

面试八股

自我介绍

我叫尹孟佳,2022级齐鲁工业大学软件工程大二学生，本科。在校期间以及学习的过程中做了一些项目，其中包括web端，app端，前后端分离的项目（全栈开发），喜欢学习前端的一些新技术，目前会的技术有:前端三件套，ts，Vue,React,React Native，还有一些UI库，打包工具之类的，node端的话会使用(express,koa框架写一些接口),还有的话就是对于其他方面项目部署，学习过使用docker通过编写dockerfile打包为镜像来进行部署前端项目。

目前写的项目有：Merikle网盘项目(一个Vue项目)，食之韵(一个react native项目但它使用的是expo的裸React Native流来写的一个项目)等等

其他方面呢:平时可能会写一些博客的习惯，在csdn和掘金平台及更新，在GitHub上面来上传自己平时写的项目，因为我刚刚不是说我在一些现有的平台上进行写文章嘛，但是我最近想要搭建一个自己的博客系统和它的后台，在里面进行书写文章，目前我使用Nextjs来进行前后端开发，也是对这一方面进行学习吧，部署的话就准备使用Vercel来部署更加方便些。

在校期间:最近参加了蓝桥杯吧，二等奖但是挺靠前的，普通话二甲，在校获得过二等奖学金，过了英语四级。

怎么学习前端的

我是转专业到软件工程专业的，也是对前端技术的热爱吧，大一的时候就学了前端三件套，接着大一下的暑假吧学了vue，跟着视频做了一个电商平台的项目，然后在做项目的时候学了typescript，然后就是一些打包工具vite嘛就是跟着项目来进行学习如何配置这个文件的，接着加了一个计算机社团，其中开始和他们一起学习前端更多技术，比如原子化css，node后端，开始使用hexo搭建自己的博客，还有前端的一些工程化因为可能会涉及到多人协作，所以也知道了git，github，gitee托管自己的代码，eslint，prettier来格式化统一的代码。大二就是这个学期准备参加了计算机设计大赛，然后我想要做一个app端的项目也是给自己一个学习新的技术的机会吧，了解了app端的很多实现方案，但最终准备使用react native技术来实现，通过翻阅文档解决问题，同时在项目中也学习了react的语法，如果可能的话，准备以后走react技术栈。这就是我的学习过程。

实习时间

6个月是可以的，因为我要升大三了，转专业后的课程都补完了，并且没有考研的打算，所以准备早点实习，并且有更多时间去实习，而且我相信实习时间长一些也会让自己学习到更多知识。

软件工程理解

以下是**软件工程**的一些关键概念和领域：

1. **需求分析**：理解和记录用户需求，以确保开发出的软件满足用户的期望。这通常涉及与客户和利益相关者的沟通，以及编写需求文档。
2. **系统设计**：将需求转化为一个详细的设计方案，包括系统架构、数据模型、界面设计等。
3. **编程和实现**：根据设计方案编写代码。选择合适的编程语言和开发工具也是这个阶段的重要部分。
4. **测试**：确保软件的质量和可靠性。测试可以分为单元测试、集成测试、系统测试和验收测试等。
5. **维护**：软件发布后，可能需要进行错误修复、性能优化和功能更新等维护工作。
6. **项目管理**：包括时间管理、资源管理、风险管理和团队管理，以确保项目按时、按预算完成。
7. **版本控制**：使用版本控制系统（如Git）来管理代码的不同版本和开发分支。

8. **软件过程模型**：常见的软件开发模型包括瀑布模型、迭代模型、敏捷开发（如Scrum、Kanban）等。这些模型提供了开发流程的框架。
9. **文档编写**：编写技术文档、用户手册和维护文档，以便于后续的维护和使用。
10. **质量保证（QA）**：通过各种技术和流程，确保软件的高质量和高可靠性。
11. **用户体验（UX）设计**：关注用户界面的设计和用户体验的优化，以提高用户满意度。

Proxy和Object.defineProperty的区别？

Proxy和Object.defineProperty都可以用来实现JavaScript对象的响应式，但是它们有一些区别：

实现方式：Proxy是ES6新增的一种特性，使用了一种代理机制来实现响应式。而Object.defineProperty是在ES5中引入的，使用了getter和setter方法来实现。

作用对象：Proxy可以代理整个对象，包括对象的所有属性、数组的所有元素以及类似数组对象的所有元素。而Object.defineProperty只能代理对象上定义的属性。

监听属性：Proxy可以监听到新增属性和删除属性的操作，而Object.defineProperty只能监听到已经定义的属性的变化。

性能：由于Proxy是ES6新增特性，其内部实现采用了更加高效的算法，相对于Object.defineProperty来说在性能方面有一定的优势。

综上所述，虽然Object.defineProperty在Vue.js 2.x中用来实现响应式，但是在Vue.js 3.0中已经采用了Proxy来替代，这是因为Proxy相对于Object.defineProperty拥有更优异的性能和更强大的能力。

讲一讲pinia

Pinia 是 Vue 官方团队成员专门开发的一个全新状态管理库，并且 Vue 的官方状态管理库已经更改为了 Pinia。在 Vuex 官方仓库中也介绍说可以把 Pinia 当成是不同名称的 Vuex 5，这也意味不会再出 5 版本了。

优点：

更加轻量级，压缩后提交只有1.6kb。

完整的 TS 的支持，Pinia 源码完全由 TS 编码完成。

移除 mutations，只剩下 state、actions、getters。

没有了像 Vuex 那样的模块镶嵌结构，它只有 store 概念，并支持多个 store，且都是互相独立隔离的。

当然，你也可以手动从一个模块中导入另一个模块，来实现模块的镶嵌结构。

无需手动添加每个 store，它的模块默认情况下创建就自动注册。

支持服务端渲染（SSR）。

支持 Vue DevTools。

讲一讲Vuex

每一个 Vuex 应用的核心就是 store（仓库），它包含着你的应用中大部分的状态 (state)。

状态管理有5个核心：state、getter、mutation、action、module。

State

- 1、单一状态树，定义应用状态的默认初始值，页面显示所需的数据从该对象中进行读取。
- 2、Vuex 使用单一状态树，用一个对象就包含了全部的应用层级状态。它便作为一个“唯一数据源”而存在。这也意味着，每个应用将仅仅包含一个 store 实例。
- 3、单一状态树让我们能够直接地定位任一特定的状态片段，在调试的过程中也能轻易地取得整个当前应用状态的快照。
- 4、不可直接对 state 进行更改，需要通过 Mutation 方法来更改。

5、由于 Vuex 的状态存储是响应式的，从 store 实例中读取状态最简单的方法就是在计算属性中返回某个状态：

Getter

- 1、可以认为是 store 的计算属性，对 state 的加工，是派生出来的数据。
- 2、就像 computed 计算属性一样，getter 返回的值会根据它的依赖被缓存起来，且只有当它的依赖值发生改变才会被重新计算。
- 3、可以在多组件中共享 getter 函数，这样做还可以提高运行效率。
- 4、在 store 上注册 getter，getter 方法接受以下参数：
state, 如果在模块中定义则为模块的局部状态
- 5、getters, 等同于 store.getters

Mutation

- 1、Vuex中store数据改变的唯一方法就是mutation
- 2、通俗的理解，mutations 里面装着改变数据的方法集合，处理数据逻辑的方法全部放在 mutations 里，使数据和视图分离。

Action

action 类似于 mutation，不同在于：

- 1、action 提交的是 mutation，通过 mutation 来改变 state，而不是直接变更状态。
- 2、action 可以包含任意异步操作。

Module

- 1、由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。
- 2、为了解决以上问题，Vuex 允许我们将 store 分割成模块（module）。每个模块拥有自己的 state、mutation、action、getter、甚至是嵌套子模块——从上至下进行同样方式的分割

vuex

是状态管理工具,项目中有一些公共的方法,数据啥的多个地方使用.

有state,getters,mutations,actions,modules.

state:组件中的data,存放数据

getters:组件的computed

mutations:组件的methods,进行的是同步的

actions:提交mutations,把方法卸载mutations里面但是actions里面就是提交mutation,可以进行任何的异步操作

modules:将以上4个属性再细分,仓库更好管理

modules的使用:

 image-20240622235934062

页面的使用

 image-20240623000053234

单向数据流还是双向数据流

单向数据流,不能在本组件中直接改数据,要通过mutations里面进行修改state数据

vuex持久化存储


1. 需要通过插件来进行持久化存储:vuex-persist
2. localStorage实现

watch和watchEffect的区别?

`watch` 和 `watchEffect` 都是监听器, `watchEffect` 是一个副作用函数。它们之间的区别有:

- `watch` : 既要指明监视的数据源, 也要指明监视的回调。
- 而 `watchEffect` 可以自动监听数据源作为依赖。不用指明监视哪个数据, 监视的回调中用到哪个数据, 那就监视哪个数据。
- `watch` 可以访问 改变之前和之后 的值, `watchEffect` 只能获取 改变后 的值。
- `watch` 运行的时候 不会立即执行, 值改变后才会执行, 而 `watchEffect` 运行后可 立即执行。这一点可以通过 `watch` 的配置项 `immediate` 改变。
- `watchEffect` 有点像 `computed` :
 - 但 `computed` 注重的是计算出来的值 (回调函数的返回值), 所以必须要写返回值。
 - 而 `watcheffect` 注重的是过程 (回调函数的函数体), 所以不用写返回值。

html和css

 image-20240625015714284

html

DOCTYPE标签

是HTML5中的一种标准通用的标记语言的文档声明, 告诉浏览器是什么样的文档类型来进行渲染, 不同的渲染模式会影响浏览器对CSS代码甚至JS脚本解析, 必须声明在HTML文档第一行。

渲染模式分为两种:CSS1Compat(标准模式): 浏览器使用W3C的标准来解析页面、BackCompat(怪异模式)

script和link标签的加载顺序

当浏览器解析到这一句的时候会暂停其他资源的下载和处理, 直至将该资源加载, 编译, 执行完毕, 图片和框架等元素也是如此, 类似于该元素所指向的资源嵌套如当前标签内, 这也是为什么要把放在底部而不是头部。

`<link href="common.css" rel="stylesheet"/>` 当浏览器解析到这一句的时候会识别该文档为css文件, 会下载并且不会停止对当前文档的处理, 这也是为什么建议使用link方式来加载css而不是使用@import。

link 一般引入css样式

```
<link rel="stylesheet" href="css/index.css" />
```

link引入强调次序

script 一般引入的是 JAVASCRIPT 脚本

```
<script src="js/jquery-1.11.3.js"></script>
```

因此, 引入jQuery, 只能用script引入而不能用link引入

link用来存外部css的链接, script存放js代码。

加载顺序和优化建议

1. **CSS优先**: 为了避免页面闪烁或无样式内容闪现(FOUC), 应优先加载CSS文件。因此, `<link>` 标签通常放在 `<head>` 中。
2. **JavaScript延迟加载**: 如果JavaScript不是必须在页面加载时立即执行, 可以将 `<script>` 标签放在 `<body>` 底部, 或者使用 `async` 或 `defer` 属性。
3. **关键资源优先**: 对于关键的CSS和JavaScript, 可以考虑内联这些资源, 减少HTTP请求次数, 并确保关键资源的加载优先级。

iframe有了解过吗

iframe概念: 会创建包含另一个文档的内联框架。

优点:

1. 用来加载速度较慢的内容(广告)
2. 可以使脚本并行下载
3. 实现跨子域通信

缺点:

1. iframe会阻塞主页面的load事件
2. 无法被一些搜索引擎识别
3. 会产生很多页面, 不宜管理

BFC和触发条件

- 内部的盒子会在垂直方向上一个接一个的放置
- **对于同一个BFC的俩个相邻的盒子的margin会发生重叠, 与方向无关。**
- 每个元素的左外边距与包含块的左边界相接触(从左到右), 即使浮动元素也是如此
- **BFC的区域不会与float的元素区域重叠**
- **计算BFC的高度时, 浮动子元素也参与计算**
- **BFC就是页面上的一个隔离的独立容器, 容器里面的子元素不会影响到外面的元素, 反之亦然**

触发BFC方法:

- 根元素, 即HTML元素
- 浮动元素: float值为left、right
- overflow值不为 visible, 为 auto、scroll、hidden
- display的值为inline-block、inltable-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid

- position的值为absolute或fixed

//在p上面包一个容器，添加一些触发BFC的东西，可以防止高度塌陷

```
<style>
  .wrap {
    overflow: hidden;// 新的BFC
  }
  p {
    color: #f55;
    background: #fcc;
    width: 200px;
    line-height: 100px;
    text-align:center;
    margin: 100px;
  }
</style>
<body>
  <p>Haha</p >
  <div class="wrap">
    <p>Hehe</p >
  </div>
</body>
```

清除内部浮动

```
<style>
  .par {
    border: 5px solid #fcc;
    width: 300px;
  }

  .child {
    border: 5px solid #f66;
    width:100px;
    height: 100px;
    float: left;
  }
</style>
<body>
  <div class="par">
    <div class="child"></div>
    <div class="child"></div>
  </div>
</body>
```

image-20240709201212409

而BFC在计算高度时，浮动元素也会参与，所以我们可以触发.par元素生成BFC，则内部浮动元素计算高度时候也会计算

```
.par {  
  overflow: hidden;  
}
```

 image-20240709201236357

```
<style>  
  body {  
    width: 300px;  
    position: relative;  
  }  
  
  .aside {  
    width: 100px;  
    height: 150px;  
    float: left;  
    background: #f66;  
  }  
  
  .main {  
    height: 200px;  
    background: #fcc;  
  }  
</style>  
<body>  
  <div class="aside"></div>  
  <div class="main"></div>  
</body>
```

 image-20240709201452180

```
.main {  
  overflow: hidden;  
}
```

 image-20240709201512376

src和href区别

src是替换当前元素，href是用于当前文档和引用的资源之间建立联系

src:指向外部资源，指向的内容将会嵌入到文档中当前标签所在的位置，请求src资源的时候会将其指向的资源下载并应用到文档中。

href是Hypertext Reference的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加

那么浏览器会识别该文档为css文件，就会并行下载资源并且不会停止对当前文档的处理。这也是为什么建议使用link方式来加载css，而不是使用@import方式。

script的defer和async

meta标签

meta标签是由 `name` 和 `content` 属性定义，包含网页文档的属性。

常见的meta标签: `charset`:描述html文档的编码结构、`keywords`:网页关键词、`description`:网站的描述、`viewport`: 适配移动端

img的title和alt

这个title就是鼠标滑动的时候显示，alt是当图片不显示的时候显示的内容。

行内元素和块元素

块级元素: `div`、`p`、`h1-h6`、`ul`、`ol`、`li`、`dl`、`dt`、`dd`

行内元素: `span`、`a`、`b`、`img`、`input`、`strong`

空元素:`br`、`hr`、`img`、`input`、`link`、`meta`

语义化标签

`header`、`footer`、`aside`、`main`、`section`、`nav`、`article`

优点:SEO,代码结构清晰

给你一个html的页面怎么优化

1. CSS放在页面最上部，JavaScript放在页面最下部
2. 减少对dom操作
3. 最好使用语义化代码
4. 使用meta标签
5. 图片的alt一定要写
6. 设置favicon.ico
7. 增加首屏必要的 CSS 和 JS

页面如果需要等待所依赖的 JS 和 CSS 加载完成才显示，则在渲染过程中页面会一直显示空白，影响用户体验，建议增加首屏必要的 CSS 和 JS，比如页面框架背景图片或者 loading 图标，内联在 HTML 页面中。这样做，首屏能快速显示出来，相对减少用户对页面加载等待过程。（比如新浪微博 M 站页面框架）

CSS

对盒子模型的理解

介绍组成盒子的四部分，怪异盒子和标准盒子，通过 `box-sizing` 控制

css选择器和优先级

选择器:id、类、标签、后代、子代、同胞、属性、伪类、伪元素

优先级:内联 > id > 类 > 标签

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

- 如果存在内联样式, 那么 $A = 1$, 否则 $A = 0$
- B的值等于 ID选择器出现的次数
- C的值等于 类选择器 和 属性选择器 和 伪类 出现的总次数
- D 的值等于 标签选择器 和 伪元素 出现的总次数

可以继承的:

文本系列、字体系列、元素可见性、表格、列表、引用、光标

不可以继承的:

 image-20240710071545186

css居中

水平垂直居中:

1. 定位 + margin:auto
2. 定位 + margin:负值
3. 定位 + transform
4. flex
5. grid

6.

```
<style>
    .father{
        width:500px;
        height:300px;
        border:1px solid #0a3b98;
        position: relative;
    }
    .son{
        width:100px;
        height:40px;
        background: #f0a238;
        position: absolute;
        top:0;
        left:0;
        right:0;
        bottom:0;
        margin:auto;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

```
7. <style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
        margin-left: -50px;
        margin-top: -50px;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

```
8. <style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

css隐藏元素

 image-20240710074945034

如果这个opacity:

如果利用 animation 动画，对 opacity 做变化（animation会默认触发GPU加速），则只会触发 GPU 层面的 composite，不会触发重绘

由于其仍然是存在于页面上的，所以他自身的事件仍然是可以触发的，但被他遮挡的元素是不能触发其事件的

需要注意的是：其子元素不能设置opacity来达到显示的效果

特点：改变元素透明度，元素不可见，占据页面空间，可以响应点击事件

其他隐藏:

height、width:0

position

css实现三角形

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    .box{
      width: 0;
      height: 0;
      border-style:solid;
      border-width: 50px 50px;
      border-color: transparent red red transparent;
    }
  </style>
</head>
<body>
  <div class="box"></div>
</body>
</html>
```

有边框的三角形

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    .border {
      width: 0;
      height: 0;
      border-style:solid;
      border-width: 0 50px 50px;
      border-color: transparent transparent #d9534f;
      position: relative;
    }
    .border:after {
      content: '';
      border-style: solid;
      border-width: 0 40px 40px;
      border-color: transparent transparent #96ceb4;
      position: absolute;
      top: 6px;
    }
  </style>
</head>
<body>
  <div class="border"></div>
</body>
</html>
```

```
left: -40px;
}
</style>
</head>
<body>
  <div class="border"></div>
</body>
</html>
```

css的flex布局

- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content

成员属性

- `order`:按照这个值越小就越靠前
- `flex-grow`:上面讲到当容器设为 `flex-wrap: nowrap`; 不换行的时候, 容器宽度有不够分的情况, 弹性元素会根据 `flex-grow` 来决定
定义项目的放大比例 (容器宽度>元素总宽度时如何伸展)
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 (容器宽度<元素总宽度时如何收缩), 默认为 1, 即如果空间不足, 该项目将缩小
- `flex-basis`:设置的是元素在主轴上的初始尺寸, 所谓的初始尺寸就是元素在 `flex-grow` 和 `flex-shrink` 生效前的尺寸
浏览器根据这个属性, 计算主轴是否有多余空间, 默认值为 `auto`, 即项目的本来大小, 如设置了 `width` 则元素尺寸由 `width/height` 决定 (主轴方向), 没有设置则由内容决定
- `flex`: `flex` 属性是 `flex-grow`, `flex-shrink` 和 `flex-basis` 的简写, 默认值为 `0 1 auto`, 也是比较难懂的一个复合属性
- `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。默认值为 `auto`, 表示继承父元素的 `align-items` 属性, 如果没有父元素, 则等同于 `stretch`

css的grid布局

- 项目属性
- 容器属性

容器属性

- `display`
- `grid-template-rows`、`grid-template-columns`, , `repeat` (次数, 宽度), `minmax`(小值, 大值)
- `grid-row-gap`, `grid-column-gap`, `grid-gap`
- `grid-auto-flow`:是容器的子元素会按照顺序, 自动放置在每一个网格。

顺序就是由 `grid-auto-flow` 决定，默认为行，代表"先行后列"，即先填满第一行，再开始放入第二行

- `justify-content`、`align-content`、`place-content`

```
.container {  
  justify-content: start | end | center | stretch | space-around | space-between  
  | space-evenly;  
  align-content: start | end | center | stretch | space-around | space-between |  
  space-evenly;  
}
```

- `justify-items`、`align-items`、`place-items`

```
.container {  
  justify-items: start | end | center | stretch;  
  align-items: start | end | center | stretch;  
}
```

指定网格项目所在的四个边框，分别定位在哪根网格线，从而指定项目的位置

- `grid-column-start` 属性：左边框所在的垂直网格线
- `grid-column-end` 属性：右边框所在的垂直网格线
- `grid-row-start` 属性：上边框所在的水平网格线
- `grid-row-end` 属性：下边框所在的水平网格线

举个例子：

```
<style>  
  #container{  
    display: grid;  
    grid-template-columns: 100px 100px 100px;  
    grid-template-rows: 100px 100px 100px;  
  }  
  .item-1 {  
    grid-column-start: 2;  
    grid-column-end: 4;  
  }  
</style>  
  
<div id="container">  
  <div class="item item-1">1</div>  
  <div class="item item-2">2</div>  
  <div class="item item-3">3</div>  
</div>
```

- `justify-self`、`align-self`、`place-self`

```
.item {
  justify-self: start | end | center | stretch;
  align-self: start | end | center | stretch;
}
```

flex和grid布局区别

flex:

一维布局: Flexbox 主要用于一维布局，可以处理行或列中的项目。

适用于较小组件的布局: 例如导航栏、按钮组、弹性容器等。

grid:

二维布局: Grid 用于二维布局，可以处理行和列中的项目。

适用于较大区域的布局: 例如整个网页的布局、大型组件的排列等。

双栏和三栏布局

双栏布局

1. float + margin-left

```
<style>
  .box{
    overflow: hidden; //添加BFC
  }
  .left {
    float: left;
    width: 200px;
    background-color: gray;
    height: 400px;
  }
  .right {
    margin-left: 210px;
    background-color: lightgray;
    height: 200px;
  }
</style>
<div class="box">
  <div class="left">左边</div>
  <div class="right">右边</div>
</div>
```

给父元素添加BFC来防止下方的元素会跑到上方

1. flex布局

```
<style>
  .box{
    display: flex;
```

```

    }
    .left {
        width: 100px;
    }
    .right {
        flex: 1;
    }
}
</style>
<div class="box">
    <div class="left">左边</div>
    <div class="right">右边</div>
</div>

```

注意的是, `flex` 容器的一个默认属性值: `align-items: stretch;`

这个属性导致了列等高的效果。为了让两个盒子高度自动, 需要设置: `align-items: flex-start`

三栏布局

1. flex
2. grid
3. float + margin
4. absolute + margin
5. 两边使用 float 和负 margin

6.


```

<style type="text/css">
    .wrap {
        display: flex;
        justify-content: space-between;
    }

    .left,
    .right,
    .middle {
        height: 100px;
    }

    .left {
        width: 200px;
        background: coral;
    }

    .right {
        width: 120px;
        background: lightblue;
    }

    .middle {
        background: #555;
        width: 100%;
        margin: 0 20px;
    }
</style>
<div class="wrap">

```

```

<div class="left">左侧</div>
<div class="middle">中间</div>
<div class="right">右侧</div>
</div>

```

7. <style>

```

.wrap {
    display: grid;
    width: 100%;
    grid-template-columns: 300px auto 300px;
}

.left,
.right,
.middle {
    height: 100px;
}

.left {
    background: coral;
}

.right {
    background: lightblue;
}

.middle {
    background: #555;
}
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="middle">中间</div>
    <div class="right">右侧</div>
</div>

```

8. <style>

```

.wrap {
    background: #eee;
    overflow: hidden; <!-- 生成BFC, 计算高度时考虑浮动的元素 -->
    padding: 20px;
    height: 200px;
}

.left {
    width: 200px;
    height: 200px;
    float: left;
    background: coral;
}

.right {
    width: 120px;
    height: 200px;
    float: right;
    background: lightblue;
}

```



```

        .middle {
            margin-left: 220px;
            height: 200px;
            background: lightpink;
            margin-right: 140px;
        }
    </style>
    <div class="wrap">
        <div class="left">左侧</div>
        <div class="right">右侧</div>
        <div class="middle">中间</div>
    </div>

```

9. `<style>`
- ```

.container {
 position: relative;
}

.left,
.right,
.main {
 height: 200px;
 line-height: 200px;
 text-align: center;
}

.left {
 position: absolute;
 top: 0;
 left: 0;
 width: 100px;
 background: green;
}

.right {
 position: absolute;
 top: 0;
 right: 0;
 width: 100px;
 background: green;
}

.main {
 margin: 0 110px;
 background: black;
 color: white;
}
</style>

<div class="container">
 <div class="left">左边固定宽度</div>
 <div class="right">右边固定宽度</div>
 <div class="main">中间自适应</div>
</div>

```

## css单位

绝对单位:px

相对单位:em、rem、vw、vh

px绝对单位，像素，是显示屏上的一个一个的点，是等大的

em、rem、vw、vh

em:是根据父元素的字体大小来变化的转换的，是不固定的，如果当前对象行内没有字体大小会使用浏览器默认的大小:16px

rem:与em不同的是，是HTML根元素来进行字体大小的转换的

vw:相对于视口的宽度，将宽度分为了100份

vh:和vw类似

与%区别:相对于父元素定位

## 伪类/伪元素

## 实现动画

### transition

- property:需要变化的css属性
- duration:完成过渡需要的时间
- timing-function:完成效果的速度曲线
- delay:动画效果的延迟触发时间

 image-20240713000353851

```
<style>
 .base {
 width: 100px;
 height: 100px;
 display: inline-block;
 background-color: #0EA9FF;
 border-width: 5px;
 border-style: solid;
 border-color: #5daf34;
 transition-property: width, height, background-color, border-width;
 transition-duration: 2s;
 transition-timing-function: ease-in;
 transition-delay: 500ms;
 }

 /*简写*/
 /*transition: all 2s ease-in 500ms;*/
 .base:hover {
 width: 200px;
 height: 200px;
 background-color: #5daf34;
 border-width: 10px;
 }
</style>
```

```
 border-color: #3a8ee6;
 }
</style>
<div class="base"></div>
```

## transform

- translate
- scale
- rotate
- skew

一般配合 transition 过度使用,不支持 inline

```
<style>
 .base {
 width: 100px;
 height: 100px;
 display: inline-block;
 background-color: #0EA9FF;
 border-width: 5px;
 border-style: solid;
 border-color: #5daf34;
 transition-property: width, height, background-color, border-width;
 transition-duration: 2s;
 transition-timing-function: ease-in;
 transition-delay: 500ms;
 }
 .base2 {
 transform: none;
 transition-property: transform;
 transition-delay: 5ms;
 }

 .base2:hover {
 transform: scale(0.8, 1.5) rotate(35deg) skew(5deg) translate(15px,
25px);
 }
</style>
<div class="base base2"></div>
```

## animation

```
@keyframes rotate{
 from{
 transform: rotate(0deg);
 }
 to{
 transform: rotate(360deg);
 }
}
```

```
@keyframes rotate{
 0%{
 transform: rotate(0deg);
 }
 50%{
 transform: rotate(180deg);
 }
 100%{
 transform: rotate(360deg);
 }
}
```

定义动画，接着使用动画

```
animation: rotate 2s;
```

image-20240713000916116

## css预编译语言的理解(变量、混合、函数、模块化)

可以让css代码容易复用

### 变量

变量无疑为 Css 增加了一种有效的复用方式，减少了原来在 Css 中无法避免的重复「硬编码」

```
$red = #c00;
strong{
 color:$red;
}
```

```
@red = @c00;
strong{
 color:@red;
}
```

### 作用域

```
$color: black;
.scoped {
 $bg: blue;
 $color: white;
 color: $color;
 background-color:$bg;
}
.unscoped {
 color:$color;
}

//编译后
.scoped {
 color:white;/*是白色*/
 background-color:blue;
```

```
}
.unscoped {
 color:white;/*白色（无全局变量概念）*/
}
```

在 sass 中最好不要定义相同的变量名。

但是 less 和 stylus 中

```
@color: black;
.scoped {
 @bg: blue;
 @color: white;
 color: @color;
 background-color:@bg;
}
.unscoped {
 color:@color;
}
//编译后
.scoped {
 color:white;/*白色（调用了局部变量）*/
 background-color:blue;
}
.unscoped {
 color:black;/*黑色（调用了全局变量）*/
}
```

## 混合

可以将一部分样式抽离出来，作为单独的模块，可以被很多选择器重复使用

在 less 中，混合的用法是指将定义好的 classA 中引入另一个已经定义的 class，也能使用够**传递参数**，参数变量为 @ 声明

```
.alert {
 font-weight: 700;
}

.highlight(@color: red) {
 font-size: 1.2em;
 color: @color;
}

.heads-up {
 .alert;
 .highlight(red);
}
```

scss的mixin

```
//基本用法
@mixin important-text {
 color: red;
}
```

```

font-size: 25px;
font-weight: bold;
border: 1px solid blue;
}
selector {
 @include mixin-name;
}
//想混入传递参数
/* 混入接收两个参数 */
@mixin bordered($color, $width) {
 border: $width solid $color;
}

.myArticle {
 @include bordered(blue, 1px); // 调用混入，并传递两个参数
}

.myNotes {
 @include bordered(red, 2px); // 调用混入，并传递两个参数
}
//设置默认值
@mixin bordered($color: blue, $width: 1px) {
 border: $width solid $color;
}
@mixin sexy-border($color, $width: 1in) {
 border: {
 color: $color;
 width: $width;
 style: dashed;
 }
}
p { @include sexy-border(blue); }
h1 { @include sexy-border(blue, 2in); }
//可变参数
@mixin box-shadow($shadows...) {
 -moz-box-shadow: $shadows;
 -webkit-box-shadow: $shadows;
 box-shadow: $shadows;
}

.shadows {
 @include box-shadow(0px 4px 5px #666, 2px 6px 10px #999);
}

```

## 函数

## 模块化

```

@import './common';
@import './github-markdown';
@import './mixin';
@import './variables';

```

## SCSS继承

```
.button-basic {
 border: none;
 padding: 15px 30px;
 text-align: center;
 font-size: 16px;
 cursor: pointer;
}

.button-report {
 @extend .button-basic;
 background-color: red;
}

.button-submit {
 @extend .button-basic;
 background-color: green;
 color: white;
}

//编译
.button-basic, .button-report, .button-submit {
 border: none;
 padding: 15px 30px;
 text-align: center;
 font-size: 16px;
 cursor: pointer;
}

.button-report {
 background-color: red;
}

.button-submit {
 background-color: green;
 color: white;
}
```

## position

五个属性: `static`、`fixed`、`sticky`、`relative`、`absolute`

`relative`: 相对定位, 通过`top`、`left`、`right`、`bottom`来控制他相对于原来位置进行设置偏移量

`absolute`: 绝对定位, 离着它最近的父级如果有定位的话, 不是`static`, 那么就会以它作为参照对象来进行设置偏移量运动

`fixed`: `fixed` 的参考点会是以浏览器视窗本身, 也就是说不论使用者如何滑动网页, `fixed` 定位的元素会一直维持在同位置。

`sticky`: `sticky` 是一种特殊的相对定位, 在某些情况下元素表现得像是 `fixed` 固定定位, 而在其他情况下仍然保持相对定位。元素在页面滚动到**特定点**时, 才会固定在特定的位置, 但如果页面回滚到元素的原始位置, 元素就会恢复到相对定位。

# 关于颜色的代码

## 重流和重绘

回流:对元素在页面上布局来进行计算尺寸和位置

重绘:当计算好了盒模型的位置, 大小及其其他属性后, 根据盒子特性进行绘制

- 解析HTML, 生成DOM树, 解析CSS, 生成CSSOM树
- 将DOM树和CSSOM树结合, 生成渲染树(Render Tree)
- Layout(回流):根据生成的渲染树, 进行回流(Layout), 得到节点的几何信息 (位置, 大小)
- Painting(重绘):根据渲染树以及回流得到的几何信息, 得到节点的绝对像素
- Display:将像素发送给GPU, 展示在页面上

如何触发这些?

回流触发:

- 元素尺寸
- 元素位置
- 添加或者删除DOM元素
- 页面刚开始渲染的时候
- 浏览器窗口尺寸变化

重绘触发:

- 回流必定会触发重绘
- 颜色的改变
- 阴影的改变
- 文本方向该百年

## 浏览器的优化

浏览器会将修改操作放入到队列里, 直到过了一段时间或者操作达到了一个阈值, 才清空队列

当你获取布局信息的操作的时候, 会强制队列刷新, 包括前面讲到的 `offsetTop` 等方法都会返回最新的数据

因此浏览器不得不清空队列, 触发回流重绘来返回正确的值

## 如何减少

- 如果要改变样式的话, 通过改变元素的 `class` 类名
- 避免使用table布局
- 对于那些复杂的动画, 可以对其设置 `position:fixed/absolute`, 使他们脱离文档流, 减少对其他元素影响
- 使用css3可以让 `transform`、`opacity`、`filters` 这些动画不会引起回流重绘
- 在使用 `JavaScript` 动态插入多个节点时, 可以使用 `DocumentFragment`. 创建后一次插入. 就能避免多次的渲染性能

```
1. const el = document.getElementById('el')
```



```

for(let i=0;i<10;i++) {
 el.style.top = el.offsetTop + 10 + "px";
 el.style.left = el.offsetLeft + 10 + "px";
}
//优化
// 缓存offsetLeft与offsetTop的值
const el = document.getElementById('el')
let offLeft = el.offsetLeft, offTop = el.offsetTop

// 在JS层面进行计算
for(let i=0;i<10;i++) {
 offLeft += 10
 offTop += 10
}

// 一次性将计算结果应用到DOM上
el.style.left = offLeft + "px"
el.style.top = offTop + "px"

```

2.
 

```

const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
//合并样式
<style>
 .basic_style {
 width: 100px;
 height: 200px;
 border: 10px solid red;
 color: red;
 }
</style>
<script>
 const container = document.getElementById('container')
 container.classList.add('basic_style')
</script>

```

3. //离线操作
 

```

let container = document.getElementById('container')
container.style.display = 'none'
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
... (省略了许多类似的后续操作)
container.style.display = 'block'

```

# 说说设备像素、css像素、设备独立像素、dpr、ppi

css像素适合web编程，根据页面缩放比会发生变化，如果页面放大了1倍，那么1px变为了2px，那么原来的320px需要160px就填充充满了。

设备像素:物理像素，出厂的时候就固定的，dt

设备独立像素:独立于设备的逻辑像素。

**一个设备独立像素里可能包含1个或者多个物理像素点，包含的越多则屏幕看起来越清晰**

至于为什么出现设备独立像素这种虚拟像素单位概念，下面举个例子：

iPhone 3GS 和 iPhone 4/4s 的尺寸都是 3.5 寸，但 iPhone 3GS 的分辨率是 320x480，iPhone 4/4s 的分辨率是 640x960

这意味着，iPhone 3GS 有 320 个物理像素，iPhone 4/4s 有 640 个物理像素

如果我们按照真实的物理像素进行布局，比如说我们按照 320 物理像素进行布局，到了 640 物理像素的手机上就会有一半的空白，为了避免这种问题，就产生了虚拟像素单位

**我们统一 iPhone 3GS 和 iPhone 4/4s 都是 320 个虚拟像素，只是在 iPhone 3GS 上，最终 1 个虚拟像素换算成 1 个物理像素，在 iPhone 4s 中，1 个虚拟像素最终换算成 2 个物理像素**

至于 1 个虚拟像素被换算成几个物理像素，这个数值我们称之为设备像素比，也就是下面介绍的 `dpr`

`dpr`:设备像素 / 设备独立像素

当设备像素比为1:1时，使用1（1×1）个设备像素显示1个CSS像素

当设备像素比为2:1时，使用4（2×2）个设备像素显示1个CSS像素

当设备像素比为3:1时，使用9（3×3）个设备像素显示1个CSS像素

`ppi`:每英寸像素，表示每英寸所包含的像素点数目，更确切的说法应该是像素密度。数值越高，说明屏幕能以更高密度显示图像

## 响应式设计，原理和怎么做

响应式设计就是:页面的设计与开发应当根据用户行为以及设备环境(系统平台、屏幕尺寸、屏幕定向等)进行相应的响应和调整

1. 在页面头部有 `meta` 声明 `viewport`

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">
```

2. 实现响应式布局方式:媒体查询、百分比、vw/vh、rem

3. 媒体查询

```
/* 超小屏幕（手机，宽度小于 576px） */
@media (max-width: 575.98px) {
 body {
 background-color: lightblue;
 }
}

/* 小屏幕（平板竖屏，宽度在 576px 到 767px 之间） */
```

```

@media (min-width: 576px) and (max-width: 767.98px) {
 body {
 background-color: lightgreen;
 }
}

/* 中等屏幕（平板横屏，宽度在 768px 到 991px 之间） */
@media (min-width: 768px) and (max-width: 991.98px) {
 body {
 background-color: lightyellow;
 }
}

/* 大屏幕（小型笔记本，宽度在 992px 到 1199px 之间） */
@media (min-width: 992px) and (max-width: 1199.98px) {
 body {
 background-color: lightcoral;
 }
}

/* 超大屏幕（桌面显示器，宽度在 1200px 及以上） */
@media (min-width: 1200px) {
 body {
 background-color: lightpink;
 }
}

```

### 3. 百分比

`height`、`width` 属性的百分比依托于父标签的宽高，但是其他盒子属性则不完全依赖父元素：

- 子元素的`top/left`和`bottom/right`如果设置百分比，则相对于直接非`static`定位(默认定位)的父元素的高度/宽度
- 子元素的`padding`如果设置百分比，不论是垂直方向或者是水平方向，都相对于直接父亲元素的`width`，而与父元素的`height`无关。
- 子元素的`margin`如果设置成百分比，不论是垂直方向还是水平方向，都相对于直接父元素的`width`
- `border-radius`不一样，如果设置`border-radius`为百分比，则是相对于自身的宽度

可以看到每个属性都使用百分比，会照成布局的复杂度，所以不建议使用百分比来实现响应式

### 4. rem

在以前也讲到，`rem`是相对于根元素`html`的`font-size`属性，默认情况下浏览器字体大小为`16px`，此时`1rem = 16px`

可以利用前面提到的媒体查询，针对不同设备分辨率改变`font-size`的值，如下

## CSS优化

1. 内联首屏关键CSS、异步加载CSS、资源加载CSS、合理使用选择器、减少使用昂贵的属性、不适用`@import`
2. 内联首屏关键CSS

打开一个页面，页面首要内容出现在屏幕的时间影响着用户的体验，而通过内联 `css` 关键代码能够使浏览器在下载完 `html` 后就能立刻渲染

而如果外部引用 `css` 代码，在解析 `html` 结构过程中遇到外部 `css` 文件，才会开始下载 `css` 代码，再渲染

所以, CSS 内联使用使渲染时间提前

3.

#### 4. 资源压缩:通过一些打包工具将代码进行压缩

## 5. 合理使用选择器

- 不要嵌套使用过多的复杂选择器，避免超过3层
- 使用id选择器就没必要进行嵌套
- 通配符和属性选择器效率最低，避免使用

## 6. 不要使用@import

因为@import会影响浏览器的**并行下载**，使得页面加载的时候增加额外的延迟。

## 多个@import可能会导致下载乱

比如一个css文件 `index.css` 包含了以下内容: `@import url("reset.css")`

那么浏览器就必须先把 `index.css` 下载、解析和执行后，才下载、解析和执行第二个文件 `reset.css`

其他

- 减少重绘和重排
- 精灵图减少了http请求
- 小的icon图片转化为base64编码
- 了解哪些属性可以继承而来，避免对这些属性重复编写

## 单行/多行溢出

单行

```
<style>
 p{
 overflow: hidden;
 line-height: 40px;
 width:400px;
 height:40px;
 border:1px solid red;
 text-overflow: ellipsis;
 white-space: nowrap;
 }
</style>
<p 这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本这是一些文
本这是一些文本这是一些文本</p >
```

多行:基于行 / 基于高度

```

<style>
 p {
 width: 400px;
 border-radius: 1px solid red;
 -webkit-line-clamp: 2;
 display: -webkit-box;
 -webkit-box-orient: vertical;
 overflow: hidden;
 text-overflow: ellipsis;
 word-wrap: break-word//如果有大量英文字母
 }
</style>
<p>
 这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本
 这是一些文本这是一些文本这是一些文本这是一些文本这是一些文本
</p>

```

## 基于高度

```

<style>
 .demo {
 position: relative;
 line-height: 20px;
 height: 40px;
 overflow: hidden;
 }
 .demo::after {
 content: "...";
 position: absolute;
 bottom: 0;
 right: 0;
 padding: 0 20px 0 10px;
 }
</style>

<body>
 <div class='demo'>这是一段很长的文本</div>
</body>

```

## Chrome支持小于12px方法

1. zoom
2. -webkit-transform:scale()
3. -webkit-text-size-adjust:none
4. zoom其支持的值类型有：
  - zoom:50%，表示缩小到原来的一半
  - zoom:0.5，表示缩小到原来的一半
2. -webkit-transform:scale() 大部分现代浏览器支持，并且对英文、数字、中文也能够生效，缩放不会改变了元素占据的空间大小，页面布局不会发生变化
3. -webkit-text-size-adjust:none

来设定文字大小是否会根据设备(浏览器)来自动调整显示大小

这样设置之后会有一个问题，就是当你放大网页时，一般情况下字体也会随着变大，而设置了以上代码后，字体只会显示你当前设置的字体大小，不会随着网页放大而变大了

所以，我们不建议全局应用该属性，而是单独对某一属性使用

## 总结

`zoom` 非标属性，有兼容问题，缩放会改变了元素占据的空间大小，触发重排

`-webkit-transform:scale()` 大部分现代浏览器支持，并且对英文、数字、中文也能够生效，缩放不会改变了元素占据的空间大小，页面布局不会发生变化

`-webkit-text-size-adjust` 对谷歌浏览器有版本要求，在27之后，就取消了该属性的支持，并且只对英文、数字生效

## 实现视差滚动效果

实现方法:

- background-attachment

值:

- scroll:默认值，背景图片会随着其他部分滚动而滚动
- fixed:当页面的其余部分滚动的时候，背景图片不滚动
- inherit

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Document</title>
 <style>
div {
 height: 100vh;
 background: rgba(0, 0, 0, .7);
 color: #fff;
 line-height: 100vh;
 text-align: center;
 font-size: 20vh;
}

.a-img1 {
 background-image:
url(https://images.pexels.com/photos/1097491/pexels-photo-1097491.jpeg);
 background-attachment: fixed;
 background-size: cover;
 background-position: center center;
}

.a-img2 {
 background-image:
url(https://images.pexels.com/photos/2437299/pexels-photo-2437299.jpeg);
```

```

 background-attachment: fixed;
 background-size: cover;
 background-position: center center;
 }

 .a-img3 {
 background-image:
url(https://images.pexels.com/photos/1005417/pexels-photo-1005417.jpeg);
 background-attachment: fixed;
 background-size: cover;
 background-position: center center;
 }
</style>
</head>
<body>

 <div class="a-text">1</div>
 <div class="a-img1">2</div>
 <div class="a-text">3</div>
 <div class="a-img2">4</div>
 <div class="a-text">5</div>
 <div class="a-img3">6</div>
 <div class="a-text">7</div>
</body>
</html>

```

## 调用三方样式UI库，如何覆盖重写

1. 全局css，将引入的组件库的样式引在自定义的css文件前面，并且选择器的优先级高于组件库的优先级。但通常不推荐，会造成样式冲突
2. 给每个文件样式做一个隔离，vue: scoped, react是css module

react的css module

```

// src/Demo.js
import styles from './demo.module.css';
export default function Demo() {
 return (
 <div className={styles.mywrapper}>
 <Calendar />
 </div>
);
}

```

```

/* src/demo.module.css */
.mywrapper {
 border: 5px solid black;
}

```

被编译后，插入到样式表和元素的class属性都会加上一个哈希值作为命名空间

```
<style>
.demo_mywrapper__Hd9Qg {
 border: 5px solid black;
}
</style>
<div class="demo_mywrapper__Hd9Qg">
 ...
</div>
```

react提供了:global将他生校内的央视当作全局css

vue:scoped

使用深度选择器: >>>、 /deep/、 ::v-deep

```
<style scoped>
.mywrapper>>>
.ant-picker-calendar-full
.ant-picker-panel
.ant-picker-calendar-date-today{
 border-color:purple
}
</style>
<template>
 <div class="mywrapper" >
 <Calendar />
 </div>
</template>
```

# Vue面经

## Vue的理解

官方:是一套用于构建用户界面的渐进式框架，核心库只关注视图层

 image-20240622011436606

### 1.1

#### 声明式框架

用的时候感受不到很复杂的过程，只在意结果。

相当于封装了一层JS，自己只要按照它制定的规则来写就行。

#### 组件系统

当业务越来越大可拆分为组件

每个组件都有着自己的功能，可以随时进行替换

组件的优势:



- 降低了耦合度，可以随时替换组件来完成需求
- 定位错误方便，因为一个系统是由组件
- 提高了可维护性，可复用性

## 路由跳转

vue-router

## 状态管理工具

pinia, vuex

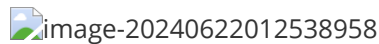
## 构建工具

对代码进行压缩，babel编译，合并，优化处理

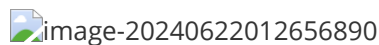
vite和webpack

## 1.2MVVM模式

MVC:



MVVM(Modal,View,ViewModal):



## 1.3 虚拟DOM

用户可能会频繁操作dom，性能会下降，所以vue封装了一层缓存，用户在这个缓存上操作，接着缓存在进行处理来作用到真实dom上

1. diff算法比较差异更新
2. 实现跨平台
3. 减少dom操作

## 1.4 组件化

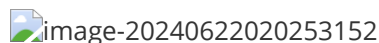
高内聚，低耦合

减低更新范围，只渲染变化的组件

## SPA

spa：单页面应用，就是vue/react中只用一个html文件，通过vue-router/react-router来切换页面，之后打包文件将这个js文件啥的挂载到html，浏览器解析代码，执行js，客户端渲染(CSR)

MPA:SSR



MPA:每个页面都是一个主页面，都是独立的当我们在访问另一个页面的时候，都需要重新加载 `html`、`css`、`js` 文件，公共文件则根据需求按需加载如下图

SPA缺点:SEO,首屏加载慢



# vue为什么需要虚拟dom

---

## 1.1 概念

js对象描述真是dom，直接操作dom比js层效率低，可以将dom变为js对象通过diff算法更新dom。


 image-20240622021500931

 image-20240622021537289

## 组件化

---

1. Vue中的每个组件都有一个渲染函数watcher、effect
2. 数据响应式，数据变化后会执行watcher或者effect
3. 合理划分，不拆分，更新的时候整个页面都要更相信
4. 过多开分导致watcher、effect多，性能降低

## 既然Vue采用了数据拦截可以精确探索数据变化，为什么现需要虚拟DOM进行diff检测差异

因为一旦一个页面数据过多(例如表单很多)，每当一个数据发生了改变就会触发watcher，导致频繁更新，产生很多watcher浪费内存。所以vue采用了将每个组件有一个watcher，这样组件里面的属性发生改变，这个组件就会更新。

diff+watcher

## 响应式

---

vue2使用了defineProperty来给对象的属性添加get和set方法，get是获取值，set是设置属性值

 image-20240622100313054

## 1.2 vue2处理缺陷

- 在vue2的时候使用defineProperty采用数据劫持，需要对属性进行重写添加getter以及setter性能差。
- 当新增属性或者删除属性的时候无法监控变化，如果设置为响应式数据还需要\$set，\$delete实现
- 数组不采用defineProperty来劫持，需要进行单独处理
- 对于ES6的Map和set这些不支持

## 1.3Vue3的Proxy

没有对对象的属性进行重写，而是对对象进行了代理，用B来代理A访问目标对象

代理了很多(get,set...)的过程

 image-20240622102159643

## 怎样检测数组变化

---

vue2在新的方法里面重写了数组的方法

 image-20240622103338847

缺点:数组索引和长度变化无法进行监测

## ref和reactive(之后再写源码方面的)

ref和reactive是vue3数据响应式中非常重要的两个概念

reactive底层采用的new Proxy()

ref采用的是Object.defineProperty()

### 总结

- ref可以存储原始类型，reactive不能
- ref通过.value才能访问，reactive不用

## watch和watchEffect区别

watch是显式地监测一个数据源，当这个数据源发生改变之后会调用用户的自定义的回调函数，能够拿到老值和新值。

watchEffect是写一个函数，通过里面的变量或者属性变化的话来让函数重新调用，不能拿到老值。

## v-show和v-if

v-show:通过css的display:none来控制是否显示

v-if:切换就是dom从渲染树中删除或者显示

- `v-show` 由 `false` 变为 `true` 的时候不会触发组件的生命周期
- `v-if` 由 `false` 变为 `true` 的时候，触发组件的 `beforeCreate`、`create`、`beforeMount`、`mounted` 钩子，由 `true` 变为 `false` 的时候触发组件的 `beforeDestroy`、`destroyed` 方法

性能消耗：`v-if` 有更高的切换消耗；`v-show` 有更高的初始渲染消耗；

使用场景：

如果需要非常频繁地切换，则使用 `v-show` 较好

如果在运行时条件很少改变，则使用 `v-if` 较好

## v-if和v-for优先级

- vue2中v-for优先级>v-if
- vue3中v-if优先级 > v-for

## 生命周期

vue3

基本的生命周期

有：`beforeCreate`、`created`、`beforeMount`、`mounted`、`beforeUpdate`、`updated`、`beforeDestroy`、`destroyed`

一旦进入了页面会执行四个生命周期，如果有keep-alive会加activated和deactivated生命周期，等第一次进入会执行xxx，第二次或者第n次进入该页面只会执行activated，因为进行了缓存。

父组件引入子组件，生命周期执行顺序

```
父组件: beforeCreate、created、beforeMounted
子组件: beforeCreate、created、beforeMounted、mounted
子
父组件: mounted
```

调用接口的时候为什么在created、mounted里面调用，而不能在beforeCreated里面调用？

```
如果请求是methods里面封装好了的，这个方法是不能在beforeCreated阶段里面拿到的。
```

在created里面怎样获取dom？

```
1. 只要异步就可以了。
 例如: setTimeout、请求、Promise
2. 使用nextTick
```

哪个阶段会有\$el,哪个阶段会有\$data

beforeCreate什么也没有

created有\$data

beforeMount,有\$data,

mouted都有



## keep-alive

缓存组件，举一个例子(场景)

## ref和nextTick

ref:通过js获取到dom

nextTick:获取更新后的dom内容



## scoped原理

样式在本组件生效，不影响其他组

原理:给元素节点**新增自定义属性**，然后根据属性选择器添加样式

## 样式穿透

深度选择器：

scss样式穿透:父元素 /deep/ 子元素

stylus:父元素 >>> 子元素

## computed, watch, method

computed和watch:

计算属性是**基于它们的响应式依赖**及进行缓存的。

method每调用一次就会进行调用。

computed和watch:

watch:当监测的东西改变了才会被调用。

computed:可以计算某一个属性的改变,如果某一个值改变了,计算属性会监听返回。

## props和data优先级

props->method=>data->computed

## 解决跨域问题

### vue代理

在vue.config.js中设置代理解决跨域问题

```
devServer: {
 host: 'localhost',
 port: 8080,
 proxy: {
 '/get': {
 target: 'http://localhost:3000', // 要跨域的域名
 changeOrigin: true, // 是否开启跨域
 },
 '/api': {
 target: 'http://localhost:3000', // 要跨域的域名
 changeOrigin: true, // 是否开启跨域
 pathRewrite: { // 重写路径
 '^/api': '/api' // 这种接口配置出来
 }
 }
 }
}
```

http://xx.xx.xx.xx:8083/api/login  
// '^/api': '/' 这种接口配置出来      http://xx.xx.xx.xx:8083/login

### jsonp

### 后端的cors

### websocket

## 打包路径和路由模式

---

**打包路径** './', 改打包路径, publicPath

**路由模式**: hash(#), history,

前端使用hash来测试

项目上线一般使用history

## 代理和环境变量

---

一般开发阶段和上线阶段所使用的url肯定不同

分为开发环境和生产环境.

开发环境: .env.development

生产环境: .env.production

判断的话: process.env.xxx变量

## vue路由

---

分为history和hash

区别:

1. 表现形式不同
2. 跳转请求:
  - history: <http://localhost:3000/id> -> 发送请求
  - hash: 不会发送请求
3. 打包后自测使用hash, 上线使用history会出现空白页

路由守卫:

全局守卫

beforeEach beforeResolve afterEach

路由守卫

beforeEnter

组件内

beforeRouteEnter beforeRouteUpdate beforeRouteLeave

## v-modal双向绑定

---

劫持数据发生改变, 数据发生改变就会触发update方法更新节点内容

## 怎样减少首屏渲染时间

---

## diff算法

# JS面经

## 常考的基础JS题

 image-20240623020812117

## ES6 语法糖 class 代码 原理

原理还是构造函数和原型链的一层封装而已

```
"use strict";

/**
 * 定义属性
 * @param {*} target
 * @param {array} props
 */
function _defineProperties(target, props) {
 for (let i = 0; i < props.length; i++) {
 let descriptor = props[i];
 descriptor.enumerable = descriptor.enumerable || false;
 descriptor.configurable = true;
 if ("value" in descriptor) descriptor.writable = true;
 Object.defineProperty(target, descriptor.key, descriptor);
 }
}

/**
 * 给构造函数添加属性/方法
 * @param {*} Constructor 构造函数
 * @param {array} protoProps 原型属性 - 添加到原型对象上
 * @param {array} staticProps 静态属性 - 直接添加到构造函数本身上
 * @returns
 */
function _createClass(Constructor, protoProps, staticProps) {
 if (protoProps) _defineProperties(Constructor.prototype, protoProps);
 if (staticProps) _defineProperties(Constructor, staticProps);
 return Constructor;
}

let Person = (function () {
 function Person(name, age) {
 // 1. 判断构造函数是否是通过new操作符调用，是的话this为Person实例，不是的话this为
 undefined,
 if (!(this instanceof Person)) {
 throw new TypeError("Cannot call a class as a function");
 }
 this.name = name;
 this.age = age;
 }
})
```

```
// 2. 给构造函数添加属性/方法，第2个参数添加原型上的属性/方法，第3个参数添加形态属性/方法
_createClass(
 Person,
 [
 {
 key: "say",
 value: function say() {
 console.log("hello!");
 },
 },
],
 [
 {
 key: "run",
 value: function run() {
 console.log("run!");
 },
 },
],
);

// 3. 返回新的构造函数
return Person;
})();

Person.prototype.say(); // hello!
Person.run(); // run!
```

Class 的 `constructor` 编译后本质还是一个构造函数

- 首先判断了 Class 的调用方式，要求必须使用 `new` 调用
- 然后通过 `_createClass` 方法区分了原型上的方法和静态方法
- 最后通过 `_defineProperties` 方法进行属性/方法的添加

## 修改dom耗时？为什么

### promise意义（回调地狱）

js不是单线程的嘛，所以当处理那些耗时的程序的时候把他们放到异步任务当中。

Promise就是来解决这种问题的，将一些异步请求啥的使用Promise来包裹起来，当任务解决了之后再释放请求。

而且解决这个异步任务还有一个发展史:回调函数(会出现回调地狱)=>Promise=>async, await

Promise三个状态:pending,fulfilled,rejected

Promise还有一些实例方法:Promise.all,Promise.race,Promise.resolve:返回一个状态已变成 resolved 的 Promise 对象。Promise.reject

`Promise` 的 `.then` 或者 `.catch` 可以被调用多次,但如果 `Promise` 内部的状态一经改变,并且有了一个值,那么后续每次调用 `.then` 或者 `.catch` 的时候都会直接拿到该值。(见3.5)



`.then` 或者 `.catch` 中 `return` 一个 `error` 对象并不会抛出错误，所以不会被后续的 `.catch` 捕获。(见3.6)

`.then` 或 `.catch` 返回的值不能是 `promise` 本身，否则会造成死循环。(见3.7)

`.then` 或者 `.catch` 的参数期望是函数，传入非函数则会发生值透传。(见3.8)

`.then` 方法是能接收两个参数的，第一个是处理成功的函数，第二个是处理失败的函数，再某些时候你可以认为 `catch` 是 `.then` 第二个参数的简便写法。(见3.9)

`.finally` 方法也是返回一个 `Promise`，他在 `Promise` 结束的时候，无论结果为 `resolved` 还是 `rejected`，都会执行里面的回调函数。

```
Promise.resolve(1)
 .then(2)
 .then(Promise.resolve(3))
 .then(console.log)
//结果为1

Promise.resolve()
 .then(function success (res) {
 throw new Error('error!!!')
 }, function fail1 (err) {
 console.log('fail1', err)
 }).catch(function fail2 (err) {
 console.log('fail2', err)
 })
```

由于`Promise`调用的是`resolve()`，因此`.then()`执行的应该是`success()`函数，可是`success()`函数抛出的是一个错误，它会被后面的`catch()`给捕获到，而不是被`fail1`函数捕获。

```
async function async1 () {
 console.log('async1 start');
 await new Promise(resolve => {
 console.log('promise1')
 })
 console.log('async1 success');
 return 'async1 end'
}

console.log('script start')
async1().then(res => console.log(res))
console.log('script end')
'script start'
'async1 start'
'promise1'
'script end'

async function async1 () {
 await async2();
 console.log('async1');
 return 'async1 success'
}

async function async2 () {
 return new Promise((resolve, reject) => {
 console.log('async2')
 reject('error')
 })
}
```

```
}
}
async1().then(res => console.log(res))
//如果在async函数中抛出了错误，则终止错误结果，不会继续向下执行。
'async2'
Uncaught (in promise) error
```

## 回调和闭包

## 运行机制

## 原型和原型链

## this指向

## new内部做了什么

### 1. 创建一个空对象：

- 首先，创建一个新的空对象。这通常是一个纯净的对象，类似于 `{}`。

### 2. 设置原型链：

- 将这个新创建的对象的原型（`__proto__`）设置为构造函数的 `prototype` 属性。这一步确保新对象能够继承构造函数的原型方法和属性。

### 3. 绑定 this：

- 将构造函数的 `this` 绑定到新创建的对象上。然后执行构造函数的代码。这一步允许构造函数为新对象添加属性和方法。

### 4. 返回对象：

- 如果构造函数显式地返回一个对象，那么这个对象将作为整个 `new` 表达式的结果返回。
- 如果构造函数没有显式地返回对象，则返回新创建的对象。

## js中如何实现继承

1. 原型链继承
2. 盗用构造函数继承
3. 原型式继承
4. 组合继承
5. 寄生组合继承

## 普通函数和箭头函数的区别

## 继承的几种方式

原型链继承但是有缺点就是在一个实例上更改了原型上的内容，会影响其他实例。解决方案盗用构造函数。

但是构造函数缺点：

综合一下就是使用组合式继承。

原型式继承

寄生继承

组合寄生继承

## DOM事件流和事件委托

## Promise和async、await

<https://segmentfault.com/a/1190000016788484#item-2-8>

## new内部做了什么

### 1. 创建一个空对象：

- 首先，创建一个新的空对象。这通常是一个纯净的对象，类似于 `{}`。

### 2. 设置原型链：

- 将这个新创建的对象的原型（`__proto__`）设置为构造函数的 `prototype` 属性。这一步确保新对象能够继承构造函数的原型方法和属性。

### 3. 绑定 this：

- 将构造函数的 `this` 绑定到新创建的对象上。然后执行构造函数的代码。这一步允许构造函数为新对象添加属性和方法。

### 4. 返回对象：

- 如果构造函数显式地返回一个对象，那么这个对象将作为整个 `new` 表达式的结果返回。
- 如果构造函数没有显式地返回对象，则返回新创建的对象。

## 如果递归多了会出现什么问题，该怎么解决

会导致程序运行速度变慢、堆栈溢出、数据混乱

递归栈溢出。

解决办法：

- 1、用循环进行替换
- 2、限制递归次数
- 3、用非递归方法实现

我们需要知道除了执行栈可以通过存放函数外，还有一个地方可以存放函数，那就是异步任务队列中，该队列一般用于存储 异步任务的回调函数，等待执行栈只有全局执行上下文时，就会因为Event Loop而被JS主线程取出放入执行栈中执行。

这样就可以达到将递归函数从执行栈中转移到异步队列中，从而减轻执行栈压力的目的。

```
function sum(total, i) {
 if(i === 0) { // 递归结束条件
 return console.log(total)
 }
 setTimeout(sum, 0, total+i, i-1) // 注意这里给sum直接传参会触发sum立即执行，而不是回调执行
}

sum(0, 10)
```

 image-20240709093040709

我们可以让 执行栈先存入部分递归函数调用，当快要到执行栈临界值时，我们就将下次递归函数调用交给异步任务队列，这样就能达到清空执行栈的目的，然后等待4ms左右再将递归函数加入执行栈执行，等到快要到临界值时，再加入异步任务队列以清空执行栈

```
function sum(total, i) {
 if(i === 0) { // 递归结束条件
 return console.log(total)
 }
 if(i%1000 === 0) {
 setTimeout(sum, 0, total+i, i-1)
 } else {
 sum(total+i, i-1)
 }
}
```

当前是将递归函数通过setTimeout加入异步宏任务队列，但是加入异步宏任务队列有几个缺点

- 1、setTimeout将递归函数加入异步宏任务队列，需要等待至少4ms，有延迟
- 2、一次事件轮询只能从异步宏任务队列中取出一个递归函数来执行，事件轮询也需要时间

如果将递归函数加入异步微任务队列，就可以解决该缺点

- 1、加入异步微任务队列没有时间延迟
- 2、一次事件轮询就可以处理异步微任务队列中所有递归函数

```
function sum(total, i) {
 if(i === 0) { // 递归结束条件
 return console.log(total)
 }
 if (i%1000 === 0) {
 Promise.resolve().then(sum.bind(null, total+i, i-1))
 } else {
 sum(total+i, i-1)
 }
}
```

，我们后台程序会打印这个结点对应的结点值；若没有，则返回对应编程语言的空结点即可。

```
/*class ListNode {
 * val: number
 * next: ListNode | null
 * constructor(val?: number, next?: ListNode | null) {
 * this.val = (val===undefined ? 0 : val)
 * this.next = (next===undefined ? null : next)
 * }
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 * @param pHead ListNode类
 * @return ListNode类
 */
export function EntryNodeOfLoop(pHead: ListNode): ListNode {
 // write code here
 const set = new Set()
 while(pHead){
 //第一次出现重复的节点就是入口点
 if(set.has(pHead)){
 return pHead
 }else{
 set.add(pHead)
 pHead = pHead.next
 }
 }
 return null
}
```

## LocalStorage和sessionStorage

### localStorage

1. **用户设置**：如主题选择、语言偏好等，在用户重新访问时仍然保留。
2. **持久登录状态**：保存用户的登录令牌，免去每次访问都需要重新登录。
3. **购物车**：保存用户选择的购物车内容，即使在浏览器关闭后再次打开也能恢复。

### sessionStorage

1. **单一会话数据**：如表单填写过程中的临时数据，在标签页关闭后清除。
2. **多步骤表单**：保存当前步骤数据，防止页面刷新时数据丢失。
3. **会话状态**：跟踪用户在单一会话期间的状态和活动数据。

## 防抖和节流

### 防抖

搜索框输入：用户停止输入一定时间后再进行搜索请求。

窗口调整：用户停止调整窗口大小一定时间后再触发调整事件。

### 节流

滚动事件：在用户滚动时每隔一段时间触发一次，减少滚动处理的频率。

按钮点击：防止按钮在短时间内被多次点击，导致多次触发事件。

## 拓展运算符好处和set, map解构

**简洁性**：扩展运算符使得代码更加简洁，避免了使用 `Array.prototype.concat` 或 `Object.assign` 等冗长的语法。

**可读性**：代码的可读性提高，使得开发者更容易理解操作的意图。

**浅拷贝**：扩展运算符进行的是浅拷贝，对于原始数据类型（如数字、字符串）是直接复制，对于复杂数据类型（如对象、数组）则复制引用。

复制，合并，函数参数

set, map解构就是转化为数组形式。

## set是否支持forEach循环

支持。

## cookie中能设置token吗

能、不设置cookie有效期、重新登录重写cookie覆盖原来的cookie。

回答：

token一般是用来判断用户是否登录的，它内部包含的信息有：uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign（签名，token的前几位以哈希算法压缩成的一定长度的十六进制字符串）token可以存放在Cookie中，token是否过期，应该由后端来判断，不该前端来判断，所以token存储在cookie中只要不设置cookie的过期时间就ok了，如果token失效，就让后端在接口中返回固定的状态表示token失效，需要重新登录，再重新登录的时候，重新设置cookie中的token就行。

## 什么时候自动添加cookie



请看上面标红的三个属性，拿一个Http POST请求来说 <http://aaa.www.com/xxxxx/list>

如果满足下面几个条件：

- 1、浏览器端某个Cookie的domain字段等于aaa.www.com或者www.com
- 2、都是http或者https，或者不同的情况下Secure属性为false
- 3、要发送请求的路径，即上面的xxxxx跟浏览器端Cookie的path属性必须一致，或者是浏览器端Cookie的path的子目录，比如浏览器端Cookie的path为/test，那么xxxxxxx必须为/test或者/test/xxxx等子目录才可以

## 判断当前脚本运行在浏览器上

```
let isBrowser = typeof window !== "undefined" && ({}).toString.call(window) === "[object window]";

let isNode = typeof global !== "undefined" && ({}).toString.call(global) === "[object global]";
```

## 造成内存泄漏情况

### 1. 隐式的全局变量

```
function foo(arg) {
 bar = "this is a hidden global variable";
}
```

### 2. 没有清除定时器

### 3. 游离DOM的引用

```
var elements = {
 button: document.getElementById("button")
};
function doStuff() {
 button.click();
}
function removeButton() {
 // button 是 body 的子节点.
 document.body.removeChild(document.getElementById("button"));
 // 因为 elements 对象中缓存了 DOM 节点引用，这里我们始终有对 id 是 button 的引用
}
```

### 4. 闭包

只要变量被任何一个闭包使用了，就会被添到词法环境中，被该作用域下所有闭包共享。这是闭包引发内存泄漏的关键。

## Reflect和Proxy区别

`Reflect` 和 `Proxy` 都是 ECMAScript 2015 (ES6) 引入的两个全新对象，它们提供了对 JavaScript 对象的更强大的操作和控制能力。尽管它们有一些重叠的功能，但它们的设计目的和使用方式存在显著差异。以下是它们的主要区别：

### Reflect

`Reflect` 对象提供了一组静态方法，这些方法与对象的内部方法一一对应。换句话说，`Reflect` 提供了一种更为函数化和一致的方式来操作对象。`Reflect` 的方法与常见的对象操作非常相似，但通常提供了更好的返回值和错误处理机制。

### 主要用途：

- 提供与对象内部方法对应的函数化接口。
- 提高代码的可读性和一致性。
- 改善错误处理和返回值（例如，大多数方法返回布尔值以指示操作是否成功）。

### Proxy

`Proxy` 对象用于定义自定义行为，以在基本操作（如属性查找、赋值、枚举、函数调用等）时拦截和重新定义这些操作。`Proxy` 是一个更为强大的工具，因为它允许你拦截和自定义几乎所有的对象操作。

### 主要用法：

```
const handler = {
 get: function(target, prop, receiver) {
 console.log(`Getting ${prop}`);
 return prop in target ? target[prop] : 42;
 },
 set: function(target, prop, value, receiver) {
 console.log(`Setting ${prop} to ${value}`);
 target[prop] = value;
 return true;
 }
};

const target = {};
const proxy = new Proxy(target, handler);

proxy.a = 1; // Setting a to 1
console.log(proxy.a); // Getting a \n 1
console.log(proxy.b); // Getting b \n 42
```

### 主要用途：

- 拦截和自定义对象操作（如属性访问、赋值等）。
- 实现某种设计模式（如虚拟属性、观察者模式等）。
- 代理和验证数据。
- 创建更复杂和动态的对象行为。

### 总结

- `Reflect` 提供了一组静态方法，主要用于简化和标准化对象操作。
- `Proxy` 允许你拦截和自定义几乎所有的对象操作，提供了更强大的控制能力。

两者可以结合使用，`Reflect` 的方法常常在 `Proxy` 的捕捉器函数中用于执行默认行为，从而使自定义操作更为简洁和明确。



# 计算机网络

## 基础

### 说一下计算机系统体系结构

OSI七层，TCP/IP四层，五层体系结构



说一下OSI七层模型

- 应用层:最靠近用户的层，负责处理特定的应用程序细节。
- 表示层:确保从一个系统发送的信息可以被另一个系统的应用层读取。负责数据的转换、压缩和加密
- 会话层:管理用户的会话，控制网络上两节点间的对话和数据交换的管理。
- 传输层:提供端到端的通信服务，保证数据的完整性和正确性。
- 网络层:负责在多个网络之间进行传输，确保数据能在复杂的网络结构中找到从远到目的地最佳路径
- 数据链路层:在物理连接中提供可靠的传输，负责建立和维护两个相邻结点间的链路。
- 物理层:负责在物理媒介中实现原始的数据传输。

说一下TCP/IP四层

- 应用层:HTTP,FTP,SMTP

例子：当一个浏览器输入了url到网页上，浏览器使用HTTP协议从web服务器请求页面。

- 传输层:TCP和UDP
- 网际层:IPv4、IPv6。负责在不同网络之间路由数据包，提供逻辑地址（IP 地址）和网络寻址功能。用于处理数据包的分组、转发和路由选择，
- 网络接口层：以太网、Wi-Fi

### 说一下每一层对应的网络协议有哪些



### 数据在各层是怎么传输的

对于发送方来说，从上层到下层层层包装，对于接收方来说，从下层到上层要层层解开包装。

- 发送方的应用进程向接收方的应用进程传送数据
- AP 先将数据交给本主机的应用层，应用层加上本层的控制信息 H5 就变成了下一层的数据单元
- 传输层收到这个数据单元后，加上本层的控制信息 H4，再交给网络层，成为网络层的数据单元
- 到了数据链路层，控制信息被分成两部分，分别加到本层数据单元的首部（H2）和尾部（T2）
- 最后的物理层，进行比特流的传输



# 网络综合

---

## 从浏览器地址栏输入url到显示主页的过程？

### websocket和socket区别

socket是一个网络编程的标准接口，而websocket是应用层的通信协议

socket是在传输层的，对TCP/IP进行了高度封装，屏蔽了网络实现细节，方便开发者更好地进行网络编程

websocket是一种持久化的协议，伴随h5实现的，用来解决http不能持久连接的问题，为浏览器和服务器的全双工通信提供了标准化解决方案。

应用场景:WebSocket 适合处理实时性要求较高且频率较高的数据传输，如在线游戏，视频直播，通知提醒等；而 Socket 则主要应用于服务器之间的通信以及大吞吐量的数据交换场景，如消息队列和数据同步。

### 说一下你了解的端口及其对应的服务

80:HTTP超文本传输协议。

443:HTTPS

1080:Sockets

3306:MYSQL默认端口号

22:SSH

53:DNS域名解析服务

端口号的作用：来区分不同的服务，因为一台主机可以提供很多服务，如果只有一个IP就无法区分网络服务了，所以采用"IP + 端口号"

## HTTP

---

### HTTP状态码及其含义

1xx:临时对话，客户端继续发送请求

2xx：请求已经成功被服务器接收

3xx: 重定向

4xx：请求可能出现了错误

5xx：服务端在尝试处理请求的时候出现了错误

200:请求成功

301：请求资源永远移动，已经分配了一个新的URL

302：请求资源临时移动，临时分配了一个URL

304：缓存

400：请求的语法错误

401: 当前请求没有认证

403: 请求的资源被服务器拒绝

404: 在服务器上找不到请求资源

405:方法不允许

500: 服务器端在执行请求时发生了错误

## HTTP请求方式有哪些

常见的GET、POST、DELETE、PUT

HEAD:获取报文的首部, 与GET相比, 不返回报文主体部分

OPTIONS: 询问支持的请求方法, 用于跨域请求

CONNECT:要求与代理服务器通信时建立隧道, 使用隧道来进行TCP通信

TRACE:回显服务器收到的请求, 主要用于测试和诊断。

## HTTP的GET方法可以实现写操作吗

可以, 但是不推荐, 会造成跨站请求伪造(后面来介绍(CSRF))

## GET和POST区别

1. GET是将请求信息放在了URL上面, POST是将请求信息放在了请求体上, 因为这方面的区别导致GET携带请求信息是有限的, 因为URL有长度限制, 但是POST没有限制。而且放在URL上有些不安全
2. GET符合幂等和安全性, POST不符合, GET只是请求服务器上的资源, 不会修改数据库的内容, 但是POST会修改。
3. 缓存方面, 对于GET请求来说, GET请求能够被缓存, GET请求能保存到浏览器记录中,但是POST请求是不能的。

## GET请求的长度限制

URL本身对请求是没有限制的, 真正受到限制的是因为浏览器。

对于IE浏览器限制字符2000, Firefox最大限制65536个字符, Chrome是8182个字符

针对的是整个URL。

## 请求头content-type的类型有哪些

### application/json

- 表示请求或响应的主体是JSON格式的数据。

### application/x-www-form-urlencoded

- 表示请求主体使用URL编码的形式发送, 通常用于HTML表单提交。

### multipart/form-data

- 通常用于文件上传, 可以包含多个部分(例如文本字段和文件字段), 每个部分有自己的Content-Type。

## text/plain

- 纯文本格式，适合传输普通文本内容。

## application/xml

- 表示请求或响应的主体是 XML 格式的数据

# UDP

---

## TCP和UDP有什么不同？

## UDP协议为什么不可靠

UDP 在传输数据之前**不需要先建立连接**，远地主机的运输层在接收到 UDP 报文后，不需要确认，提供不可靠交付。总结就以下四点：

- 不保证消息交付：不确认，不重传，无超时
- 不保证交付顺序：不设置包序号，不重排，不会发生队首阻塞
- 不跟踪连接状态：不必建立**连接或重启状态机**
- 不进行拥塞控制：不内置客户端或网络反馈机制

## DNS使用了UDP

当进行**区域传送（主域名服务器向辅助域名服务器传送变化的那部分数据）**时会使用 TCP，因为数据同步传送的数据量比一个请求和应答的数据量要多，而 TCP 允许的报文长度更长，因此为了保证数据的**正确性**，会使用基于可靠连接的 TCP。

当客户端想 **DNS 服务器查询域名（域名解析）**的时候，一般返回的内容不会超过 UDP 报文的**最大长度**，即 512 字节，用 UDP 传输时，不需要创建连接，从而大大提高了响应速度，但这**要求域名解析服务器和域名服务器都必须自己处理超时和重传从而保证可靠性**。

## UDP 不一定比 TCP 快的原因

1. **网络条件**：在不可靠的网络条件下，UDP 的数据包丢失率高，可能导致需要重发数据包，从而增加延迟。
2. **应用需求**：某些应用需要数据的可靠性和有序性，如果用 UDP 传输，还需要在应用层实现重传和排序逻辑，这可能导致更高的复杂度和延迟。
3. **拥塞控制**：TCP 有拥塞控制机制，在网络拥塞时可以自动调整传输速度，而 UDP 在拥塞时可能导致严重的数据包丢失。
4. **重传机制**：TCP 有内置的重传机制，而 UDP 没有。如果 UDP 数据包丢失，需要应用层处理重传，这增加了复杂度和潜在的延迟。

主要是因为UDP的不可靠性，导致需要在应用层重新进行设置。

## TCP为什么要三次握手，挥手却要四次

- **三次握手**：三次握手是为了确保双方都能确认对方的存在，并且能够协商初始序列号。第一次和第二次握手确保客户端和服务端都知道对方的存在，第三次握手则是客户端确认收到服务器的应答，完成连接的建立。
- **四次挥手**：四次挥手是因为TCP连接是全双工的，双方都需要单独关闭发送方向的数据流。第一次和第二次挥手用于关闭客户端到服务器的数据流，第三次和第四次挥手用于关闭服务器到客户端的数据流。这个过程确保了双方都有机会完成各自的剩余数据传输。

## TCP撤销后，客户端都断开连接了嘛

在TCP连接的撤销过程中，当所有的**四次挥手完成并且TIME\_WAIT状态结束后**，客户端和服务端都会断开连接并释放相应的资源。因此，最终的结果是客户端和服务端都断开了连接。

## IP

### IP协议的定义和作用

IP 协议（Internet Protocol）用于在计算机网络之间传输数据包，它定义了数据包的格式和处理规则，确保数据能够从一个设备传输到另一个设备，可能跨越多个中间网络设备（如路由器）。

IP的作用

1. 寻址:每一个连接在网络上的设备都有IP，IP使用这些地址来为数据包设置源地址和目标地址，确保数据包能准确地发送到目标设备上
2. 路由:IP 协议负责决定数据包在网络传输中的路径。比如说路由器使用路由表和 IP 地址信息来确定数据包的最佳传输路径。
3. 分片和重组:当数据包过大的时候，IP就会对它进行分片，接受方会根据这个头部信息来进行重组数据包。

### IP地址有哪些分类

IP是由网络号和主机号组成的。

1. **网络号**：它标志主机所连接的网络地址表示属于互联网的哪一个网络。
2. **主机号**：它标志主机地址表示其属于该网络中的哪一台主机。

### 域名和 IP 的关系？一个 IP 可以对应多个域名吗？

一个域名可以对应多个 IP，但这种情况 DNS 做负载均衡的，在用户访问过程中，一个域名只能对应一个 IP。

而一个 IP 却可以对应多个域名，是一对多的关系。

# 操作系统

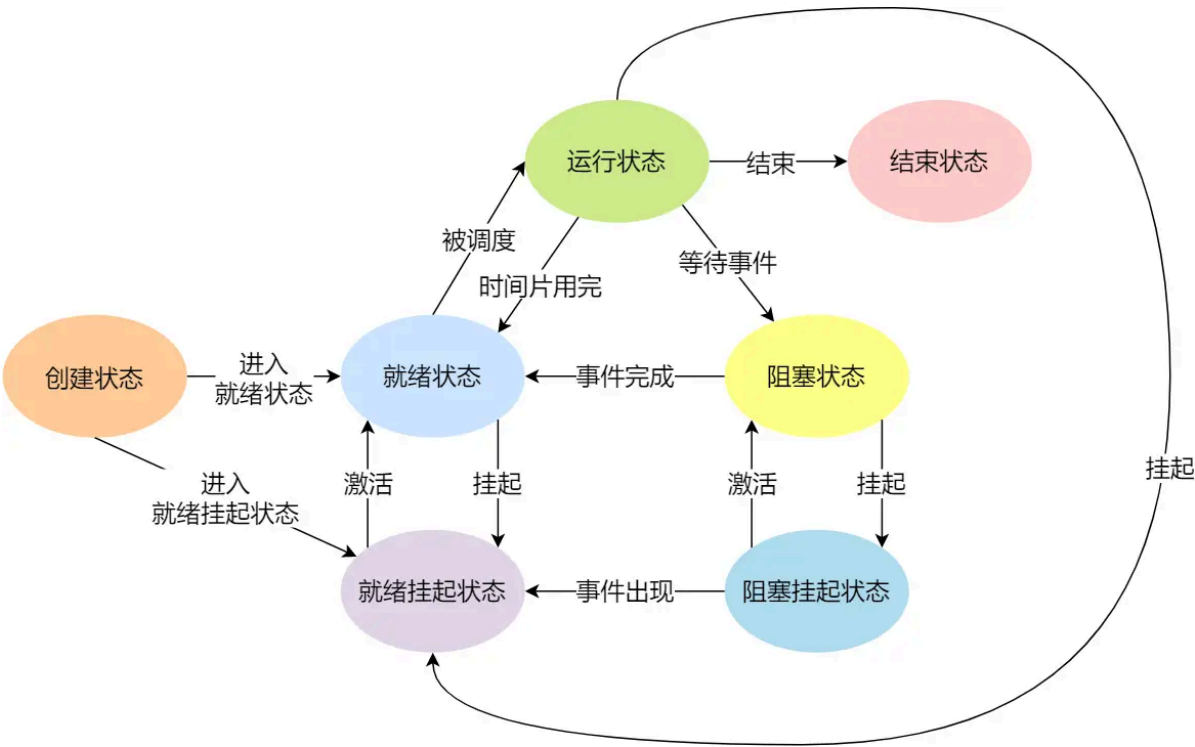
## 进程管理

### 进程

进程是操作系统进行资源分配的最小单元，线程是操作系统进行运算调度的最小单元。

1. 概念: 编写的代码是存储在硬盘的静态文件，通过编译就会生成二进制的可执行文件，当我们运行这个可执行文件的时候，就会被装载在内存中，接着CPU会执行程序中的每一条指令，称为进程。
2. 但是这个读取磁盘的时间很长，这个cpu也不会进行等待数据返回，接着进行其他的进程，当有一个进程数据返回了，cpu收到了就会中断当前的，接着执行以前的。
3. 虽然单核的 CPU 在某一个瞬间，只能运行一个进程。但在 1 秒钟期间，它可能会运行多个进程，这样就产生**并行的错觉**，实际上这是**并发**。
4. CPU状态:在一个进程的活动期间至少具备三种基本状态，即运行状态、就绪状态、阻塞状态。
5. 状态:
  - 运行:该时刻进程占用CPU
  - 就绪: 可运行，由于其他进程处于运行状态而暂停运行
  - 阻塞: 该进程正在等待事件发生而暂停运行。
  - 创建
  - 结束

image-20240709102348340



# Vue大厂面经总结

## vue生命周期

主要讲active、deactive、 onMounted、 onUnmounted

# vue3新属性，核心原理 proxy 代码

```
// 定义一个 handler 对象，包含各种拦截操作
const handler = {
 get(target, key, receiver) {
 console.log(`Getting ${key}`);
 return Reflect.get(target, key, receiver);
 },
 set(target, key, value, receiver) {
 console.log(`Setting ${key} to ${value}`);
 return Reflect.set(target, key, value, receiver);
 }
};

// 创建一个原始对象
const data = {
 message: 'Hello, vue 3!'
};

// 使用 Proxy 创建一个响应式对象
const proxyData = new Proxy(data, handler);

// 访问和修改属性
console.log(proxyData.message); // Getting message -> Hello, vue 3!
proxyData.message = 'Hello, Proxy!'; // Setting message to Hello, Proxy!
console.log(proxyData.message); // Getting message -> Hello, Proxy!
```

为什么vue3会使用proxy和defineProxy区别

vue2的这个在添加和删除对象属性时，vue检测不到，只能通过手动的\$set来调用处理。还无法检测数组下标和长度的变化。

vue3因为代理的是整个对象，所以这些都是可以进行监测出来的。

实现原理:

get收集依，set和delete触发依赖

## vue3的响应模式

1.说一下响应式是什么，

2.vue2和vue3这个的区别

3.Vue 3 使用 Proxy 来实现响应式系统的主要步骤如下：

1. **创建响应式对象**：使用 Proxy 创建一个代理对象，通过传入目标对象和拦截处理器（handler）。
2. **拦截 get 操作**：在读取属性时，通过 get 拦截器可以记录依赖（依赖收集），并返回属性值。
3. **拦截 set 操作**：在设置属性时，通过 set 拦截器可以触发更新（依赖触发），并更新属性值。
4. **依赖收集和触发**：Vue 3 使用一个全局的依赖收集器来跟踪哪些组件或函数依赖于哪些数据。当数据变化时，触发这些依赖以更新视图。

```
const targetMap = new WeakMap();
```

```

function track(target, key) {
 let depsMap = targetMap.get(target);
 if (!depsMap) {
 targetMap.set(target, (depsMap = new Map()));
 }
 let dep = depsMap.get(key);
 if (!dep) {
 depsMap.set(key, (dep = new Set()));
 }
 dep.add(effect);
}

function trigger(target, key) {
 const depsMap = targetMap.get(target);
 if (!depsMap) return;
 const dep = depsMap.get(key);
 if (dep) {
 dep.forEach(effect => effect());
 }
}

let activeEffect = null;

function effect(fn) {
 activeEffect = fn;
 fn();
 activeEffect = null;
}

const handler = {
 get(target, key, receiver) {
 if (activeEffect) {
 track(target, key);
 }
 return Reflect.get(target, key, receiver);
 },
 set(target, key, value, receiver) {
 const result = Reflect.set(target, key, value, receiver);
 trigger(target, key);
 return result;
 }
};

const data = { message: 'Hello, Vue 3!' };
const proxyData = new Proxy(data, handler);

effect(() => {
 console.log(proxyData.message); // This will re-run whenever `message` changes
});

proxyData.message = 'Hello, Proxy!'; // Setting message to Hello, Proxy!

```



## 使用Object.defineProperty做响应式的缺点

1. 深度监听，需要一次性递归到底，计算量比较大
2. 描述符只有get和set，无法监听新增属性和删除属性操作
3. 无法原生监听数组

这三个缺点中，第2点是defineProperty本身API缺陷，而第1点和第3点都是出于性能考虑而做的取舍。

Vue3中可以在get中深度监听是因为Proxy无需对一个对象的每一个属性单独设置，只需要代理对象整体就行了，时间复杂度只有 $O(1)$ ，而Object.defineProperty需要对每个属性单独设置，时间负载度为 $O(n)$ ，性能差一个数量级，所以一次性递归监听是最优解。

## vue组件之间通信方式

1. 父子的props
2. 兄弟之间要么在父组件进行传递一层，要么使用pinia或者vuex这些状态管理工具
3. provide和inject
4. vuex或者pinia全局状态管理工具
5. emits
6. ref和defineExpose

## vue为什么是mvvm框架

首先介绍一下什么式MVVM，接着说下面的四点。

1. 视图和数据进行分离，首先MVVM指的是M为Model层也就是数据层，View是视图层，VM通过ViewModal来进行连接，也就是通过数据绑定的方法，将视图与数据关联起来，使得数据的变化可以响应到页面上，反之亦然
2. 数据驱动视图，vue中我们只需要关注数据的变化，无需手动操作dom，就可以反映视图上面，因为双向绑定，视图的更新也会同步更新数据，数据的变化也会同步更新视图
3. 响应式编程:Vue通过数据拦截，自动触发视图的更新。通过对数据的监听与更新，实现了响应式编程。当数据发生改变时，Vue会自动更新相应的视图。这大大简化了界面更新的操作，提高了开发效率。
4. 组件化开发

综上所述，选择mvvm框架是可以实现视图和数据分离、数据驱动视图、响应式编程、组件化开发

## v-if和v-for为什么不建议放一起用

vue2是v-for > v-if

vue3是v-if > v-for

## vue和react区别

相同点

- 都使用组件式开发，函数式编程
- 都是数据驱动视图
- 都有虚拟DOM
- 都有支持的native方案:vue是weex，react是rn

#### 不同点

- 数据流向不同，react一直推崇单项数据流，但是vue是双向数据流
- 数据变化实现原理不同，react是不可变数据，vue是可变的数据
- 组件化通信：vue有三种方式实现，父组件和子组件通过props/回调函数，provide和inject。  
react：万物可以通过props
- 响应式不同，vue会使用Proxy来代理对象，通过getter和setter来监测数据是否改变，自动收集依赖，使视图更新。react需要通过useState中的setState来更新状态，状态发生改变之后，组件也会重新渲染。
- 写法不同:vue是在一个组件中将样式，html和js逻辑分开了，但是react是使用的jsx语法，将html，css等样式以js形式嵌入到JavaScript中。

## v2和v3区别

- 在数据拦截上vue2使用的是Object.defineProperty()来监听对象属性，对新增和删除的属性无法检测，还无法监测数组以及上面的方法。vue3使用的是Proxy，代理的是整个对象，解决了上面的问题
- webpack的tree-shaking：将无用的模块剪辑，仅打包需要的  
能够 `tree-shaking`，打包体积更小了
- vue2选项式，vue3组合式：更容易复用了
- vue3更好地支持typescript
- **vue2**：vue2**不支持**碎片。vue3：**vue3**支持碎片（Fragments），就是说可以拥有多个根节点
- 生命周期不同。

 image-20240802092705389

#### vue3新增的

- fragment：可以支持有多个根节点
- Teleport:可以将我们的模板移动到 DOM 中 VUE APP 之外的其他技术
- suspense

```
<button @click="showToast" class="btn">打开 toast</button>
<!-- to 属性就是目标位置 -->
<teleport to="#teleport-target">
 <div v-if="visible" class="toast-wrap">
 <div class="toast-msg">我是一个 Toast 文案</div>
 </div>
</teleport>
```

- componet的api
- v-for和v-if的优先级发生了改变
- 生命周期发生了改变

- 去除了vue2的事件总线
- emits
- mixin去除了

## 为什么要有虚拟DOM

### 1. 说一下虚拟dom的概念

将真实DOM抽象了出来，使用JS的对象的形式来抽象出来的。其中JS对象作为基础的树，用对象属性描述节点，最终通过一系列的操作使这棵树映射到真实环境中。其中虚拟DOM的属性和真实DOM的属性使相对应的

### 2. vue和react都封装了虚拟DOM

### 3. 为什么需要虚拟DOM

因为DOM很慢，元素非常大，页面的性能问题，大部分都是DOM操作引起的。

例子:

你用传统的原生 api 或 jquery 去操作 DOM 时，浏览器会从构建 DOM 树开始从头到尾执行一遍流程

当你在一次操作时，需要更新10个 DOM 节点，浏览器没这么智能，收到第一个更新 DOM 请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程

而通过 VNode，同样更新10个 DOM 节点，虚拟 DOM 不会立即操作 DOM，而是将这10次更新的 diff 内容保存到本地的一个 js 对象中，最终将这个 js 对象一次性 attach 到 DOM 树上，避免大量的无谓计算

很多人认为虚拟 DOM 最大的优势是 diff 算法，减少 JavaScript 操作真实 DOM 的带来的性能消耗。虽然这一个虚拟 DOM 带来的一个优势，但并不是全部。虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种GUI

## diff算法

就是通过同层的树节点通过比较的高效算法

特点:

- 同层比较，不会跨层比较
- 在diff比较中，循环从两边向中间比较。

<https://vue3js.cn/interview/vue/diff.html#%E4%BA%8C%E3%80%81%E6%AF%94%E8%BE%83%E6%96%B9%E5%BC%8F>

分别遍历新旧虚拟 DOM 节点的数组，接着通过循环左右双指针比较判断。

新的头 newStartIndex 和老的头 oldStartIndex 对比

新的尾 newEndIndex 和老的尾 oldEndIndex 对比

新的头 newStartIndex 和老的尾 oldEndIndex 对比

新的尾 newEndIndex 和老的头 oldStartIndex 对比

源码分析:

<https://www.cnblogs.com/PaturNax/p/16637349.html>

- 当数据发生改变时，订阅者 `watcher` 就会调用 `patch` 给真实的 `DOM` 打补丁
- 通过 `isSameVnode` 进行判断，相同则调用 `patchVnode` 方法
- `patchVnode`

做了以下操作：

- 找到对应的真实 `dom`，称为 `e1`
- 如果都有都有文本节点且不相等，将 `e1` 文本节点设置为 `vnode` 的文本节点
- 如果 `oldVnode` 有子节点而 `VNode` 没有，则删除 `e1` 子节点
- 如果 `oldVnode` 没有子节点而 `VNode` 有，则将 `VNode` 的子节点真实化后添加到 `e1`
- 如果两者都有子节点，则执行 `updateChildren` 函数比较子节点

- `updateChildren`

主要做了以下操作：

- 设置新旧 `VNode` 的头尾指针
- 新旧头尾指针进行比较，循环向中间靠拢，根据情况调用 `patchVnode` 进行 `patch` 重复流程、调用 `createElement` 创建一个新节点，从哈希表寻找 `key` 一致的 `VNode` 节点再分情况操作

## v-for为什么有key

`key`是给每一个`vnode`的唯一`id`，也是`diff`的一种优化策略，可以根据`key`，更准确，更快的找到对应的`vnode`节点。

当我们在使用 `v-for` 时，需要给单元加上 `key`

- 如果不用`key`，`Vue`会采用就地复地原则：最小化`element`的移动，并且会尝试尽最大程度在同适当的地方对相同类型的`element`，做`patch`或者`reuse`。
- 如果使用了`key`，`Vue`会根据`keys`的顺序记录`element`，曾经拥有了`key`的`element`如果不再出现的话，会被直接`remove`或者`destroyed`

当 `Vue.js` 用 `v-for` 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，`Vue` 将不会移动 `DOM` 元素来匹配数据项的顺序，而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素

## vue框架和原生有什么区别

vue:

- 数据绑定方便
- 组件式开发，容易复用
- 有着相关生态
- 有着良好性能和跨平台能力，应为虚拟`dom`

js:

- 可以与很多框架和库兼容
- 真实`DOM`

# vue框架做了什么？好处是？

---

vue使一种构建用户界面的渐进式JavaScript框架

在数据绑定方面，我们只需要处理数据即可，只关注于业务逻辑即可

帮助我们更好地组织代码和更加号维护代码

好处

- 数据双向绑定，响应式
- 组件化
- 生态和社区丰富

## vue中绑定节点的原理

---

### vuex和pinia的区别

---

- 设计与使用:Vuex采用的是全局单例模式，通过一个store来管理所有的状态，组件通过store来获取和修改状态。Pinia采用的是分离模式，每个组件都有自己的store实例，通过在组件中创建store实例管理状态。
- 数据的修改，pinia没有mutation，只有state，getter，action。Vuex有state，getter，mutation，模块化，但是pinia没有模块化，每个独立仓库都是defineStore生成出来的。每个store都说是独立的，谁也不属于谁
- 语法的使用，语法上pinia比vuex更容易理解，上手。pinia提供了ts支持，vue支持。
- 体积pinia小
- pina使用了更新的ES6语法和数据处理方式
- api设计:vuex时使用严格单一的store模式，pinia允许使用多个store

### vue3中 watch和watcheffect两个api的区别

---

1. watch需要明确指定要监视的数据源，而watchEffect则可以自动收集其使用的响应式数据，因此代码会更简洁，但是，它的回调函数无法获得旧值和新值的参数。
2. watch在监听对象或数组时，需要开启deep选项才能深度监听其属性或元素的变化。而watchEffect会自动追踪响应式数据的依赖，并在赋值、方法调用等操作后重新执行回调函数。

### vue3中ref和reactive的区别

---

- ref是创建单个响应式数据。reactive是创建多个响应式数据属性的对象。
- 对于基本数据类型推荐使用ref，对于数组、对象，推荐使用reactive。
- reactive会递归将所有的属性转化为响应式数据
- ref 返回一个由 RefImpl 类构造出来的对象，而 reactive 返回一个原始对象的响应式代理 Proxy。
- 其中watch在这两方面上也不一样:当ref包裹对象的时候需要加deep，reactive不需要
- reactive如果解构出来就没有了响应式了

## 项目中vue3的周边生态你用过哪些（组件库elementplus 插件vuex）

---

1. elementplus组件库
2. vuex/pinia状态管理
3. vue-router
4. vite
5. vitest
6. vitepress

## vuex的缺点（持久化问题，本地存储）

---

### vuex工作流程

---

1. **State** 存储应用的状态。
2. **Getter** 从 state 中派生出状态。
3. **Mutation** 更改 state，必须是同步的。
4. **Action** 提交 mutation，可以包含异步操作。
5. **Module** 使 store 结构化，便于管理。

## vuerouter问题以及页面权限问题（有点没印象了）

---

### Vue可以监听数组吗

---

vue2是不能监听数组上的增加和删除数组上的值得

vue3可以监听

### Vue的渲染是异步的吗

---

是异步渲染的，主要是为了提高性能和效率。

Vue中某个组件的状态发生了改变，vue不会立刻更新DOM，而是把这个组件标记为待更新，然后事件循环的下一个tick中，**vue会遍历并执行所有待更新的组件，最后一次性更新DOM。**

异步渲染的优势在于，如果一个组件的状态在同一个事件循环中发生多次变化，那么 Vue 只会执行一次 DOM 更新，从而避免了不必要的计算和 DOM 操作，提高了性能。

### 事件修饰符.sync

---

简化了父子组件之间的双向绑定数据更新。

**父组件传递属性：**

- 使用 `.sync` 修饰符传递的属性，如 `:visible.sync="isModalVisible"`，在子组件中会被处理为 `visible` 属性。

**子组件触发事件：**

- 子组件在需要更新父组件的数据时，使用 `$emit('update:visible', newValue)` 触发更新事件。

父组件接收事件并更新数据：

- Vue 内部会自动将 `update:visible` 事件处理为对 `isModalVisible` 数据的更新，从而实现双向绑定。

无需显式地在父组件中监听和处理 `update:propName` 事件，只需在子组件中使用

`$emit('update:propName', newValue)`。

## Vue2中Computed计算属性的特性以及实现原理

---

### 组件封装的原因？具体做了哪些封装

---

1. 提高复用性
2. 提高了可维护性
3. 提高拓展性

做过的封装：

1. Modal的封装(弹窗)：定义props来接收显示或者还有标题啥的，组件内还可以通过插槽来传入父组件给子组件定义的一些内容，接着就是父组件自定义事件，在外面控制子组件什么时候显示什么时候不显示。
2. Table组件封装(看一下简历的网盘项目)：

### Vue的双向绑定原理

Vue的双向绑定采用的是**发布订阅模式**。

1. 初始化的时候对数据的各个属性的setter/getter进行数据劫持，当数据发生变动的时候，会通知订阅者，触发响应的监听回调。遍历订阅者容器发布的消息。
2. 每一个组件都有一个Watcher实例，把组件渲染的过程中把接触过的数据作为依赖，当依赖的setter发出的时候会通知Watcher，会是组件重新渲染。

### 单页面应用怎么缓存组件？能说一下具体的属性吗？

---

Vue提供了一个keep-alive可以用来动态缓存组件

`include`：字符串或正则表达式，匹配的组件会被缓存。

`exclude`：字符串或正则表达式，匹配的组件不会被缓存。

`max`：数字，缓存组件的最大数量。

### Vue3新属性

---

1. 组合式api
2. Teleport:允许将一个组件渲染到DOM的另一个位置
3. Fragments支持在模板中返回多个根元素，不需要包裹在单一的根元素中。
4. Suspense允许你在异步组件加载时显示一个备用内容
5. Vue 3 将一些全局 API 进行了变更，使其更加模块化和可树摇

```
// Vue 2
import Vue from 'vue';
Vue.component('MyComponent', MyComponent);

// Vue 3
import { createApp } from 'vue';
import App from './App.vue';
import MyComponent from './MyComponent.vue';

const app = createApp(App);
app.component('MyComponent', MyComponent);
app.mount('#app');
```

## 6. 更好支持typescript

# nextTick

Vue.js 的 `nextTick` 实现依赖于 JavaScript 的微任务队列 (microtask queue)。当你调用 `Vue.nextTick` 或者在组件实例上调用 `$nextTick` 时, Vue 会将回调函数推入一个微任务队列中。这个队列会在当前的同步代码执行完毕之后、在下一次 DOM 更新循环结束之后执行。

vue 在更新 DOM 时是异步执行的。

Vue异步更新策略, 就是更新数据的时候。Vue不会立刻更新DOM, 而是开启一个队列, 把组件更新函数保存在队列中, 在同一事件循环中发生的数据变化会异步的批量更新。

应用场景:

1. 在created中获取DOM
2. 响应式数据变化后获取DOM更新后的状态

```
const resolvedPromise = /*#__PURE__*/ Promise.resolve();
let currentFlushPromise = null;
function nextTick(fn) {
 const p = currentFlushPromise || resolvedPromise;
 return fn ? p.then(this ? fn.bind(this) : fn) : p;
}
//nextTick的实现其实是一个Promise的封装。
```

执行原理:

首先是生成一个响应的副作用, 当一个响应式数据发生了改变之后, 与他相关的一个函数会加入到队列中。通过queueJob来维护queue队列, 接着刷新queue

```
const effect = (instance.effect = new ReactiveEffect(
 componentUpdateFn,
 ()=>queueJob(update),
 instance.scope
))
```

```
// 添加任务, 这个方法会在下面的 queueFlush 方法中被调用
function queueJob(job) {
```



```

// 通过 Array.includes() 的 startIndex 参数来搜索任务队列中是否已经存在相同的任务
// 默认情况下，搜索的起始索引包含了当前正在执行的任务
// 所以它不能递归地再次触发自身
// 如果任务是一个 watch() 回调，那么搜索的起始索引就是 +1，这样就可以递归调用了
// 但是这个递归调用是由用户来保证的，不能无限递归
if (!queue.length ||
 !queue.includes(job, isFlushing && job.allowRecurse ? flushIndex + 1 :
flushIndex)) {
 // 如果任务没有 id 属性，那么就将任务插入到任务队列中
 if (job.id == null) {
 queue.push(job);
 }

 // 如果任务有 id 属性，那么就将任务插入到任务队列的合适位置
 else {
 queue.splice(findInsertionIndex(job.id), 0, job);
 }

 // 刷新任务队列
 queueFlush();
}
}

```

这个刷新的queue是使用异步的方式来执行flushJobs，flushJobs函数就是来执行队列里面的函数的（首先会进行排序queue）

```

// 是否正在刷新
let isFlushing = false;

// 是否有任务需要刷新
let isFlushPending = false;

// 刷新任务队列
function queueFlush() {
 // 如果正在刷新，并且没有任务需要刷新
 if (!isFlushing && !isFlushPending) {

 // 将 isFlushPending 设置为 true，表示有任务需要刷新
 isFlushPending = true;

 // 将 currentFlushPromise 设置为一个 Promise，并且在 Promise 的 then 方法中执行 flushJobs
 currentFlushPromise = resolvedPromise.then(flushJobs);
 }
}

```

```

// 任务队列
const queue = [];

// 当前正在刷新的任务队列的索引
let flushIndex = 0;

// 刷新任务
function flushJobs(seen) {

```

```

// 将 isFlushPending 设置为 false, 表示当前没有任务需要等待刷新了
isFlushPending = false;

// 将 isFlushing 设置为 true, 表示正在刷新
isFlushing = true;

// 非生产环境下, 将 seen 设置为一个 Map
if ((process.env.NODE_ENV !== 'production')) {
 seen = seen || new Map();
}

// 刷新前, 需要对任务队列进行排序
// 这样可以确保:
// 1. 组件的更新是从父组件到子组件的。
// 因为父组件总是在子组件之前创建, 所以它的渲染优先级要低于子组件。
// 2. 如果父组件在更新的过程中卸载了子组件, 那么子组件的更新可以被跳过。
queue.sort(comparator);

// 非生产环境下, 检查是否有递归更新
// checkRecursiveUpdates 方法的使用必须在 try ... catch 代码块之外确定,
// 因为 Rollup 默认会在 try-catch 代码块中进行 treeshaking 优化。
// 这可能会导致所有警告代码都不会被 treeshaking 优化。
// 虽然它们最终会被像 terser 这样的压缩工具 treeshaking 优化,
// 但有些压缩工具会失败 (例如: https://github.com/evanw/esbuild/issues/1610)
const check = (process.env.NODE_ENV !== 'production')
 ? (job) => checkRecursiveUpdates(seen, job)
 : NOOP;

// 检测递归调用是一个非常巧妙的操作, 感兴趣的可以去看看源码, 这里不做讲解
try {
 for (flushIndex = 0; flushIndex < queue.length; flushIndex++) {
 const job = queue[flushIndex];
 if (job && job.active !== false) {
 if ((process.env.NODE_ENV !== 'production') && check(job)) {
 continue;
 }

 // 执行任务
 callWithErrorHandling(job, null, 14 /* ErrorCodes.SCHEDULER */);
 }
 }
}
finally {
 // 重置 flushIndex
 flushIndex = 0;

 // 快速清空队列, 直接给 数组的 length 属性 赋值为 0 就可以清空数组
 queue.length = 0;

 // 刷新生命周期的回调
 flushPostFlushCbs(seen);

 // 将 isFlushing 设置为 false, 表示当前刷新结束
 isFlushing = false;

 // 将 currentFlushPromise 设置为 null, 表示当前没有任务需要刷新了


```

```
currentFlushPromise = null;

// pendingPostFlushCbs 存放的是生命周期的回调，
// 所以可能在刷新的过程中又有新的任务需要刷新
// 所以这里需要判断一下，如果有新添加的任务，就需要再次刷新
if (queue.length || pendingPostFlushCbs.length) {
 flushJobs(seen);
}
}
```

## v-modal

---

 image-20240801103034451

vue3.4出现的defineModal


## 拓展组件

---

 image-20240801103534668

 image-20240801103734202

 image-20240801103807919

 image-20240801103849711

## 单向数据流

---

 image-20240801104256036

## 虚拟DOM

---

 image-20240802085502883

 image-20240802085718733


 image-20240802085823243

## diff算法

---

 image-20240802090949304

 image-20240802091403008

 image-20240802091845990

## key作用

---

 image-20240802145443286

## compunted和watch

---

 image-20240802145749976

# keep-alive

---


 image-20240802151255647## 从0-1创建架构

 image-20240802151605222

## vue最佳实践

---

看vue文档的最佳实践

 image-20240802151752834

## vuex

---

 image-20240808142330908

刷新浏览器，vuex中的state会重新变为初始状态

解决办法

1. 插件vuex-persistedstate（我个人没用过）

2. 在刷新前将vuex中的数据直接保存到浏览器缓存中，页面刷新后，在页面刷新的时候再次请求远程数据，使之动态更新vuex数据，具体步骤：监听页面刷新事件，在页面刷新之前，将vuex里的数据存到sessionStorage里，然后在页面刷新之后，调取获取数据的接口，在接口还没有返回数据的时候，就先用sessionStorage里的数据，等接口返回数据后，就使用接口返回的，顺便更新vuex里的数据

## 从template到render的处理过程

---

 image-20240808142752738

 image-20240808143023212

vue中的编译器何时执行？

执行时间：在vue的运行的环境而不同，如果是webpack环境的话会有一个预编译的阶段，提前将我们编写的模板进行编译。

携带编译器版本的vue在运行时编译，组件创建阶段。

## vue实例挂载发生了什么

---

 image-20240808144941254

渲染器和响应式来进行结合使用，将那些渲染器放到了副作用中，将数据使用ref或者reactive进行包裹，当副作用执行的时候就建立起了数据和视图之间的联系，最后通过渲染器来进行渲染真实dom

## vue3设计目标

---

 image-20240808145945243

## vue实例为什么只能是一个跟

---

 image-20240808150439540

# React面经

## react router有哪些常用的路由模式？

- hash: HashRouter
- history:BrowserRouter

它通过监听 URL 的变化，然后渲染相应的组件，从而实现页面之间的切换和跳转。当用户点击链接或执行前进/后退操作时，React Router 感知到 URL 的变化，然后根据匹配的路由规则来决定渲染哪个组件，最终呈现给用户相应的页面内容。这种机制让我们能够创建单页面应用，并且在不同的 URL 地址下展示不同的内容，就好像是在多个页面间进行导航一样。

React Router 有两种主要的路由模式：HashRouter 和 BrowserRouter。HashRouter 使用 URL 中的哈希部分 (#) 来控制路由，适合不支持 HTML5 history API 的环境；而BrowserRouter 则使用 HTML5 提供的 history API 来控制路由，它提供了更加友好和直观的 URL 格式，适用于大多数现代浏览器环境下的单页面应用。

## 设计模式

### 你在项目中有用过的设计模式有哪些

#### 构造器

```
//他们是相似的，所以可以封装为构造函数,构造器来复用
let obj1 = {
 name:"kerwain",
 age:18
}
let obj2 = {
 name:"tom",
 age:19
}
function Person(username,age){
 this.username = username
 this.age = age
}
const person1 = new Person('kerwain',18)
const person2 = new Person('tom',19)
console.log(person1,person2)
```

封装axios的时候使用了构造器模式

#### 单例模式

就是一个类只对应一个实例之后的就会直接复用之前的实例。

比如创建弹窗的时候

```
const Model = (function (){
 let instance = null
 return function (){
 if(!instance){
 instance = document.createElement('div')
```

```

instance.innerHTML = '登录对话框'
instance.className = 'login'

document.body.appendChild(instance)
}
return instance
}
})();
const open = document.querySelector('.open')
open.addEventListener('click',function(){
 const model = new Model()
 model.style.display = 'block'
})

```

//为了防止每次点击都是多个对话框，这样就是每次点击创建的都是一个实例了

```

class Singleton{
 constructor(name,age) {
 if(!Singleton.instance){
 this.name = name
 this.age = age
 Singleton.instance = this
 }
 return Singleton.instance
 }
}

```

//拿到的都是同一个实例

## 工厂模式

# 前端工程化

## 打包工具

### 为什么使用webpack

### Webpack为什么打包、进行压缩之后，性能就会提高呢

### webpack构建流程

#### 1. 初始化：

- Webpack 从配置文件（如 `webpack.config.js`）中读取配置信息。
- 初始化各种内部插件和加载器。
- 创建一个 `Compiler` 实例，这个实例是整个构建流程的核心。

#### 2. 解析入口：

- 根据配置文件中的 `entry` 字段找到所有的入口文件。
- 从这些入口文件开始，**递归地解析所有依赖的模块**。

#### 3. 模块解析：

- Webpack **使用加载器（Loaders）** 将不同类型的文件转换为可以被 Webpack 处理的模块。
- 每个模块都会有一个唯一的模块 ID。

#### 4. 依赖图生成：

- Webpack 递归解析所有模块及其依赖，生成一个依赖图（Dependency Graph）。
- 这个图描述了各个模块之间的依赖关系。

#### 5. 模块打包：

- Webpack 根据依赖图将所有模块打包成一个或多个 bundle 文件。
- 在这个过程中，Webpack 会根据配置文件中的 `optimization` 选项进行代码分割、压缩等优化操作。

#### 6. 输出生成：

- 将打包好的 bundle 文件输出到配置文件中的 `output` 目录。
- 在输出过程中，Webpack 还会应用各种插件（Plugins）来进一步优化和处理输出文件，例如生成 HTML 文件、提取 CSS、压缩代码等。

#### 7. 完成：

- 构建完成后，Webpack 会输出构建结果，并可能会启动一个开发服务器（如果配置了 `webpack-dev-server`），以便进行实时开发和调试。

## webpack的loader和plugin的区别

loader是**文件加载器**，能够加载资源文件，并对这些文件做一些处理，例如编译、压缩等，最终一块打包的文件中。例如ES6及其以上的高级语法，可以通过loader转化为ES5以下的语法。

plugin:给webpack一些灵活的功能吧。例如打包优化，资源管理之类的。

运行时间不同:

loader是在打包文件之前执行的，plugins是整个编译周期都起作用的。

## 常见的loader有哪些

1. 讲一下loader是什么:对模块的源代码进行转化
2. webpack要做的就是，分析各个模块之间的依赖关系，形成资源列表，最终打包到指定的文件中。
3. 在webpack内部中，任何文件都是模块，默认情况下，遇到import/require加载模块的时候，webpack只支持js和json文件打包，默认不支持css、scss、png等类型文件的这时候就需要进行配置loader了。
4. 配置方式（推荐）：在 webpack.config.js 文件中指定 loader

常见的loader

- style-loader: 将css添加到DOM的内联样式标签style里
- css-loader :允许将css文件通过require的方式引入，并返回css代码
- less-loader: 处理less
- sass-loader: 处理sass
- postcss-loader: 用postcss来处理CSS
- autoprefixer-loader: 处理CSS3属性前缀，已被弃用，建议直接使用postcss
- file-loader: 分发文件到output目录并返回相对路径
- url-loader: 和file-loader类似，但是当文件小于设定的limit时可以返回一个Data Url
- html-minify-loader: 压缩HTML

- babel-loader :用babel来转换ES6文件到ES

## webpack和vite区别或者和其他打包工具的区别

1. 讲一下模块化工具:是一种处理复杂系统分解成**更好地可管理模块的方式**
2. 可用来分割, 组织和打包应用

### vite

主要是由两部分构成:

一部分是开发服务器, **基于原生的ES模块**, 提供了丰富的内建功能

一套构建指令, **它使用Rollup打包你的代码**, 可以预配置。

其作用类似 webpack + webpack-dev-server, 其特点如下:

- 快速的冷启动
- 即时的模块热更新
- 真正的按需编译

vite会直接启动开发服务器, 不需要进行打包操作, 也意味着**不需要分析模块的依赖、不需要编译**, 因此启动速度非常快。

利用现代浏览器支持 ES Module 的特性, **当浏览器请求某个模块的时候, 再根据需要对模块的内容进行编译, 这种方式大大缩短了编译时间。**

在 webpack 中, **每次修改代码后都需要对整个项目进行重新编译, 然后重新生成大量的代码和资源文件。**而在 vite 中, **它使用了浏览器原生的ES模块加载器, 当开发者修改代码后, vite 会即时在浏览器中编译和打包代码**, 然后将更改的部分直接传递给浏览器, 并重新加载这部分代码。因此, vite 的编译和打包速度比 webpack 更快, 因为它避免了重复的编译和打包步骤, 以及更高效地利用了现代浏览器的功能。另外, vite 还使用了缓存机制和按需加载的方式, 这也是它快速打包的原因之一。当开发者第一次访问项目时, vite 会对项目进行编译和打包, 并缓存生成的文件。这样, 当开发者下一次打开项目时, vite 只需要编译和打包发生更改的部分, 而不需要重新编译和打包整个项目。这种按需加载的方式也能够进一步提高 vite 的打包速度。

## 打包文件的时候, 怎么把每个包控制在1兆以内

### 1. Webpack

Webpack 是一个流行的模块打包工具, 可以通过 splitChunks 插件来分割代码, 将每个包的大小控制在 1 兆以内。

```
const path = require('path');

module.exports = {
 entry: './src/index.js',
 output: {
 filename: '[name].[contenthash].js',
 path: path.resolve(__dirname, 'dist'),
```



```

},
optimization: {
 splitChunks: {
 chunks: 'all',
 maxSize: 1024 * 1024, // 1 MB
 },
},
};

```

## 2. Vite

vite的build.rollupOptions.output.manualChunks.maxChunkSize

```

import { defineConfig } from 'vite';

export default defineConfig({
 build: {
 rollupOptions: {
 output: {
 manualChunks(id) {
 if (id.includes('node_modules')) {
 return id
 .toString()
 .split('node_modules/')[1]
 .split('/')[0]
 .toString();
 }
 },
 chunkFileNames: '[name]-[hash].js',
 maxChunkSize: 1024, // 1 MB
 },
 },
 },
});

```

## Tree Shaking

消除未使用的代码，减少打包后的文件大小，提高应用性能。。

## 包管理工具，以及区别

npm、yarn、pnpm

### 1. npm:

是Node.js官方包管理器

- 优点

官方支持：npm 是 Node.js 的官方包管理器，因此它得到了广泛的支持和使用。

自动依赖项解析：npm 可以自动解析项目中的依赖项，并安装所需的软件包。

版本控制：npm 使用 package-lock.json 文件来确保安装过程中使用相同的依赖项版本。

社区支持：npm 有一个庞大的社区，可以提供大量的软件包和资源。

- 缺点
- 安装速度较慢：由于 npm 是单线程解析依赖项，因此安装速度可能相对较慢。
- 文件冲突：由于 npm 使用 package-lock.json 文件来锁定依赖项版本，因此在多人协作时可能会出现文件冲突问题。

2. yarn:

优点:

安装速度快：由于 yarn 可以并行下载和缓存软件包，因此安装速度通常比 npm 快。

版本控制：yarn 使用 yarn.lock 文件来确保安装过程中使用相同的依赖项版本。

离线模式：yarn 支持离线模式，可以在没有互联网连接的情况下工作。

更好的脚本执行：yarn 通过更好的脚本执行功能，使得运行脚本变得更加容易。

缺点:

- Facebook 集中控制：由于 yarn 是由 Facebook 开发的，因此有人担心 Facebook 可能会在未来控制 yarn 的发展方向。
- 依赖项缓存：yarn 将所有软件包都缓存在本地，这可能会占用大量磁盘空间。

3. pnpm

pnpm 具有类似于 yarn 的速度和稳定性，但与 yarn 不同的是，它采用了一种不同的依赖项解析方法，即将依赖项安装在单个位置，并使用符号链接将它们链接到每个项目中。

优点:

安装速度最快：由于 pnpm 可以共享依赖项，因此它可以更快地安装和更新模块。

多线程解析：与 yarn 类似，pnpm 也可以使用多线程解析依赖项。

离线模式：pnpm 支持离线模式，并且可以在没有互联网连接的情况下工作。

更好的内存管理：pnpm 使用更好的内存管理技术，可以更有效地利用系统资源。

缺点:

- 相对较新：由于 pnpm 是较新的包管理器，因此可能会缺乏一些 npm 和 yarn 中的功能和资源。
- 可能存在兼容性问题：由于 pnpm 采用了不同的依赖项解析方法，因此可能会存在一些兼容性问题。

## CommonJs和esModule的区别

---

基本的不同

- **CommonJS：**
  - 主要用于 Node.js。
  - 设计初衷是为了在服务器端使用模块化系统。
  - 通过 `require()` 和 `module.exports` 进行模块导入和导出。
- **ESModules (ESM)：**
  - 是 ECMAScript 2015 (ES6) 标准的一部分。
  - 设计初衷是为了在浏览器和服务端都能使用模块化系统。

- 通过 `import` 和 `export` 进行模块导入和导出。

## 加载方式

- **CommonJS:**
  - 模块是**同步加载**的。
  - 适用于服务器端环境，因为服务器端的文件读取是**本地的、快速的**。
- **ESModules:**
  - 模块是**异步加载**的。
  - 适用于浏览器环境，因为浏览器需要通过**网络请求获取模块文件**。

## 执行时机

- CommonJS
  - 模块在第一次被 `require` 时执行，并且结果会被**缓存**。
  - 后续的 `require` 调用会返回缓存的结果。
- ESModules
  - **模块在解析阶段执行**，因此在模块之间可以有循环依赖。
  - **模块的导入和导出是静态的**，编译时就可以确定依赖关系。

## 兼容性

- **CommonJS:**
  - 主要在 Node.js 环境中使用。
  - 在浏览器中使用需要通过工具（如 Browserify 或 Webpack）进行打包。
- **ESModules:**
  - 现代浏览器和 Node.js（自 v12.17.0 起）都原生支持。
  - 需要在 Node.js 中使用 `.mjs` 扩展名，或者在 `package.json` 中配置 `"type": "module"`。

## 导出方式

### Commonjs

- 导出是一个对象，可以动态修改。

### ESModule

- 导出是静态的，编译时确定。

# ts

## 理解

是JavaScript的超集，是一个静态类型检查的语言，提供了类型注解，在代码编译阶段就可以检查出数据类型的错误

而且为了兼容性，需要通过tsc将typescript转化为JavaScript。

报错的时候仍然产出文件:

大多数时候，这样没什么问题，但有的时候，这些检查会对我们造成阻碍。

举个例子，想象你现在正把 JavaScript 代码迁移到 TypeScript 代码，并产生了很多类型检查错误。

最后，你不得不花费时间解决类型检查器抛出的错误，但问题在于，原始的 JavaScript 代码本身就是可以运行的！为什么把它们转换为 TypeScript 代码之后，反而就不能运行了呢？

## ts类型

string、number、boolean、Arrays、any、联合类型、类型别名、接口、null、undefined、枚举

```
//类型别名
type Point = {
 x: number;
 y: number;
};
```

拓展的使用extends,&

使用 `as const` 将整个对象转换为类型文字

## 要实现一个类型，其值类型为枚举类型的键

可以使用 TypeScript 中的 `keyof` 操作符，它可以获取一个对象类型的所有键并将其作为联合类型返回。

```
enum MyEnum {
 First = "firstValue",
 Second = "secondValue",
 Third = "thirdValue"
}
type EnumKeys = keyof typeof MyEnum;
```

## type和interface区别

`interface` 更适合用于定义对象和类的结构，支持声明合并和继承。`type` 更灵活，可以用于定义各种复杂类型，包括联合类型、交叉类型等。

## Omit

`Omit` 是 TypeScript 中的一个内置辅助类型，它用于创建一个新类型，该类型从现有类型中排除指定的键。你可以借助 `Omit` 来排除不需要的属性，从而简化类型定义。

```
interface Person {
 name: string;
 age: number;
 email: string;
 address: string;
}
type PersonWithoutEmail = Omit<Person, 'email'>;

const person: PersonWithoutEmail = {
 name: "Alice",
 age: 30,
 address: "123 Main St"
};
```

