsoft Internet Explorer (Browser)：IE 浏览器对 URL 的最大限制为 2083 个字符，如果超过这个数字，提交按钮没有任何反应。

Firefox (Browser)：对于 Firefox 浏览器 URL 的长度限制为 65,536 个字符。

Safari (Browser)：URL 最大长度限制为 80,000 个字符。

Opera (Browser)：URL 最大长度限制为 190,000 个字符。

Google (chrome)：URL 最大长度限制为 8182 个字符。

主流的服务器对 get 方法中 url 的长度限制范围：

Apache (Server)：能接受最大 url 长度为 8192 个字符。

Microsoft Internet Information Server(IIS)：能接受最大 url 的长度为 16384 个字符。

根据上面的数据，可以知道，get 方法中的 URL 长度最长不超过 2083 个字符，这样所有的浏览器和服务器都可能正常工作。

11. 对 keep-alive 的理解?

HTTP1.0 中默认是在每次请求/应答，客户端和服务器都要新建一个连接，完成之后立即断开连接，这就是短连接。当使用 Keep-Alive 模式时，Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接，这就是长连接。其使用方法如下：

HTTP1.0 版本是默认没有 Keep-alive 的（也就是默认会发送 keep-alive），所以要想连接得到保持，必须手动

配置发送 Connection: keep-alive 字段。若想断开 keep-alive 连接，需发送 Connection:close 字段；

HTTP1.1 规定了默认保持长连接，数据传输完成了保持 TCP 连接不断开，等待在同域名下继续用这个通道传输数据。如果需要关闭，需要客户端发送 Connection：close 首部字段。

Keep-Alive 的建立过程：

客户端向服务器在发送请求报文同时在首部添加发送 Connection 字段

服务器收到请求并处理 Connection 字段

服务器回送 Connection:Keep-Alive 字段给客户端

客户端接收到 Connection 字段

Keep-Alive 连接建立成功

服务端自动断开过程（也就是没有 keep-alive）：

客户端向服务器只是发送内容报文（不包含 Connection 字段）

服务器收到请求并处理

服务器返回客户端请求的资源并关闭连接

客户端接收资源，发现没有 Connection 字段，断开连接

客户端请求断开连接过程：

客户端向服务器发送 Connection:close 字段

服务器收到请求并处理 connection 字段

服务器回送响应资源并断开连接

客户端接收资源并断开连接

开启 Keep-Alive 的优点：

较少的 CPU 和内存的使用（由于同时打开的连接的减少了）；

允许请求和应答的 HTTP 管线化；

降低拥塞控制 （TCP 连接减少了）；

减少了后续请求的延迟（无需再进行握手）；

报告错误无需关闭 TCP 连；

开启 Keep-Alive 的缺点：

长时间的 Tcp 连接容易导致系统资源无效占用，浪费系统资源。

12. 页面有多张图片，HTTP 是怎样的加载表现?

在 HTTP 1 下，浏览器对一个域名下最大 TCP 连接数为 6，所以会请求多次。可以用多域名部署解决。这样可以提高同时请求的数目，加快页面图片的获取速度。

在 HTTP 2 下，可以一瞬间加载出来很多资源，因为，HTTP2 支持多路复用，可以在一个 TCP 连接中发送多个 HTTP 请求。

13. HTTP 与 HTTPS 有什么区别？？

HTTP2 的头部压缩算法是怎样的？

HTTP2 的头部压缩是 HPACK 算法。在客户端和服务器两端建立“字典”，用索引号表示重复的字符串，采用哈夫曼编码来压缩整数和字符串，可以达到 50%~90%的高压缩率。

具体来说:

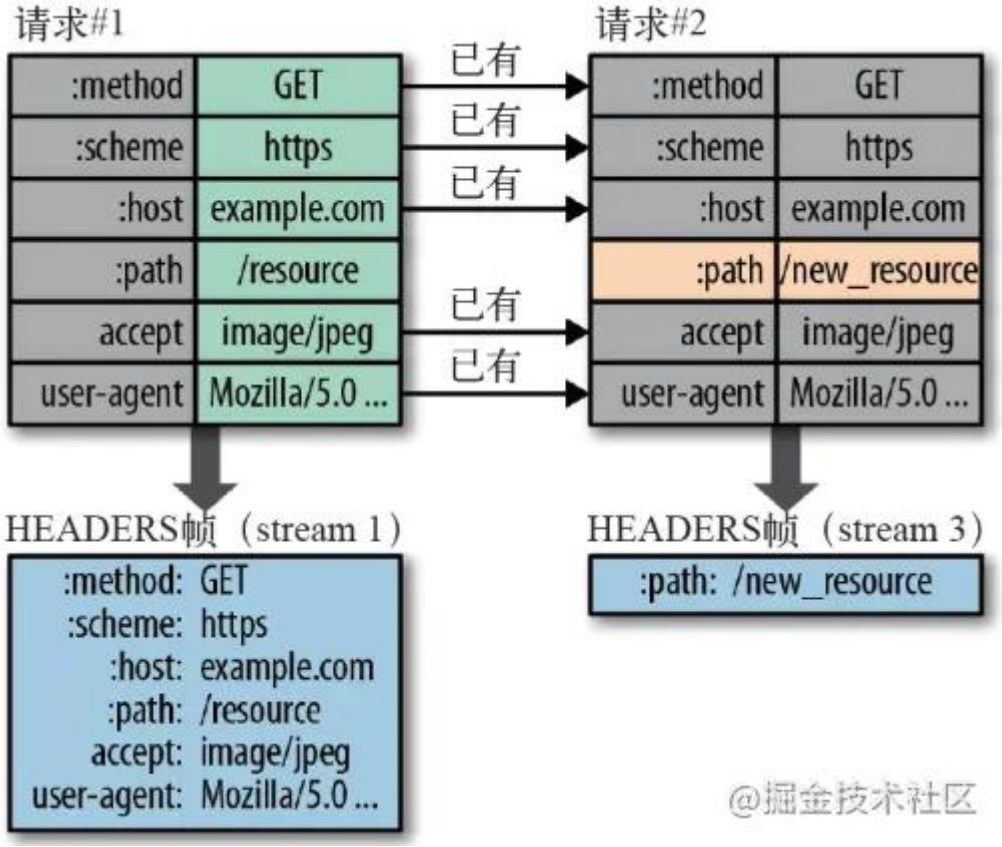
在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键值对，对于相同的数据，不再通过每次

请求和响应发送；

首部表在 HTTP/2 的连接存续期内始终存在，由客户端和服务器共同渐进地更新；

每个新的首部键值对要么被追加到当前表的末尾，要么替换表中之前的值。

例如下图中的两个请求， 请求一发送了所有的头部字段，第二个请求则只需要发送差异数据，这样可以减少冗余数据，降低开销。



14. HTTP 请求报文的是什么样的？？

请求报文有 4 部分组成:

请求行

请求头部

空行

请求体

其中：

（1）请求行包括：请求方法字段、URL 字段、HTTP 协议版本字段。它们用空格分隔。例如，GET /index.html HTTP/1.1。

（2）请求头部:请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔

User-Agent：产生请求的浏览器类型。

Accept：客户端可识别的内容类型列表。

Host：请求的主机名，允许多个域名同处一个 IP 地址，即虚拟主机。

（3）请求体: post put 等请求携带的数据

15. HTTP 响应报文的是什么样的?

请求报文有 4 部分组成:

响应行

响应头

空行

响应体

响应行：由网络协议版本，状态码和状态码的原因短语组成，例如 HTTP/1.1 200 OK 。

响应头：响应部首组成

响应体：服务器响应的数据。

16. HTTP 协议的优点和缺点?

HTTP 是超文本传输协议，它定义了客户端和服务器之间交换报文的格式和方式，默认使用 80 端口。它使用 TCP 作为传输层协议，保证了数据传输的可靠性。

HTTP 协议具有以下优点：

支持客户端/服务器模式

简单快速：客户向服务器请求服务时，只需传送请求方法和路径。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

无连接：无连接就是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接，采用这种方式可以节省传输时间。

无状态：HTTP 协议是无状态协议，这里的状态是指通信过程的上下文信息。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能会导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就比较快。

灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

HTTP 协议具有以下缺点：

无状态： HTTP 是一个无状态的协议，HTTP 服务器不会保存关于客户的任何信息。

明文传输： 协议中的报文使用的是文本形式，这就直接暴露给外界，不安全。

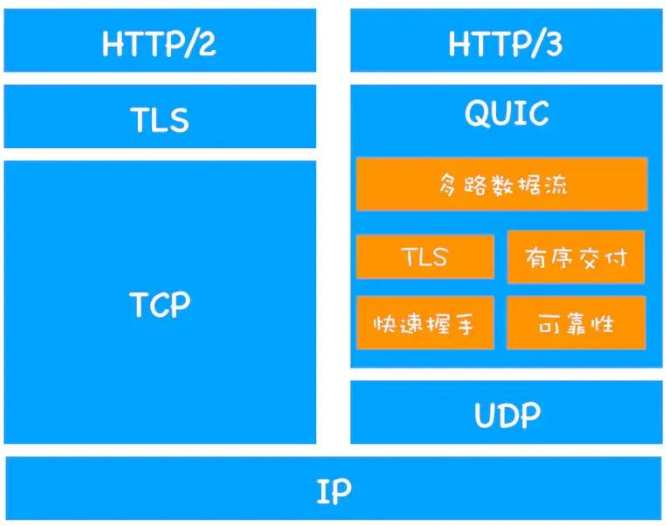
不安全：

（1）通信使用明文（不加密），内容可能会被窃听；

（2）不验证通信方的身份，因此有可能遭遇伪装；

（3）无法证明报文的完整性，所以有可能已遭篡改

17. 说一下 HTTP 3.0?



HTTP/3 基于 UDP 协议实现了类似于 TCP 的多路复用数据流、传输可靠性等功能，这套功能被称为 QUIC 协议。

流量控制、传输可靠性功能：QUIC 在 UDP 的基础上增加了一层来保证数据传输可靠性，它提供了数据包重传、拥塞控制、以及其他一些 TCP 中的特性。

集成 TLS 加密功能：目前 QUIC 使用 TLS1.3，减少了握手所花费的 RTT 数。

多路复用：同一物理连接上可以有多个独立的逻辑数据流，实现了数据流的单独传输，解决了 TCP 的队头阻塞问题。

快速握手：由于基于 UDP，可以实现使用 0 ~ 1 个 RTT 来建立连接

18. HTTP 协议的性能怎么样?

HTTP 协议是基于 TCP/IP，并且使用了请求-应答的通信模式，所以性能的关键就在这两点里。

长连接、管道网络传输

长连接

HTTP 协议有两种连接模式，一种是持续连接，一种非持续连接。

（1）非持续连接指的是服务器必须为每一个请求的对象建立和维护一个全新的连接。

（2）持续连接下，TCP 连接默认不关闭，可以被多个请求复用。采用持续连接的好处是可以避免每次建立 TCP 连接三次握手时所花费的时间。

对于不同版本的采用不同的连接方式：

在 HTTP/1.0 每发起一个请求，都要新建一次 TCP 连接（三次握手），而且是串行请求，做了无畏的 TCP 连接建立和断开，增加了通信开销。该版本使用的非持续的连接，但是可以在请求时，加上 Connection: keep-a live 来要求服务器不要关闭 TCP 连接。

在 HTTP/1.1 提出了长连接的通信方式，也叫持久连接。这种方式的好处在于减少了 TCP 连接的重复建立和断开所造成的额外开销，减轻了服务器端的负载。该版本及以后版本默认采用的是持续的连接。目前对于同一个域，大多数浏览器支持同时建立 6 个持久连接。

管道网络传输

HTTP/1.1 采用了长连接的方式，这使得管道（pipeline）网络传输成为了可能。

管道（pipeline）网络传输是指：可以在同一个 TCP 连接里面，客户端可以发起多个请求，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。但是服务器还是按照顺序回应请求。如果前面的回应特别慢，后面就会有许多请求排队等着。这称为队头堵塞。

队头堵塞

HTTP 传输的报文必须是一发一收，但是，里面的任务被放在一个任务队列中串行执行，一旦队首的请求处理太慢，就会阻塞后面请求的处理。这就是 HTTP 队头阻塞问题。

队头阻塞的解决方案：

（1）并发连接：对于一个域名允许分配多个长连接，那么相当于增加了任务队列，不至于一个队伍的任务阻塞其它所有任务。

（2）域名分片：将域名分出很多二级域名，它们都指向同样的一台服务器，能够并发的长连接数变多，解决了队头阻塞的问题。

19. 与缓存相关的 HTTP 请求头有哪些?

强缓存：

Expires

Cache-Control

协商缓存：

Etag、If-None-Match

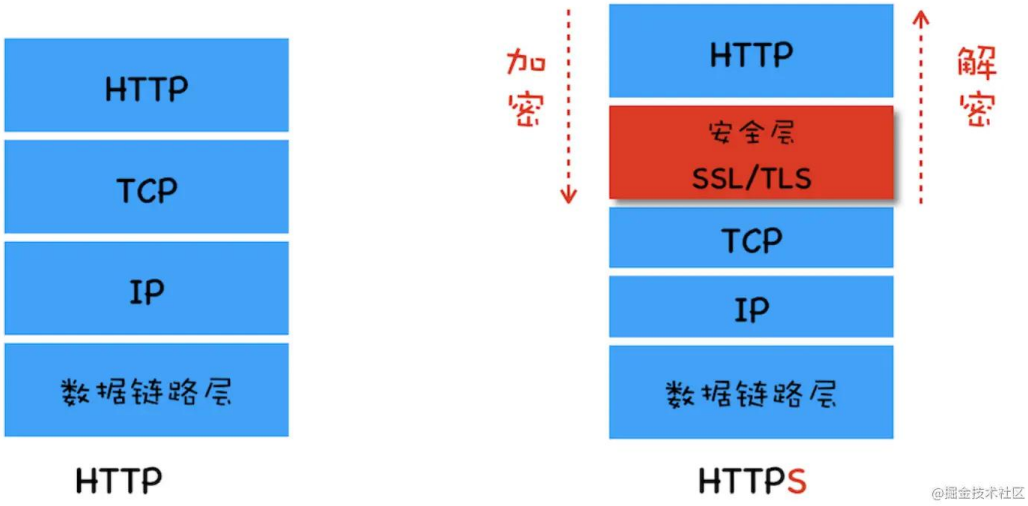
Last-Modified、If-Modified-Since。

六、HTTPS

1. HTTP 与 HTTPS 有什么区别?

什么是 HTTPS 协议？

超文本传输安全协议（Hypertext Transfer Protocol Secure，简称：HTTPS）是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，利用 SSL/TLS 来加密数据包。HTTPS 的主要目的是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。



HTTP 协议采用明文传输信息，存在信息窃听、信息篡改和信息劫持的风险，而协议 TLS/SSL 具有身份验证、信息加密和完整性校验的功能，可以避免此类问题发生。

安全层的主要职责就是对发起的 HTTP 请求的数据进行加密操作 和 对接收到的 HTTP 的内容进行解密操作。

2. TLS/SSL 的工作原理?

TLS/SSL 全称安全传输层协议（Transport Layer Security），是介于 TCP 和 HTTP 之间的一层安全协议，不影响原有的 TCP 协议和 HTTP 协议，所以使用 HTTPS 基本上不需要对 HTTP 页面进行太多的改造。

TLS/SSL 的功能实现主要依赖三类基本算法：散列函数 hash、对称加密、非对称加密。这三类算法的作用如下：

基于散列函数验证信息的完整性

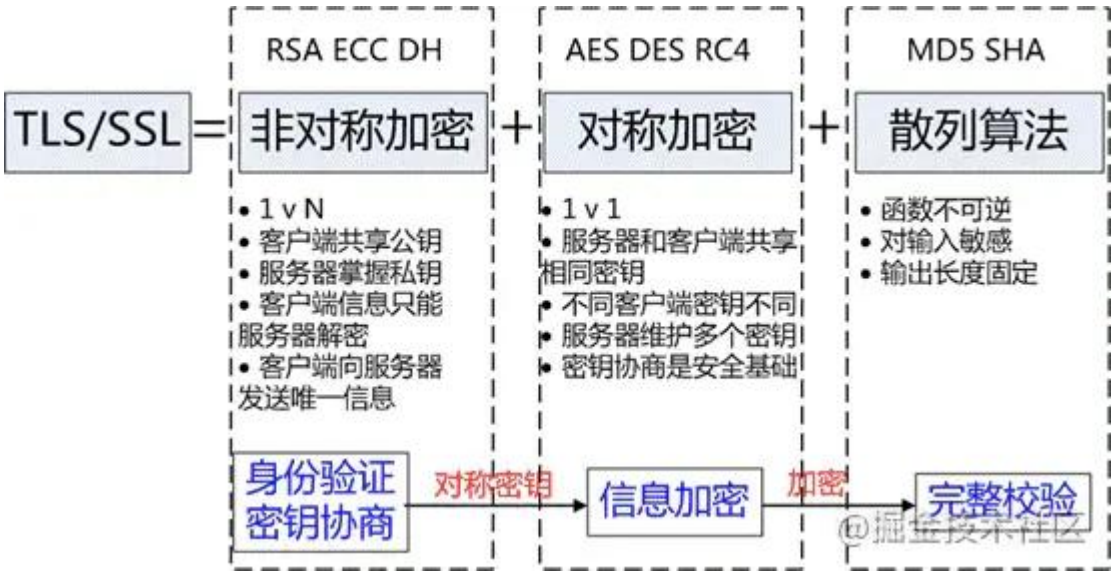
对称加密算法采用协商的秘钥对数据加密

非对称加密实现身份认证和秘钥协商

（1）散列函数 hash

常见的散列函数有 MD5、SHA1、SHA256。该函数的特点是单向不可逆，对输入数据非常敏感，输出的长度固定，任何数据的修改都会改变散列函数的结果，可以用于防止信息篡改并验证数据的完整性。

特点： 在信息传输过程中，散列函数不能三都实现信息防篡改，由于传输是明文传输，中间人可以修改信息后重新计算信息的摘要，所以需要对传输的信息和信息摘要进行加密。



（2）对称加密

对称加密的方法是，双方使用同一个秘钥对数据进行加密和解密。但是对称加密的存在一个问题，就是如何保证秘钥传输的安全性，因为秘钥还是会通过网络传输的，一旦秘钥被其他人获取到，那么整个加密过程就毫无作用了。 这就要用到非对称加密的方法。

常见的对称加密算法有 AES-CBC、DES、3DES、AES-GCM 等。相同的秘钥可以用于信息的加密和解密。掌握秘钥才能获取信息，防止信息窃听，其通讯方式是一对一。

特点： 对称加密的优势就是信息传输使用一对一，需要共享相同的密码，密码的安全是保证信息安全的基础，服务器和 N 个客户端通信，需要维持 N 个密码记录且不能修改密码。

（3）非对称加密

非对称加密的方法是，我们拥有两个秘钥，一个是公钥，一个是私钥。公钥是公开的，私钥是保密的。用私钥加密的数据，只有对应的公钥才能解密，用公钥加密的数据，只有对应的私钥才能解密。我们可以将公钥公布出去，任何想和我们通信的客户， 都可以使用我们提供的公钥对数据进行加密，这样我们就可以使用私钥进行解密，这样就能保证数据的安全了。但是非对称加密有一个缺点就是加密的过程很慢，因此如果每次通信都使用非对称加密的方式的话，反而会造成等待时间过长的问题。

常见的非对称加密算法有 RSA、ECC、DH 等。秘钥成对出现，一般称为公钥（公开）和私钥（保密）。公钥加密的信息只有私钥可以解开，私钥加密的信息只能公钥解开，因此掌握公钥的不同客户端之间不能相互解密信息，只能和服务器进行加密通信，服务器可以实现一对多的的通信，客户端也可以用来验证掌握私钥的服务器的身份。

特点： 非对称加密的特点就是信息一对多，服务器只需要维持一个私钥就可以和多个客户端进行通信，但服务器发出的信息能够被所有的客户端解密，且该算法的计算复杂，加密的速度慢。

综合上述算法特点，TLS/SSL 的工作方式就是客户端使用非对称加密与服务器进行通信，实现身份的验证并协商对称加密使用的秘钥。对称加密算法采用协商秘钥对信息以及信息摘要进行加密通信，不同节点之间采用的对称秘钥不同，从而保证信息只能通信双方获取。这样就解决了两个方法各自存在的问题。

3. 数字证书是什么?

现在的方法也不一定是安全的，因为没有办法确定得到的公钥就一定是安全的公钥。可能存在一个中间人，

截取了对方发给我们的公钥，然后将他自己的公钥发送给我们，当我们使用他的公钥加密后发送的信息，就可以被他用自己的私钥解密。然后他伪装成我们以同样的方法向对方发送信息，这样我们的信息就被窃取了，然而自己还不知道。为了解决这样的问题，可以使用数字证书。

首先使用一种 Hash 算法来对公钥和其他信息进行加密，生成一个信息摘要，然后让有公信力的认证中心（简称 CA ）用它的私钥对消息摘要加密，形成签名。最后将原始的信息和签名合在一起，称为数字证书。当接收方收到数字证书的时候，先根据原始信息使用同样的 Hash 算法生成一个摘要，然后使用公证处的公钥来对数字证书中的摘要进行解密，最后将解密的摘要和生成的摘要进行对比，就能发现得到的信息是否被更改了。

这个方法最要的是认证中心的可靠性，一般浏览器里会内置一些顶层的认证中心的证书，相当于我们自动信任了他们，只有这样才能保证数据的安全。

4. HTTPS 通信（握手）过程?

客户端向服务器发起请求，请求中包含使用的协议版本号、生成的一个随机数、以及客户端支持的加密方法。

服务器端接收到请求后，确认双方使用的加密方法、并给出服务器的证书、以及一个服务器生成的随机数。

客户端确认服务器证书有效后，生成一个新的随机数，并使用数字证书中的公钥，加密这个随机数，然后发给服 务器。并且还会提供一个前面所有内容的 hash 的值，用来供服务器检验。

服务器使用自己的私钥，来解密客户端发送过来的随机数。并提供前面所有内容的 hash 值来供客户端检验。

客户端和服务器端根据约定的加密方法使用前面的三个随机数，生成对话秘钥，以后的对话过程都使用这个秘钥来加密信息。

5. HTTPS 的特点?

HTTPS 的优点如下：

使用 HTTPS 协议可以认证用户和服务器，确保数据发送到正确的客户端和服务器；

使用 HTTPS 协议可以进行加密传输、身份认证，通信更加安全，防止数据在传输过程中被窃取、修改，确保数据安全性；

HTTPS 是现行架构下最安全的解决方案，虽然不是绝对的安全，但是大幅增加了中间人攻击的成本；

HTTPS 的缺点如下：

HTTPS 需要做服务器和客户端双方的加密个解密处理，耗费更多服务器资源，过程复杂；

HTTPS 协议握手阶段比较费时，增加页面的加载时间；

SSL 证书是收费的，功能越强大的证书费用越高；

HTTPS 连接服务器端资源占用高很多，支持访客稍多的网站需要投入更大的成本；

SSL 证书需要绑定 IP，不能再同一个 IP 上绑定多个域名。

6. HTTPS 是如何保证安全的?

对称加密：

即通信的双方都使用同一个秘钥进行加解密，对称加密虽然很简单性能也好，但是无法解决首次把秘钥发给对方的问题，很容易被黑客拦截秘钥。

非对称加密：

私钥 + 公钥= 密钥对

即用私钥加密的数据,只有对应的公钥才能解密,用公钥加密的数据,只有对应的私钥才能解密

因为通信双方的手里都有一套自己的密钥对,通信之前双方会先把自己的公钥都先发给对方

然后对方再拿着这个公钥来加密数据响应给对方,等到到了对方那里,对方再用自己的私钥进行解密

非对称加密虽然安全性更高，但是带来的问题就是速度很慢，影响性能。

解决方案：

结合两种加密方式，将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。

此时又带来一个问题，中间人问题：

如果此时在客户端和服务器之间存在一个中间人,这个中间人只需要把原本双方通信互发的公钥,换成自己的公钥,这样中间人就可以轻松解密通信双方所发送的所有数据。

所以这个时候需要一个安全的第三方颁发证书（CA），证明身份的身份，防止被中间人攻击。 证书中包括：签发者、证书用途、使用者公钥、使用者私钥、使用者的 HASH 算法、证书到期时间等。

但是问题来了，如果中间人篡改了证书，那么身份证明是不是就无效了？这个证明就白买了，这个时候需要一个新的技术，数字签名。

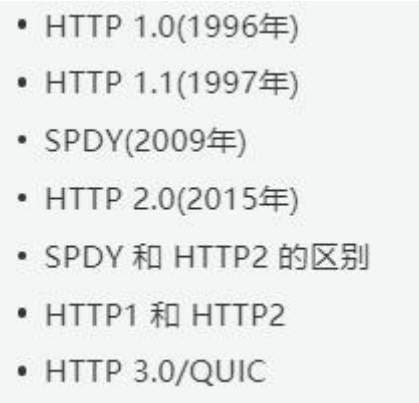
数字签名就是用CA自带的 HASH 算法对证书的内容进行HASH 得到一个摘要，再用CA 的私钥加密，最终组成数字签名。当别人把他的证书发过来的时候,我再用同样的 Hash 算法,再次生成消息摘要，然后用CA 的公钥对数字签名解密,得到 CA 创建的消息摘要,两者一比,就知道中间有没有被人篡改了。这个时候就能最大程度保证通信的安全了。

七、HTTP 参考

1. 参考?

https://juejin.cn/post/6994629873985650696#heading-12

2. HTTP 版本?



1991 年 HTTP 0.9 版，只有一个 GET，而且只支持纯文本内容，早已过时就不讲了

HTTP 1.0(1996 年)

任意数据类型都可以发送

有 GET、POST、HEAD 三种方法

无法复用 TCP 连接(长连接)

有丰富的请求响应头信息。以 header 中的 Last-Modified/If-Modified-Since 和 Expires 作为缓存标识

HTTP 1.1(1997 年)

引入更多的请求方法类型 PUT、PATCH、DELETE、OPTIONS、TRACE、CONNECT

引入长连接，就是 TCP 连接默认不关闭，可以被多个请求复用，通过请求头 connection:keep-alive 设置

引入管道连接机制，可以在同一 TCP 连接里，同时发送多个请求

强化了缓存管理和控制 Cache-Control、ETag/If-None-Match

支持分块响应，断点续传，利于大文件传输，能过请求头中的 Range 实现

使用了虚拟网络，在一台物理服务器上可以存在多个虚拟主机，并且共享一个 IP 地址

缺点：主要是连接缓慢，服务器只能按顺序响应，如果某个请求花了很长时间，就会出现请求队头阻塞

虽然出了很多优化技巧：为了增加并发请求，做域名拆分、资源合并、精灵图、资源预取...等等

最终为了推进从协议上进行优化，Google 跳出来，推出 SPDY 协议

SPDY(2009 年)

SPDY（读作“SPeeDY”）是 Google 开发的基于 TCP 的会话层协议

主要通过帧、多路复用、请求优先级、HTTP 报头压缩、服务器推送以最小化网络延迟，提升网络速度，优化用户的网络使用体验

原理是在 SSL 层上增加一个 SPDY 会话层，以在一个 TCP 连接中实现并发流。通常的 HTTP GET 和 POST 格式仍然是一样的，然而 SPDY 为编码和传输数据设计了一个新的帧格式。因为流是双向的，所以可以在客户端和服务端启动

虽然诞生后很快被所有主流浏览器所采用，并且服务器和代理也提供了支持，但是 SPDY 核心人员后来都参加到 HTTP 2.0 开发中去了，自 HTTP2.0 开发完成就不再支持 SPDY 协议了，并在 Chrome 51 中删掉了 SPDY 的支持

HTTP 2.0(2015 年)

说出 http2 中至少三个新特性？

使用新的二进制协议，不再是纯文本，避免文本歧义，缩小了请求体积

多路复用，同域名下所有通信都是在单链接(双向数据流)完成，提高连接的复用率，在拥塞控制方面有更好的能力提升

使用 HPACK 算法将头部压缩，用哈夫曼编码建立索表，传送索引大大节约了带宽

允许服务端主动推送数据给客户端

增加了安全性，使用 HTTP 2.0，要求必须至少 TLS 1.2

使用虚拟的流传输消息，解决了应用层的队头阻塞问题

缺点

TCP 以及 TCP+TLS 建立连接的延时，HTTP2 使用 TCP 协议来传输的，而如果使用 HTTPS 的话，还需要 TLS 协议进行安全传输，而使用 TLS 也需要一个握手过程，在传输数据之前，导致我们花掉 3~4 个 RTT

TCP 的队头阻塞并没有彻底解决。在 HTTP2 中，多个请求跑在一个 TCP 管道中，但当 HTTP2 出现丢包时，整个 TCP 都要开始等待重传，那么就会阻塞该 TCP 连接中的所有请求

SPDY 和 HTTP2 的区别

头部压缩算法，SPDY 是通用的 deflate 算法，HTTP2 是专门为压缩头部设计的 HPACK 算法

SPDY 必须在 TLS 上运行，HTTP2 可在 TCP 上直接使用，因为增加了 HTTP1.1 的 Upgrade 机制

SPDY 更加完善的协议商讨和确认流程

SPDY 更加完善的 Server Push 流程

SPDY 增加控制帧的种类，并对帧的格式考虑的更细致

HTTP1 和 HTTP2

HTTP2 是一个二进制协议，HTTP1 是超文本协议，传输的内容都不是一样的

HTTP2 报头压缩，可以使用 HPACK 进行头部压缩，HTTP1 则不论什么请求都会发送

HTTP2 服务端推送(Server push)，允许服务器预先将网页所需要的资源 push 到浏览器的内存当中

HTTP2 遵循多路复用，代替同一域名下的内容，只建立一次连接，HTTP1.x 不是，对域名有 6~8 个连接限制

HTTP2 引入二进制数据帧和流的概念，其中帧对数据进行顺序标识，这样浏览器收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况，同样是因为有了序列，服务器就可以并行的传输数据，这就是流所做的事情。HTTP2 对同一域名下所有请求都是基于流的，也就是说同一域名下不管访问多少文件，只建立一次连接

HTTP 3.0/QUIC

由于 HTTP 2.0 依赖于 TCP，TCP 有什么问题那 HTTP2 就会有什么问题。最主要的还是队头阻塞，在应用层的问题解决了，可是在 TCP 协议层的队头阻塞还没有解决。

TCP 在丢包的时候会进行重传，前面有一个包没收到，就只能把后面的包放到缓冲区，应用层是无法取数据的，也就是说 HTTP2 的多路复用并行性对于 TCP 的丢失恢复机制不管用，因此丢失或重新排序的数据都会导致交互挂掉

为了解决这个问题，Google 又发明了 QUIC 协议

并在 2018 年 11 月将 QUIC 正式改名为 HTTP 3.0

特点：

在传输层直接干掉 TCP，用 UDP 替代

实现了一套新的拥塞控制算法，彻底解决 TCP 中队头阻塞的问题

实现了类似 TCP 的流量控制、传输可靠性的功能。虽然 UDP 不提供可靠性的传输，但 QUIC 在 UDP 的基础之上增加了一层来保证数据可靠性传输。它提供了数据包重传、拥塞控制以及其他一些 TCP 中存在的特性

实现了快速握手功能。由于 QUIC 是基于 UDP 的，所以 QUIC 可以实现使用 0-RTT 或者 1-RTT 来建立连接，这意味着 QUIC 可以用最快的速度来发送和接收数据。

集成了 TLS 加密功能。目前 QUIC 使用的是 TLS1.3。

八、状态码

1. 状态码的类别?

类别	信息	描述
1xx	Informational(信息性状态码)	接受的请求正在处理
2xx	Success(成功状态码)	请求正常处理完毕
3xx	Redirection(重定向状态码)	需要进行附加操作一完成请求
4xx	Client Error (客户端错误状态码)	服务器无法处理请求
5xx	Server Error(服务器错误状态码)	服务器处理请求出错。

2. 2XX (Success 成功状态码)

状态码 2XX 表示请求被正常处理了。

（1）200 OK

200 OK 表示客户端发来的请求被服务器端正常处理了。

（2）204 No Content

该状态码表示客户端发送的请求已经在服务器端正常处理了，但是没有返回的内容，响应报文中不包含实体的主体部分。一般在只需要从客户端往服务器端发送信息，而服务器端不需要往客户端发送内容时使用。

（3）206 Partial Content

该状态码表示客户端进行了范围请求，而服务器端执行了这部分的 GET 请求。响应报文中包含由 Content-Range 指定范围的实体内容。

3. 3XX (Redirection 重定向状态码)

3XX 响应结果表明浏览器需要执行某些特殊的处理以正确处理请求。

（1）301 Moved Permanently

永久重定向。

该状态码表示请求的资源已经被分配了新的 URI，以后应使用资源指定的 URI。新的 URI 会在 HTTP 响应头中的 Location 首部字段指定。若用户已经把原来的 URI 保存为书签，此时会按照 Location 中新的 URI 重新保存该书签。同时，搜索引擎在抓取新内容的同时也将旧的网址替换为重定向之后的网址。

使用场景：

当我们想换个域名，旧的域名不再使用时，用户访问旧域名时用 301 就重定向到新的域名。其实也是告诉搜索引擎收录的域名需要对新的域名进行收录。

在搜索引擎的搜索结果中出现了不带 www 的域名，而带 www 的域名却没有收录，这个时候可以用 301 重定向来告诉搜索引擎我们目标的域名是哪一个。

（2）302 Found

临时重定向。

该状态码表示请求的资源被分配到了新的 URI，希望用户（本次）能使用新的 URI 访问资源。和 301 Moved Permanently 状态码相似，但是 302 代表的资源不是被永久重定向，只是临时性质的。也就是说已移动的资源对应的 URI 将来还有可能发生改变。若用户把 URI 保存成书签，但不会像 301 状态码出现时那样去更新书签，而是仍旧保留返回 302 状态码的页面对应的 URI。同时，搜索引擎会抓取新的内容而保留旧的网址。因为服务器返回 302 代码，搜索引擎认为新的网址只是暂时的。

使用场景：

当我们在做活动时，登录到首页自动重定向，进入活动页面。

未登陆的用户访问用户中心重定向到登录页面。

访问 404 页面重新定向到首页。

（3）303 See Other

该状态码表示由于请求对应的资源存在着另一个 URI，应使用 GET 方法定向获取请求的资源。

303 状态码和 302 Found 状态码有着相似的功能，但是 303 状态码明确表示客户端应当采用 GET 方法获取资源。

303 状态码通常作为 PUT 或 POST 操作的返回结果，它表示重定向链接指向的不是新上传的资源，而是另外一个页面，比如消息确认页面或上传进度页面。而请求重定向页面的方法要总是使用 GET。

注意：

当 301、302、303 响应状态码返回时，几乎所有的浏览器都会把 POST 改成 GET，并删除请求报文内的主体，之后请求会再次自动发送。

301、302 标准是禁止将 POST 方法变成 GET 方法的，但实际大家都会这么做。

（4）304 Not Modified

浏览器缓存相关。

该状态码表示客户端发送附带条件的请求时，服务器端允许请求访问资源，但未满足条件的情况。304 状态码返回时，不包含任何响应的主体部分。304 虽然被划分在 3XX 类别中，但是和重定向没有关系。

带条件的请求（Http 条件请求）：使用 Get 方法 请求，请求报文中包含（if-match、if-none-match、if-modified-since、if-unmodified-since、if-range）中任意首部。

状态码 304 并不是一种错误，而是告诉客户端有缓存，直接使用缓存中的数据。返回页面的只有头部信息，是没有内容部分的，这样在一定程度上提高了网页的性能。

（5）307 Temporary Redirect

307 表示临时重定向。 该状态码与 302 Found 有着相同含义，尽管 302 标准禁止 POST 变成 GET，但是实际使用时还是这样做了。

307 会遵守浏览器标准，不会从 POST 变成 GET。但是对于处理请求的行为时，不同浏览器还是会出现不同的情况。规范要求浏览器继续向 Location 的地址 POST 内容。规范要求浏览器继续向 Location 的地址 POST 内容。

4. 4XX (Client Error 客户端错误状态码)

4XX 的响应结果表明客户端是发生错误的原因所在。

（1）400 Bad Request

该状态码表示请求报文中存在语法错误。当错误发生时，需修改请求的内容后再次发送请求。另外，浏览器会像 200 OK 一样对待该状态码。

（2）401 Unauthorized

该状态码表示发送的请求需要有通过 HTTP 认证(BASIC 认证、DIGEST 认证)的认证信息。若之前已进行过一次请求，则表示用户认证失败

返回含有 401 的响应必须包含一个适用于被请求资源的 WWW-Authenticate 首部用以质询(challenge)用户信息。当浏览器初次接收到 401 响应，会弹出认证用的对话窗口。

以下情况会出现 401：

401.1 - 登录失败。

401.2 - 服务器配置导致登录失败。

401.3 - 由于 ACL 对资源的限制而未获得授权。

401.4 - 筛选器授权失败。

401.5 - ISAPI/CGI 应用程序授权失败。

401.7 - 访问被 Web 服务器上的 URL 授权策略拒绝。这个错误代码为 IIS 6.0 所专用。

（3）403 Forbidden

该状态码表明请求资源的访问被服务器拒绝了，服务器端没有必要给出详细理由，但是可以在响应报文实体的主体中进行说明。进入该状态后，不能再继续进行验证。该访问是永久禁止的，并且与应用逻辑密切相关。

IIS 定义了许多不同的 403 错误，它们指明更为具体的错误原因：

403.1 - 执行访问被禁止。

403.2 - 读访问被禁止。

403.3 - 写访问被禁止。

403.4 - 要求 SSL。

403.5 - 要求 SSL 128。

403.6 - IP 地址被拒绝。

403.7 - 要求客户端证书。

403.8 - 站点访问被拒绝。

403.9 - 用户数过多。

403.10 - 配置无效。

403.11 - 密码更改。

403.12 - 拒绝访问映射表。

403.13 - 客户端证书被吊销。

403.14 - 拒绝目录列表。

403.15 - 超出客户端访问许可。

403.16 - 客户端证书不受信任或无效。

403.17 - 客户端证书已过期或尚未生效

403.18 - 在当前的应用程序池中不能执行所请求的 URL。这个错误代码为 IIS 6.0 所专用。

403.19 - 不能为这个应用程序池中的客户端执行 CGI。这个错误代码为 IIS 6.0 所专用。

403.20 - Passport 登录失败。这个错误代码为 IIS 6.0 所专用。

（4）404 Not Found

该状态码表明服务器上无法找到请求的资源。除此之外，也可以在服务器端拒绝请求且不想说明理由时使用。以下情况会出现 404：

404.0 -（无） – 没有找到文件或目录。

404.1 - 无法在所请求的端口上访问 Web 站点。

404.2 - Web 服务扩展锁定策略阻止本请求。

404.3 - MIME 映射策略阻止本请求。

（5）405 Method Not Allowed

该状态码表示客户端请求的方法虽然能被服务器识别，但是服务器禁止使用该方法。GET 和 HEAD 方法，服务器应该总是允许客户端进行访问。客户端可以通过 OPTIONS 方法（预检）来查看服务器允许的访问方法，如下

Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE

5. 5XX (Server Error 服务器错误状态码)

5XX 的响应结果表明服务器本身发生错误.

（1）500 Internal Server Error

该状态码表明服务器端在执行请求时发生了错误。也有可能是 Web 应用存在的 bug 或某些临时的故障。

（2）502 Bad Gateway

该状态码表明扮演网关或代理角色的服务器，从上游服务器中接收到的响应是无效的。注意，502 错误通常不是客户端能够修复的，而是需要由途经的 Web 服务器或者代理服务器对其进行修复。以下情况会出现 502：

502.1 - CGI （通用网关接口）应用程序超时。

502.2 - CGI （通用网关接口）应用程序出错。

（3）503 Service Unavailable

该状态码表明服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。如果事先得知解除以上状况需要的时间，最好写入 RetryAfter 首部字段再返回给客户端。

使用场景：

服务器停机维护时，主动用 503 响应请求；

nginx 设置限速，超过限速，会返回 503。

（4）504 Gateway Timeout

该状态码表示网关或者代理的服务器无法在规定的时间内获得想要的响应。他是 HTTP 1.1 中新加入的。

使用场景：代码执行时间超时，或者发生了死循环。

6. 状态码查询?

（1）2XX 成功

200 OK，表示从客户端发来的请求在服务器端被正确处理

204 No content，表示请求成功，但响应报文不含实体的主体部分

205 Reset Content，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容

206 Partial Content，进行范围请求

（2）3XX 重定向

301 moved permanently，永久性重定向，表示资源已被分配了新的 URL

302 found，临时性重定向，表示资源临时被分配了新的 URL

303 see other，表示资源存在着另一个 URL，应使用 GET 方法获取资源

304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况

307 temporary redirect，临时重定向，和 302 含义类似，但是期望客户端保持请求方法不变向新的地址发出请求

（3）4XX 客户端错误

400 bad request，请求报文存在语法错误

401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息

403 forbidden，表示对请求资源的访问被服务器拒绝

404 not found，表示在服务器上没有找到请求的资源

（4）5XX 服务器错误

500 internal sever error，表示服务器端在执行请求时发生了错误

501 Not Implemented，表示服务器不支持当前请求所需要的某个功能

503 service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求

7. 307，303，302 的区别?

302 是 http1.0 的协议状态码，在 http1.1 版本的时候为了细化 302 状态码又出来了两个 303 和 307。

303 明确表示客户端应当采用get方法获取资源，他会把 POST 请求变为 GET 请求进行重定向。

307 会遵照浏览器标准，不会从 post 变为 get。

8. HTTP 状态码 304 是多好还是少好?

服务器为了提高网站访问速度，对之前访问的部分页面指定缓存机制，当客户端在此对这些页面进行请求，服务器会根据缓存内容判断页面与之前是否相同，若相同便直接返回 304，此时客户端调用缓存内容，不必进行二次下载。

状态码 304 不应该认为是一种错误，而是对客户端有缓存情况下服务端的一种响应。

搜索引擎蜘蛛会更加青睐内容源更新频繁的网站。通过特定时间内对网站抓取返回的状态码来调节对该网站的抓取频次。若网站在一定时间内一直处于 304 的状态，那么蜘蛛可能会降低对网站的抓取次数。相反，若网站

变化的频率非常之快，每次抓取都能获取新内容，那么日积月累，的回访率也会提高。

产生较多 304 状态码的原因：

页面更新周期长或不更新

纯静态页面或强制生成静态 html

304 状态码出现过多会造成以下问题：

网站快照停止；

收录减少；

权重下降。