1. HTTP 和 HTTPS

1.http 和 https 的基本概念

http: 是一个客户端和服务器端请求和应答的标准（TCP），用于从 WWW 服务器传输超文本到本地浏览器的超文本传输协议。
https:是以安全为目标的 HTTP 通道，即 HTTP 下 加入 SSL 层进行加密。其作用是：建立一个信息安全通道，来确保数据的传输，确保网站的真实性。

2.http 和 https 的区别及优缺点？

http 是超文本传输协议，信息是明文传输，HTTPS 协议要比 http 协议安全，https 是具有安全性的 ssl 加密传输协议，可防止数据在传输过程中被窃取、改变，确保数据的完整性(当然这种安全性并非绝对的，对于更深入的 Web 安全问题，此处暂且不表)。
http 协议的默认端口为 80，https 的默认端口为 443。
http 的连接很简单，是无状态的。https 握手阶段比较费时，会使页面加载时间延长 50%，增加 10%~20%的耗电。
https 缓存不如 http 高效，会增加数据开销。
Https 协议需要 ca 证书，费用较高，功能越强大的证书费用越高。
SSL 证书需要绑定 IP，不能再同一个 IP 上绑定多个域名，IPV4 资源支持不了这种消耗。

3.https 协议的工作原理

客户端在使用 HTTPS 方式与 Web 服务器通信时有以下几个步骤：

1. 客户端使用 https url 访问服务器，则要求 web 服务器建立 ssl 链接。
2. web 服务器接收到客户端的请求之后，会将网站的证书（证书中包含了公钥），传输给客户端。
3. 客户端和 web 服务器端开始协商 SSL 链接的安全等级，也就是加密等级。
4. 客户端浏览器通过双方协商一致的安全等级，建立会话密钥，然后通过网站的公钥来加密会话密钥，并传送给网站。
5. web 服务器通过自己的私钥解密出会话密钥。
6. web 服务器通过会话密钥加密与客户端之间的通信。

TCP 三次握手

1. 第一次握手：建立连接时，客户端发送 syn 包（syn=j）到服务器，并进入 SYN_SENT 状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。
2. 第二次握手：服务器收到 syn 包并确认客户的 SYN（ack=j+1），同时也发送一个自己的 SYN 包（syn=k），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

3. 第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK(ack=k+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED（TCP 连接成功）状态，完成三次握手。

   握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。

TCP 四次挥手

1. 客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。 TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。

2）服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=v，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时间。

3）客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最 后的数据）。

4）服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 seq=w，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。

5）客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是 seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放，必须经过 2* * MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态。

6）服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些。

TCP/IP / 如何保证数据包传输的有序可靠？

对字节流分段并进行编号然后通过 ACK 回复和超时重发这两个机制来保证。

（1）为了保证数据包的可靠传递，发送方必须把已发送的数据包保留在缓冲区；
（2）并为每个已发送的数据包启动一个超时定时器；

（3）如在定时器超时之前收到了对方发来的应答信息（可能是对本包的应答，也可以是对本包后续包的应答），则释放该数据包占用的缓冲区;
（4）否则，重传该数据包，直到收到应答或重传次数超过规定的最大次数为止。
（5）接收方收到数据包后，先进行 CRC 校验，如果正确则把数据交给上层协议，然后给发送方发送一个累计应答包，表明该数据已收到，如果接收方正好也有数据要发给发送方，应答包也可方在数据包中捎带过去。

TCP 和 UDP 的区别

1. TCP 是面向链接的，而 UDP 是面向无连接的。
2. TCP 仅支持单播传输，UDP 提供了单播，多播，广播的功能。
3. TCP 的三次握手保证了连接的可靠性; UDP 是无连接的、不可靠的一种数据传输协议，首先不可靠性体现在无连接上，通信都不需要建立连接，对接收到的数据也不发送确认信号，发送端不知道数据是否会正确接收。
4. UDP 的头部开销比 TCP 的更小，数据传输速率更高，实时性更好。

HTTP 请求跨域问题

1.

跨域的原理

2.

跨域，是指浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的。
同源策略,是浏览器对 JavaScript 实施的安全限制，只要协议、域名、端口有任何一个不同，都被当作是不同的域。
跨域原理，即是通过各种方式，避开浏览器的安全限制。

3.
4.

解决方案

5.

最初做项目的时候，使用的是 jsonp，但存在一些问题，使用 get 请求不安全，携带数据较小，后来也用过 iframe，但只有主域相同才行，也是存在些问题，后来通过了解和学习发现使用代理和 proxy 代理配合起来使用比较方便，就引导后台按这种方式做下服务器配置，在开发中使用 proxy，在服务器上使用 nginx 代理，这样开发过程中彼此都方便，效率也高；现在 h5 新特性还有 windows.postMessage()

6.

-

JSONP：
ajax 请求受同源策略影响，不允许进行跨域请求，而 script 标签 src 属性中的链 接却可以访问跨域的 js 脚本，利用这个特性，服务端不再返回 JSON 格式的数据，而是 返回一段调用某个函数的 js 代码，在 src 中进行了调用，这样实现了跨域。

-

步骤：

-

1. 去创建一个 script 标签
2. script 的 src 属性设置接口地址
3. 接口参数，必须要带一个自定义函数名，要不然后台无法返回数据
4. 通过定义函数名去接受返回的数据

```
//动态创建 scriptvar script =
document.createElement('script');
// 设置回调函数 function getData(data) {
    console.log(data);
}
//设置 script 的 src 属性，并设置请求地址
script.src = 'http://localhost:3000/?callback=getData';
// 让 script 生效 document.body.appendChild(script);复制
代码
```

JSONP 的缺点:
JSON 只支持 get,因为 script 标签只能使用 get 请求；JSONP 需要后端配合返回指定格式的数据。

-

document.domain 基础域名相同 子域名不同

-
-

window.name 利用在一个浏览器窗口内，载入所有的域名都是共享一个 window.name

-

-

CORS CORS(Cross-origin resource sharing)跨域资源共享 服务器设置对 CORS 的支持原理：服务器设置 Access-Control-Allow-Origin HTTP 响应头之后，浏览器将会允许跨域请求

-
-

proxy 代理 目前常用方式,通过服务器设置代理

-
-

window.postMessage() 利用 h5 新特性 window.postMessage()

-

跨域传送门 ☞ # 跨域，不可不知的基础概念

Cookie、sessionStorage、localStorage 的区别

相同点：

存储在客户端

不同点：

cookie 数据大小不能超过 4k；sessionStorage 和 localStorage 的存储比 cookie 大得多，可以达到 5M+
cookie 设置的过期时间之前一直有效；localStorage 永久存储，浏览器关闭后数据不丢失除非主动删除数据；sessionStorage 数据在当前浏览器窗口关闭后自动删除
cookie 的数据会自动的传递到服务器；sessionStorage 和 localStorage 数据保存在本地

粘包问题分析与对策

TCP 粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

粘包出现原因

简单得说，在流传输中出现，UDP 不会出现粘包，因为它有**消息边界**

粘包情况有两种，一种是粘在一起的包都是完整的数据包，另一种情况是粘在一起的包有不完整的包。

为了**避免粘包**现象，可采取以下几种措施：

（1）对于发送方引起的粘包现象，用户可通过编程设置来避免，TCP 提供了强制数据立即传送的操作指令 push，TCP 软件收到该操作指令后，就立即将本段数据发送出去，而不必等待发送缓冲区满；

（2）对于接收方引起的粘包，则可通过优化程序设计、精简接收进程工作量、提高接收进程优先级等措施，使其及时接收数据，从而尽量避免出现粘包现象；

（3）由接收方控制，将一包数据按结构字段，人为控制分多次接收，然后合并，通过这种手段来避免粘包。分包多发。

以上提到的三种措施，都有其不足之处。

（1）第一种编程设置方法虽然可以避免发送方引起的粘包，但它关闭了优化算法，降低了网络发送效率，影响应用程序的性能，一般不建议使用。

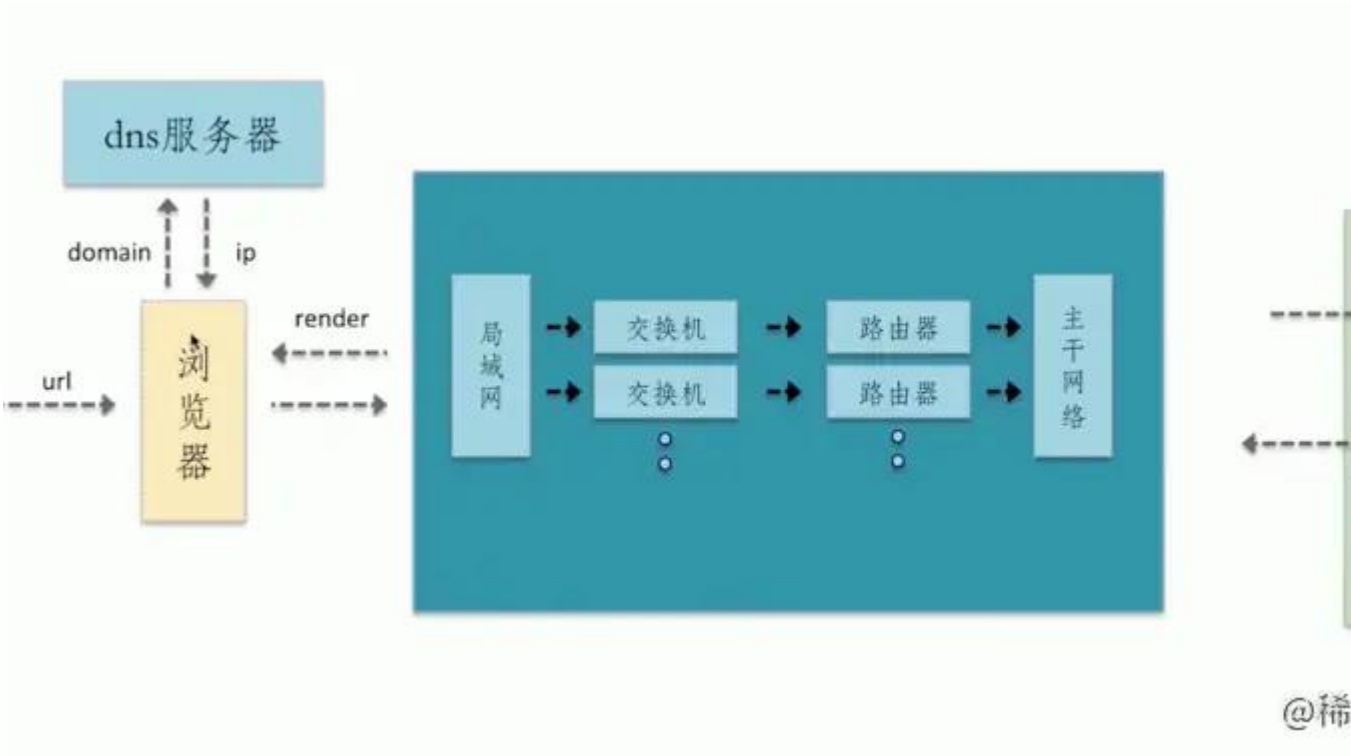
（2）第二种方法只能减少出现粘包的可能性，但并不能完全避免粘包，当发送频率较高时，或由于网络突发可能使某个时间段数据包到达接收方较快，接收方还是有可能来不及接收，从而导致粘包。

（3）第三种方法虽然避免了粘包，但应用程序的效率较低，对实时应用的场合不适合。

> 一种比较周全的对策是：接收方创建一预处理线程，对接收到的数据包进行预处理，将粘连的包分开。实验证明这种方法是高效可行的。

浏览器

从输入 URL 到页面加载的全过程



1.

首先在浏览器中输入 URL

2.
3.

查找缓存：浏览器先查看浏览器缓存-系统缓存-路由缓存中是否有该地址页面，如果有则显示页面内容。如果没有则进行下一步。

4.

- 浏览器缓存：浏览器会记录 DNS 一段时间，因此，只是第一个地方解析 DNS 请求；
- 操作系统缓存:如果在浏览器缓存中不包含这个记录，则会使系统调用操作系统， 获取操作系统的记录(保存最近的 DNS 查询缓存)；
- 路由器缓存：如果上述两个步骤均不能成功获取 DNS 记录，继续搜索路由器缓存；
- ISP 缓存：若上述均失败，继续向 ISP 搜索。

5.

DNS 域名解析：浏览器向 DNS 服务器发起请求，解析该 URL 中的域名对应的 IP 地址。DNS 服务器是基于 UDP 的，因此会用到 UDP 协议。

6.
7.

建立 TCP 连接：解析出 IP 地址后，根据 IP 地址和默认 80 端口，和服务器建立 TCP 连接

8.
9.

发起 HTTP 请求：浏览器发起读取文件的 HTTP 请求，，该请求报文作为 TCP 三次握手的第三次数据发送给服务器

10.
11.

服务器响应请求并返回结果：服务器对浏览器请求做出响应，并把对应的 html 文件发送给浏览器

12.
13.

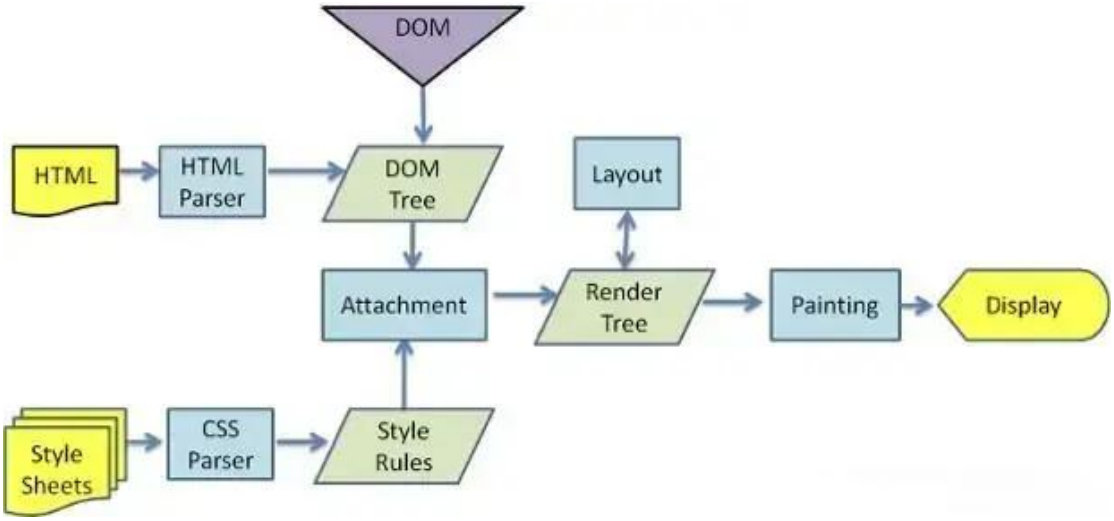
关闭 TCP 连接：通过四次挥手释放 TCP 连接

14.
15.

浏览器渲染：客户端（浏览器）解析 HTML 内容并渲染出来，浏览器接收到数据包后的解析流程为：

16.

- 构建 DOM 树：词法分析然后解析成 DOM 树（dom tree），是由 dom 元素及属性节点组成，树的根是 document 对象
- 构建 CSS 规则树：生成 CSS 规则树（CSS Rule Tree）
- 构建 render 树：Web 浏览器将 DOM 和 CSSOM 结合，并构建出渲染树（render tree）
- 布局（Layout）：计算出每个节点在屏幕中的位置
- 绘制（Painting）：即遍历 render 树，并使用 UI 后端层绘制每个节点。



17.

JS 引擎解析过程：调用 JS 引擎执行 JS 代码（JS 的解释阶段，预处理阶段，执行阶段生成执行上下文，VO，作用域链、回收机制等等）

18.

- 创建 window 对象：window 对象也叫全局执行环境，当页面产生时就被创建，所有的全局变量和函数都属于 window 的属性和方法，而 DOM Tree 也会映射在 window 的 doucment 对象上。当关闭网页或者关闭浏览器时，全局执行环境会被销毁。
- 加载文件：完成 js 引擎分析它的语法与词法是否合法，如果合法进入预编译
- 预编译：在预编译的过程中，浏览器会寻找全局变量声明，把它作为 window 的属性加入到 window 对象中，并给变量赋值为'undefined'；寻找全局函数声明，把它作为 window 的方法加入到 window 对象中，并将函数体赋值给他（匿名函数是不参与预编译的，因为它是变量）。而变量提升作为不合理的地方在 ES6 中已经解决了，函数提升还存在。
- 解释执行：执行到变量就赋值，如果变量没有被定义，也就没有被预编译直接赋值，在 ES5 非严格模式下这个变量会成为 window 的一个属性，也就是成为全局变量。string、int 这样的值就是直接把值放在变量的存储空间里，object 对象就是把指针指向变量的存储空间。函数执行，就将函数的环境推入一个环境的栈中，执行完成后再弹出，控制权交还给之前的环境。JS 作用域其实就是这样的执行流机制实现的。

传送门 ☞ # DNS 域名解析过程 ☞# 浏览器的工作原理

浏览器重绘与重排的区别？

重排/回流（Reflow）：当 DOM 的变化影响了元素的几何信息，浏览器需要重新计算元素的几何属性，将其安放在界面中的正确位置，这个过程叫做重排。表现为重新生成布局，重新排列元素。
重绘(Repaint)：当一个元素的外观发生改变，但没有改变布局,重新把元素外观绘制出来的过程，叫做重绘。表现为某些元素的外观被改变

单单改变元素的外观，肯定不会引起网页重新生成布局，但当浏览器完成重排之后，将会重新绘制受到此次重排影响的部分

重排和重绘代价是高昂的，它们会破坏用户体验，并且让 UI 展示非常迟缓，而相比之下重排的性能影响更大，在两者无法避免的情况下，一般我们宁可选择代价更小的重绘。

『重绘』不一定会出现『重排』，『重排』必然会出现『重绘』。

如何触发重排和重绘？

任何改变用来构建渲染树的信息都会导致一次重排或重绘：

添加、删除、更新 DOM 节点
通过 display: none 隐藏一个 DOM 节点-触发重排和重绘
通过 visibility: hidden 隐藏一个 DOM 节点-只触发重绘，因为没有几何变化
移动或者给页面中的 DOM 节点添加动画
添加一个样式表，调整样式属性
用户行为，例如调整窗口大小，改变字号，或者滚动。

如何避免重绘或者重排？

1.

集中改变样式，不要一条一条地修改 DOM 的样式。

2.
3.

不要把 DOM 结点的属性值放在循环里当成循环里的变量。

4.
5.

为动画的 HTML 元件使用 fixed 或 absoult 的 position，那么修改他们的 CSS 是不会 reflow 的。

6.

7.

不使用 table 布局。因为可能很小的一个小改动会造成整个 table 的重新布局。

8.
9.

尽量只修改 position：absolute 或 fixed 元素，对其他元素影响不大

10.
11.

动画开始 GPU 加速，translate 使用 3D 变化

12.
13.

提升为合成层

14.

将元素提升为合成层有以下优点：

15.

- 合成层的位图，会交由 GPU 合成，比 CPU 处理要快
- 当需要 repaint 时，只需要 repaint 本身，不会影响到其他的层
- 对于 transform 和 opacity 效果，不会触发 layout 和 paint

提升合成层的最好方式是使用 CSS 的 will-change 属性：

```
#target {
  will-change: transform;
}复制代码
```

关于合成层的详解请移步[无线性能优化：Composite]

介绍下 304 过程

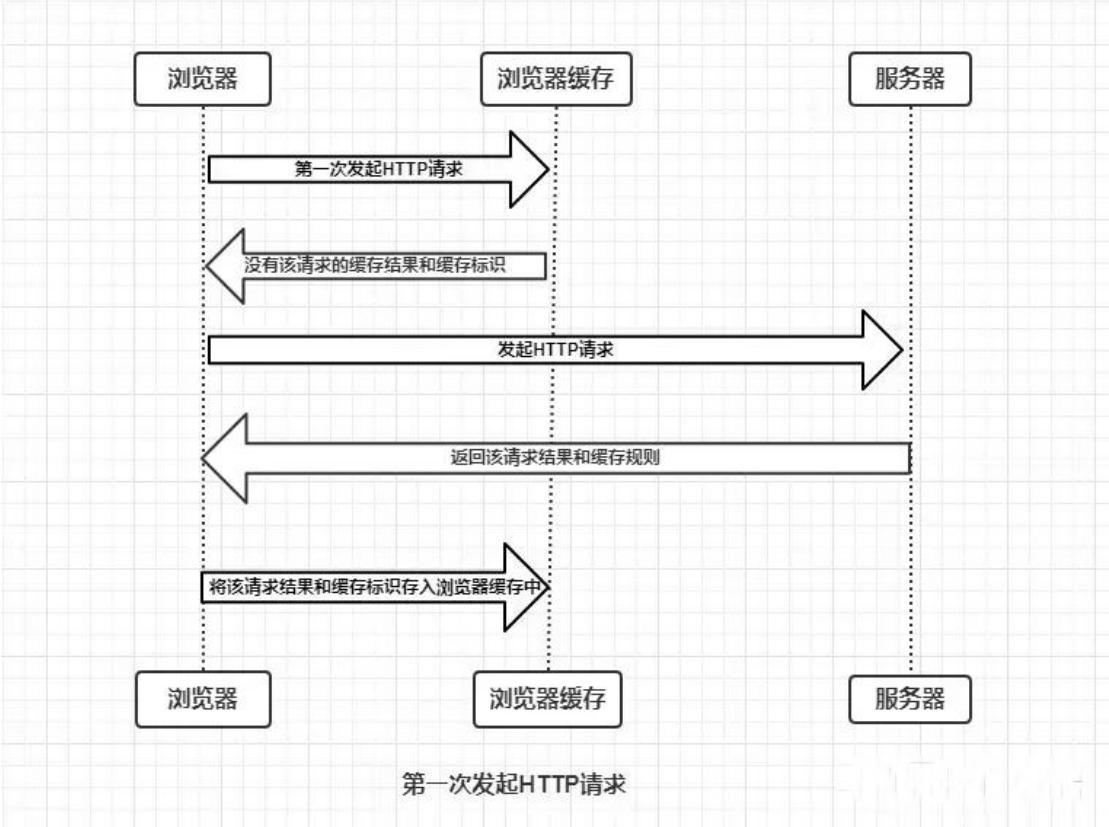
a. 浏览器请求资源时首先命中资源的 Expires 和 Cache-Control，Expires 受限于本地时间，如果修改了本地时间，可能会造成缓存失效，可以通过 Cache-control: max-age 指定最大生命周期，状态仍然返回 200，但不会请求数据，在浏览器中能明显看到 from cache 字样。

b. 强缓存失效，进入协商缓存阶段，首先验证 ETagETag 可以保证每一个资源是唯一的，资源变化都会导致 ETag 变化。服务器根据客户端上送的 If-None-Match 值来判断是否命中缓存。

c. 协商缓存 Last-Modify/If-Modify-Since 阶段，客户端第一次请求资源时，服务服返回的 header 中会加上 Last-Modify，Last-modify 是一个时间标识该资源的最后修改时间。再次请求该资源时，request 的请求头中会包含 If-Modify-Since，该值为缓存之前返回的 Last-Modify。服务器收到 If-Modify-Since 后，根据资源的最后修改时间判断是否命中缓存。

浏览器的缓存机制 强制缓存 && 协商缓存

浏览器与服务器通信的方式为应答模式，即是：浏览器发起 HTTP 请求 – 服务器响应该请求。那么浏览器第一次向服务器发起该请求后拿到请求结果，会根据响应报文中 HTTP 头的缓存标识，决定是否缓存结果，是则将请求结果和缓存标识存入浏览器缓存中，简单的过程如下图：



由上图我们可以知道：

浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识
浏览器每次拿到返回的请求结果都会将该结果和缓存标识存入浏览器缓存中

以上两点结论就是浏览器缓存机制的关键，他确保了每个请求的缓存存入与读取，只要我们再理解浏览器缓存的使用规则，那么所有的问题就迎刃而解了。为了方

便理解，这里根据是否需要向服务器重新发起 HTTP 请求将缓存过程分为两个部分，分别是强制缓存和协商缓存。

强制缓存

强制缓存就是向浏览器缓存查找该请求结果，并根据该结果的缓存规则来决定是否使用该缓存结果的过程。当浏览器向服务器发起请求时，服务器会将缓存规则放入 HTTP 响应报文的 HTTP 头中和请求结果一起返回给浏览器，控制强制缓存的字段分别是 `Expires` 和 `Cache-Control`，其中 Cache-Control 优先级比 Expires 高。

强制缓存的情况主要有三种(暂不分析协商缓存过程)，如下：

1. 不存在该缓存结果和缓存标识，强制缓存失效，则直接向服务器发起请求（跟第一次发起请求一致）。
2. 存在该缓存结果和缓存标识，但该结果已失效，强制缓存失效，则使用协商缓存。
3. 存在该缓存结果和缓存标识，且该结果尚未失效，强制缓存生效，直接返回该结果

协商缓存

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，同样，协商缓存的标识也是在响应报文的 HTTP 头中和请求结果一起返回给浏览器的，控制协商缓存的字段分别有：`Last-Modified / If-Modified-Since` 和 `Etag / If-None-Match`，其中 Etag / If-None-Match 的优先级比 Last-Modified / If-Modified-Since 高。协商缓存主要有以下两种情况：

1. 协商缓存生效，返回 304
2. 协商缓存失效，返回 200 和请求结果结果

说下进程、线程和协程

进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。进程是一种抽象的概念，从来没有统一的标准定义。

线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。一个标准的线程由线程 ID、当前指令指针(PC)、寄存器和堆栈组成。而进程由内存空间(代码、数据、进程空间、打开的文件)和一个或多个线程组成。

协程，英文 Coroutines，是一种基于线程之上，但又比线程更加轻量级的存在，这种由程序员自己写程序来管理的轻量级线程叫做『用户空间线程』，具有对内核来说不可见的特性。

进程和线程的区别与联系

【区别】：
调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位；
并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行；
拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。
系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。但是进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个进程死掉就等于所有的线程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

【联系】： 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程；
资源分配给进程，同一进程的所有线程共享该进程的所有资源；
处理机分给线程，即真正在处理机上运行的是线程；
线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。

HTML && CSS

HTML5 新特性、语义化

1.

概念：

2.

HTML5 的语义化指的是合理正确的使用语义化的标签来创建页面结构。【正确的标签做正确的事】

3.
4.

语义化标签：

5.

header nav main article section aside footer

6.
7.

语义化的优点：

8.

- 在没 CSS 样式的情况下，页面整体也会呈现很好的结构效果
- 代码结构清晰，易于阅读，
- 利于开发和维护 方便其他设备解析（如屏幕阅读器）根据语义渲染网页。
- 有利于搜索引擎优化（SEO），搜索引擎爬虫会根据不同的标签来赋予不同的权重

CSS 选择器及优先级

选择器

id 选择器(#myid)
类选择器(.myclass)
属性选择器(a[rel="external"])
伪类选择器(a:hover, li:nth-child)
标签选择器(div, h1,p)
相邻选择器（h1 + p）
子选择器(ul > li)
后代选择器(li a)
通配符选择器(*)

优先级：

!important
内联样式（1000）
ID 选择器（0100）
类选择器/属性选择器/伪类选择器（0010）
元素选择器/伪元素选择器（0001）
关系选择器/通配符选择器（0000）

带!important 标记的样式属性优先级最高； 样式表的来源相同时：!important > 行内样式>ID 选择器 > 类选择器 > 标签 > 通配符 > 继承 > 浏览器默认属性

position 属性的值有哪些及其区别

固定定位 fixed： 元素的位置相对于浏览器窗口是固定位置，即使窗口是滚动的它也不会移动。Fixed 定 位使元素的位置与文档流无关，因此不占据空间。 Fixed 定位的元素和其他元素重叠。

相对定位 relative：如果对一个元素进行相对定位，它将出现在它所在的位置上。然后，可以通过设置垂直 或水平位置，让这个元素“相对于”它的起点进行移动。在使用相对定位时，无论是 否进行移动，元素仍然占据原来的空间。因此，移动元素会导致它覆盖其它框。

绝对定位 absolute： 绝对定位的元素的位置相对于最近的已定位父元素，如果元素没有已定位的父元素，那 么它的位置相对于。absolute 定位使元素的位置与文档流无关，因此不占据空间。 absolute 定位的元素和其他元素重叠。

粘性定位 sticky：元素先按照普通文档流定位，然后相对于该元素在流中的 flow root（BFC）和 containing block（最近的块级祖先元素）定位。而后，元素定位表现为在跨越特定阈值前为相对定 位，之后为固定定位。

默认定位 Static： 默认值。没有定位，元素出现在正常的流中（忽略 top, bottom, left, right 或者 z-index 声 明）。 inherit: 规定应该从父元素继承 position 属性的值。

box-sizing 属性

box-sizing 规定两个并排的带边框的框，语法为 box-sizing：
content-box/border-box/inherit

content-box：宽度和高度分别应用到元素的内容框，在宽度和高度之外绘制元素的内边距和边框。【标准盒子模型】

border-box：为元素设定的宽度和高度决定了元素的边框盒。【IE 盒子模型】

inherit：继承父元素的 box-sizing 值。

CSS 盒子模型

CSS 盒模型本质上是一个盒子，它包括：边距，边框，填充和实际内容。CSS 中的盒子模型包括 IE 盒子模型和标准的 W3C 盒子模型。
在标准的盒子模型中，width 指 content 部分的宽度。
在 IE 盒子模型中，width 表示 content+padding+border 这三个部分的宽度。

故在计算盒子的宽度时存在差异：

标准盒模型： 一个块的总宽度 = width+margin(左右)+padding(左右)+border(左右)

怪异盒模型： 一个块的总宽度 = width+margin（左右）（既 width 已经包含了 padding 和 border 值）

BFC（块级格式上下文）

BFC 的概念

BFC 是 Block Formatting Context 的缩写，即块级格式化上下文。BFC 是 CSS 布局的一个概念，是一个独立的渲染区域，规定了内部 box 如何布局， 并且这个区域的子元素不会影响到外面的元素，其中比较重要的布局规则有内部 box 垂直放置，计算 BFC 的高度的时候，浮动元素也参与计算。

BFC 的原理布局规则

内部的 Box 会在垂直方向，一个接一个地放置
Box 垂直方向的距离由 margin 决定。属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠
每个元素的 margin box 的左边， 与包含块 border box 的左边相接触(对于从左往右的格式化，否则相反
BFC 的区域不会与 float box 重叠
BFC 是一个独立容器，容器里面的子元素不会影响到外面的元素
计算 BFC 的高度时，浮动元素也参与计算高度
元素的类型和 display 属性，决定了这个 Box 的类型。不同类型的 Box 会参与不同的 Formatting Context。

如何创建 BFC？

根元素，即 HTML 元素
float 的值不为 none
position 为 absolute 或 fixed

display 的值为 inline-block、table-cell、table-caption
overflow 的值不为 visible

BFC 的使用场景

去除边距重叠现象
清除浮动（让父元素的高度包含子浮动元素）
避免某元素被浮动元素覆盖
避免多列布局由于宽度计算四舍五入而自动换行

让一个元素水平垂直居中

水平居中

- 对于 行内元素 ：`text-align: center;`
- 对于确定宽度的块级元素：
  - （1）width 和 margin 实现。`margin: 0 auto;`
  - （2）绝对定位和 margin-left: (父 width - 子 width）/2, 前提是父元素 position: relative
- 对于宽度未知的块级元素

（1）table 标签配合 margin 左右 auto 实现水平居中。使用 table 标签（或直接将块级元素设值为 display:table），再通过给该标签添加左右 margin 为 auto。

- （2）inline-block 实现水平居中方法。display：inline-block 和 text-align:center 实现水平居中。

- （3）绝对定位+transform，translateX 可以移动本身元素的 50%。

- （4）flex 布局使用 justify-content:center

- **垂直居中**

1. 利用 line-height 实现居中，这种方法适合纯文字类
2. 通过设置父容器 相对定位 ，子级设置 绝对定位，标签通过 margin 实现自适应居中
3. 弹性布局 flex :父级设置 display: flex; 子级设置 margin 为 auto 实现自适应居中
4. 父级设置相对定位，子级设置绝对定位，并且通过位移 transform 实现
5. table 布局，父级通过转换成表格形式，然后子级设置 vertical-align 实现。（需要注意的是：vertical-align: middle 使用的前提条件是内联元素以及 display 值为 table-cell 的元素）。

隐藏页面中某个元素的方法

1.opacity：0，该元素隐藏起来了，但不会改变页面布局，并且，如果该元素已经绑定 一些事件，如 click 事件，那么点击该区域，也能触发点击事件的

2.visibility：hidden，该元素隐藏起来了，但不会改变页面布局，但是不会触发该元素已 经绑定的事件 ，隐藏对应元素，在文档布局中仍保留原来的空间（重绘）

3.display：none，把元素隐藏起来，并且会改变页面布局，可以理解成在页面中把该元素。 不显示对应的元素，在文档布局中不再分配空间（回流+重绘）

该问题会引出 回流和重绘

用 CSS 实现三角符号

```
/*记忆口诀：盒子宽高均为零，三面边框皆透明。 */div:after{
    position: absolute;
    width: 0px;
    height: 0px;
    content: ″ ″;
    border-right: 100px solid transparent;
    border-top: 100px solid #ff0;
    border-left: 100px solid transparent;
    border-bottom: 100px solid transparent;
}复制代码
```

页面布局

1.Flex 布局

布局的传统解决方案，基于盒状模型，依赖 display 属性 + position 属性 + float 属性。它对于那些特殊布局非常不方便，比如，垂直居中就不容易实现。

Flex 是 Flexible Box 的缩写，意为"弹性布局",用来为盒状模型提供最大的灵活性。指定容器 display: flex 即可。 简单的分为容器属性和元素属性。

容器的属性：

flex-direction：决定主轴的方向（即子 item 的排列方法）flex-direction: row | row-reverse | column | column-reverse;
flex-wrap：决定换行规则 flex-wrap: nowrap | wrap | wrap-reverse;
flex-flow： .box { flex-flow: || ; }
justify-content：对其方式，水平主轴对齐方式
align-items：对齐方式，竖直轴线方向
align-content

项目的属性（元素的属性）：

order 属性：定义项目的排列顺序，顺序越小，排列越靠前，默认为 0
flex-grow 属性：定义项目的放大比例，即使存在空间，也不会放大

flex-shrink 属性：定义了项目的缩小比例，当空间不足的情况下会等比例的缩小，如果 定义个 item 的 flow-shrink 为 0，则为不缩小
flex-basis 属性：定义了在分配多余的空间，项目占据的空间。
flex：是 flex-grow 和 flex-shrink、flex-basis 的简写，默认值为 0 1 auto。
align-self：允许单个项目与其他项目不一样的对齐方式，可以覆盖
align-items，默认属 性为 auto，表示继承父元素的 align-items 比如说，用 flex 实现圣杯布局

2.Rem 布局

首先 Rem 相对于根(html)的 font-size 大小来计算。简单的说它就是一个相对单例 如:font-size:10px;,那么（1rem = 10px）了解计算原理后首先解决怎么在不同设备上设置 html 的 font-size 大小。其实 rem 布局的本质是等比缩放，一般是基于宽度。

优点：可以快速适用移动端布局，字体，图片高度

缺点：

①目前 ie 不支持，对 pc 页面来讲使用次数不多；
②数据量大：所有的图片，盒子都需要我们去给一个准确的值；才能保证不同机型的适配；
③在响应式布局中，必须通过 js 来动态控制根元素 font-size 的大小。也就是说 css 样式和 js 代码有一定的耦合性。且必须将改变 font-size 的代码放在 css 样式之前。

3.百分比布局

通过百分比单位 "%" 来实现响应式的效果。通过百分比单位可以使得浏览器中的组件的宽和高随着浏览器的变化而变化，从而实现响应式的效果。 直观的理解，我们可能会认为子元素的百分比完全相对于直接父元素，height 百分比相 对于 height，width 百分比相对于 width。 padding、border、margin 等等不论是垂直方向还是水平方向，都相对于直接父元素的 width。除了 border-radius 外，还有比如 translate、background-size 等都是相对于自身的。

缺点：

（1）计算困难
（2）各个属性中如果使用百分比，相对父元素的属性并不是唯一的。造成我们使用百分比单位容易使布局问题变得复杂。

4.浮动布局

浮动布局:当元素浮动以后可以向左或向右移动，直到它的外边缘碰到包含它的框或者另外一个浮动元素的边框为止。元素浮动以后会脱离正常的文档流，所以文档的普通流中的框就变的好像浮动元素不存在一样。

优点

这样做的优点就是在图文混排的时候可以很好的使文字环绕在图片周围。另外当元素浮动了起来之后，它有着块级元素的一些性质例如可以设置宽高等，但它与 inline-block 还是有一些区别的，第一个就是关于横向排序的时候，float 可以设置方向而 inline-block 方向是固定的；还有一个就是 inline-block 在使用时有时会有空白间隙的问题

缺点

最明显的缺点就是浮动元素一旦脱离了文档流，就无法撑起父元素，会造成父级元素高度塌陷。

如何使用 rem 或 viewport 进行移动端适配

rem 适配原理：

改变了一个元素在不同设备上占据的 css 像素的个数

rem 适配的优缺点

> 优点：没有破坏完美视口
> 缺点：px 值转换 rem 太过于复杂(下面我们使用 less 来解决这个问题)

viewport 适配的原理

viewport 适配方案中，每一个元素在不同设备上占据的 css 像素的个数是一样的。但是 css 像素和物理像素的比例是不一样的，等比的

viewport 适配的优缺点

> 在我们设计图上所量取的大小即为我们可以设置的像素大小，即所量即所设
> 缺点破坏完美视口

清除浮动的方式

> 添加额外标签

```
<div class="parent">
    //添加额外标签并且添加 clear 属性
```

```
    <div style="clear:both"></div>
    //也可以加一个 br 标签</div>复制代码

        父级添加 overflow 属性，或者设置高度
        建立伪类选择器清除浮动

//在 css 中添加:after 伪元素
.parent:after{
    /* 设置添加子元素的内容是空 */
    content: '';
    /* 设置添加子元素为块级元素 */
    display: block;
    /* 设置添加的子元素的高度 0 */
    height: 0;
    /* 设置添加子元素看不见 */
    visibility: hidden;
    /* 设置 clear：both */
    clear: both;
}复制代码
```

JS、TS、ES6

JS 中的 8 种数据类型及区别

包括值类型(基本对象类型)和引用类型(复杂对象类型)

基本类型(值类型)： Number(数字),String(字符串),Boolean(布尔),Symbol(符号),null(空),undefined(未定义)在内存中占据固定大小，保存在栈内存中

引用类型(复杂数据类型)： Object(对象)、Function(函数)。其他还有 Array(数组)、Date(日期)、RegExp(正则表达式)、特殊的基本包装类型(String、Number、Boolean)以及单体内置对象(Global、Math)等 引用类型的值是对象 保存在堆内存中，栈内存存储的是对象的变量标识符以及对象在堆内存中的存储地址。

JS 中的数据类型检测方案

1.typeof

```
console.log(typeof 1);              // numberconsole.log(typeof true);
// booleanconsole.log(typeof 'mc');          //
stringconsole.log(typeof Symbol)          // functionconsole.log(typeof
function(){});    // functionconsole.log(typeof console.log());   //
```

```
functionconsole.log(typeof []);            // object console.log(typeof {});            // objectconsole.log(typeof null);          // objectconsole.log(typeof undefined);      // undefined 复制代码
```

优点：能够快速区分基本数据类型

缺点：不能将 Object、Array 和 Null 区分，都返回 object

2.instanceof

```
console.log(1 instanceof Number);                  // falseconsole.log(true instanceof Boolean);              // falseconsole.log('str' instanceof String);              // falseconsole.log([] instanceof Array);                  // trueconsole.log(function(){} instanceof Function);      // trueconsole.log({} instanceof Object);                // true 复制代码
```

优点：能够区分 Array、Object 和 Function，适合用于判断自定义的类实例对象

缺点：Number，Boolean，String 基本数据类型不能判断

3.Object.prototype.toString.call()

```
var toString = Object.prototype.toString;console.log(toString.call(1));//[object Number]console.log(toString.call(true));//[object Boolean]console.log(toString.call('mc'));//[object String]console.log(toString.call([]));//[object Array]console.log(toString.call({}));//[object Object]console.log(toString.call(function(){}));//[object Function]console.log(toString.call(undefined));//[object Undefined]console.log(toString.call(null));//[object Null]复制代码
```

优点：精准判断数据类型

缺点：写法繁琐不容易记，推荐进行封装后使用

var && let && const

ES6 之前创建变量用的是 var,之后创建变量用的是 let/const

三者区别:

1. var 定义的变量，没有块的概念，可以跨块访问, 不能跨函数访问。
   let 定义的变量，只能在块作用域里访问，不能跨块访问，也不能跨函数访问。

const 用来定义常量，使用时必须初始化(即必须赋值)，只能在块作用域里访问，且不能修改。

2. var 可以先使用，后声明，因为存在变量提升；let 必须先声明后使用。
3. var 是允许在相同作用域内重复声明同一个变量的，而 let 与 const 不允许这一现象。
4. 在全局上下文中，基于 let 声明的全局变量和全局对象 GO（window）没有任何关系 ；var 声明的变量会和 GO 有映射关系；
5. 会产生暂时性死区：

暂时性死区是浏览器的 bug：检测一个未被声明的变量类型时，不会报错，会返回 undefined
如：console.log(typeof a) //undefined
而：console.log(typeof a)//未声明之前不能使用
let a

1. let /const/function 会把当前所在的大括号(除函数之外)作为一个全新的块级上下文，应用这个机制，在开发项目的时候，遇到循环事件绑定等类似的需求，无需再自己构建闭包来存储，只要基于 let 的块作用特征即可解决

JS 垃圾回收机制

1.

项目中，如果存在大量不被释放的内存（堆/栈/上下文），页面性能会变得很慢。当某些代码操作不能被合理释放，就会造成内存泄漏。我们尽可能减少使用闭包，因为它会消耗内存。

2.

3.

浏览器垃圾回收机制/内存回收机制:

4.

浏览器的 Javascript 具有自动垃圾回收机制(GC:Garbage Collecation)，垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存。

5.

标记清除:在 js 中，最常用的垃圾回收机制是标记清除：当变量进入执行环境时，被标记为“进入环境”，当变量离开执行环境时，会被标记为“离开环境”。垃圾回收器会销毁那些带标记的值并回收它们所占用的内存空间。
谷歌浏览器：“查找引用”，浏览器不定时去查找当前内存的引用，如果没

有被占用了，浏览器会回收它；如果被占用，就不能回收。
IE 浏览器：“引用计数法”，当前内存被占用一次，计数累加 1 次，移除占用就减 1，减到 0 时，浏览器就回收它。

6.
7.

优化手段：内存优化 ；手动释放：取消内存的占用即可。

8.

（1）堆内存：fn = null 【null：空指针对象】

9.

（2）栈内存：把上下文中，被外部占用的堆的占用取消即可。

10.
11.

内存泄漏

12.

在 JS 中，常见的内存泄露主要有 4 种,全局变量、闭包、DOM 元素的引用、定时器

13.

作用域和作用域链

创建函数的时候，已经声明了当前函数的作用域==>当前创建函数所处的上下文。如果是在全局下创建的函数就是[[scope]]:EC(G)，函数执行的时候，形成一个全新的私有上下文 EC(FN)，供字符串代码执行(进栈执行)

定义：简单来说作用域就是变量与函数的可访问范围，由当前环境与上层环境的一系列变量对象组成
1.全局作用域：代码在程序的任何地方都能被访问，window 对象的内置属性都拥有全局作用域。
2.函数作用域：在固定的代码片段才能被访问

作用：作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

作用域链参考链接一般情况下，变量到 创建该变量 的函数的作用域中取值。但是如果在当前作用域中没有查到，就会向上级作用域去查，直到查到全局作用域，这么一个查找过程形成的链条就叫做作用域链。

闭包的两大作用：保存/保护

闭包的概念

函数执行时形成的私有上下文 EC(FN)，正常情况下，代码执行完会出栈后释放;但是特殊情况下，如果当前私有上下文中的某个东西被上下文以外的事物占用了，则上下文不会出栈释放，从而形成不销毁的上下文。 函数执行函数执行过程中，会形成一个全新的私有上下文，可能会被释放，可能不会被释放，不论释放与否，他的作用是：

（1）保护：划分一个独立的代码执行区域，在这个区域中有自己私有变量存储的空间，保护自己的私有变量不受外界干扰（操作自己的私有变量和外界没有关系）；

（2）保存：如果当前上下文不被释放【只要上下文中的某个东西被外部占用即可】，则存储的这些私有变量也不会被释放，可以供其下级上下文中调取使用，相当于把一些值保存起来了；

我们把函数执行形成私有上下文，来保护和保存私有变量机制称为闭包。

> 闭包是指有权访问另一个函数作用域中的变量的函数--
> 《JavaScript 高级程序设计》

稍全面的回答： 在 js 中变量的作用域属于函数作用域, 在函数执行完后,作用域就会被清理,内存也会随之被回收,但是由于闭包函数是建立在函数内部的子函数,由于其可访问上级作用域,即使上级函数执行完, 作用域也不会随之销毁, 这时的子函数(也就是闭包),便拥有了访问上级作用域中变量的权限,即使上级函数执行完后作用域内的值也不会被销毁。

闭包的特性：

-

1、内部函数可以访问定义他们外部函数的参数和变量。(作用域链的向上查找，把外围的作用域中的变量值存储在内存中而不是在函数调用完毕后销毁)设计私有的方法和变量，避免全局变量的污染。

-

1.1.闭包是密闭的容器，，类似于 set、map 容器，存储数据的

-

1.2.闭包是一个对象，存放数据的格式为 key-value 形式

-
  -

2、函数嵌套函数

-
-

3、本质是将函数内部和外部连接起来。优点是可以读取函数内部的变量，让这些变量的值始终保存在内存中，不会在函数被调用之后自动清除

-

闭包形成的条件：

1. 函数的嵌套
2. 内部函数引用外部函数的局部变量，延长外部函数的变量生命周期

闭包的用途：

1. 模仿块级作用域
2. 保护外部函数的变量 能够访问函数定义时所在的词法作用域(阻止其被回收)

3. 封装私有化变量
4. 创建模块

闭包应用场景

闭包的两个场景，闭包的两大作用：保存/保护。 在开发中, 其实我们随处可见闭包的身影, 大部分前端 JavaScript 代码都是“事件驱动”的,即一个事件绑定的回调方法; 发送 ajax 请求成功|失败的回调;setTimeout 的延时回调;或者一个函数内部返回另一个匿名函数,这些都是闭包的应用。

闭包的优点：延长局部变量的生命周期

闭包缺点：会导致函数的变量一直保存在内存中，过多的闭包可能会导致内存泄漏

JS 中 this 的五种情况

1. 作为普通函数执行时，`this` 指向 `window`。
2. 当函数作为对象的方法被调用时，`this` 就会指向该对象。
3. 构造器调用，`this` 指向返回的这个对象。
4. 箭头函数 箭头函数的 `this` 绑定看的是 `this` 所在函数定义在哪个对象下，就绑定哪个对象。如果有嵌套的情况，则 this 绑定到最近的一层对象上。
5. 基于 Function.prototype 上的 `apply` 、 `call` 和 `bind` 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。`apply` 接收参数的是数组，`call` 接受参数列表，`` bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new `时会被改变，其他情况下都不会改变。若为空默认是指向全局对象 window。

原型 && 原型链

原型关系：

每个 class 都有显示原型 prototype
每个实例都有隐式原型 _ proto_
实例的_ proto_指向对应 class 的 prototype

原型：在 JS 中，每当定义一个对象（函数也是对象）时，对象中都会包含一些预定义的属性。其中每个函数对象都有一个 prototype 属性，这个属性指向函数的原型对象。

原型链：函数的原型链对象 constructor 默认指向函数本身，原型对象除了有原型属性外，为了实现继承，还有一个原型链指针__proto__,该指针是指向上一层的原型对象，而上一层的原型对象的结构依然类似。因此可以利用__proto__一直指向 Object 的原型对象上，而 Object 原型对象用 Object.prototype.__ proto__ = null 表示原型链顶端。如此形成了 js 的原型链继承。同时所有的 js 对象都有 Object 的基本防范

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

new 运算符的实现机制

1. 首先创建了一个新的空对象
2. 设置原型，将对象的原型设置为函数的 prototype 对象。
3. 让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）
4. 判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

EventLoop 事件循环

JS 是单线程的，为了防止一个函数执行时间过长阻塞后面的代码，所以会先将同步代码压入执行栈中，依次执行，将异步代码推入异步队列，异步队列又分为宏任务队列和微任务队列，因为宏任务队列的执行时间较长，所以微任务队列要优先于宏任务队列。微任务队列的代表就是，Promise.then，MutationObserver，宏任务的话就是 setImmediate setTimeout setInterval

JS 运行的环境。一般为浏览器或者 Node。 在浏览器环境中，有 JS 引擎线程和渲染线程，且两个线程互斥。 Node 环境中，只有 JS 线程。 不同环境执行机制有差异，不同任务进入不同 Event Queue 队列。 当主程结束，先执行准备好微任务，然后再执行准备好的宏任务，一个轮询结束。

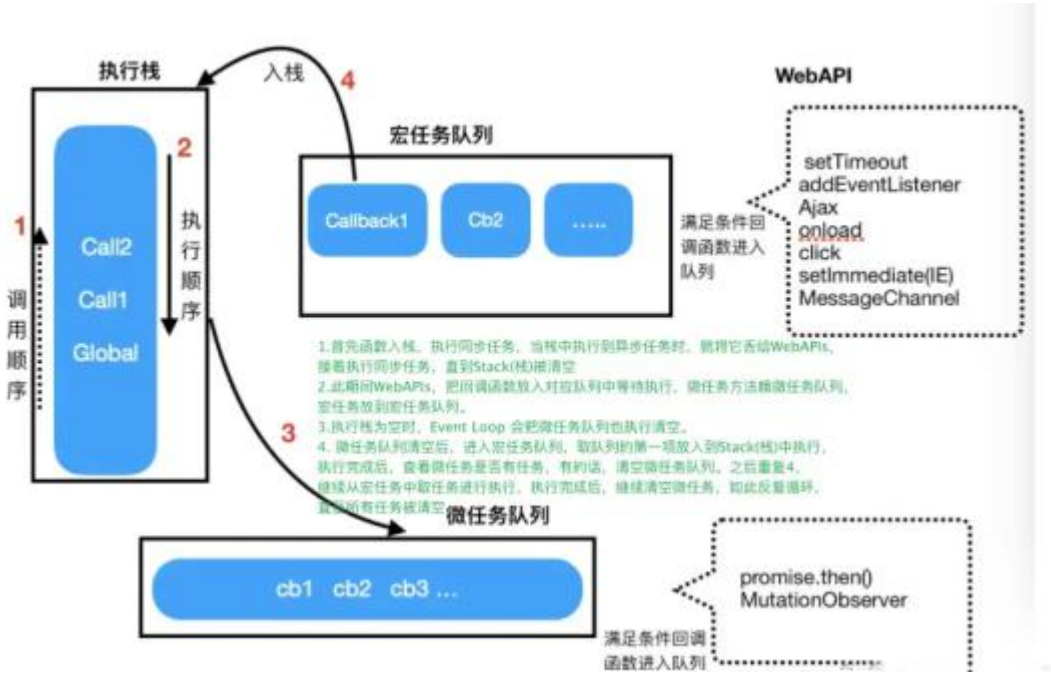
浏览器中的事件环（Event Loop）

事件环的运行机制是，先会执行栈中的内容，栈中的内容执行后执行微任务，微任务清空后再执行宏任务，先取出一个宏任务，再去执行微任务，然后在取宏任务清微任务这样不停的循环。

eventLoop 是由 JS 的宿主环境（浏览器）来实现的；

事件循环可以简单的描述为以下四个步骤:

1. 函数入栈，当 Stack 中执行到异步任务的时候，就将他丢给 WebAPIs,接着执行同步任务,直到 Stack 为空；
2. 此期间 WebAPIs 完成这个事件，把回调函数放入队列中等待执行（微任务放到微任务队列，宏任务放到宏任务队列）
3. 执行栈为空时，Event Loop 把微任务队列执行清空；
4. 微任务队列清空后，进入宏任务队列，取队列的第一项任务放入 Stack(栈）中执行，执行完成后，查看微任务队列是否有任务，有的话，清空微任务队列。重复 4，继续从宏任务中取任务执行，执行完成之后，继续清空微任务，如此反复循环，直至清空所有的任务。

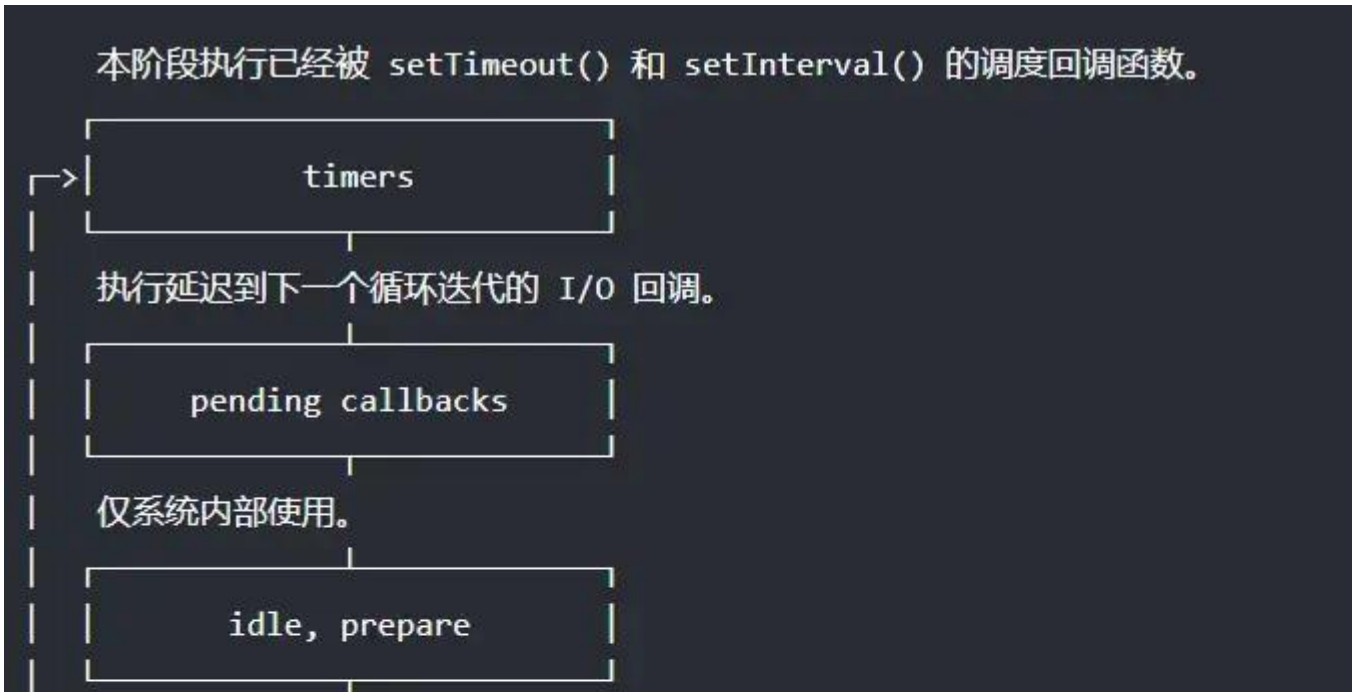


浏览器中的任务源(task):

- 宏任务(macrotask)：
  宿主环境提供的，比如浏览器
  ajax 、 setTimeout 、 setInterval 、 setTmmediate(只 兼 容 ie) 、 script 、 requestAnimationFrame、messageChannel、UI 渲染、一些浏览器 api
- 微任务(microtask)：
  语言本身提供的，比如 promise.then
  then 、 queueMicrotask(基 于 then) 、 mutationObserver(浏 览 器 提 供) 、 messageChannel 、mutationObersve

Node 环境中的事件环（Event Loop）

Node 是基于 V8 引擎的运行在服务端的 JavaScript 运行环境，在处理高并发、I/O 密集(文件操作、网络操作、数据库操作等)场景有明显的优势。虽然用到也是 V8 引擎，但由于服务目的和环境不同，导致了它的 API 与原生 JS 有些区别，其 Event Loop 还要处理一些 I/O，比如新的网络连接等，所以 Node 的 Event Loop(事件环机制)与浏览器的是不太一样。



执行顺序如下：

timers: 计时器，执行 setTimeout 和 setInterval 的回调
pending callbacks: 执行延迟到下一个循环迭代的 I/O 回调
idle, prepare: 队列的移动，仅系统内部使用
poll 轮询: 检索新的 I/O 事件;执行与 I/O 相关的回调。事实上除了其他几个阶段处理的事情，其他几乎所有的异步都在这个阶段处理。
check: 执行 setImmediate 回调，setImmediate 在这里执行
close callbacks: 执行 close 事件的 callback，一些关闭的回调函数，如：socket.on('close', ...)

setTimeout、Promise、Async/Await 的区别

1.

setTimeout

2.

settimeout 的回调函数放到宏任务队列里，等到执行栈清空以后执行。

3.

4.

Promise

5.

Promise 本身是**同步的立即执行函数**， 当在 executor 中执行 resolve 或者 reject 的时候, 此时是异步操作， 会先执行 then/catch 等，当主栈完成后，才会去调用 resolve/reject 中存放的方法执行。

6.

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

复制代码

7.

8.

async/await

9.

async 函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

10.

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

11.

Async/Await 如何通过同步的方式实现异步

Async/Await 就是一个**自执行**的 generate 函数。利用 generate 函数的特性把异步的代码写成“同步”的形式,第一个请求的返回值作为后面一个请求的参数,其中每一个参数都是一个 promise 对象.

介绍节流防抖原理、区别以及应用

节流：事件触发后，规定时间内，事件处理函数不能再次被调用。也就是说在规定的时间内，函数只能被调用一次，且是最先被触发调用的那次。

防抖：多次触发事件，事件处理函数只能执行一次，并且是在触发操作结束时执行。也就是说，当一个事件被触发准备执行事件函数前，会等待一定的时间（这时间是码农自己去定义的，比如 1 秒），如果没有再次被触发，那么就执行，如果被触发了,那就本次作废,重新从新触发的时间开始计算,并再次等待 1 秒，直到能最终执行!

使用场景：
节流：滚动加载更多、搜索框搜的索联想功能、高频点击、表单重复提交……
防抖：搜索框搜索输入，并在输入完以后自动搜索、手机号，邮箱验证输入检测、窗口大小 resize 变化后，再重新渲染。

/**

```
 * 节流函数 一个函数执行一次后，只有大于设定的执行周期才会执行第二次。有个需要频繁触发的函数，出于优化性能的角度，在规定时间内，只让函数触发的第一次生效，后面的不生效。
 * @param fn 要被节流的函数
 * @param delay 规定的时间
 */function throttle(fn, delay) {
    //记录上一次函数触发的时间
    var lastTime = 0;
    return function(){
        //记录当前函数触发的时间
        var nowTime = Date.now();
        if(nowTime - lastTime > delay){
            //修正 this 指向问题
            fn.call(this);
            //同步执行结束时间
            lastTime = nowTime;
        }
    }
}
document.onscroll = throttle(function () {
    console.log('scllor 事件被触发了' + Date.now());
}, 200);
/**
 * 防抖函数  一个需要频繁触发的函数，在规定时间内，只让最后一次生效，前面的不生效
 * @param fn 要被节流的函数
 * @param delay 规定的时间
 */function debounce(fn, delay) {
    //记录上一次的延时器
    var timer = null;
    return function () {
       //清除上一次的演示器
        clearTimeout(timer);
        //重新设置新的延时器
        timer = setTimeout(function(){
            //修正 this 指向问题
            fn.apply(this);
        }, delay);
    }
}document.getElementById('btn').onclick = debounce(function () {
    console.log('按钮被点击了' + Date.now());
}, 1000);复制代码
```

简述 MVVM

什么是 MVVM？

视图模型双向绑定，是 Model-View-ViewModel 的缩写，也就是把 MVC 中的 Controller 演变成 ViewModel。Model 层代表数据模型，View 代表 UI 组件，ViewModel 是 View 和 Model 层的桥梁，数据会绑定到 viewModel 层并自动将数据渲染到页面中，视图变化的时候会通知 viewModel 层更新数据。以前是操作 DOM 结构更新视图，现在是数据驱动视图。

MVVM 的优点：

1.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；
2.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。
3.独立开发。开发人员可以专注于业务逻辑和数据的开发(ViewModel)，设计人员可以专注于页面设计。
4.可测试。

Vue 底层实现原理

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty()来劫持各个属性的 setter 和 getter，在数据变动时发布消息给订阅者，触发相应的监听回调
Vue 是一个典型的 MVVM 框架，模型（Model）只是普通的 javascript 对象，修改它则试图（View）会自动更新。这种设计让状态管理变得非常简单而直观

Observer（数据监听器）：Observer 的核心是通过 Object.defineProprtty()来监听数据的变动，这个函数内部可以定义 setter 和 getter，每当数据发生变化，就会触发 setter。这时候 Observer 就要通知订阅者，订阅者就是 Watcher

Watcher（订阅者）：Watcher 订阅者作为 Observer 和 Compile 之间通信的桥梁，主要做的事情是：

1. 在自身实例化时往属性订阅器(dep)里面添加自己
2. 自身必须有一个 update()方法
3. 待属性变动 dep.notice()通知时，能调用自身的 update()方法，并触发 Compile 中绑定的回调

Compile（指令解析器）：Compile 主要做的事情是解析模板指令，将模板中变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加鉴定数据的订阅者，一旦数据有变动，收到通知，更新试图

谈谈对 vue 生命周期的理解？

每个 Vue 实例在创建时都会经过一系列的初始化过程，vue 的生命周期钩子，就是说在达到某一阶段或条件时去触发的函数，目的就是为了完成一些动作或者事件

create 阶段：vue 实例被创建
beforeCreate: 创建前，此时 data 和 methods 中的数据都还没有初始化
created： 创建完毕，data 中有值，未挂载
mount 阶段： vue 实例被挂载到真实 DOM 节点
beforeMount：可以发起服务端请求，去数据
mounted: 此时可以操作 DOM
update 阶段：当 vue 实例里面的 data 数据变化时，触发组件的重新渲染
beforeUpdate :更新前
updated：更新后
destroy 阶段：vue 实例被销毁
beforeDestroy：实例被销毁前，此时可以手动销毁一些方法
destroyed:销毁后

组件生命周期

生命周期（父子组件） 父组件 beforeCreate --> 父组件 created --> 父组件 beforeMount --> 子组件 beforeCreate --> 子组件 created --> 子组件 beforeMount --> 子组件 mounted --> 父组件 mounted -->父组件 beforeUpdate -->子组件 beforeDestroy--> 子组件 destroyed --> 父组件 updated

加载渲染过程 父 beforeCreate->父 created->父 beforeMount->子 beforeCreate->子 created->子 beforeMount->子 mounted->父 mounted

挂载阶段 父 created->子 created->子 mounted->父 mounted

父组件更新阶段 父 beforeUpdate->父 updated

子组件更新阶段 父 beforeUpdate->子 beforeUpdate->子 updated->父 updated

销毁阶段 父 beforeDestroy->子 beforeDestroy->子 destroyed->父 destroyed

computed 与 watch

通俗来讲，既能用 computed 实现又可以用 watch 监听来实现的功能，推荐用 computed， 重点在于 computed 的缓存功能 computed 计算属性是用来声明式的描述一个值依赖了其它的值，当所依赖的值或者变量 改变时，计算属性也会跟着改变； watch 监听的是已经在 data 中定义的变量，当该变量变化时，会触发 watch 中的方法。

watch 属性监听 是一个对象，键是需要观察的属性，值是对应回调函数，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,监听属性的变化，需要在数据变化时执行异步或开销较大的操作时使用

computed 计算属性 属性的结果会被缓存，当 computed 中的函数所依赖的属性没有发生改变的时候，那么调用当前函数的时候结果会从缓存中读取。除非依赖的响应式属性变化时才会重新计算，主要当做属性来使用 computed 中的函数必须用 return 返回最终的结果 computed 更高效，优先使用。data 不改变，computed 不更新。

使用场景 computed：当一个属性受多个属性影响的时候使用，例：购物车商品结算功能 watch：当一条数据影响多条数据的时候使用，例：搜索数据

组件中的 data 为什么是一个函数？

1.一个组件被复用多次的话，也就会创建多个实例。本质上，这些实例用的都是同一个构造函数。 2.如果 data 是对象的话，对象属于引用类型，会影响到所有的实例。所以为了保证组件不同的实例之间 data 不冲突，data 必须是一个函数。

为什么 v-for 和 v-if 不建议用在一起

1.当 v-for 和 v-if 处于同一个节点时，v-for 的优先级比 v-if 更高，这意味着 v-if 将分别重复运行于每个 v-for 循环中。如果要遍历的数组很大，而真正要展示的数据很少时，这将造成很大的性能浪费
2.这种场景建议使用 computed，先对数据进行过滤

注意：3.x 版本中 v-if 总是优先于 v-for 生效。由于语法上存在歧义，建议避免在同一元素上同时使用两者。比起在模板层面管理相关逻辑，更好的办法是通过创建计算属性筛选出列表，并以此创建可见元素。

React/Vue 项目中 key 的作用

key 的作用是为了在 diff 算法执行时更快的找到对应的节点，提高 diff 速度，更高效的更新虚拟 DOM;

vue 和 react 都是采用 diff 算法来对比新旧虚拟节点，从而更新节点。在 vue 的 diff 函数中，会根据新节点的 key 去对比旧节点数组中的 key，从

而找到相应旧节点。如果没找到就认为是一个新增节点。而如果没有 key，那么就会采用遍历查找的方式去找到对应的旧节点。一种一个 map 映射，另一种是遍历查找。相比而言。map 映射的速度更快。

为了在数据变化时强制更新组件，以避免"就地复用"带来的副作用。

当 Vue.js 用 v-for 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。重复的 key 会造成渲染错误。

vue 组件的通信方式

props/$emit 父子组件通信

父->子 props，子->父 $on、$emit 获取父子组件实例 parent、children Ref 获取实例的方式调用组件的属性或者方法 父->子孙 Provide、inject 官方不推荐使用，但是写组件库时很常用

$emit/$on 自定义事件 兄弟组件通信

Event Bus 实现跨组件通信 Vue.prototype.$bus = new Vue() 自定义事件

vuex 跨级组件通信

Vuex、$attrs、$listeners Provide、inject

nextTick 的实现

1. nextTick 是 Vue 提供的一个全局 API,是在下次 DOM 更新循环结束之后执行延迟回调，在修改数据之后使用$nextTick，则可以在回调中获取更新后的 DOM;
2. Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，Vue 将开启 1 个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中-次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。nextTick 方法会在队列中加入一个回调函数，确保该函数在前面的 dom 操作完成后才调用;
3. 比如，我在干什么的时候就会使用 nextTick，传一个回调函数进去，在里面执行 dom 操作即可;
4. 我也有简单了解 nextTick 实现，它会在 callbacks 里面加入我们传入的函数，然后用 timerFunc 异步方式调用它们，首选的异步方式会是 Promise。这让我明白了为什么可以在 nextTick 中看到 dom 操作结果。

nextTick 的实现原理是什么？

在下次 DOM 更新循环结束之后执行延迟回调，在修改数据之后立即使用 nextTick 来获取更新后的 DOM。 nextTick 主要使用了宏任务和微任务。 根据执行环境分别尝试采用 Promise、MutationObserver、setImmediate，如果以上都不行则采用 setTimeout 定义了一个异步方法，多次调用 nextTick 会将方法存入队列中，通过这个异步方法清空当前队列。

使用过插槽么？用的是具名插槽还是匿名插槽或作用域插槽

vue 中的插槽是一个非常好用的东西 slot 说白了就是一个占位的 在 vue 当中插槽包含三种一种是默认插槽(匿名)一种是具名插槽还有一种就是作用域插槽 匿名插槽就是没有名字的只要默认的都填到这里具名插槽指的是具有名字的

keep-alive 的实现

作用：实现组件缓存，保持这些组件的状态，以避免反复渲染导致的性能问题。需要缓存组件 频繁切换，不需要重复渲染

场景：tabs 标签页 后台导航，vue 性能优化

原理：Vue.js 内部将 DOM 节点抽象成了一个个的 VNode 节点，keep-alive 组件的缓存也是基于 VNode 节点的而不是直接存储 DOM 结构。它将满足条件（pruneCache 与 pruneCache）的组件在 cache 对象中缓存起来，在需要重新渲染的时候再将 vnode 节点从 cache 对象中取出并渲染。

mixin

mixin 项目变得复杂的时候，多个组件间有重复的逻辑就会用到 mixin
多个组件有相同的逻辑，抽离出来
mixin 并不是完美的解决方案，会有一些问题
vue3 提出的 Composition API 旨在解决这些问题【追求完美是要消耗一定的成本的，如开发成本】
场景：PC 端新闻列表和详情页一样的右侧栏目，可以使用 mixin 进行混合
劣势：1.变量来源不明确，不利于阅读
2.多 mixin 可能会造成命名冲突 3.mixin 和组件可能出现多对多的关系，使得项目复杂度变高

Vuex 的理解及使用场景

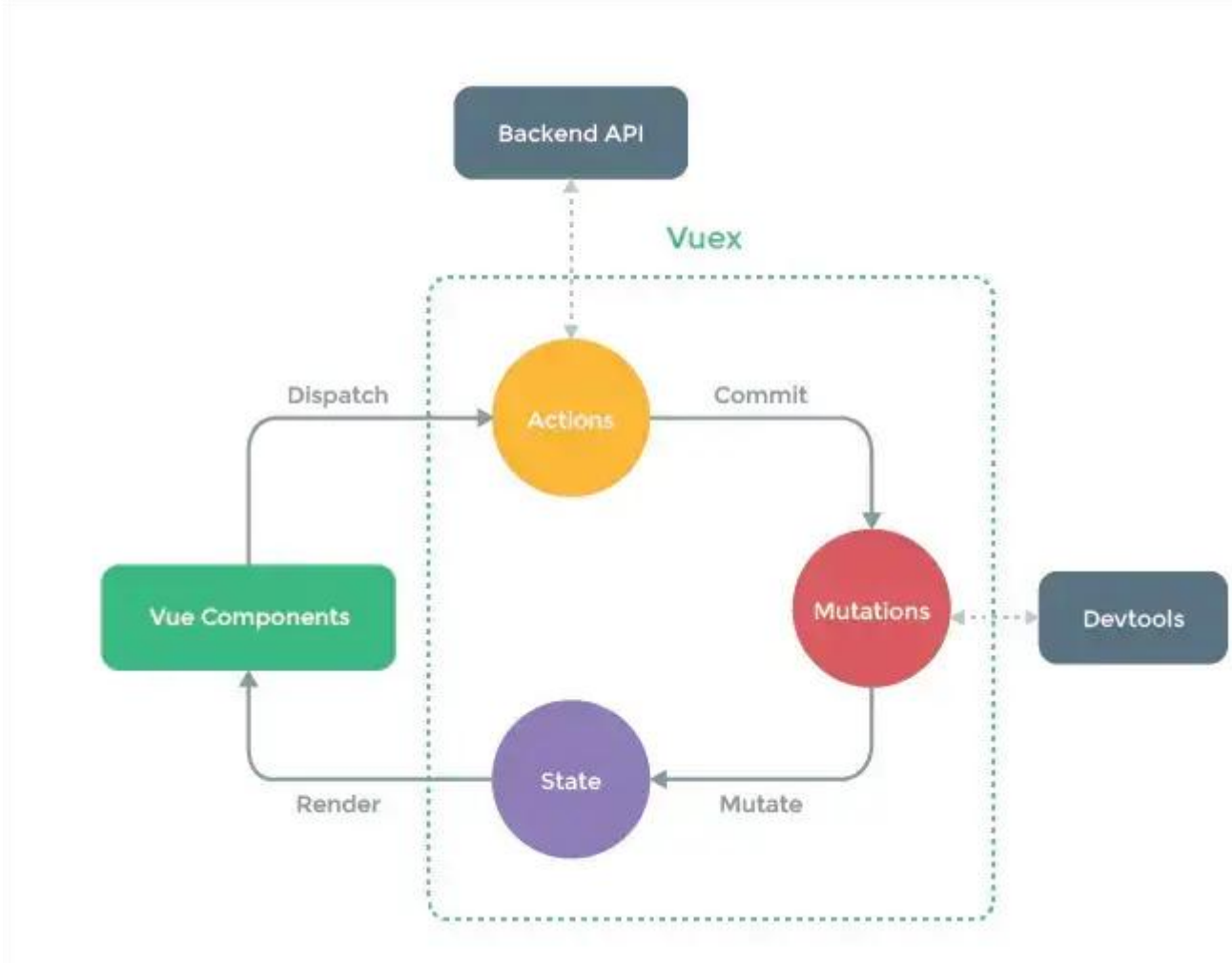
Vuex 是一个专为 Vue 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。

1. Vuex 的状态存储是响应式的；当 Vue 组件从 store 中读取状态的时候，

若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新 2. 改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation， 这样使得我们可以方便地跟踪每一个状态的变化 Vuex 主要包括以下几个核心模块：

1. State：定义了应用的状态数据
2. Getter：在 store 中定义“getter”（可以认为是 store 的计算属性），

就像计算属性一样，getter 的返回值会根据它的依赖被缓存起来， 且只有当它的依赖值发生了改变才会被重新计算 3. Mutation：是唯一更改 store 中状态的方法，且必须是同步函数 4. Action：用于提交 mutation，而不是直接变更状态，可以包含任意异步操作 5. Module：允许将单一的 Store 拆分为多个 store 且同时保存在单一的状态树中



hooks 用过吗？聊聊 react 中 class 组件和函数组件的区别

类组件是使用 ES6 的 class 来定义的组件。 函数组件是接收一个单一的 `props` 对象并返回一个 React 元素。

关于 React 的两套 API（类（class）API 和基于函数的钩子（hooks） API）。官方推荐使用钩子（函数），而不是类。因为钩子更简洁，代码量少，用起来比较"轻"，而类比较"重"。而且，钩子是函数，更符合 React 函数式的本质。

函数一般来说，只应该做一件事，就是返回一个值。 如果你有多个操作，每个操作应该写成一个单独的函数。而且，数据的状态应该与操作方法分离。根据函数这种理念，React 的函数组件只应该做一件事情：返回组件的 HTML 代码，而没有其他的功能。函数的返回结果只依赖于它的参数。不改变函数体外部数据、函数执行过程里面没有副作用。

类（class）是数据和逻辑的封装。 也就是说，组件的状态和操作方法是封装在一起的。如果选择了类的写法，就应该把相关的数据和操作，都写在同一个 class 里面。

类组件的缺点：

大型组件很难拆分和重构，也很难测试。
业务逻辑分散在组件的各个方法之中，导致重复逻辑或关联逻辑。
组件类引入了复杂的编程模式，比如 render props 和高阶组件。
难以理解的 class，理解 JavaScript 中 `this` 的工作方式。

区别：

函数组件的性能比类组件的性能要高，因为类组件使用的时候要实例化，而函数组件直接执行函数取返回结果即可。

1.状态的有无
hooks 出现之前，函数组件没有实例，没有生命周期，没有 `state`，没有 `this`，所以我们称函数组件为无状态组件。 hooks 出现之前，react 中的函数组件通常只考虑负责 UI 的渲染，没有自身的状态没有业务逻辑代码，是一个纯函数。它的输出只由参数 props 决定，不受其他任何因素影响。

2.调用方式的不同
函数组件重新渲染，将重新调用组件方法返回新的 react 元素。类组件重新渲染将 new 一个新的组件实例，然后调用 render 类方法返回 react 元素，这也说明为什么类组件中 this 是可变的。

3.因为调用方式不同，在函数组件使用中会出现问题
在操作中改变状态值，类组件可以获取最新的状态值，而函数组件则会按照顺序返回状态值

React Hooks（钩子的作用）

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

React Hooks 的几个常用钩子:

1. `useState()` //状态钩子
2. `useContext()` //共享状态钩子
3. `useReducer()` //action 钩子
4. `useEffect()` //副作用钩子

还有几个不常见的大概的说下，后续会专门写篇文章描述下

1.useCallback 记忆函数 一般把**函数式组件理解为 class 组件 render 函数的语法糖**，所以每次重新渲染的时候，函数式组件内部所有的代码都会重

新执行一遍。而有了 useCallback 就不一样了，你可以通过 useCallback 获得一个记忆后的函数。

```
function App() {
  const memoizedHandleClick = useCallback(() => {
    console.log('Click happened')
  }, []); // 空数组代表无论什么情况下该函数都不会发生改变
  return <SomeComponent onClick={memoizedHandleClick}>Click
Me</SomeComponent>;
}复制代码
```

第二个参数传入一个数组，数组中的每一项一旦值或者引用发生改变，useCallback 就会重新返回一个新的记忆函数提供给后面进行渲染。

2.useMemo 记忆组件 useCallback 的功能完全可以由 useMemo 所取代，如果你想通过使用 useMemo 返回一个记忆函数也是完全可以的。 唯一的区别是：**useCallback 不会执行第一个参数函数，而是将它返回给你，而 useMemo 会执行第一个函数并且将函数执行结果返回给你**。
所以 useCallback 常用记忆事件函数，生成记忆后的事件函数并传递给子组件使用。而 useMemo 更适合经过函数计算得到一个确定的值，比如记忆组件。

3.useRef 保存引用值

useRef 跟 createRef 类似，都可以用来生成对 DOM 对象的引用。useRef 返回的值传递给组件或者 DOM 的 ref 属性，就可以通过 ref.current 值**访问组件或真实的 DOM 节点，重点是组件也是可以访问到的**，从而可以对 DOM 进行一些操作，比如监听事件等等。

4.useImperativeHandle 穿透 Ref

通过 useImperativeHandle 用于让父组件获取子组件内的索引

5.useLayoutEffect 同步执行副作用

大部分情况下，使用 useEffect 就可以帮我们处理组件的副作用，但是如果想要同步调用一些副作用，比如对 DOM 的操作，就需要使用 useLayoutEffect，useLayoutEffect 中的副作用会在 DOM 更新之后同步执行。

useEffect 和 useLayoutEffect 有什么区别：简单来说就是调用时机不同，useLayoutEffect 和原来 componentDidMount&componentDidUpdate 一致，在 react 完成 DOM 更新后马上同步调用的代码，会阻塞页面渲染。而 useEffect 是会在整个页面渲染完才会调用的代码。官方建议优先使用 `useEffect`

React 组件通信方式

react 组件间通信常见的几种情况:

1. 父组件向子组件通信

1. 子组件向父组件通信

1. 跨级组件通信

1. 非嵌套关系的组件通信

1）父组件向子组件通信

父组件通过 props 向子组件传递需要的信息。父传子是在父组件中直接绑定一个正常的属性，这个属性就是指具体的值，在子组件中，用 props 就可以获取到这个值

```
// 子组件: Childconst Child = props =>{
  return <p>{props.name}</p>
}
// 父组件 Parentconst Parent = ()=>{
    return <Child name="京程一灯"></Child>
}复制代码
```

2）子组件向父组件通信

props+回调的方式，使用公共组件进行状态提升。子传父是先在父组件上绑定属性设置为一个函数，当子组件需要给父组件传值的时候，则通过 props 调用该函数将参数传入到该函数当中，此时就可以在父组件中的函数中接收到该参数了，这个参数则为子组件传过来的值

```
// 子组件: Childconst Child = props =>{
  const cb = msg =>{
      return ()=>{
          props.callback(msg)
      }
  }
  return (
      <button onClick={cb("京程一灯欢迎你!")}>京程一灯欢迎你</button>
  )
}
// 父组件 Parentclass Parent extends Component {
    callback(msg){
        console.log(msg)
    }
    render(){
        return <Child callback={this.callback.bind(this)}></Child>
    }
}复制代码
```

3）跨级组件通信

即父组件向子组件的子组件通信，向更深层子组件通信。

> 使用 props，利用中间组件层层传递,但是如果父组件结构较深，那么中间每一层组件都要去传递 props，增加了复杂度，并且这些 props 并不是中间组件自己需要的。

使用 context，context 相当于一个大容器，我们可以把要通信的内容放在这个容器中，这样不管嵌套多深，都可以随意取用，对于跨越多层的全局数据可以使用 context 实现。

```
// context 方式实现跨级组件通信 // Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据
const BatteryContext = createContext();
//  子组件的子组件 class GrandChild extends Component {
    render(){
       return (
          <BatteryContext.Consumer>
                {
                    color => <h1 style={{"color":color}}>我是红色的:{color}</h1>
                }
            </BatteryContext.Consumer>
       )
    }
}
//  子组件 const Child = () =>{
    return (
       <GrandChild/>
    )
}// 父组件 class Parent extends Component {
     state = {
        color:"red"
     }
     render(){
        const {color} = this.state
        return (
        <BatteryContext.Provider value={color}>
              <Child></Child>
          </BatteryContext.Provider>
        )
     }
}复制代码
```

4）非嵌套关系的组件通信

即没有任何包含关系的组件，包括兄弟组件以及不在同一个父级中的非兄弟组件。

1. 可以使用自定义事件通信（发布订阅模式），使用 pubsub-js

1. 可以通过 redux 等进行全局状态管理

1. 如果是兄弟组件通信，可以找到这两个兄弟节点共同的父节点, 结合父子间通信方式进行通信。

1. 也可以 new 一个 Vue 的 EventBus,进行事件监听，一边执行监听，一边执行新增 VUE 的 eventBus 就是发布订阅模式，是可以在 React 中使用的;

setState 既存在异步情况也存在同步情况

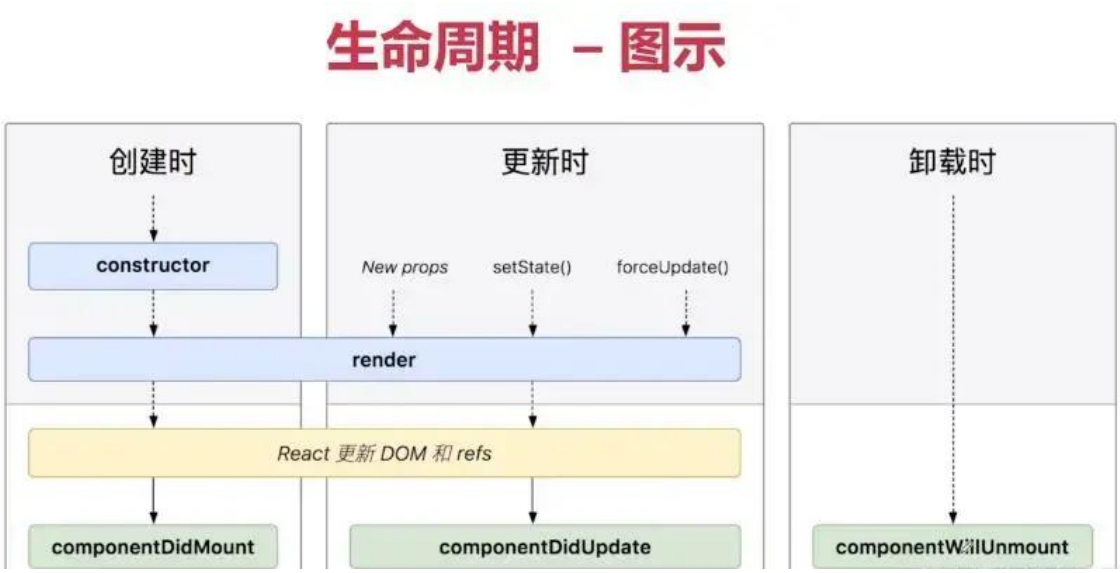
1.异步情况 在 React 事件当中是异步操作

2.同步情况 如果是在 setTimeout 事件或者自定义的 dom 事件中，都是同步的

```
//setTimeout 事件 import React,{ Component } from "react";class Count extends Component{
    constructor(props){
        super(props);
        this.state = {
            count:0
        }
    }

    render(){
        return (
            <>
                    <p>count:{this.state.count}</p>
                    <button onClick={this.btnAction}>增加</button>
              </>
        )
    }

    btnAction = ()=>{
        //不能直接修改 state，需要通过 setState 进行修改
        //同步
        setTimeout(()=>{
            this.setState({
                count: this.state.count + 1
```

```
            });
            console.log(this.state.count);
        })
    }
}
export default Count;复制代码
//自定义 dom 事件 import React,{ Component } from "react";class Count
extends Component{
    constructor(props){
        super(props);
        this.state = {
            count:0
        }
    }

    render(){
        return (
            <>
                  <p>count:{this.state.count}</p>
                  <button id="btn">绑定点击事件</button>
              </>
        )
    }

    componentDidMount(){
        //自定义 dom 事件，也是同步修改
        document.querySelector('#btn').addEventListener('click',()=>{
            this.setState({
                count: this.state.count + 1
            });
            console.log(this.state.count);
        });
    }
}
export default Count;复制代码
```

生命周期



安装

当组件的实例被创建并插入到 DOM 中时，这些方法按以下顺序调用：

constructor()static getDerivedStateFromProps()render()componentDidMount()

更新中

更新可能由道具或状态的更改引起。当重新渲染组件时，这些方法按以下顺序调用：

static getDerivedStateFromProps()shouldComponentUpdate()render()getSnapshotBeforeUpdate()componentDidUpdate()

卸载

当组件从 DOM 中移除时调用此方法：

componentWillUnmount()复制代码

说一下 react-fiber

1）背景

react-fiber 产生的根本原因，是大量的同步计算任务阻塞了浏览器的 UI 渲染。默认情况下，JS 运算、页面布局和页面绘制都是运行在浏览器的主线程当中，他们之间是互斥的关系。如果 JS 运算持续占用主线程，页面就没法得到及时的更新。当我们调用 setState 更新页面的时候，React 会遍历应用的所有节点，计算出差异，然后再更新 UI。如果页面元素很多，整个过程占用的时机就可能超过 16 毫秒，就容易出现掉帧的现象。

2）实现原理

react 内部运转分三层：

- Virtual DOM 层，描述页面长什么样。
- Reconciler 层，负责调用组件生命周期方法，进行 Diff 运算等。
- Renderer 层，根据不同的平台，渲染出相应的页面，比较常见的是 ReactDOM 和 ReactNative。

Fiber 其实指的是一种数据结构，它可以用一个纯 JS 对象来表示：

```
const fiber = {
    stateNode,    // 节点实例
    child,        // 子节点
    sibling,      // 兄弟节点
    return,       // 父节点
}复制代码
```

为了实现不卡顿，就需要有一个调度器 (Scheduler) 来进行任务分配。优先级高的任务（如键盘输入）可以打断优先级低的任务（如 Diff）的执行，从而更快的生效。任务的优先级有六种：

- synchronous，与之前的 Stack Reconciler 操作一样，同步执行
- task，在 next tick 之前执行
- animation，下一帧之前执行
- high，在不久的将来立即执行
- low，稍微延迟执行也没关系
- offscreen，下一次 render 时或 scroll 时才执行

Fiber Reconciler（react ）执行过程分为 2 个阶段：

- 阶段一，生成 Fiber 树，得出需要更新的节点信息。这一步是一个渐进的过程，可以被打断。阶段一可被打断的特性，让优先级更高的任务先执行，从框架层面大大降低了页面掉帧的概率。
- 阶段二，将需要更新的节点一次过批量更新，这个过程不能被打断。

Fiber 树：React 在 render 第一次渲染时，会通过 React.createElement 创建一颗 Element 树，可以称之为 Virtual DOM Tree，由于要记录上下文信息，加入了 Fiber，每一个 Element 会对应一个 Fiber Node，将 Fiber Node 链接起来的结构成为 Fiber Tree。Fiber Tree 一个重要的特点是链表结构，将递归遍历编程循环遍历，然后配合 requestIdleCallback API, 实现任务拆分、中断与恢复。

从 Stack Reconciler 到 Fiber Reconciler，源码层面其实就是干了一件递归改循环的事情

传送门 ☞# 深入了解 Fiber

Portals

Portals 提供了一种一流的方式来将子组件渲染到存在于父组件的 DOM 层次结构之外的 DOM 节点中。结构不受外界的控制的情况下就可以使用 portals 进行创建

何时要使用异步组件？如和使用异步组件

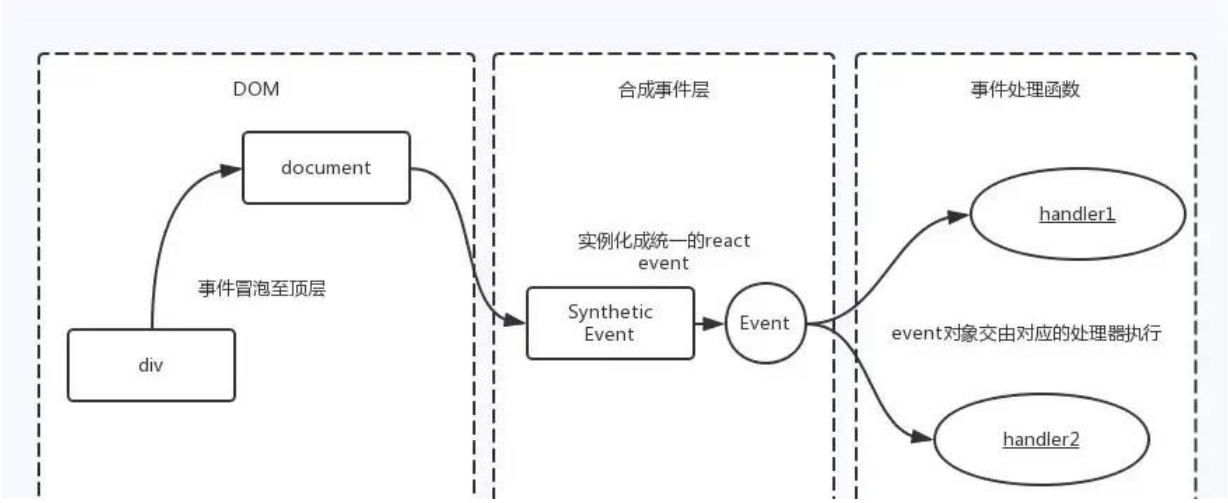
加载大组件的时候
路由异步加载的时候

react 中要配合 Suspense 使用

// 异步懒加载 const Box = lazy(()=>import('./components/Box'));// 使用组件的时候要用 suspense 进行包裹<Suspense fallback={<div>loading...</div>}>
{show && <Box/>}</Suspense>复制代码

React 事件绑定原理

React 并不是将 click 事件绑在该 div 的真实 DOM 上，而是在 document 处监听所有支持的事件，当事件发生并冒泡至 document 处时，React 将事件内容封装并交由真正的处理函数运行。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。
另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 event.stopPropagation 是无效的，而应该调用 event.preventDefault。



webpack

webpack 做过哪些优化，开发效率方面、打包策略方面等等

1）优化 Webpack 的构建速度

使用高版本的 Webpack （使用 webpack4）
多线程/多实例构建：HappyPack(不维护了)、thread-loader
缩小打包作用域：

- exclude/include (确定 loader 规则范围)
- resolve.modules 指明第三方模块的绝对路径 (减少不必要的查找)
- resolve.extensions 尽可能减少后缀尝试的可能性
- noParse 对完全不需要解析的库进行忽略 (不去解析但仍会打包到 bundle 中，注意被忽略掉的文件里不应该包含 import、require、define 等模块化语句)
- IgnorePlugin (完全排除模块)
- 合理使用 alias

充分利用缓存提升二次构建速度：

- babel-loader 开启缓存
- terser-webpack-plugin 开启缓存
- 使用 cache-loader 或者 hard-source-webpack-plugin
  注意：thread-loader 和 cache-loader 兩個要一起使用的話，請先放 cache-loader 接著是 thread-loader 最後才是 heavy-loader

DLL：

- 使用 DllPlugin 进行分包，使用 DllReferencePlugin(索引链接) 对 manifest.json 引用，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间。

2）使用 webpack4-优化原因

(a)V8 带来的优化(for of 替代 forEach、Map 和 Set 替代 Object、includes 替代 indexOf)
(b)默认使用更快的 md4 hash 算法
(c)webpacks AST 可以直接从 loader 传递给 AST，减少解析时间
(d)使用字符串方法替代正则表达式

①noParse

不去解析某个库内部的依赖关系
比如 jquery 这个库是独立的， 则不去解析这个库内部依赖的其他的东西
在独立库的时候可以使用

```
module.exports = {
  module: {
    noParse: /jquery/,
    rules:[]
  }
}复制代码
```

②IgnorePlugin

忽略掉某些内容 不去解析依赖库内部引用的某些内容
从 moment 中引用 ./locol 则忽略掉
如果要用 local 的话 则必须在项目中必须手动引入

```
import 'moment/locale/zh-cn'module.exports = {
    plugins: [
        new Webpack.IgnorePlugin(/./local/, /moment/),
    ]
}复制代码
```

③dillPlugin

不会多次打包， 优化打包时间
先把依赖的不变的库打包
生成 manifest.json 文件
然后在 webpack.config 中引入
webpack.DllPlugin Webpack.DllReferencePlugin

④happypack -> thread-loader

大项目的时候开启多线程打包
影响前端发布速度的有两个方面，一个是构建，一个就是压缩，把这两个东西优化起来，可以减少很多发布的时间。

⑤thread-loader
thread-loader 会将您的 loader 放置在一个 worker 池里面运行，以达到多线程构建。
把这个 loader 放置在其他 loader 之前（如下图 example 的位置）， 放置在这个 loader 之后的 loader 就会在一个单独的 worker 池(worker pool)中运行。

```
// webpack.config.jsmodule.exports = {
  module: {
    rules: [
      {
        test: /.js$/,
        include: path.resolve("src"),
        use: [
          "thread-loader",
          // 你的高开销的 loader 放置在此 (e.g babel-loader)
        ]
      }
    ]
  }
}复制代码
```

每个 worker 都是一个单独的有 600ms 限制的 node.js 进程。同时跨进程的数据交换也会被限制。请在高开销的 loader 中使用，否则效果不佳

⑥压缩加速——开启多线程压缩

不推荐使用 webpack-paralle-uglify-plugin，项目基本处于没人维护的阶段，issue 没人处理，pr 没人合并。
Webpack 4.0 以前：uglifyjs-webpack-plugin，parallel 参数

```
module.exports = {
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        parallel: true,
      }),
    ],
  },};复制代码
```

推荐使用 terser-webpack-plugin

```
module.exports = {
  optimization: {
    minimizer: [new TerserPlugin(
      parallel: true   // 多线程
    )],
  },
};复制代码
```

2）优化 Webpack 的打包体积

压缩代码
提取页面公共资源：
Tree shaking
Scope hoisting
图片压缩
动态 Polyfill

3）speed-measure-webpack-plugin

简称 SMP，分析出 Webpack 打包过程中 Loader 和 Plugin 的耗时，有助于找到构建过程中的性能瓶颈。 **开发阶段**

开启多核压缩 插件：** terser-webpack-plugin **

```
const TerserPlugin = require('terser-webpack-plugin')module.exports = {
    optimization: {
        minimizer: [
            new TerserPlugin({
                parallel: true,
                terserOptions: {
                    ecma: 6,
                },
            }),
        ]
    }
};复制代码
```

Babel

简单描述一下 Babel 的编译过程

Babel 是一个 JavaScript 编译器，是一个工具链，主要用于将采用 ECMAScript 2015+ 语法编写的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中。

Babel 本质上就是在操作 AST 来完成代码的转译。AST 是抽象语法树（Abstract Syntax Tree, AST）

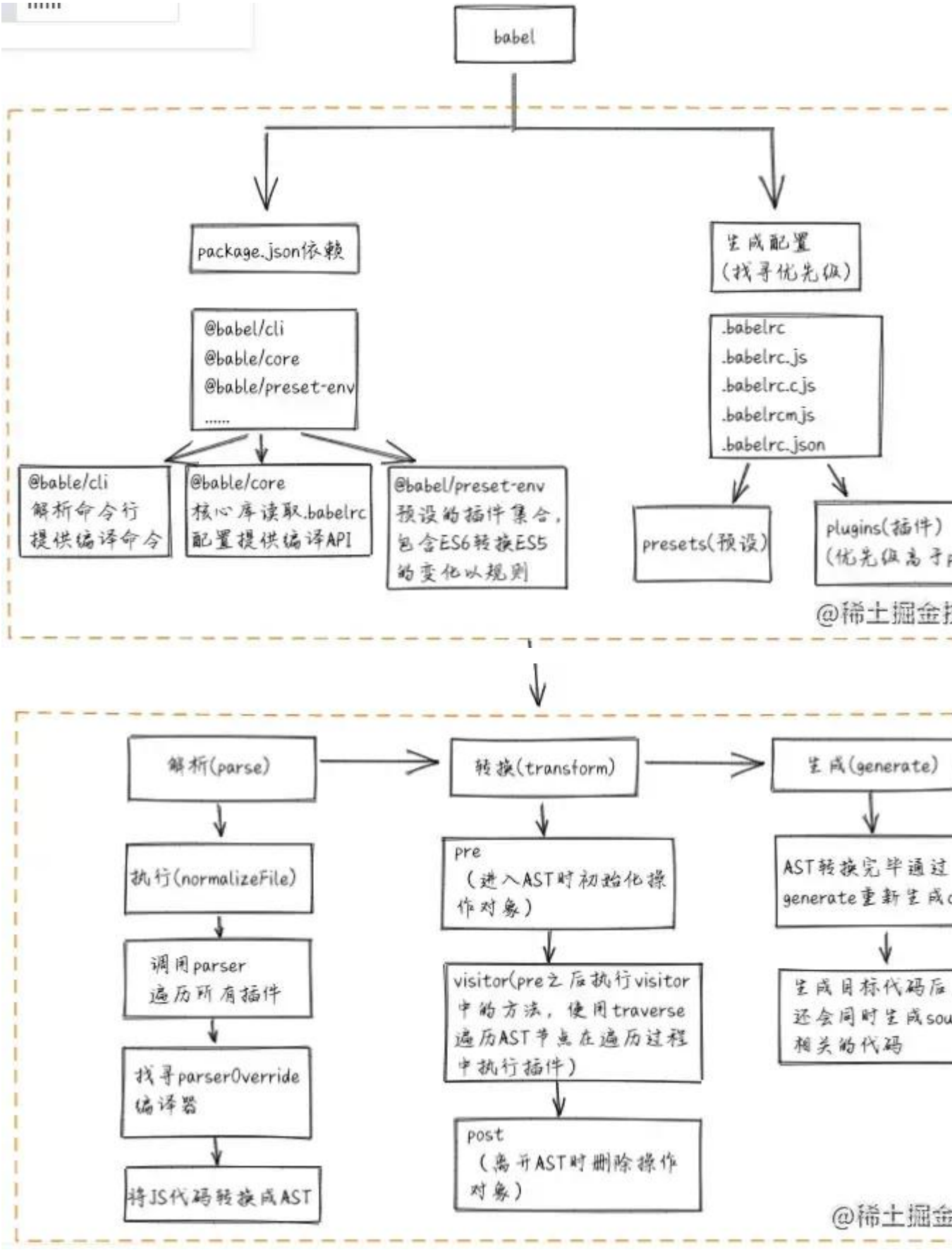
如果想要了解更多，可以阅读和尝试：

> 分析 AST：ASTexplorer.net
> AST 规范：github.com/estree/estr...

Babel 的功能很纯粹，它只是一个编译器。大多数编译器的工作过程可以分为三部分：

1. **解析（Parse）**：将源代码转换成更加抽象的表示方法（例如抽象语法树）。包括词法分析和语法分析。词法分析主要把字符流源代码（Char Stream）转换成令牌流（ Token Stream），语法分析主要是将令牌流转换成抽象语法树（Abstract Syntax Tree，AST）。
2. **转换（Transform）**：通过 Babel 的插件能力，对（抽象语法树）做一些特殊处理，将高版本语法的 AST 转换成支持低版本语法的 AST。让它符合编译器的期望，当然在此过程中也可以对 AST 的 Node 节点进行优化操作，比如添加、更新以及移除节点等。
3. **生成（Generate）**：将 AST 转换成字符串形式的低版本代码，同时也能创建 Source Map 映射。

经过这三个阶段，代码就被 Babel 转译成功了。

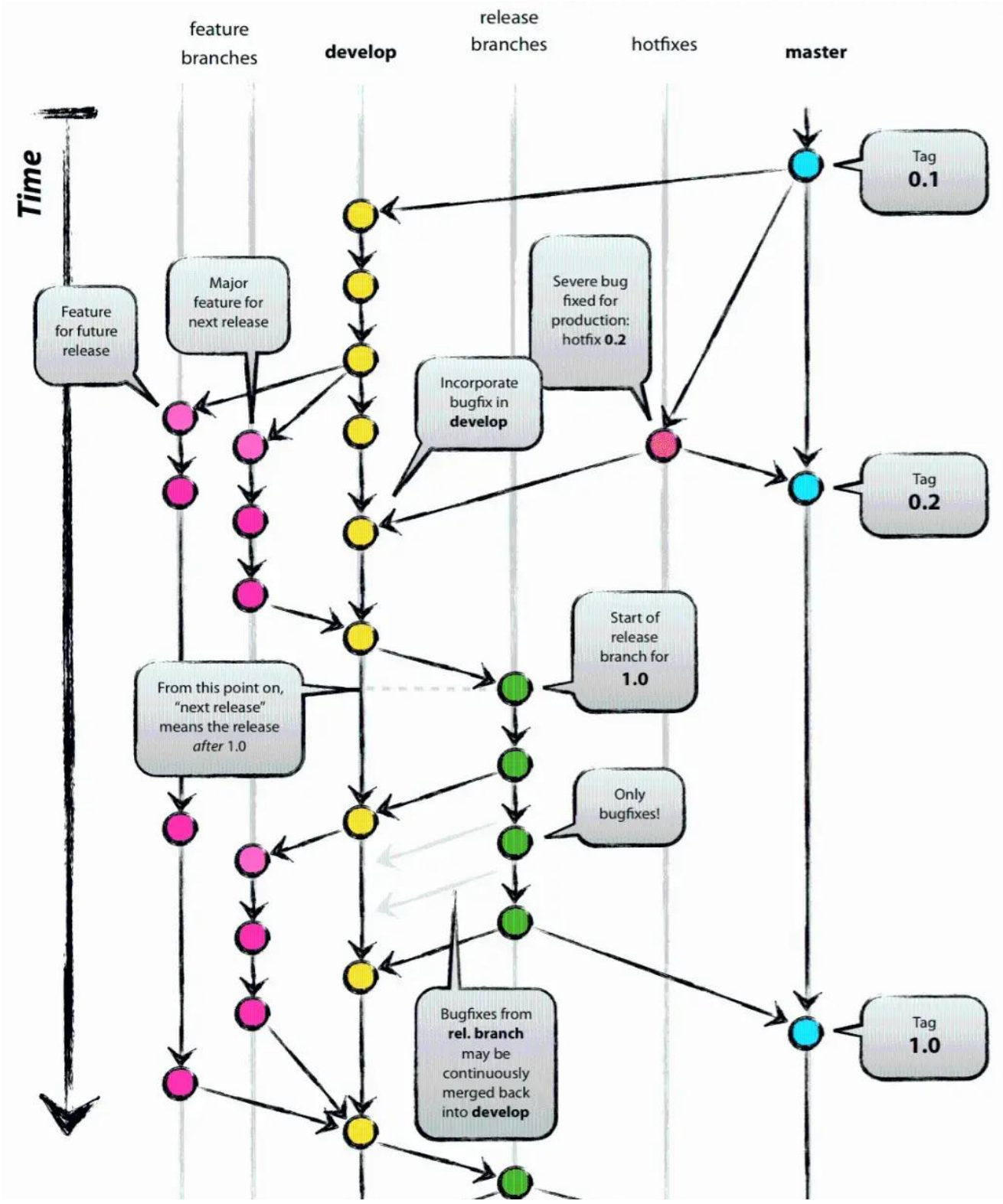

babel
package.json依赖
@babel/cli
@bable/core
@bable/preset-env
.....
@bable/cli
解析命令行
提供编译命令
@bable/core
核心库读取.babelrc
配置提供编译API
@babel/preset-env
预设的插件集合，
包含ES6转换ES5
的变化以规则
生成配置
(找寻优先级)
.babelrc
.babelrc.js
.babelrc.cjs
.babelrcmjs
.babelrc.json
presets(预设)
plugins(插件)
@稀土掘金
解析(parse)
执行(normalizeFile)
调用parser
遍历所有插件
找寻parserOverride
编译器
将JS代码转换成AST
转换(transform)
pre
(进入AST时初始化操
作对象)
visitor(pre之后执行visitor
中的方法，使用traverse
遍历AST节点在遍历过程
中执行插件)
post
(离开AST时删除操作
对象)
生成(generate)
AST转换完毕通过
generate重新生成
生成目标代码后
还会同时生成sou
相关的代码
@稀土掘金

Git

Git 常用命令

查看分支：git branch
创建分支：git branch
切换分支：git checkout
创建+切换分支：git checkout -b
合并某分支到当前分支：git merge
删除分支：git branch -d

如何使用 Git 管理项目



实际开发中，一个仓库（一般只放一个项目）主要存在两条主分支：master 与 develop 分支。这个两个分支的生命周期是整个项目周期。

我们可能使用的不同类型的分支对项目进行管理是：

功能分支 功能分支（或有时称为主题分支）用于为即将发布或遥远的未来版本开发新功能。在开始开发某个功能时，将包含该功能的目标版本在那时很可能是未知的。功能分支的本质在于，只要该功能处于开发阶段，它就存在，但最终会被合并回 develop（明确将新功能添加到即将发布的版本中）或丢弃。功能分支通常只存在于开发者仓库中，而不存在于 origin。

发布分支 发布分支支持准备新的生产版本。它们允许在最后一刻打点 i 和交叉 t。此外，它们允许修复小错误并为发布准备元数据（版本号、构建日期等）。通过在发布分支上完成所有这些工作，该 develop 分支被清除以接收下一个大版本的功能。

- 从 develop 分支拉取，且必须合并回 develop 和 master
- 分支命名约定：release-*

修补程序分支

Hotfix 分支与发布分支非常相似，因为它们也旨在为新的生产版本做准备，尽管是计划外的。它们产生于需要立即对现场制作版本的不良状态采取行动。当必须立即解决生产版本中的关键错误时，可以从标记生产版本的主分支上的相应标记中分支出一个修补程序分支。

master：这个分支最为稳定，这个分支表明项目处于可发布的状态。
develop：做为开发的分支，平行于 master 分支。
Feature branches：这种分支和咱们程序员平常开发最为密切，称做功能分支。必须从 develop 分支建立，完成后合并回 develop 分支。
Release branches：这个分支用来分布新版本。从 develop 分支建立，完成后合并回 develop 与 master 分支。这个分支上能够作一些很是小的 bug 修复，固然，你也能够禁止在这个分支作任何 bug 的修复工做，而只作版本发布的相关操做，例如设置版本号等操做，那样的话那些发现的小 bug 就必须放到下一个版本修复了。若是在这个分支上发现了大 bug，那么也绝对不能在这个分支上改，须要 Featrue 分支上改，走正常的流程。
Hotfix branches：这个分支主要为修复线上特别紧急的 bug 准备的。必须从 master 分支建立，完成后合并回 develop 与 master 分支。这个分支主要是解决线上版本

的紧急 bug 修复的，例如忽然版本 V0.1 上有一个致命 bug，必须修复。那么咱们就能够从 master 分支上发布这个版本那个时间点 例如 tag v0.1（通常代码发布后会及时在 master 上打 tag），来建立一个 hotfix-v0.1.1 的分支，而后在这个分支上改 bug，而后发布新的版本。最后将代码合并回 develop 与 master 分支。

项目优化

移除生产环境的控制台打印。方案很多，esling+pre-commit、使用插件自动去除，插件包括 babel-plugin-transform-remove-console、uglifyjs-webpack-plugin、terser-webpack-plugin。最后选择了 terser-webpack-plugin，脚手架 vue-cli 用这个插件来开启缓存和多线程打包，无需安装额外的插件，仅需在 configureWebpack 中设置 terser 插件的 drop_console 为 true 即可。最好还是养成良好的代码习惯，在开发基本完成后去掉无用的 console，vscode 中的 turbo console 就蛮好的。

第三方库的按需加载。echarts，官方文档里是使用配置文件指定使用的模块，另一种使用 babel-plugin-equire 实现按需加载。element-ui 使用 babel-plugin-component 实现按需引入。

前后端数据交换方面，推动项目组使用蓝湖、接口文档，与后端同学协商，规范后台数据返回。

雅虎军规提到的，避免 css 表达式、滤镜，较少 DOM 操作，优化图片、精灵图，避免图片空链接等。

性能问题：页面加载性能、动画性能、操作性能。Performance API，记录性能数据。

winter 重学前端 优化技术方案：

缓存：客户端控制的强缓存策略。

降低请求成本：DNS 由客户端控制，隔一段时间主动请求获取域名 IP，不走系统 DNS（完全看不懂）。TCP/TLS 连接复用，服务器升级到 HTTP2，尽量合并域名。

减少请求数：JS、CSS 打包到 HTML。JS 控制图片异步加载、懒加载。小型图片使用 data-uri。

较少传输体积：尽量使用 SVG\gradient 代替图片。根据机型和网络状况控制图片清晰度。对低清晰度图片使用锐化来提升体验。设计上避免大型背景图。

使用 CDN 加速，内容分发网络，是建立再承载网基础上的虚拟分布式网络，能够将源站内容缓存到全国或全球的节点服务器上。用户就近获取内容，提高了资源的访问速度，分担源站压力。