目录

[通用 4](#_Toc150353165)

[微前端 4](#_Toc150353166)

[性能评价及相应工具 5](#_Toc150353167)

[三次握手和四次挥手 6](#_Toc150353168)

[TCP和UDP的区别 7](#_Toc150353169)

[Http1 Http2 Http3的区别 8](#_Toc150353170)

[http特点和请求方法 9](#_Toc150353171)

[进程间通信 9](#_Toc150353172)

[死锁产生的必要条件 10](#_Toc150353173)

[WebSocket 10](#_Toc150353174)

[常见状态码 10](#_Toc150353175)

[http报文格式简介 11](#_Toc150353176)

[MVC和MVVM 12](#_Toc150353177)

[浏览器垃圾回收 13](#_Toc150353178)

[浏览器缓存 14](#_Toc150353179)

[浏览器渲染 16](#_Toc150353180)

[浏览器中的线程与进程 17](#_Toc150353181)

[短信发送原理 17](#_Toc150353182)

[浏览器输入一行url会发生什么 18](#_Toc150353183)

[服务端推送 18](#_Toc150353184)

[CSR、SSR和SSG 19](#_Toc150353185)

[单点登录SSO 20](#_Toc150353186)

[前端安全 21](#_Toc150353187)

[token组成、特点和存储位置 23](#_Toc150353188)

[cookie和session的区别 24](#_Toc150353189)

[禁止脚本操作cookie 24](#_Toc150353190)

[跨域请求 24](#_Toc150353191)

[HTTP常见的请求头 25](#_Toc150353192)

[HTTPS如何保证安全 26](#_Toc150353193)

[AOP模式 27](#_Toc150353194)

[JSON 28](#_Toc150353195)

[Git 29](#_Toc150353196)

[定义 29](#_Toc150353197)

[代码回滚 29](#_Toc150353198)

[git rebase 和 git merge 29](#_Toc150353199)

[常用指令 30](#_Toc150353200)

[JavaScript 32](#_Toc150353201)

[基本数据类型 32](#_Toc150353202)

[ES6新特性 32](#_Toc150353203)

[闭包 32](#_Toc150353204)

[节流和防抖 33](#_Toc150353205)

[手写Promise.all 34](#_Toc150353206)

[事件循环 34](#_Toc150353207)

[创建对象的几种方法 35](#_Toc150353208)

[Symbol的概念及应用场景 35](#_Toc150353209)

[JavaScript 的继承类型 36](#_Toc150353210)

[判断一个字符串是否是正则表达式 36](#_Toc150353211)

[V8执行JavaScript的过程 37](#_Toc150353212)

[比较值是否相等方法 37](#_Toc150353213)

[常见正则表达式 38](#_Toc150353214)

[实现红绿灯 38](#_Toc150353215)

[try catch 39](#_Toc150353216)

[localStorage设置过期时间 40](#_Toc150353217)

[iframe父子页面间通信 41](#_Toc150353218)

[CSS 43](#_Toc150353219)

[flex:1和flex:auto的区别 43](#_Toc150353220)

[用纯CSS创建一个三角形 43](#_Toc150353221)

[div水平垂直居中 43](#_Toc150353222)

[CSS画一条0.5px的线 44](#_Toc150353223)

[CSS清除浮动 44](#_Toc150353224)

[CSS双飞燕布局 45](#_Toc150353225)

[盒子模型 47](#_Toc150353226)

[BFC块级格式化上下文 47](#_Toc150353227)

[CSS选择器 48](#_Toc150353228)

[CSS Sticky Footer布局 48](#_Toc150353229)

[CSS隐藏元素 49](#_Toc150353230)

[Vue 51](#_Toc150353231)

[Vue生命周期 51](#_Toc150353232)

[Vue2和Vue3的区别 51](#_Toc150353233)

[Vue中watch和computed的区别 52](#_Toc150353234)

[Vue diff算法 52](#_Toc150353235)

[Vue 双向绑定的原理 53](#_Toc150353236)

[SPA首屏优化方式 54](#_Toc150353237)

[Vue组件间通信方式 54](#_Toc150353238)

[hash模式和history模式的区别 55](#_Toc150353239)

[Vue中key的作用 55](#_Toc150353240)

[KeepAlive组件 56](#_Toc150353241)

[Vue中父子组件的生命周期顺序 56](#_Toc150353242)

[Vue中常用的修饰符 56](#_Toc150353243)

[Vue中自定义指令 57](#_Toc150353244)

[虚拟DOM 58](#_Toc150353245)

[权限管理 58](#_Toc150353246)

[Vue3 性能提升 59](#_Toc150353247)

[Vue 3 Tree shaking特性 59](#_Toc150353248)

[Vue3中ref与reactive 60](#_Toc150353249)

[React 61](#_Toc150353250)

[React特性 61](#_Toc150353251)

[React和Vue区别 61](#_Toc150353252)

[React Hooks如何存储组件中的状态 61](#_Toc150353253)

[React中的函数式思想 62](#_Toc150353254)

[React生命周期 62](#_Toc150353255)

[Webpack 63](#_Toc150353256)

[Webpack的构建过程 63](#_Toc150353257)

[Webpack的优点 63](#_Toc150353258)

[Webpack和Vite的区别 63](#_Toc150353259)

[loader和plugin的区别 64](#_Toc150353260)

[动态引入原理 65](#_Toc150353261)

[HMR实现 65](#_Toc150353262)

[webpack proxy工作原理 66](#_Toc150353263)

[设计模式 67](#_Toc150353264)

[设计模式七大原则 67](#_Toc150353265)

[单例模式 67](#_Toc150353266)

[工厂模式 68](#_Toc150353267)

[策略模式 68](#_Toc150353268)

[代理模式 68](#_Toc150353269)

[观察者模式 69](#_Toc150353270)

[发布订阅模式 69](#_Toc150353271)

## 通用

### 微前端

概念

微前端（Micro-Frontends）是一种类似于微服务的架构，它将微服务的理念应用于浏览器端，即将 Web 应用由单一的单体应用转变为多个小型前端应用聚合为一的应用。各个前端应用还可以独立运行、独立开发、独立部署。微前端不是单纯的前端框架或者工具，而是一套架构体系。

目标

拆分和细化：将庞大应用进行拆分，并随之解耦，每个部分可以单独进行维护和部署，提升效率。

整合历史系统：微前端可以将采用老框架类似（Backbone.js，Angular.js 1）的B端管理系统等进行整合，在基本不修改逻辑的同时来兼容新老两套系统并行运行。

实现方案

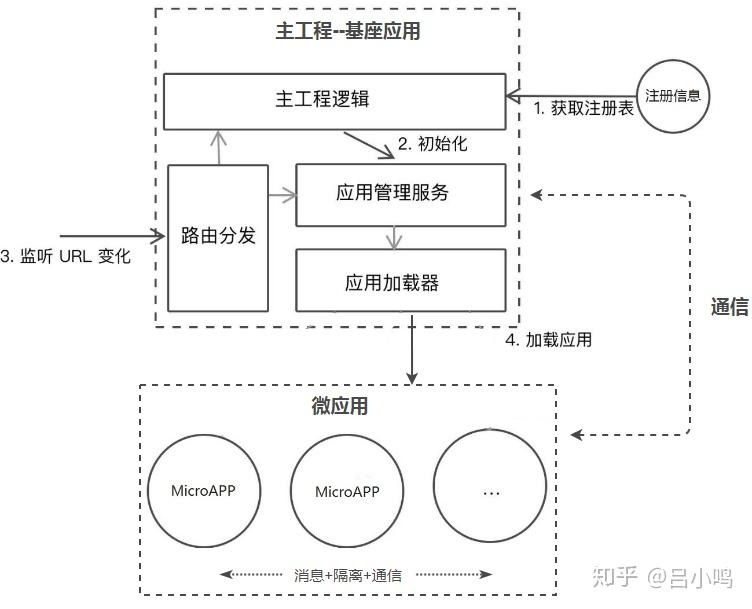


上述方案中，每种都有自己的优劣。最原始的Nginx配置反向代理是从接入层的角度来将系统进行分离，但是需要运维配置；而iframe嵌套是最简单和最快速的方案，但是iframe的弊端也是无法避免的；而Web Components的方案则需要大量的改造成本；最后的组合式应用路由分发方案改造成本中等并且能满足大部分需求，也不影响各前端应用的体验，是当下各个业务普遍采用的一种方案。

模块组成

当下微前端主要采用的是组合式应用路由方案，该方案的核心是“主从”思想，即包括一个基座（MainApp）应用和若干个微（MicroApp）应用。基座应用大多数是一个前端SPA项目，主要负责应用注册、路由映射、消息下发等。而微应用是独立前端项目，这些项目不限于采用React，Vue，Angular或者JQuery开发。每个微应用注册到基座应用中，由基座进行管理，但是如果脱离基座也是可以单独访问。

基本流程如下：



### 性能评价及相应工具

性能评价

·FP（First Paint）首次绘制时间，表示渲染出第一个像素点的时间，值越低越好；

·FCP（First Contentful Paint）浏览器首次绘制来自DOM的内容的时间。白屏时间，值越低越好。这是用户第一次看到的内容（可以是文本、图片、非空白canvas）；

·LCP（Largest Contentful Paint）可视窗口最大内容渲染时间，值越低越好；

·FID（First Input Delay）用户首次交互时间，值越低越好

·TTI（Time to Interactive）用户可交互时间（可流畅交互时间），值越低越好；

·TBT（Total Blocking Time）用户行为阻塞时间，值越低越好；

·CLS（Cumulative Layout Shift）可视窗口中累计可见元素布局偏移；

·SI（Speed Index）页面渲染时间（look usable），值越低越好。

工具

现代浏览器工具

Network、Performance

Lighthouse

Lighthouse是谷歌开发并开源的web性能测试工具，用于改进网络应用的质量，可以将其作为一个Chrome扩展程序运行，或从命令行运行。只需要为其提供一个需要审查的地址，Lighthouse就会对页面进行一连串的测试，生成一个有关页面性能的报告。

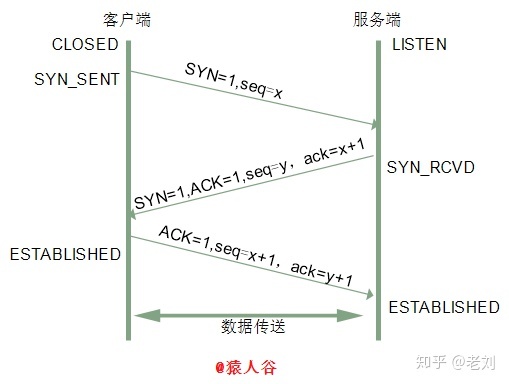
WebPageTest

在线web性能测试工具(https://www.webpagetest.org), 提供多地点测试，能测试已经发布了的网站。输入需要测试的网页地址，选择测试地理位置、测试的浏览器等，点击start test按钮就开始测试，

### 三次握手和四次挥手

三次握手

三次握手的目的是建立可靠的通信通道，双方确认自己与对方的发送与接收是否正常。

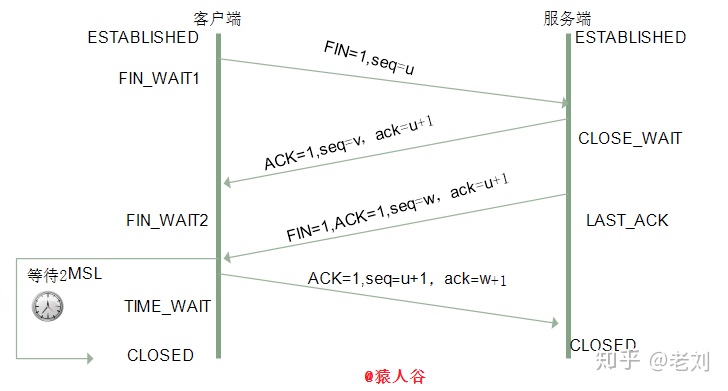


第一次握手：Client什么都不能确认，Server确认了：对方发送正常，自己接收正常。

第二次握手：Client确认了：自己发送、接收正常，对方发送正常、接收正常；Server确认了：对方发送正常，自己接收正常。

第三次握手：Client确认了：自己发送、接收正常，对方发送正常接收正常；Server 确认了：对方发送正常，接收正常，自己发送正常，接收正常。

四次挥手



其中MSL(Maximum Segment Lifetime)为报文最长存活时间

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。

这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

三次挥手不可以，理由是：

如果是三次挥手，在服务器接收到客户端发送关闭的请求后，把FIN和ACK包一起发过去。这样会造成服务端还有数据没有发送完，造成了数据的丢失。所以中间的这一段时间，等待服务器把剩余的数据发送完是很有必要的。

2MSL

这2个MSL中的第一个MSL是为了等自己发出去的最后一个ACK从网络中消失，而第二MSL是为了等在对端收到ACK之前的一刹那可能重传的FIN报文从网络中消失。

### TCP和UDP的区别

TCP：传输控制协议(Transmission Control Protocol)

UDP：用户数据报协议(User Datagram Protocol)

1、基于连接（TCP）与无连接（UDP）；

2、对系统资源的要求（TCP较多，UDP少）；

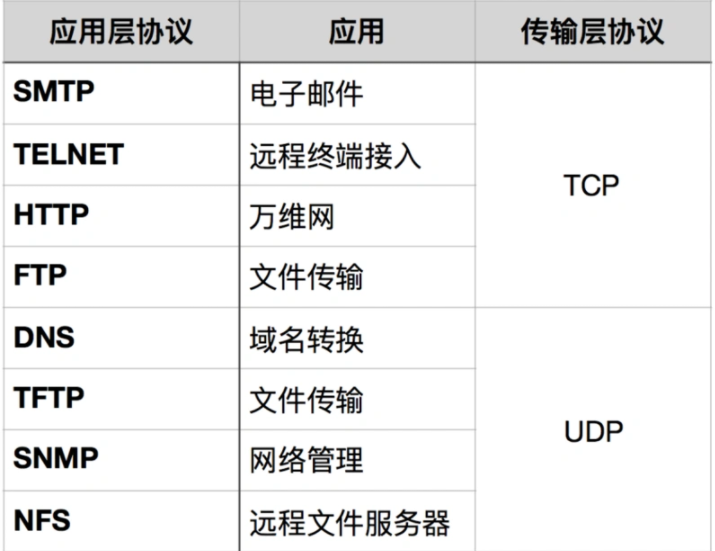
3、UDP程序结构较简单；

4、流模式（TCP）与数据报模式（UDP）；

5、TCP保证数据正确性，UDP可能丢包；

6、TCP保证数据顺序，UDP不保证。

应用场景：



### Http1 Http2 Http3的区别

HTTP1.0

浏览器和服务器只保持短暂的连接，浏览器的每次请求都需要对服务器建立一个 TCP 连接（TCP 连接的成本很高没因为每次新建都需要对服务器三次握手），服务器请求完成后立刻断开 TCP 连接，服务器也不会记录过往的每次请求。

解决方式：在请求的头信息中添加 Connection 字段并设定值为 keep-alive。

HTTP1.1

优点：

引入持久连接 — 默认支持长连接（Connection: keep-alive），即在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟

管道机制 — 同一个 TCP 连接中，客户端可以同时发送多个请求

分块传输编码 — 服务器每产生一块数据，就发送一块，用流模式取代缓存模式

新增请求方式 — PUT、DELETE等

缺点：

高延迟 — 队头阻塞(Head-Of-Line Blocking)

无状态特性 — 阻碍交互

明文传输 — 不安全性

SPDY 协议

SPDY 是由 google 推行的改进版本的 HTTP1.1

特性：

多路复用 — 解决队头阻塞

头部压缩 — 解决巨大的 HTTP 头部（静态字典、动态字典、哈夫曼编码）

请求优先级 — 先获取重要数据

服务端推送 — 填补空缺，允许服务端推送资源给客户端

强制使用 SSL 传输协议 — 提高安全性

HTTP2

HTTP2 基于 SPDY，专注于性能，最大的一个目标是在用户和网站间只用一个连接。

新增特性：

二进制分帧 — HTTP2 性能增强的核心，将请求和响应数据分割为更小的帧，并且它们采用二进制编码

多路复用 — 解决串行的文件传输和连接数过多，客户端和浏览器都可以同时发送多个请求或回应，而且不用按照顺序

HTTP 3.0 (QUIC)

QUIC (Quick UDP Internet Connections), 快速 UDP 互联网连接。

QUIC是基于UDP协议的。

关键特性：

大大缩短连接建立时间

改进的拥塞控制

无线头阻塞的多路复用

前向纠错

连接迁移

### http特点和请求方法

特点：

（1）支持客户/服务器模式

（2）简单快速：客户向服务器请求服务时，只需传送请求方法和路径。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快

（3）灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记

（4）无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间

（5）无状态：HTTP协议无法根据之前的状态进行本次的请求处理

与HTTPS的区别：

HTTPS 让HTTP运行在安全的SSL（Secure Socket Layer，安全套接层）/TLS（Transport Layer Security，传输层安全，实际上相当于SSL的后续版本）协议上，即 HTTPS = HTTP + SSL/TLS，通过 SSL证书来验证服务器的身份，并为浏览器和服务器之间的通信进行加密。

请求方法：

HTTP 定义了一组请求方法，以表明要对给定资源执行的操作。

（1）GET

GET 方法请求一个指定资源的表示形式，使用 GET 的请求应该只被用于获取数据。

（2）HEAD

HEAD 方法请求一个与 GET 请求的响应相同的响应，但没有响应体。

（3）POST

POST 方法用于将实体提交到指定的资源，通常导致在服务器上的状态变化或副作用。

（4）PUT

PUT 方法用有效载荷请求替换目标资源的所有当前表示。

（5）DELETE

DELETE 方法删除指定的资源。

（6）CONNECT

CONNECT 方法建立一个到由目标资源标识的服务器的隧道。

（7）OPTIONS

OPTIONS 方法用于描述目标资源的通信选项。

（8）TRACE

TRACE 方法沿着到目标资源的路径执行一个消息环回测试。

（9）PATCH

PATCH 方法用于对资源应用部分修改。

### 进程间通信

1.管道(pipe)

2.命名管道(FIFO)

3.消息队列(msg)

4.信号量(sem)

5.共享内存(shm)

6.套接字(Socket)

7.信号(signal)

### 死锁产生的必要条件

（1）互斥条件

（2）请求与保持条件

（3）不可剥夺条件

（4）循环等待条件

### WebSocket

WebSocket，是一种网络传输协议，位于OSI模型的应用层。可在单个TCP连接上进行全双工通信，能更好的节省服务器资源和带宽并达到实时通讯。

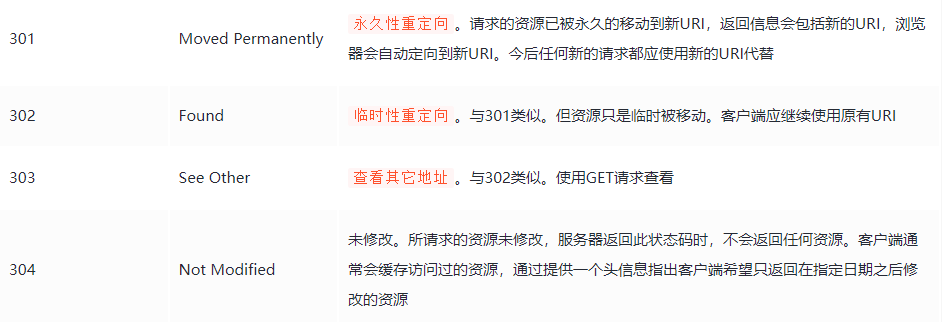
客户端和服务器只需要完成一次握手，两者之间就可以创建持久性的连接，并进行双向数据传输。

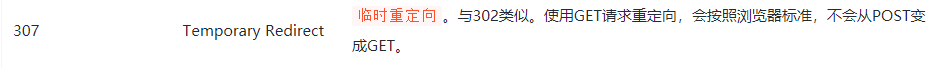
应用场景：

（1）弹幕（2）媒体聊天（3）协同编辑（4）基于位置的应用（5）体育实况更新、（7）股票基金报价实时更新

### 常见状态码



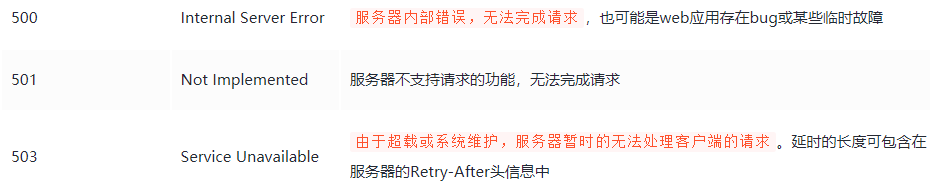






426：Upgrade Required

客户端错误响应代码指示服务器拒绝使用当前协议执行请求，但可能在客户端升级到其他协议后愿意这样做。



502：Bad Gateway

是一种服务端错误状态代码，表示作为网关或代理的服务器，从上游服务器中接收到的响应是无效的。502并不是服务端应用发出的，而是网关代理（nginx）发出的。

504：Gateway Timeout

是一种 HTTP 协议的服务器端错误状态代码，表示扮演网关或者代理的服务器无法在规定的时间内获得想要的响应。

### http报文格式简介

http 报文格式分为两种，请求报文和响应报文。

http 请求报文

一个 http 请求报文由四个部分组成：

（1）请求行（Request-Line）

请求方法（Method）

请求 URL

HTTP 协议版本

（2）请求头部（Request Header Fields）

（3）回车换行（CRLF）

（4）消息体（Message Body）

http 响应报文

一个 http 响应报文也由四个部分组成：

（1）状态行（Status-Line）

http 协议版本

状态码（Status Code）

状态码的文本描述（Reason-Phrase）

（2）响应头部（Response Header Fields）

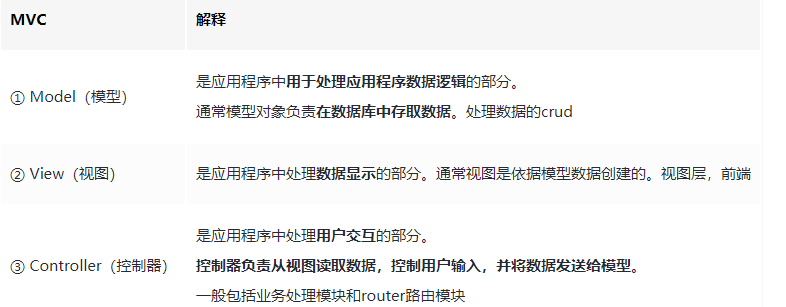
（3）回车换行（CRLF）

（4）消息体（Message Body）

### MVC和MVVM

MVC

MVC思想 ：Controller负责将Model的数据用View显示出来。



优点：

·耦合度低（运用MVC的应用程序的三个部件是相互独立的，改变其中一个不会影响其他两个）

·重用性高（多个视图可以使用同一个模型）

·生命周期成本低

·部署快（业务分工明确）

·可维护性高

缺点：

·不适合小型项目开发

·视图与控制器联系过于紧密，妨碍了它们的独立重用

MVVM

MVVM由Model、View、ViewModel三部分构成。



优点：

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

·可重用性：你可以把一些视图逻辑放在一个Model里面，让很多View重用这段视图逻辑。

·独立开发：双向数据绑定的模式，实现了View和Model的自动同步，因此开发者只需要专注对数据的维护操作即可，而不需要一直操作 dom。

MVC与MVVM有什么区别

MVVM 和 MVC都是一种设计思想。

MVVM 与 MVC 最大的区别就是：它实现了View和Model的自动同步当Model属性改变时，不用手动操作Dom元素去改变View的显示。而改变属性后，该属性对应View的显示会自动改变。

阐述一下你所理解的MVVM响应式原理

·vue是采用数据劫持配合发布者-订阅者的模式的方式。

通过Object.defineProperty()来劫持各个属性的getter和setter；

在数据变动时，发布消息给依赖收集器（dep中的subs），去通知（notify）观察者，做出对应的回调函数，更新视图。

·MVVM作为绑定的入口，整合Observer,Compile和Watcher三者。

通过Observer来监听model数据变化；

通过Compile来解析编译模板指令；

最终利用Watcher搭起Observer，Compile之间的通信桥路；

达到数据变化Observer=>视图更新；视图交互变化=>数据model变更的双向绑定效果。

### 浏览器垃圾回收

一、解决策略

（1）标记清除

标记清除分为：标记阶段和清除阶段。

首先它会遍历堆内存上所有的对象，分别给它们打上标记，然后在代码执行过程结束之后，对所使用过的变量取消标记。在清除阶段再把具有标记的内存对象进行整体清除，从而释放内存空间。

标记整理

标记整理算法会将活着的对象（即不需要清理的对象）向内存的一端移动，最后清理掉边界的内存。

（2）引用计数

引用计数是一种不常见的垃圾回收策略，其思路就是对每个值都记录其的引用次数。

当其进行循环引用时就会出现问题，会造成大量的内存不会被释放。

二、V8对于垃圾回收机制的优化

分代式垃圾回收

V8 的垃圾回收策略主要基于分代式垃圾回收机制，V8 中将堆内存分为新生代和老生代两区域，采用不同的垃圾回收器也就是不同的策略管理垃圾回收。

新生代的对象为存活时间较短的对象，简单来说就是新产生的对象，通常只支持 1～8M 的容量，而老生代的对象为存活事件较长或常驻内存的对象，简单来说就是经历过新生代垃圾回收后还存活下来的对象，容量通常比较大。

V8 整个堆内存的大小就等于新生代加上老生代的内存，对于新老两块内存区域的垃圾回收，V8 采用了两个垃圾回收器来管控。

新生代内存回收

对于新生代内存的回收，通常是通过Scavenge 算法进行垃圾回收，就是将新生代内存进行一分为二，正在被使用的内存空间称为使用区，而限制状态的内存空间称为空闲区。

原理如下：

·新加入的对象都会存放在使用区，当使用区快写满时就进行一次垃圾清理操作。

·在开始进行垃圾回收时，新生代回收器会对使用区内的对象进行标记

·标记完成后，需要对使用区内的活动对象拷贝到空闲区进行排序

·而后进入垃圾清理阶段，将非活动对象占用的内存空间进行清理

·最后对使用区和空闲区进行交换，使用区->空闲区，空闲区->使用区

新生代中的变量如果经过回收之后依然一直存在，那么会放入到老生代内存中，只要是已经经历过一次Scavenge算法回收的，就可以晋升为老生代内存的对象。

老生代内存回收

当然，Scavenge算法也有其适用场景范围，对于内存空间较大的就不适合使用Scavenge算法。此时应该使用Mark-Sweep（标记清除）和Mark-Compact（标记整理）的策略进行老生代内存中的垃圾回收。

首先是标记阶段，从一组根元素开始，递归遍历这组根元素，遍历过程中能到达的元素称为活动对象，没有到达的元素就可以判断为非活动对象。清除阶段老生代垃圾回收器会直接将非活动对象，也就是数据清理掉。

同样的标记清除策略会产生内存碎片，因此还需要进行标记整理策略进行优化。

### 浏览器缓存

浏览器缓存其实就是指在本地使用的计算机中开辟一个内存区，同时也开辟一个硬盘区作为数据传输的缓冲区，然后用这个缓冲区来暂时保存用户以前访问过的信息。

浏览器缓存过程：强缓存，协商缓存。

浏览器缓存位置一般分为四类：Service Worker-->Memory Cache-->Disk Cache-->Push Cache。

强缓存

强缓存是当我们访问URL的时候，不会向服务器发送请求，直接从缓存中读取资源，但是会返回200的状态码（浏览器会把资源缓存在memory cache 或 disk cache中）。

·Service Worker：是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用 Service Worker的话，传输协议必须为 HTTPS。

·Memory Cache：内存中的缓存，主要包含的是当前中页面中已经抓取到的资源。

·Disk Cache：存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。

·Push Cache：Push Cache（推送缓存）是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。

协商缓存

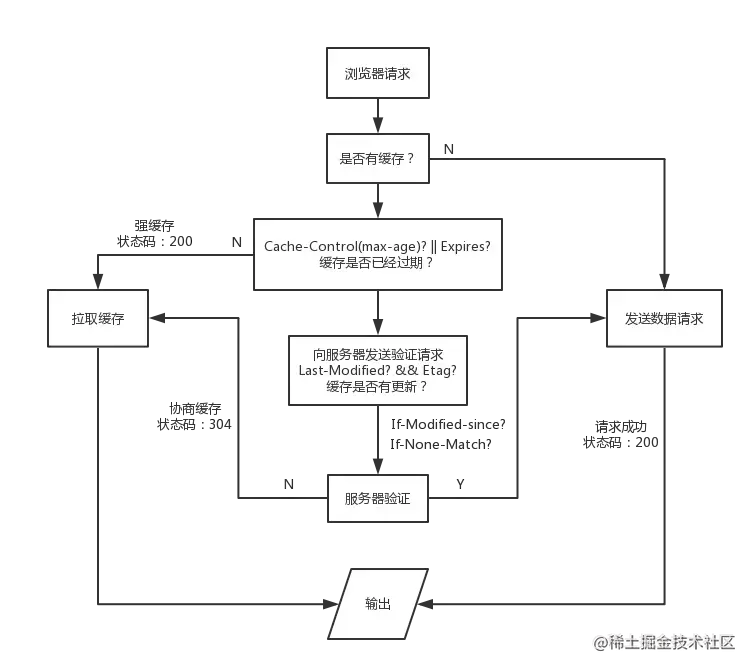
协商缓存就是强缓存失效后，浏览器携带缓存标识向服务器发送请求，由服务器根据缓存标识来决定是否使用缓存的过程。

强缓存与协商缓存的区别

1. 强缓存不发请求到服务器，所以有时候资源更新了浏览器还不知道，但是协商缓存会发请求到服务器，所以资源是否更新，服务器肯定知道。

2. 大部分web服务器都默认开启协商缓存。

过程



如何判断强缓存

第一次访问页面，浏览器会根据服务器返回的 response Header 来判断是否对资源进行缓存，如果响应头中有 cache-control 或 expires 字段，代表该资源是强缓存。

cache-control

cache-control是 HTTP1.1 中控制网页缓存的字段。

主要取值

public：资源客户端和服务器都可以缓存

privite：资源只有客户端可以缓存

no-cache：客户端缓存资源，但是是否缓存需要经过协商缓存来验证

no-store：不使用缓存

max-age：缓存保质期，是相对时间

Expires

Expires是HTTP1.0控制网页缓存的字段，值为一个时间戳，服务器返回该资源缓存的到期时间。

Expires 有个缺点，就是它判断是否过期是用本地时间来判断的，而本地时间是可以自己修改的。

注意：当 Cache-Control 与 expires 两者都存在时，Cache-Control 优先级更高

协商缓存触发条件

Cache-Control 的值为 no-cache（协商缓存）或者 Cache-Control: max-age=0。

Last-Modified

文件在服务器最后被修改的时间，从服务器 Respnse Headers 上获取。

客户端再次发起该请求时，请求头 If-Modified-Since 字段会携带上次请求返回的 Last-Modified 值。

服务器根据 if-modified-since 的值，与该资源在服务器最后被修改时间做对比，若服务器上的时间大于 Last-Modified 的值，则重新返回资源，返回200，表示资源已更新；反之则返回304，代表资源未更新，可继续使用缓存

Etag

当前资源文件的一个唯一标识(由服务器生成)，若文件内容发生变化该值就会改变。

（1）第一次访问页面时，服务器的响应头会返回 etag 字段；

（2）客户端再次发起该请求时，请求头 If-None-Match 字段会携带上次请求返回的 etag 值；

（3）服务器根据 If-None-Match 的值，与该资源在服务器的Etag值做对比，若值发生变化，状态码为200，表示资源已更新；反之则返回304，代表资源无更新，可继续使用缓存。

为什么要有 Etag

Etag 的出现主要是为了解决一些 Last-Modified 难处理的问题：

（1）一些文件也许会周期性的更改，但是内容并不改变(仅仅改变的修改时间)，这时候并不希望客户端认为这个文件被修改了而重新去请求；

（2）某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说 1s 内修改了 N 次)，If-Modified-Since 能检查到的粒度是秒级的，使用 Etag 就能够保证这种需求下客户端在 1 秒内能刷新 N 次 cache。

注意：Etag 优先级高于 Last-Modified，若 Etag 与 Last-Modified 两者同时存在，服务器优先校验 Etag。

### 浏览器渲染

解析html、构建dom树 -> 解析css、构建cssom树 -> 构建render树 -> 布局render树 -> 绘制render树

网页请求从 HTML 文件请求开始。服务器返回 HTML —— 响应头和数据。然后浏览器开始解析 HTML，转换收到的数据为 DOM 树。浏览器每次发现外部资源就初始化请求，无论是样式、脚本或者嵌入的图片引用。有时请求会阻塞，这意味着解析剩下的 HTML 会被终止直到重要的资源被处理。浏览器接着解析 HTML，发请求和构造 DOM 直到文件结尾，这时开始构造 CSS 对象模型。等到 DOM 和 CSSOM 完成之后，浏览器构造渲染树，计算所有可见内容的样式。一旦渲染树完成布局开始，定义所有渲染树元素的位置和大小。完成之后，页面被渲染完成，或者说是绘制到屏幕上。

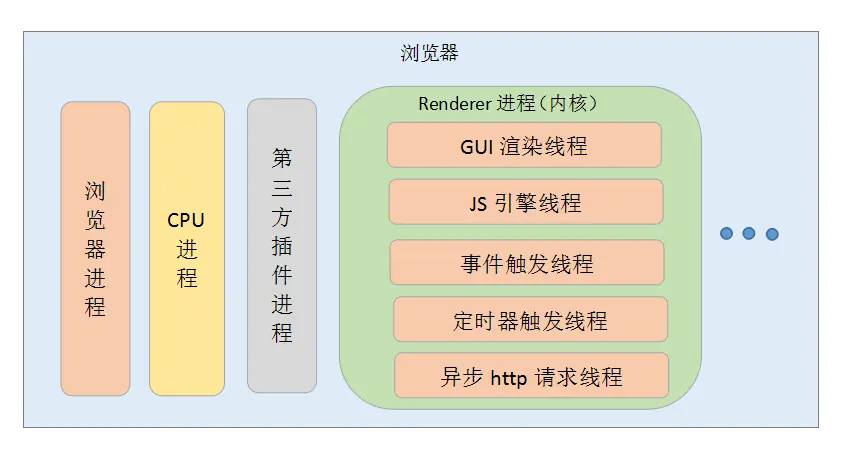
注意：在head中如果发现CSS和JS链接，就会发请求。先解析CSS形成CSSOM，此过程会屏蔽JS引擎，相当于给DOM上锁，防止CSS、JS同时修改的现象发生。形成CSSOM后会取消屏蔽 JS引擎，然后接收并执行JavaScript，然后我们就可以继续构建DOM。获取DOM和CSSOM后，我们将合并二者并构建RenderTree，然后绘制网页。

### 浏览器中的线程与进程

浏览器从关闭到启动，然后新开一个页面至少需要：1个浏览器进程，1个GPU进程，1个网络进程，和1个渲染进程，一共4个进程。

最新的Chrome浏览器包括：1个浏览器主进程，1个GPU进程，1个网络进程，多个渲染进程，和多个插件进程。

浏览器进程和线程关系图：



### 短信发送原理

手机A编辑好短信后，按照协议规定把短信打包，包内包含短信内容、目的地号码、编码格式、类型等信息，然后发送给A基站。基站收到数据包后，把它转发给短信中心(MSC)。短信中心再根据短信的目的地号码派发给此号码所在的B基站，B基站再发给手机终端B。手机终端收到短信后按照协议规定的格式解码，然后提示用户收到短信。

### 浏览器输入一行url会发生什么

1. 查询DNS(域名解析)，获取域名对应的IP地址；

（1）查询浏览器缓存

（2）检查系统缓存，检查hosts文件

（3）检查路由器缓存

（4）查询ISP DNS 缓存

（5）将请求发向根域名服务器进行搜索

2. 浏览器与服务器建立tcp链接（三次握手）；

3. 浏览器向服务器发送http请求(请求和传输数据）；

4. 服务器接受到相应请求后，根据路径参数，经过后端的一些处理返回 HTTP 报文；

5. 浏览器解析渲染页面；

（1）根据 HTML 解析出 DOM 树

（2）浏览器向服务器发送请求获取嵌入在HTML中的对象

（3）根据 CSS 解析生成 CSS 规则树

（4）结合 DOM 树和 CSS 规则树，生成渲染树

（5）根据渲染树计算每一个节点的信息

（6）根据计算好的信息绘制页面

6.关闭TCP连接（四次挥手）。

### 服务端推送

构建一个实时的Web应用程序，需要考虑如何将数据从服务器发送到客户端。实现这种“主动”的技术已经存在了相当长的一段时间，并且仅限于两种通用方法：客户端拉取或服务器推送。

实现这些有几种方法：

·长/短轮询（客户端拉取）

·WebSockets（服务器发送）

·Server-Sent Events（服务器发送）

客户端拉取——客户端要求服务器定期更新

服务器推送——服务器主动向客户端推送更新（与客户端拉取相反）

（1）轮询

轮询是一种客户机定期向服务器请求新数据的技术。我们可以用两种方式进行轮询：短轮询和长轮询。

简单地说，短轮询是一个基于Ajax的计时器，它以固定的延迟进行调用，而长轮询则基于Comet（即当服务器事件发生时，服务器将实时向客户机发送数据）。

（2）WebSocket

在Javascript中创建了WebSockets之后，会有一个 Http 的 Upgrade 请求发送到服务器，在取得服务器响应后，建立的连接会从 Http升级，从 Http协议转换为WebSocket协议。

过程：客户端发送 Http GET请求，upgrade —— 服务器响应给客户端 switching protocol【Http => WebSocket】—— 可以进行 WebSocket 通信。

（3）SSE (Server-Sent Events)

SSE是一种机制，允许服务器在建立客户端-服务器连接后将数据异步推送到客户端。然后，服务器可以决定在新的“数据块”可用时发送数据。它可以看作是一个单向发布订阅模型。

Server-Sent Events 规范描述了一个内建的类 EventSource，它能保持与服务器的连接，并允许从中接收事件。与 WebSocket 类似，其连接是持久的。

与WebSocket的区别



### CSR、SSR和SSG

CSR

CSR全称是 Client Side Rendering，代表的是客户端渲染。顾名思义，就是在渲染工作在客户端（浏览器）进行，而不是在服务器端进行。

客户端渲染的优缺点

优点

（1）前后端分离。前端专注于界面开发，后端专注于api开发，且前端有更多的选择性，可以使用vue，react框架开发，而不需要遵循后端特定的模板。

（2）服务器压力变轻了，渲染工作在客户端进行，服务器直接返回不加工的html。

（3）用户在后续访问操作体验好，（首屏渲染慢）可以将网站做成SPA，可以增量渲染。

缺点

（1）不利于SEO，因为搜索引擎不执行JS相关操作，无法获取渲染后的最终html。

（2）首屏渲染时间比较长，因为需要页面执行ajax获取数据来渲染页面，如果请求接口多，不利于首屏渲染。

SSR

SSR全称是 Server Side Rendering，代表的是服务端渲染。与客户端渲染不同的是，SSR输出的是一个渲染完成的html，整个渲染过程是在服务器端进行的。例如传统的JSP，PHP都是服务端渲染

服务端渲染的优缺点

优点

（1）有利于SEO，由于页面在服务器生成，搜索引擎直接抓取到最终页面结果。

（2）有利于首屏渲染，html所需要的数据都在服务器处理好，直接生成html，首屏渲染时间变短。

缺点

（1）占用服务器资源，渲染工作都在服务端渲染。

（2）用户体验不好，每次跳转到新页面都需要在重新服务端渲染整个页面，不能只渲染可变区域。

SSG

SSG全称是 Static Site Generation ，代表的是静态站点生成。在构建的时候直接把结果页面输出html到磁盘，每次访问直接把html返回给客户端，相当于一个静态资源。

静态站点生成的优缺点

优点

（1）减轻服务器压力，可以把生成的静态资源（html）放到CDN上，合理利用缓存

（2）有利于SEO，由于html已经提前生成好，不需要服务端和客户端去渲染

缺点

（1）只适用于静态数据，对于经常改动的数据，需要每次重新生成页面。

（2）用户体验不好，每次打开新页面都需要重新渲染整个页面，不能只渲染可变区域。

### 单点登录SSO

单点登录（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一。

SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

SSO 一般都需要一个独立的认证中心（passport），子系统的登录均得通过passport，子系统本身将不参与登录操作。

当一个系统成功登录以后，passport将会颁发一个令牌给各个子系统，子系统可以拿着令牌去获取各自的受保护资源。为了减少频繁认证，各个子系统在被passport授权以后，会建立一个局部会话，在一定时间内可以无需再次向passport发起认证。

同域名下的单点登录

cookie的domain属性设置为当前域的父域，并且父域的cookie会被子域所共享。path属性默认为web应用的上下文路径。

利用 Cookie 的这个特点，只需要将Cookie的domain属性设置为父域的域名（主域名），同时将 Cookie的path属性设置为根路径，将 Session ID（或 Token）保存到父域中。这样所有的子域应用就都可以访问到这个Cookie。

不同域名下的单点登录

方法一（标准做法）

如果是不同域的情况下，Cookie是不共享的，这里我们可以部署一个认证中心，用于专门处理登录请求的独立的 Web服务。

（1）用户统一在认证中心进行登录，登录成功后，认证中心记录用户的登录状态，并将 token 写入 Cookie（这个 Cookie是认证中心的，应用系统是访问不到的）。

（2）应用系统检查当前请求有没有 Token，如果没有，说明用户在当前系统中尚未登录，那么就将页面跳转至认证中心。由于这个操作会将认证中心的 Cookie 自动带过去，因此，认证中心能够根据 Cookie 知道用户是否已经登录过了。

（3）如果认证中心发现用户尚未登录，则返回登录页面，等待用户登录。

（4）如果发现用户已经登录过了，就不会让用户再次登录了，而是会跳转回目标 URL，并在跳转前生成一个 Token，拼接在目标URL的后面，回传给目标应用系统。

（5）应用系统拿到 Token之后，还需要向认证中心确认下 Token 的合法性，防止用户伪造。确认无误后，应用系统记录用户的登录状态，并将 Token写入Cookie，然后给本次访问放行（这个 Cookie 是当前应用系统的）。当用户再次访问当前应用系统时，就会自动带上这个 Token，应用系统验证 Token 发现用户已登录，于是就不会有认证中心什么事了。

此种实现方式相对复杂，支持跨域，扩展性好，是单点登录的标准做法。

方法二

可以选择将 Session ID （或 Token ）保存到浏览器的 LocalStorage 中，让前端在每次向后端发送请求时，主动将LocalStorage的数据传递给服务端。

这些都是由前端来控制的，后端需要做的仅仅是在用户登录成功后，将 Session ID（或 Token）放在响应体中传递给前端。前端拿到 Session ID（或 Token ）后，除了将它写入自己的 LocalStorage 中之外，还可以通过特殊手段将它写入多个其他域下的 LocalStorage 中。

### 前端安全

CIA三元组

（1）机密性（Confidentiality）

（2）完整性（Integrity）

（3）可用性（Availability）

XSS攻击

定义

黑客在用户的浏览器中插入一段恶意 JavaScript 脚本，窃取用户的隐私信息、冒充用户的身份进行操作，这就是 XSS 攻击(Cross-Site Scripting，跨站脚本攻击)。

类型

（1）反射型 XSS (也叫非持久型)

恶意 JavaScript 脚本属于用户发送给网站请求中的一部分，随后网站又将这部分返回给用户，恶意脚本在页面中被执行。一般发生在前后端一体的应用中，服务端逻辑会改变最终的网页代码。目前更流行前后端分离的项目，反射型 XSS 无用武之地。

（2）基于 DOM 的 XSS

这种攻击不需要经过服务器，网页本身的 JavaScript 也是可以改变 HTML 的，黑客正是利用这一点来实现插入恶意脚本。

（3）存储型 XSS (也叫持久型 XSS)

黑客将恶意 JavaScript 脚本长期保存在服务端数据库中，用户一旦访问相关页面数据，恶意脚本就会被执行。常见于搜索、微博、社区贴吧评论等。

区别

反射型的 XSS 的恶意脚本存在 URL 里，存储型 XSS 的恶意代码存在数据库里。

而基于DOM型的XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，其他两种 XSS 都属于服务端的安全漏洞。

危害

·盗取用户 Cookie

·未授权操作

·修改 DOM

·刷浮窗广告

·发动 XSS 蠕虫攻击

·劫持用户行为，进一步渗透内网

·…

防护

·一切用户输入皆不可信，在输出时进行验证

·将 HTML 元素内容、属性以及 URL 请求参数、CSS 值进行编码

·当编码影响业务时，使用白名单规则进行检测和过滤

·使用 W3C 提出的 CSP (Content Security Policy，内容安全策略)，定义域名白名单

·设置 Cookie 的 HttpOnly 属性

CSRF攻击

定义

CSRF 英文全称是 Cross-site request forgery，又称为“跨站请求伪造”。

攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证， 达到冒充用户对被攻击的网站执行某项操作的目的。

实现

（1）最容易实现的是 Get 请求，一般进入黑客网站后，可以通过设置 img的 src 属性来自动发起请求。

（2）在黑客的网站中，构造隐藏表单来自动发起 Post 请求。

（3）通过引诱链接诱惑用户点击触发请求，利用 a 标签的 href。

CSRF攻击和XSS攻击的区别

CSRF 攻击不需要将恶意代码注入用户的页面，仅仅是利用服务器的漏洞和用户的登录状态来实施攻击。

CSRF 攻击成本也比 XSS 低，用户每天都要访问大量网页，无法确认每一个网页的合法性，

从用户角度来说，无法彻底防止 CSRF 攻击。

防护

·针对实际情况，设置关键 Cookie 的 SameSite 属性为 Strict 或 Lax（宽松模式）

·服务端验证请求来源站点(Referer、Origin)

·使用 CSRF Token，服务端随机生成返回给浏览器的 Token，每一次请求都会携带不同的 CSRF Token

·加入二次验证(独立的支付密码)

XSSI攻击

定义

XSSI全称Cross-Site Script Inclusion，在命名上和 XSS（Cross-Site Script）相似，但漏洞利用上和 CSRF 更为相像。

实现

当一个网站通过 <script> 标签请求 js 文件时，服务器所返回的 js 文件中若包含敏感信息，那么可以通过 js 间接的方式获取到这些敏感信息，则称为 XSSI。可以诱使受害者包含含有敏感信息的 js 文件，并通过 xhr 将信息传送到攻击者控制的服务器上。

XSSI攻击和CSRF攻击的区别

在CSRF中，攻击者想要在受害者的页面中执行一个状态改变的动作，比如在一个在线银行应用中进行转帐。在XSSI中攻击者想要跨域泄露数据，以便然后再执行上述。

防护

开发者永远也不要把敏感数据放在Javascript文件中，也不要放在JSONP中。

Web密码学

对称加密算法

对称加密算法就是加密和解密使用同一个密钥。

常见的经典对称加密算法有 DES、AES(AES-128)、IDEA、国密SM1、国密SM。4

非对称加密算法

非对称加密就是加密和解密使用不同的密钥。发送方使用公钥对信息进行加密，接收方收到密文后，使用私钥进行解密。

主要解决了密钥分发的难题，常说的签名就是私钥加密。

常见的经典非对称加密算法有RSA、ECC和国密SM2。

散列算法

不可逆性、鲁棒性、唯一性。

MD5、SHA(SHA-256)、国密SM3。

使用时记得加盐。

### token组成、特点和存储位置

server 会有一套校验机制，校验 token 是否合法。

token 本身携带 uid 信息。

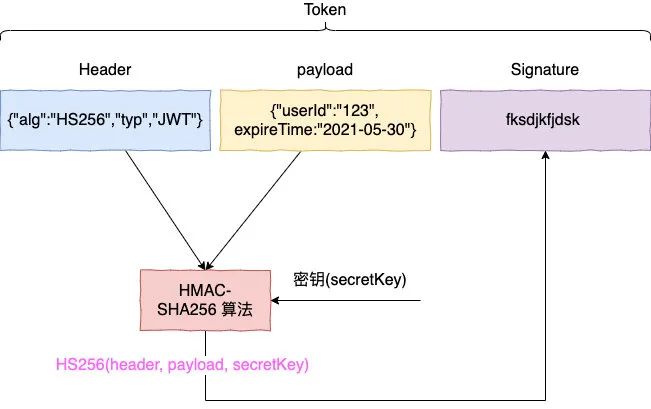
校验 token：可以借鉴 HTTPS 的签名机制来校验。

JWT ( JSON Web Token ) token组成

header：指定了签名算法。

payload：可以指定用户 id，过期时间等非敏感数据。

Signature: 签名。server 根据 header 知道它该用哪种签名算法，再用密钥根据此签名算法对 head + payload 生成签名，这样一个 token 就生成了。



特点

token 具有存储实现简单，扩展性好的特点。

存储位置

token在用户登录成功之后会返回给客户端，客户端主要有以下几种存储方式：

（1）存储在localStorage中，每次调用接口的时候都把它当成一个字段传给后台；

（2）存储在cookie中，让它自动发送，不过缺点就是不能跨域；

（3）拿到之后存储在localStorage中，每次调用接口的时候放在HTTP请求头的Authorization字段里面。

存储在cookie中：不符合Restful 最佳实践，容易遭受CSRF攻击。

存储在localStorage中：容易受到XSS攻击。

### cookie和session的区别

Cookie：有时也用其复数形式 Cookies。类型为“小型文本文件”，是某些网站为了辨别用户身份，进行 Session 跟踪而储存在用户本地终端上的数据（通常经过加密），由用户客户端计算机暂时或永久保存的信息。

Session（会话机制）：能识别哪个请求由哪个用户发起的机制，生成的能识别用户身份信息的字符串称为 sessionId。

cookie 是存储在 client 的，而 session 保存在 server，sessionId需要借助 cookie 的传递才有意义。

### 禁止脚本操作cookie

HttpOnly：包含在Set-Cookie HTTP响应头文件中的附加标识，生成cookie时使用HttpOnly标识有助于降低客户端脚本访问受保护cookie的风险（如果浏览器支持的话）。

设置Cookie的HttpOnly属性为true，可以禁止javascript脚本操作cookie。

### 跨域请求

浏览器安全的基石是「同源策略」（same-origin policy）。

同源即：

·协议相同

·域名（主机名）相同

·端口相同

同源政策是为了保证用户信息的安全，防止恶意的网站窃取数据。

一个源访问另一个源的资源时即为「跨源」，最常见的跨源场景是域名不同，即常说的“跨域”。

注意：服务器与服务器之间请求数据并不会存在跨域行为，跨域行为是浏览器安全策略限制。

跨域方法

JSONP

JSONP（JSON with Padding）的原理非常简单，就是HTML标签中，很多带src属性的标签都可以跨域请求内容，比如我们熟悉的img图片标签、script标签。可以利用script标签来执行跨域的javascript代码。通过这些代码，我们就能实现前端跨域请求数据。

优点：

JSONP 的兼容性很好，在古老的浏览器中都可以运行。

缺点：

·JSONP 只支持 GET 请求，不支持 POST 请求等其他类型的 HTTP 请求；

·请求过程无法终止，导致弱网络下处理超时请求比较麻烦；

·无法捕获服务端返回的异常信息。

document.domain

利用document.domain 可以修改访问页面的域，多用在父子关系的域，子域可以设置document.domain等于父域，从而解决同父域跨域问题。

例如：

www.ios.xingyu.com 访问 www.web.xingyu.com 跨域，可以利用 document.domain 设置成 [www.xingyu.com](http://www.xingyu.com) 。

CORS

CORS（Cross-Origin Resource Sharing，跨源资源共享）是一种基于 HTTP 头的机制，该机制通过允许服务器标示除了它自己以外的其他源（域、协议或端口），使得浏览器允许这些源访问加载自己的资源。

如：

（1）Access-Control-Allow-Origin 参数指定了源，告诉浏览器允许该源访问资源

response.setHeader("Access-Control-Allow-Origin","http://google.com")

（2）Access-Control-Allow-Headers 标头字段用于预检请求的响应，其指明了实际请求中允许携带的标头字段

response.setHeader("Access-Control-Allow-Headers","X-Request-Width")

（3）Access-Control-Allow-Methods 标头字段指定了访问资源时允许使用的请求方法，用于预检请求的响应

response.setHeader("Access-Control-Allow-Methods","PUT,POST,GET,DELETE,OPTIONS")

（4）Access-Control-Allow-Credentials 头指定了当浏览器的 credentials 设置为 true 时是否允许浏览器读取 response 的内容（是否接收跨域的 cookie）

response.setHeader("Access-Control-Allow-Credentials","true")

Nginx 配置代理

·前端和运维商量好协议路径代理规则，比如/api 代表域名

·前端配置webpack-dev-server 代理

·服务器利用ngnix 配置相同转发代理

CORB

CORB（Cross-Origin Read Blocking，跨源读取阻止）

浏览器在加载可以跨域资源时，在资源载入页面之前，对其进行识别和拦截的算法。拦截加载跨域资源（如JSONP），提升攻击者进行幽灵攻击的成本。

当前有三种内容类型受保护，分别是 JSON、HTML 和 XML。

### HTTP常见的请求头

HTTP头字段（HTTP header fields）,是指在超文本传输协议（HTTP）的请求和响应消息中的消息头部分，它们定义了一个超文本传输协议事务中的操作参数。

常见请求头

Accept

能够接受的回应内容类型（Content-Types），比如 Accept: text/plain

Accept-Charset

能够接受的字符集，比如 Accept-Charset: utf-8

Accept-Encoding

能够接受的编码方式列表，比如 Accept-Encoding: gzip, deflate

Accept-Language

能够接受的回应内容的自然语言列表，比如 Accept-Language: en-US

Cache-Control

用来指定在这次的请求/响应链中的所有缓存机制都必须遵守的指令

比如 Cache-Control: no-cache

Connection

该浏览器想要优先使用的连接类型，比如 Connection: keep-alive、Connection: Upgrade

Cookie

服务器通过 Set- Cookie发送的一个小型文本文件，比如 Cookie: $Version=1; Skin=new;

Content-Length

以八位字节数组（8位的字节）表示的请求体的长度，比如 Content-Length: 348

Content-Type

请求体的多媒体类型，比如 Content-Type: application/x-www-form-urlencoded

If-Modified-Since

允许在对应的内容未被修改的情况下返回304未修改

比如 If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

If-None-Match

允许在对应的内容未被修改的情况下返回304未修改

比如 If-None-Match: "737060cd8c284d8af7ad3082f209582d"

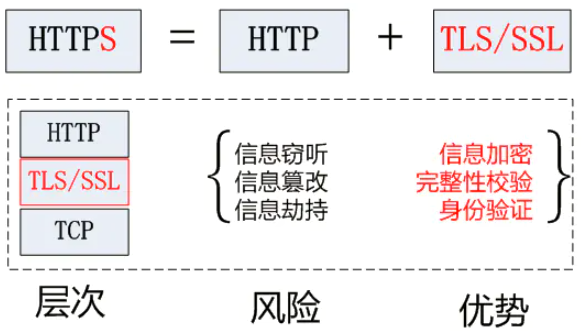
### HTTPS如何保证安全

HTTP在通信过程中，存在以下问题：

·通信使用明文（不加密），内容可能被窃听

·不验证通信方的身份，因此有可能遭遇伪装

而HTTPS的出现正是解决这些问题，HTTPS是建立在SSL之上，其安全性由SSL来保证。在采用SSL后，HTTP就拥有了HTTPS的加密、证书和完整性保护这些功能。



实现

SSL实现这些功能主要依赖的手段

（1）对称加密

采用协商的密钥对数据加密。

（2）非对称加密

实现身份认证和密钥协商。

（3）混合加密

具体做法是发送密文的一方使用对方的公钥进行加密处理“对称的密钥”，然后对方用自己的私钥解密拿到“对称的密钥”。

（4）摘要算法

摘要算法用于验证信息的完整性，类似于常说的散列函数、哈希函数。

可以理解成一种特殊的压缩算法，它能够把任意长度的数据“压缩”成固定长度、而且独一无二的“摘要”字符串，就好像是给这段数据生成了一个数字“指纹”。

摘要算法保证了“数字摘要”和原文是完全等价的。所以，我们只要在原文后附上它的摘要，就能够保证数据的完整性。

（5）数字签名

数字签名用于身份验证。

数字签名能确定消息确实是由发送方签名并发出来的，因为别人假冒不了发送方的签名。

原理其实很简单，就是用私钥加密，公钥解密。

同时引入第三方证书机构，确保公开秘钥的安全性。

### AOP模式

AOP（Aspect Oriented Program，面向切面编程），主要作用是把一些和跟核心业务逻辑无关的功能包括日志统计、打点上报、异常处理等抽离出来后，再通过动态织入的方式掺入业务逻辑模块中。

如：

// 验证函数

function validate(value) {

if (!value.length) {

return false;

}

if (value.length > 10) {

return false;

}

return true;

}

// before函数

Function.prototype.before = function (beforeFn) {

// 保存对原函数的引用

let self = this;

// 返回包含原函数和参数函数的 代理函数

return function () {

// 先执行 beforeFn

let ret = beforeFn.apply(this, arguments);

// 当beforeFn返回false不再往下走了

if (!ret) {

return ret;

}

// 再执行原函数

return self.apply(this, arguments);

};

};

// 定义提交函数

let submit = function (value) {

form.submit(value);

};

// 添加 before 切面函数

submit = submit.before(validate);

submit(value);

### JSON

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，因其简洁、易读、易于解析的特点，在前后端数据交互、API 设计、配置文件等领域广泛应用。

优点

（1）简洁易读、使用方便

（2）跨平台、可拓展性

（3）体积小、解析速度快

JSON数据压缩实现

CJSON

CJSON 的压缩算法，主要是将资料抽离成Template与Value，节省掉重复的“Key 值”。

HPack

HPack 的压缩算法，也是将 Key、Value 抽离，阵列中第一个值，就是 HPack 的 Template，后面依序就是 Value。

移除空白字符

简化键名

## Git

### 定义

git是一个分布式版本控制软件，最初目的是为更好地管理Linux内核开发而设计。

分布式版本控制系统的客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。

### 代码回滚

git reset

git reset用于回退版本，可以遗弃不再使用的提交。

执行遗弃时，需要根据影响的范围而指定不同的参数，可以指定是否复原索引或工作树内容。

如：

git reset < commit\_id >

git revert

git revert 撤销某次操作，此次操作之前和之后的 commit和history都会保留，并且把这次撤销，作为一次最新的提交。

如：

git revert <commit\_id>

两者主要区别如下：

（1）git revert是用一次新的commit来回滚之前的commit，git reset是直接删除指定的commit；

（2）git reset 是把HEAD向后移动了一下，而git revert是HEAD继续前进，只是新的commit的内容和要revert的内容正好相反，能够抵消要被revert的内容；

（3）在回滚这一操作上看，效果差不多。但是在日后继续 merge 以前的老版本时有所区别。

### git rebase 和 git merge

在使用 git 进行版本管理的项目中，当完成一个特性的开发并将其合并到 master 分支时，会有两种方式：

（1）git merge

（2）git rebase

git merge

将当前分支合并到指定分支：

git merge xxx

git merge会把两个分支的最新快照以及二者最近的共同祖先进行三方合并，合并的结果是生成一个新的快照。

git rebase

将当前分支移植到指定分支或指定commit之上：

git rebase -i <commit>

常见的参数有--continue，用于解决冲突之后，继续执行rebase：

git rebase –continue

git rebase会找到不同的分支的最近共同祖先，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件（老的提交X和Y也没有被销毁，只是简单地不能再被访问或者使用）。然后将当前分支指向目标最新位置D, 然后将之前另存为临时文件的修改依序应用。

区别

merge

通过merge合并分支会新增一个merge commit，然后将两个分支的历史联系起来。

其实是一种非破坏性的操作，对现有分支不会以任何方式被更改，保持源分支的原始上下文，但是会导致历史记录相对复杂。

rebase

rebase会将整个分支移动到另一个分支上，有效地整合了所有分支上的提交。

主要的好处是历史记录更加清晰，是在原有提交的基础上将差异内容反映进去，消除了 git merge所需的不必要的合并提交，但是可能会丢失上下文。

### 常用指令

git branch

查看本地所有分支。

git branch -r

查看远程所有分支。

git branch -a

查看本地和远程所有分支。

git merge <分支名>

合并分支。

git checkout <分支名>

切换到本地某个分支。

git checkout -b <新分支名>

基于当前分支新建分支，并切换为这个分支。

git branch -D <本地分支名>

删除本地分支。

git push origin –d <远程分支名>

删除远程分支。

git stash

暂时将未提交的变化移除。

git stash pop

取出储藏中最后存入的工作状态进行恢复，会删除储藏。

git remote -v | --verbose

列出已经存在的远程分支详细信息，在每一个名字后面列出其远程url，显示对应的克隆地址。

## JavaScript

### 基本数据类型

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

这些数据可以分为原始数据类型和引用数据类型：

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

（2）堆：引用数据类型（对象、数组和函数）

两种类型的区别在于存储位置的不同：

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

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

### ES6新特性

（1）let 和 const 的块级作用域

（2）数组的解构：根据位置进行获取的

（3）对象的解构：根据属性名进行获取

（4）模板自变量 ` ` ：

·支持换行;

·采用${}的方式进行插值表达式，{}中的内容就是js代码;

·在模板自变量之前，我们可以加一个标签函数用来对模板自变量的值进行处理。

（5）默认参数和剩余参数

·给参数设置默认值的特性

·剩余操作符、spread用法

（6）箭头函数

（7）引入新的基本数据类型symbol

…

### 闭包

闭包是指一个函数可以记住其外部变量并可以访问这些变量。

JavaScript 中的函数会自动通过隐藏的 [[Environment]] 属性记住创建它们的位置，所以它们都可以访问外部变量。

### 节流和防抖

节流

定时器版

function throttle(fn, delay) {

let timer = null;

return function() {

if(!timer) {

fn.apply(this, arguments);

timer = setTimeout(() => {

timer = null;

}, delay);

}

}

}

时间戳版

function throttle(fn, delay){

let last = Date.now(); // 使用时间戳实现

return function (){

let now = Date.now();

if(now - last >= delay){

fn.apply(this, arguments);

last = now;

}

}

}

防抖

function debounce(fn, delay) {

let timer;

return function() {

let this\_ = this;

clearTimeout(timer);

timer = setTimeout(() => {

fn.apply(this\_, arguments)

}, delay);

}

}

如：let f = debounce((x) => console.log(x), 1000);

### 手写Promise.all

var promiseAll = function(functions) {

const len = functions.length;

const res = new Array(len);

let count = 0;

return new Promise((resolve, reject) => {

functions.forEach((fn, index) => {

fn().then((val) => {

res[index] = val;

count++;

if(count == len) {

resolve(res);

}

}).catch(reject);

})

});

};

### 事件循环

JS的运行机制是事件循环。

在JavaScript中，所有的任务都可以分为：

（1）同步任务：立即执行的任务，同步任务一般会直接进入到主线程中执行。

（2）异步任务：异步执行的任务，比如ajax网络请求，setTimeout定时函数等。

步骤

（1）首先清空调用栈中的同步代码。

而异步任务的返回结果会被放到一个任务队列中，根据异步事件的类型，这个事件实际上会被放到对应的宏任务和微任务队列中去。

（2）在当前调用栈为空时，主线程会依次执行微任务队列中的事件对应的回调，直到微任务队列为空，（尝试DOM渲染），然后去宏任务队列中取出一个事件执行。

微任务执行时机比宏任务要早，微任务在DOM渲染前触发，宏任务在DOM渲染后触发。

宏任务

setTimeout

setTimeout() 方法设置一个定时器，一旦定时器到期，就会执行一个函数或指定的代码片段。

setInterval

setInterval() 方法重复调用一个函数或执行一个代码片段，在每次调用之间具有固定的时间间隔。

setImmediate

setImmediate() 相当于使用 setTimeout(() => {}, 0)，主要用于配合 Node.js Event Loop。

注意：setImmediate不是标准的，在大部分浏览器上不可用。

postMessage

postMessage() 方法可以安全地实现跨源通信。

I/O

I/O指的是JavaScript 中处理输入/输出操作的任务。这些任务通常涉及与浏览器环境交互，比如从服务器加载数据、读取本地文件或者与用户界面进行交互。

微任务

Promise.then

Promise 实例的 then() 方法最多接受两个参数：用于 Promise 兑现和拒绝情况的回调函数。它立即返回一个等效的 Promise 对象，允许你链接到其他 Promise 方法，从而实现链式调用。

process.nextTick

Node 执行完所有同步任务，接下来就会执行process.nextTick的任务队列。

process.nextTick的回调函数执行早于Promise。

MutationObserver

MutationObserver 接口提供了监视对 DOM 树所做更改的能力。

采用的是异步的监听方式，所有的操作会统一放在回调中，当有操作时在下一个微任务执行时会触发监听回调。或者可以理解为：一个节点同时进行多个操作时，其变化会被记录到一个异步队列中，最终一次性展示，这样做既不会影响页面加载，也保证了DOM变化的监听。

### 创建对象的几种方法

1、对象字面量{...}

2、构造函数

3、new Object()

4、Object.create()

Object.create() 和new Object() 创建对象的区别

Object.create()创建的对象只是原型指向源对象，并不会继承它的任何属性；而new出来的对象会继承原型对象的属性和方法。

### Symbol的概念及应用场景

Symbol是由ES6规范引入的一项新特性，它的功能类似于一种标识唯一性的ID。通常情况下，我们可以通过调用Symbol()函数来创建一个Symbol实例：

let s = Symbol()

Symbol类型的key不能通过Object.keys()或者for...in来枚举的，它未被包含在对象自身的属性名集合(property names)之中。

获取以Symbol方式定义的对象属性：

// 使用Object的API

Object.getOwnPropertySymbols(obj) // [Symbol(name)]

// 使用新增的反射API

Reflect.ownKeys(obj) // [Symbol(name), 'age', 'title']

应用场景：

（1）使用Symbol来作为对象属性名(key)

（2）使用Symbol来替代常量

（3）使用Symbol定义类的私有属性/方法

### JavaScript 的继承类型

（1）原型链继承

核心：将父类的实例作为子类的原型。

缺点：两个实例使用的是同一个原型对象，内存空间是共享的。

（2）构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类。

缺点：只能继承父类的实例属性和方法，不能继承原型属性或者方法。

（3）组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用。

缺点：造成两次构造的性能开销。

（4）原型式继承

核心：主要借助Object.create方法实现普通对象的继承。

缺点：Object.create方法实现的是浅拷贝，多个实例的引用类型属性指向相同的内存，存在篡改的可能。

（5）寄生组合继承

核心：通过寄生方式，砍掉父类的实例属性。

优点：在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点。

（6）es6继承 —— 使用“extend”

### 判断一个字符串是否是正则表达式

const isRegExp = (v) => {

return Object.prototype.toString.call(v) === '[object RegExp]';

}

或者使用 reg instanceof RegExp

如：

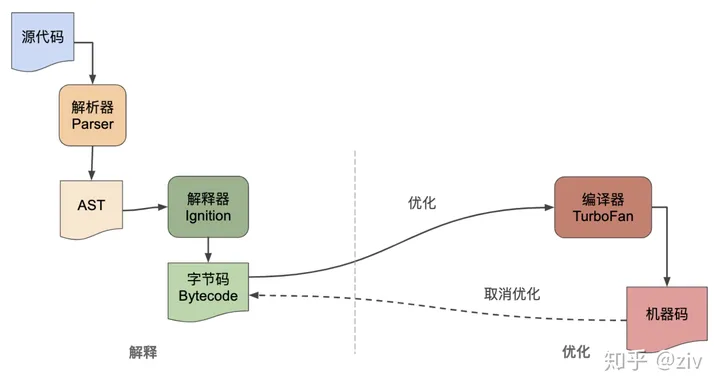
const reg = /[0-9a-z]/;

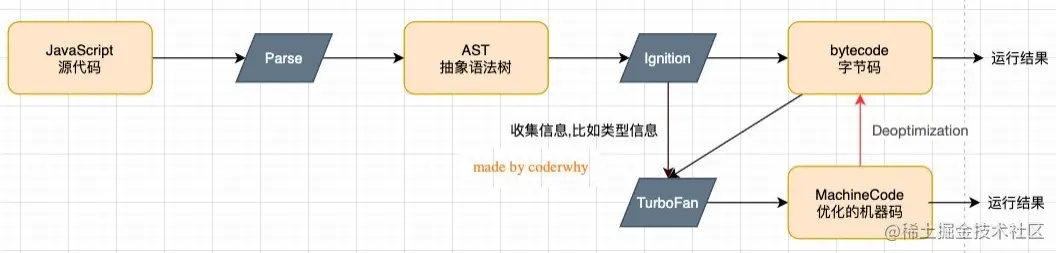
console.log('是否是正则reg：', isRegExp(reg)); // 输出 true

console.log('是否是正则reg：', isRegExp(1111)); // 输出 false

### V8执行JavaScript的过程

整体过程如下：





在这个过程中，V8同时使用了Parser（解析器）、Ignition（解释器） 和TurboFan（编译器） 来执行Js代码。

（1）Parser生成抽象语法树

（2）Ignition生成字节码

（3）执行代码及优化

### 比较值是否相等方法

JavaScript 提供三种不同的值比较运算：

（1）== 宽松相等（两个等号）

（2）=== 严格相等（三个等号）

（3）Object.is()

注意：Object.is() 和 === 之间的唯一区别在于它们处理带符号的 0 和 NaN 值的时候。

=== 运算符（和 == 运算符）将数值 -0 和 +0 视为相等，但是会将 NaN 视为彼此不相等。

分别与 JavaScript 四个相等算法中的三个相对应：

（1）IsLooselyEqual：==

（2）IsStrictlyEqual：===

（3）SameValue：Object.is()

（4）SameValueZero：被许多内置运算使用

SameValueZero实现：

function sameValueZero(x, y) {

if (typeof x === "number" && typeof y === "number") {

// x 和 y 相等（可能是 -0 和 0）或它们都是 NaN

return x === y || (x !== x && y !== y);

}

return x === y;

}

### 常见正则表达式

手机号码

const regex = /^1[3-9]\d{9}$/;

邮箱

const regex = /^([A-Za-z0-9\_-])+@([A-Za-z0-9\_-])+\.([A-Za-z]{2,4})$/

日期 YYYY-MM-DD

const regex = /^\d{4}\-\d{1,2}\-\d{1,2}$/

### 实现红绿灯

页面上存在一个div，#TrafficLight。背景颜色默认颜色是红色，2s后变成绿色，再2s后变成黄色，再1s后变成红色，再2s后变成绿色……如此一致循环。

代码：

// 需要展示动画的dom

const dom = document.getElementById("TrafficLight");

// 动画开始时间

let startTime = new Date().getTime();

// 动画总方法

function Animate() {

let now = new Date().getTime();

let animateTime = now - startTime;

let process = getProcess(animateTime);

draw(process);

window.requestAnimationFrame(Animate);

}

// 动画进程

function getProcess(animateTime) {

return (animateTime / 1000) % 5;

}

//绘制到页面上

function draw(process) {

if (process < 2) {

//1-2s显示红色

setBg("red");

} else if (process < 4) {

//2-4s显示绿色

setBg("green");

} else {

//4-5s显示黄色

setBg("yellow");

}

}

function setBg(bgColor) {

dom.style.backgroundColor = bgColor;

}

//执行

Animate();

### try catch

try catch 异常捕获

try-catch只能去捕获同步任务执行过程中的错误，也就是说，代码必须在try-catch执行线程中被捕获。

try catch 捕获不到 Promise.reject()

如：

try {

Promise.reject(1);

} catch (error) {

console.log(error);

}

// 报错 UnhandledPromiseRejection

async/ await结合try...catch使用

async function fn() {

try {

await new Promise((resolve, reject) => reject(1));

} catch (error) {

console.log(error);

}

}

fn();

// 1

但是，如果await都拿不到结果，那就更加别提能用try catch能捕获了, await表示的是落定状态(fulfilled)返回的结果。

此外，多个await 报错，只会catch到第一个，但是仍然可以继续运行。

如：

const p1 = new Promise((resolve, reject) => reject(1));

const p2 = new Promise((resolve, reject) => reject(2));

async function fn() {

try {

await p1;

await p2;

} catch (error) {

console.log(error);

}

}

fn();

// 1 UnhandledPromiseRejection

finally 块

return 语句无法阻止finally的执行

function testFinally() {

try {

return 2;

} catch (error) {

return 1;

} finally {

return 0;

}

}

console.log(testFinally()); // 0

### localStorage设置过期时间

浏览器端需要在localStorage存储用户标识，而localStorage一经存储除非手动删除是不会过期的，那就需要自己实现过期机制。

如：

Storage.prototype.setExpire = (key, value, expire) => {

let obj = {

data: value,

time: Date.now(),

expire: expire,

};

// localStorage 设置的值不能为对象,转为 JSON 字符串

localStorage.setItem(key, JSON.stringify(obj));

};

Storage.prototype.getExpire = (key) => {

let val = localStorage.getItem(key);

if (!val) {

return val;

}

val = JSON.parse(val);

if (Date.now() - val.time > val.expire) {

localStorage.removeItem(key);

return null;

}

return val.data;

};

localStorage.setExpire("userId", "zhangsan", 5000);

window.setInterval(() => {

console.log(localStorage.getExpire("userId"));

}, 1000);

结果：前5秒还是有值的，之后即为null。

该实现并非最优，还有其他实现方式。

### iframe父子页面间通信

同域下父子页面的通信

父页面调用子iframe

<iframe name="iframeName" id="iframeId" src="child.html"></iframe>

（1）通过iframe的ID获取子页面的dom，然后通过内置属性contentWindow取得子窗口的window对象

document.getElementById('iframeId').contentWindow.func();

document.getElementById('iframeId').contentWindow.document.getElementById('子页面中的元素ID');

（2）通过iframe的name（名字）直接获取子窗口的window对象

iframeName.window.func();

iframeName.window.document.getElementById('子页面中的元素ID');

子iframe页面调用父页面

通过parent或top对象获取父页面的window对象内元素及方法

parent.window.func();

parent.window.document.getElementById('父页面中的元素ID');

//同理

top.window.func();

top.window.document.getElementById('父页面中的元素ID');

跨域下父子页面的通信

使用postMessage

otherWindow.postMessage(message, targetOrigin, [transfer]);

otherWindow —— 其他窗口的一个引用。

message —— 将要发送到其他 window 的数据。

targetOrigin —— 通过窗口的 origin 属性来指定哪些窗口能接收到消息事件，其值可以是字符串"\*"（表示无限制）或者一个 URI。

transfer（可选） —— 一串和 message 同时传递的 Transferable 对象。这些对象的所有权将被转移给消息的接收方，而发送一方将不再保有所有权。

如：

<!-- a.index.html -->

<h1>父页面</h1>

<iframe id="iframe" src="http://b.index.com"></iframe>

父页面向子页面发送一条消息

const iFrame = document.getElementById("iframe");

iFrame.onload = function () {

// iFrame.contentWindow获取到iframe的window对象

iFrame.contentWindow.postMessage("父页面发送的消息", "http://b.index.com");

};

子页面接收父页面的消息

有发送就有接收，与postMessage配套使用的就是message事件。

window.addEventListener(

"message",

(e) => {

// 对消息来源origin做一下过滤，避免接收到非法域名的消息导致的xss攻击

if (e.origin === "http://a.index.com") {

console.log(e.origin); // 父页面URL，这里是http://a.index.com

console.log(e.source); // 父页面window对象，全等于window.parent/window.top

console.log(e.data); // 父页面发送的消息

}

},

false

);

注意：window.postMessage中的window指的是你想发送跨域消息的那个窗口(你需要通信的目标窗口)，而不是自身窗口的window。

## CSS

### flex:1和flex:auto的区别

flex:1

flex-grow : 1;

flex-shrink : 1;

flex-basis : 0%;

基准值取 0% 的时候，是把该项目视为零尺寸的

flex:auto

flex-grow : 1;

flex-shrink : 1;

flex-basis : auto;

基准值取 auto 的时候，根据规则基准值使用值是主尺寸值

### 用纯CSS创建一个三角形

div {

width: 0;

height: 0;

border-top: red solid 100px;

border-right: transparent solid 100px;

border-bottom: transparent solid 100px;

border-left: transparent solid 100px;

}

### div水平垂直居中

父、子元素宽高未知

(1) flex 布局

.parent {

display: flex;

justify-content: center;

align-items: center;

}

(2) absolute + margin: auto

.parent {

  position: relative;

}

.children {

  position: absolute;

  top: 0;

  left: 0;

  right: 0;

  bottom: 0;

  margin: auto;

}

(3) absolute + transform

.parent {

  position: relative;

}

.children {

  position: absolute;

  top: 50%;

  left: 50%;

  transform: translate(-50%, -50%);

}

子元素宽高已知（假设子元素宽高为 200px）

.parent {

  position: relative;

}

.children {

  position: absolute;

  top: 50%;

  left: 50%;

  margin-top: -100px;

  margin-left: -100px;

}

### CSS画一条0.5px的线

.scale-half {

height: 1px;

transform: scaleY(0.5);

transform-origin: 50% 100%; /\*为了防止线模糊\*/

}

### CSS清除浮动

style:

.clearfloat {

background-color: aqua;

}

.clearfloat::after {

content: ".";

clear: both;

display: block;

height: 0;

overflow: hidden;

visibility: hidden;

}

.floatdiv {

float: right;

height: 100px;

width: 100px;

background-color: bisque;

}

html:

<div class="clearfloat">

<div class="floatdiv"></div>

</div>

### CSS双飞燕布局

原文：<https://zhuanlan.zhihu.com/p/58355168>

left盒子设置margin:-100%的原因：<https://segmentfault.com/a/1190000014546205>

<!DOCTYPE html>

<html>

<head>

<style>

.header {

height: 40px;

background: lightblue;

}

.footer {

height: 100px;

background: lightcoral;

}

.center {

width: 100%;

height: 500px;

background: lightgray;

float: left;

}

.left {

margin-left: -100%;

width: 200px;

height: 500px;

background: lightpink;

float: left;

}

.right {

margin-left: -300px;

width: 300px;

height: 500px;

background: lightgreen;

float: left;

}

.clearfix::after {

content: "";

display: block;

clear: both;

}

.inner {

/\* 以下是设置的 margin \*/

margin-left: 200px;

margin-right: 300px;

}

</style>

</head>

<body>

<div class="header">头部</div>

<div class="main clearfix">

<div class="center">

<div class="inner">中间自适应</div>

</div>

<div class="left">左列定宽</div>

<div class="right">右列定宽</div>

</div>

<div class="footer">底部</div>

</body>

</html>

### 盒子模型

在CSS中，盒子模型可以分成：

·W3C 标准盒子模型

盒子总宽度 = width + padding + border + margin;

盒子总高度 = height + padding + border + margin

·IE 怪异盒子模型

盒子总宽度 = width + margin;

盒子总高度 = height + margin;

width/height 包含了 padding和 border值

Box-sizing

CSS 中的 box-sizing 属性定义了引擎应该如何计算一个元素的总宽度和总高度

语法：

box-sizing: content-box|border-box|inherit:

（1）content-box 默认值，与标准盒子模型表现一致

（2）border-box，与怪异盒子模型表现一致

（3）inherit 指定 box-sizing 属性的值，应该从父元素继承

### BFC块级格式化上下文

BFC（Block Formatting Context），即块级格式化上下文，它是页面中的一块渲染区域，并且有一套属于自己的渲染规则。

BFC目的是形成一个相对于外界完全独立的空间，让内部的子元素不会影响到外部的元素。

触发BFC的条件包含不限于：

·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

应用场景

·防止margin重叠（塌陷）：同一个BFC的俩个相邻的盒子的margin会发生重叠

·清除内部浮动

·自适应多栏布局

### CSS选择器

（1）类型选择器 如：p

（2）类选择器 如：.box

（3）ID 选择器 如：#box

（4）标签属性选择器 如：a[title]、a[href="https://example.com"]

（5）伪类选择器

如：:first-child、:last-child、:nth-child、:hover、:focus、:visited

（6）伪元素选择器

如：::first-line、::before、::after

（7）关系选择器（运算符选择器）

如：>、+、~、后代选择器（单空格）

### CSS Sticky Footer布局

所谓的 Sticky Footer 布局（绝对底部）并不是一种新的前端技术和概念，它就是一种网页布局。如果页面内容不够长时，底部栏就会固定到浏览器的底部；如果足够长时，底部栏就后跟随在内容的后面。

（1）

<style>

html,

body {

height: 100%;

margin: 0;

}

.wrapper {

min-height: 100%;

margin-bottom: -50px;

}

.footer,

.push {

height: 50px;

}

</style>

<div class="wrapper">

content

<div class="push"></div>

</div>

<footer class="footer">footer</footer>

（2）

<style>

body {

margin: 0;

}

.content {

min-height: calc(100vh - 70px);

}

.footer {

height: 50px;

}

</style>

<div class="content">content</div>

<footer class="footer">footer</footer>

.content给70px而不是50px是为了跟底部隔开20px，防止紧靠在一起。

（3）

<style>

html {

height: 100%;

}

body {

min-height: 100%;

display: flex;

flex-direction: column;

margin: 0;

}

.content {

flex: auto;

}

.footer {

height: 50px;

}

</style>

<div class="content">content</div>

<footer class="footer">footer</footer>

### CSS隐藏元素

（1）display:none

特点：元素不可见，不占据空间，无法响应点击事件。

（2）visibility:hidden

特点：元素不可见，占据页面空间，无法响应点击事件。

（3）opacity:0

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。

（4）设置height、width属性为0

将元素的margin，border，padding，height和width等影响元素盒模型的属性设置成0，如果元素内有子元素或内容，还应该设置其overflow:hidden来隐藏其子元素。

特点：元素不可见，不占据页面空间，无法响应点击事件。

（5）position:absolute

将元素移出可视区域。

特点：元素不可见，不影响页面布局。

（6）clip-path

特点：元素不可见，占据页面空间，无法响应点击事件。

## Vue

### Vue生命周期

主要

·beforeCreate阶段：vue实例的挂载元素el和数据对象data都是undefined，还没有初始化

·created阶段：vue实例的数据对象data有了，可以访问里面的数据和方法，未挂载到DOM，el还没有

·beforeMount阶段：vue实例的el和data都初始化了，但是挂载之前为虚拟的dom节点

·mounted阶段：vue实例挂载到真实DOM上，就可以通过DOM获取DOM节点

·beforeUpdate阶段：响应式数据更新时调用，发生在虚拟DOM打补丁之前，适合在更新之前访问现有的DOM，比如手动移除已添加的事件监听器

·updated阶段：虚拟DOM重新渲染和打补丁之后调用，组成新的DOM已经更新，避免在这个钩子函数中操作数据，防止死循环

·beforeDestroy阶段：实例销毁前调用，实例还可以用，this能获取到实例，常用于销毁定时器，解绑事件

·destroyed阶段：实例销毁后调用，调用后所有事件监听器会被移除，所有的子实例都会被销毁

其他

onErrorCaptured()​：注册一个钩子，在捕获了后代组件传递的错误时调用。

onRenderTracked() [Dev only]：注册一个调试钩子，当组件渲染过程中追踪到响应式依赖时调用。这个钩子仅在开发模式下可用。

onRenderTriggered() ​[Dev only]：注册一个调试钩子，当响应式依赖的变更触发了组件渲染时调用。这个钩子仅在开发模式下可用。

onActivated()​：注册一个回调函数，若组件实例是 <KeepAlive> 缓存树的一部分，当组件被插入到 DOM 中时调用。

onDeactivated()​：注册一个回调函数，若组件实例是 <KeepAlive> 缓存树的一部分，当组件从 DOM 中被移除时调用。

onServerPrefetch()：注册一个异步函数，在组件实例在服务器上被渲染之前调用。

### Vue2和Vue3的区别

生命周期

Vue3 在组合式API（Composition API）中使用生命周期钩子时需要先引入，而 Vue2 在选项API（Options API）中可以直接调用生命周期钩子。

多根节点

Vue2中在template里存在多个根节点会报错，而Vue3 支持多个根节点。

Composition API

Vue2 是选项API（Options API），一个逻辑会散乱在文件不同位置（data、props、computed、watch、生命周期钩子等），导致代码的可读性变差。当需要修改某个逻辑时，需要上下来回跳转文件位置。

Vue3 组合式API（Composition API）则很好地解决了这个问题，可将同一逻辑的内容写到一起，增强了代码的可读性、内聚性，其还提供了较为完美的逻辑复用性方案。

Teleport

Vue3 提供 Teleport 组件可将部分 DOM 移动到 Vue app 之外的位置，比如项目中常见的 Dialog 弹窗。

响应式原理

Vue2 响应式原理基础是 Object.defineProperty；Vue3 响应式原理基础是 Proxy。

虚拟DOM

Vue3 相比于 Vue2，虚拟DOM上增加 patchFlag 字段。

Diff算法优化

patchFlag 帮助 diff 时区分静态节点，以及不同类型的动态节点，一定程度地减少节点本身及其属性的比对。

TypeScript支持

Vue3 由 TypeScript 重写，相对于 Vue2 有更好的 TypeScript 支持。

### Vue中watch和computed的区别

computed 计算属性

这个属性依赖其他属性，由其他属性计算而来的，基于其响应式依赖被缓存。仅当初始化显示或者相关的 data、props 等属性数据发生变化的时候调用。

在computed中的，属性都有一个 get 和一个 set 方法，当数据变化时，调用 set 方法。

watch 监听属性

主要用来监听某些特定数据的变化，从而进行某些具体的业务逻辑操作，可以看作是 computed 和 methods 的结合体；

可以监听的数据来源：data，props，computed内的数据；

watch支持异步；

不支持缓存，监听的数据改变，直接会触发相应的操作。

### Vue diff算法

diff整体策略为：深度优先，同层比较。

先根据真实DOM生成一颗virtual DOM，当virtual DOM某个节点的数据改变后会生成一个新的Vnode，然后Vnode和oldVnode作对比，发现有不一样的地方就直接修改在真实的DOM上，然后使oldVnode的值为Vnode。

在采取diff算法比较新旧节点的时候，比较只会在同层级进行, 不会跨层级比较。

（1）当数据发生改变时，set方法会调用Dep.notify通知所有订阅者Watcher，订阅者就会调用patch给真实的DOM打补丁，更新相应的视图。

（2）通过isSameVnode进行判断，相同则调用patchVnode方法。

（3）patchVnode做了以下操作：

·找到对应的真实dom，称为el；

·如果都有都有文本节点且不相等，将el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el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el；

·如果两者都有子节点，则执行updateChildren函数比较子节点。

（4）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createElem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。

### Vue 双向绑定的原理

vue2采用数据劫持结合发布订阅模式，通过Object.defineProperty来劫持各属性的setter、getter方法实现数据的双向绑定，但只能检测到对象现有的属性。

而vue3是通过proxy代理实现，能监听整个对象，无论是否是新增的属性，都可以检测到。

（1）Object.defineProperty

定义：Object.defineProperty() 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象

实现响应式——通过defineProperty 两个属性，get及set

如：

function update() {

app.innerText = obj.foo

}

function defineReactive(obj, key, val) {

Object.defineProperty(obj, key, {

get() {

console.log(`get ${key}:${val}`);

return val

},

set(newVal) {

if (newVal !== val) {

val = newVal

update()

}

}

})

}

缺点：

·检测不到对象属性的添加和删除

·数组API方法无法监听到

·需要对每个属性进行遍历监听，如果嵌套对象，需要深层监听，造成性能问题

（2）Proxy

Proxy的监听是针对一个对象的，对这个对象的所有操作会进入监听操作，这就完全可以代理所有属性了。Proxy可以劫持整个对象，并返回一个新对象，我们可以只操作新的对象达到响应式目的。

Proxy 不兼容IE，也没有 polyfill，defineProperty 能支持到IE9

### SPA首屏优化方式

（1）减小入口文件积

常用的手段是路由懒加载，在vue-router配置路由的时候，采用动态加载路由的形式。

（2）静态资源本地缓存

后端返回资源问题：

·采用HTTP缓存，设置Cache-Control，Last-Modified，Etag等响应头

·采用Service Worker离线缓存

前端合理利用localStorage

（3）UI框架按需加载

按需引用UI框架中所需组件

（4）图片资源的压缩

对于所有的图片资源，我们可以进行适当的压缩

对页面上使用到的icon，可以使用在线字体图标，或者雪碧图，将众多小图标合并到同一张图上，用以减轻http请求压力

（5）组件重复打包

假设A.js文件是一个常用的库，现在有多个路由使用了A.js文件，这就造成了重复下载

解决方案：在webpack的config文件中，修改CommonsChunkPlugin的配置

minChunks: 3

minChunks为3表示会把使用3次及以上的包抽离出来，放进公共依赖文件，避免了重复加载组件

（6）开启GZip压缩

拆完包之后，我们再用gzip做一下压缩

在服务器做一定配置，如果发送请求的浏览器支持gzip，就发送给它gzip格式的文件

（7）使用SSR

组件或页面通过服务器生成html字符串，再发送到浏览器

### Vue组件间通信方式

（1）父子关系的组件数据传递

·props 与 $emit

·ref （配合defineExpose使用）

（2）兄弟关系的组件数据传递

·$bus（EventBus，发布订阅模式）

·$parent

·以父组件为中间桥梁，借助 emit 和 props

（3）祖先与后代组件数据传递

·$attrs 与 $listeners

·provide 与 inject

（4）复杂关系的组件数据传递

·Vuex 或 Pinia存放共享的变量

### hash模式和history模式的区别

（1）hash模式：监听浏览器地址hash值变化，执行相应的js切换网页。

hash 虽然出现在 URL 中，但不会被包括在 HTTP 请求中，对后端完全没有影响，因此改变 hash 不会重新加载页面。

路由的哈希模式其实是利用了window.onhashchange事件，url中的哈希值（#后面的值）如果有变化，就会自动调用hashchange的监听事件，在hashchange的监听事件内可以得到改变后的url，这样能够找到对应页面进行加载。

（2）history模式：利用history API实现url地址改变，网页内容改变。

利用了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法（需要特定浏览器支持）。除此之外，这种模式还需要后台配置支持。

### Vue中key的作用

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。

详细作用

在没有 key 的情况下（此时，key的值为undefined），Vue 将使用一种最小化元素移动的算法，并尽可能地就地更新/复用相同类型的元素。如果传了 key，则将根据 key 的变化顺序来重新排列元素，并且将始终移除/销毁 key 已经不存在的元素。

同一个父元素下的子元素必须具有唯一的 key。重复的 key 将会导致渲染异常。

例子

（1）最常见的用例是与 v-for 结合

如：

<ul>

<li v-for="item in items" :key="item.id">...</li>

</ul>

（2）也可以用于强制替换一个元素/组件而不是复用它

·在适当的时候触发组件的生命周期钩子

·触发过渡

如：

<transition>

<span :key="text">{{ text }}</span>

</transition>

### KeepAlive组件

定义

KeepAlive是Vue中内置的一个组件。

KeepAlive可以设置以下props属性：

（1）include：字符串或正则表达式，只有名称匹配的组件会被缓存。

（2）exclude：字符串或正则表达式，任何名称匹配的组件都不会被缓存。

（3）max：数字，最多可以缓存多少组件实例。

设置了KeepAlive缓存的组件，会多出两个生命周期钩子（activated与deactivated）：

·首次进入组件时：beforeRouteEnter > beforeCreate > created> mounted > activated > ... ... > beforeRouteLeave > deactivated

·再次进入组件时：beforeRouteEnter >activated > ... ... > beforeRouteLeave > deactivated

原理

this.cache是一个对象，用来存储需要缓存的组件。

如果this.keys中缓存组件的数量超过了设置的最大缓存数量值this.max，则把第一个缓存组件删掉。

调用pruneCacheEntry函数将不需要缓存的组件从this.cache对象剔除。

### Vue中父子组件的生命周期顺序

父：created

父：beforeMount

子：created

子：beforeMount

子：mounted

父：mounted

父：beforeUpdate

子：beforeUpdate

子：updated

父：updated

### Vue中常用的修饰符

（1）.stop：阻止事件冒泡

（2）.native：绑定原生事件

（3）.once：事件只执行一次

（4）.self ：将事件绑定在自身身上，相当于阻止事件冒泡

（5）.prevent：阻止默认事件

（6）.capture：用于事件捕获

### Vue中自定义指令

注册一个自定义指令有全局注册与局部注册。

全局注册

app.directive('demo', (el, binding) => {

console.log(binding.value.color) // => "white"

console.log(binding.value.text) // => "hello!"

})

<div v-demo="{ color: 'white', text: 'hello!' }"></div>

局部注册

组合式

// 使用 <script setup>

const vFocus = {

mounted: (el) => el.focus()

}

选项式

directives: {

focus: {

mounted: (el) => el.focus();

}

}

钩子参数

el：指令绑定到的元素。这可以用于直接操作 DOM。

binding：一个对象，包含多个属性。

vnode：代表绑定元素的底层 VNode。

prevNode：代表之前的渲染中指令所绑定元素的 VNode。仅在 beforeUpdate 和 updated 钩子中可用。

如：

<div v-example:foo.bar="baz">

binding 参数会是一个这样的对象：

{

arg: 'foo',

modifiers: { bar: true },

value: /\* `baz` 的值 \*/,

oldValue: /\* 上一次更新时 `baz` 的值 \*/

}

应用场景

·表单防止重复提交

·图片懒加载

·一键 Copy的功能

·页面水印

·权限校验

### 虚拟DOM

定义

虚拟 DOM 为Vue、React等框架带来了跨平台的能力（React-Native 和 Weex）。

实际上它只是一层对真实DOM的抽象，以JavaScript 对象 (VNode 节点) 作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上。

在Javascript对象中，虚拟DOM 表现为一个 Object对象。并且最少包含标签名 (tag)、属性 (attrs) 和子元素对象 (children) 三个属性，不同框架对这三个属性的命名可能有所差别。

Vue中虚拟DOM技术。

通过VNode，Vue可以对抽象树进行创建节点，删除节点以及修改节点的操作，经过diff算法得出一些需要修改的最小单位，再更新视图，减少了dom操作，提高了性能。

### 权限管理

前端权限控制可以分为四个方面：

（1）接口权限

（2）按钮权限

（3）菜单权限

（4）路由权限

接口权限

接口权限目前一般采用jwt的形式来验证，没有通过的话一般返回401，跳转到登录页面重新进行登录。

登录完拿到token，将token存起来，通过axios请求拦截器进行拦截，每次请求的时候头部携带token。

路由权限控制

方案一

初始化即挂载全部路由，并且在路由上标记相应的权限信息，每次路由跳转前做校验。

方案二

初始化的时候先挂载不需要权限控制的路由，比如登录页、404等错误页。如果用户通过URL进行强制访问，则会直接进入404，相当于从源头上做了控制。

登录后，获取用户的权限信息，然后筛选有权限访问的路由，在全局路由守卫里进行调用addRoutes添加路由。

菜单权限

菜单权限可以理解成将页面与路由进行解耦。

方案一

菜单与路由分离，菜单由后端返回。

方案二

菜单和路由都由后端返回。

按钮权限

方案一

按钮权限可以用v-if判断。

方案二

通过自定义指令进行按钮权限的判断。

### Vue3 性能提升

编译阶段

Vue3在编译阶段，做了进一步优化。主要有如下：

·diff算法优化

·静态提升

·事件监听缓存

·SSR优化

源码体积

相比Vue2，Vue3整体体积变小了，除了移出一些不常用的API，再重要的是Tree shanking。

任何一个函数，如ref、reavtived、computed等，仅仅在用到的时候才打包，没用到的模块都被摇掉，打包的整体体积变小。

响应式系统

vue2中采用 defineProperty来劫持整个对象，然后进行深度遍历所有属性，给每个属性添加getter和setter，实现响应式。

vue3采用proxy重写了响应式系统，因为proxy可以对整个对象进行监听，所以不需要深度遍历。

### Vue 3 Tree shaking特性

定义

Tree shaking是一种通过清除多余代码方式来优化项目打包体积的技术，专业术语叫 Dead code elimination。简单来讲，就是在保持代码运行结果不变的前提下，去除无用的代码。

Tree shaking 其实是找出使用的代码。

Vue3源码引入tree shaking特性，将全局 API 进行分块。如果您不使用其某些功能，它们将不会包含在您的基础包中。

实现

Tree shaking是基于ES6模板语法（import与exports），主要是借助ES6模块的静态编译思想，在编译时就能确定模块的依赖关系，以及输入和输出的变量。

Tree shaking无非就是做了两件事：

（1）编译阶段利用ES6 Module判断哪些模块已经加载；

（2）判断那些模块和变量未被使用或者引用，进而删除对应代码。

作用

通过Tree shaking，Vue3给我们带来的好处是：

·减少程序体积（更小）

·减少程序执行时间（更快）

·便于将来对程序架构进行优化（更友好）

### Vue3中ref与reactive

Vue3中响应式是通过ES6的Proxy实现的，其中reactive和ref可以实现响应式数据。

reactive

特点

（1）reactive的参数一般是对象或者数组，他能够将复杂数据类型变为响应式数据。

（2）reactive的响应式是深层次的，底层本质是将传入的数据转换为Proxy对象。

ref

特点

（1）ref的参数一般是基本数据类型，也可以是对象类型。

（2）如果参数是对象类型，其实底层的本质还是reactive，系统会自动将ref转换为reactive。例如：ref(1) ===> reactive({value:1})。

（3）当在模板中使用时，ref 会自动解包；在JS中访问ref中的数据，则需要手动添加 .value。

（4）ref的底层原理同reactive一样，都是Proxy。

## React

### React特性

（1）JSX 语法

（2）单向数据绑定

（3）虚拟 DOM

（4）声明式编程

（5）Component

### React和Vue区别

相同点

（1）都有组件化思想

（2）都支持服务器端渲染

（3）都有Virtual DOM（虚拟dom，提供跨平台的能力）

（4）数据驱动视图

（5）都有支持native的方案：Vue的weex、React的React native

（6）都有自己的构建工具：Vue的vue-cli、React的Create React App

区别

（1）数据变化的实现原理不同

React使用的是不可变数据，而Vue使用的是可变的数据。

（2）组件写法差异

React推荐的做法是JSX，把 HTML 和 CSS 全都写进 JavaScript 中；Vue 推荐的做法是 template 的单文件组件格式，即 html、css、JS 写在同一个文件(vue也支持JSX写法)。

（3）diff算法不同

react主要使用diff队列保存需要更新哪些DOM，得到patch树，再统一操作批量更新DOM。Vue 使用双向指针，边对比，边更新DOM。

### React Hooks如何存储组件中的状态

在React中通过currentRenderingFiber来标识当前渲染节点，每个组件都有一个对应的fiber节点，用来保存组件的相关数据信息。

每次函数组件渲染时，currentRenderingFiber就被赋值为当前组件对应的fiber，所以实际上hook是通过currentRenderingFiber来获取状态信息的。

currentRenderingFiber.memorizedState中保存一条hook对应数据的单向链表。

当函数组件渲染时，每执行到一个hook，就会将currentRenderingFiber.memorizedState的指针向后移一下。这也是hook的调用顺序不能改变的原因（不能在条件语句中使用hook）

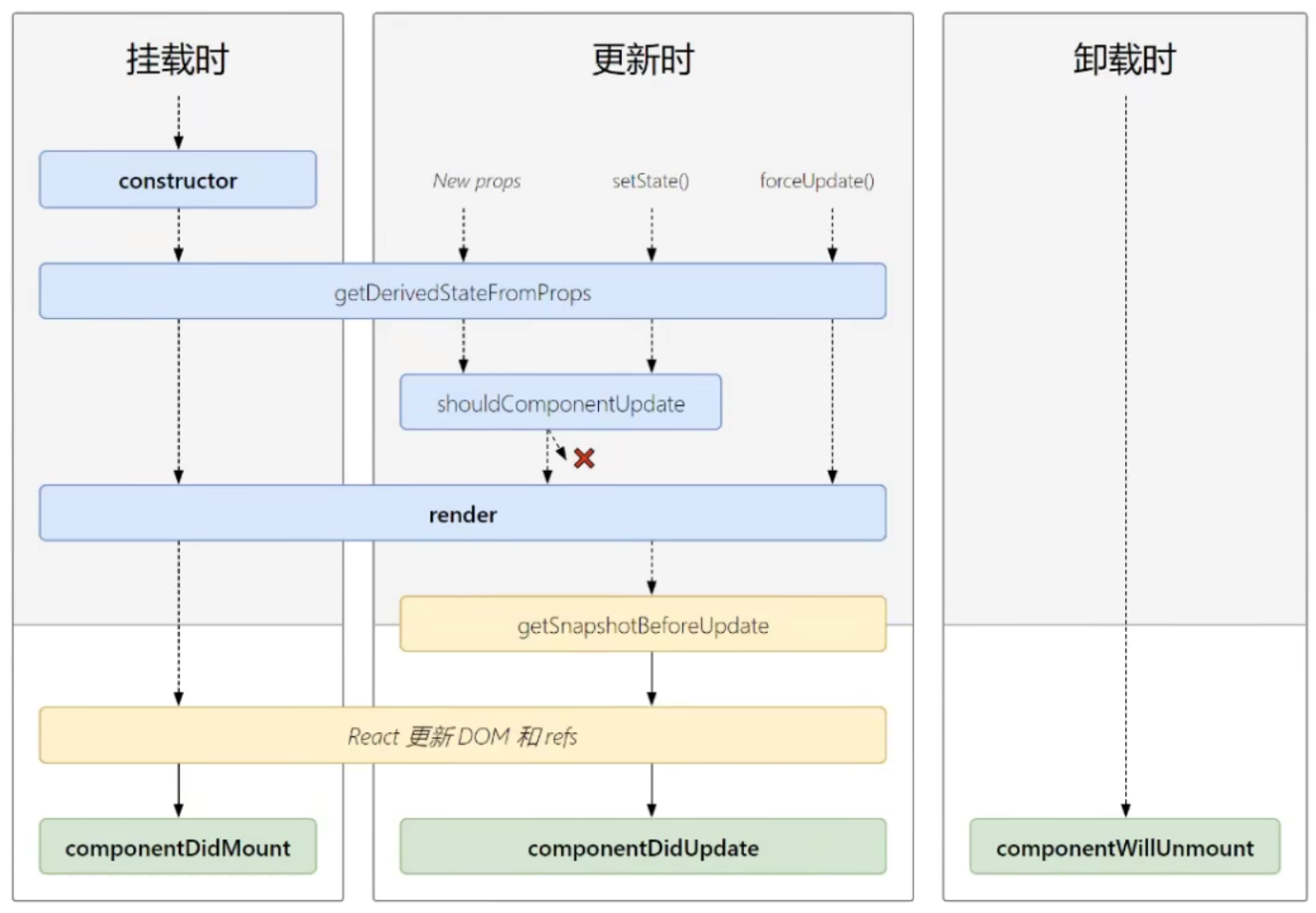
### React中的函数式思想

（1）React中每个组件都是纯函数，编写的 React 组件必须始终在给定相同输入的情况下返回相同的 JSX；

（2）组件不应更改渲染前存在的任何对象或变量，执行过程没有副作用；

（3）没有额外的状态依赖。

### React生命周期



挂载阶段

·constructor

·getDerivedStateFromProps

·render

·componentDidMount

更新阶段

·getDerivedStateFromProps

·shouldComponentUpdate

·render

·getSnapshotBeforeUpdate

·componentDidUpdate

卸载阶段

·componentWillUnmount

## Webpack

### Webpack的构建过程

（1）初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；

（2）开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；

（3）确定入口：根据配置中的 entry 找出所有的入口文件；

（4）编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；

（5）完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；

（6）输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；

（7）输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

### Webpack的优点

相比其他的模块化工具，webpack大而全，很多常用的功能做到开箱即用。

有两大最核心的特点：一切皆模块和按需加载。

与其他构建工具相比，有如下优势：

智能解析：对 CommonJS 、 AMD 、ES6 的语法做了兼容

万物模块：对 js、css、图片等资源文件都支持打包

开箱即用：HMR、Tree-shaking等功能

代码分割：可以将代码切割成不同的 chunk，实现按需加载，降低了初始化时间

插件系统：具有强大的 Plugin 接口，具有更好的灵活性和扩展性

易于调试：支持 SourceUrls 和 SourceMaps

快速运行：webpack 使用异步 IO 并具有多级缓存，这使得 webpack 很快且在增量编译上更加快

生态环境好：社区更丰富，出现的问题更容易解决

### Webpack和Vite的区别

（1）构建原理

Webpack 是一个静态模块打包器，通过对项目中的 JavaScript、CSS、图片等文件进行分析，生成对应的静态资源，并且可以通过一些插件和加载器来实现各种功能；Vite 则是一种基于浏览器原生 ES 模块解析的构建工具。Webpack更多地关注兼容性、支持更广, 而 Vite关注浏览器端的开发体验

（2）插件和加载器

Webpack 有大量的插件和加载器可以使用，可以实现各种复杂的构建场景，例如代码分割、按需加载、CSS 预处理器等；Vite 的插件和加载器相对较少。此外，Webpack 的配置比较复杂，Vite 的配置相对简单。

（3）HMR

在HMR（热更新）方面，当改动了一个模块后，Vite仅需让浏览器重新请求该模块即可，而Webpack需要把该模块的相关依赖模块全部编译一次。

（4）开发效率不同

Webpack是先打包再启动开发服务器，Vite是直接启动开发服务器，然后按需编译依赖文件。由于Vite在启动的时候不需要打包，也就意味着不需要分析模块的依赖、不需要编译，当浏览器请求某个模块时，再根据需要对模块内容进行编译，因此启动速度非常快。而当需要打包到生产环境时，Vite使用传统的rollup进行打包，因此，Vite的主要优势在开发阶段。

### loader和plugin的区别

loader

loader 用于对模块的源代码进行转换。让 webpack 能够去处理其他类型的文件，并将它们转换为有效模块，以供应用程序使用。

配置loader的方式

（1）配置方式（推荐）：在 webpack.config.js文件中指定 loader => [ module.rules ]

（2）内联方式：在每个 import 语句中显式指定 loader

（3）CLI 方式：在 shell 命令中指定它们

loader支持链式调用，顺序为相反的顺序执行。

常见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

babel-loader：用babel来转换ES6文件到ES

ts-loader：将 TypeScript 编译成 JavaScript

plugin

webpack 插件是一个具有 apply 方法的 JavaScript 对象。apply 方法会被webpack compiler 调用，并且在整个编译生命周期都可以访问 compiler 对象。

插件可以用于执行范围更广的任务。包括：打包优化，资源管理，注入环境变量等。

常见plugin

ProgressPlugin：报告编译进度

TerserPlugin：在项目中使用 Terser 插件来压缩 JS

HtmlWebpackPlugin：快速创建 HTML 文件来服务 bundles

区别

loader 运行在打包文件之前，plugins 在整个编译周期都起作用。

在Webpack 运行的生命周期中会广播出许多事件，plugin 可以监听这些事件，在合适的时机通过Webpack提供的 API改变输出结果。而loader，实质是一个转换器，将A文件进行编译形成B文件，操作的是文件。

### 动态引入原理

（1）模块加载

webpack通过\_\_webpack\_require\_\_加载模块代码

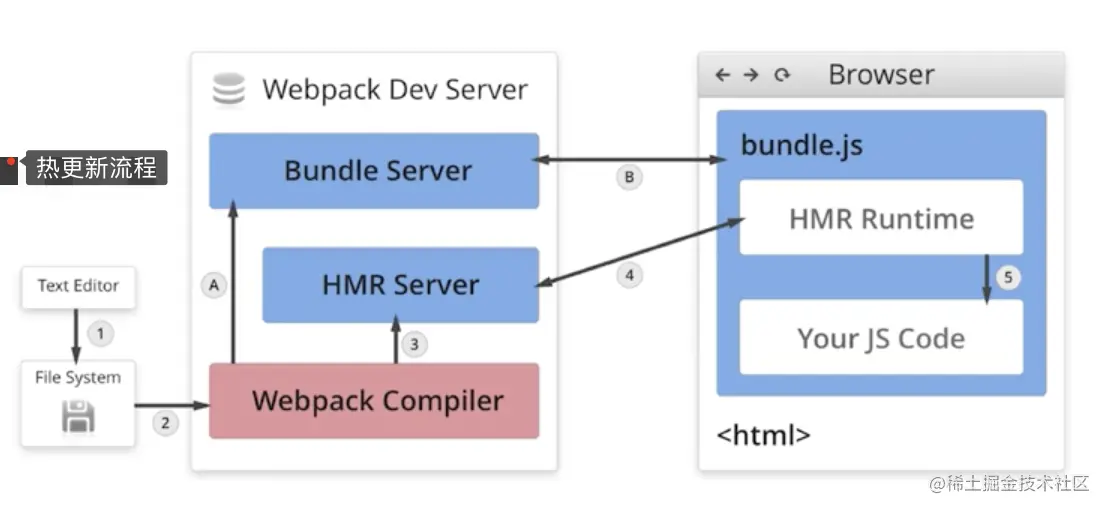
（2）动态加载script

（3）执行异步脚本

（4）执行异步模块代码

### HMR实现

HMR全称 Hot Module Replacement，可以理解为模块热替换，指在应用程序运行过程中，替换、添加、删除模块，而无需重新刷新整个应用。



启动阶段

编写未经过webpack打包的源代码，文件经过 Webpack-complier 编译好后传输给 Bundle Server（静态资源文件服务器，提供文件访问路径），Bundle Server可以让浏览器访问到我们打包出来的文件。

文件热更新阶段

文件经过 Webpack-complier 编译好后传输给 HMR Server，HMR Server 知道哪个资源(模块)发生了改变，并通知 HMR Runtime 有哪些变化，HMR Runtime 就会更新我们的代码，这样我们浏览器就会更新并且不需要刷新。

变化情况“update”由两部分组成：

（1）更新后的 manifest (JSON)

manifest 包括新的 compilation hash 和所有的 updated chunk 列表。

（2）一个或多个 updated chunk (JavaScript)

每个 chunk 都包含着全部更新模块的最新代码（或一个 flag 表明此模块需要被移除）。

### webpack proxy工作原理

webpack proxy，即webpack提供的代理服务。基本行为就是接收客户端发送的请求后转发给其他服务器，其目的是为了便于开发者在开发模式下解决跨域问题（浏览器安全策略限制）。

想要实现代理首先需要一个中间服务器，webpack中提供服务器的工具为webpack-dev-server。

webpack-dev-server

webpack-dev-server是 webpack 官方推出的一款开发工具，将自动编译和自动刷新浏览器等一系列对开发友好的功能全部集成在了一起。目的是为了提高开发者日常的开发效率，只适用在开发阶段。

关于配置方面，在webpack配置对象属性中通过devServer属性提供。

工作原理

proxy工作原理实质上是利用http-proxy-middleware 这个http代理中间件，实现请求转发给其他服务器。

当本地发送请求的时候，代理服务器响应该请求，并将请求转发到目标服务器。目标服务器响应数据后再将数据返回给代理服务器，最终再由代理服务器将数据响应给本地。在代理服务器传递数据给本地浏览器的过程中，两者同源，并不存在跨域行为，这时候浏览器就能正常接收数据。

## 设计模式

### 设计模式七大原则

（1）单一职责原则 (Single Responsibility Principle)

（2）开放-关闭原则 (Open-Closed Principle)

（3）里氏替换原则 (Liskov Substitution Principle)

（4）依赖倒转原则 (Dependence Inversion Principle)

（5）接口隔离原则 (Interface Segregation Principle)

（6）迪米特法则（Law Of Demeter）

（7）组合/聚合复用原则 (Composite/Aggregate Reuse Principle)

### 单例模式

单例模式（Singleton Pattern）：创建型模式，提供了一种创建对象的最佳方式，这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。

在前端中，很多情况都是用到单例模式，例如页面存在一个模态框的时候，只有用户点击的时候才会创建，而不是加载完成之后再创建弹窗和隐藏，并且保证弹窗全局只有一个。

如：

// 单例构造函数

function CreateSingleton (name) {

this.name = name;

this.getName();

};

// 获取实例的名字

CreateSingleton.prototype.getName = function() {

console.log(this.name)

};

// 单例对象

const Singleton = (function(){

var instance;

return function (name) {

if(!instance) {

instance = new CreateSingleton(name);

}

return instance;

}

})();

// 创建实例对象1

const a = new Singleton('a');

// 创建实例对象2

const b = new Singleton('b');

console.log(a===b); // true

### 工厂模式

工厂模式是用来创建对象的一种最常用的设计模式，不暴露创建对象的具体逻辑，而是将将逻辑封装在一个函数中，那么这个函数就可以被视为一个工厂

工厂模式根据抽象程度的不同可以分为：

·简单工厂模式（Simple Factory）

简单工厂模式也叫静态工厂模式，用一个工厂对象创建同一类对象类的实例。

·工厂方法模式（Factory Method）

工厂方法模式跟简单工厂模式差不多，但是把具体的产品放到了工厂函数的prototype中。

·抽象工厂模式（Abstract Factory）

简单工厂模式和工厂方法模式都是直接生成实例，但是抽象工厂模式不同，抽象工厂模式并不直接生成实例， 而是用于对产品类簇的创建。

### 策略模式

策略模式（Strategy Pattern）指的是定义一系列的算法，把它们一个个封装起来，目的就是将算法的使用与算法的实现分离开来。

一个基于策略模式的程序至少由两部分组成：

（1）策略类，策略类封装了具体的算法，并负责具体的计算过程；

（2）环境类Context，Context 接受客户的请求，随后把请求委托给某一个策略类。

### 代理模式

代理模式（Proxy Pattern）是为一个对象提供一个代用品或占位符，以便控制对它的访问。

代理模式的关键是，当客户不方便直接访问一个对象或者不满足需要时，提供一个替身对象来控制这个对象的访问，客户实际上访问的是替身对象。

按照功能来划分，javascript代理模式常用的有：

（1）缓存代理

缓存代理可以为一些开销大的运算结果提供暂时的存储，在下次运算时，如果传递进来的参数跟之前一致，则可以直接返回前面存储的运算结果。

（2）虚拟代理

虚拟代理把一些开销很大的对象，延迟到真正需要它的时候才去创建。

### 观察者模式

观察者模式定义了对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知，并自动更新。

如：

class Subject {

constructor() {

this.observerList = [];

}

addObserver(observer) {

this.observerList.push(observer);

}

removeObserver(observer) {

const index = this.observerList.findIndex(o => o.name === observer.name);

this.observerList.splice(index, 1);

}

notifyObservers(message) {

const observers = this.observeList;

observers.forEach(observer => observer.notified(message));

}

}

### 发布订阅模式

在软件架构中，发布/订阅是一种消息范式，消息的发送者（称为发布者）不会将消息直接发送给特定的接收者（称为订阅者），而是通过消息通道广播出去，让订阅该消息主题的订阅者消费到。发布的消息分为不同的类别，发布者无需了解哪些订阅者（如果有的话）可能存在。

如：

// 公众号对象

let eventEmitter = {};

// 缓存列表，存放 event 及 fn

eventEmitter.list = {};

// 订阅

eventEmitter.on = function (event, fn) {

let \_this = this;

// 如果对象中没有对应的 event 值，也就是说明没有订阅过，就给 event 创建个缓存列表

// 如有对象中有相应的 event 值，把 fn 添加到对应 event 的缓存列表里

(\_this.list[event] || (\_this.list[event] = [])).push(fn);

return \_this;

};

// 发布

eventEmitter.emit = function () {

let \_this = this;

// 第一个参数是对应的 event 值，直接用数组的 shift 方法取出

let event = [].shift.call(arguments);

let fns = [...\_this.list[event]];

// 如果缓存列表里没有 fn 就返回 false

if (!fns || fns.length === 0) {

return false;

}

// 遍历 event 值对应的缓存列表，依次执行 fn

fns.forEach((fn) => {

fn.apply(\_this, arguments);

});

return \_this;

};