

尚硅谷 WEB 前端技术之高频面试题

(作者：尚硅谷研究院)

版本：V1.1.1

目录

第 1 章 核心技术.....	8
1.1 Html5	8
1.1.1 Html5 新特性.....	8
1.1.2 常用元素分类及特点.....	13
1.1.3 如何解决 inline & inline-block 的空白间隙问题?	15
1.1.4 iframe.....	15
1.1.5 说说前台数据存储:	20
1.1.6 谈谈 websocket 的理解	21
1.2 Css3.....	22
1.2.1 新增特性.....	22
1.2.2 盒模型.....	23
1.2.3 BFC	24
1.2.4 选择权权重 & 优先级.....	25
1.2.5 CSS 预处理器(Sass/Less/Stylus)	25
1.2.6 flex(伸缩盒模型)布局.....	26
1.2.7 经典布局实现两栏布局(左侧固定 + 右侧自适应布局)	27
1.2.8 隐藏页面元素方式.....	29
1.2.9 让元素垂直水平居中方式.....	29
1.2.10 使用 css 实现一个三角形.....	30
1.2.11 盒子垂直水平居中.....	31
1.3 Javascript.....	33
1.3.1 JS 中原型的理解	33
1.3.2 JS 执行上下文的理解:	34
1.3.3 JS 作用域?	35
1.3.4 JS 闭包.....	35
1.3.5 JS 单线程异步编程语言的理解.....	36
1.3.6 JS 数据类型.....	37
1.3.7 函数传递是值传递还是引用传递?	37

1.3.8 区分执行函数定义与执行函数.....	38
1.3.9 说说变量提升与函数提升.....	38
1.3.10 如何判断函数中的 this.....	39
1.3.11 如何改变函数中的 this.....	39
1.3.12 如何判断 js 的数据类型.....	40
1.3.13 说说面向对象的三大特性.....	41
1.3.14 说说 JS 的垃圾回收机制.....	42
1.3.15 谈谈强缓存和协商缓存.....	45
1.3.16 区分内存溢出与内存泄露.....	46
1.3.17 区分一下同步和异步.....	47
1.3.18 正则表达式.....	48
1.3.19 谈谈对事件冒泡和事件委托的理解.....	48
1.3.20 区分 call 和 bind.....	49
1.3.21 区分函数防抖和节流.....	49
1.3.22 自定义 new.....	52
1.3.23 说说重排与重绘.....	53
1.4 ES6.....	55
1.4.1 var, let, const 的区别?	55
1.4.2 谈谈对 promise 的理解?	55
1.4.3 谈谈对 ES6 新特性的理解.....	57
1.4.4 谈谈对 TS 的理解.....	58
1.4.5 编码实现继承.....	61
1.4.6 HTTP 协议的理解.....	63
1.4.7 长链接和短链接的理解.....	65
1.4.8 HTTP1.1 和 HTTP2.0 的区别.....	66
1.4.9 HTTP 和 HTTPS 的区别.....	67
1.4.10 常见的 HTTP 状态码以及代表的意义.....	67
1.4.11 AJAX 的理解.....	68
1.4.12 区分 AJAX 与一般 HTTP 请求.....	69
1.4.13 AJAX 跨域.....	70
1.4.14 说说项目中的 axios 的二次封装.....	71
1.4.15 说说 axios 的整体执行流程.....	72
1.4.16 从输入 url 到渲染出页面的整个过程.....	74
1.4.17 自定义数组扁平化.....	75
1.4.18 axios 的理解:	76

1.4.19 谈谈 webpack.....	77
1.4.20 vite 和 webpack 的区别.....	80
1.5 Vue	81
1.5.1 谈谈数据代理的理解？谈谈对 vue 的理解？	81
1.5.2 模版解析的理解？	81
1.5.3 数据劫持的理解.....	82
1.5.4 数据响应式的理解.....	84
1.5.5 双向数据绑定指令 v-model 实现的流程：	85
1.5.6 谈谈 nextTick 的理解.....	85
1.5.7 谈谈对 vue2 的理解.....	85
1.5.8 vue2 与 vue3 的区别	93
1.6 react.....	98
1.6.1 谈谈 react 和 vue 的区别	98
1.6.2 redux 和 mobx 的区别？	100
1.6.3 react 中常见的 hook.....	100
1.6.4 react 中定义组件的 2 种方式：	101
1.6.5 react 中的路由	102
1.7 小程序.....	103
1.7.1 谈谈小程序的生命周期.....	103
1.7.2 小程序中通信方案.....	104
1.7.3 原生小程序和 uniapp 的区别	104
1.7.4 小程序登陆流程.....	105
1.7.5 小程序的支付流程.....	105
1.7.6 小程序的分包.....	105
第 2 章 Vue2 前台项目	105
2.1 项目介绍.....	105
2.1.1 说说流程.....	105
2.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块	106
2.2 人员配置参考.....	106
2.2.1 整体架构.....	106
2.2.2 人员配置.....	106
2.3 特定功能实现.....	106
2.3.1 首页三级分类业务逻辑思路.....	106
2.3.2 详情页中销售属性功能实现流程.....	107
2.3.3 购物车功能实现的业务逻辑思路.....	108

2.3.4 模态对话框组件封装思路:	109
2.3.5 项目中的 loading 是怎么实现的? 具体怎么实现的?	110
2.3.6 不停点击按钮不停发请求怎么解决?	110
2.3.7 单点登陆.....	110
2.3.8 token 无感刷新怎么实现	110
2.3.9 项目中有前台商城, 支付功能怎么实现的	110
2.3.10 做项目时有没有做过断网处理.....	110
2.3.11 大文件上传思路	110
2.4 封装组件.....	111
2.4.1 你自己封装过组件吗, 以及二次封装? 如果让你封装一个 UI 组件, 你会考虑到什么问题?	111
2.4.2 封装 axios, 公司没有自己封装好的, 直接使用的么?	111
2.4.3 项目开发中, 特别大项目中, 怎么封装抽离复用逻辑?	111
2.5 项目优化.....	111
2.5.1 代码层面的优化:	111
2.5.2 打包工具方面的优化:	112
2.5.3 网络层面的优化 (需要开启服务端的设置):	112
2.5.4 下拉框里的数据可能会有上万条, 如何处理	114
2.5.5 seo 优化怎么做的	114
2.5.6 白屏是因为什么	114
2.5.7 vue 项目的首屏优化.....	114
2.5.8 在项目中如果有卡顿的情况, 怎么解决和排查	114
2.5.9 如果一个项目中很多 v-if 假如说 1-10 有十个功能, 如何进行一个项目优化	114
2.5.10 项目久了之后, 内存占用会越来越大, 启动会特别慢, 应该怎么解决 ...	114
2.5.11 重复点击优化 (除了防抖节流, 还有什么方法)	114
2.5.12 10MB 的图片, 通过什么方法压缩到 1MB, 减少数据大小	114
2.6 大屏适配方案.....	114
2.6.1 移动端适配方案.....	114
2.6.2 1px 像素框.....	117
2.6 针对 word、excel、pdf 等文档处理, 所用到的 插件:	118
2.7 diff 算法的理解.....	119
2.9 项目中遇到哪些问题及解决方案.....	120
2.10 项目非技术问题.....	121
2.10.1 公司这么久, 就做这几个项目?	121

2.10.2 项目啥时候做的，上架了么	121
2.10.3 一直问项目里面的具体操作是项目经理安排的任务还是谁安排的，你做登录注册花费了几天，你到那是新项目还是已经搭建好了的	121
2.10.4 若依框架了解过吗.....	121
2.10.5 项目数量，离职原因，人员比例	121
2.10.6 有没有带组经验.....	121
2.10.7 上一家公司主要从事什么业务.....	121
2.10.8 详细分享简历中一个项目，这个项目是从 0 开始的吗	121
2.10.9 组长分配给这个项目什么模块，项目最开始要上线吗	121
2.10.10 个人负责的模块中技术难点，技术难点怎么解决，解决思路或解决方案	121
2.10.11 为什么要把技术难点交给我，这个项目人员分配，项目周期是多少	121
2.10.12 大概有这些:项目迭代周期，	121
2.10.13 每次迭代需求多少个，你负责多少个，	121
2.10.14 每次上线你负责的需求有多少 bug.....	121
2.10.15 说一下产品每次迭代有多少需求，每次上线有多少 bug 产生，怎么和产品后端沟通.....	121
2.10.16 怎么跟项目组的人相处.....	121
2.10.17 后台系统开发周期多长？全部是你独立完成的吗?.....	122
2.10.18 后台项目给那些人用，对应模块哪种类型人员	122
2.10.19 工作流程.....	122
2.10.20 开发人员和项目经理是否在一个工作台上	122
2.10.21 怎样收到测试测出的 bug.....	122
2.10.22 上一家公司的话团队是有几个人？人员分配.....	122
2.10.23 对于之前的项目有什么心得，或者是有什么难，包括难点，包括心得体会。	122
2.10.24 然后你们那个项目管理用的什么工具？就代码管理 Git 的常用命令你知道吗.....	122
2.10.25 然后你们平时项目的整个开发流程能简单的介绍一下吗？比如产品出了一个需求，然后你到你们这里，你们需要做一些什么？	122
2.10.26 开了需求评审之后到你开发这个里面的步骤你可以详细介绍一下	122
2.10.27 你和里面和后端的一些接口，接口的一些对接是通过什么方式？	122
2.10.28 那你们联调阶段是有有联调阶段吗？还是在开发阶段就直接有分的这么细吗？	122
2.10.29 你之前的那个公司的话，你主要是团队组成是怎样子的？	122

2.10.30 那你们那边的话主要分工的话是怎么进行一个分工? 如说你们怎么去日常的话是怎么去安排你的一些前端的一些需求的? 你和你的前端的话又是怎么进行一个分工.....	122
2.10.31 你简单说一下, 你在上一家公司, 在团队里面充当的一个角色, 然后话你所负责的模块有哪些?	123
2.10.32 相对来说亮一点方面的话, 可能没有说特别的突出, 那譬如说你觉得你在项目之前那个项目里面, 你觉得比较有挑战性的一个需求, 你是怎么去实现它的? 最后话你是如何去解决它的?	123
2.10.33 线上部署项目做哪些配置知道么?最后输入的文件是什么?dist 文件夹如果文件比较多,怎么处理,有没有做过分包的处理?	123
2.10.34 有没有了解 nginx?.....	123
2.10.35 你工作三年多,对项目的产品设计这个概念有什么理解?	123
2.10.36 作为一个前端开发人员,你觉得你最大的优势是什么?	123
2.10.37 平时开发用的什么开发工具,前一家公司研发同事有多少人,一个部门多少人.....	123
2.10.38 平时做项目的时候怎么管理需求.....	123
2.10.39 代码提交测试发布的流程, 项目发布管理的流程	123
2.10.40 测试通过后各种环境, 上线一整个流程.....	123
2.10.41 怎么托管项目.....	123
2.10.42 怎么和后端对接.....	123
2.10.43 功能提出需求, 到发布, 到测试, 到发布生产的流程	123
2.10.44 本地会和开发进行连调吗, 开发会提供接口地	123
2.10.45 平时做项目的时候怎么管理需求.....	123
2.10.46 代码提交测试发布的流程, 项目发布管理的流程	123
2.10.47 测试通过后各种环境, 上线一整个流程.....	123
2.10.48 本地会和开发进行联调吗, 开发会提供接口地址吗	124
2.10.49 本地调用服务测试后怎么发布的流程.....	124
2.10.50 提测用什么工具.....	124
2.10.51 测试提出 bug, 修改 bug 的流程.....	124
2.10.52 改完 bug 后发布测试的流程.....	124
2.10.53 线上出 bug 后怎么快速定位 bug.....	124
2.10.54 有什么代码规范.....	124
2.10.55 代码 codereview 有人管理代码提交审核吗.....	124
2.10.56 有发布吗, 知道测试环境吗.....	124
2.10.57 生产环境有吗.....	124

2.10.58 后端给的接口地址一般是什么	124
2.10.59 本地怎么测.....	124
2.10.60 后端会给什么后台地址.....	124
2.10.61 后端给的地址调不通，怎么处理.....	124
2.10.62 服务地址有对应的测试环境，生产环境的地址，怎么区别测试环境和生产环境.....	124
2.10.63 你认为什么是好的代码习惯.....	124
2.10.64 有没有作品可以看的.....	124
2.10.65 技术更新迭代这么快，你怎么保持一个持续学习的动力，有没有关注技术的发展.....	124
第 3 章 Vue3 后台项目	124
3.1 项目介绍.....	125
3.1.1 说说流程.....	125
3.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块	125
3.2 特定功能实现.....	125
3.2.1 登陆实现思路.....	125
3.2.2 平台属性管理思路.....	126
3.2.3 路由鉴权.....	127
3.2.4 按钮鉴权.....	128
3.2.5 项目中有没有做过断网处理.....	129
3.2.6 vue3 项目的兼容问题.....	129
3.2.7 vue3 中常的组件通信方式和 vue2 的有什么不一样.....	129
3.2.8 自定义指令的应用场景.....	129
3.2.9 登录鉴权权限控制流程.....	129
3.3 项目遇到的最大问题及解决方案（项目中让你成长的点）	129
3.4 项目优化.....	129
3.4.1 Vue3 运行项目很慢,怎么解决?	129
第 4 章 小程序项目.....	129
4.1 项目介绍.....	129
4.1.1 说说流程.....	129
4.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块	129
4.2 人员配置.....	129
4.3 特定功能实现.....	129
4.3.1 小程序做过分包吗？ 遇到过主包太大无法上传的问题吗?	129
4.3.2 小程序上线时怎么解决 ios 和安卓不适配	130

4.3.3 小程序登陆功能.....	130
4.3.4 小程序支付功能实现流程.....	130
4.4 项目优化.....	130
4.5 项目遇到的最大问题及解决方案（项目中让你成长的点）	130
第 5 章 react 项目	130
5.1 项目介绍.....	130
5.1.1 说说流程.....	130
5.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块	130
5.2 特定功能实现.....	130
5.3 项目遇到的最大问题及解决方案（项目中让你成长的点）	130
5.4 项目优化.....	130
第 6 章 算法题（LeetCode）	130
6.1 基本算法.....	130
6.1.1 冒泡排序.....	130
6.1.2 快速排序.....	131
6.1.3 选择排序.....	131
6.1.4 插入排序.....	132
第 7 章 场景题.....	133
7.1 弹性盒实现 332 或者 331 布局，怎么让第三排的盒子对齐左侧（就是三阶魔方没有第九个块呈现出的效果）	133
第 8 章 面试说明.....	133
8.1 面试过程最关键的是什么？	133
8.2 面试时该怎么说？	133
1) 语言表达清楚.....	133
2) 所述内容不犯错.....	133
8.3 面试技巧.....	133
8.3.1 六个常见问题.....	133
8.3.2 两个注意事项.....	134
8.3.3 自我介绍.....	134

第 1 章 核心技术

1.1 Html5

1.1.1 Html5 新特性

1) 语义化标签

新的语义化元素：main、footer、header、nav、section

语义化理解： 根据页面内容的结构，选择合适的 HTML 标签

语义化标签优点：

- a. 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；
- b. 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。

前端能做的 seo：

\1.使用语义化标签

\1.meta name=discription

\1.img 标签 alt 属性能够被搜索引擎爬取

\1.代码要规范，工整

```
<header>我是头部</header>
<main>
<nav>我是导航</nav>
<section>
我是内容区域
</section>
<footer>我是底部</footer>
</main>
```

2) 新的表单控件: date、time、email、url、search

正则表达式:

^: 以指定内容开头

\$: 以指定内容结尾

.: 任意字符 a-zA-Z0-9 _ -

*: {0,} // 最少 0 位 最多不限

+: {1,} // 最少 1 位 最多不限

?: {0, 1} // 最少 0 位 最多 1 位

指定个数范围: {start, end} 例如: {1,9} // 最少 1 位 最多 9 位

/d: 0-9

\: 进行转义

要会使用正则来写:

手机号,

邮箱 [要求的类型+个数]

```
<form action="">
<!-- 年月日 -->
<input type="date">
<!-- 时间 -->
<input type="time">
<!-- 邮箱 -->
<input type="email" pattern="^[a-zA-Z_-]+@[a-zA-Z_-]{2,6}" placeholder
value="提交邮箱地址">
<!-- 手机号 -->
<input type="tel" pattern="^[1]{1}[3-9]{1}[0-9]{9}" placeholder="请输入手机
<input type="tel" pattern="^[1]{1}[3-9]{1}\d{9}$" placeholder="请输入手机号
<!-- URL 地址 -->
<input type="url">
<!-- 搜索 -->
<input type="search">
</form>
```

3) 新的 API:

a. 音视频: audio , video 元素

b. 绘图图形: canvas 元素

实现步骤:

// 1. 获取 canvas 元素

```
let canvas = document.querySelector('#canvas')
```

// 2. 获取画笔

```
let canvasCtx = canvas.getContext('2d')
```

// 3. 开始画

```
canvasCtx.fillStyle = 'yellowgreen'; // 元素的颜色
```

```
1 canvasCtx.fillRect(0, 0, 50, 50) // 元素在画布的位置及元素宽高 // top:0
```

```
left:0 width:50 height:50
```

canvas 画布尺寸默认是 300px*150px 如果需要放大或缩小需按照这 2:1 的比例, 画布的放

大或缩小, 元素会随之缩放【但元素也会因此变糊, 发虚】

```
<script>
// 1. 获取 canvas 元素
let canvas = document.querySelector('#canvas')
// 2. 获取画笔
let canvasCtx = canvas.getContext('2d')
// 3. 开始画画
function draw() {
  canvasCtx.fillStyle = 'red';
  canvasCtx.fillRect(0, 0, 50, 50)
  canvasCtx.fillStyle = 'yellowgreen';
  canvasCtx.fillRect(50, 50, 50, 50)
}
draw();
</script>
```

绘制数据可视化图表: echarts [echarts 底层是 canvas]

4) 本地存储: localStorage, sessionStorage、indexedDB

H5:

1. localStorage

- 生命周期: 永久存储(存储在硬盘)。除非人为删除, 数据才会被销毁
- 大小: 5M

2. sessionStorage

- 生命周期: 会话存储(存储在浏览器内存)。会话结束, 内存释放, 数据随之销毁
- 大小: 5M

身份验证:

1、cookie

- 生命周期:
- 持久化 cookie: 存储在硬盘
- 会话/临时 cookie: 存储在内存
- 产生及保存位置:
- 由服务器端产生
- 保存在浏览器端
- 大小: 4kb

2 — 过期时间: 一般 7 天

- 缺点:
- 携带在请求头中, 容易被截获,
- 高级浏览器可以支持用户主动禁用 cookie[浏览器也可以禁用 js]
- 产生及保存位置
- 由服务器端产生
- 保存在服务器端
- 缺点:
- sessionId 需要使用 cookie 作为载体携带的
- 完美的继承了 cookie 的缺点

3、token

- 通过指定的算法对原数据进行加密, 生成一个唯一的字段
- 相对于 cookie 来说安全
- 保存在浏览器本地
- 参与加密的密钥:
- 保存在服务器端, 本地, 不会明文出现在请求数据中
- 但如果被知道了密钥 就会被解谜 可以使用 md5 加密 只能加密 无法破解(只能暴力)


5) 多线程操作: Web Worker

➤ 请思考: js 为什么设计为单线程

js为什么是单线程的

“这主要和js的用途有关,js是作为浏览器的脚本语言,主要是实现用户与浏览器的交互,以及操作dom;这决定了它只能是单线程,否则会带来很复杂的同步问题。举个例子:如果js被设计了多线程,如果有一个线程要修改一个dom元素,另一个线程要删除这个dom元素,此时浏览器就会一脸茫然,不知所措。所以,为了避免复杂性,从一诞生,JavaScript就是单线程,这已经成了这门语言的核心特征,将来也不会改变。” [更多>](#)

1、js 特点

- 单线程(主线程), js 所有的代码最终都会在主线程上执行
- 为什么这样设计?
- js 可以操作 DOM(增删改), 为了保证页面渲染的安全性
- 特点:
- 所有的代码都在一个线程上执行, 就会阻塞现象, 影响用户体验
- 阻塞现象举例:  执行运算量比较大的任务, 这个任务不执行完, 后面的代码都将

2、web Worker:

1. 理解

- 独立的线程, 可以和 js 线程同时运行

2. 特点:

- 完全受控于 js 主线程
- 不能操作 DOM
- 不能使用 window 对象的方法

3. 作用:

- 可以执行运算量比较大的任务, 不会阻塞 js 主线程任务的执行
- 提高页面的渲染效率, 减少阻塞的时间, 提高用户体验

4. 语法:

- Worker 构造函数

➤ 请注意:

测试的时候不能在本地直接打开页面, 因为浏览器(chrome, edeg[IE])有

安全策略[火狐浏览器没有], 需要部署到服务器上, 或者使用 `vscode` || `webstorm` ||

HbuilderX 等自带服务器的开发工具打开[vscode 用 Open with live Server 打开]

web Worker 测试练习:

需要 2 个文件: 一个是 js 线程: 06. Web Worker-使用 worker.html, 一个是 Worker 独立线程:

// 06. Web Worker-使用 worker.html

```
<script>
```

```
let arr = new Array(10000000);
```

```
// 1. 获取 worker 的实例: 语法: new Worker(独立线程的相对路径)
```

```
let myWorker = new Worker('./worker.js');
```

```
// 2. 向 worker 线程发送数据: 语法: Worker 独立线程的实例.postMessage(要发送的数
```

```
myWorker.postMessage(arr);
```

```
// 5. 接收 worker 线程发送过来的数据
```

```
myWorker.onmessage = function(result){
```

```
console.log(result.data);
```

```
}
```

```
</script>
```

```
// worker.js
```

```
// 3、接收 js 主线程发送过来的数据: 语法: 当前分线程事件总线对象[可省略不写].onmessage
```

```
// 当前分线程事件总线对象.onmessage => onmessage
```

```
onmessage = function (result) {
```

```
console.log(result.data);
```

```
// alert(123123); // 不能使用 window 对象的方法
```

```
let arr = result.data;
```

```
// 进行运算量比较大的计算
```

```
for (var i = 0; i < arr.length; i++) {
```

```
arr[i] = i + 1;
```

```
}
```

```
// 4、 将计算之后的数据发送给 js 主线程: 语法: postMessage(计算好的数据)
```

```
postMessage(arr)
```

```
}
```

1.1.2 常用元素分类及特点

1) 行内元素(内联元素)

1. 常见:

a、span、sub、sup、br、strong、b、em、i、label

2. 设置方式: `display: inline`

3. 特点:

☐ 一行内可以存在多个;

☐ 无法设置 `width`、`height`、`padding`、`margin` 值不能设置垂直方向, 只可以设置水平方向, 可以设置 `line-height`;

☐ 一个行内元素内可以包括行内元素和文本内容, `a` 标签特殊: 可以放块级元素、行内块元素, 但不能再放一个 `a` 标签;

☐ 宽度默认随文本内容变化。【宽度默认由内容撑开】

2) 块级元素

1. 常见标签有: `div`、`ul`、`dl`、`ol`、`li`、`table`、`h1-h6`、`p`、`form`、`hr`

2. 设置方式: `display: block`

3. 特点:

☐ 一个块级元素占据一行;

☐ 可以设置 `width`、`height`、`padding` 以及 `margin` 值;

☐ 块级元素可以包含块级元素、行内元素以及行内块元素, 文本类型块级元素特殊: 如 `h1-h6`、`p` 标签, 只能包含文本;

☐ 宽度默认为父级元素宽度。

3) 行内块元素

1. 常见标签有: `img`、`input`、`td`

2. 设置方式: `display: inline-block`

3. 特点:

② 一行可存在多个行内块元素，但它们之间存在空隙

② 可以设置 width、height、padding 以及 margin 值

② 宽度默认随文本内容变化【宽度默认由内容撑开】

1.1.3 如何解决 inline & inline-block 的空白间隙问题？

方法 1、父元素中设置 font-size: 0，但存在一个缺点，若子元素需要字体大小，则还需重新设置

方法 2、父元素中设置 word-spacing 单词间距【注意要大于空白间隙的值】

方法 3【不推荐】、手动去除标签之间的换行以及空格【及将标签写在一行】，会导致代码

不美观，且可能会出现格式化代码后，标签又会换行。

```
<span>1111</span><span>2222</span><span>3333</span>
```

方法 3【不推荐】、子元素中设置 margin 为负值(不推荐使用，不好把控)

方法 3【不推荐】、letter-spacing 字符间距，但这个会导致重叠现象出现，个人使用无法达

到预期效果(不推荐使用，不好把控)

1.1.4 iframe

1) 概念

iframe 标签规定了一个内联框架，可以在当前的文档（页面）中嵌入另外一个文档（页面）。本质是在父页面中放置另一个网页。

同源情况下：父页面可以访问子页面所有内容

不同源情况下：父页面不能访问子页面的内容② 【

判断是否同源：协议、域名、端口号全部相同就属于是同源，有一个不一样都是

不同源】

2) 基本使用

子 iframe 的引入：

```
<iframe src="http://localhost:8000/2.html"></iframe>
```

3) 实际应用情景

② A 系统接入 B 系统（前端微应用）

② 当前页面引入已经写好的页面功能，比如：视频/客服等功能

4) 如何获取 `iframe` 内部的内容 (DOM 元素、方法等)

父页面中获取子页面的全局对象 `window` 语法：子 `iframe.contentWindow`

子页面中获取父页面全局对象 `window` 语法： `window.parent`

`iframe` 标签可以理解为 异步解析 【父页面的 `iframe` 标签加载的时候，需要加载对应子

`iframe`】的，所以在当前页面执行 `script` 标签内容时，此时 子 `iframe` 还未渲染成功，也就获

取不到其中的 DOM 元素，打印结果是 `null`。

必须等 子 `iframe.contentWindow.onload` 事件触发， `iframe` 内部的元素全部加载完成，才能

获取 DOM 元素。

注意： 线下测试的时候，同 `Web Worker` 一样不能在本地直接访问页面

a.html 页面【父页面】

```
<h1>这是父页面</h1>
<iframe id="iframe" src="./10. iframe-b.html" frameborder="0"></iframe> //
<script>
let aIframe = document.querySelector('#iframe');
aIframe.contentWindow.onload = function(){
// 在父页面直接获取子 iframe 的元素： 以下是 2 种写法，都能获取到
console.log(aIframe.contentWindow.document.querySelector('.list'))
console.log(aIframe.contentDocument.querySelector('.list')); // u
}
</script>
```

b.html 页面【子页面】

```
<h2>这是子页面</h2>
<ul class="list">
<li>111</li>
<li>222</li>
<li>333</li>
</ul>
```

同源的情况下，父子页面如何通信

父传子:

a.html 页面【父页面】

父向子传递数据的语法: 子 `iframe.contentWindow.postMessage('要传递的数据', '子 iframe 的 IP 地址')`

```
<h1>这是父页面</h1>
<button id="btn">点击父传子</button>
<hr>
<iframe id="iframe" src="/10. iframe-b.html" frameborder="0"></iframe>
<script>
let aIframe = document.querySelector('#iframe');
let btn = document.querySelector('#btn');
btn.onclick = function(){
aIframe.contentWindow.postMessage('123', 'http://127.0.0.1:5500') // 子
}
</script>
```

b.html 页面【子页面】

子接收父传递的数据的语法: `window.addEventListener('message', 回调)`

```
<script>
// 接收父页面传递过来的数据
window.addEventListener('message', function(result){
console.log('子 iframe 接收到父页面传递的数据', result.data); // 数据在
})
</script>
```

子传父

a.html 页面【父页面】

父接收子传递的数据的语法: `window.addEventListener('message', 回调)`

```
<script>
// 父页面绑定事件
window.addEventListener('message', function(result){
console.log('父页面接收到子 iframe 传递的数据', result.data);
})
</script>
```

b.html 页面【子页面】

子向父传递数据的语法: `window.parent.postMessage('要传递的数据', '父页面的 IP 地址')`

```
1 <h2>这是子页面</h2>
  <button id="btn2">点击进行子传父</button>
  <ul class="list">
    <li>111</li>
    <li>222</li>6 <li>333</li>
  </ul>
  <script>
    let btn2 = document.querySelector('#btn2');
    btn2.onclick = function(){
      // window.parent: 获取父页面全局对象 window
      window.parent.postMessage('aaaaa', 'http://127.0.0.1:5500/')
    }
  </script>
```

不同源的情况下，父子页面如何通信

父传子:

a.html 页面【父页面】

父向子传递数据的语法: 子 `iframe.contentWindow.postMessage('要传递的数据', '子 iframe 的 IP 地址')`

```
<h1>这是父页面</h1>
<button id="btn">点击父传子</button>
<hr>
<iframe id="iframe2" src="http://127.0.0.1:5501/10. iframe-b.html" frameborder="1">
</iframe>
<script>
  let iframe2 = document.querySelector('#iframe2');
  let btn = document.querySelector('#btn');
  btn.onclick = function(){
    aIframe.contentWindow.postMessage('123', 'http://127.0.0.1:5501') //
  }
</script>
```

b.html 页面【子页面】

子接收父传递的数据的语法: `window.addEventListener('message',回调)`

```
1 <script>
// 接收父页面传递过来的数据

window.addEventListener('message', function(result){
  console.log('子 iframe 接收到父页面传递的数据', result.data); // 数据在
})
</script>
```

子传父

a.html 页面【父页面】

父接收子传递的数据的语法: `window.addEventListener('message',回调)`

```
<script>
// 父页面绑定事件

window.addEventListener('message', function(result){4 console.log('父页面接收到子 iframe 传
递的数据', result.data);
})
</script>
```

b.html 页面【子页面】

子向父传递数据的语法: `window.parent.postMessage('要传递的数据', '父页面的 IP 地址')`

```
1 <h2>这是子页面</h2>

<button id="btn2">点击进行子传父</button>

<ul class="list">
<li>111</li>
<li>222</li>
<li>333</li>
</ul>

<script>
let btn2 = document.querySelector('#btn2');
btn2.onclick = function(){
  // window.parent: 获取父页面全局对象 window
  window.parent.postMessage('aaaaa', 'http://127.0.0.1:5500/')
}
</script>
```

1.1.5 说说前台数据存储：

存储方式：localStorage, sessionStorage, cookie

H5:

1. localStorage

- 生命周期：

永久存储(存储在硬盘)。除非人为删除，数据才会被销毁

- 大小：5M

2. sessionStorage

- 生命周期：

会话存储(存储在内存)。会话结束，内存释放，数据随之销毁

- 大小：5M

身份验证：

1、cookie

- 生命周期：

- 持久化 cookie：存储在硬盘

- 会话 cookie：存储在内存

- 产生及保存位置：

- 由服务器端产生

- 保存在浏览器端

- 大小：4kb

- 缺点：

- 携带在请求头中，容易被截获，

- 高级浏览器可以支持用户主动禁用 cookie

2、session

- 产生及保存位置
- 由服务器端产生
- 保存在服务器端
- 缺点：
 - sessionid 需要使用 cookie 作为载体携带的
 - 完美的继承了 cookie 的缺点

3、token

- 通过指定的算法对原数据进行加密，生成一个唯一的字段
- 保存在浏览器本地
- 参与加密的密钥：
 - 保存在服务器端，本地，不会明文出现在请求数据中

1.1.6 谈谈 websocket 的理解

WebSocket 是 HTML5 新增的一种关于浏览器与服务器进行全双工通讯的应用层协议。

基于 TCP 传输协议，浏览器和服务器完成一次握手后，两者之间的连接是持久性的，可以进行双向数据传输。

之前浏览器想要获取服务端的处理进度，☐ 要么使用 ajax 轮询，☐ 要么采用长时间占用连接的方式，给服务器带来压力。

WebSocket 出现后可以解决同步延迟问题，并且进行双向通信，实时性更强，可以发送文本、

也可以发送二进制数据，数据属于轻量级，性能高，通信高效，没有同源限制。

可以与任意服务器进行通信，与 HTTP 协议具有良好的兼容性，因此握手时不容易屏蔽。

默认端口是 80 和 443，协议的标识符是 ws，如果加密是 wss。

通过 send 方法可以发送消息给 WebSocket 服务器，通过 message 方法可以接收消息。

应用场景有：

1. 弹幕
2. 媒体标签（音视频数据传输、即时聊天）
3. 协同编辑
4. 位置数据更新
5. 体育实况更新
6. 股票基金报价更新
7. 大屏展示实时传递数据等

1.2 Css3

1.2.1 新增特性

(1) 新增了伪元素选择器

- a. `:last-child` 匹配父元素的最后一个子元素
- b. `:nth-child(n)` 匹配父元素的第 n 个子元素

扩展：伪类与伪元素的区别：伪元素的目标是找元素，而伪类的目标不是找元素，

是找元素最前或最后的一个位置。【伪类: `::before` `::after`】

(2) 边框特性

- a. `border-radius` 圆角

(3) 颜色与不透明度

- a. `opacity: 0.5;`
- b. `color: rgba(0, 0, 0, 0.5)`

(4) 阴影

a. `text-shadow` 文字阴影

b. `box-shadow` 盒子阴影

(5) transform 变形 a. `transform: rotate(9deg)` 旋转

b. `transform: scale(0.5)` 缩放

c. `transform: translate(100px, 100px)` 位移

(6) 过渡与动画

a. `transition` 过渡

i. `animation` 动画【from..to..:只能有 2 个关键帧 / 0% 100%: 可以有多个关键帧】

(7) 媒体查询

a. `@media` 用来做【响应式】布局 地址:<https://developer.mozilla.org/zh-CN/docs/Web/CSS/@media>

b. 百分比布局【自适应】

i. 百分比是相对于 包含块 的计量单位, 通过对属性设置百分比来适应不同的屏幕

c. rem 布局

i. rem (font size of the root element) 是指相对于根元素的字体大小的单位, rem 只是一个相对单位

1.2.2 盒模型

1) 概念

页面渲染时, DOM 元素所采用的布局模型。 可通过 `box-sizing` 进行设置。

2) 分类

(1) 标准盒模型(W3C)

a. `box-sizing: content-box;`

b. 当给元素设置 `width` 和 `height` 时, 只会改变 `width + height`

(2) 怪异盒模型(IE)

a. `box-sizing: border-box;`

b. 当给元素设置 `width` 和 `height` 时, 会改变 `width + height + padding`

1.2.3 BFC

1) 概念

BFC, 又称为块级格式化上下文, 指的是: 一个独立的渲染区域, 让处于 **BFC** 内部的元素与外部的元素相互隔离, 使内外元素的定位不会相互影响。

2) 触发条件(开启 BFC)

设置浮动: 不包括 `none`

❑ 设置定位: `absolute` 或者 `fixed`

❑ 行内块显示模式: `inline-block`❑ 设置 `overflow`, 即 `hidden`, `auto`, `scroll`

❑ 表格单元格, `table-cell`

3) BFC 特点

❑ BFC 是一个块级元素, 块级元素在垂直方向上依次排列。

❑ BFC 是一个独立的容器, 内部元素不会影响容器外部的元素。

❑ 属于同一个 BFC 的两个盒子, 外边距 `margin` 会发生重叠, 并且取最大外边距。

❑ 计算 BFC 高度时, 浮动子元素也要参与计算。

4) 应用场景

❑ 阻止 `margin` 重叠 【垂直排列的两个盒子, 上面的盒子设置 `margin-bottom:100px`; 下面的盒子设置 `margin-top:200px`; 此时 `margin` 值会重叠, 只保留 200 这个大的外边距;

解决方案:下面的盒子加一个父容器, 添加 `border` 或者 `padding` 样式, 即可解决。】

❑ 包含内部浮动:

清除浮动【overflow:hidden;】，防止高度塌陷

❑ 排除外部浮动：阻止标准流元素被浮动元素覆盖

1.2.4 选择权权重 & 优先级

❑ !important > 行内样式 > #id > .class > 标签选择器 > * > 继承 > 默认

❑ CSS 选择器浏览器是 从右往左 依次解析

1.2.5 CSS 预处理器(Sass/Less/Stylus)

1) 概念

❑ CSS 预处理器定义了一种新的语言，主要是通过用一种专门的编程语言，为 CSS 添加一些编程特性，再编译生成 CSS 文件。

❑ 它可以帮助我们编写可维护的、与时俱进的代码，也可以减少需要编写的 CSS 数量，对于那些需要大量样式表和样式规则的大型用户界面是非常有帮助的。

❑ CSS 预处理器可以更方便的维护和管理 CSS 代码，让整个网页变得更加灵活可变。

2) 语法示例

使用变量，常量

```
$primary-color: pink;
.title {
  color: $primary-color;
}
```

层级嵌套

```
.parent {
  color: red;
  .child {
    color: red;
  }
}
```

1

1 混入

/* 定义混合 */

```
@mixin clearfix {
  &:after {
```

```
content: ".";
display: block;
height: 0;
clear: both;
visibility: hidden;
}
}
/* 使用混合 */
.content{
@include clearfix;
}
继承
.border {
border: 1px solid pink;
}
.content {
@extend .border;
font-size: 20px;
}
```

1.2.6 flex(伸缩盒模型)布局

开启伸缩盒模型： `display: flex;`

1) 概念

Flex 是 Flexible Box 的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。

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：主轴和交叉轴（也叫做侧轴）。默认水平方向的为主轴，垂直方向为侧轴。

2) 容器的属性

🔖 **flex-direction** 定义主轴的方向

🔖 **flex-wrap** 定义是否换行

🔖 **flex-flow** 是 **flex-direction** 属性和 **flex-wrap** 属性的简写形式

🔖 **justify-content** 定义项目在主轴上的对齐方式

🔖 **align-items** 定义项目在侧轴上的对齐方式

3) 项目（伸缩个体的属性）

order 定义项目的排列顺序。数值越小，排列越靠前，默认为 0。

flex-grow 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** 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** 定义了再分配多余空间之前，项目占据的主轴空间。它的默认值为

auto，即项目的本来大小。

🔖 **flex** 是 **flex-grow**, **flex-shrink** 和 **flex-basis** 的简写，默认值为 0 1 auto。

🔖 **align-self** 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。

4) 请回答：flex: 1 代表什么？

🔖 **flex-grow: 1** 如果存在剩余空间，该项目会放大。

🔖 **flex-shrink: 1** 如果剩余空间不足，该项目会缩小。

🔖 **flex-basis: 0%** 设置为 0% 之后，即不占据主轴空间，但是因为存在 **flex-grow** 和

flex-shrink 的设置，该项目会自动放大或缩小。

1.2.7 经典布局实现两栏布局（左侧固定 + 右侧自适应布局）

DOM 结构

```
<div class="container">
  <div class="left">左侧</div>
```

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

css 样式(flex 实现)

父组件 **display:flex**; 开启伸缩盒模型，右侧容器: **flex:1**;

```
container {
  display: flex;
  height: 100px;
}
.left {
  width: 200px;
  height: 100%;
  background: pink;
}
.right {
  flex: 1;
  height: 100%;
  background: deeppink;
}
```

css 样式(float 实现)

左侧容器: **float:left**; 右侧容器: **margin-left**: 左侧容器的宽度;

```
.container {
width: 100%;
height: 100px;
}
.left {
float: left;
width: 200px;
height: 100%;
background: pink;
}
.right {
height: 100%;
background: red;
margin-left: 200px; // 200px 是左边元素的宽度
}
```

1.2.8 隐藏页面元素方式

- ❏ `display: none` 不占位。不会响应 DOM 事件。
- ❏ `opacity: 0` 占位，但不可见。会响应 DOM 事件。
- ❏ `visibility: hidden` 占位，但不可见。不会响应 DOM 事件。
- ❏ `position: absolute; left: -10000px` 移动到屏幕外
- ❏ `z-index: -1` 将别的定位元素遮盖掉当前元素

1.2.9 让元素垂直水平居中方式

利用定位 + transform, 子元素未知宽高

```
father {  
  position: relative;  
}  
.son {  
  position: absolute;  
  left: 50%;  
  top: 50%;  
  transform: translate(-50%, -50%);  
}
```

利用定位 + margin(负值), 子元素必须明确宽高

```
father {  
  position: relative;  
}  
.son {  
  position: absolute;  
  left: 50%;  
  top: 50%;margin-left: -100px; // 子元素宽度的一半  
  margin-top: -100px; // 子元素高度的一半  
  width: 200px;  
  height: 200px;  
}
```

利用定位 + margin:auto;

```
.father {
position: relative;
}
.son {
position: absolute;
left: 0;
right: 0;
top: 0;
bottom: 0;
margin: auto;
}
```

利用 flex

```
.father {
display: flex;
justify-content: center; // 主轴居中
align-items: center; // 侧轴居中
}
```

1.2.10 使用 css 实现一个三角形

```
<div class="triangle"></div>
```

方案一:

```
.triangle {
width: 0;
height: 0;
border: 100px solid pink;
border-left: 100px solid transparent;
border-right: 100px solid transparent;
border-bottom: 100px solid transparent;
}
```

方案二:

```
.triangle {
width: 0;
height: 0;
```

```
border-top: 50px solid skyblue;
border-right: 50px solid transparent;
border-left: 50px solid transparent;
}
```

1.2.11 盒子垂直水平居中

方案一：利用绝对定位 + transform, 子元素未知宽高

```
.father {
position: relative;
width: 200px;
height: 200px;
background: skyblue;
}
.son {
position: absolute;
left: 50%;
top: 50%;
transform: translate(-50%, -50%);
width: 100px;
height: 100px;
background-color: pink;
}
```

方案二：利用绝对定位 + margin(负值), 子元素必须明确宽高

```
.father {
position: relative;
width: 200px;
height: 200px;
background: skyblue;
}
.son {
position: absolute;
left: 50%;
top: 50%;
margin-left: -50px; // 子元素宽度的一半
margin-top: -50px; // 子元素高度的一半
width: 100px;
```



```
height: 100px;
background-color: pink;
}
```

方案三：利用 flex

```
.father {
display: flex;
justify-content: center;
align-items: center;
width: 200px;
height: 200px;
background: skyblue;
}
.son {
width: 100px;
height: 100px;
background-color: pink;
}
```

方案四：

```
.father {
position: relative;
width: 200px;
height: 200px;
background: skyblue;
}
.son {
position: absolute;
top: 0;
left: 0;
right: 0;
bottom: 0;
margin: auto;
width: 100px;
height: 100px;
background-color: pink;
}
```

1.3 Javascript

1.3.1 JS 中原型的理解

1) 在 js 中有 2 个原型：分别是

`prototype`：是显示原型，是对象中的一个属性，它本身也是一个对象，属于浏览器的标准属性，给程序员使用。存在于函数中；

`__proto__`：是隐式原型，也是对象中的一个属性，它本身也是一个对象，默认指向的是这个隐式原型所在的实例对象对应的构造函数中的显示原型，是浏览器的非标准属性，给浏览器使用的。存在于实例对象中。

2) 产生时机：

显示原型是在函数创建的时候产生的，所以函数也是对象，里面有显示原型和隐式原型。

隐式原型是在 `new` 或者可以说是创建实例对象的时候产生的。

3) 原型的作用：

为了实现数据共享，节省内存空间；

实现继承，其实也是为了数据共享。

4) 继承的方式有：

1、改变原型指向实现继承；

2、借用构造函数（`call/apply`）实现继承；

3、拷贝继承（浅拷贝、深拷贝【只有引用数据类型才有浅拷贝和深拷贝，基本数据类型的话就是赋值】），es6 中有 `extend/super` 可以实现继承

5) 原型链：

原型链指的是隐式原型和显示原型的链式关系【其作用是为了查找对象中的属性或方法】，所以也叫做隐式原型链

6)查找对象上基本属性的流程：

1. 先在对象自身上查找，如果有，直接返回
2. 如果没有，根据__proto__在原型对象上查找，如果有，直接返回
3. 如果没有，根据原型对象的__proto__在原型对象的原型对象上查找，一直找到 Object 原型对象为止
4. 如果找到了就返回，如果还找不到，由于它的__proto__为 null，只能返回 undefined

7)应用：

在我之前做过的 xxx 项目 xxx 模块或功能中，为了实现 xxx 组件和 xxx 组件通信或传递数据，使用

了事件总线，内部的原理就是在 Vue 的原型对象上挂载一个属性，存储 Vue 的实例对象，其他组件实例和 main.js 文件中 new Vue 是继承关系，所以可以直接访问原型对象上挂载的这个属性或添加的这个属性，也就意味着可以调用 Vue 实例中\$on 和\$emit 方法进行事件的绑定和分发，从而实现组件通信。解决路由器的版本 bug，利用重写路由器原型上的 replace 方法和 push 方法。

1.3.2 JS 执行上下文的理解：

1) 执行上下文环境：

分为全局执行上下文和函数执行上下文（也叫局部执行上下文）。

2) 执行上下文步骤：

代码将要执行之前会先出现全局执行上下文环境，内部会有预解析的操作，会设置变量对象 this 指向 window，收集变量、函数的声明；代码继续往后执行，如果遇到函数调用，会产生函数执行上下文，收集变量、函数和函数中的参数的声明；当函数调用结束后，调用栈中最上面的执行上下文先销毁，所有代码执行完毕之后，所有执行上下文会被销毁。

3) 执行上下文是动态创建，动态销毁的。

1.3.3 JS 作用域？

概念：

用于约束(限制)代码执行及访问的区域(范围)。

作用：

隔离变量，避免数据污染。

分类：

全局作用域，

函数作用域（局部作用域）、

块级作用域（es6 中 let const 声明变量所在的区域-->花括号 {} 区域：比如：

if {}、循环 {} 等）。

产生时机：

函数定义完毕，作用域就确定了，所以作用域是静态的。

【明确全局作用域、局部作用域、块级作用域中的变量的使用范围。 let、const 声明的变量是否有提升及是否可使用。】

1.3.4 JS 闭包

概念：

闭包指的是：函数及其捆绑的周边环境状态的引用的组合。形成条件：

函数嵌套函数，内部函数用到了外部函数中的变量，外部函数调用内部函数，此时形成闭包。

闭包产生&使用闭包 & 释放闭包

产生：当内部函数对象创建的时候【同理可以说是执行内部函数定义的时候】

使用：执行内部函数

释放：让内部函数对象成为垃圾对象，断开指向它的所有引用

闭包是个对象：

通过浏览器的调试工具查看，发现 closure 闭包存储于函数内部 scopes 作用域链数组中，且闭包

中有隐式原型，所以可以确定闭包也是个对象。

作用及缺点：

1、闭包实际上是用来延长局部变量的生命周期的，也就是缓存局部变量的数据。但是，局部

变量应该随着函数调用而被销毁，本该销毁释放的数据依然存活，长时间或数据量较大的时

候，没有被释放，会出现内存的泄漏，导致内存溢出，这也是闭包的缺点。

2、让函数外部能间接操作内部的局部变量。

应用场景：

在 xxx 项目 xxx 模块 xxx 功能中：我之前做的删除当前条数据的功能，就使用到了闭包。给删除

按钮绑定点击事件及对应的回调函数，回调函数内部调用相关的 api 接口函数且传入要被删除

的当前条数据的 id，此时就是个闭包。

对官网资料的不同见解：

通过观察闭包，我发现：如果闭包创建，会占用内存，但是如果长时间不用，内存会一直被占

用，不是很合理，所以我个人觉得：闭包比较好的创建时机，应该是在内部函数对象调用的时

候更合理一些，这是我的想法。您有什么不同的见解，请指明。

1.3.5 JS 单线程异步编程语言的理解

JS 单线程原因：

首先，

js 是一门单线程编程语言，这是由他的作用所决定的，

js 是作为浏览器的脚本语言，主

要是实现用户与浏览器的交互，以及操作 DOM；这决定了它只能是单线程的，否则会带来很复杂的同步问题。

事件轮询机制 event loop：按理说 js 中的代码应该都是同步执行的，但是，从实际的情况来看，

js 中是可以做异步操作

的。其原因在于，

js 中有事件轮询机制，其本质是浏览器中有辅助它单线程异步执行的分线程

管理模块，也就是代码从上到下执行的时候，同步执行的过程中遇到了异步操作对应的回调函

数，比如：定时器、mutation observer、promise、ajax、事件等等，就会把这些回调放在对应的

分线程管理模块中进行管理，代码继续向后执行，promise 对应的回调直接会放在待执行队列

中的微队列中，其他的回调也会放在待执行队列的宏队列中，然后 mutation observer 的回调也

会放在微队列中。然后初始化代码执行完毕后，再执行微队列中的所有微任务，然后渲染界

面，再执行宏队列中第一个宏任务，然后执行微队列中所有微任务，然后渲染界面，然后执行

宏队列中的第一个宏任务.....依此继续轮询。

1.3.6 JS 数据类型

JS 原始数据类型 6 个：string number boolean null undefined Object

基本数据类型 7 个：symbol BigInt string number boolean null undefined

数据类型 8 个：Object symbol BigInt string number boolean null undefined

1.3.7 函数传递是值传递还是引用传递？

函数调用时，是将实参拷贝一份赋值给形参

实参可能是基本类型值 ==> 值传递

也可能是引用类型的值(也就是地址值) ==> 引用传递/值传递

```
// 注意下面的代码，准确的说不是将 a 内存的地址赋值给 b，而是将 a 中保存的地址值赋值给 b
var a = {}
var b = a
```

1.3.8 区分执行函数定义与执行函数

执行函数定义：其实就是创建一个函数对象，如果指定了函数名，同时会定义变量并指向这个函数对象

执行函数：其实就是调用一个函数，执行函数内部的语句

必须先执行函数定义（创建一个函数对象），再执行函数（调用函数） ==> 注意：函数定义有可能会提升到最上面执行（预解析）

```
// 定义函数/创建函数有两种方式:方法一：函数声明--声明一个函数；方法二：函数表达式
// 执行函数定义就是创建一个函数对象 创建了 fl
function fl(){
  var num = 10
}
// 定义变量并指向这个函数对象
var ff= fl()
// 执行函数其实就是调用一个函数
fl()
```

1.3.9 说说变量提升与函数提升

变量提升 【关键字：var】

变量声明提升：变量的声明部分被提升到了作用域的最前面执行 ==> 预解析

结果/现象：在变量声明语句前，就可以访问这个变量，只是值为 undefined

函数提升 【关键字：function】

函数声明提升：函数声明语句被提升到了作用域的最前面执行 ==> 预解析

结果/现象：在函数声明语句前，就可以调用这个函数了

扩展说明:

let、const 声明的变量有变量提升，但是不能使用

同名的变量与函数: 内部先提升函数, 变量被忽略 ==> 这样只用赋值一次

1.3.10 如何判断函数中的 this

常规情况下, 函数中的 **this** 取决于执行函数的方式

fn(): 直接调用 ==> **this** 是? window

new fn(): new 调用 ==> **this** 是? 新建的对象

obj.fn(): 通过对象调用 ==> **this** 是? obj

fn.call/apply(obj): 通过函数对象的 call/apply 来调用 ==> **this** 是? obj

特殊情况:

箭头函数 ==> **this** 是? 外部作用域的 this ==> 沿着作用域链去外部找 this

bind(obj)返回的函数 ==> **this** 是? obj

回调函数 它不是我们调用的

定时器/ajax/promise/数组遍历相关方法回调 ==> **this** 是? window

DOM 事件监听回调 ==> 发生事件的 DOM 元素

Vue 控制的回调函数(生命周期/methods/watch/computed) ==> **this** 是? 组件的实例

React 控制的生命周期回调, 事件监听回调 ==> **this** 是? 组件对象 / undefined

1.3.11 如何改变函数中的 this

情况一: 将任意对象指定为当前函数中的 **this**

函数立即调用: call() / apply() 函数后面某个时候调用: bind()

情况二: 将外部的 **this** 指定为当前函数中的 **this**

ES6 之前: 将外部 this 保存为其它名称变量, 当前函数中不使用 this, 而使用这个变量

ES6 之后: 箭头函数

1.3.12 如何判断 js 的数据类型

1) typeof

缺点: 不能准确的判断出目标数据的数据类型。【null 和 array 类型判断错误。】

```
typeof 123; // 'number'
typeof "123"; // 'string'
typeof null; // 'object'
typeof undefined; // 'undefined'
typeof Symbol(); // 'symbol'
typeof true; // 'boolean'
typeof {}; // 'object'
typeof function () {}; // 'function'
typeof []; // 'object'
```

2) instanceof

语法:

```
a instanceof b
```

缺点:

不能准确的判断出目标数据的数据类型。【例如数组, 判断出的结果是又属于 Array

类型, 又属于 Object 类型。】

简单理解: 检查 a 是否是 b 的实例

真正理解: 检查 a 的隐式原型属性 (包含隐式原型上的隐式原型) 是否与 b 的显示

原型属性 指向同一个对象

a 的原型链上是否包含 b 的原型

```
// let arr = [1,2,3]; arr instanceof Object; // true
// arr.隐式原型.隐式原型 <==> Objec.显式原型
({}) instanceof Object; // true
```

```
([]) instanceof Object; // true
null instanceof Object; // false
([]) instanceof Array; // true
({}) instanceof Array; // false
```

3) 如何准确获取指定目标数据的数据类型

```
Object.prototype.toString.call(target).slice(8, -1)
// call() 方法使用一个指定的 this 值和单独给出的一个或多个参数来调用一个函数。
// target: 要被判断数据类型的目标数据
// slice(startIndex,endIndex), 从 0 开始索引, 其中 8 代表从第 8 位 (包含第 8 位) 开始截取 (本例中代表空格后面的位置), -1 表示截取从右往左数的第 1 位 (不包含倒数第 1 位), 所以正好截取到[object String]中的 String。[slice: 左闭右开]
Object.prototype.toString.call(123).slice(8, -1); // 'Number'
Object.prototype.toString.call("123").slice(8, -1); // 'String'
Object.prototype.toString.call(null).slice(8, -1); // 'Null'
Object.prototype.toString.call(undefined).slice(8, -1); // 'Undefined'
Object.prototype.toString.call(Symbol()).slice(8, -1); // 'Symbol'
Object.prototype.toString.call(true).slice(8, -1); // 'Boolean'
Object.prototype.toString.call({}).slice(8, -1); // 'Object'
Object.prototype.toString.call(function () {}).slice(8, -1); // 'Function'
Object.prototype.toString.call([]).slice(8, -1); // 'Array'
Object.prototype.toString.call(new Date()).slice(8, -1); // 'Date'
Object.prototype.toString.call(new RegExp()).slice(8, -1); // 'RegExp'
```

1.3.13 说说面向对象的三大特性

封装:

将可复用的代码用一个结构包装起来, 后面可以反复使用

js 的哪些语法体现了封装性: 函数 ==> 对象 ==> 模块 ==> 组件 ==> 库封装都要有个特点: 不需

要外部看到的必须隐藏起来, 只向外部暴露想让外部使用的功能或数

据

继承:

为什么要有继承？复用代码，从而减少编码

js 中的继承都是基于原型的继承；ES6 的类继承本质也是

(原型链+借用构造函数的组合 / ES6 的类继承)

多态：多种形态

理解：

声明时指定一个类型对象，并调用其方法，

实际使用时可以指定任意子类型对象，运行的方法就是当前子类型对象的方法。

JS 中有多态：

由于 JS 是弱类型语言，在声明时都不用指定类型

在使用时可以指定任意类型的数据 ==> 这已经就是多态的体现了

1.3.14 说说 JS 的垃圾回收机制

在 JS 中对象的释放（回收）是靠浏览器中的垃圾回收器来回收处理的

垃圾回收器

浏览器中有个专门的线程，

它每隔很短的时间就会运行一次

主要工作：判断一个对象是否是垃圾对象，

如果是，

清除其内存数据，并标记内存是空闲状

态。

如何判断对象是垃圾对象呢？

机制 1：引用计数法

机制 2：标记清除法

垃圾回收机制 1：引用计数法

最初级的垃圾收集算法，现在都不用了，把“对象是否不再需要”简化定义为“对象有没有引用指向它”

每个对象内部都标记一下引用它的总个数，

如果引用个数为 0 时，即为垃圾对象

有循环引用问题：如果 2 个对象内部存在相互引用，断开对象的引用后，

它们还不是垃圾对象

垃圾回收机制 2：标记清除法

当前垃圾回收算法的基础：把“对象是否不再需要”简化定义为“对象是否可以获得”

从根对象（也就是 window）开始查找所有引用的对象，

并标记为‘使用中’，没有标记为使用中

的对象就是垃圾对象，没有循环引用问题。

垃圾回收机制的理解：

首先在浏览器中有一个专门的线程，每隔很短的时间就会运行一次，判断一个对象是否是垃圾对象，

如果是，

清除其内存数据，并标记内存是空闲状态。，这种机制叫做垃圾回收机制。

清理垃圾对象的工具叫做垃圾回收器，如果这个对象为 null，说明是垃圾对象。一个数据在内存中可以被访问或者说被使用，表示是可用数据，也叫做可达性，如果是垃圾对象，数据不可用，

叫做不可达。垃圾回收机制中有 2 种算法，分别是引用计数法和标记清除法。引用计数

法：判断这个对象的引用数量是否为 0，如果为 0，表示可以清理，这种算法有一个缺点：出现

循环引用会导致无法清理，出现内存泄露，如果数量比较多，会导致内存溢出。标记清除法：

判断该对象是否可用，或者说能不能获取到这个对象，如果获取不到，表示该对象是垃圾对象，可以被清理。垃圾回收机制其原理：就是定期找出不再用到的内存，然后将其释放，不是实时找出无用内存并释放，其原因是：实时开销太大。标记清除法的流程：垃圾回收器在运行的时候，为内存中的变量添加一个标记，如果是垃圾对象就标记为 0，否则标记为 1，然后从各个对象开始遍历，清理所有标记为 0 的垃圾对象，销毁并且回收占用的内存空间，然后再把所有内存中的对象标记为 0，等待下一轮的回收（回收的过程当中，正在占用和使用的标记为 1），这种方式的优点：实现简单，无非就是打不打标记两种情况，而且采用 0 和 1 二进制的标记，简单。缺点：清除之后，剩余的内存位置不变，导致内存空间不连续，出现了内存碎片，如果再有新对象，会出现内存分配问题，为了存储新对象，需要遍历所有的内存空间，找到大于等于新对象大小的块返回，或者遍历整个空闲列表，找到大于等于新对象的最小块，或者是找到最大的块，切成两部分，一部分和新对象大小一样，这种方式会产生很多小块，再次出现内存碎片化，如果考虑到速度和效率，第一个方式是比较合理的，所以，标记清除法是在清理后新对象分配的时候出现了内存碎片化和分配速度慢两个问题。所以，可以使用标记整理算法（Mark-Compact），有效的解决这个问题。其实就是在标记结束后将活着的对象所占用的内存移动到一端，需要清理的对象占用的内存移动到挨着未清理对象内存的一端。后来，V8 引擎增加了回收策略，采用了分代式垃圾回收，也就是把大、老、长的对象放在老年代区，小、新、短的对象放在新生代区。新生代的容量为 1-8M，新生代区域多次回收存活下来的对象会放在老年代区域中，并且新生代对象是通过 Scavenge 算法进行垃圾回收，内部采用了 Cheney 复制的算法，Cheney 算法将堆内存分为使用区和空闲区，也就是新对象放在使用区，快满的时候就执行一次垃圾清理操作，然后对存活着的对象进行标记，干掉垃圾对象，转换成空闲区，然后空闲区变成使用区，如果一个对象，在新生代区域中，存活的时间比较长，或者一个对象占用或超过空闲区的四分之一后，直接扔进老年代区域，为了不影响内存使用，老年代区域直

接干掉垃圾对象。分代的目的是为了快速清理内存，提升效率，当然，V8 引擎后期采用了并行回收机制，也就是 js 是单线程，垃圾回收如果开始，会阻塞 js 执行，回收结束后，js 才可以继

续执行，叫全停顿，页面出现卡顿，所以 V8 采用了并行回收，也就是用多个垃圾回收器一起回收垃圾，缩短时间，增加回收效果。后来，又做了增量标记优化，也就是多个垃圾回收器同时进行回收，只回收一点，然后切换 js 执行，反复的方式，也增加了三色标记法和惰性清理（可以先不清理，让 js 执行，抽空再清理）再次进行优化。

1.3.15 谈谈强缓存和协商缓存

首先无论是强制缓存还是协商缓存，都是浏览器的缓存方案，就是对之前请求过的文件进行缓存，等到下一次访问的时候，重复使用，节省带宽，提高速度，降低服务器压力。强制缓存：指的浏览器不会向服务器发送请求，直接从本地的内存和硬盘中读取缓存文件，响应状态码为 200，不访问服务器，优先从内存中读取，再从硬盘中读取，最后才是请求网络资源。强缓存是由服务器设置的，浏览器向服务器发送请求，服务器返回资源的时候，会在响应头中嵌入字段

Expires (Http1.0 协议的强制缓存方案) 和 Cache-Control (Http1.1 协议的强制缓存方案)，如果同时设置两个字段，Cache-Control 生效，因为它的权限更高，Expires 里面存的是一个时间戳，就是一个时间标识，Cache-Control 当中可以通过 max-age 设置缓存时间，单位是秒，public 任何人都可以访问的资源，private 只能个人访问，no-cache 强制客户端直接向服务器发送请求，no-store 停止一切缓存。协商缓存：浏览器必须向服务器发送请求，由服务器判断请求的资源是否变化来决定是否读取缓存，响应码 304 走缓存，响应码 200 不走缓存。协商缓存的流程：浏览器向服务器发送请求的时候，服务器会在响应头中嵌入 Etag 或 Last-Modified，浏览器接收资源的时候，会保存这两个字段，如果刷新或者重新请求资源时，会在请求头中携

带 If-None-Match（它的值就是之前保存的 Etag）

和 If-Modified-Since（它的值就是之前保存的

Last-Modified）两个字段发送给服务器，服务器会判断 If-None-Match 和 If-Modified-Since 和服

务器保存的 Etag 和 Last-Modified 字段是否一致，如果一致，返回 304，读缓存；如果不一致，

返回新资源和新的 Etag 和 Last-Modified 字段，响应码 200，Etag 由文件索引节大小、最后修改

时间、计算 hash 值，Last-Modified 文件最后修改的时间。流程：当浏览器打开页面向服务器发

送 Http 请求的时候，先判断是否命中响应式缓存（Expires、Cache-Control），如果没有，直接

向服务器请求新资源，如果有强制缓存，要判断是否过期，如果没有过期，直接从内存和硬盘

中读取，不向服务器发送请求，如果过期，判断是否有协商缓存，依据就是有没有 Etag 和

Last-Modified，如果有，向服务器发送请求，请求头中嵌入 If-None-Match 和 If-Modified-

Since，由服务器决策资源是否变化，没有变化，返回 304，读缓存，变化了，返回新资源，响

应码 200，携带新的 Etag 和 Last-Modified，如果没有协商缓存，访问新资源。

1.3.16 区分内存溢出与内存泄露

内存溢出

运行程序需要分配的内存超过了系统能给你分配的最大剩余内存

抛出内存溢出的错误，程序中断运行

演示代码：

```
const arr = []
for (let index = 0; index < 1000000000; index++) {
  arr[index] = new Array(1000)
}
```

内存泄漏（泄露会导致溢出，【先泄露再溢出】）

理解：当程序中的某个内存数据不再需要使用，

而由于某种原因,

没有被释放

常见情况:

1、意外的全局变量

```
function fn () {a = new Array(100000)}  
fn()
```

2、没有及时清除的定时器

```
this.intervalId = setInterval(() => {}, 1000)  
// clearInterval(this.intervalId)
```

3、没有及时解绑的监听

```
this.$bus.$on('xxx', this.handle)  
// this.$bus.$off('xxx')
```

4、没有及时释放的闭包

1.3.17 区分一下同步和异步

同步：

从上往下按顺序依次执行

只有将一个任务完全执行完后, 才执行后面的

会阻塞后面的代码执行

异步：

启动任务后, 立即向下继续执行, 等同步代码执行完后才执行异步回调

不会阻塞后面的代码执行

异步回调函数即使触发了, 也是要先放入待执行队列, 只有当同步任务或前面的异步任务执行完

才执行。

1.3.18 正则表达式

/	转义符
^	表示以指定字符开头
\$	表示以指定字符结尾
*	匹配最少0个字符
+	匹配最少1个字符
?	匹配0-1个字符
.	匹配除“\n”之外的任何单个字符
\d	匹配0-9任意一个数字字符
\s	匹配任何不可见字符，包括空格、制表符、换页符等等。
\S	匹配任何可见字符。

1.3.19 谈谈对事件冒泡和事件委托的理解

事件冒泡：

指标签或组件之间有嵌套关系，都注册或绑定了相同或类似的事件【例如外部元素绑定了

click，内部的元素绑定了 mousedown】，当内部的标签或组件的事件触发了，外部的标签或组

件的事件也会自动的触发，这个是事件冒泡。事件：

指的是视图层到逻辑层的一种通讯方式，事件委托基于事件冒泡，父级包含许多子级元素，只

需要给父级绑定事件，子级元素就可以触发到父级上绑定的事件。这种方式可以减少事件绑定

的次数，从 n 个变成一个，减少内存的使用，提升效率。即使父级内部再动态添加子级元素，

新添加的子级元素依旧可以触发父级身上的事件，事件也会有对应的响应。

应用场景：

比如我之前做过的 x 项目里面的二或三级分类信息的展示及跳转操作，就利用了事件委托的方式，减少事件绑定次数，提升效率，这也是项目优化的表现。

1.3.20 区分 call 和 bind

共同点：

- 1、都是用来改变函数的 this 对象的指向的。
- 2、第一个参数都是 this 要指向的对象。
- 3、都可以利用后续参数传参。

注意：

call 方法调用一个函数，其具有一个指定的 this 值和分别地提供的参数(参数的列表)。该方法的作用和 apply() 方法类似，只有一个区别，就是 call() 方法接受的是若干个参数的列表，而 apply()方法接受的是一个包含多个参数的数组，bind 方法调用一个具有给定 this 值的函数，以及作为一个数组（或类似数组对象）提供的参数。

注意：

1. call()方法的作用和 apply() 方法类似，区别就是 call()方法接受的是参数列表，而 apply()方法接受的是一个参数数组
2. bind()方法创建一个新的函数，当这个新的函数被调用时，其 this 值为提供的值，其参数列表前几项，置为创建时指定的参数序列

1.3.21 区分函数防抖和节流

进行窗口的 resize、scroll，输入框内容校验等操作时，如果事件处理函数调用的频率无限制，会加重浏览器的负担，导致用户体验非常糟糕，此时我们可以采用 debounce（防抖）和 throttle（节流）的方式来减少调用频率，同时又不影响实际效果。

函数防抖：

函数防抖（debounce）：当持续触发事件时，一定时间段内没有再触发事件，事件处理函数

才会执行一次，如果设定的时间到来之前，又一次触发了事件，就重新开始延时。

如下，持续触发 scroll 事件时，并不执行 handle 函数，当 1000 毫秒内没有触发 scroll 事件时，

才会延时触发 scroll 事件

```
function debounce(fn, wait) {  
  var timeout = null;  
  return function() {  
    if(timeout !== null) clearTimeout(timeout);  
    timeout = setTimeout(fn, wait);  
  }  
} // 处理函数  
function handle() {  
  console.log(Math.random());  
}  
// 滚动事件 window.addEventListener('scroll', debounce(handle, 1000));
```

函数节流（throttle）：

当持续触发事件时，保证一定时间段内只调用一次事件处理函数节

流，通俗解释就比如我们水龙头放水，阀门一打开，水哗哗的往下流，秉着勤俭节约的优良传

统美德，我们要把水龙头关小点，最好是如我们心意按照一定规律在某个时间间隔内一滴一滴

的往下滴。

如下，持续触发 scroll 事件时，并不立即执行 handle 函数，每隔 1000 毫秒才会执行一次

handle 函数

```
var throttle=function(func, delay) {  
  var prev = Date.now();  
  return function() {  
    var context = this;  
    var args = arguments;  
    var now = Date.now();
```

```
if (now - prev >= delay) {  
  func.apply(context, args);  
  prev = Date.now();  
}  
}  
}  
  
function handle() {console.log(Math.random());}  
window.addEventListener('scroll', throttle(handle, 1000));
```

总结：

函数防抖：

将几次操作合并为一此操作进行。原理是维护一个计时器，规定在延迟时间后触发函数，但是在 延迟时间内再次触发的话，就会取消之前的计时器而重新设置。只有最后一次操作能被触发。

函数节流：

使得一定时间内只触发一次函数。原理是通过判断是否到达一定时间来触发函数

区别：

函数节流不管事件触发有多频繁，都会保证在规定时间内一定会执行一次真正的事件处理函

数，

而函数防抖只是在最后一次事件后才触发一次函数。

结合应用场景：

防抖(debounce)：

1. search 搜索关键字，用户在不断输入值时，用防抖来节约请求资源。
2. window 触发 resize 的时候，不断的调整浏览器窗口大小会不断的触发这个事件，用防抖来让其只触发一次

节流(throttle)：

1. 鼠标不断点击触发，mousedown(单位时间内只触发一次)

2. 监听滚动事件, 比如是否滑到底部自动加载更多, 用 `throttle` 来判断

1.3.22 自定义 new

new 操作符具体干了什么呢?

1. 创建一个空对象: 并且 `this` 变量引入该对象, 同时还继承了函数的原型
2. 设置原型链 空对象指向构造函数的原型对象
3. 执行函数体 修改构造函数 `this` 指针指向空对象, 并执行函数体
4. 判断返回值 返回对象就用该对象, 没有的话就创建一个对象

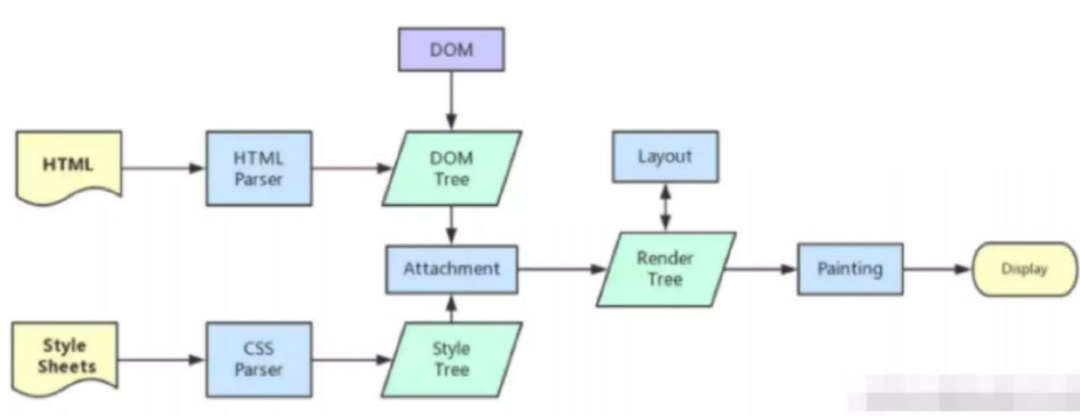
```
<script>
/*
1. 自定义 new 工具函数
语法: newInstance(Fn, ...args)
功能: 创建 Fn 构造函数的实例对象 实现: 创建空对象 obj, 调用 Fn 指定 this 为 obj, 返回 obj
2. 自定义 instanceof 工具函数
语法: myInstanceOf(obj, Type)
功能: 判断 obj 是否是 Type 类型的实例
实现: Type 的原型对象是否是 obj 的原型链上的某个对象, 如果是返回 true, 否则返回 false
*/
</script>
<script src="../../dist/atguigu-utils.js"></script>
<script>
function Person(name, age) {
  this.name = name
  this.age = age
return {}
// return []
// return function (){}
// return 1
// return undefined
}
const p = new Person('tom', 12)
console.log(p)
```

```
const p2 = aUtils.newInstance(Person, 'Jack', 13)
console.log(p2, p2.constructor)
</script>
<script>
  console.log(aUtils.myInstanceOf(p, Object))
console.log(aUtils.myInstanceOf(p, Person))
  console.log(aUtils.myInstanceOf(aUtils.myInstanceOf, Function))
  console.log(aUtils.myInstanceOf(p, String), aUtils.myInstanceOf(p, Funct
</script>
```

1.3.23 说说重排与重绘

页面显示过程：

1. 解析 HTML 生成 DOM 树
2. 解析 CSS 生成 CSSOM 树
3. 解析 JS 更新 DOM 树和 CSSOM 树
4. DOM 树 + CSSOM 树生成渲染树
5. 布局(也称回流, 确定个节点显示的位置)
6. 渲染绘制



更新 DOM 或 Style

1. 可能会导致局部重排(也称回流, 重新布局)
2. 可能会导致局部重绘

注意:

1. 重排肯定会重绘, 但重绘不一定有重排
2. 重排比重绘开销更大, 更消耗性能

哪些操作会导致重排 (大小、位置) :

1. 浏览器窗口尺寸改变
2. 元素位置和尺寸发生改变的时候
3. 新增和删除可见元素
4. 内容发生改变 (文字数量或图片大小等等)
5. 元素字体大小变化。
6. 激活 CSS 伪类 (例如:

:hover) 。

7. 设置 style 属性
8. 查询某些属性。比如说:

offsetTop、offsetLeft、offsetWidth、offsetHeight、scrollTop、

scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight

哪些操作会导致重绘 (css 样式) :

更新元素的部分属性(影响元素的外观, 风格, 而不会影响布局), 比如 visibility、outline、背景色等属性的改变。

如何减少重排次数:

1. 更新节点的样式,
尽量通过类名而不是通过 style 来更新
2. 分离样式的读定操作, 不要将读写操作混合调用

3. 将 DOM 操作离线处理, 比如使用 DocumentFragment

例子代码:

```
var s = document.body.style;
s.padding = "2px"; // 回流+重绘
s.border = "1px solid red"; // 再一次 回流+重绘
s.color = "blue"; // 重绘
s.backgroundColor = "#ccc"; // 重绘
s.fontSize = "14px"; // 再一次 回流+重绘
document.body.appendChild(document.createTextNode('abc!')); // 添加 node, 再一次 回流+重绘
```

1.4 ES6

1.4.1 var, let, const 的区别?

1. const 定义常量, let 和 var 定义变量
2. let 相对于 var
 - a. 有块作用域
 - b. 没有变量提升
 - c. 不会添加到 window 上
 - d. 不能重复声明

1.4.2 谈谈对 promise 的理解?

首先 Promise 是 es6 中新增的更好的异步编程解决方案, 但未真正解决回调地狱。

原生 Ajax 与 Promise 的区别:

1、原生 Ajax:

1. 原生 Ajax 可以实现异步操作, 属于宏任务, 进行 DOM 操作时, 界面渲染的次数会增加, 效率比较低。
2. 【宏任务在界面更新渲染后执行.如果任务中更新了 DOM,会增加一次新的界面渲染】
3. 原生 Ajax 发送请求的时候, 指定函数生效的时机比较固定, 而且有回调地狱的问题。

2、Promise:

1. Promise 可以实现异步操作, 属于微任务, 进行 DOM 操作时, 界面渲染的次数不会增加,

效率比较高。

2. 【微任务在界面更新渲染之前执行,如果任务中更新 DOM,不会增加新的界面渲染】

3. promise 指定函数生效的时机更加灵活, 可以在请求前、请求后、请求完成后指定回调函

数, 虽然也有回调地狱的问题, 但减少了回调地狱的嵌套层数, 属于纯回调的方式。

Promise3 种状态:

pending、resolved/fulfilled 、rejected

Promise2 种变化效果:

pending 可以转变为 resolved

pending 可以转变为 rejected

Promise 的 then 方法:

promise.then()链式调用来解决回调地狱问题,

then()总会返回一个新的 promise, 新 promise 结果

状态由 then 指定的回调函数执行的结果来决定:

1. 抛出错误 => 失败且 reason 就是抛出的错误 2. 返回失败的 promise => 失败且 reason 是返回的

promise 的 reason

3. 返回成功的 promise => 成功且 value 是返回的 promise 的 value

4. 返回非 promise 的任务值 => 成功且 value 是返回的值

5. 返回 promise 中传入了一个空回调, 也就是 pending 状态的 promise => 中断 promise 链

Promise 中 2 个常用方法:

1. all: 接收包含多个 promise 的数组, 当所有的 promise 是成功的, 其结果就是成功的; 有一个 promise 是失败的, 结果就是失败的。
2. race: 接收包含多个 promise 的数组, 结果由第一个完成的 promise 决定。

async 和 await:

1. async 和 await 是 Generator 的语法糖, 可以用来简化 promise 的写法, 是消灭异步编程的终极武器, 真正的解决了回调地狱问题。
2. async 可以没有 await; 但 await 不能没有 async。
3. 调用 async 可以返回一个 promise, 其结果状态由 async 函数体执行的结果决定。
4. await 的右侧也可以不是 promise, 如果不是, 直接返回表达式的值。

1.4.3 谈谈对 ES6 新特性的理解

es6 是 2015 年发布的, 全称是 ECMA2015, 简称 es6。到 2023 年版本更新到 es14, es6 之后平均每年更新一个版本。

es6 中新增了一些特性, 如:

1. let、const: 用来声明变量和常量, 也就意味着出现了 `let` 和 `const` 声明的变量或常量是 `块级作用域` 的, 根据 TC39.ES, 也就是 ECMA262 官方英文文档明确指出: `let` 和 `const` 声明的变量或常量虽然提升但是不能使用。
2. 解构赋值运算符: `...三点扩展运算符`, 可以拆包和打包
3. 形参默认值: 简化函数的形参语法
4. class: 用来定义类的, 可以使用 `extends` 和 `super` 进行继承

5. 导入 import 导出 export 模块化相关的关键字
6. Symbol 数据类型：用来定义唯一的数据标识
7. 数组、对象、字符串都增加了扩展的相关方法
8. promise / async & await: async 和 await 是 Generator 的语法糖，用来简化 Promise 的写法，因为用 iterator 接口机制和 Generator 函数实现异步操作比较麻烦
9. Set 和 Map 容器：Set 用来存储：无序不可重复【自动去重】的多个值的集合体；Map 以键值对的方式存储唯一不可重复【自动去重】的集合体。
10. proxy 对象：用来实现数据的代理和响应式。
11. 箭头函数：定义匿名函数，
this 使用的是外层作用域的 this。
12. 模板字符串：我是\${name}，今年\${age}

1.4.4 谈谈对 TS 的理解

ts 始于 js【底层是 js 实现】终于 js【ts 需要编译为 js 浏览器才能识别】。

js 是一门基于对象的编程语言，而 ts 是一门面向对象的编程语言。

js 是弱类型的编程语言，而 ts 是强类型的编程语言。

js 中声明变量使用 var let const，而 ts 中也可以用，但是变量在声明的时候需要先指定数据类型。

ts 中是可以使用 es6 的。

ts 中常用的基本数据类型：

- a. string
- b. number
- c. boolean

- d. null
- e. undefined
- f. array
- g. object
- h. enum 枚举：在数据值个数固定的情况下，或者供用户或开发者选择数据值的情况下，

可以使用枚举类型

- i. tuple 元组：可以设定数组中存储的数据类型不一致
- j. any 任意类型：缺点：存时容易，取时难
- k. void 表示没有任何类型，当一个函数没有返回值时，就写返回值类型为 void
- l. | 联合类型，表示取值可以为多种类型中的一种。

ts 的类 class：

- 1. 类中的成员是有访问修饰符的：【用来描述类内部的属性/方法的可访问性】
 - a. public：默认值，任何的地方都可以访问；
 - b. private：只能在此类内部访问；
 - c. protected：类内部和子类可以访问。
- 2. 类和类之间可以有继承关系，使用 extends 来实现，被继承的类叫基类（父类），继承的类叫做派生类（子类）。
- 3. 类中的属性成员，可以使用 readonly 进行修饰，表示只读；4. 类中的属性成员，可以使用 static 进行修饰，表示的是静态成员，给类名使用的；
- 5. 类中的属性成员，可以单独的设置 get 和 set 方法，叫存取器，可以对属性值进行具体的读取和设置操作。

ts 的抽象类：

- 1. ts 中如果在一个类，也就是 class 前面使用 abstract 进行修饰，这个类叫做抽象类。

2. 抽象类是不能实例化的【不能 new，不占内存】，抽象类中的成员可以是实例成员。
3. 抽象方法必须在抽象类中【方法名前面使用 abstract 进行修饰】，抽象方法不能有具体的实现，抽象类存在的意义：就是为子类服务的，如果子类也是抽象类，那么父类中的抽象成员是可以不进行实现的。

ts 的函数：

1. ts 中的函数，可以有重载，也可以有重写，但是在 js 中不能有重载。
2. 函数中的参数有可选参数和默认参数，如果这个参数是可选参数，建议放在参数列表的最后。

ts 的接口：

ts 中的接口，表示的是一种规范，或者是一种能力，也是一种约束。

1. 接口可以作为对象的类型使用，规范对象类型定义，限定对象中有哪些属性成员。
2. 接口也可以作为函数的类型使用，规范函数定义，限定函数的参数及返回值。
3. 接口也可以作为类的类型来使用，类和接口之间使用通过 implements 来实现，也就意味着接口在定义的时候里面的方法是没有具体实现的，一旦让某个类实现了这个接口，这个类就需要把接口中定义的方法进行实现，也可以理解为这个类有了接口的这种能力。

一个类可以实现多个接口，使用的是关键字 implements。

接口和接口之间使用的是 extends 来实现继承，接口可以继承多个接口。

ts 的泛型：

在定义函数、接口或者类的时候，不预先指定具体的类型，而是在使用的时候，再去指定类型，这是一种特性，也是一种约束。比如：

1. 函数 `T`，不确定参数及返回值的类型，此时，可以使用泛型来定义这个函数，这种函数叫泛型函数。`T`，传入类型，那么对应的参数和返回值的类型就已经

明确了。

2. 接口 也可以使用泛型，此时叫泛型接口，也叫泛型类，也叫泛型约束。定义的时候通过<>尖括号写法来指定不具体的类型，如 `T K V` 。 ，再去明确具体的类型。

应用场景：

之前在 Vue3 框架做的 xxx 项目，里面 api 接口函数定义的时候，用到了泛型的 axios；父子组件通信的时候，子级组件接收父级组件传递的数据，用到了 `define.props`。

ts 泛型 —— interface 和 type 的区别：

1. type 的作用：

- a. 定义数据类型的别名
- b. 定义联合类型及元组的类型
- c. 描述一个对象或函数的类型
- d. 进行接口或类的继承

2. interface 的作用：

- a. 定义接口
- b. 规范函数、对象、类。

1.4.5 编码实现继承

1) 基于原型链+构造函数的继承

(1) 让子类的原型为父类的实例:

```
Student.prototype = new Person()
```

(2) 让子类的原型对象的构造器为子类:

```
Student.prototype.constructor = Student
```

(3) 借用父类构造函数:

```
Person.call(this, name, age)
// 父类型
function Person(name, age) {
  this.name = name
  this.age = age
}
Person.prototype.fn = function () {}
Person.prototype.sayHello = function () {
  console.log(`我叫${this.name}, 年方${this.age}`)
}
// 子类型
function Student(name, age, price) {
  // this.name = name
  // this.age = age
  // 借用父类型的构造函数
  Person.call(this, name, age) // 相当于执行 this.Person(name, age)
  this.price = price
}
// 让子类的原型为父类的实例
Student.prototype = new Person()
// 让原型对象的构造器为子类型
Student.prototype.constructor = Student
// 重写方法
Student.prototype.sayHello = function () {
  console.log(`我叫${this.name}, 年方${this.age}, 身价: ${this.price}`)
}
const s = new Student('tom', 23, 14000)
s.sayHello()
s.fn()
```

2) 基于 ES6 的 class 类的继承

子类 extends 父类: class Teacher extends Person2

子类构造器中调用 super 父类的构造: super(name, age)

```
1// 父类
class Person2 {
  constructor (name, age) {
    this.name = name
    this.age = age
  }
  fn () {}
  sayHello () {
    console.log(`我叫${this.name}, 年方${this.age}`)
  }
}
// 子类
class Teacher extends Person2 {
  constructor (name, age, course) {
    super(name, age)
    this.course = course
  }
  // 重写父类的方法
  sayHello () {
    console.log(`我叫${this.name}, 年方${this.age}, 课程:${this.course}`)
  }
}
const t = new Teacher('bb', 34, 'CC')
t.sayHello()
t.fn()
```

1.4.6 HTTP 协议的理解

是一个超文本传输协议，也就是浏览器与服务端通信的连接协议。

HTTP 协议是无状态的，对于事物的处理没有记忆能力，服务器不知道客户端是什么状态，也

就是说这次打开的页面和上次打开的页面之间没有任何联系。

HTTP 协议是 TCP 协议和 IP 协议之上的，HTTP 协议属于应用层协议，TCP 属于传输层的协议，

IP 属于网络层的协议。

报文分类：请求报文、响应报文

报文：

报文概念：协议当中通信的内容

报文分为：报文首行、报文头部、空行、报文体

请求报文：

请求报文概念：浏览器发给服务器的内容

请求报文分为：请求首行、请求头、空行，请求体

响应报文：

响应报文概念：服务器返回给浏览器的内容

响应报文分为：响应首行，响应头，空行，响应体

报文由 4 部分组成：报文首行、报文头部、空行、报文体

1. 报文首行中包含请求方式、请求地址、协议名及版本号。
2. 报文头部：请求头 / 响应头
 - a. 请求头中包含 Connection: keep-alive 保持长链接、Content-Type 请求体参数类型、User-Agent 用户代理、Referer 请求来源地址、Cookie 等。
 - b. 响应头中包含：access-control-allow-origin 允许跨域、Content-Type 响应体参数类型、Cache-Control 强制缓存控制、Etag / Last-Modified 协商缓存控制。
3. 空行就是一个换行，用来隔开头和体的。
4. 报文体：请求体 / 响应体
 - a. 请求体中包含请求体数据，请求参数。
 - b. 响应体中包含服务器响应的数据，也是页面渲染所需的真正“干货”。

1.4.7 长链接和短链接的理解

无论是长链接还是短链接，指的都是客户端和服务端进行 HTTP 操作的一种链接方式。

HTTP 协议的长链接和短链接实际上是 TCP 的长链接和短链接。

短链接：

HTTP 协议 1.0 中，默认使用的是短链接。也就是浏览器和服务端进行 HTTP 操作的时候建立一次链接，操作一些任务，任务结束后，就中断链接，如果再有请求操作，再重新建立链接，进行数据传输，完毕后再次关闭。也就是多次请求进行多次链接建立，再多次关闭链接。

1. 操作步骤：

a. 建立链接 --> 数据传输 --> 关闭链接 建立链接 --> 数据传输 --> 关闭链接

2. 优点：

a. 服务端管理简单，存在的链接都是有价值的，不需要额外的控制手段，但是，TCP 的建立和关闭会浪费很多时间和带宽。

3. 使用场景：

a. 客户量比较大，请求又频繁，建议使用短链接。

长链接：

HTTP 协议 1.1 中，默认使用长链接。需要设置 Connection: keep-alive，表示保持长链接，也就是

打开一个页面后，浏览器端和服务端传输 HTTP 数据的时候，TCP 的链接不会关闭，这链接会一直保持，但是，如果长时间保存，对于服务端而言，也是有一定压力，而且链接不断开，客户端容易出现恶意链接，拖垮服务器，所以，服务端一般情况下会采用一些策略，给长链接设置一个保持的时间，超过这个时间就会关闭链接。

1. 操作步骤：

a. 建立链接 --> 数据传输 --> 保持链接 --> 数据传输 --> 关闭链接

2. 优点:

- a. 省去较多的 TCP 建立和关闭的操作, 减少浪费, 节约时间。

3. 使用场景:

- a. 频繁的请求资源适合使用长链接, 需要服务端控制最大的链接数及保持时间。

1.4.8 HTTP1.1 和 HTTP2.0 的区别

1. 二进制格式:

HTTP1.1 采用的是文本或二进制的方式, 头部信息必须是文本, 数据体可以是文本, 也可以是二进制。

HTTP2.0 采用的是二进制协议, 也就是头部信息和数据体都是二进制的,

2. 多路复用:

HTTP1.1 采用的是若干个请求排队串行, 单线程处理, 前面的请求完成后, 后面的请求才可以执行, 一旦某个请求超时, 就会阻塞后面的请求, 也叫线头阻塞。

HTTP2.0 中有多路复用, 仍然复用 TCP 协议链接, 浏览器和服务器都可以同时发送多个请求或响应, 而且不用按照顺序依次发送。

3. 头部压缩:

HTTP1.1 没有状态, 每次请求需要携带所有信息, 请求的很多字段都是重复的, 比如, cookie 和 User Agent, 一样的内容每次都要携带, 浪费带宽, 影响速度。

HTTP2.0 中采用头部压缩机制, 使用 gzip 和 compress 压缩后再发送, 也就是浏览器和服务器同时维护一张头信息表, 所有的字段都存入这个表, 生成一个索引好, 发送的请求中, 只发送索引号, 速度会明显提升。

4. 服务器推送:

HTTP2.0 中允许服务器未经请求，主动向客户端发送资源，也叫服务器推送。可以减少延迟时间，推送的资源是静态的，和 WebSocket 向客户端发送即时数据的推送是不同的。

1.4.9 HTTP 和 HTTPS 的区别

Http 协议：

1. 超文本传输协议
2. 信息是明文传输
3. 默认端口是 80

HTTPS 协议：

1. 由 SSL 和 HTTP 协议构建的，可进行加密传输、身份认证的网络协议
2. 具有安全性的，是 SSL（对称加密）加密传输协议，需要身份认证，比 HTTP 更加安全
3. 需要申请 CA 证书，免费的很少，需要花钱
4. 默认端口是 443

1.4.10 常见的 HTTP 状态码以及代表的意义

5 种常见的 HTTP 状态码及代表的意义：

200 （OK）：请求已成功，请求所希望的响应头或数据体将随此响应返回。

400 （Bad Request）：请求格式错误。

1. 语义有误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求；

2. 请求参数有误。

404 （Not Found）：请求失败，请求所希望得到的资源未被在服务器上发现。

500 （

Internal Server Error）：服务器遇到了一个未曾预料的状态，导致了它无法完成对请求的

处理。

其他：

1XX：信息性状态码，表示接受的请求正在处理

2XX：成功状态码，表示请求正常处理

3XX：重定向状态码，表示需要附加操作来完成请求

4XX：客户端错误状态，表示服务器无法处理请求

5XX：服务器错误状态，表示服务器处理请求出错

1.4.11 AJAX 的理解

原生 ajax 封装一共是四个步骤：

1. 先创建一个 ajax 的核心 XMLHttpRequest
2. 使用 open 创建请求，如果是 post 请求，需要 setRequestHeader 写请求头
3. 需要 onreadystatechange 设置监听事件，内部判断状态码及响应状态码
 - a. 如果状态值不为 4，直接结束（请求还没有结束）
 - b. 如果响应码在 200~299 之间，说明请求都是成功的，否则请求失败
4. 最终通过，就调用 send 发送请求 如果不需要参数就写一个 null

```
<script>
function ajax(url) {
return new Promise((resolve, reject) => {
// 创建一个 XHR 对象 XMLHttpRequest
const xhr = new XMLHttpRequest()
// 初始化一个异步请求(还没发请求) open 创建请求
xhr.open('GET', url, true)
// 绑定状态改变的监听 监听状态码和响应状态码
xhr.onreadystatechange = function () {
/* ajax 引擎得到响应数据后
将 xhr 的 readyState 属性指定为 4
```

```
将响应数据保存在 response / responseText 属性上
调用此回调函数

*/
// 如果状态值不为 4, 直接结束(请求还没有结束)
if (xhr.readyState !== 4) {
  return
}
// 如果响应码在 200~~299 之间, 说明请求都是成功的
if (xhr.status >= 200 && xhr.status < 300) {
  // 指定 promise 成功及结果值
  resolve(JSON.parse(xhr.responseText))
} else {
  // 请求失败了 // 指定 promise 失败及结果值
  reject(new Error('request error status' + request.status))
}
}
// 最终通过, 就调用 send 发送请求 如果不需要参数就写一个 null
xhr.send()
})
}
</script>
```

1.4.12 区分 AJAX 与一般 HTTP 请求

1. ajax 请求是一种比较特殊的 http 请求。
2. 对于服务器而言, ajax 请求和一般的 http 请求没有区别, 区别在浏览器端。在浏览器端如果使用 XHR 或者 fetch 发出的请求才是 ajax 请求, 其他方式都属于非 ajax 请求。
 - a. 一般 HTTP 请求: 浏览器会直接显示响应体数据, 也就是页面刷新或页面跳转;
 - b. ajax 请求: 浏览器不会对界面进行任何的更新操作, 只是调用监听的回调函数并传入响应体数据。【思路扩展: ajax 操作数据宏任务...[自己补充]】

1.4.13 AJAX 跨域

跨域的概念：

所谓的跨域指的是同源策略中协议、域名、端口三者都相同，如果有一个不相同，就是跨域。

1. 普通的 HTTP 协议请求是不存在跨域的。
2. 只有浏览器端的 ajax 请求存在跨域。
3. 浏览器要求当前页面和服务器必须同源，目的是为了安全，服务器与服务器之间没有跨

域，页面中加载图片、线上的 js 或者 css 都不受同源策略限制，浏览器的 script、img、form 等标签都没有跨域。

解决跨域：解决跨域常见的方式有 3 种：

JSONP、CORS、proxy。

1. JSONP 的原理：利用的是 script 发送请求，不受限制。需要在前台 script 的 src 属性中设置目的地址及回调函数。后台需要处理请求，产生返回的数据，读取传入过来的回调函数，包括里面的请求参数，返回执行函数的 js 代码。这种方式只能处理 get 请求，每个请求在后台都要做处理，很麻烦。
2. CORS 原理：后台中返回浏览器在某个域上，发送的跨域请求的相关响应头，需要配置跨域的地址、跨域的请求方式、跨域的请求字段、请求缓存的时间、跨域携带的 Cookie 等相关信息，前台不需要做任何处理，后台需要进行相关处理，但很麻烦。
3. proxy 方式：只需要在对应的脚手架中设置 webpack 的配置项，比如处理地址的开头标识，目标服务器地址，是否允许跨域，路径是否重写等。

代理服务器：帮助我们转发请求的

1. 代理可以分为正向代理和反向代理
2. 正向代理代理的是客户端

3. 反向代理代理的是服务器端
4. 可以利用 nginx 进行反向代理，实现项目上线。

1.4.14 说说项目中的 axios 的二次封装

1. 配置通用的基础路径和超时
2. 显示请求进度条
 - a. 显示进度条：请求拦截器回调
 - b. 结束进度条：响应拦截器回调
3. 成功返回的数据不再是 response，而直接是响应体数据 response.data
4. 统一处理请求错误，具体请求也可以选择处理或不处理
5. 每个请求自动携带 userTempld 的请求头：再请求拦截器中实现
6. 如果当前有 token，自动携带 token 的请求头
7. 对 token 过期的错误进行处理

```
// 代码展示：Vue2 项目：src/utlis/request
import userId from './userId';
//axios 二次封装
import axios from "axios";
//引入进度条插件
import nprogress from 'nprogress';
//引入样式
import 'nprogress/nprogress.css'
import store from '@store';
//关闭右侧加载小球
nprogress.configure({ showSpinner: false });
//利用 axios 对象 create 方法:创建 axios 对象
let request = axios.create({
  baseURL: "/api",//基础路径
  timeout: 5000//超时的设置
});
```



```
//请求拦截器
request.interceptors.request.use((config) => {
  //config:配置对象,请求头 config.headers,携带公共请求参数
  //请求之前进度条开始动
  //请求头携带用户身份(用户身份唯一不变的:随机数、时间戳)
  // config.headers.userId = 100; 这种写法不对,用户身份一样了
  config.headers.userId = getUserId();
  //说明 Vuex 已经获取到 token,用户已经登录成功
  if(store.state.user.token){
    config.headers.token = store.state.user.token;
  }
  nprogress.start();
  return config;
});
//响应拦截器
request.interceptors.response.use((res) => {
  //根据状态码进行其他判断
  let data = res.data;
  //请求成功进度条消失
  nprogress.done();
  //简化数据
  return data;
}, (error) => {
  //返回一个失败的 promise 对象
  return Promise.reject(error)
});
//对外暴露
export default request;
```

1.4.15 说说 axios 的整体执行流程

基本执行顺序:

1. 求拦截器的回调函数
2. xhr 发请求

3. 响应拦截器成功/失败的回调

4. 具体请求成功/失败的回调

内部原理:

通过 `promise.then` 的链式调用将这 4 个任务串连起来, 依次执行并进行数据传递:

```
Promise.resolve(config)
.then((config) => { // 请求拦截器
// 显示进度/携带 token/userTempId
return config
}))
  .then((config) => { // 使用 xhr 发 ajax 请求
return new Promise((resolve, reject) => {
// 根据 config 创建 xhr 对象发送异步 ajax
// 如果请求成功(响应状态码 200--299 之间)
const response = {
data: JSON.parse(xhr.responseText),
status: xhr.statusCode
}
resolve(response)
// 如果请求失败
reject(new Error('请求失败 code ' + xhr.statusCode))
})
})
  .then( // 响应拦截器
response => {
return response.data
},
error => {
throw error
}
)
  .then(result => { // 特定请求的回调
}).catch(error => {})
```

1.4.16 从输入 url 到渲染出页面的整个过程

1. 进行 url 的判断，也就是解析 url 是否合法有效，然后进行缓存判断。
2. DNS 解析将域名解析成 IP 地址。
 - a. 缓存：浏览器的 DNS，计算机的 DNS，路由器的 DNS，网络运营商的 DNS。
 - b. 要有一个递归查询过程。
3. TCP 连接三次握手（
 - a. 浏览器向服务器发送消息，
 - b. 服务器向浏览器发送消息，
 - c. 浏览器确认消息），目的是为了建立连接。
4. 开始发送请求（请求报文发送过去），返回响应（响应报文发送回来），解析并渲染页面（
 - a. 遇到 HTML，通过 HTML 的解析器解析成 DOM 树；
 - b. 遇到 CSS，调用 CSS 解析器解析成 CSSOM 树；
 - c. 遇到 JS，先执行初始化同步代码，再执行微队列中所有微任务，
 - d. 如果遇到要修改的元素，重新调用 HTML 解析器解析成 DOM 树，
 - e. 如果遇到样式修改，重新调用 CSS 解析器解析成新成 CSSOM 树，
 - f. 然后将 DOM 和 CSSOM 进行渲染树的操作，
 - g. 然后设置内容布局及渲染操作，
 - h. 然后执行宏队列中的第一个宏任务）。
5. 断开连接，TCP 四次挥手，断开请求和响应连接各两次。




1.4.17 自定义数组扁平化

```
<script>
/*
数组扁平化: 取出嵌套数组(多维)中的所有元素放到一个新数组(一维)中
如: [1, [3, [2, 4]]] ==> [1, 3, 2, 4]
[1, [2, 3], [4, [5, [6, 7]]]]
*/
/*
方法一: 递归 + reduce() + concat()
[1, 2].concat([3, [4]], {}) ==> [1, 2, 3, [4], {}]
[1, [3, [2, 4]]] => [1] => [1, 3, [2, 4]]
[].concat(1) ==> [1]
*/
function flatten1(arr) {
return arr.reduce((pre, item) => {
// 如果 item 是一个二维数组, 需要先进行递归处理
if (Array.isArray(item) && item.some(cItem => Array.isArray(cItem)))
return pre.concat(flatten1(item))
}
return pre.concat(item)
}, [])
}
// console.log(flatten1([1, [2, 3], [4, [5, [6, 7]]]]))
/*
方法二: while + ... + some() + concat()
[1, [3, [2, [4, 5]]]] ==> [1, 3, [2, [4, 5]]] ==> [1, 3, 2, [4, 5]] =
*/
function flatten2(arr) {
// [1, [3, [2, 4]]] // [1, 3, [2,4]]
let result = [].concat(...arr)
while(result.some(item => Array.isArray(item))) {
result = [].concat(...result)
}
return result
}
console.log(flatten2([1, [2, 3], [4, [5, [6, 7]]]]))

```

```
</script>
```

1.4.18 axios 的理解:

首先它是一个函数，可以直接传入地址发送请求，也可以传入一个配置对象，设置请求方式、地址、params 参数及请求体参数发送请求，也可以直接通过 axios 调用方法（get post delete put 等等）发送请求，用来实现前后端交互。

内部封装了 ajax,

react 或者 Vue 框架中都可以使用 axios 进行异步操作发送请求【低版本的

Vue 框架之前使用的是 Vue-resource，原生的方式也可以使用 fetch（想在服务器端进行异步操作，也可以使用 flyio），小程序中使用的是 wx.request，uni-app 中使用的是 uni.request】。

通常情况下要对 axios 进行二次封装，需封装进度条的显示效果、根路径、超时时间、请求拦截器和响应拦截器。

1. 请求拦截器中：一般会在请求头嵌入 token、临时用户 id、cookie 等。
2. 响应拦截器中：成功的回调，内部返回的是响应的数据 result.data；失败的回调，内部对错误信息进行判断和处理。比如：判断 token 是否过期，然后进行退出登录操作，或者是重置用户信息操作，通过路由跳转到登录页，并且设置提示信息。如果需要外部处理错误信息，需要返回一个 promise.reject 失败的 promise。如果不想把错误信息传递下去，可以中断 promise。

axios 当中有 4 个任务，分别是：

1. 请求拦截器中的回调
2. 发送请求的回调
3. 响应拦截器中成功或失败的回调

4. 请求完毕后的成功或失败的回调

内部使用的是 `promise.then` 链式调用的方式串联这四个任务，所以，请求拦截器中返回的

`config` 对象实际上是一个 `promise`，通过 `promise.then` 串联请求的回调函数，内部仍然返回一个

`promise` 对象，再用 `.then` 去串联响应拦截器中的成功和失败回调，最后，再 `.then` 串联请求完

毕后成功或失败的回调。（自行扩展：`promise` 的理解）

1.4.19 谈谈 webpack

首先它是一个静态模块打包工具，可以将静态模块编译、打包、输出成一个或多个文件。

文件中主要有 5 个核心对象，分别是：

1. `entry`（入口）：表示的是 `webpack` 从哪个文件开始打包。
2. `output`（输出）：`webpack` 编译打包后的文件输出到哪里。`output` 中打包的输出路径建议使用绝对路径，输出的文件名字最好是使用根据内容生成 `hash` 值的方式，也就是内容变化，`hash` 值就变化。
3. `loader`（加载器）：`webpack` 只能识别 `js`、`json` 文件，其他类型的文件需要通过 `loader` 转化成有效的模块才能识别。
4. `plugin`（插件）：可以进行打包的优化、资源管理、注入环境变量等。
5. `mode`（模式）：可以选择生产环境（`production`）或开发环境（`development`）或者 `none` 中的一个。不同模式会加载不同的配置。

常见的 `Loader`：

1. 处理样式资源
 - a. `style-loader` 动态创建一个 `Style` 标签，里面放置 `Webpack` 中 `CSS` 模块内容
 - b. `css-loader` 负责将 `CSS` 文件编译成 `Webpack` 能识别的模块

- c. less-loader 将 Less 文件编译成 CSS 文件
- d. sass-loader 将 Sass 文件编译成 CSS 文件
- e. stylus-loader 将 Stylus 文件编译成 CSS 文件
- f. postcss-loader 根据要求增加 CSS 的前缀(css 代码兼容性处理)

2. 处理 js 资源

- a. babel-loader 将 ES6 转化为 ES5(js 代码兼容性处理)
- b. eslint-loader 过去用来进行 js 代码语法检查, 最新 Webpack5 改用插件实现: eslint-webpack-plugin

3. 加载其他资源

- a. file-loader 过去用来转化图片、字体图标等资源, 现在 Webpack5 内置了
- b. url-loader 过去用来转化图片等资源, 现在 Webpack5 内置了
- c. Vue-loader 用来编译 Vue 单文件组件

常见的 Plugin:

- 1. html-webpack-plugin 简化创建 HTML 文件
- 2. eslint-webpack-plugin 用来进行 js 代码语法检查
- 3. terser-webpack-plugin 压缩 js 代码
- 4. mini-css-extract-plugin 提取 CSS 成单独文件
- 5. css-minimizer-webpack-plugin 压缩 CSS 代码

Webpack 优化相关的:

- 1. 提升开发体验
 - a. 使用 Source Map 让开发或上线时代码报错能有更加准确的错误提示。
- 2. 提升 webpack 提升打包构建速度

a. 使用 HotModuleReplacement 让开发时只重新编译打包更新变化了的代码，不变的代码使用缓存，从而使更新速度更快。

b. 使用 OneOf 让资源文件一旦被某个 loader 处理了，就不会继续遍历了，打包速度更快。c. 使用 Include/Exclude 排除或只检测某些文件，处理的文件更少，速度更快。

d. 使用 Cache 对 eslint 和 babel 处理的结果进行缓存，让第二次打包速度更快。

e. 使用 Thead 多进程处理 eslint 和 babel 任务，速度更快。（需要注意的是，进程启动通信都有开销的，要在比较多代码处理时使用才有效果）

3. 减少代码体积

a. 使用 Tree Shaking 剔除了没有使用的多余代码，让代码体积更小。

b. 使用 @babel/plugin-transform-runtime 插件对 babel 进行处理，让辅助代码从中引入，而不是每个文件都生成辅助代码，从而体积更小。

c. 使用 Image Minimizer 对项目中图片进行压缩，体积更小，请求速度更快。（需要注意的是，如果项目中图片都是在线链接，那么就不需要了。本地项目静态图片才需要进行压缩。）

4. 优化代码运行性能

a. 使用 Code Split 对代码进行分割成多个 js 文件，从而使单个文件体积更小，并行加载 js 速度更快。并通过 import 动态导入语法进行按需加载，从而达到需要使用时才加载该资源，不用时不加载资源。

b. 使用 Preload / Prefetch 对代码进行提前加载，等未来需要使用时就能直接使用，从而用户体验更好。

c. 使用 Network Cache 能对输出资源文件进行更好的命名，将来好做缓存，从而用户体验更好。

- d. 使用 Core-js 对 js 进行兼容性处理，让我们代码能运行在低版本浏览器。
- e. 使用 PWA 能让代码离线也能访问，从而提升用户体验。

1.4.20 vite 和 webpack 的区别

底层语言不同：

- 1. Vite 是基于 esbuild 预构建依赖，底层使用的是 go 语言，是纳秒级别的。
- 2. webpack js 的执行是毫秒为单位的，所以使用 Vite 比用 js 编写的打包器快 10-100 倍。

启动方式不同：

- 1. webpack 启动方式：分析依赖 => 编译打包 => 再交给本地服务器进行渲染。
 - a. 要分析各个模块之间的依赖，然后再进行打包。
 - b. 如果模块越多，关系越复杂，处理的就越慢，更新也就越来越慢。
- 2. vite 启动方式：启动服务器 => 请求模块时按需动态编译显示。
 - a. 采用的是懒加载方式，启动时不需要打包，不需要分析模块之间的关系，也不需要进行编译。
 - b. 只有在浏览器请求某个模块时，根据内容再进行编译，所以缩短了编译时间。
 - c. 项目越复杂，模块越多的情况下，优先使用 Vite，但是也有缺点。

缺点比较：

- 1. vite 生态不及 webpack，加载器、插件不够丰富。
- 2. 生产环境构建对于 css 和代码分割不够友好，vite 的优势是体现在开发阶段。
- 3. vite 首屏性能不及 webpack。
 - a. webpack 中浏览器发送请求，服务端直接把打包后的首屏内容发送给浏览器，没有性能问题。

b. 而 vite 在首屏方面, 就有些差了。i. 需要做一些额外的事情, 没有对源文件做合并捆绑操作, 会产生大量的 HTTP 请

求。

ii. dev server 运行期间对源文件做 resolve、load、transform、parse 操作。

iii. 预构建、二次预构建操作也会阻塞首屏请求, 直到预构建完成为止。

1.5 Vue

1.5.1 谈谈数据代理的理解? 谈谈对 vue 的理解?

Vue 是渐进式框架, Vue 当中是有数据代理的, vm 实例是代理者, data 是被代理者, 通过 vm 可以直接访问 data 中的属性, 进行属性值的获取, 或者设置和修改操作。

实现的流程:

在 new MVVM 或者 new Vue 的时候, 传入一个配置对象, 内部保存配置对象及内部的数据, 进行数据初始化操作, 然后遍历 data 中所有的属性, 调用 proxyData 方法, 进行数据代理实现, 该方法内部通过 Object.defineProperty 方法, 给 vm 实例对象添加 data 中的每个属性, 并且重写 get 和 set 方法, 数据代理实现。

1.5.2 模版解析的理解?

MVVM/Vue 中是有模版解析的, 在数据劫持之后开始进行模版解析, 先创建编译实例对象, 传入选择器和 vm 对象, 内部根据选择器获取模板容器, 模板容器如果存在, 才开始进行解析, 然后把模板容器中所有节点剪切到创建的文档碎片对象中, 调用 init 方法开始模版解析, 整个解析过程是在文档碎片对象中进行的, 文档碎片对象在内存中, 所以整个解析的流程是在内存中完成的, 最后会把解析后的文档碎片对象重新放到模板容器中。

init 方法内部调用的是 compileElement 方法, 并传入文档碎片对象, 该方法内部把文档碎片对象中所有的节点进行遍历, 同时采用的是深度递归解析的方式, 先判断节点是不是标签:

如果节点是标签，调用 `compile` 方法，传入当前的标签节点，该方法内部获取标签中所有的属性，然后进行遍历，判断每个属性是不是以 `v-` 开头，如果是，就说明是指令，然后干掉 `v-`；判断指令是不是以 `on` 开头，如果不是，说明是事件对象，那就调用 `compileUtil` 对象中的相关方法，内部调用 `bind` 方法，传入节点 `vm` 表达式和标识，最后调用 `updater` 对象中相关的方法进行普通指令的解析；如果此时的指令是事件指令，会调用 `compileUtil` 对象中的 `eventHandler` 方法传入节点 `vm` 表达式和去掉 `v-` 的事件指令名字，该方法内部在事件名字和回调函数都存在的情况下，为当前的节点使用 `addEventListener` 的方法绑定事件及对应的回调函数，并且回调函数内部 `this` 的指向是 `vm`，最后，无论是普通指令还是事件指令，都会通过节点的 `removeAttribute` 方法移除指令属性。如果是 `v-text` 指令，那就设置标签的 `textContent` 属性赋值进行解析；如果是 `v-html` 指令，调用的

是节点的 `innerHTML` 属性进行赋值解析；如果是 `v-class` 指令，最终调用的是 `class` 属性进行赋值解析。

如果不是，就判断节点是不是文本，并且是否符合插值的正则，然后提取出插值的文本节点和插值中的表达式，调用相关的方法，该方法内部调用 `compileUtil` 对象中 `text` 方法，内部调用的是 `bind` 方法，传入插值文本节点和表达式和 `vm` 和标识，在 `bind` 方法中，调用 `updater` 对象中的 `textUpdater` 方法，这个方法里面，最终使用节点的 `textContent` 属性进行表达式值的替换，插值的解析结束。

1.5.3 数据劫持的理解

数据劫持的目的是为了定义数据的响应式，在数据代理之后，模板解析之前。

通过 `observe` 函数调用，传入 `data` 对象所在空间的地址，进行数据的监视和劫持，该函数内部判断传入进来的数据是否是一个对象，并且值是有意义的。

开始创建劫持的实例对象，内部首先保存传入的对象数据，然后调用方法，方法内部遍历

data 对象中所有的属性，调用 convert 方法进行数据转换，该方法内部调用了 defineReactive 方

法，传入劫持对象的 data 属性和 data 中属性的名字和值，进行数据响应式的定义，这个方法

内部会创建属性对应的 dep 对象（

id、subs 数组），再次调用 observe 函数传入属性值，进行深度

递归数据劫持操作，最后使用 object.defineProperty 方法为劫持对象的 data，也就是外部的 data 对

象添加属性，并且重写 get 和 set 方法，也就意味着如果外部通过 vm 操作属性就会进入到数据代

理的 get 或 set 方法中，同时会自动的记录到劫持的 get 或 set 方法中。

其实所谓的数据劫持，就是定义响应式数据。

data 对象中有多少个属性，就会产生对应个数的 dep 对象。

数据劫持后开始进行模板解析，在模板解析中所有的表达式解析的过程中会调用 bind 方

法，该方法中会创建表达式对应的 watcher 实例对象，也就是模板中有多少个表达式就会创建

多少个 watcher 对象，创建 watcher 对象的时候会传入 vm、表达式及更新界面的回调函数，内部

会先进行这三个内容的存储，然后获取表达式的值，这个过程中会先跳入到数据代理的 get 方

法中，因为数据被劫持过，所以，会跳入到数据劫持的 get 方法中。

先判断表达式对应的 watcher 是否存在，如果存在，调用 watcher 对象的 addDep 方法，方法

内部会判断 watcher 对象的 depIds 中是否有传入进来的 dep 的 id；如果不存在，就把 watcher 对象添

加到 dep 对象的 subs 数组中，把 dep 的 id 作为键，dep 作为值，以键值对的方式添加到 watcher 对

象

的 depIds 中，至此建立了 dep 对象与 watcher 对象的关系，dep 对象和 watcher 对象的关系一共有 4

种，如下：

一对一的关系：

指的一个 dep 对象对应一个 watcher 对象

一对多的关系：

指的一个 dep 对象对应多个 watcher 对象

多对一的关系：

指的多个 dep 对象对应一个 watcher 对象

多对多的关系：

指的多个 dep 对象对应多个 watcher 对象

当关系建立后，界面中如果对 data 中的数据进行修改，就会自动的进入到数据代理的 set 方

法中，因数据劫持过，所以会进入到数据劫持的 set 方法中，该方法内部会先判断新旧值的变

化，然后调用属性对应的 dep 对象中的 notify 方法，方法内部会遍历 dep 对象中的 subs 数组，也就

是找到 dep 对象对应的所有 watcher 对象，调用 run 方法，先进行更新后的数据的获取，然后判断

新旧值的变化，并且重新建立 dep 与 watcher 的关系，然后数据发生变化，调用实例化 watcher 的

时候，传入进来的更新界面回调的函数，对界面进行重新渲染操作，以上，就是数据响应式的

原理

1.5.4 数据响应式的理解

进行数据劫持，根据 data 中属性的个数，产生对应个数的 dep 对象。进行模板解析，根据模板中表

达式的数量，产生对应个数的 watcher 对象。

同时建立 dep 对象和 watcher 对象的关系。

把 watcher 加入到 dep 中，更新数据，dep 通知对应的 watcher，更新界面【具体操作参考数据

劫持流程】；

把 dep 加入到 watcher 中，解除两者的关系【具体操作参考 watcher 建立流程】。

1.5.5 双向数据绑定指令 v-model 实现的流程:

在模板解析的时候，获取文本框节点，遍历里面的属性。找到 v-model 指令，属于普通指令，调用 compileUtil 对象的 model 的方法，内部通过 bind 方法，调用 updater 方法为文本框的 value 属性赋值，同时建立 dep 和 watcher 的关系。最终使用，addEventListener 方法绑定事件及对应的回调函数，当文本框的值发生改变，就会触发 input 事件，通过事件源对象获取改变后的新值，然后修改 data 中的属性值，会触发数据代理和数据劫持中的 set 方法，内部通过修改属性对应的 dep 对象，调用 notify 方法，通知关联的 watcher 对象调用 run 方法，对界面进行渲染。

1.5.6 谈谈 nextTick 的理解

首先在下次 DOM 更新循环之后，执行延迟回调，应该是在数据更新之后，立刻调用这个方法，获取更新后的 DOM，当这个方法执行的时候，会传入回调，内部有一个 callBacks 数组，用来存储传入的回调，以及更新界面的回调，如果，在更新数据后使用 nextTick 方法，内部会先把更新界面 watcher 对象对应的回调先加入到 callBacks 数组中，通过定时器的方式把手动传入的回调变成宏任务，这样就做到了先更新数据，再更新界面，再执行传入的宏任务回调。如果是先调用 nextTick 方法，后更新数据，那么内部 callBacks 数组中手动传入的回调会通过 promise 或者 mutation observer 变成微任务，更新界面的 watcher 对象对应的回调后执行，那么这个执行流程就是：更新数据，执行微任务，再更新界面。应用：XXX

1.5.7 谈谈对 vue2 的理解

1) 初识 vue2

首先 Vue 是一个渐进式框架，所谓的渐进式指的是本身实现的功能是有限的，相关的插件有很多，安装使用后可以实现更多的功能。

在 Vue 中，有数据代理、数据劫持和模板解析，diff 算法。

1、Vue 中常见的全局 api 有 (9 个)：

1. `Vue.extend` 方法：通过传入一个组件对象，可以得到该组件对应的构造器，继承了 `Vue`。
2. `Vue.nextTick` 方法：在下次 DOM 更新循环之后执行延迟回调。
3. `Vue.delete` 方法：可以用来移除响应式对象上的属性且触发视图更新。
4. `Vue.set` 方法：可以向响应式对象中添加一个属性，也是响应式的，触发视图更新。
5. `Vue.directive` 方法：用来定义全局指令。
6. `Vue.filter` 方法：可以用来定义过滤器，通常可以实现日期或时间的格式化操作。
7. `Vue.component` 方法：可以用来定义全局组件，其返回值是该组件实例对应的构造器。
8. `Vue.use` 方法：用来注册和使用插件，内部实现原理：调用 `install` 方法，把对象挂载到 `Vue` 的原型上。
9. `Vue.compile` 方法：可以传入模板字符串，最终编译成 `render` 函数。

2、Vue 中常见的组件选项有 (8 个)：

1. `data`：在组件中 `data` 是一个函数，不能是一个对象，原因是：组件多次复用的时候，要保证 `data` 中的数据是响应式的，而且不能出现冲突。
2. `computed`：计算属性，指的是一个数据发生变化，相关联的数据也会发生变化。计算属性的数据是有缓存的，是响应式的。
3. `methods`：是一个对象，里面用来定义方法。
4. `watch`：监视属性。当一个数据发生变化，需要做什么事情的时候。内部有深度监视 `deep:true` 和默认执行操作 `immediate:true` 的设置。
5. `directives`：用来在组件内部定义多个局部的自定义指令的方法。
6. `filters`：用来定义组件内部的过滤器的方法。
7. `components`：用来注册组件。

8. name: 组件的名字。如果是递归组件, 必须要有 name 属性。

3、Vue 中常见的指令有 (7 个) :

1. v-text: 相当于插值语法 (插值语法效率高于 v-text), 设置标签之间的文本内容, 本质是 textContent, 或者 innerText (textContent 是浏览器的标准属性, innerText 并没有进行使用)。
2. v-html: 用来设置标签中间的 HTML 内容, 其原理使用的是 innerHTML。
3. v-for: 用来遍历对象或数组, 内部需要使用到 in, 也可以使用 of。
4. v-if / v-else-if / v-else / v-show : 用来控制组件或标签显示或隐藏。v-show 控制的是 css 中 display 属性。v-if / v-else-if / v-else 控制 DOM 创建或销毁的。v-show 的效率更高。vue2 中 vue3 中都不建议 v-if 和 v-for 同时使用。Vue2 中 v-for 的优先级高于 v-if, Vue3 中 v-if 的优先级高于 v-for。
5. v-bind: 强制数据绑定指令。多数情况下用在组件或普通标签上。v-bind 中可以传入对象, 键是属性, 值是对应的表达式。
6. v-model: 双向数据绑定指令。多数情况下用在组件或表单标签上。
7. v-on: 用来实现组件或标签的事件绑定操作。v-on 后面可以绑定一个对象, 键是事件名, 值是回调函数。

4、Vue 中特殊的属性有 (4 个) :

1. key: 需要传入标识。一组数据的展示, 如果没有增删改操作, 只是遍历显示, 可以使用索引作为 key 值。
2. ref: 用来获取组件对象或者是 DOM 对象。
3. is: 需要配合内置组件 component 进行动态组件的创建和使用。
4. slot / slot-scope / scope: 用来操作插槽的, 这三个已经被废弃了, 但可以使用。

5、Vue 中内置的组件有 (4 个) :

1. component: 用来定义动态组件
2. transition: 实现过渡效果的
3. keep-alive: 用来缓存组件的
4. slot: 插槽, 分为普通插槽、具名插槽、作用域插槽。

2) Vue2 生命周期 (11 个)

Vue 中是有生命周期的。所谓的生命周期, 就是 Vue 实例或者是组件实例对象从创建到销毁的一个过程, 在这个过程中会牵扯到 11 个钩子, 在不同的时机被自动的调用, 分别是:

1. **beforeCreate**: 表示的是实例对象已经创建, 但是数据还没有初始化, 是数据初始化之前的一个钩子, 这里是不能使用 data 和 methods 的。(几乎不用)
2. **created**: 数据初始化完毕后执行的钩子, 这里可以使用 data 和 methods, 如果涉及到页面首屏数据展示, 这里可以发送 ajax 请求进行异步操作。此时数据代理已经完成, 开始进行数据劫持操作。(自行扩展: 数据代理、数据劫持原理)
3. **beforeMount**: 挂载之前, 也可以叫界面渲染之前的钩子, 此时页面呈现的是数据没有更新前的假页面, 也就是内部已经进行了模板解析, 但是没有完事儿。
4. **mounted**: 挂载后的生命周期钩子, 此时界面已经渲染完毕, 也就意味着模板解析已经结束。
5. **beforeUpdate**: 如果界面中有响应式数据改变的情况, 此生命周期钩子被调用, 此时 data 中的数据是最新的, 数据对应的 dep 会通知对应的 watcher 对象, 进行页面的更新操作。
6. **updated**: 在界面更新完毕后被调用。7. **beforeDestroy**: 表示的是组件卸载前的生命周期钩子, 关闭定时器、解绑自定义事件、取消消息订阅等都在这个钩子进行。
8. **destroyed**: 组件销毁后调用的生命周期钩子, 比如: 某组件中用到了一次性的缓存信息, 并且组件切换后, 缓存信息不需要了, 此时可以在这个钩子中清空缓存数据, 如添加购物

车成功界面，跳转后销毁数据操作。

② 如果有父子组件生命周期钩子执行顺序是：先执行父级组件的 `beforeCreate`、`created`、`beforeMount`，然后执行子级组件的 `beforeCreate`、`created`、`beforeMount`、`mounted`，最后是父级组件的 `mounted`，以上是父子组件加载的时候前 4 个生命执行的顺序，如果涉及到父子组件公用的数据被修改，钩子的执行顺序是先执行父级组件的 `beforeUpdate`，子级组件的 `beforeUpdate` 和 `updated`，最后是父级组件的 `updated`，如果父子组件进行销毁操作，先执行父级组件的 `beforeDestroy`，再执行子级组件的 `beforeDestroy` 和 `destroyed`，最后执行父级组件的 `destroyed`。

9. `deactivated`：如果组件被缓存了，组件进行切换操作，先执行子级组件的失活的生命周期 `deactivated`，然后再执行父级组件的 `deactivated`。

10. `activated`：如果是激活操作，先执行子级组件的 `activated`，再执行父级组件的 `activated`。

11. `errorCaptured`：用来抓捕子级组件报错的生命周期钩子。

3) 组件通信 (13 个)：

Vue2 中是有组件通信的。组件指的具有特定功能效果的集合，包含 HTML、CSS、JS，组件里可以没有 模板、CSS，组件其实也是对象。组件和组件的关系有：父子/祖孙关系（直接/间接）、兄弟关系、任意关系。组件通信指的是：不同关系的组件进行数据的传递。常见的组件通信方式有：

1. `props`：父子、子父组件通信【vue3 借助 `defineProps`】
2. `自定义事件`：父子组件通信
3. `插槽`：父子、子父组件通信。插槽分为普通插槽，具名插槽和作用域插槽，用来占位。
4. `ref`：父子组件通信。用来获取 DOM 对象或组件对象，然后调用组件中的属性或方法。
5. `v-model 指令`：父子组件通信。本质是 `input` 事件和 `value` 属性。

6. `.sync`: 父子组件通信。本质是 `update` 事件 + 动态属性。
7. `$attrs` 和 `$listeners`: 父子组件通信。封装复用率比较高的组件中会用到, 可以称之为高级组件。
8. `$parent` 和 `$children`: 父子组件通信
9. `provide` 和 `inject`: 父子/祖孙组件通信 10. `事件总线` (一般不用): 任意组件通信。原理是在 `Vue` 的原型对象上绑定一个 `Vue` 的实例, 各个组件的实例对象与它是继承关系。举例: 原型对象上可以绑定一个 `$api` 存储所有的接口函数。
11. `vuex`: 任意组件通信。集中式状态管理工具, 里面有 5 个对象:
 - a. `state`: 包含了多个状态数据的对象;
 - b. `mutations`: 包含了多个直接修改状态数据的方法的对象;
 - c. `actions`: 包含了多个间接修改状态数据的方法的对象;
 - d. `getters`: 包含了多个状态数据的计算属性的 `get` 方法的对象;
 - e. `modules`: 包含了多个子级模块的对象。
 - f. 数据修改方式为: 组件中分发 `action`, `actions` 内部 `commit mutation`, `mutation` 内部修改状态数据, 或者组件中直接 `commit` 对应的 `mutation` 然后修改数据。
 - g. 总的 `action` 和子的 `action`, 先执行总的, 再执行子的。
 - h. `mutation` 中可以做异步操作, 但是不推荐, 因为浏览器的 `vue` 调试插件中不能够及时的更新数据, 有可能会造成界面不能及时更新。
 - i. `vuex` 中常用的辅助函数: `mapState`、`mapMutations`、`mapActions`、`mapGetters`。
12. `pubsub`: 任意组件或页面通信。不属于任何框架, 独立存在。
13. `路由传参和本地缓存`: 任意组件通信

我的项目中，一般情况下会用到 3-5 种左右，比如：前台项目中会用到：props、自定义事件、路由传参和本地缓存；后台项目中会用到：插槽、ref、vuex、v-model、路由传参和本地缓存。

4) 谈谈路由的理解：

1. 路由指的是一种映射关系：地址和组件的关系，也就是 path 和 component 的关系。路由需要通过路由器进行管理。多个路由对象会形成一个数组，而路由对象形成的数组，每个路由其实也是对象，也就意味着浏览器的地址栏中，输入地址，回车后，可以看到地址对应的组件内容。

2. 路由分为 hash 和 history 模式，区别在于地址栏中的地址是否带 #，以及项目上线后，history 模式的路由如果直接访问某个地址，页面中的 js 解析前台路由路径的时候，在服务器的根目录下找不到对应的资源，会返回 404，服务器需要配置 try_files，当 404 时就返回首页。

3. 路由分为路由链接和路由视图，也就是 router-link 和 router-view，前者最终会产生 a 标签，

to 属性中的值会存储到 href 属性中，页面中点击 a 标签，路由视图会显示对应的路由组件内容，这种方式属于声明式路由，也就是不需要书写 js 代码，就可以实现路由跳转

了。编程式路由，需要通过书写 js 代码来实现路由跳转，需要通过路由器对象调用 push

和 replace 方法，路由跳转的过程中，可以进行参数的传递，有 query 和 params 还有 props

以及 meta。params 方式需要使用：

冒号占位，props 分为布尔、对象、函数模式，？

问

号表示参数可有可无。4. 路由也分为一级路由二级路由，还有多级路由。

5. 路由 3 跳转 bug: 路由在程式跳转的过程中, 重复跳转同一路由地址会出现 bug。原因是路由升级的相关版本当中, 程式路由跳转返回的是一个 promise 对象, 需要在 push 或 replace 方法中传入成功或失败的回调, 或者是调用 .then、.catch 方法。一个项目中会出现多次的程式路由的跳转, 每次跳转都要传入回调很麻烦, 所以, 通过在路由器的原型对象中重写 push 和 replace 方法, 默认传入成功和失败的回调, 从根本上解决问题, 再有程式路由的时候, 就不用那么麻烦了, 当然, 现在的路由版本中 (路由 4) 官方已解决此 bug。
6. 路由守卫 / 导航守卫: 是用来控制路由跳转的, 分为全局导航守卫 (3 个)、路由独享守卫 (1 个) 和组件内的守卫 (3 个), 一共有 7 个钩子, 313 的形式。全局导航守卫: 分为全局前置守卫 (beforeEach)、全局解析守卫 (beforeResolve) 和全局后置守卫 (afterEach)。
- 路由独享守卫 (beforeEnter)。组件内的守卫: 分为前置守卫 (beforeRouteEnter)、解析守卫 (beforeRouteUpdate) 和后置守卫 (beforeRouteLeave)。
7. 路由的本质 (原理):
- router-link 和 router-view 在路由器插件安装后, 都变成了全局组件。router-link 最终被编译为 a 标签, a 标签的跳转实际上是通过 location.hash 属性来设置的。程式路由中的 push, 实际上是 BOM 中的 history 对象中的 pushState 方法的调用, 程式路由中的 replace 方法实际上是 BOM 对象中的 history 对象中的 replaceState 方法的调用。路由跳转的时候传递的参数, 可以通过 BOM 中的 history 对象中 state 来获取。

5) vue 中常见的各种组件 (7 个):

1. 普通组件
2. 全局组件: 使用 vue.component 方法进行注册
3. 路由组件: 如果一个普通组件进行了路由注册, 此时这个组件就叫做路由组件。

4. 动态组件：使用 `component` 组件 + `is` 属性动态产生的组件
5. 异步组件：可以通过工厂函数动态编译的组件，实现了按需加载，工厂函数内部可以使用 `template` 进行模板编写，或者是 `require` 引入，或者是 `import()` 的方式引入组件。
6. 函数组件：需要设置组件内的选项，

`functional: true` + `render` 函数，可以实现函数组件的定义。特点：无状态、无实例 `this`、无生命周期。所需数据需要父级组件传递。
7. 递归组件：组件内部调用组件自己，必须要有 `name`。

1.5.8 vue2 与 vue3 的区别

1. 首先 `vue3` 中支持 `vue2` 的大多数特性，更好的支持 `ts`，打包文件变小了，初次渲染速度和更新渲染速度变快了，内存占用率变小了。
2. `vue2` 是 2016 年出现的，`vue3` 是 2020 年出现的。
3. `vue2` 中使用的是选项式 `api`，`vue3` 中使用的是组合式 `api`。
4. `vue2` 中组件内部必须要有唯一的根标签，`vue3` 中可以没有根标签。
5. 可以使用 `vue-cli` 脚手架 或者 `vite` 创建 `vue3` 项目（`vite` 也可以创建 `React`、`vue2`、`vue3` 项目）。
6. 技术栈：
 - a. `vue2` 的技术栈：`vue-cli` 脚手架、`vue2`、`js`、`vue-router3`、`vuex3/4`、`axios`、组件库（`ui`）、插件库
 - b. `vue3` 的技术栈：`vite`、`vue3`、`ts`、`vue-router4`、`pinia`、`axios`、组件库（`plus`）、插件库
 - c. `vue2` 中也可以使用 `pinia`，需要安装一个组合 `api`：`$vue/composition-api`，然后在 `main.js` 文件中引入 `PiniaVuePlugin`，再通过 `vue.use` 声明使用一下，还需要在 `new vue` 的时候，注册 `pinia`。

d.

如果您的应用使用 Vue 2，您还需要安装组合 API：@vue/composition-api。

如果您使用的是 Vue 2，您还需要安装一个插件并将创建的 pinia 注入应用程序的根目录：

```
import { createPinia, PiniaVuePlugin } from 'pinia'
Vue.use(PiniaVuePlugin)
const pinia = createPinia()
new Vue({
  el: '#app',
  // 其他选项...
  // ...
  // 注意同一个 `pinia` 实例可以在多个 Vue 应用程序中使用
  // 同一个页面
  pinia,
})
```

e. vue3 中推荐使用 pinia，pinia 中可以使用 \$patch 方法对状态数据进行批量修改，该方法中

可以直接传入对象，对象中书写键值对的方式直接修改状态数据；也可以传入一个回

调，回调中的参数 state 在方法内部调用属性进行数据修改；也可以直接通过模块对象调

用 \$state 赋值对象内部属性键值对，直接替换状态数据的方式来进行修改。

7. vue3 中增加了 2 个内置组件：

a. <Teleport>，叫瞬移组件，也可以叫传送门

b. <Suspense>，占位具有加载效果组件

8. 关于 setup 函数：

a. setup 函数是组合 api 中的一个，也是组件中的一个新选项，也是入口点。

b. setup 方法执行的时机比 beforeCreate 更早，所以里面是没有组件实例对象的，也就意味

着 this 是 undefined，不能使用的。

c. setup 方法返回值是一个对象，对象中的数据可以在模板中直接使用，该方法的参数有 2

个：

i. 参数 1 是 props，用来接收父级组件传递进来的数据，并且通过 props 接收的；

ii. 参数 2 是 context，可以认为是组件中的执行上下文，相当于 this。

1. 可以调用 attrs 属性，里面包含了：没有通过 props 接收的属性；

2. 也可以调用 slots，获取传入进来的插槽内容的对象；

3. 还可以调用 emit，用来分发自定义事件的函数。

d. 不建议在 setup 中进行异步操作。

9. 如果在 vue3 中，使用 vue2 语法的 data 和 methods 里面的数据和方法名和 setup 中定义的数据和方法名重名了，官方最初给的解决方案是：优先使用 setup 中的，现在官方给的解决方案是直接报错。

10. vue 中定义响应式数据：

a. 通过 ref 定义基本类型数据的响应式，当然，也可以来定义复杂/引用类型数据的响应式，但是内部的数据的类型，都是 proxy 类型。

b. 也可通过 reactive 定义复杂/引用类型数据的响应式，定义的响应式数据是代理数据，也是深度响应式的。

11. 定义响应式数据的原理：

a. vue2 中响应式数据的原理使用的是 Object.defineProperty 【扩展数据代理、数据劫持】

方法，并且进行重写操作，以及重写了数组中用来更新数据的方法（push、pop、

unshift、shift、sort、reverse、splice），缺点：响应式对象通过 .点语法，直接添加属性

和数组通过索引添加及修改数据，都不是响应式的。需要调用 set 方法【当 data 为对象

时】或者是更新数组的方法【当 data 为数组时】，才可以实现响应式。b. vue3 中数据代理及响应

式，使用的是 proxy（代理）【扩展数据代理、数据劫持、模版

解析、响应式数据】 + Reflect（反射：

动态对被代理对象的相应属性进行特定的操

作)

。

12. 关于生命周期:

- a. vue2 中的生命周期 11 个,
- b. vue3 中的生命周期 12 个。
- c. vue3 中干掉了 beforeCreate 和 created。
- d. vue3 中的生命周期都是 api, 内部传入回调函数, 可以重复多次调用。
 - i. vue3 中的 onBeforeMount()和 onMounted()相当于 vue2 中的 beforeMount 和 mounted,
 - ii. vue3 中的 onBeforeUpdate()和 onUpdated()相当于 vue2 中的 beforeUpdate 和 updated,
 - iii. vue3 中的 onBeforeUnmount()和 onUnmounted()相当于 vue2 中的 beforeDestroy 和 destroyed,
 - iv. 捕获子级报错: onErrorCaptured(),
 - v. 激活和失活: onActivated()、onDeactivated(),
 - vi. 服务端: onRenderTracked()、onRenderTriggered()、onServerPrefetch()
- e. vue3 中所有的生命周期钩子要比 vue2 中对应的生命周期钩子早一步。如下图:

```
[webpack-dev-server] Server started
3.x的setup
2.x===beforeCreate===执行了
2.x===created===执行了
3.x===onBeforeMount===执行了
2.x===beforeMount===执行了
3.x===onMounted===执行了
2.x===mounted===执行了
>
```

13. 关于通信：

a. vue2 中通信：

i. 父子组件通信：props、自定义事件、插槽、ref、v-model、.sync、\$attrs 和\$listeners、

\$parent 和\$children、provide 和 inject（父子、祖孙）

ii. 任意组件通信：pubsub、事件总线、路由传参和本地缓存、vuex

b. vue3 中通信：

i. 没有.sync、\$listenerts

ii. 组件中如果使用 v-model，可以书写多个，其本质是 update 事件+动态属性

iii. 可以使用 mitt 插件实现事件总线，但是几乎不用

14. 关于计算属性与监视属性：

a. vue3 中的计算属性与监视属性与 vue2 中的略有不同。

b. vue3 中的计算属性也是一个 api，

i. 可以传入一个对象，对象中可以书写 `get` 和 `set`，对数据进行修改或者是获取操作。ii. 也可以直接传入一个回调，代表的是 `get` 操作，计算属性的返回值是一个 `ref` 类型的数据/对象。

c. vue3 中的监视属性常用的有 2 个，一个是 `watch`，一个是 `watchEffect`。

i. `watch` 在使用的时候可以传入 3 个参数，

1. 参数 1: 可以是一个带有返回值的回调，也可以是一个 `ref`，还可以是一个响应式的

对象，或者可以...写拆包的形式，三点运算符拆包后的数据也应该是响应式的，

也可以是一个数组，数组中的数据必须是前面方式中的一种；

2. 参数 2: 是用来书写逻辑的回调函数，

3. 参数 3: 是一个对象，里面可以设置 `deep`、`immediate`。

ii. `watchEffect` 当中传入一个回调，默认会执行一次。

1.6 react

1.6.1 谈谈 react 和 vue 的区别

数据流：

无论是 react 还是 vue，都是单向数据流，但是 vue 中有双向数据绑定，

虚拟 DOM：

都有虚拟 DOM，

react 中每当应用状态发生改变时，全部的子级组件会重新渲染，也可以通过

`PureComponent/shouldComponentUpdate` 这个生命周期方法来进行控制。

但是 vue 中，每个组件都有对应的依赖关系，不需要重新渲染整个组件数据，这个行为是默认的。

组件化：

关于组件化，vue 提供了单文件组件的方式，里面包含了 html、css、js，类似于 html 语法，而

react 是通过 jsx 语法来定使用组件，二（

r 创建 vue2、vue3、react 项目 ue3）都支持 ts 和 tsx 语法，

更新用户界面方式：

更新用户界面的方式不一样。vue 中的数据是响应式的，数据更新界面也会更新，而 react 需要

使用 setState 方法，或者是 useState 提供的函数来触发用户界面更新，

react 不是响应式的。

构建工具：

构建：

1、有各自的脚手架，vue 使用的是 vue-cli

2、react 使用的是 create-react-app

3、而 vite 都可以创建 react、vue2、vue3

跨平台方案：

vue 中跨平台使用 Weex

react 跨平台使用 native

数据可变性：

vue 强调的是数据的可变性，

react 强调的是不可变数据，也就是不修改原数据，每次产生一个

全新的数据，然后进行差异对比，界面就更新了。**组件通信：**

react 中组件通信，可以使用 props 的方式，实现父子组件通信。也可以使用回调函数实现子父

组件的通信。还可以使用自定义事件的方式实现兄弟组件通信。还可以使用 context 实现祖孙组

件通信，还有 redux 和 mobx 实现任意组件的通信。

1.6.2 redux 和 mobx 的区别？

redux：

redux 其他的框架中也可以使用，现在和 react 实现了深度绑定，是一种集中式的管理状态方

案，由 store（用来管理状态数据）、actions（用来返回一个 action 对象的函数模块）和 reducers

（根据之前的状态和 action 对象来计算生成新的状态函数模块）组成。

流程：组件中通过 store.dispatch 传入 action 对象，触发 reducers，内部根据 action 的 type 使用 switch

case 进行对比和计算，修改状态数据，手动更新组件。

mobx：

mobx 这种方式是将数据保存在分散的多个 store 中，使用 observable 保存数据，数据是响应式

的，当数据发生变化后，自动处理用户界面的更新，

reduce 使用的不可变状态，也就是状态是

只读的，不能直接修改，而是返回一个新的状态，mobx 中的状态是可变的，可以直接进行修

改。

流程：通过 class 的方式来定义，内部的数据是状态数据，方法是用来更新状态数据的，构造器

中需要使用 makeAutoObservable 方法传入 this，让实例上的属性和方法变成响应式的，组件内

部可以通过这个类的实例调用方法修改数据，也可以直接修改数据，最终使用 observer 高阶组

件对当前组件进行包裹，用来监听数据变化，一旦变化就会自动的重新渲染组件。

1.6.3 react 中常见的 hook

1. useState：让函数组件定义并维护状态数据 state
2. useEffect：让函数组件使用生命周期
3. useCallback：缓存函数，不必重新生成一个新的函数
4. useMemo：缓存计算结果值

1.6.4 react 中定义组件的 2 种方式：

类组件和函数组件。

1. 类组件需要继承 class，函数组件不需要。
2. 类组件可以访问生命周期钩子，函数组件不能。
3. 类组件中可以使用 this，函数组件不可以。
4. 类组件中可以定义和维护 state，函数组件不能。
5. 函数组件想要做这些事，需要使用 hook 函数。

class 类组件生命周期：

react15 类组件的生命周期：

初始阶段：componentDidMount() {}

更新阶段：

shouldComponentUpdate() {}

卸载阶段：componentWillUnmount() {}

react16 类组件的生命周期：

初始阶段：componentDidMount() {}：用来发送请求，设置定时器，绑定事件等任务；更新阶段：

shouldComponentUpdate() {}：性能优化，减少 render 次数；

卸载阶段：componentWillUnmount() {}：清除定时器，解绑事件等收尾工作

函数式组件生命周期：

通常使用 React.useEffect 来完成

扩展：useEffect 依赖为空数组与 componentDidMount 区别在 render 执行之后，

componentDidMount 会执行，如果在这个生命周期中再一次 setState，会导致再次 render，但

是浏览器只会渲染第二次 `render` 返回的值，这样可以避免闪屏。

但是 `useEffect` 是在真实的 DOM 渲染之后才会去执行，这会造成两次 `render`，这两次可能都会渲染出来，可能会闪屏。

实际上 `useLayoutEffect` 会更接近 `componentDidMount` 的表现，它们都同步执行且会阻碍真实的 DOM 渲染的。

扩展：`useEffect` 和 `useLayoutEffect` 区别对于 React 的函数组件来说，其更新过程大致分为以下步骤：

1. 因为某个事件回调导致 `state` 发生变化。
2. React 内部更新 `state` 变量。
3. React 处理更新组件 `render` 方法中 `return` 出来的 DOM 节点（进行一系列 `dom diff`、调度等流程）。
4. 将更新过后的 DOM 数据绘制到浏览器中。
5. 用户看到新的页面。

`useEffect` 在第 4 步之后执行，且是异步的，保证了不会阻塞浏览器进程。。`useLayoutEffect` 在第 3 步至第 4 步之间执行，且是同步代码，所以会阻塞后面代码的执行

1.6.5 react 中的路由

需要使用 `createBrowserRouter` 创建路由对象，设置路由的配置选项 `path` 和 `element`，通过

`RouterProvider` 组件来渲染路由，路由配置项中 `children` 设置子路由，子路由默认是不渲染的，

需要通过父路由加载 `outlet` 组件才会渲染子路由，

react 也有声明式路由和编程式路由，声明式

路由使用的是 `link` 和 `NavLink`，`NavLink` 激活的时候有高亮，默认会有一个类名 `active`。编程式路

由使用的是 `useNavigate` 和 `redirect` 方法，路由的参数有 `query`、`params` 和 `state`，但是 `state` 地址栏

不可见，重新加载页面，参数会丢失，可以使用 `Suspense` 组件和 `lazy` 函数实现路由懒加载。

1.7 小程序

1.7.1 谈谈小程序的生命周期

在微信小程序中生命周期分为 3 种，分别是：

1. 应用实例的生命周期（3 个）：

- a. `onLaunch`：表示的是小程序实例初始化的时候执行。
- b. `onShow`：表示的是小程序启动或者切换到前台时执行。
- c. `onHide`：表示的是小程序切换到后台时执行。

2. 页面实例的生命周期（5 个）：

- a. `onLoad`：表示的是页面实例对象创建的时候执行。
- b. `onShow`：表示的是页面刚要绘制的时候执行（即页面刚要显示的时候执行）。
- c. `onReady`：表示的是页面第一次渲染完毕的时候执行。
- d. `onHide`：表示的是页面隐藏的时候执行。
- e. `onUnload`：表示的是页面卸载的时候执行。

3. 组件实例的生命周期（5 个）：

- a. `created`：表示的是组件实例刚被创建的时候执行。
- b. `attached`：表示的是组件实例进入到页面节点树的时候执行。
- c. `ready`：表示的是组件在布局完成后执行。
- d. `moved`：表示的是到另一个位置的时候执行。
- e. `detached`：件实例从页面中移除时执行

1.7.2 小程序中通信方案

1、页面和页面之间的通信：

1. 可以使用全局唯一小程序实例，也就是在 app.js 文件中定义 globalData 全局数据，其他页面中通过 getApp 方法得到全局唯一实例对象，页面中可以使用这个数据。
2. storage：主要是为了缓存数据最多存储 10M，但是也可以实现页面之间数据的传递。
3. 也可以使用自定义全局事件总线 eventBus 的方式。
4. 还可以使用 pubsub 消息订阅的方式实现页面通信。
5. 还可以使用 getCurrentPages 方法获取某一个页面实例，调用里面的数据进行页面的通信。

2、组件和组件之间的通信：

1. 父级组件直接向子级组件标签以动态属性的方式传递数据，子级组件内部通过 properties 接收父级组件传过来的数据到然后进行使用，实现通信。
2. 还可以使用自定义事件的方式，父级组件向子级组件传递自定义事件，子级组件内部通过 triggerEvent 方法分发事件及传递数据，实现通信。

1.7.3 原生小程序和 uniapp 的区别

原生小程序只能开发小程序，里面使用的是组件。而 uniapp 是基于 vue 的框架，不仅可以做小程序开发，还可以做 webApp 和混合 app，一套代码多个平台运行，以及发布安卓或 ios 的应用。

原生小程序是 wx 开头的 api，像素单位是 rpx；uniapp 框架前缀是 uni 开头的 api，像素单位是 upx，现在的 uniapp 中直接支持 rpx。

uniapp 框架中开发小程序的时候可以使用 HTML 代码，可以使用 vue 的生命周期钩子。

1.7.4 小程序登陆流程

首先在 app.js 文件中，onLaunch 生命周期里面判断本地缓存中是否有用户信息，如果有，加载用户信息跳转到首页；如果没有，调用 wx.login 方法得到授权码 code 值，然后将 code 码值调用后台接口发送的开发者服务器，开发者服务器内部在该接口对应的回调函数内部通过 req.query.code 获取 code 码值，然后再准备开发小程序的 app.id 以及开发小程序的密钥，再准备 code2session 接口地址，【通过 flyio 调用方法传入地址及准备好的参数向微信服务器发送请求，获取 session_key 和 openid 数据，然后通过 jsonWebToken（简称 jwt）调用 sign 方法传入用户的相关信息数据（可以包含 session_key 或 openid）及密钥产生 openid，可以通过 md5 进行二次加密，把 token 返回到小程序中。】让后端人员调用该接口获取 openid 等相关信息，产生对应的 token，发送给前台小程序。小程序中把获取到的唯一标识 token 缓存到 storage 中，然后通过 wx.getUserProfile 方法获取用户的相关信息进行存储，然后进行页面的跳转，在相关的页面中展示用户信息即可

1.7.5 小程序的支付流程

1.7.6 小程序的分包

第 2 章 Vue2 前台项目

2.1 项目介绍

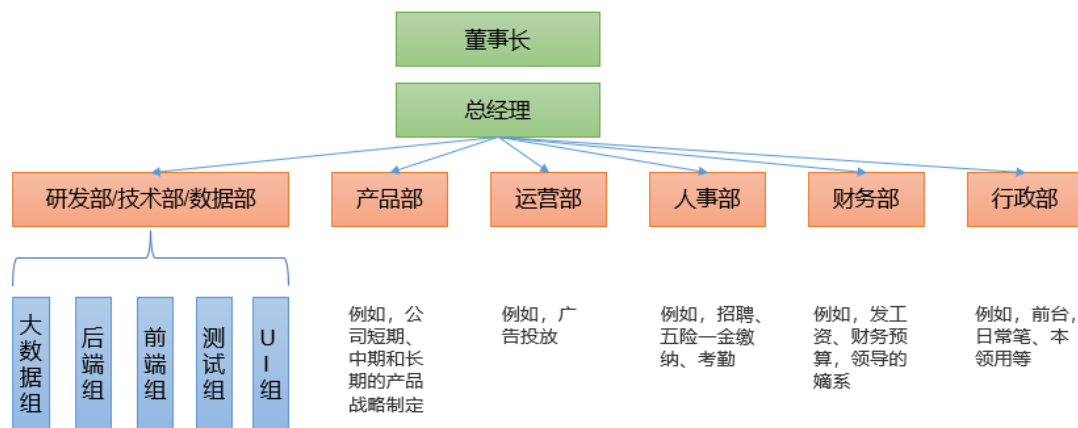
2.1.1 说说流程

首先根据需求文档和 ui 设计图，进行项目搭建，需要多方商讨，确定项目的技术栈，搭建前端项目的架子，设定技术栈：vue-cli、vue2、vue-router3、vuex、axios、js 等，要把项目所用到的一些基本的插件、目录、重置 css 搭建好之后，推送到对应的代码管理仓库中（github/gitee，拉取分支，创建并切换到子分支，进行项目的开发。）。

2.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块

2.2 人员配置参考

2.2.1 整体架构



2.2.2 人员配置

小型公司：

中小型公司：。

中型公司：

中大型公司：

2.3 特定功能实现

2.3.1 首页三级分类业务逻辑思路

搭建项目首页（具体写个人负责的页面），把首页进行组件的拆分，拆分成 header 组件、footer

组件、home 组件（涉及到路由注册）。header 组件搭建思路：通过 html+css（less/sass/stylus）的

方式搭建页面显示对应的效果。home 组件也是先通过 html+css（less/sass/stylus）的方式搭建

页面，然后进行组件的拆分，考虑到三级分类信息的展示要在多个组件中使用，所以也抽取出来

来变成一个组件（全局公共组件），因为此时后台没有接口，所以我采用的是根据效果图自己

定义 json 文件，设计了一些数据，通过 mockjs 进行拦截，生成随机数据的方式进行展示。具体的步骤如下：首先下载 mockjs，然后进行安装，然后在对应的 json 文件当中，定义一二三级分类部分数据，然后获取三级分类 mock 相关的接口，并且返回 json 文件中的数据。然后二次封装 axios，设置根路径、超时时间、进度条效果、请求以及响应拦截器，然后定义获取三级分类的 api 接口函数，配置代理，设置跨域信息。安装 vuex，拆分首页的 js 模块，定义状态数据，修改数据的 mutation，引入 api 接口函数，封装获取三级分类的 action，在 app 组件中，通过 dispatch 分发对应的 action，在三级分类对应的组件中，通过 mapState 获取三级分类信息数据，使用三层 v-for 指令和 router-link 遍历及显示三个级别的分类信息数据，发现页面卡顿，调整成 a 标签直接显示分类数据，然后绑定事件，实现路由跳转及参数传递，发现页面卡顿，通过事件委托，为最外层的父级标签绑定一个点击事件，内部判断触发事件的源是 a 标签的情况下，实现路由跳转，并且从路由对象中，获取对应的 params 参数，如果有，就合并到 query，进行参数传递，如果没有，直接传 query 参数。通过节流的方式，优化鼠标在一级分类信息滑动事件展示高亮显示效果，减少事件触发次数。后台接口准备成功后，直接替换 mockjs 的接口即可。

2.3.2 详情页中销售属性功能实现流程

根据接口调用返回来的销售属性对象数组，通过双层 v-for 遍历销售属性对象数组和属性值对象数组展示信息，为每个属性值绑定点击事件，传入销售属性对象、属性值对象、销售属性索引，内部首先把当前的销售属性对象中属性值对象数组遍历，设置选中状态为 false，点击的这个属性值对象的选中状态为 true，也就是排他功能。获取属性值的 id，以及其他属性值选中的 id，使用 | 将多个属性值的 id 拼接，形成字符串，在 json 格式的对象中判断拼接后的属性值 id 是否有对应的产品 id，如果有，通过路由器对象调用 replace 方法进行路由的跳转，如果没有，就设置购物车取消高亮显示，禁用按钮点击。

2.3.3 购物车功能实现的业务逻辑思路

首先在详情页，给‘加入购物车’按钮绑定点击事件，回调函数内部先调用加入购物车的 api 接口函数，传入商品的 id，接口调用成功后，再进行路由跳转，并且传递商品数量，同时把商品信息对象存储到 sessionStorage 中【因为加入购物车成功页面，还需要继续显示商品信息，就可以从缓存中读取这个数据】。

此时进入到加入购物车成功页面，从 sessionStorage 中读取商品信息进行显示。

然后点击‘去购物车结算’的按钮，进行购物车路由的跳转，此时可以把 sessionStorage 中的商品信息清空，同时路由跳转到购物车界面。

1. 用户在没有登录的情况下，依然可以看到添加到购物车的商品列表数据，采用的是临时用户 id 的方式，把商品加入到购物车，用到了一个 uuid 的插件，把产生的临时用户 id 缓存到 vuex，同时在请求拦截器中请求头嵌入了临时 id；
2. 如果用户登录了，把商品加入到了购物车，进入到购物车页面后，展示的是用户未登录和已登录时添加到购物车的商品列表数据（后台人员已经把两种数据进行了合并）。

购物车组件页面，是一个路由组件，采用 html+css 的方式搭建静态页面，然后调用获取购物车列表的 api 接口函数，得到购物车的数组数据，vuex 内部通过计算属性 getters 计算出全选的状态、商品的总数和总价格，组件内部通过 v-for 指令遍历所有的商品列表数据进行展示，并获取计算属性中的数据，展示全选的状态、商品的总数和总价格，

- ① 每个商品都可以进行选中、未选中状态的改变，采用防抖方式优化事件触发的次数，减少请求数量，减轻服务器压力。
- ② 商品数量的加减修改，也采用了防抖或节流优化事件触发次数。
- ③ 单个商品的删除，以及选中商品的删除都进行了弹框提示处理，也是为了提升用户的体验。

④ 全选全不选的操作，也需要进行优化。

点击‘结算’按钮进行路由跳转（未登录就跳转到登录页，登录之后回到购物车页面；已登录就跳转到订单提交页面），传递订单等相关信息。

2.3.4 模态对话框组件封装思路：

定义组件文件，搭建静态页面，定义组件中所需要的数据，通过 props 定义组件中所需要的外部组件传入进来的数据，通过 watch 监视区域的 id 数据，进行对应城市接口的调用，以及监视外部组件传入进来的数据，组件内部需要绑定下拉框的选中事件，根据区域 id 的变化把获取到的省份数组信息进行遍历。定义一个 js 文件，引入 vue 及组件对象，通过 vue.extend 方法，根据组件对象产生对应的组件构造函数，创建对象，内部设置 init 方法，通过单例模式的方式创建组件实例对象，以及产生真实的 DOM，并且挂载到 body 标签里。需要绑定一个 unmount 方法，从 body 中移除这个组件；还定义了一个 clear 方法，清空内部的数据；也为当前组件对象通过 Object.defineProperty 方法绑定了确认、取消数据对应的属性，最后，为 vue 的原型对象添加一个属性绑定的是一个方法，内部调用 init 方法，也就意味着在其他组件中通过组件实例对象调用原型上的这个方法，组件就会产生，并且在界面当中显示。

单例模式代码：

```
function createObj() {
  var instance = null
  return function (name) {
    if (!instance) {
      instance = new Object()
      instance.name = name
    }
    return instance
  }
}

var getObj = createObj()
var obj1 = getObj('小明')
```

```
var obj2 = getObj('小红')
console.log(obj1,obj2)
console.log(obj1===obj2)
```

2.3.5 项目中的 loading 是怎么实现的？具体怎么实现的？

2.3.6 不停点击按钮不停发请求怎么解决？

2.3.7 单点登陆

2.3.8 token 无感刷新怎么实现

2.3.9 项目中有前台商城，支付功能怎么实现的

2.3.10 做项目时有没有做过断网处理

2.3.11 大文件上传思路

1. 首先通过 element-ui/plus 中的 upload 组件封装一个上传文件的组件，该组件中设置 http-request 自定义上传功能的事件/钩子，内部要限制上传文件的个数及上传文件的大小。然后在该事件/钩子中生成文件的碎片，也就是按照指定的大小对一个大的文件以循环的方式进行切割，使用了数组的 slice 方法，把切割后的每个文件形成的切片文件存储到一个数组中，每个文件的类型是 Blob 类型。然后通过遍历的方式产生切片文件信息对象，该对象中通过切片文件、hash 值数据（包括文件名、hash 值和索引值），形成一个切片文件的名字。
2. 通过 webWorker 开一个分线程，把切片文件数组传进去，然后分线程内部通过遍历文件读取文件内容，以及 md5 的方式计算每个切片文件的 hash 值（因为大量的计算会导致当前页面无法向后执行，所以开启 webWorker 分线程），在确认每个切片文件产生对象信息的同时，还要设置文件上传的进度条。
3. 开始上传切片，遍历切片文件数组，然后调用接口，设置请求地址、上传文件及开启进度条，通过 promise.all 的方式传入切片文件数组，采用定型发送请求的方式，也就是一次性

上传多个小文件。最后，通知服务器文件是否上传成功，当所有的切片文件全部上传成功

后，服务器端根据切片文件的名字中的序号进行文件的合并，切片上传文件成功。

4. 如果采用断点续传，有 2 套方案：

方案 1：通过前端的缓存机制记录已经上传的切片的 hash 值，这种方式有缺陷，一旦更换

浏览器，就无法确认哪些切片是上传成功的。

方案 2：在服务器端保存已经上传的切片文件的 hash 值，浏览器中每次上传文件之前，先

向服务端发送请求，获取已上传的切片的 hash 值进行判断，过滤掉相同 hash 值的切片文件，把

不一样的进行上传即可。

1. 断点续传的前提是文件做了切片处理。

2. 秒传实现：文件上传之前，把当前文件对应的 hash 值发送到服务端，服务端进行大文件的

hash 值的对比，如果一样，直接响应给客户端：提示上传成功。

2.4 封装组件

2.4.1 你自己封装过组件吗，以及二次封装？如果让你封装一个 UI 组件，你会考虑到什么问题？

2.4.2 封装 axios，公司没有自己封装好的，直接使用的么？

2.4.3 项目开发中，特别大项目中，怎么封装抽离复用逻辑？

2.5 项目优化

2.5.1 代码层面的优化：

1. v-if 和 v-show，频繁切换显隐推荐用 v-show，不频繁用 v-if。

2. 如果页面中的表达式比较长或者使用次数比较多，用 computed 进行缓存。

3. 如果使用了大量的数据，可以使用 Object.freeze 冻结数据，这样做可以切断数据的劫持，

节省内存，提高性能。

4. 如果遇到了不涉及用户界面更新操作，可以直接把数据绑定在 `this` 上。
5. 涉及用户界面更新，如果是一些切换操作，可以把组件使用 `keep-alive` 进行缓存。
6. `v-for` 遍历数据时，要设置 `key` 值，`vue` 中避免 `v-for` 和 `v-if` 同时使用。【

`vue2` 中 `v-for` 优先

级 `v-if`，`vue3` 中相反】

7. 需要大量数据渲染的时候，可以采用分页或者虚拟长列表的方式进行优化。
8. 图片懒加载
9. 路由懒加载
10. 第三方的插件，进行按需引入（例如 `element-ui` 按需加载配置）
11. 高频事件的触发，使用防抖或节流（例如购物车商品数量的加减、搜索框搜索、页面窗口大小改变、滚轮、抢购等）
12. 大量事件的使用，可以使用时间委派进行优化（例如前台项目中三级分类导航）

2.5.2 打包工具方面的优化：

1. `vue2` 项目：配置 `webpack` 中的 `splitChunk` 进行代码分割，将组件库单独打包，也可以将 `vue` 相关的库单独打包
2. `vue3` 项目：解决 `vite` 首次打开页面加载过慢的问题，可以配置插件：`vite-plugin-optimize-persist`、`vite-plugin-package-config`（`vite@2.9.1` 之前），新版本已经内置了插件，但是第一次加载的速度相比 `webpack` 还是稍微慢了一点。

2.5.3 网络层面的优化（需要开启服务端的设置）：

1. 开启 `gzip` 进行文件的压缩设置
 - a. `Http` 协议版本的设置

b. 压缩级别的设置

c. 文件类型的设置

```
# 开启服务器实时gzip
gzip on;

# 开启静态gz文件返回
gzip_static on;

# 启用gzip压缩的最小文件，小于设置值的文件将不会压缩
gzip_min_length 1k;

# 设置压缩所需要的缓冲区大小
gzip_buffers 32 4k;

# 设置gzip压缩针对的HTTP协议版本
gzip_http_version 1.0;

# gzip 压缩级别，1-9，数字越大压缩的越好，也越占用CPU时间
gzip_comp_level 7;

# 进行压缩的文件类型。javascript有多种形式。其中的值可以在 mime.types 文件中找到。
gzip_types text/plain application/javascript application/x-javascript text/css application/

# 是否在http header中添加Vary: Accept-Encoding，建议开启
gzip_vary on;
```

2. 开启浏览器的缓存机制（强缓存和协商缓存）

3. 结合 webpack 配置使用 CDN 的方式

4. 升级 Http 协议为 2.0

2.5.4 下拉框里的数据可能会有上万条，如何处理

2.5.5 seo 优化怎么做的

2.5.6 白屏是因为什么

2.5.7 vue 项目的首屏优化

2.5.8 在项目中如果有卡顿的情况，怎么解决和排查

2.5.9 如果一个项目中很多 v-if 假如说 1-10 有十个功能,如何进行一个项目优化

2.5.10 项目久了之后，内存占用会越来越大，启动会特别慢，应该怎么办解决

2.5.11 重复点击优化（除了防抖节流，还有什么方法）

2.5.12 10MB 的图片，通过什么方法压缩到 1MB，减少数据大小

2.6 大屏适配方案

2.6.1 移动端适配方案

1) rem

📌 （1）通过 js 动态设置根标签字体大小

```
const docEl = document.documentElement;
// 设置 1rem = 屏幕宽度 / 100
function setRemUnit() {
  const rem = docEl.clientWidth / 100
  docEl.style.fontSize = rem + 'px'
}
setRemUnit();
// 当浏览器大小调整或后退时，重新更新字体大小
window.addEventListener('resize', setRemUnit)
// 设置浏览器的前进后退事件
window.addEventListener('pageshow', function(e) {
```

```
if (e.persisted) {
  setRemUnit();
}
});
```

❏ （2）根据设计稿来计算元素的 rem 大小问题：设计稿是基于 iPhone 6/7/8 机型设计的，设计稿宽度为 375px，现有元素 100px，
请问转化为 rem 多少？

解决：1rem=3.75px xrem=100px --> x = 100 / 3.75 = 26.667rem

所以计算公式为：设计稿元素大小 / 3.75

```
/* 1rem 等于多少 px */
$unit-rem: 3.75; /* 定义变量 */
/* 定义函数 */
@function px-to-rem($px) {
  @return calc($px / $unit-rem) * 1rem;
}
.container {
  width: px-to-rem(100); /* 通过函数进行计算 */
}
```

❏ （3）通过插件配置自动将 px 转化成 rem 单位

```
const { defineConfig } = require("@vue/cli-service");
module.exports = defineConfig({
  transpileDependencies: true,
  css: {
    loaderOptions: {
      postcss: {
        postcssOptions: {
          // 插件文档: https://www.npmjs.com/package/postcss-pxtorem
          plugins: [
            [
              "postcss-pxtorem",
              {
                rootValue: 3.75, // 1rem 等于多少 px
              }
            ]
          ]
        }
      }
    }
  }
});
```

```
unitPrecision: 3, // 精确到小数点几位
propList: ["*"], // 哪些属性要转 rem 单位
selectorBlackList: [], // 哪些属性不要转 rem 单位
},
],
],
},
},
},
},
});
```

❏ 缺点：

- ❏ 在宽度较大的设备上，整体缩放的太大了。而用户的预期应该是看到更多内容
- ❏ 它的兼容性并不好，依然存在一些问题，尤其是尺寸遇到小数点时，bug 较多

3) viewport

❏ （1）设置 meta 标签即可

```
<!--
width=device-width 让布局视口宽度等于视觉视口宽度
initial-scale=1; maximum-scale=1; minimum-scale=1; user-scalable=no; 为
了初始化缩放系数为 1，同时禁止用户缩放
-->
<meta name="viewport" content="width=device-width; initial-scale=1; maxim
um-scale=1; minimum-scale=1; user-scalable=no;">
```

❏ （2）根据设计稿来计算元素的 vw/vh 大小

问题：设计稿是基于 iPhone 6/7/8 机型设计的，设计稿宽度为 375px，现有元素 100px，
请问转化为 vw 多少？

解决：1vw=屏幕宽度的 1%=375 / 100=3.75px xvw=100px --> x = 100 / 3.75 = 26.667vw

所以计算公式为：设计稿元素大小 / 3.75

❏ 通过插件配置自动将 px 转化成 vw 单位

```
1 const { defineConfig } = require("@vue/cli-service");
```

```

module.exports = defineConfig({
  transpileDependencies: true,
  css: {
    loaderOptions: {
      postcss: {
        postcssOptions: {
          // 插件文档: https://www.npmjs.com/package/postcss-px-to-viewport
          plugins: [
            [
              "postcss-px-to-viewport",
              {
                unitToConvert: 'px',
                viewportWidth: 375, // 设计稿宽度
                unitPrecision: 3, // 精度
                propList: ['*'], // 所有属性都要处理
                viewportUnit: 'vw', // 单位
                fontViewportUnit: 'vw',
                selectorBlackList: [], // 哪些属性不需要处理
              },
            ],
          ],
        },
      },
    },
  },
});

```

2.6.2 1px 像素框

window.devicePixelRatio 可以获取像素大小的 dpr 比, PC 端是 1,移动端是 2/3/4...

```

.border-b {
  position: relative;
}
.border-b::after {
  content: "";
  position: absolute;
  bottom: 0;
}

```

```

left: 0;
width: 100%;
height: 1px;
background-color: #000;
}
/* dpr 是 2 的时候 y 缩放 0.5 */
@media screen and (-webkit-min-device-pixel-ratio: 2) {
.border-b::after {
transform: scaleY(0.5);
}
}
@media screen and (-webkit-min-device-pixel-ratio: 3) {
.border-b::after {
transform: scaleY(0.33333);
}
}
@media screen and (-webkit-min-device-pixel-ratio: 4) {
.border-b::after {
transform: scaleY(0.25);
}
}
/* 在下面的权重高,不能换位置 */

```

可以采用 rem 或 viewport 适配, 以及 scale 缩放适配。

scale 缩放适配主要是通过 css 的 scale 属性, 根据屏幕的大小, 对图表进行整体的等比缩放, 从而达到自适应效果。判断屏幕尺寸的比例, 达到页面的全屏展示内容占满显示屏, 比如: 当屏幕的尺寸比例刚好是 16:9 的时候, 页面占满显示器; 如果比例小于 16:9, 页面上下留白, 左右占满, 并上下居中, 显示比例保持 16:9; 如果比例大于 16:9 时, 页面左右留白, 上下占满并居中, 显示比例保持 16:9。也就是应该先确定设计稿的大小, 有宽度和高度, 然后基于宽和高进行缩放。

2.6 针对 word、excel、pdf 等文档处理, 所用到的 插件:

使用了一个 html2canvas 的插件和 jsPDF 两个插件, 前者是 js 库, 可以将 HTML 渲染为 canvas

元素，也就是将页面中的图像、文本、SVG 都可以转换为像素图像，就可以在页面上进行内容的截图、生成 pdf 文件，进行预览实现。后者是一个在客户端用来生成 pdf 文档的 js 库，可以用来创建和下载 pdf 文档，不需要使用服务器，或者是第三方服务，该插件功能很强大，有很多扩展性的插件，可以添加水印、导入导出、生成二维码和条形码等操作。

excel 表格导入导出：

依赖 js-xlsx 还依赖 file-saver 和 script-loader

word 文档导入导出。插件：

```
npm install file-saver
npm install docxtemplater-image-module-free
npm install docxtemplater
npm install pizzip5 npm install jszip-utils
```

2.7 diff 算法的理解

目的：更快的渲染 DOM 树。

原理：对比新旧 DOM 中的节点及 key 的比较。

具体流程：在更新操作的时候，最终会调用一个 update children，内部首先定义 8 个变量，分别是：旧前指针（旧树开始前的索引位置），旧前节点，旧后指针，旧后节点，新前指针，新前节点，新后指针，新后节点。内部首先通过循环的方式，判断旧前指针是否小于等于旧后指针，并且新前指针是否小于等于新后指针，进行节点和 key 的对比。首先判断旧前节点和旧后节点是否有意义，有意义才做比较。先判断旧前节点和新前节点是否一致，如果一致要进一步对比里面的子节点，并且移动旧前指针+。（向不一致前指针+1。然后再判断旧后节点和新后节点是否一致，如果一致，依然要对内部的子节点进一步比较，同时旧后指针和新后指针同时向前移动一位，继续判断旧前节点和新后节点是否一致，如果一致，把旧前节点插入到新后节点对应索引位置的旧后节点的后面，同时移动旧前指针到下一位，新后指针向前一位，再继续

比较旧后节点和新前节点是否一致，如果一致，把旧后节点移动到新前节点的索引位置对应的旧前节点的前面，如果节点都不一样，那就比较 key，首先要判断旧 key 是否存在，然后判断新节点树中的这个 key 在旧树中所有 key 形成的数组中是否存在，如果不在，就删除旧树中当前正在做对比的节点内容，创建新的节点内容插入到这个位置。

以上就是 diff 算法的一个流程，采用的是两侧向中间双向对比的算法来实现 DOM 更新计算。

2.9 项目中遇到哪些问题及解决方案

2.10 项目非技术问题

2.10.1 公司这么久，就做这几个项目？

2.10.2 项目啥时候做的，上架了么

2.10.3 一直问项目里面的具体操作是项目经理安排的任务还是谁安排的，你做登录注册花费了几天，你到那是新项目还是已经搭建好了的

2.10.4 若依框架了解过吗

2.10.5 项目数量，离职原因，人员比例

2.10.6 有没有带组经验

2.10.7 上一家公司主要从事什么业务

2.10.8 详细分享简历中一个项目，这个项目是从 0 开始的吗

2.10.9 组长分配给这个项目什么模块，项目最开始要上线吗

2.10.10 个人负责的模块中技术难点，技术难点怎么解决，解决思路或解决方案

2.10.11 为什么要把技术难点交给我，这个项目人员分配，项目周期是多少

2.10.12 大概有这些:项目迭代周期，

2.10.13 每次迭代需求多少个，你负责多少个，

2.10.14 每次上线你负责的需求有多少 bug

2.10.15 说一下产品每次迭代有多少需求，每次上线有多少 bug 产生，怎么和产品后端沟通

2.10.16 怎么跟项目组的人相处

- 2.10.17 后台系统开发周期多长？全部是你独立完成的吗？
- 2.10.18 后台项目给那些人用，对应模块哪种类型人员
- 2.10.19 工作流程
- 2.10.20 开发人员和项目经理是否在一个工作台上
- 2.10.21 怎样收到测试测出的 bug
- 2.10.22 上一家公司的话团队是有几个人？人员分配
- 2.10.23 对于之前的项目有什么心得，或者是有什么难，包括难点，包括心得体会。
- 2.10.24 然后你们那个项目管理用的什么工具？就代码管理 Git 的常用命令你知道吗
- 2.10.25 然后你们平时项目的整个开发流程能简单的介绍一下吗？比如产品出了一个需求，然后你到你们这里，你们需要做一些什么？
- 2.10.26 开了需求评审之后到你开发这个里面的步骤你可以详细介绍一下
- 2.10.27 你和里面和后端的一些接口，接口的一些对接是通过什么方式？
- 2.10.28 那你们联调阶段是有有联调阶段吗？还是在开发阶段就直接有分的这么细吗？
- 2.10.29 你之前的那个公司的话，你主要是团队组成是怎样子的？
- 2.10.30 那你们那边的话主要分工的话是怎么进行一个分工？如说你们怎么去日常的话是怎么去安排你的一些前端的一些需求的？你和你的前端的话又是怎么进行一个分工

2.10.31 你简单说一下，你在上一家公司，在团队里面充当的一个角色，然后话你所负责的模块有哪些？

2.10.32 相对来说亮一点方面的话，可能没有说特别的突出，那譬如说你觉得你在项目之前那个项目里面，你觉得比较有挑战性的一个需求，你是怎么去实现它的？最后话你是如何去解决它的？

2.10.33 线上部署项目做哪些配置知道么？最后输入的文件是什么？dist 文件夹如果文件比较多,怎么处理,有没有做过分包的处理？

2.10.34 有没有了解 nginx？

2.10.35 你工作三年多,对项目的产品设计这个概念有什么理解？

2.10.36 作为一个前端开发人员,你觉得你最大的优势是什么？

2.10.37 平时开发用的什么开发工具,前一家公司研发同事有多少人,一个部门多少人

2.10.38 平时做项目的时候怎么管理需求

2.10.39 代码提交测试发布的流程，项目发布管理的流程

2.10.40 测试通过后各种环境，上线一整个流程

2.10.41 怎么托管项目

2.10.42 怎么和后端对接

2.10.43 功能提出需求，到发布，到测试，到发布生产的流程

2.10.44 本地会和开发进行连调吗，开发会提供接口地

2.10.45 平时做项目的时候怎么管理需求

2.10.46 代码提交测试发布的流程，项目发布管理的流程

2.10.47 测试通过后各种环境，上线一整个流程

- 2.10.48 本地会和开发进行联调吗，开发会提供接口地址吗
- 2.10.49 本地调用服务测试后怎么发布的流程
- 2.10.50 提测用什么工具
- 2.10.51 测试提出 bug，修改 bug 的流程
- 2.10.52 改完 bug 后发布测试的流程
- 2.10.53 线上出 bug 后怎么快速定位 bug
- 2.10.54 有什么代码规范
- 2.10.55 代码 codereview 有人管理代码提交审核吗
- 2.10.56 有发布吗，知道测试环境吗
- 2.10.57 生产环境有吗
- 2.10.58 后端给的接口地址一般是什么
- 2.10.59 本地怎么测
- 2.10.60 后端会给什么后台地址
- 2.10.61 后端给的地址调不通，怎么处理
- 2.10.62 服务地址有对应的测试环境，生产环境的地址，怎么区别测试环境和生产环境
- 2.10.63 你认为什么是好的代码习惯
- 2.10.64 有没有作品可以看的
- 2.10.65 技术更新迭代这么快，你怎么保持一个持续学习的动力，有没有关注技术的发展

第 3 章 Vue3 后台项目

3.1 项目介绍

3.1.1 说说流程

3.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块

3.2 特定功能实现

3.2.1 登陆实现思路

- 1、先通过 `element-plus + sass` 搭建登录组件，然后进行路由组件的注册。
- 2、通过 `Swagger` 测试接口，使用 `ts` 中的 `interface` 定义该接口返回值的数据类型。
- 3、通过枚举 `enum` 【枚举类型】定义登录相关的接口地址。
- 4、封装登录相关【包括登录、获取用户信息、退出】的 `api` 接口函数。
- 5、在 `pinia` 中拆分出一个用户信息相关的模块，引入登录相关的 `api` 接口函数，定义用来存储用户信息的相关状态数据，定义登录相关的 `action` 【包括登录、获取用户信息、退出】，登录组件内部先进行表单规则的定义及验证表单的操作，通过标识设置密码框是否明文，为登录按钮绑定点击事件，在对应的回调函数内部，进行所有表单的验证，通过后，取出对应的账号密码，分发登录的 `action` 传入账号密码，如果失败，设置提示信息，并且 `finally` 中关闭加载效果【即控制加载标识】，如果登录成功，通过程式路由跳转到首页，然后进入到全局前置导航守卫中，判断 `token` 信息是否存在，如果不存在：判断跳转的路径是不是白名单（登录）中的地址，如果是，放行到登录界面，如果不是，跳转到登录页面，同时记录要跳转的目标路由地址，如果 `token` 存在，判断要跳转的路径是否是登录界面的地址，如果是，控制路由跳转到首页，如果不是，判断用户信息是否存在，存在则直接放行，如果用户信息不存在，在请求头

中嵌入 token，调用获取用户信息的 action，扩展【路由鉴权的实现】。接口调用失败，则重置用户信息并跳转到登录界面，如果接口调用成功，直接放行到目标路由地址即可。

3.2.2 平台属性管理思路

考虑到三级分类组件要在不同的组件中使用，所以抽取出来变成一个全局组件，采用 el-form 组件还有 el-select 组件，以及 sass 进行界面搭建，通过封装三级分类的 pinia，内部的一级二级三级分类 id 及数组数据进行状态数据的定义。并且通过计算属性计算出一二三级分类的 id 及数组数据的计算，并且封装一级分类数据、二级分类数据、三级分类数据，然后在组件内部通过计算属性针对一级分类的 id 的计算需要设置 get、set，get 中获取一级分类 id，set 中根据设置一级分类 id 的数据分发获取二级分类的 action，下拉框组件中通过 v-model 绑定一级分类 id，通过 v-for 指令遍历一级分类数组数据，并且绑定 key、label 及 value 属性，这样在组件加载的时候，先获取一级分类数据，并且展示。当选中的分类内容发生变化后，通过计算属性的 set 方法立刻获取二级分类信息数据，并且清空二级分类 id，三级分类 id，以及三级分类数组数据；二级分类数据选中，同上。

在平台属性管理组件内部，通过 el-button、el-table 组件搭建平台属性界面，并且在该组件内部，针对三级分类的 id 进行监听操作，根据一二三级分类 id 的变化获取不同分类信息下的平台属性数据，也就是监听三级分类 id，调用平台属性的接口。在实现平台属性删除操作时，调用了气泡确认框组件 Popconfirm，并且在该组件内部绑定了一个 confirm 事件，内部调用删除平台属性的 api 接口函数传入 id，删除操作后，设置提示信息，并且进行刷新操作。

前两年通过 vue2 框架+element ui 组件库开发 xxx 后台管理项目的时候，也用到了这个气泡确认框组件，当时我印象中好像应该是 2.13.0 版本，confirm 事件的写法是 onConfirm【DOM 的操作方式】，当然，现在已经没有这个 bug 了。（我也看了看 element ui 的源码。。。。）

修改平台属性的时候，为修改按钮绑定了点击事件，切换主界面和修改界面，采用的是 v-show

指令传入标识的方式来实现，并且通过深拷贝（Object.assign）的方式传入了平台属性对象信息，使用 el-table 组件，绑定平台属性对象中的属性值对象数据进行展示，每一行的属性值都有编辑和查看模式的操作，所以使用了作用域插槽的方式进行占位，内部使用 span 标签，展示属性值，使用 el-input 组件用来编辑这一行的属性值，依然使用 v-if 指令用来切换 span 和 el-input 的显示和隐藏，默认情况下，显示的是 span 标签及属性值内容，为 span 标签绑定点击事件，内部设置标识为 true，展示 el-input 组件标签，并且使用了 nextTick 方法，让文本框自动获取焦点，目的是为了用户进行编辑模式的时候，自动获取焦点，有一个更好的体验。文本框失去焦点后，设置标识为 false，文本框隐藏，span 标签显示，为了让用户有一个更好的体验，span 标签从行内元素变成了行内块元素，并且设置宽度为 100%。

3.2.3 路由鉴权

路由是一种映射关系，是地址和组件的关系。

当路由注册后，可以实现的是浏览器的地址栏输入地址，回车后，页面展示对应组件。

为了实现路由鉴权，先把最基本的登录、首页和 404 三种路由拆分成默认路由（也可称为静态路由、常量路由、基本路由、普通权限路由），进行路由的注册，其他的路由再次进行拆分，拆分成动态路由（也可称为异步路由、高级权限路由），和任意路由（也可称为追加路由、错误信息路由），也就是只有默认路由被注册了，这样任何的用户都可以看到登录界面，登录后可以看到首页及 404 界面。

在获取用户信息的时候，先把用户信息中的路由名字标识数组存储到 pinia 中，然后引入路由器对象，对静态路由数组、动态路由数组、任意路由对象进行路由的过滤操作。也就是定义一个函数，传入路由名字标识数组和动态路由数组，内部调用 filter 方法+递归的方式进行路由名字对比后的过滤，把过滤后的路由对象组成一个新数组，通过路由器对象调用 addRoute 方法，把

过滤后的每个路由对象二次（再次）进行路由注册，并且要把任意路由也要注册，同时把所有的路由对象形成数组存储到 pinia 中（任意路由不要）。

此时用户登录成功后，在浏览器的地址栏中输入地址，可以看到该用户所拥有的权限对应的路由组件内容，但是，这种方式用户体验不好，而且左边菜单栏也不展示权限对应的菜单内容，所以在路由跳转到首页的过程中，

layout 组件内部 sidebar 组件在展示的时候把 pinia 中存储的路

由对象数组获取到，并且通过 v-for 指令进行遍历，还要绑定对应的路由对象的 title 属性和路由地址，最终页面显示完毕后，该用户就可以看到他所拥有权限对应的菜单内容，用户点击菜单，就可以看到对应的组件内容了。

以上是路由鉴权实现的思路。

路由鉴权的目的是：控制用户对应的角色可以看到哪些菜单。

3.2.4 按钮鉴权

依然是在获取用户信息的时候，把用户信息中按钮名字标识数组存储到 pinia 中，然后通过 vue.directive 方法，定义一个全局的指令，并且在该方法的第二个参数配置对象中，mounted 生命周期中获取 pinia 中用户信息模块中存储的按钮名字标识数组，判断该数组中是否有使用指令的时候传入进来的标识，如果没有，通过使用这个指令的标签的父级节点调用 removeChild 移除这个标签，所有的组件中的操作按钮都要使用这个全局的自定义指令传入相关的标识，可以实现按钮的权限操作。

按钮鉴权的目的是：控制用户对应的角色可以看到哪些组件界面有哪些操作按钮，组件界面有哪些按钮显示隐藏。

3.2.5 项目中有没有做过断网处理

3.2.6 vue3 项目的兼容问题

3.2.7 vue3 中常的组件通信方式和 vue2 的有什么不一样

3.2.8 自定义指令的应用场景

3.2.9 登录鉴权权限控制流程

3.3 项目遇到的最大问题及解决方案（项目中让你成长的点）

3.4 项目优化

3.4.1 Vue3 运行项目很慢,怎么解决?

第4章 小程序项目

4.1 项目介绍

4.1.1 说说流程

4.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块

4.2 人员配置

4.3 特定功能实现

4.3.1 小程序做过分包吗？遇到过主包太大无法上传的问题吗？

4.3.2 小程序上线时怎么解决 ios 和安卓不适配

4.3.3 小程序登陆功能

4.3.4 小程序支付功能实现流程

4.4 项目优化

4.5 项目遇到的最大问题及解决方案（项目中让你成长的点）

第5章 react 项目

5.1 项目介绍

5.1.1 说说流程

5.1.2 具体介绍一下项目，项目是如何分工的，介绍一下你负责的模块

5.2 特定功能实现

5.3 项目遇到的最大问题及解决方案（项目中让你成长的点）

5.4 项目优化

第 6 章 算法题（LeetCode）

6.1 基本算法

6.1.1 冒泡排序

冒泡排序是一种简单的排序算法。

它的基本原理是：重复地扫描要排序的数列，一次比较两个元素，如果它们的大小顺序错误，就把它交换过来。这样，一次扫描结束，我们可以确保最大（小）的值被移动到序列末尾。这个算法的名字由来，就是因为越小的元素会经由交换，慢慢“浮”到数列的顶端。

```
function bubbleSort(array) {  
  // 1.获取数组的长度  
  var length = array.length;  
  // 2.反向循环, 因此次数越来越少
```

```

for (var i = length - 1; i >= 0; i--) {
// 3.根据 i 的次数, 比较循环到 i 位置
for (var j = 0; j < i; j++) {
// 4.如果 j 位置比 j+1 位置的数据大, 那么就交换
if (array[j] > array[j + 1]) {
// 交换
// const temp = array[j+1]
// array[j+1] = array[j]
// array[j] = temp
[array[j + 1], array[j]] = [array[j], array[j + 1]];
}
}
}
return arr;
}

```

6.1.2 快速排序

快速排序的基本思想：通过一趟排序，将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

快排应用了分治思想，一般会用递归来实现。

6.1.3 选择排序

```

function selectSort(array) {
// 1.获取数组的长度
var length = array.length

// 2.外层循环: 从 0 位置开始取出数据, 直到 length-2 位置
for (var i = 0; i < length - 1; i++) {
// 3.内层循环: 从 i+1 位置开始, 和后面的内容比较
var min = i
for (var j = min + 1; j < length; j++) {
// 4.如果 i 位置的数据大于 j 位置的数据, 记录最小的位置
if (array[min] > array[j]) {
min = j
}
}
}
}

```

```
}  
if (min !== i) {  
  // 交换  
  [array[min], array[i]] = [array[i], array[min]];  
}  
}  
  
return arr;  
}
```

6.1.4 插入排序

```
function insertSort(array) {  
  // 1.获取数组的长度  
  var length = array.length  
  
  // 2.外层循环: 外层循环是从 1 位置开始, 依次遍历到最后  
  for (var i = 1; i < length; i++) {  
    // 3.记录选出的元素, 放在变量 temp 中  
    var j = i  
    var temp = array[i]  
  
    // 4.内层循环: 内层循环不确定循环的次数, 最好使用 while 循环  
    while (j > 0 && array[j - 1] > temp) {  
      array[j] = array[j - 1]  
      j--  
    }  
  
    // 5.将选出的 j 位置, 放入 temp 元素  
    array[j] = temp  
  }  
  return array  
}
```

第 7 章 场景题

7.1 弹性盒实现 332 或者 331 布局, 怎么让第三排的盒子对齐左侧(就是三阶魔方没有第九个块呈现出的效果)

第 8 章 面试说明

8.1 面试过程最关键的是什么?

- (1) 大大方方的聊, 放松
- (2) 体现优势, 避免劣势

8.2 面试时该怎么说?

1) 语言表达清楚

- (1) 思维逻辑清晰, 表达流畅
- (2) 一二三层次表达

2) 所述内容不犯错

- (1) 不说前东家或者自己的坏话
- (2) 往自己擅长的方面说
- (3) 实质, 对考官来说, 内容听过, 就是自我肯定; 没听过, 那就是个学习的过程。

8.3 面试技巧

8.3.1 六个常见问题

1) 你的优点是什么?

大胆的说出自己各个方面的优势和特长

2) 你的缺点是什么?

不要谈自己真实问题; 用“缺点”衬托自己的优点

3) 你的离职原因是什么?

- 不说前东家坏话, 哪怕被伤过
- 合情合理合法
- 不要说超过 1 个以上的原因

4) 您对薪资的期望是多少?

- 非终面不深谈薪资
- 只说区间，不说具体数字
- 底线是不低于当前薪资
- 非要具体数字，区间取中间值，或者当前薪资的+20%

5) 您还有什么想问的问题？

- 这是体现个人眼界和层次的问题
- 问题本身不在于面试官想得到什么样的答案，而在于你跟别的应聘者的对比
- 标准答案：

公司希望我入职后的 3-6 个月内，给公司解决什么样的问题

公司（或者对这个部门）未来的战略规划是什么样子的？

以你现在对我的了解，您觉得我需要多长时间融入公司？

6) 您最快多长时间能入职？

一周左右，如果公司需要，可以适当提前。

8.3.2 两个注意事项

- (1) 职业化的语言
- (2) 职业化的形象

8.3.3 自我介绍

1) 个人基本信息

2) 工作经历

时间、公司名称、任职岗位、主要工作内容、工作业绩、离职原因。