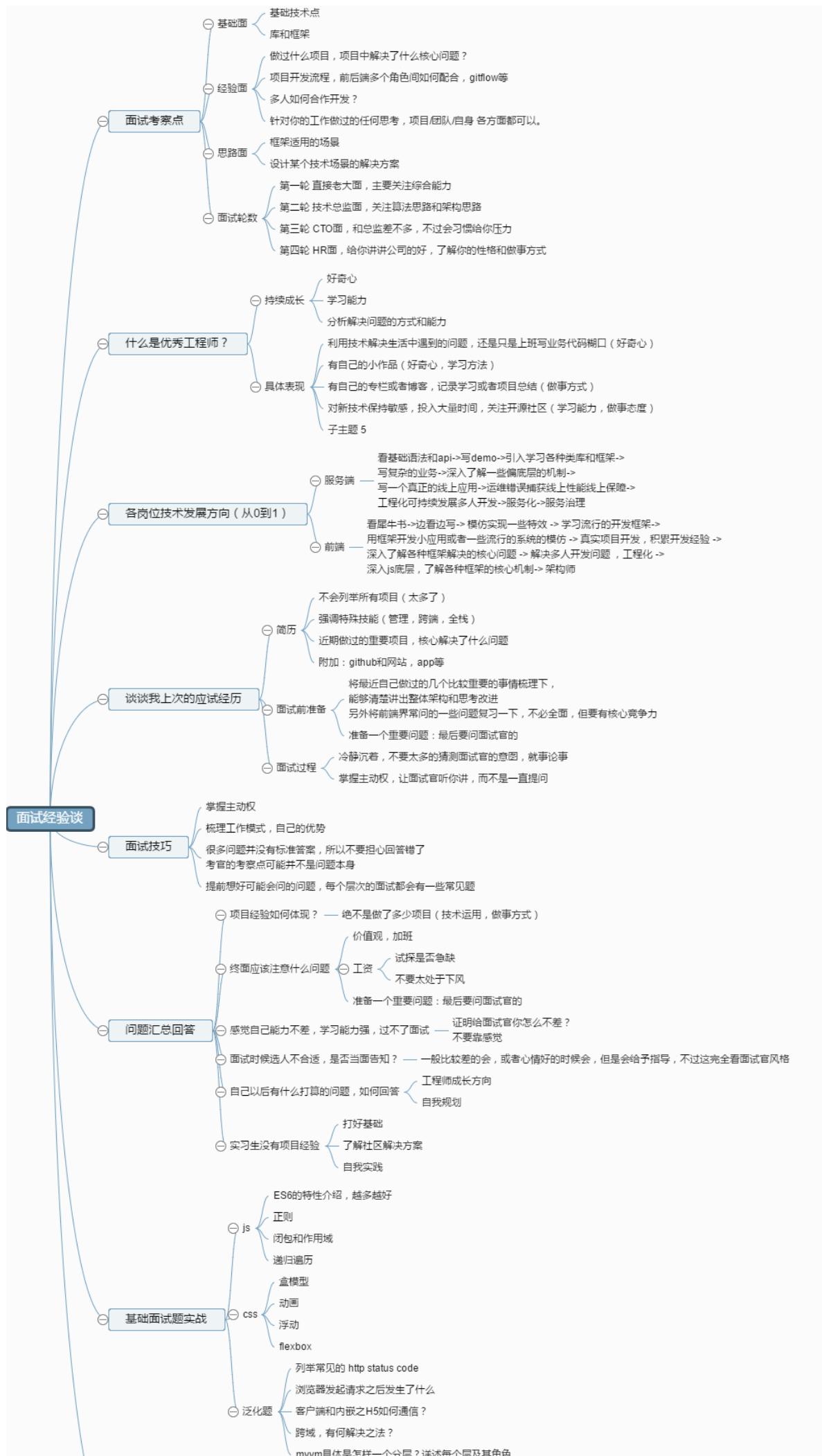
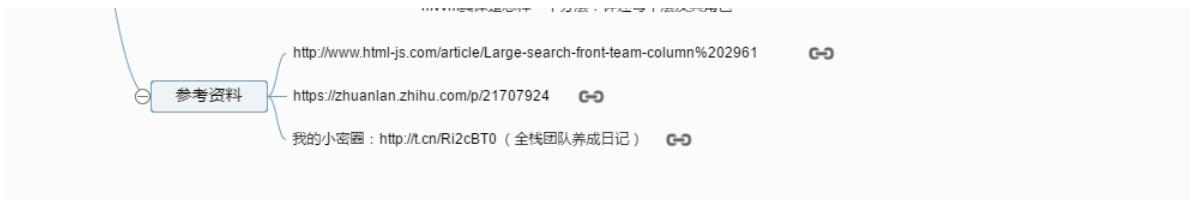


基础篇





#一、HTML、HTTP、web综合问题

#1 前端需要注意哪些SEO

- 合理的 title、description、keywords：搜索对着三项的权重逐个减小，title 值强调重点即可，重要关键词出现不要超过2次，而且要靠前，不同页面 title 要有所不同；description 把页面内容高度概括，长度合适，不可过分堆砌关键词，不同页面 description 有所不同；keywords 列举出重要关键词即可
- 语义化的 HTML 代码，符合W3C规范：语义化代码让搜索引擎容易理解网页
- 重要内容 HTML 代码放在最前：搜索引擎抓取 HTML 顺序是从上到下，有的搜索引擎对抓取长度有限制，保证重要内容一定会被抓取
- 重要内容不要用 js 输出：爬虫不会执行js获取内容
- 少用 iframe：搜索引擎不会抓取 iframe 中的内容
- 非装饰性图片必须加 alt
- 提高网站速度：网站速度是搜索引擎排序的一个重要指标

#2 的 title 和 alt 有什么区别

- 通常当鼠标滑动到元素上的时候显示
- alt 是 的特有属性，是图片内容的等价描述，用于图片无法加载时显示、读屏器阅读图片。可提高图片高可访问性，除了纯装饰图片外都必须设置有意义的值，搜索引擎会重点分析。

#3 HTTP的几种请求方法用途

- GET 方法
 - 发送一个请求来取得服务器上的某一资源
- POST 方法
 - 向 URL 指定的资源提交数据或附加新的数据
- PUT 方法
 - 跟 POST 方法很像，也是想服务器提交数据。但是，它们之间有不同。PUT 指定了资源在服务器上的位置，而 POST 没有
- HEAD 方法
 - 只请求页面的首部
- DELETE 方法
 - 删除服务器上的某资源
- OPTIONS 方法
 - 它用于获取当前 URL 所支持的方法。如果请求成功，会有一个 Allow 的头包含类似“GET, POST”这样的信息
- TRACE 方法
 - TRACE 方法被用于激发一个远程的，应用层的请求消息回路
- CONNECT 方法

- 把请求连接转换到透明的 TCP/IP 通道

#4 从浏览器地址栏输入url到显示页面的步骤

基础版本

- 浏览器根据请求的 URL 交给 DNS 域名解析，找到真实 IP，向服务器发起请求；
- 服务器交给后台处理完成后返回数据，浏览器接收文件（HTML、JS、CSS、图象等）；
- 浏览器对加载到的资源（HTML、JS、CSS 等）进行语法解析，建立相应的内部数据结构（如 HTML 的 DOM）；
- 载入解析到的资源文件，渲染页面，完成。

详细版

1. 在浏览器地址栏输入URL

2. 浏览器查看

缓存

，如果请求资源在缓存中并且新鲜，跳转到转码步骤

- 如果资源未缓存，发起新请求
- 如果已缓存，检验是否足够新鲜，足够新鲜直接提供给客户端，否则与服务器进行验证。
- 检验新鲜通常有两个HTTP头进行控制

Expires

和

Cache-Control

：

- HTTP1.0提供Expires，值为一个绝对时间表示缓存新鲜日期
- HTTP1.1增加了Cache-Control: max-age=，值为以秒为单位的最大新鲜时间

3. 浏览器解析URL获取协议，主机，端口，path

4. 浏览器组装一个HTTP（GET）请求报文

5. 浏览器

获取主机ip地址

，过程如下：

- 浏览器缓存
- 本机缓存
- hosts文件
- 路由器缓存
- ISP DNS缓存
- DNS递归查询（可能存在负载均衡导致每次IP不一样）

6. 打开一个socket与目标IP地址，端口建立TCP链接

，三次握手如下：

- 客户端发送一个TCP的SYN=1，Seq=X的包到服务器端口
- 服务器发回SYN=1，ACK=X+1，Seq=Y的响应包
- 客户端发送ACK=Y+1，Seq=Z

7. TCP链接建立后发送HTTP请求

8. 服务器接受请求并解析，将请求转发到服务程序，如虚拟主机使用HTTP Host头部判断请求的服务程序

9. 服务器检查HTTP请求头是否包含缓存验证信息如果验证缓存新鲜，返回304等对应状态码

10. 处理程序读取完整请求并准备HTTP响应，可能需要查询数据库等操作

11. 服务器将响应报文通过TCP连接发送回浏览器

12. 浏览器接收HTTP响应，然后根据情况选择

关闭TCP连接或者保留重用，关闭TCP连接的四次握手如下

:

1. 主动方发送Fin=1, Ack=Z, Seq=X报文

2. 被动方发送ACK=X+1, Seq=Z报文

3. 被动方发送Fin=1, ACK=X, Seq=Y报文

4. 主动方发送ACK=Y, Seq=X报文

13. 浏览器检查响应状态吗：是否为1XX, 3XX, 4XX, 5XX，这些情况处理与2XX不同

14. 如果资源可缓存，进行缓存

15. 对响应进行解码（例如gzip压缩）

16. 根据资源类型决定如何处理（假设资源为HTML文档）

17. 解析HTML文档，构件DOM树，下载资源，构造CSSOM树，执行js脚本，这些操作没有严格的先后顺序，以下分别解释

18. 构建DOM树

:

1. **Tokenizing**: 根据HTML规范将字符流解析为标记

2. **Lexing**: 词法分析将标记转换为对象并定义属性和规则

3. **DOM construction**: 根据HTML标记关系将对象组成DOM树

19. 解析过程中遇到图片、样式表、js文件，启动下载

20. 构建

CSSOM树

:

1. **Tokenizing**: 字符流转换为标记流

2. **Node**: 根据标记创建节点

3. **CSSOM**: 节点创建CSSOM树

21. 根据DOM树和CSSOM树构建渲染树

:

1. 从DOM树的根节点遍历所有可见节点，不可见节点包括：1) `script, meta` 这样本身不可见的标签。2)被css隐藏的节点，如 `display: none`

2. 对每一个可见节点，找到恰当的CSSOM规则并应用

3. 发布可视节点的内容和计算样式

22. js解析如下

:

1. 浏览器创建Document对象并解析HTML，将解析到的元素和文本节点添加到文档中，此时 `document.readyState` 为 `loading`

2. HTML解析器遇到没有async和defer的script时，将他们添加到文档中，然后执行行内或外部脚本。这些脚本会同步执行，并且在脚本下载和执行时解析器会暂停。这样就可以用 `document.write()` 把文本插入到输入流中。**同步脚本经常简单定义函数和注册事件处理程序，他们可以遍历和操作script和他们之前的文档内容**

3. 当解析器遇到设置了 `async` 属性的 script 时，开始下载脚本并继续解析文档。脚本会在它 **下载完成后尽快执行**，但是 **解析器不会停下来等它下载**。异步脚本 **禁止使用 `document.write()`**，它们可以访问自己 script 和之前的文档元素
 4. 当文档完成解析，`document.readyState` 变成 `interactive`
 5. 所有 `defer` 脚本会 **按照在文档出现的顺序执行**，延迟脚本 **能访问完整文档树**，禁止使用 `document.write()`
 6. 浏览器在 **Document 对象上触发 DOMContentLoaded 事件**
 7. 此时文档完全解析完成，浏览器可能还在等待如图片等内容加载，等这些 **内容完成载入并且所有异步脚本完成载入和执行**，`document.readyState` 变为 `complete`，`window` 触发 `load` 事件
23. **显示页面** (HTML 解析过程中会逐步显示页面)

详细简版

1. 从浏览器接收 `url` 到开启网络请求线程（这一部分可以展开浏览器的机制以及进程与线程之间的关系）
2. 开启网络线程到发出一个完整的 `HTTP` 请求（这一部分涉及到 dns 查询，`TCP/IP` 请求，五层因特网协议栈等知识）
3. 从服务器接收到请求到对应后台接收到请求（这一部分可能涉及到负载均衡，安全拦截以及后台内部的处理等等）
4. 后台和前台的 `HTTP` 交互（这一部分包括 `HTTP` 头部、响应码、报文结构、`cookie` 等知识，可以提下静态资源的 `cookie` 优化，以及编码解码，如 `gzip` 压缩等）
5. 单独拎出来的缓存问题，`HTTP` 的缓存（这部分包括 http 缓存头部，`ETag`，`Cache-control` 等）
6. 浏览器接收到 `HTTP` 数据包后的解析流程（解析 `html`-词法分析然后解析成 dom 树、解析 `css` 生成 `css` 规则树、合并成 `render` 树，然后 `layout`、`painting` 渲染、复合图层的合成、`GPU` 绘制、外链资源的处理、`loaded` 和 `DOMContentLoaded` 等）
7. `css` 的可视化格式模型（元素的渲染规则，如包含块，控制框，`BFC`，`IFC` 等概念）
8. `JS` 引擎解析过程（`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）
9. 其它（可以拓展不同的知识模块，如跨域，web 安全，`hybrid` 模式等等内容）

#5 如何进行网站性能优化

- `content` 方面
 - **减少 HTTP 请求** 合并文件、`css` 精灵、`inline Image`
 - 减少 DNS 查询：`DNS` 缓存、将资源分布到恰当数量的主机名
 - 减少 DOM 元素数量
- `Server` 方面
 - 使用 `CDN`
 - 配置 `ETag`
 - 对组件使用 `Gzip` 压缩
- `Cookie` 方面
 - 减小 `cookie` 大小
- `css` 方面
 - 将样式表放到页面顶部
 - 不使用 `css` 表达式
 - ✓ ◦ **使用 `<link>` 不使用 `@import`**
- `Javascript` 方面
 - ✓ ◦ **将脚本放到页面底部**
 - ✓ ◦ **将 javascript 和 css 从外部引入**
 - 压缩 `javascript` 和 `css`

- 删除不需要的脚本
- 减少 DOM 访问
- 图片方面
 - 优化图片：根据实际颜色需要选择色深、压缩
 - 优化 CSS 精灵
 - 不要在 HTML 中拉伸图片

你有用过哪些前端性能优化的方法？

- 减少http请求次数：CSS Sprites, JS、CSS源码压缩、图片大小控制合适；网页Gzip，CDN托管，data缓存，图片服务器。
- 前端模板JS+数据，减少由于HTML标签导致的带宽浪费，前端用变量保存AJAX请求结果，每次操作本地变量，不用请求，减少请求次数
- 用innerHTML代替DOM操作，减少DOM操作次数，优化javascript性能。
- 当需要设置的样式很多时设置className而不是直接操作style
- 少用全局变量、缓存DOM节点查找的结果。减少IO读取操作
- 避免使用CSS Expression (css表达式)又称Dynamic properties(动态属性)
- 图片预加载，将样式表放在顶部，将脚本放在底部 加上时间戳
- 避免在页面的主体布局中使用table，table要等其中的内容完全下载之后才会显示出来，显示比div+css布局慢

谈谈性能优化问题

- 代码层面：避免使用css表达式，避免使用高级选择器，通配选择器
- 缓存利用：缓存Ajax，使用CDN，使用外部js和css文件以便缓存，添加Expires头，服务端配置Etag，减少DNS查找等
- 请求数量：合并样式和脚本，使用css图片精灵，初始首屏之外的图片资源按需加载，静态资源延迟加载
- 请求带宽：压缩文件，开启GZIP

前端性能优化最佳实践？

- 性能评级工具 (PageSpeed 或 YSlow)
- 合理设置 HTTP 缓存：Expires 与 Cache-control
- 静态资源打包，开启 Gzip 压缩 (节省响应流量)
- CSS3 模拟图像，图标base64 (降低请求数)
- 模块延迟(defer)加载/异步(async)加载
- Cookie 隔离 (节省请求流量)
- localStorage (本地存储)
- 使用 CDN 加速 (访问最近服务器)
- 启用 HTTP/2 (多路复用，并行加载)
- 前端自动化 (gulp/webpack)

#6 HTTP状态码及其含义

- 1XX

: 信息状态码

- 100 Continue 继续，一般在发送 post 请求时，已发送了 http header 之后服务端将返回此信息，表示确认，之后发送具体参数信息

- 2XX

: 成功状态码

- 200 OK 正常返回信息
- 201 Created 请求成功并且服务器创建了新的资源
- 202 Accepted 服务器已接受请求，但尚未处理
- 3XX
 - : 重定向
 - 301 Moved Permanently 请求的网页已永久移动到新位置。
 - 302 Found 临时性重定向。
 - 303 See Other 临时性重定向，且总是使用 GET 请求新的 URI。
 - 304 Not Modified 自从上次请求后，请求的网页未修改过。
- 4XX
 - : 客户端错误
 - 400 Bad Request 服务器无法理解请求的格式，客户端不应当尝试再次使用相同的内容发起请求。
 - 401 Unauthorized 请求未授权。
 - 403 Forbidden 禁止访问。
 - 404 Not Found 找不到如何与 URI 相匹配的资源。
- 5XX:

服务器错误

- 500 Internal Server Error 最常见的服务器端错误。
- 503 Service Unavailable 服务器端暂时无法处理请求（可能是过载或维护）。

#7 语义化的理解

- 用正确的标签做正确的事情！
- HTML 语义化就是让页面的内容结构化，便于对浏览器、搜索引擎解析；
- 在没有样式 CSS 情况下也以一种文档格式显示，并且是容易阅读的。
- 搜索引擎的爬虫依赖于标记来确定上下文和各个关键字的权重，利于 SEO。
- 使阅读源代码的人对网站更容易将网站分块，便于阅读维护理解

#8 介绍一下你对浏览器内核的理解？

- 主要分成两部分：渲染引擎 (Layout engine 或 Rendering Engine) 和 JS 引擎
- 渲染引擎：负责取得网页的内容 (HTML、XML、图像等等)、整理讯息 (例如加入 CSS 等)，以及计算网页的显示方式，然后会输出至显示器或打印机。浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同。所有网页浏览器、电子邮件客户端以及其它需要编辑、显示网络内容的应用程序都需要内核
- JS 引擎则：解析和执行 JavaScript 来实现网页的动态效果
- 最开始渲染引擎和 JS 引擎并没有区分的很明确，后来 JS 引擎越来越独立，内核就倾向于只指渲染引擎

常见的浏览器内核有哪些

- Trident 内核：IE, Maxthon, TT, The world, 360, 搜狗浏览器等。[又称 MSHTML]
- Gecko 内核：Netscape6 及以上版本，FF, Mozilla Suite/SeaMonkey 等
- Presto 内核：Opera7 及以上。[Opera 内核原为：Presto，现为：Blink；]
- Webkit 内核：Safari, Chrome 等。[Chrome 的 Blink (Webkit 的分支)]

#9 HTML5有哪些新特性、移除了哪些元素？

- HTML5 现在已经不是 SGML 的子集，主要是关于图像，位置，存储，多任务等功能的增加
 - 新增选择器 `document.querySelector`、`document.querySelectorAll`
 - 拖拽释放(drag and drop) API
 - 媒体播放的 `video` 和 `audio`
 - 本地存储 `localStorage` 和 `sessionStorage`
 - 离线应用 `manifest`
 - 桌面通知 `Notifications`
 - 语意化标签 `article`、`footer`、`header`、`nav`、`section`
 - 增强表单控件 `calendar`、`date`、`time`、`email`、`url`、`search`
 - 地理位置 `Geolocation`
 - 多任务 `webworker`
 - 全双工通信协议 `websocket`
 - 历史管理 `history`
 - 跨域资源共享(CORS) `Access-Control-Allow-Origin`
 - 页面可见性改变事件 `visibilitychange`
 - 跨窗口通信 `PostMessage`
 - `Form Data` 对象
 - 绘画 `canvas`
- 移除的元素：
 - 纯表现的元素： `basefont`、`big`、`center`、`font`、`s`、`strike`、`tt`、`u`
 - 对可用性产生负面影响的元素： `frame`、`frameset`、`noframes`
- 支持 HTML5 新标签：
 - IE8/IE7/IE6 支持通过 `document.createElement` 方法产生的标签
 - 可以利用这一特性让这些浏览器支持 HTML5 新标签
 - 浏览器支持新标签后，还需要添加标签默认的样式
 - 当然也可以直接使用成熟的框架，比如 `html5shim`

如何区分 HTML 和 HTML5

- `DOCTYPE` 声明、新增的结构元素、功能元素

#10 HTML5 的离线储存怎么使用，工作原理能不能解释一下？

- 在用户没有与因特网连接时，可以正常访问站点或应用，在用户与因特网连接时，更新用户机器上的缓存文件
- 原理：HTML5 的离线存储是基于一个新建的 `.appcache` 文件的缓存机制（不是存储技术），通过这个文件上的解析清单离线存储资源，这些资源就会像 `cookie` 一样被存储了下来。之后当网络在处于离线状态下时，浏览器会通过被离线存储的数据进行页面展示
- 如何使用：
 - 页面头部像下面一样加入一个 `manifest` 的属性；
 - 在 `cache.manifest` 文件的编写离线存储的资源
 - 在离线状态时，操作 `window.applicationCache` 进行需求实现

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

#11 浏览器是怎么对 HTML5 的离线储存资源进行管理和加载的呢

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

#12 请描述一下 `cookies`, `sessionStorage` 和 `localStorage` 的区别？

- `cookie` 是网站为了标示用户身份而储存在用户本地终端（Client Side）上的数据（通常经过加密）
- `cookie` 数据始终在同源的 http 请求中携带（即使不需要），记会在浏览器和服务器间来回传递
- `sessionStorage` 和 `localStorage` 不会自动把数据发给服务器，仅在本地保存
- 存储大小：
 - `cookie` 数据大小不能超过 4k
 - `sessionStorage` 和 `localStorage` 虽然也有存储大小的限制，但比 `cookie` 大得多，可以达到 5M 或更大
- 有效期：
 - `localStorage` 存储持久数据，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` 设置的 `cookie` 过期时间之前一直有效，即使窗口或浏览器关闭

#13 iframe 有哪些缺点？

- `iframe` 会阻塞主页面的 `onload` 事件
- 搜索引擎的检索程序无法解读这种页面，不利于 SEO
- `iframe` 和主页面共享连接池，而浏览器对相同域的连接有限制，所以会影响页面的并行加载
- 使用 `iframe` 之前需要考虑这两个缺点。如果需要使用 `iframe`，最好是通过 `javascript` 动态给 `iframe` 添加 `src` 属性值，这样可以绕开以上两个问题

#14 WEB 标准以及 W3C 标准是什么？

- 标签闭合、标签小写、不乱嵌套、使用外链 `css` 和 `js`、结构行为表现的分离

#15 XHTML和HTML有什么区别?

- 一个是功能上的差别
 - 主要是 XHTML 可兼容各大浏览器、手机以及 PDA，并且浏览器也能快速正确地编译网页
- 另外是书写习惯的差别
 - XHTML 元素必须被正确地嵌套，闭合，区分大小写，文档必须拥有根元素

#16 Doctype作用？严格模式与混杂模式如何区分？它们有何意义？

- 页面被加载时，`link` 会同时被加载，而 `@import` 页面被加载时，`link` 会同时被加载，而 `@import` 引用的 css 会等到页面被加载完再加载 `import` 只在 IE5 以上才能识别，而 `link` 是 XHTML 标签，无兼容问题 `link` 方式的样式的权重高于 `@import` 的权重
- `<!DOCTYPE>` 声明位于文档中的最前面，处于 `<html>` 标签之前。告知浏览器的解析器，用什么文档类型规范来解析这个文档
- 严格模式的排版和 JS 运作模式是以该浏览器支持的最高标准运行
- 在混杂模式中，页面以宽松的向后兼容的方式显示。模拟老式浏览器的行为以防止站点无法工作。`DOCTYPE` 不存在或格式不正确会导致文档以混杂模式呈现

#17 行内元素有哪些？块级元素有哪些？空(void)元素有那些？行内元素和块级元素有什么区别？

- 行内元素有：`a b span img input select strong`
- 块级元素有：`div ul ol li dl dt dd h1 h2 h3 h4... p`
- 空元素：`
 <hr> <input> <link> <meta>`
- 行内元素不可以设置宽高，不独占一行
- 块级元素可以设置宽高，独占一行

#18 HTML全局属性(global attribute)有哪些

- `class`：为元素设置类标识
- `data-*`：为元素增加自定义属性
- `draggable`：设置元素是否可拖拽
- `id`：元素 `id`，文档内唯一
- `lang`：元素内容的语言
- `style`：行内 CSS 样式
- `title`：元素相关的建议信息

#19 Canvas和SVG有什么区别？

- `svg` 绘制出来的每一个图形的元素都是独立的 DOM 节点，能够方便的绑定事件或用来修改。`canvas` 输出的是一整幅画布
- `svg` 输出的图形是矢量图形，后期可以修改参数来自由放大缩小，不会失真和锯齿。而 `canvas` 输出标量画布，就像一张图片一样，放大会失真或者锯齿

#20 HTML5 为什么只需要写

- `HTML5` 不基于 `SGML`，因此不需要对 `DTD` 进行引用，但是需要 `doctype` 来规范浏览器的行为
- 而 `HTML4.01` 基于 `SGML`，所以需要对 `DTD` 进行引用，才能告知浏览器文档所使用的文档类型

#21 如何在页面上实现一个圆形的可点击区域?

- `svg`
- `border-radius`
- 纯 `js` 实现需要求一个点在不在圆上简单算法、获取鼠标坐标等等

~~#22 网页验证码是干嘛的，是为了解决什么安全问题~~

- 区分用户是计算机还是人的公共全自动程序。可以防止恶意破解密码、刷票、论坛灌水
- 有效防止黑客对某一个特定注册用户用特定程序暴力破解方式进行不断的登陆尝试

#23 viewport

```
<meta name="viewport" content="width=device-width,initial-scale=1.0,minimum-scale=1.0,maximum-scale=1.0,user-scalable=no" />
    // width    设置viewport宽度，为一个正整数，或字符串‘device-width’
    // device-width 设备宽度
    // height   设置viewport高度，一般设置了宽度，会自动解析出高度，可以不用设置
    // initial-scale 默认缩放比例（初始缩放比例），为一个数字，可以带小数
    // minimum-scale 允许用户最小缩放比例，为一个数字，可以带小数
    // maximum-scale 允许用户最大缩放比例，为一个数字，可以带小数
    // user-scalable 是否允许手动缩放
```

- 延伸提问
 - 怎样处理移动端 `1px` 被渲染成 `2px` 问题

局部处理

- `meta` 标签中的 `viewport` 属性，`initial-scale` 设置为 `1`
- `rem` 按照设计稿标准走，外加利用 `transform` 的 `scale(0.5)` 缩小一倍即可；

全局处理

- `mate` 标签中的 `viewport` 属性，`initial-scale` 设置为 `0.5`
- `rem` 按照设计稿标准走即可

~~#24 渲染优化~~

- 禁止使用 `iframe` (阻塞父文档 `onload` 事件)
 - `iframe` 会阻塞主页面的 `onload` 事件
 - 搜索引擎的检索程序无法解读这种页面，不利于SEO
 - `iframe` 和主页面共享连接池，而浏览器对相同域的连接有限制，所以会影响页面的并行加载
 - 使用 `iframe` 之前需要考虑这两个缺点。如果需要使用 `iframe`，最好是通过 `javascript`
 - 动态给 `iframe` 添加 `src` 属性值，这样可以绕开以上两个问题
- 禁止使用 `gif` 图片实现 `loading` 效果 (降低 `CPU` 消耗，提升渲染性能)
- 使用 css3 代码代替 js 动画 (尽可能避免重绘重排以及回流)
- 对于一些小图标，可以使用 `base64` 位编码，以减少网络请求。但不建议大图使用，比较耗费 `CPU`
 - 小图标优势在于
 - ~~减少 HTTP 请求~~
 - 避免文件跨域
 - 修改及时生效

- 页面头部的 `<style></style>` `<script></script>` 会阻塞页面；（因为 Renderer 进程中 JS 线程和渲染线程是互斥的）
- 页面中空的 `href` 和 `src` 会阻塞页面其他资源的加载（阻塞下载进程）
- 网页 gzip，CDN 托管，data 缓存，图片服务器
- 前端模板 JS+数据，减少由于 HTML 标签导致的带宽浪费，前端用变量保存 AJAX 请求结果，每次操作本地变量，不用请求，减少请求次数
- 用 `innerHTML` 代替 DOM 操作，减少 DOM 操作次数，优化 javascript 性能
- 当需要设置的样式很多时设置 `className` 而不是直接操作 `style`
- 少用全局变量、缓存 DOM 节点查找的结果。减少 IO 读取操作
- 图片预加载，将样式表放在顶部，将脚本放在底部 加上时间戳
- 对普通的网站有一个统一的思路，就是尽量向前端优化、减少数据库操作、减少磁盘 IO

#25 meta viewport相关

```

<!DOCTYPE html> <!--H5标准声明，使用 HTML5 doctype，不区分大小写-->
<head lang="en"> <!--标准的 lang 属性写法-->
<meta charset='utf-8'> <!--声明文档使用的字符编码-->
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1"/> <!--优先使用 IE 最新版本和 Chrome-->
<meta name="description" content="不超过150个字符"/> <!--页面描述-->
<meta name="keywords" content="" /> <!-- 页面关键词-->
<meta name="author" content="name, email@gmail.com"/> <!--网页作者-->
<meta name="robots" content="index,follow"/> <!--搜索引擎抓取-->
<meta name="viewport" content="initial-scale=1, maximum-scale=3, minimum-scale=1, user-scalable=no"/> <!--为移动设备添加 viewport-->
<meta name="apple-mobile-web-app-title" content="标题"> <!--ios 设备 begin-->
<meta name="apple-mobile-web-app-capable" content="yes"/> <!--添加到主屏后的标题 (ios 6 新增)
是否启用 WebApp 全屏模式，删除苹果默认的工具栏和菜单栏-->
<meta name="apple-itunes-app" content="app-id=myAppStoreID, affiliate-data=myAffiliateData, app-argument=myURL">
<!--添加智能 App 广告条 Smart App Banner (ios 6+ Safari) -->
<meta name="apple-mobile-web-app-status-bar-style" content="black"/>
<meta name="format-detection" content="telephone=no, email=no"/> <!--设置苹果工具栏颜色-->
<meta name="renderer" content="webkit"> <!-- 启用360浏览器的极速模式(webkit)-->
<meta http-equiv="X-UA-Compatible" content="IE=edge"> <!--避免IE使用兼容模式-->
<meta http-equiv="Cache-Control" content="no-siteapp" /> <!--不让百度转码-->
<meta name="HandheldFriendly" content="true"> <!--针对手持设备优化，主要是针对一些老的不识别viewport的浏览器，比如黑莓-->
<meta name="MobileOptimized" content="320"> <!--微软的老式浏览器-->
<meta name="screen-orientation" content="portrait"> <!--uc强制竖屏-->
<meta name="x5-orientation" content="portrait"> <!--QQ强制竖屏-->
<meta name="full-screen" content="yes"> <!--UC强制全屏-->
<meta name="x5-fullscreen" content="true"> <!--QQ强制全屏-->
<meta name="browsermode" content="application"> <!--UC应用模式-->
<meta name="x5-page-mode" content="app"> <!-- QQ应用模式-->
<meta name="msapplication-tap-highlight" content="no"> <!--windows phone 点击无高亮
设置页面不缓存-->
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0"/>

```

#26 你做的页面在哪些浏览器测试过？这些浏览器的内核分别是什么？

- IE: trident 内核
- Firefox: gecko 内核
- Safari: webkit 内核
- Opera: 以前是 presto 内核, Opera 现已改用 Google - Chrome 的 Blink 内核
- Chrome: Blink (基于 webkit, Google 与 Opera Software 共同开发)

④ #27 div+css的布局较table布局有什么优点？

- 改版的时候更方便 只要改 css 文件。
- 页面加载速度更快、结构化清晰、页面显示简洁。
- 表现与结构相分离。
- 易于优化 (seo) 搜索引擎更友好，排名更容易靠前。

#28 a: img的alt与title有何异同? b: strong与em的异同?

- alt(alt text): 为不能显示图像、窗体或 applets 的用户代理 (UA), alt 属性用来指定替换文字。替换文字的语言由 lang 属性指定。(在 IE 浏览器下会在没有 title 时把 alt 当成 tool tip 显示)
- title(tool tip): 该属性为设置该属性的元素提供建议性的信息
- strong: 粗体强调标签, 强调, 表示内容的重要性
- em: 斜体强调标签, 更强烈强调, 表示内容的强调点

#29 你能描述一下渐进增强和优雅降级之间的不同吗

- 渐进增强: 针对低版本浏览器进行构建页面, 保证最基本的功能, 然后再针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。
- 优雅降级: 一开始就构建完整的功能, 然后再针对低版本浏览器进行兼容。

区别: 优雅降级是从复杂的现状开始, 并试图减少用户体验的供给, 而渐进增强则是从一个非常基础的, 能够起作用的版本开始, 并不断扩充, 以适应未来环境的需要。降级 (功能衰减) 意味着往回看; 而渐进增强则意味着朝前看, 同时保证其根基处于安全地带

④ #30 为什么利用多个域名来存储网站资源会更有效?

- CDN 缓存更方便
- 突破浏览器并发限制
- 节约 cookie 带宽
- 节约主域名的连接数, 优化页面响应速度
- 防止不必要的安全问题

#31 简述一下src与href的区别

- src 用于替换当前元素, href 用于在当前文档和引用资源之间确立联系。
- src 是 source 的缩写, 指向外部资源的位置, 指向的内容将会嵌入到文档中当前标签所在位置; 在请求 src 资源时会将其指向的资源下载并应用到文档内, 例如 js 脚本, img 图片和 frame 等元素

当浏览器解析到该元素时, 会暂停其他资源的下载和处理, 直到将该资源加载、编译、执行完毕, 图片和框架等元素也如此, 类似于将所指向资源嵌入当前标签内。这也是为什么将js脚本放在

底部而不是头部

- `href` 是 Hypertext Reference 的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加
- `<link href="common.css" rel="stylesheet"/>` 那么浏览器会识别该文档为 `css` 文件，就会并行下载资源并且不会停止对当前文档的处理。这也是为什么建议使用 `link` 方式来加载 `css`，而不是使用 `@import` 方式

#32 知道的网页制作会用到的图片格式有哪些？

- `png-8`、`png-24`、`jpeg`、`gif`、`svg`

但是上面的那些都不是面试官想要的最后答案。面试官希望听到是 `webp`, `Apng`。（是否有关注新技术，新鲜事物）

- **Webp**: `WebP` 格式，谷歌（google）开发的一种旨在加快图片加载速度的图片格式。图片压缩体积大约只有 `JPEG` 的 `2/3`，并能节省大量的服务器带宽资源和数据空间。`Facebook` `Ebay` 等知名网站已经开始测试并使用 `WebP` 格式。
- 在质量相同的情况下，`WebP` 格式图像的体积要比 `JPEG` 格式图像小 `40%`。
- **Apng**: 全称是“`Animated Portable Network Graphics`”，是 `PNG` 的位图动画扩展，可以实现 `png` 格式的动态图片效果。04 年诞生，但一直得不到各大浏览器厂商的支持，直到日前得到 `iOS` `safari 8` 的支持，有望代替 `GIF` 成为下一代动态图标准

#33 在css/js代码上线之后开发人员经常会优化性能，从用户刷新网页开始，一次js请求一般情况下有哪些地方会有缓存处理？

`dns` 缓存，`cdn` 缓存，浏览器缓存，服务器缓存

#33 一个页面上有大量的图片（大型电商网站），加载很慢，你有哪些方法优化这些图片的加载，给用户更好的体验。

- **图片懒加载**：在页面上的未可视区域可以添加一个滚动事件，判断图片位置与浏览器顶端的距离与页面的距离，如果前者小于后者，优先加载。
- 如果为幻灯片、相册等，可以使用图片预加载技术，将当前展示图片的前一张和后一张优先下载。
- 如果图片为 `css` 图片，可以使用 `CSSsprite`、`SVGSprite`、`Iconfont`、`Base64` 等技术。
- 如果图片过大，可以使用特殊编码的图片，加载时会先加载一张压缩的特别厉害的缩略图，以提高用户体验。
- 如果图片展示区域小于图片的真实大小，则因在服务器端根据业务需要先行进行图片压缩，图片压缩后大小与展示一致。

#34 常见排序算法的时间复杂度,空间复杂度

#35 web开发中会话跟踪的方法有哪些

- `cookie`
- `session`
- `url 重写`
- 隐藏 `input`
- `ip 地址`

#36 HTTP request报文结构是怎样的

1. 首行是Request-Line包括：请求方法，请求URI，协议版本，CRLF
2. 首行之后是若干行请求头，包括general-header，request-header或者entity-header，每一个一行以CRLF结束
3. 请求头和消息实体之间有一个CRLF分隔
4. 根据实际请求需要可能包含一个消息实体 一个请求报文例子如下：

```
GET /Protocols/rfc2616/rfc2616-sec5.html HTTP/1.1
Host: www.w3.org
Connection: keep-alive
Cache-Control: max-age=0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/35.0.1916.153 Safari/537.36
Referer: https://www.google.com.hk/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6
Cookie: authorstyle=yes
If-None-Match: "2cc8-3e3073913b100"
If-Modified-Since: wed, 01 Sep 2004 13:24:52 GMT

name=qiu&age=25
```

#37 HTTP response报文结构是怎样的

- 首行是状态行包括：HTTP版本，状态码，状态描述，后面跟一个CRLF
- 首行之后是若干行响应头，包括：通用头部，响应头部，实体头部
- 响应头部和响应实体之间用一个CRLF空行分隔
- 最后是一个可能的消息实体 响应报文例子如下：

```
HTTP/1.1 200 OK
Date: Tue, 08 Jul 2014 05:28:43 GMT
Server: Apache/2
Last-Modified: wed, 01 Sep 2004 13:24:52 GMT
ETag: "40d7-3e3073913b100"
Accept-Ranges: bytes
Content-Length: 16599
Cache-Control: max-age=21600
Expires: Tue, 08 Jul 2014 11:28:43 GMT
P3P: policyref="http://www.w3.org/2001/05/P3P/p3p.xml"
Content-Type: text/html; charset=iso-8859-1

{"name": "qiu", "age": 25}
```

#38 title与h1的区别、b与strong的区别、i与em的区别

- `title` 属性没有明确意义只表示是个标题，H1则表示层次明确的标题，对页面信息的抓取也有很大的影响
- `strong` 是标明重点内容，有语气加强的含义，使用阅读设备阅读网络时：`` 会重读，而 `` 是展示强调内容
- i内容展示为斜体，em表示强调的文本

#39 请你谈谈Cookie的弊端

cookie 虽然在持久保存客户端数据提供了方便，分担了服务器存储的负担，但还是有很多局限性的

- 每个特定的域名下最多生成 20 个 cookie
- IE6 或更低版本最多 20 个 cookie
- IE7 和之后的版本最后可以有 50 个 cookie
- Firefox 最多 50 个 cookie
- chrome 和 safari 没有做硬性限制
- IE 和 Opera 会清理近期最少使用的 cookie，Firefox 会随机清理 cookie
- cookie 的最大大约为 4096 字节，为了兼容性，一般设置不超过 4095 字节
- 如果 cookie 被人拦截了，就可以取得所有的 session 信息

#git fetch和git pull的区别

- git pull：相当于是从远程获取最新版本并 merge 到本地
- git fetch：相当于是从远程获取最新版本到本地，不会自动 merge

#二、CSS部分

#1 css sprite是什么,有什么优缺点

- 概念：将多个小图片拼接到一个图片中。通过 background-position 和元素尺寸调节需要显示的背景图案。
- 优点：
 - 减少 HTTP 请求数，极大地提高页面加载速度
 - 增加图片信息重复度，提高压缩比，减少图片大小
 - 更换风格方便，只需在一张或几张图片上修改颜色或样式即可实现
- 缺点：
 - 图片合并麻烦
 - 维护麻烦，修改一个图片可能需要从新布局整个图片，样式

#2 display: none;与 visibility: hidden;的区别

- 联系：它们都能让元素不可见
- 区别：
 - display:none ;会让元素完全从渲染树中消失，渲染的时候不占据任何空间； visibility: hidden; 不会让元素从渲染树消失，渲染时元素继续占据空间，只是内容不可见
 - display: none ;是非继承属性，子孙节点消失由于元素从渲染树消失造成，通过修改子孙节点属性无法显示； visibility: hidden; 是继承属性，子孙节点消失由于继承了 hidden，通过设置 visibility: visible; 可以让子孙节点显式
 - 修改常规流中元素的 display 通常会造成文档重排。修改 visibility 属性只会造成本元素的重绘。
 - 读屏器不会读取 display: none ;元素内容；会读取 visibility: hidden; 元素内容

#3 ~~link与@import的区别~~

1. `link` 是 HTML 方式, `@import` 是 CSS 方式
2. `link` 最大限度支持并行下载, `@import` 过多嵌套导致串行下载, 出现 FOUc (文档样式短暂失效)
3. `link` 可以通过 `rel="alternate stylesheet"` 指定候选样式
4. 浏览器对 `link` 支持早于 `@import`, 可以使用 `@import` 对老浏览器隐藏样式
5. `@import` 必须在样式规则之前, 可以在 css 文件中引用其他文件
6. 总体来说: `link` 优于 `@import`

#4 什么是FOUC?如何避免

- `Flash of Unstyled Content`: 用户定义样式表加载之前浏览器使用默认样式显示文档, 用户样式加载渲染之后再从新显示文档, 造成页面闪烁。
- **解决方法:** 把样式表放到文档的 `<head>`

#5 如何创建块级格式化上下文(block formatting context),BFC有什么用

BFC(Block Formatting Context), 块级格式化上下文, 是一个独立的渲染区域, 让处于 BFC 内部的元素与外部的元素相互隔离, 使内外元素的定位不会相互影响

触发条件(以下任意一条)

- `float` 的值不为 `none`
- `overflow` 的值不为 `visible`
- `display` 的值为 `table-cell`、`table-caption` 和 `inline-block` 之一
- `position` 的值不为 `static` 或则 `relative` 中的任何一个

在 IE 下, `Layout`, 可通过 `zoom:1` 触发

.BFC布局与普通文档流布局区别 普通文档流布局:



- 浮动的元素是不会被父级计算高度
- 非浮动元素会覆盖浮动元素的位置
- `margin` 会传递给父级元素
- 两个相邻元素上下的 `margin` 会重叠

BFC布局规则:

- 浮动的元素会被父级计算高度(父级元素触发了 BFC)
- 非浮动元素不会覆盖浮动元素的位置(非浮动元素触发了 BFC)
- `margin` 不会传递给父级(父级触发 BFC)
- 属于同一个 BFC 的两个相邻元素上下 `margin` 会重叠

开发中的应用

- 阻止 `margin` 重叠
- 可以包含浮动元素 —— 清除内部浮动(清除浮动的原理是两个 `div` 都位于同一个 BFC 区域之中)
- 自适应两栏布局
- 可以阻止元素被浮动元素覆盖

#6 display、float、position的关系

- 如果 `display` 取值为 `none`，那么 `position` 和 `float` 都不起作用，这种情况下元素不产生框
- 否则，如果 `position` 取值为 `absolute` 或者 `fixed`，框就是绝对定位的，`float` 的计算值为 `none`，`display` 根据下面的表格进行调整。
- 否则，如果 `float` 不是 `none`，框是浮动的，`display` 根据下表进行调整
- 否则，如果元素是根元素，`display` 根据下表进行调整
- 其他情况下 `display` 的值为指定值
- 总结起来：**绝对定位、浮动、根元素都需要调整 `display`**

#7 清除浮动的几种方式，各自的优缺点

- 父级 `div` 定义 `height`
- 结尾处加空 `div` 标签 `clear:both`
- 父级 `div` 定义伪类 `:after` 和 `zoom`
- 父级 `div` 定义 `overflow:hidden`
- 父级 `div` 也浮动，需要定义宽度
- 结尾处加 `br` 标签 `clear:both`
- 比较好的是第3种方式，好多网站都这么用

#8 为什么要初始化CSS样式？

- 因为浏览器的兼容问题，不同浏览器对有些标签的默认值是不同的，如果没对 `css` 初始化往往会出现浏览器之间的页面显示差异。
- 当然，初始化样式会对 `seo` 有一定的影响，但鱼和熊掌不可兼得，但力求影响最小的情况下初始化

#9 css3有哪些新特性

- 新增选择器 `p:nth-child(n){color: rgba(255, 0, 0, 0.75)}`
- 弹性盒模型 `display: flex;`
- 多列布局 `column-count: 5;`
- 媒体查询 `@media (max-width: 480px) { .box: {column-count: 1;}}`
- 个性化字体 `@font-face{font-family: Borderweb; src:url(BORDERW0.eot);}`
- 颜色透明度 `color: rgba(255, 0, 0, 0.75);`
- 圆角 `border-radius: 5px;`
- 渐变 `background: linear-gradient(red, green, blue);`
- 阴影 `box-shadow: 3px 3px 3px rgba(0, 64, 128, 0.3);`
- 倒影 `box-reflect: below 2px;`
- 文字装饰 `text-stroke-color: red;`
- 文字溢出 `text-overflow: ellipsis;`
- 背景效果 `background-size: 100px 100px;`
- 边框效果 `border-image:url(bt_blue.png) 0 10;`
- 转换
 - 旋转 `transform: rotate(20deg);`
 - 倾斜 `transform: skew(150deg, -10deg);`
 - 位移 `transform: translate(20px, 20px);`

- 缩放 `transform: scale(.5);`
- 平滑过渡 `transition: all .3s ease-in .1s;`
- 动画 `@keyframes anim-1 {50% {border-radius: 50%;}} animation: anim-1 1s;`

CSS3新增伪类有那些？

- `p:first-of-type` 选择属于其父元素的首个 `<p>` 元素的每个 `<p>` 元素。
- `p:last-of-type` 选择属于其父元素的最后 `<p>` 元素的每个 `<p>` 元素。
- `p:only-of-type` 选择属于其父元素唯一的 `<p>` 元素的每个 `<p>` 元素。
- `p:only-child` 选择属于其父元素的唯一子元素的每个 `<p>` 元素。
- `p:nth-child(2)` 选择属于其父元素的第二个子元素的每个 `<p>` 元素。
- `:after` 在元素之前添加内容,也可以用来做清除浮动。
- `:before` 在元素之后添加内容。
- `:enabled` 已启用的表单元素。
- `:disabled` 已禁用的表单元素。
- `:checked` 单选框或复选框被选中。

#10 display有哪些值？说明他们的作用

- `block` 转换成块状元素。
- `inline` 转换成行内元素。
- `none` 设置元素不可见。
- `inline-block` 象行内元素一样显示，但其内容象块类型元素一样显示。
- `list-item` 象块类型元素一样显示，并添加样式列表标记。
- `table` 此元素会作为块级表格来显示
- `inherit` 规定应该从父元素继承 `display` 属性的值

#11 介绍一下标准的CSS的盒子模型？低版本IE的盒子模型有什么不同的？

- 有两种，`IE` 盒子模型、`W3C` 盒子模型；
- 盒模型：内容(`content`)、填充(`padding`)、边界(`margin`)、边框(`border`)；
- 区别：`IE` 的 `content` 部分把 `border` 和 `padding` 计算了进去；
- 盒子模型构成：内容(`content`)、内填充(`padding`)、边框(`border`)、外边距(`margin`)
- `IE8` 及其以下版本浏览器，未声明 `DOCTYPE`，内容宽高会包含内填充和边框，称为怪异盒模型（`IE` 盒模型）
- 标准(`W3C`)盒模型：元素宽度 = `width + padding + border + margin`
- 怪异(`IE`)盒模型：元素宽度 = `width + margin`
- 标准浏览器通过设置 `css3` 的 `box-sizing: border-box` 属性，触发“怪异模式”解析计算宽高

`box-sizing` 常用的属性有哪些？分别有什么作用

- `box-sizing: content-box;` 默认的标准(W3C)盒模型元素效果
- `box-sizing: border-box;` 触发怪异(IE)盒模型元素的效果
- `box-sizing: inherit;` 继承父元素 `box-sizing` 属性的值

#12 CSS优先级算法如何计算？

- 优先级就近原则，同权重情况下样式定义最近者为准
- 载入样式以最后载入的定位为准
- 优先级为：`!important > id > class > tag; !important` 比内联优先级高

#13 对BFC规范的理解?

- 一个页面是由很多个 Box 组成的,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` (决定如何渲染文档的容器),因此Box内的元素会以不同的方式渲染,也就是说BFC内部的元素和外部的元素不会互相影响

#14 谈谈浮动和清除浮动

- 浮动的框可以向左或向右移动,直到他的外边缘碰到包含框或另一个浮动框的边框为止。由于浮动框不在文档的普通流中,所以文档的普通流的块框表现得就像浮动框不存在一样。浮动的块框会漂浮在文档普通流的块框上

#15 position的值, relative和absolute定位原点是

- `absolute`: 生成绝对定位的元素,相对于 `static` 定位以外的第一个父元素进行定位
- `fixed`: 生成绝对定位的元素,相对于浏览器窗口进行定位
- `relative`: 生成相对定位的元素,相对于其正常位置进行定位
- `static` 默认值。没有定位,元素出现在正常的流中
- `inherit` 规定从父元素继承 `position` 属性的值

#16 display:inline-block 什么时候不会显示间隙? (携程)

- 移除空格
- 使用 `margin` 负值
- 使用 `font-size:0`
- `letter-spacing`
- `word-spacing`

#17 PNG\GIF\JPEG的区别及如何选

- `GIF`
 - 8位像素, 256 色
 - 无损压缩
 - 支持简单动画
 - 支持 boolean 透明
 - 适合简单动画
- `JPEG`
 - 颜色限于 256
 - 有损压缩
 - 可控制压缩质量
 - 不支持透明
 - 适合照片
- `PNG`
 - 有 PNG8 和 truecolor PNG
 - PNG8 类似 GIF 颜色上限为 256, 文件小, 支持 alpha 透明度, 无动画
 - 适合图标、背景、按钮

#18 行内元素float:left后是否变为块级元素?

行内元素设置成浮动之后变得更加像是 inline-block (行内块级元素，设置成这个属性的元素会同时拥有行内和块级的特性，最明显的是它的默认宽度不是 100%)，这时候给行内元素设置 padding-top 和 padding-bottom 或者 width、height 都是有效果的

#19 在网页中的应该使用奇数还是偶数的字体？为什么呢？

- 偶数字号相对更容易和 web 设计的其他部分构成比例关系

#20 ::before 和 :after 中双冒号和单冒号有什么区别？解释一下这两个伪元素的作用

- 单冒号(:)用于 css3 伪类，双冒号(::)用于 css3 伪元素
- 用于区分伪类和伪元素

#21 如果需要手动写动画，你认为最短时间间隔是多久，为什么？(阿里)

- 多数显示器默认频率是 60Hz，即 1 秒刷新 60 次，所以理论上最小间隔为 $1/60 * 1000ms = 16.7ms$

#22 CSS 合并方法

- 避免使用 @import 引入多个 css 文件，可以使用 css 工具将 css 合并为一个 css 文件，例如使用 Sass\Compass 等

#23 CSS 不同选择器的权重(CSS 层叠的规则)

- ! important 规则最重要，大于其它规则
- 行内样式规则，加 1000
- 对于选择器中给定的各个 ID 属性值，加 100
- 对于选择器中给定的各个类属性、属性选择器或者伪类选择器，加 10
- 对于选择其中给定的各个元素标签选择器，加 1
- 如果权值一样，则按照样式规则的先后顺序来应用，顺序靠后的覆盖靠前的规则

以下是权重的规则：标签的权重为 1， class 的权重为 10， id 的权重为 100，以下 // 例子是演示各种定义的权重值：

```
/* 权重为1 */
div{
}

/* 权重为10 */
.class1{
}

/* 权重为100 */
#id1{
}

/* 权重为100+1=101 */
#id1 div{
}

/* 权重为10+1=11 */
.class1 div{}
```

```
/*权重为10+10+1=21*/
.class1 .class2 div{}
```

如果权重相同，则最后定义的样式会起作用，但是应该避免这种情况出现

#24 列出你所知道可以改变页面布局的属性

- position、display、float、width、height、margin、padding、top、left、right、

#25 CSS在性能优化方面的实践

- css 压缩与合并、Gzip 压缩
- css 文件放在 head 里、不要用 @import
- 尽量用缩写、避免用滤镜、合理使用选择器

#26 CSS3动画（简单动画的实现，如旋转等）

- 依靠 css3 中提出的三个属性：transition、transform、animation
- transition：定义了元素在变化过程中是怎么样的，包含 transition-property、transition-duration、transition-timing-function、transition-delay。
- transform：定义元素的变化结果，包含 rotate、scale、skew、translate。
- animation：动画定义了动作的每一帧 (@keyframes) 有什么效果，包括 animation-name、animation-duration、animation-timing-function、animation-delay、animation-iteration-count、animation-direction

#27 base64的原理及优缺点

- 优点可以加密，减少了 HTTP 请求
- 缺点是需要消耗 CPU 进行编解码

#28 几种常见的CSS布局

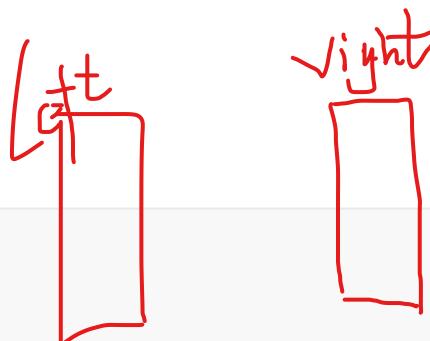
#流体布局

```
.left {
    float: left;
    width: 100px;
    height: 200px;
    background: red;
}

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

.main {
    margin-left: 120px;
    margin-right: 220px;
    height: 200px;
    background: green;
}

<div class="container">
```



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

#圣杯布局

- 要求：三列布局；中间主体内容前置，且宽度自适应；两边内容定宽

好处：重要的内容放在文档流前面可以优先渲染

原理：利用相对定位、浮动、负边距布局，而不添加额外标签



```
.container {
    padding-left: 150px;
    padding-right: 190px;
}
.main {
    float: left;
    width: 100%;
}
.left {
    float: left;
    width: 190px;
    margin-left: -100%;
    position: relative;
    left: -150px;
}
.right {
    float: left;
    width: 190px;
    margin-left: -190px;
    position: relative;
    right: -190px;
}
<div class="container">
    <div class="main"></div>✓
    <div class="left"></div>✓
    <div class="right"></div>✓
</div>
```

#双飞翼布局

- 双飞翼布局：对圣杯布局（使用相对定位，对以后布局有局限性）的改进，消除相对定位布局
- 原理：主体元素上设置左右边距，预留两翼位置。左右两栏使用浮动和负边距归位，消除相对定位。

```
.container {
    /*padding-left:150px;*/
    /*padding-right:190px;*/
}
.main-wrap {
    width: 100%;
    float: left;
}
.main {
    margin-left: 150px;
    margin-right: 190px;
```

```
}

.left {
    float: left;
    width: 150px;
    margin-left: -100%;
    /*position: relative;*/
    /*left:-150px;*/
}

.right {
    float: left;
    width: 190px;
    margin-left: -190px;
    /*position: relative;*/
    /*right:-190px;*/
}

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

#29 stylus/sass/less区别

- 均具有“变量”、“混合”、“嵌套”、“继承”、“颜色混合”五大基本特性
- `Scss` 和 `LESS` 语法较为严谨，`LESS` 要求一定要使用大括号“{}”，`Scss` 和 `stylus` 可以通过缩进表示层次与嵌套关系
- `Scss` 无全局变量的概念，`LESS` 和 `stylus` 有类似于其它语言的作用域概念
- `Sass` 是基于 Ruby 语言的，而 `LESS` 和 `stylus` 可以基于 `NodeJS` `NPM` 下载相应库后进行编译；

#30 postcss的作用

- 可以直观的理解为：它就是一个平台。为什么说它是一个平台呢？因为我们直接用它，感觉不能干什么事情，但是如果让一些插件在它上面跑，那么将会很强大
- `PostCSS` 提供了一个解析器，它能够将 `CSS` 解析成抽象语法树
- 通过在 `PostCSS` 这个平台上，我们能够开发一些插件，来处理我们的 `CSS`，比如热门的：
`autoprefixer`
- `postcss` 可以对 `sass` 处理过后的 `CSS` 再处理 最常见的就是 `autoprefixer`

#31 css样式（选择器）的优先级

- 计算权重确定
- `!important`
- 内联样式
- 后写的优先级高

#32 自定义字体的使用场景

- 宣传/品牌/ banner 等固定文案
- 字体图标

#33 如何美化CheckBox

- <label> 属性 for 和 id
- 隐藏原生的 <input>
- :checked + <label>

#34 伪类和伪元素的区别

- 伪类表状态
- 伪元素是真的有元素
- 前者单冒号，后者双冒号

#35 base64 的使用

- 用于减少 HTTP 请求
- 适用于小图片
- base64 的体积约为原图的 4/3

#36 自适应布局

思路：

- 左侧浮动或者绝对定位，然后右侧 margin 撑开
- 使用 <div> 包含，然后靠负 margin 形成 bfc
- 使用 flex

#37 请用CSS写一个简单的幻灯片效果页面

知道是要用 css3。使用 animation 动画实现一个简单的幻灯片效果

```
/**css*/
.ani{
    width:480px;
    height:320px;
    margin:50px auto;
    overflow: hidden;
    box-shadow:0 0 5px rgba(0,0,0,1);
    background-size: cover;
    background-position: center;
    -webkit-animation-name: "loops";
    -webkit-animation-duration: 20s;
    -webkit-animation-iteration-count: infinite;
}
@-webkit-keyframes "loops" {
    0% {

        background:url(http://d.hiphotos.baidu.com/image/w%3D400/sign=c01e6adca964034f0fc3069fc27980/e824b899a9014c08e5e38ca4087b02087af4f4d3.jpg) no-repeat;
    }
    25% {

        background:url(http://b.hiphotos.baidu.com/image/w%3D400/sign=edee1572e9f81a4c632edc9e72b6029/30adcbe76094b364d72bceba1cc7cd98c109dd0.jpg) no-repeat;
    }
    50% {
```

```
background:url(http://b.hiphotos.baidu.com/image/w%3D400/sign=937dace2552c11dfd  
ed1be2353266255/d8f9d72a6059252d258e7605369b033b5bb5b912.jpg) no-repeat;  
}  
75% {  
  
background:url(http://g.hiphotos.baidu.com/image/w%3D400/sign=7d37500b8544ebf86  
d171653fe9f9d736/0df431adcbe76095d61f0972cdda3cc7cd99e4b.jpg) no-repeat;  
}  
100% {  
  
background:url(http://c.hiphotos.baidu.com/image/w%3D400/sign=cfb239ceb0fb43161  
a1f7b7a10a54642/3b87e950352ac65ce2e73f76f9f2b21192138ad1.jpg) no-repeat;  
}  
}
```

#38 什么是外边距重叠？重叠的结果是什么？

外边距重叠就是margin-collapse

- 在CSS当中，相邻的两个盒子（可能是兄弟关系也可能是祖先关系）的外边距可以结合成一个单独的外边距。这种合并外边距的方式被称为折叠，并且因而所结合成的外边距称为折叠外边距。

折叠结果遵循下列计算规则：

- 两个相邻的外边距都是正数时，折叠结果是它们两者之间较大的值。
- 两个相邻的外边距都是负数时，折叠结果是两者绝对值的较大值。
- 两个外边距一正一负时，折叠结果是两者的相加的和。

#39 rgba()和opacity的透明效果有什么不同？

- rgba() 和 opacity 都能实现透明效果，但最大的不同是 opacity 作用于元素，以及元素内的所有内容的透明度。
- 而 rgba() 只作用于元素的颜色或其背景色。（设置 rgba 透明的元素的子元素不会继承透明效果！）

#40 css中可以让文字在垂直和水平方向上重叠的两个属性是什么？

- 垂直方向：line-height
- 水平方向：letter-spacing

#41 如何垂直居中一个浮动元素？

```
/**方法一：已知元素的高宽**/  
  
#div1{  
background-color:#6699FF;  
width:200px;  
height:200px;  
position: absolute; //父元素需要相对定位  
top: 50%;  
left: 50%;  
margin-top:-100px ; //二分之一的height, width  
margin-left: -100px;  
}
```

```
/**方法二:**/
#div1{
    width: 200px;
    height: 200px;
    background-color: #6699FF;
    margin: auto;
    position: absolute;           //父元素需要相对定位
    left: 0;
    top: 0;
    right: 0;
    bottom: 0;
}
```

如何垂直居中一个 `` ? (用更简便的方法。)

```
#container      /**<img>的容器设置如下*/
{
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
```

#42 px和em的区别

- `px` 和 `em` 都是长度单位，区别是，`px` 的值是固定的，指定是多少就是多少，计算比较容易。`em` 得值不是固定的，并且 `em` 会继承父级元素的字体大小。
- 浏览器的默认字体高都是 `16px`。所以未经调整的浏览器都符合：`1em=16px`。那么 `12px=0.75em`，`10px=0.625em`。
 - `px` 相对于显示器屏幕分辨率，无法用浏览器字体放大功能
 - `em` 值并不是固定的，会继承父级的字体大小： $em = \text{像素值} / \text{父级 font-size}$

#43 Sass、LESS是什么？大家为什么要使用他们？

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

为什么要使用它们？

- 结构清晰，便于扩展。
- 可以方便地屏蔽浏览器私有语法差异。这个不用多说，封装对浏览器语法差异的重复处理，减少无意义的机械劳动。
- 可以轻松实现多重继承。
- 完全兼容 CSS 代码，可以方便地应用到老项目中。LESS 只是在 CSS 语法上做了扩展，所以老的 CSS 代码也可以与 LESS 代码一同编译

#44 知道 CSS 有个 content 属性吗？有什么作用？有什么应用？

css 的 `content` 属性专门应用在 `before/after` 伪元素上，用于来插入生成内容。最常见的应用是利用伪类清除浮动。

```
/*一种常见利用伪类清除浮动的代码*/
.clearfix:after {
    content: ".";
    //这里利用到了content属性
    display: block;
    height: 0;
    visibility: hidden;
    clear: both;
}
.clearfix {
    *zoom: 1;
}
```

#45 水平居中的方法

- 元素为行内元素，设置父元素 `text-align:center`
- 如果元素宽度固定，可以设置左右 `margin` 为 `auto`；
- 绝对定位和移动: `absolute + transform`
- 使用 `flex-box` 布局，指定 `justify-content` 属性为 `center`
- `display` 设置为 `table-cell`

#46 垂直居中的方法

- 将显示方式设置为表格，`display:table-cell`,同时设置 `vertical-align: middle`
- 使用 `flex` 布局，设置为 `align-item: center`
- 绝对定位中设置 `bottom:0,top:0`,并设置 `margin:auto`
- 绝对定位中固定高度时设置 `top:50%`, `margin-top` 值为高度一半的负值
- 文本垂直居中设置 `line-height` 为 `height` 值
- 如果是单行文本, `line-height` 设置成和 `height` 值

```
.vertical {
    height: 100px;
    line-height: 100px;
}
```

- 已知高度的块级子元素，采用绝对定位和负边距

```
.container {
    position: relative;
}
.vertical {
    height: 300px; /*子元素高度*/
    position: absolute;
    top: 50%; /*父元素高度50%*/
    margin-top: -150px; /*自身高度一半*/
}
```

- 未知高度的块级父子元素居中，模拟表格布局
- 缺点：IE67不兼容，父级 `overflow: hidden` 失效

```
.container {  
    display: table;  
}  
.content {  
    display: table-cell;  
    vertical-align: middle;  
}
```

- 新增 inline-block 兄弟元素，设置 vertical-align
 - 缺点：需要增加额外标签，IE6/7不兼容

```
.container {  
    height: 100%; /* 定义父级高度，作为参考 */  
}  
.extra .vertical {  
    display: inline-block; /* 行内块显示 */  
    vertical-align: middle; /* 垂直居中 */  
}  
.extra {  
    height: 100%; /* 设置新增元素高度为100% */  
}
```

- 绝对定位配合 CSS3 位移

```
.vertical {  
    position: absolute;  
    top: 50%; /* 父元素高度50% */  
    transform: translateY(-50%, -50%);  
}
```

- CSS3弹性盒模型

```
.container {  
    display: flex;  
    justify-content: center; /* 子元素水平居中 */  
    align-items: center; /* 子元素垂直居中 */  
}
```

#47 如何使用CSS实现硬件加速？

硬件加速是指通过创建独立的复合图层，让GPU来渲染这个图层，从而提高性能。

- 一般触发硬件加速的css属性有 `transform`、`opacity`、`filter`，为了避免2D动画在开始和结束的时候的 `repaint` 操作，一般使用 `transform:translateZ(0)`

#48 重绘和回流（重排）是什么，如何避免？

- 重绘：当渲染树中的元素外观（如：颜色）发生改变，不影响布局时，产生重绘
- 回流：当渲染树中的元素的布局（如：尺寸、位置、隐藏/状态状态）发生改变时，产生重绘回流
- 注意：JS获取Layout属性值（如：`offsetLeft`、`scrollTop`、`getComputedStyle`等）也会引起回流。因为浏览器需要通过回流计算最新值
- 回流必将引起重绘，而重绘不一定会引起回流

如何最小化重绘(repaint)和回流(reflow)：

- 需要对元素进行复杂的操作时，可以先隐藏(`display:"none"`)，操作完成后再显示
- 需要创建多个DOM节点时，使用`DocumentFragment`创建完后一次性的加入`document`
- 缓存`Layout`属性值，如：`var left = elem.offsetLeft;`这样，多次使用`left`只产生一次回流
- 尽量避免用`table`布局（`table`元素一旦触发回流就会导致`table`里所有的其它元素回流）
- 避免使用css表达式(`expression`)，因为每次调用都会重新计算值（包括加载页面）
- 尽量使用css属性简写，如：用`border`代替`border-width`,`border-style`,`border-color`
- 批量修改元素样式：`elem.className`和`elem.style.cssText`代替`elem.style.xxx`

#49 说一说css3的animation

- css3的`animation`是css3新增的动画属性，这个css3动画的每一帧是通过`@keyframes`来声明的，`keyframes`声明了动画的名称，通过`from`、`to`或者是百分比来定义
- 每一帧动画元素的状态，通过`animation-name`来引用这个动画，同时css3动画也可以定义动画运行的时长、动画开始时间、动画播放方向、动画循环次数、动画播放的方式，
- 这些相关的动画子属性有：`animation-name`定义动画名、`animation-duration`定义动画播放的时长、`animation-delay`定义动画延迟播放的时间、`animation-direction`定义动画的播放方向、`animation-iteration-count`定义播放次数、`animation-fill-mode`定义动画播放之后的状态、`animation-play-state`定义播放状态，如暂停运行等、`animation-timing-function`
- 定义播放的方式，如恒速播放、艰涩播放等。

#50 左边宽度固定，右边自适应

左侧固定宽度，右侧自适应宽度的两列布局实现

html结构

```
<div class="outer">
    <div class="left">固定宽度</div>
    <div class="right">自适应宽度</div>
</div>
```

在外层`div`（类名为`outer`）的`div`中，有两个子`div`，类名分别为`left`和`right`，其中`left`为固定宽度，而`right`为自适应宽度

方法1：左侧div设置成浮动：`float: left`，右侧div宽度会自拉升适应

```
.outer {
    width: 100%;
    height: 500px;
    background-color: yellow;
}
.left {
    width: 200px;
    height: 200px;
    background-color: red;
    float: left;
}
.right {
    height: 200px;
    background-color: blue;
}
```

方法2：对右侧:div进行绝对定位，然后再设置`right=0`，即可以实现宽度自适应

绝对定位元素的第一个高级特性就是其具有自动伸缩的功能，当我们设置 `width` 为 `auto` 的时候（或者不设置，默認為 `auto`），绝对定位元素会根据其 `left` 和 `right` 自动伸缩其大小

```
.outer {  
    width: 100%;  
    height: 500px;  
    background-color: yellow;  
    position: relative;  
}  
.left {  
    width: 200px;  
    height: 200px;  
    background-color: red;  
}  
.right {  
    height: 200px;  
    background-color: blue;  
    position: absolute;  
    left: 200px;  
    top: 0;  
    right: 0;  
}
```

方法3：将左侧 `div` 进行绝对定位，然后右侧 `div` 设置 `margin-left: 200px`

```
.outer {  
    width: 100%;  
    height: 500px;  
    background-color: yellow;  
    position: relative;  
}  
.left {  
    width: 200px;  
    height: 200px;  
    background-color: red;  
    position: absolute;  
}  
.right {  
    height: 200px;  
    background-color: blue;  
    margin-left: 200px;  
}
```

方法4：使用flex布局

```
.outer {  
    width: 100%;  
    height: 500px;  
    background-color: yellow;  
    display: flex;  
    flex-direction: row;  
}  
.left {  
    width: 200px;  
    height: 200px;  
    background-color: red;
```

```
}

.right {
    height: 200px;
    background-color: blue;
    flex: 1;
}
```

#51 两种以上方式实现已知或者未知宽度的垂直水平居中

```
/** 1 */
.wraper {
    position: relative;
    .box {
        position: absolute;
        top: 50%;
        left: 50%;
        width: 100px;
        height: 100px;
        margin: -50px 0 0 -50px;
    }
}

/** 2 */
.wraper {
    position: relative;
    .box {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
    }
}

/** 3 */
.wraper {
    .box {
        display: flex;
        justify-content:center;
        align-items: center;
        height: 100px;
    }
}

/** 4 */
.wraper {
    display: table;
    .box {
        display: table-cell;
        vertical-align: middle;
    }
}
```

#52 如何实现小于12px的字体效果

`transform:scale()` 这个属性只可以缩放可以定义宽高的元素，而行内元素是没有宽高的，我们可以加上一个`display:inline-block;`

```
transform: scale(0.7);
```

css 的属性，可以缩放大小

#53 css hack原理及常用hack

- 原理：利用不同浏览器对CSS的支持和解析结果不一样编写针对特定浏览器样式。
- 常见的hack有
 - 属性hack
 - 选择器hack
 - IE条件注释

#54 CSS有哪些继承属性

- 关于文字排版的属性如：

- `font`
 - `word-break`
 - `letter-spacing`
 - `text-align`
 - `text-rendering`
 - `word-spacing`
 - `white-space`
 - `text-indent`
 - `text-transform`
 - `text-shadow`
- `line-height`
- `color`
- `visibility`
- `cursor`

#55 外边距折叠(collapsing margins)

- 毗邻的两个或多个

```
margin
```

会合并成一个

```
margin
```

，叫做外边距折叠。规则如下：

- 两个或多个毗邻的普通流中的块元素垂直方向上的`margin`会折叠

- 浮动元素或 inline-block 元素或绝对定位元素的 margin 不会和垂直方向上的其他元素的 margin 折叠
- 创建了块级格式化上下文的元素，不会和它的子元素发生 margin 折叠
- 元素自身的 margin-bottom 和 margin-top 相邻时也会折

#56 CSS选择符有哪些？哪些属性可以继承

- id 选择器 (`# myid`)
- 类选择器 (`.myclassname`)
- 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 (`*`)
- 属性选择器 (`a[rel = "external"]`)
- 伪类选择器 (`a:hover, li:nth-child`)

CSS 哪些属性可以继承？哪些属性不可以继承

- 可继承的样式: `font-size font-family color, UL LI DL DD DT`
- 不可继承的样式: `border padding margin width height`

#57 CSS3新增伪类有那些

- `:root` 选择文档的根元素，等同于 `html` 元素
- `:empty` 选择没有子元素的元素
- `:target` 选取当前活动的目标元素
- `:not(selector)` 选择除 `selector` 元素意外的元素
- `:enabled` 选择可用的表单元素
- `:disabled` 选择禁用的表单元素
- `:checked` 选择被选中的表单元素
- `:after` 在元素内部最前添加内容
- `:before` 在元素内部最后添加内容
- `:nth-child(n)` 匹配父元素下指定子元素，在所有子元素中排序第n
- `:nth-last-child(n)` 匹配父元素下指定子元素，在所有子元素中排序第n，从后向前数
- `:nth-child(odd)`
- `:nth-child(even)`
- `:nth-child(3n+1)`
- `:first-child`
- `:last-child`
- `:only-child`
- `:nth-of-type(n)` 匹配父元素下指定子元素，在同类子元素中排序第n
- `:nth-last-of-type(n)` 匹配父元素下指定子元素，在同类子元素中排序第n，从后向前数
- `:nth-of-type(odd)`
- `:nth-of-type(even)`
- `:nth-of-type(3n+1)`
- `:first-of-type`
- `:last-of-type`
- `:only-of-type`
- `::selection` 选择被用户选取的元素部分
- `::first-line` 选择元素中的第一行

- `:first-letter` 选择元素中的第一个字符

#58 如何居中div? 如何居中一个浮动元素? 如何让绝对定位的div居中

- 给 `div` 设置一个宽度, 然后添加 `margin:0 auto` 属性

```
div{  
    width:200px;  
    margin:0 auto;
```

- 居中一个浮动元素

```
/* 确定容器的宽高 宽500 高 300 的层  
设置层的外边距 */
```

```
.div {  
    width:500px ; height:300px; //高度可以不设  
    margin: -150px 0 0 -250px;  
    position:relative; //相对定位  
    background-color:pink; //方便看效果  
    left:50%;  
    top:50%;  
}
```

让绝对定位的div居中

```
position: absolute;  
width: 1200px;  
background: none;  
margin: 0 auto;  
top: 0;  
left: 0;  
bottom: 0;  
right: 0;
```

#59 用纯CSS创建一个三角形的原理是什么

```
/* 把上、左、右三条边隐藏掉（颜色设为 transparent） */  
#demo {  
    width: 0;  
    height: 0;  
    border-width: 20px;  
    border-style: solid;  
    border-color: transparent transparent red transparent;  
}
```

#60 一个满屏品字布局如何设计?

- 简单的方式：
 - 上面的 `div` 宽 `100%`,
 - 下面的两个 `div` 分别宽 `50%`,
 - 然后用 `float` 或者 `inline` 使其不换行即可

#61 li与li之间有看不见的空白间隔是什么原因引起的？有什么解决办法

行框的排列会受到中间空白（回车\空格）等的影响，因为空格也属于字符，这些空白也会被应用样式，占据空间，所以会有间隔，把字符大小设为0，就没有空格了

#62 为什么要初始化CSS样式

因为浏览器的兼容问题，不同浏览器对有些标签的默认值是不同的，如果没对CSS初始化往往会出现浏览器之间的页面显示差异

#63 请列举几种隐藏元素的方法

- `visibility: hidden;` 这个属性只是简单的隐藏某个元素，但是元素占用的空间任然存在
- `opacity: 0;` CSS3 属性，设置 `0` 可以使一个元素完全透明
- `position: absolute;` 设置一个很大的 `left` 负值定位，使元素定位在可见区域之外
- `display: none;` 元素会变得不可见，并且不会再占用文档的空间。
- `transform: scale(0);` 将一个元素设置为缩放无限小，元素将不可见，元素原来所在的位置将被保留
- `<div hidden="hidden">` HTML5 属性，效果和 `display:none;` 相同，但这个属性用于记录一个元素的状态
- `height: 0;` 将元素高度设为 `0`，并消除边框
- `filter: blur(0);` CSS3 属性，将一个元素的模糊度设置为 `0`，从而使这个元素“消失”在页面中



#64 rgba() 和 opacity 的透明效果有什么不同

- `opacity` 作用于元素以及元素内的所有内容（包括文字）的透明度
- `rgba()` 只作用于元素自身的颜色或其背景色，子元素不会继承透明效果

#65 css 属性 content 有什么作用

- `content` 属性专门应用在 `before/after` 伪元素上，用于插入额外内容或样式

#66 请解释一下 CSS3 的 Flexbox (弹性盒布局模型) 以及适用场景

1Flexbox1 用于不同尺寸屏幕中创建可自动扩展和收缩布局

#67 经常遇到的浏览器的JS兼容性有哪些？解决方法是什么

- 当前样式：`getComputedStyle(el, null)` vs `el.currentStyle`
- 事件对象：`e` vs `window.event`
- 鼠标坐标：`e.pageX, e.pageY` vs `window.event.x, window.event.y`
- 按键码：`e.which` vs `event.keyCode`
- 文本节点：`el.textContent` vs `el.innerText`

#68 请写出多种等高布局

- 在列的父元素上使用这个背景图进行Y轴的铺放，从而实现一种等高列的假像
- 模仿表格布局等高列效果：兼容性不好，在ie6-7无法正常运行
- css3 flexbox 布局：.container{display: flex; align-items: stretch;}

#69 浮动元素引起的问题

- 父元素的高度无法被撑开，影响与父元素同级的元素
- 与浮动元素同级的非浮动元素会跟随其后

#70 CSS优化、提高性能的方法有哪些

- 多个css合并，尽量减少HTTP请求
- 将css文件放在页面最上面
- 移除空的css规则
- 避免使用css表达式
- 选择器优化嵌套，尽量避免层级过深
- 充分利用css继承属性，减少代码量
- 抽象提取公共样式，减少代码量
- 属性值为0时，不加单位
- 属性值为小于1的小数时，省略小数点前面的0
- css雪碧图

link href

#71 浏览器是怎样解析CSS选择器的

- 浏览器解析CSS选择器的方式是从右到左

#72 在网页中的应该使用奇数还是偶数的字体

- 在网页中的应该使用“偶数”字体：
 - 偶数字号相对更容易和web设计的其他部分构成比例关系
 - 使用奇数字号时文本段落无法对齐
 - 宋体的中文网页排版中使用最多的就是12和14

#73 margin和padding分别适合什么场景使用

- 需要在border外侧添加空白，且空白处不需要背景（色）时，使用margin
- 需要在border内侧添加空白，且空白处需要背景（色）时，使用padding

#74 抽离样式模块怎么写，说出思路

- CSS可以拆分成2部分：公共CSS和业务CSS：
 - 网站的配色，字体，交互提取出为公共CSS。这部分CSS命名不应涉及具体的业务
 - 对于业务CSS，需要有统一的命名，使用公用的前缀。可以参考面向对象的CSS

#75 元素竖向的百分比设定是相对于容器的高度吗

元素竖向的百分比设定是相对于容器的宽度，而不是高度

#76 全屏滚动的原理是什么？用到了CSS的那些属性

- 原理类似图片轮播原理，超出隐藏部分，滚动时显示
- 可能用到的CSS属性：`overflow:hidden; transform:translate(100%, 100%); display:none;`

#77 什么是响应式设计？响应式设计的基本原理是什么？如何兼容低版本的IE

- 响应式设计就是网站能够兼容多个终端，而不是为每个终端做一个特定的版本
- 基本原理是利用CSS3媒体查询，为不同尺寸的设备适配不同样式
- 对于低版本的IE，可采用JS获取屏幕宽度，然后通过resize方法来实现兼容：

```
$(window).resize(function () {
    screenRespond();
});

screenRespond();
function screenRespond(){
var screenWidth = $(window).width();
if(screenwidth <= 1800){
    $("body").attr("class", "w1800");
}
if(screenwidth <= 1400){
    $("body").attr("class", "w1400");
}
if(screenwidth > 1800){
    $("body").attr("class", "");
}
}
```

#78 什么是视差滚动效果，如何给每页做不同的动画

- 视差滚动是指多层背景以不同的速度移动，形成立体的运动效果，具有非常出色的视觉体验
- 一般把网页解剖为：背景层、内容层和悬浮层。当滚动鼠标滚轮时，各图层以不同速度移动，形成视差的
- 实现原理
 - 以“页面滚动条”作为“视差动画进度条”
 - 以“滚轮刻度”当作“动画帧度”去播放动画的
 - 监听 mousewheel 事件，事件被触发即播放动画，实现“翻页”效果

#79 a标签上四个伪类的执行顺序是怎么样的

`link > visited > hover > active`

- L-V-H-A Love hate 用喜欢和讨厌两个词来方便记忆

#80 伪元素和伪类的区别和作用

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

```
p::before {content: "第一章："}
p::after {content: "Hot!"}
p::first-line {background: red;}
p::first-letter {font-size: 30px;}
```

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

```
a:hover {color: #FF00FF}
p:first-child {color: red}
```

#81 ::before 和 :after 中双冒号和单冒号有什么区别

- 在 CSS 中伪类一直用 `:` 表示，如 `:hover`, `:active` 等
- 伪元素在 CSS1 中已存在，当时语法是用 `::` 表示，如 `::before` 和 `::after`
- 后来在 CSS3 中修订，伪元素用 `::` 表示，如 `::before` 和 `::after`，以此区分伪元素和伪类
- 由于低版本 IE 对双冒号不兼容，开发者为了兼容性各浏览器，继续使用 `:after` 这种老语法表示伪元素
- 综上所述：`::before` 是 CSS3 中写伪元素的新语法；`::after` 是 CSS1 中存在的、兼容 IE 的老语法

#82 如何修改 Chrome 记住密码后自动填充表单的黄色背景

- 产生原因：由于 Chrome 默认会给自动填充的 input 表单加上 `input:-webkit-autofill` 私有属性造成的
- 解决方案1：在 form 标签上直接关闭了表单的自动填充：`autocomplete="off"`
- 解决方案2：`input:-webkit-autofill { background-color: transparent; }`

`input [type=search]` 搜索框右侧小图标如何美化？

```
input[type="search"]::-webkit-search-cancel-button{
    -webkit-appearance: none;
    height: 15px;
    width: 15px;
    border-radius: 8px;
    background: url("images/searchicon.png") no-repeat 0 0;
    background-size: 15px 15px;
}
```

#83 网站图片文件，如何点击下载？而非点击预览

[下载](#)

#63 iOS safari 如何阻止“橡皮筋效果”

```
$(document).ready(function() {
    var stopScrolling = function(event) {
        event.preventDefault();
    }
    document.addEventListener('touchstart', stopscrolling, false);
    document.addEventListener('touchmove', stopscrolling, false);
}),
```

#84 你对 line-height 是如何理解的

- `line-height` 指一行字的高度，包含了字间距，实际上是下一行基线到上一行基线距离
- 如果一个标签没有定义 `height` 属性，那么其最终表现的高度是由 `line-height` 决定的
- 一个容器没有设置高度，那么撑开容器高度的是 `line-height` 而不是容器内的文字内容
- 把 `line-height` 值设置为 `height` 一样大小的值可以实现单行文字的垂直居中
- `line-height` 和 `height` 都能撑开一个高度，`height` 会触发 `hasLayout`，而 `line-height` 不会

#85 line-height 三种赋值方式有何区别？（带单位、纯数字、百分比）

- 带单位：`px` 是固定值，而 `em` 会参考父元素 `font-size` 值计算自身的行高
- 纯数字：会把比例传递给后代。例如，父级行高为 `1.5`，子元素字体为 `18px`，则子元素行高为 $1.5 * 18 = 27px$
- 百分比：将计算后的值传递给后代

#86 设置元素浮动后，该元素的 display 值会如何变化

设置元素浮动后，该元素的 `display` 值自动变成 `block`

行内块

#87 让页面里的字体变清晰，变细用CSS怎么做？（IOS手机浏览器字体齿轮设置）

```
-webkit-font-smoothing: antialiased;
```

#88 font-style 属性 oblique 是什么意思

`font-style: oblique;` 使没有 `italic` 属性的文字实现倾斜

#89 display:inline-block 什么时候会显示间隙

- 相邻的 `inline-block` 元素之间有换行或空格分隔的情况下会产生间距
- 非 `inline-block` 水平元素设置为 `inline-block` 也会有水平间距
- 可以借助 `vertical-align: top;` 消除垂直间隙
- 可以在父级加 `font-size: 0;` 在子元素里设置需要的字体大小，消除垂直间隙
- 把 `li` 标签写到同一行可以消除垂直间隙，但代码可读性差

#90 一个高度自适应的div，里面有两个div，一个高度100px，希望另一个填满剩下的高度

- 方案1：

`.sub { height: calc(100%-100px); }`

- 方案2：

`.container { position: relative; }`

`.sub { position: absolute; top: 100px; bottom: 0; }`

- 方案3：

`.container { display: flex; flex-direction: column; }`

`.sub { flex: 1; }`

#三、JavaScript

#1 闭包

- 闭包就是能够读取其他函数内部变量的函数
- 闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量，利用闭包可以突破作用域链
- 闭包的特性：
 - 函数内再嵌套函数
 - 内部函数可以引用外层的参数和变量
 - 参数和变量不会被垃圾回收机制回收

说说你对闭包的理解

- 使用闭包主要是为了设计私有的方法和变量。闭包的优点是可以避免全局变量的污染 缺点是闭包会常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。在js中，函数即闭包，只有函数才会产生作用域的概念
- 闭包的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中
- 闭包的另一个用处，是封装对象的私有属性和私有方法
- 好处：能够实现封装和缓存等；
- 坏处：就是消耗内存、不正当使用会造成内存溢出的问题

使用闭包的注意点

- 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露
- 解决方法是，在退出函数之前，将不使用的局部变量全部删除

#2 说说你对作用域链的理解

- 作用域链的作用是保证执行环境里有权访问的变量和函数是有序的，作用域链的变量只能向上访问，变量访问到 window 对象即被终止，作用域链向下访问变量是不被允许的
- 简单的说，作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期

#3 JavaScript原型，原型链？有什么特点？

- 每个对象都会在其内部初始化一个属性，就是 prototype (原型)，当我们访问一个对象的属性时
- 如果这个对象内部不存在这个属性，那么他就会去 prototype 里找这个属性，这个 prototype 又会有自己的 prototype，于是就这样一直找下去，也就是我们平时所说的原型链的概念
- 关系：`instance.constructor.prototype = instance.__proto__`
- 特点：
 - JavaScript 对象是通过引用传递的，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
 - 当我们需要一个属性时，JavaScript 引擎会先看当前对象中是否有这个属性，如果没有的就会查找他的 Prototype 对象是否有这个属性，如此递推下去，一直检索到 Object 内建对象
 - 原型：
 - JavaScript 的所有对象中都包含了一个 `[__proto__]` 内部属性，这个属性所对应的就是该对象的原型

- JavaScript 的函数对象，除了原型 `[__proto__]` 之外，还预置了 `prototype` 属性
- 当函数对象作为构造函数创建实例时，该 `prototype` 属性值将被作为实例对象的原型 `[__proto__]`。
- 原型链：
 - 当一个对象调用的属性/方法自身不存在时，就会去自己 `[__proto__]` 关联的前辈 `prototype` 对象上去找
 - 如果没找到，就会去该 `prototype` 原型 `[__proto__]` 关联的前辈 `prototype` 去找。依次类推，直到找到属性/方法或 `undefined` 为止。从而形成了所谓的“原型链”
- 原型特点：
 - `JavaScript` 对象是通过引用来传递的，当修改原型时，与之相关的对象也会继承这一改变

#4 请解释什么是事件代理

- 事件代理（Event delegation），又称之为事件委托。是 `JavaScript` 中常用绑定事件的常用技巧。顾名思义，“事件代理”即是把原本需要绑定的事件委托给父元素，让父元素担当事件监听的职务。事件代理的原理是DOM元素的事件冒泡。使用事件代理的好处是可以提高性能
- 可以大量节省内存占用，减少事件注册，比如在 `table` 上代理所有 `td` 的 `click` 事件就非常棒
- 可以实现当新增子对象时无需再次对其绑定

#5 Javascript如何实现继承？

- 构造继承
- 原型继承
- 实例继承
- 拷贝继承
- 原型 `prototype` 机制或 `apply` 和 `call` 方法去实现较简单，建议使用构造函数与原型混合方式

```

function Parent(){
    this.name = 'wang';
}

function child(){
    this.age = 28;
}

child.prototype = new Parent(); // 继承了 Parent，通过原型

var demo = new child();
alert(demo.age);
alert(demo.name); // 得到被继承的属性

```

#6 谈谈This对象的理解

- `this` 总是指向函数的直接调用者（而非间接调用者）
- 如果有 `new` 关键字，`this` 指向 `new` 出来的那个对象
- 在事件中，`this` 指向触发这个事件的对象，特殊的是，IE 中的 `attachEvent` 中的 `this` 总是指向全局对象 `window`

绑定

#7 事件模型

w3c 中定义事件的发生经历三个阶段：捕获阶段（capturing）、目标阶段（targetin）、冒泡阶段（bubbling）

- 冒泡型事件：当你使用事件冒泡时，子级元素先触发，父级元素后触发
- 捕获型事件：当你使用事件捕获时，父级元素先触发，子级元素后触发
- DOM 事件流：同时支持两种事件模型：捕获型事件和冒泡型事件
- 阻止冒泡：在 w3c 中，使用 stopPropagation() 方法；在IE下设置 cancelBubble = true
- 阻止捕获：阻止事件的默认行为，例如 click - <a> 后的跳转。在 w3c 中，使用 preventDefault() 方法，在 IE 下设置 window.event.returnValue = false

#8 new操作符具体干了什么呢？

- 创建一个空对象，并且 this 变量引用该对象，同时还继承了该函数的原型
- 属性和方法被加入到 this 引用的对象中
- 新创建的对象由 this 所引用，并且最后隐式的返回 this

#9 Ajax原理

- Ajax 的原理简单来说是在用户和服务器之间加了一个中间层（AJAX 引擎），通过 XMLHttpRequest 对象来向服务器发异步请求，从服务器获得数据，然后用 javascript 来操作 DOM 而更新页面。使用户操作与服务器响应异步化。这其中最关键的一步就是从服务器获得请求数据
- Ajax 的过程只涉及 JavaScript、XMLHttpRequest 和 DOM。XMLHttpRequest 是 ajax 的核心机制

```
/** 1. 创建连接 */
var xhr = null;
xhr = new XMLHttpRequest()
/** 2. 连接服务器 */
xhr.open('get', url, true)
/** 3. 发送请求 */
xhr.send(null);
/** 4. 接受请求 */
xhr.onreadystatechange = function(){
    if(xhr.readyState == 4){
        if(xhr.status == 200){
            success(xhr.responseText);
        } else {
            /** false */
            fail && fail(xhr.status);
        }
    }
}
```

ajax 有那些优缺点？

- 优点：
 - 通过异步模式，提升了用户体验。
 - 优化了浏览器和服务器之间的传输，减少不必要的数据往返，减少了带宽占用。
 - Ajax 在客户端运行，承担了一部分本来由服务器承担的工作，减少了大用户量下的服务器负载。
 - Ajax 可以实现动态不刷新（局部刷新）
- 缺点：

- 安全问题 AJAX 暴露了与服务器交互的细节。
- 对搜索引擎的支持比较弱。
- 不容易调试。

#10 如何解决跨域问题?

首先了解下浏览器的同源策略 同源策略 /SOP (Same origin policy) 是一种约定, 由Netscape公司1995年引入浏览器, 它是浏览器最核心也最基本的安全功能, 如果缺少了同源策略, 浏览器很容易受到 XSS、CSRF 等攻击。所谓同源是指“协议+域名+端口”三者相同, 即便两个不同的域名指向同一个ip地址, 也非同源

那么怎样解决跨域问题的呢?

- 通过jsonp跨域

```
var script = document.createElement('script');
script.type = 'text/javascript';

// 传参并指定回调执行函数为onBack
script.src = 'http://www.....:8080/login?user=admin&callback=onBack';
document.head.appendChild(script);

// 回调执行函数
function onBack(res) {
  alert(JSON.stringify(res));
}
```

- document.domain + iframe跨域

此方案仅限主域相同, 子域不同的跨域应用场景

1.) 父窗口: (<http://www.domain.com/a.html>)

```
<iframe id="iframe" src="http://child.domain.com/b.html"></iframe>
<script>
  document.domain = 'domain.com';
  var user = 'admin';
</script>
```

2.) 子窗口: (<http://child.domain.com/b.html>)

```
document.domain = 'domain.com';
// 获取父窗口中变量
alert('get js data from parent ---> ' + window.parent.user);
```

- nginx代理跨域
- nodejs中间件代理跨域
- 后端在头部信息里面设置安全域名

#11 模块化开发怎么做?

- 立即执行函数, 不暴露私有成员

```

var module1 = (function(){
    var _count = 0;
    var m1 = function(){
        //...
    };
    var m2 = function(){
        //...
    };
    return {
        m1 : m1,
        m2 : m2
    };
})();

```

#12 异步加载JS的方式有哪些?

- ✓ 设置 `<script>` 属性 `async="async"` (一旦脚本可用，则会异步执行)
- ✓ 动态创建 `script DOM`: `document.createElement('script');`
- `XMLHttpRequest` 脚本注入
- 异步加载库 `LABjs`
- 模块加载器 `Sea.js`

#13 那些操作会造成内存泄漏?

JavaScript 内存泄露指对象在不需要使用它时仍然存在，导致占用的内存不能使用或回收

- ✓ 未使用 `var` 声明的全局变量
- ✓ 闭包函数(Closures)
 - 循环引用(两个对象相互引用)
 - 控制台日志(`console.log`)
 - 移除存在绑定事件的DOM元素(IE)
 - ✓ `setTimeout` 的第一个参数使用字符串而非函数的话，会引发内存泄漏
 - 垃圾回收器定期扫描对象，并计算引用了每个对象的其他对象的数量。如果一个对象的引用数量为 0 (没有其他对象引用过该对象)，或对该对象的唯一引用是循环的，那么该对象的内存即可回收

#14 XML和JSON的区别?

- 数据体积方面
 - JSON 相对于 XML 来讲，数据的体积小，传递的速度更快些。
- 数据交互方面
 - JSON 与 Javascript 的交互更加方便，更容易解析处理，更好的数据交互
- 数据描述方面
 - JSON 对数据的描述性比 XML 较差
- 传输速度方面
 - JSON 的速度要远远快于 XML

#15 谈谈你对webpack的看法

- WebPack 是一个模块打包工具，你可以使用 WebPack 管理你的模块依赖，并编译输出模块们所需的静态文件。它能够很好地管理、打包 web 开发中所用到的 HTML、Javascript、css 以及各种静态文件（图片、字体等），让开发过程更加高效。对于不同类型的资源，webpack 有对应的模块加载器。webpack 模块打包器会分析模块间的依赖关系，最后生成了优化且合并后的静态资源

#16 说说你对AMD和Commonjs的理解

- CommonJS 是服务器端模块的规范，Node.js 采用了这个规范。CommonJS 规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。AMD 规范则是非同步加载模块，允许指定回调函数
- AMD 推荐的风格通过返回一个对象做为模块对象，CommonJS 的风格通过对 module.exports 或 exports 的属性赋值来达到暴露模块对象的目的

#17 常见web安全及防护原理

- sql注入原理
 - 就是通过把 SQL 命令插入到 Web 表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令
- 总的来说有以下几点
 - 永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对单引号和双 "-" 进行转换等
 - 永远不要使用动态拼装SQL，可以使用参数化的SQL或者直接使用存储过程进行数据查询存取
 - 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接
 - 不要把机密信息明文存放，请加密或者 hash 掉密码和敏感的信息

XSS原理及防范

- XSS(cross-site scripting) 攻击指的是攻击者往 web 页面里插入恶意 html 标签或者 javascript 代码。比如：攻击者在论坛中放一个看似安全的链接，骗取用户点击后，窃取 cookie 中的用户私密信息；或者攻击者在论坛中加一个恶意表单，当用户提交表单的时候，却把信息传送到攻击者的服务器中，而不是用户原本以为的信任站点

XSS防范方法

- 首先代码里对用户输入的地方和变量都需要仔细检查长度和对 "<" , ">" , ";" , "'''" 等字符做过滤；其次任何内容写到页面之前都必须加以 encode，避免不小心把 html tag 弄出来。这一个层面做好，至少可以堵住超过一半的XSS 攻击

XSS与CSRF有什么区别吗？

- XSS 是获取信息，不需要提前知道其他用户页面的代码和数据包。CSRF 是代替用户完成指定的动作，需要知道其他用户页面的代码和数据包。要完成一次 CSRF 攻击，受害者必须依次完成两个步骤
- 登录受信任网站 A，并在本地生成 cookie
- 在不登出 A 的情况下，访问危险网站 B

CSRF的防御

- 服务端的 CSRF 方式方法很多样，但总的思想都是一致的，就是在客户端页面增加伪随机数
- 通过验证码的方法

#18 用过哪些设计模式？

- 工厂模式：
 - 工厂模式解决了重复实例化的问题，但还有一个问题，那就是识别问题，因为根本无法
 - 主要好处就是可以消除对象间的耦合，通过使用工厂方法而不是 new 关键字
- 构造函数模式
 - 使用构造函数的方法，即解决了重复实例化的问题，又解决了对象识别的问题，该模式与工厂模式的不同之处在于

- 直接将属性和方法赋值给 `this` 对象;

#19 为什么要有同源限制?

- 同源策略指的是：协议，域名，端口相同，同源策略是一种安全协议
- 举例说明：比如一个黑客程序，他利用 `Iframe` 把真正的银行登录页面嵌到他的页面上，当你使用真实的用户名，密码登录时，他的页面就可以通过 `Javascript` 读取到你的表单中 `input` 中的内容，这样用户名，密码就轻松到手了。

#20 `offsetWidth/offsetHeight,clientWidth/clientHeight` 与 `scrollWidth/scrollHeight` 的区别

- `offsetWidth/offsetHeight` 返回值包含 content + padding + border，效果与 `e.getBoundingClientRect()` 相同
- `clientWidth/clientHeight` 返回值只包含 content + padding，如果有滚动条，也不包含滚动条
- `scrollWidth/scrollHeight` 返回值包含 content + padding + 溢出内容的尺寸

#21 javascript有哪些方法定义对象

- 对象字面量：`var obj = {};`
- 构造函数：`var obj = new Object();`
- `Object.create(): var obj = Object.create(Object.prototype);`

#22 常见兼容性问题?

- `png24` 位的图片在 `iE6` 浏览器上出现背景，解决方案是做成 `PNG8`
- 浏览器默认的 `margin` 和 `padding` 不同。解决方案是加一个全局的 `*{margin:0;padding:0;}` 来统一，但是全局效率很低，一般是如下这样解决：

```
body,ul,li,ol,d1,dt,dd,form,input,h1,h2,h3,h4,h5,h6,p{
margin:0;
padding:0;
}
```

- `IE` 下，`event` 对象有 `x, y` 属性，但是没有 `pageX, pageY` 属性
- `Firefox` 下，`event` 对象有 `pageX, pageY` 属性，但是没有 `x, y` 属性.

#23 说说你对promise的了解

- 依照 `Promise/A+` 的定义，`Promise` 有四种状态：
 - `pending`: 初始状态，非 `fulfilled` 或 `rejected`.
 - `fulfilled`: 成功的操作.
 - `rejected`: 失败的操作.
 - `settled`: `Promise` 已被 `fulfilled` 或 `rejected`，且不是 `pending`
- 另外，`fulfilled` 与 `rejected` 一起合称 `settled`
- `Promise` 对象用来进行延迟(`deferred`)和异步(`asynchronous`)计算

Promise 的构造函数

- 构造一个 `Promise`，最基本的用法如下：

```

var promise = new Promise(function(resolve, reject) {
    if (...) { // succeed
        resolve(result);
    } else { // fails
        reject(Error(errorMessage));
    }
});

```

- `Promise` 实例拥有 `then` 方法（具有 `then` 方法的对象，通常被称为 `thenable`）。它的使用方法如下：

```
promise.then(onFulfilled, onRejected)
```

- 接收两个函数作为参数，一个在 `fulfilled` 的时候被调用，一个在 `rejected` 的时候被调用，接收参数就是 `future`，`onFulfilled` 对应 `resolve`，`onRejected` 对应 `reject`

#24 你觉得jQuery源码有哪些写的好地方

- `jquery` 源码封装在一个匿名函数的自执行环境中，有助于防止变量的全局污染，然后通过传入 `window` 对象参数，可以使 `window` 对象作为局部变量使用，好处是当 `jquery` 中访问 `window` 对象的时候，就不用将作用域链退回到顶层作用域了，从而可以更快的访问 `window` 对象。同样，传入 `undefined` 参数，可以缩短查找 `undefined` 时的作用域链
- `jquery` 将一些原型属性和方法封装在了 `jquery.prototype` 中，为了缩短名称，又赋值给了 `jquery.fn`，这是很形象的写法
- 有一些数组或对象的方法经常能使用到，`jquery` 将其保存为局部变量以提高访问速度
- `jquery` 实现的链式调用可以节约代码，所返回的都是同一个对象，可以提高代码效率

#25 vue、react、angular

- `vue.js` 一个用于创建 `web` 交互界面的库，是一个精简的 `MVVM`。它通过双向数据绑定把 `view` 层和 `Model` 层连接了起来。实际的 `DOM` 封装和输出格式都被抽象为了 `Directives` 和 `Filters`
- `AngularJS` 是一个比较完善的前端 `MVVM` 框架，包含模板，数据双向绑定，路由，模块化，服务，依赖注入等所有功能，模板功能强大丰富，自带了丰富的 `Angular` 指令
- `react` `React` 仅仅是 `VIEW` 层是 `facebook` 公司。推出的一个用于构建 `ui` 的一个库，能够实现服务器端的渲染。用了 `virtual dom`，所以性能很好。

#26 Node的应用场景

- 特点：
 - 1、它是一个 `Javascript` 运行环境
 - 2、依赖于 `Chrome v8` 引擎进行代码解释
 - 3、事件驱动
 - 4、非阻塞 `I/O`
 - 5、单进程，单线程
- 优点：
 - 高并发（最重要的优点）
- 缺点：

- 1、只支持单核 CPU，不能充分利用 CPU
- 2、可靠性低，一旦代码某个环节崩溃，整个系统都崩溃

#27 谈谈你对AMD、CMD的理解

- CommonJS 是服务器端模块的规范，Node.js 采用了这个规范。CommonJS 规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。AMD 规范则是非同步加载模块，允许指定回调函数
- AMD 推荐的风格通过返回一个对象做为模块对象，CommonJS 的风格通过对 module.exports 或 exports 的属性赋值来达到暴露模块对象的目的

es6模块 CommonJS、AMD、CMD

- CommonJS 的规范中，每个 JavaScript 文件就是一个独立的模块上下文（module context），在这个上下文中默认创建的属性都是私有的。也就是说，在一个文件定义的变量（还包括函数和类），都是私有的，对其他文件是不可见的。
- CommonJS 是同步加载模块，在浏览器中会出现堵塞情况，所以不适用
- AMD 异步，需要定义回调 define 方式
- es6 一个模块就是一个独立的文件，该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用 export 关键字输出该变量 es6 还可以导出类、方法，自动适用严格模式

#28 那些操作会造成内存泄漏

- 内存泄漏指任何对象在您不再拥有或需要它之后仍然存在
- setTimeout 的第一个参数使用字符串而非函数的话，会引发内存泄漏
- 闭包、控制台日志、循环（在两个对象彼此引用且彼此保留时，就会产生一个循环）

#29 web开发中会话跟踪的方法有哪些

- cookie
- session
- url 重写
- 隐藏 input
- ip 地址

#30 JS的基本数据类型和引用数据类型

- 基本数据类型：undefined、null、boolean、number、string、symbol
- 引用数据类型：object、array、function

#31 介绍js有哪些内置对象

- object 是 JavaScript 中所有对象的父对象
- 数据封装类对象：Object、Array、Boolean、Number 和 String
- 其他对象：Function、Arguments、Math、Date、RegExp、Error

#32 说几条写JavaScript的基本规范

- 不要在同一行声明多个变量
- 请使用 ===/!== 来比较 true/false 或者数值
- 使用对象字面量替代 new Array 这种形式
- 不要使用全局函数
- switch 语句必须带有 default 分支

- `If` 语句必须使用大括号
- `for-in` 循环中的变量 应该使用 `var` 关键字明确限定作用域，从而避免作用域污染

#33 JavaScript有几种类型的值

- 栈：原始数据类型 (`undefined`, `null`, `Boolean`, `Number`、`String`)
- 堆：引用数据类型 (对象、数组和函数)
- 两种类型的区别是：存储位置不同；
- 原始数据类型直接存储在栈(`stack`)中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储；
- 引用数据类型存储在堆(`heap`)中的对象，占据空间大、大小不固定，如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其
- 在栈中的地址，取得地址后从堆中获得实体

#34 javascript创建对象的几种方式

`javascript` 创建对象简单的说，无非就是使用内置对象或各种自定义对象，当然还可以用 `JSON`；但写法有很多种，也能混合使用

- 对象字面量的方式

```
person={firstname:"Mark", lastname:"Yun", age:25, eyecolor:"black"};
```

- 用 `function` 来模拟无参的构造函数

```
function Person(){}  
var person=new Person(); // 定义一个function，如果使用new"实例化"，该function可以看作是一个class  
person.name="Mark";  
person.age="25";  
person.work=function(){  
    alert(person.name+" hello...");  
}  
person.work();
```

- 用 `function` 来模拟参构造函数来实现（用 `this` 关键字定义构造的上下文属性）

```
function Pet(name,age,hobby){  
    this.name=name; // this作用域：当前对象  
    this.age=age;  
    this.hobby=hobby;  
    this.eat=function(){  
        alert("我叫"+this.name+", 我喜欢"+this.hobby+", 是个程序员");  
    }  
}  
var maidou =new Pet("麦兜",25,"coding"); // 实例化、创建对象  
maidou.eat(); // 调用eat方法
```

- 用工厂方式来创建（内置对象）

```

var wcDog =new Object();
wcDog.name="旺财";
wcDog.age=3;
wcDog.work=function(){
    alert("我是"+wcDog.name+",汪汪汪.....");
}
wcDog.work();

```

- 用原型方式来创建

```

function Dog(){}
Dog.prototype.name="旺财";
Dog.prototype.eat=function(){
    alert(this.name+"是个吃货");
}
var wangcai =new Dog();
wangcai.eat();

```

- 用混合方式来创建

```

function Car(name,price){
    this.name=name;
    this.price=price;
}
Car.prototype.sell=function(){
    alert("我是"+this.name+", 我现在卖"+this.price+"万元");
}
var camry =new Car("凯美瑞",27);
camry.sell();

```

#35 eval是做什么的

- 它的功能是把对应的字符串解析成js代码并运行
- 应该避免使用 eval，不安全，非常耗性能（2次，一次解析成 js 语句，一次执行）
- 由 JSON 字符串转换为JSON对象的时候可以用 eval， var obj =eval('('+ str +')')

#36 null, undefined 的区别

- `undefined` 表示不存在这个值。
- `undefined` 是一个表示“无”的原始值或者说表示“缺少值”，就是此处应该有一个值，但是还没有定义。当尝试读取时会返回 `undefined`
- 例如变量被声明了，但没有赋值时，就等于 `undefined`
- `null` 表示一个对象被定义了，值为“空值”
- `null` 是一个对象（空对象，没有任何属性和方法）
- 例如作为函数的参数，表示该函数的参数不是对象；
- 在验证 `null` 时，一定要使用 `==`，因为 `==` 无法分别 `null` 和 `undefined`

↑ #37 ["1", "2", "3"].map(parseInt) 答案是多少

- [1, NaN, NaN] 因为 `parseInt` 需要两个参数 (`val, radix`)，其中 `radix` 表示解析时用的基数。
- `map` 传了 3 个 `(element, index, array)`，对应的 `radix` 不合法导致解析失败。

#38 javascript 代码中的“use strict”;是什么意思

- `use strict` 是一种 ECMAScript 5 添加的（严格）运行模式，这种模式使得 Javascript 在更严格的条件下运行，使 JS 编码更加规范化的模式，消除 Javascript 语法的一些不合理、不严谨之处，减少一些怪异行为

#39 JSON 的了解

- `JSON`(JavaScript Object Notation) 是一种轻量级的数据交换格式
- 它是基于 `JavaScript` 的一个子集。数据格式简单，易于读写，占用带宽小
- `JSON` 字符串转换为 `JSON` 对象：

```
var obj = eval('('+ str +')');
var obj = str.parseJSON();
var obj = JSON.parse(str);
```

- `JSON` 对象转换为 `JSON` 字符串：

```
var last=obj.toJSONString();
var last=JSON.stringify(obj);
```

#40 js延迟加载的方式有哪些

- 设置 `<script>` 属性 `defer="defer"` （脚本将在页面完成解析时执行）
- 动态创建 `script DOM`：`document.createElement('script');`
- `XMLHttpRequest` 脚本注入
- 延迟加载工具 `LazyLoad`

#41 同步和异步的区别

- 同步：浏览器访问服务器请求，用户看得到页面刷新，重新发请求，等请求完，页面刷新，新内容出现，用户看到新内容进行下一步操作
- 异步：浏览器访问服务器请求，用户正常操作，浏览器后端进行请求。等请求完，页面不刷新，新内容也会出现，用户看到新内容

#42 渐进增强和优雅降级

- 渐进增强：针对低版本浏览器进行构建页面，保证最基本的功能，然后再针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。
- 优雅降级：一开始就构建完整的功能，然后再针对低版本浏览器进行兼容

#43 defer和async

- `defer` 并行加载 `js` 文件，会按照页面上 `script` 标签的顺序执行
- `async` 并行加载 `js` 文件，下载完成立即执行，不会按照页面上 `script` 标签的顺序执行

#44 说说严格模式的限制

- 变量必须声明后再使用
- 函数的参数不能有同名属性，否则报错
- 不能使用 `with` 语句
- 不能对只读属性赋值，否则报错
- 不能使用前缀 `0` 表示八进制数，否则报错

- 不能删除不可删除的属性，否则报错
- 不能删除变量 `delete prop`，会报错，只能删除属性 `delete global[prop]`
- `eval` 不会在它的外层作用域引入变量
- `eval` 和 `arguments` 不能被重新赋值
- `arguments` 不会自动反映函数参数的变化
- 不能使用 `arguments.callee`
- 不能使用 `arguments.caller`
- 禁止 `this` 指向全局对象
- 不能使用 `fn.caller` 和 `fn.arguments` 获取函数调用的堆栈
- 增加了保留字（比如 `protected`、`static` 和 `interface`）

Q #45 attribute和property的区别是什么

- `attribute` 是 `dom` 元素在文档中作为 `html` 标签拥有的属性；
- `property` 就是 `dom` 元素在 `js` 中作为对象拥有的属性。
- 对于 `html` 的标准属性来说，`attribute` 和 `property` 是同步的，是会自动更新的
- 但是对于自定义的属性来说，他们是不同步的

#46 谈谈你对ES6的理解

- 新增模板字符串（为 `Javascript` 提供了简单的字符串插值功能）
- 箭头函数
- `for-of`（用来遍历数据—例如数组中的值。）
- `arguments` 对象可被不定参数和默认参数完美代替。
- ES6 将 promise 对象纳入规范，提供了原生的 Promise 对象。
- 增加了 let 和 const 命令，用来声明变量。
- 增加了块级作用域
- `let` 命令实际上就增加了块级作用域。
- 还有就是引入 module 模块的概念

#47 ECMAScript6 怎么写class么

- 这个语法糖可以让有 `oop` 基础的人更快上手 `js`，至少是一个官方的实现了
- 但对熟悉 `js` 的人来说，这个东西没啥大影响；一个 `Object.create()` 搞定继承，比 `class` 简洁清晰的多

#48 什么是面向对象编程及面向过程编程，它们的异同和优缺点

- 面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了
- 面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描述某个事物在整个解决问题的步骤中的行为
- 面向对象是以功能来划分问题，而不是步骤

#49 面向对象编程思想

- 基本思想是使用对象，类，继承，封装等基本概念来进行程序设计
- 优点
 - 易维护
 - 采用面向对象思想设计的结构，可读性高，由于继承的存在，即使改变需求，那么维护也只是在局部模块，所以维护起来是非常方便和较低成本的
 - 易扩展

- 开发工作的重用性、继承性高，降低重复工作量。
- 缩短了开发周期

#50 对web标准、可用性、可访问性的理解

- 可用性 (Usability) : 产品是否容易上手，用户能否完成任务，效率如何，以及这过程中用户的主观感受。是从用户的角度来看产品的质量。可用性好意味着产品质量高，是企业的核心竞争力
- 可访问性 (Accessibility) : Web内容对于残障用户的可阅读和可理解性
- 可维护性 (Maintainability) : 一般包含两个层次，一是当系统出现问题时，快速定位并解决问题的成本，成本低则可维护性好。二是代码是否容易被人理解，是否容易修改和增强功能。

#51 如何通过JS判断一个数组

- `instanceof`

方法

- `instanceof` 运算符是用来测试一个对象是否在其原型链原型构造函数的属性

```
var arr = [];
arr instanceof Array; // true
```

- `constructor`

方法

- `constructor` 属性返回对创建此对象的数组函数的引用，就是返回对象相对应的构造函数

```
var arr = [];
arr.constructor == Array; // true
```

- 最简单的方法
 - 这种写法，是 `jquery` 正在使用的

```
object.prototype.toString.call(value) == '[object Array]'
// 利用这个方法，可以写一个返回数据类型的方法
var isType = function (obj) {
  return Object.prototype.toString.call(obj).slice(8,-1);
}
```

- ES5 新增方法 `isArray()`

```
var a = new Array(123);
var b = new Date();
console.log(Array.isArray(a)); // true
console.log(Array.isArray(b)); // false
```

#52 谈一谈let与var的区别

- let 命令不存在变量提升，如果在 let 前使用，会导致报错
- 如果块区中存在 let 和 const 命令，就会形成封闭作用域
- 不允许重复声明，因此，不能在函数内部重新声明参数

#53 map与forEach的区别

- forEach 方法，是最基本的方法，就是遍历与循环，默认有3个传参：分别是遍历的数组内容 item、数组索引 index、和当前遍历数组 Array
- map 方法，基本用法与 forEach 一致，但是不同的，它会返回一个新的数组，所以在callback需要有 return 值，如果没有，会返回 undefined

#54 谈一谈你理解的函数式编程

- 简单说，“函数式编程”是一种“编程范式”（programming paradigm），也就是如何编写程序的方法论
- 它具有以下特性：闭包和高阶函数、惰性计算、递归、函数是“第一等公民”、只用“表达式”

#55 谈一谈箭头函数与普通函数的区别？

- 函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象
- 不可以当作构造函数，也就是说，不可以使用 new 命令，否则会抛出一个错误
- 不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 Rest 参数代替
- 不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数

#56 谈一谈函数中this的指向

- this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁，实际上this的最终指向的是那个调用它的对象
- 《javascript语言精髓》中大概概括了4种调用方式：
- 方法调用模式
- 函数调用模式
- 构造器调用模式

```
graph LR
A-->B
```

- apply/call调用模式

#57 异步编程的实现方式

- 回调函数
 - 优点：简单、容易理解
 - 缺点：不利于维护，代码耦合高
- 事件监听(采用时间驱动模式，取决于某个事件是否发生):
 - 优点：容易理解，可以绑定多个事件，每个事件可以指定多个回调函数
 - 缺点：事件驱动型，流程不够清晰
- 发布/订阅(观察者模式)
 - 类似于事件监听，但是可以通过‘消息中心’，了解现在有多少发布者，多少订阅者
- Promise对象
 - 优点：可以利用then方法，进行链式写法；可以书写错误时的回调函数；

- 缺点：编写和理解，相对比较难
- Generator函数
 - 优点：函数体内外的数据交换、错误处理机制
 - 缺点：流程管理不方便
- async函数
 - 优点：内置执行器、更好的语义、更广的适用性、返回的是Promise、结构清晰。
 - 缺点：错误处理机制

#58 对原生Javascript了解程度

- 数据类型、运算、对象、Function、继承、闭包、作用域、原型链、事件、`RegExp`、`JSON`、`Ajax`、`DOM`、`BOM`、内存泄漏、跨域、异步装载、模板引擎、前端 `MVC`、路由、模块化、`Canvas`、`ECMAScript`

#59 Js动画与CSS动画区别及相应实现

- `CSS3`

的动画的优点

- 在性能上会稍微好一些，浏览器会对 `CSS3` 的动画做一些优化
- 代码相对简单
- 缺点
 - 在动画控制上不够灵活
 - 兼容性不好
- `JavaScript` 的动画正好弥补了这两个缺点，控制能力很强，可以单帧的控制、变换，同时写得好完全可以兼容 `IE6`，并且功能强大。对于一些复杂控制的动画，使用 `javascript` 会比较靠谱。而在实现一些小的交互动效的时候，就多考虑考虑 `CSS` 吧

#60 JS 数组和对象的遍历方式，以及几种方式的比较

通常我们会用循环的方式来遍历数组。但是循环是导致js 性能问题的原因之一。一般我们会采用下几种方式来进行数组的遍历

- `for in` 循环
- `for` 循环
- `forEach`
 - 这里的 `forEach` 回调中两个参数分别为 `value`, `index`
 - `forEach` 无法遍历对象
 - IE不支持该方法；`Firefox` 和 `chrome` 支持
 - `forEach` 无法使用 `break`, `continue` 跳出循环，且使用 `return` 是跳过本次循环
- 这两种方法应该非常常见且使用很频繁。但实际上，这两种方法都存在性能问题
- 在方式一中，`for-in` 需要分析出 `array` 的每个属性，这个操作性能开销很大。用在 `key` 已知的数组上是非常不划算的。所以尽量不要用 `for-in`，除非你不清楚要处理哪些属性，例如 `JSON` 对象这样的情况
- 在方式2中，循环每进行一次，就要检查一下数组长度。读取属性（数组长度）要比读局部变量慢，尤其是当 `array` 里存放的都是 `DOM` 元素，因为每次读取都会扫描一遍页面上的选择器相关元素，速度会大大降低

#61 gulp是什么

- `gulp` 是前端开发过程中一种基于流的代码构建工具，是自动化项目的构建利器；它不仅能对网站资源进行优化，而且在开发过程中很多重复的任务能够使用正确的工具自动完成
- Gulp的核心概念：流
- 流，简单来说就是建立在面向对象基础上的一种抽象的处理数据的工具。在流中，定义了一些处理数据的基本操作，如读取数据，写入数据等，程序员是对流进行所有操作的，而不用关心流的另一头数据的真正流向
- gulp正是通过流和代码优于配置的策略来尽量简化任务编写的工作
- Gulp的特点：
 - 易于使用：通过代码优于配置的策略，gulp 让简单的任务简单，复杂的任务可管理
 - 构建快速 利用 `Node.js` 流的威力，你可以快速构建项目并减少频繁的 `IO` 操作
 - 易于学习 通过最少的 `API`，掌握 `gulp` 毫不费力，构建工作尽在掌握：如同一系列流管道



#62 说一下Vue的双向绑定数据的原理

- `vue.js` 则是采用数据劫持结合发布者-订阅者模式的方式，通过 `Object.defineProperty()` 来劫持各个属性的 `setter`, `getter`，在数据变动时发布消息给订阅者，触发相应的监听回调

#63 事件的各个阶段

- 1：捕获阶段 --> 2：目标阶段 --> 3：冒泡阶段
- `document` --> `target` 目标 ----> `document`
- 由此，

`addEventListener`

的第三个参数设置为

`true`

和

`false`

的区别已经非常清晰了

- `true` 表示该元素在事件的“捕获阶段”（由外往内传递时）响应事件
- `false` 表示该元素在事件的“冒泡阶段”（由内向外传递时）响应事件

#64 let var const

`let`

- 允许你声明一个作用域被限制在块级中的变量、语句或者表达式
- let绑定不受变量提升的约束，这意味着let声明不会被提升到当前
- 该变量处于从块开始到初始化处理的“暂存死区”

`var`

- 声明变量的作用域限制在其声明位置的上下文中，而非声明变量总是全局的

- 由于变量声明（以及其他声明）总是在任意代码执行之前处理的，所以在代码中的任意位置声明变量总是等效于在代码开头声明

`const`

- 声明创建一个值的只读引用（即指针）
- 基本数据当值发生改变时，那么其对应的指针也将发生改变，故造成 `const` 声明基本数据类型时再将其值改变时，将会造成报错，例如 `const a = 3; a = 5` 时将会报错
- 但是如果是复合类型时，如果只改变复合类型的其中某个 `value` 项时，将还是正常使用

#65 快速的让一个数组乱序

```
var arr = [1,2,3,4,5,6,7,8,9,10];
arr.sort(function(){
  return Math.random() - 0.5;
})
console.log(arr);
```

#66 如何渲染几万条数据并不卡住界面

这道题考察了如何在不卡住页面的情况下渲染数据，也就是说不能一次性将几万条都渲染出来，而应该一次渲染部分 `DOM`，那么就可以通过 `requestAnimationFrame` 来每 `16 ms` 刷新一次

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <ul>控件</ul>
  <script>
    setTimeout(() => {
      // 插入十万条数据
      const total = 100000
      // 一次插入 20 条，如果觉得性能不好就减少
      const once = 20
      // 渲染数据总共需要几次
      const loopCount = total / once
      let countOfRender = 0
      let ul = document.querySelector("ul");
      function add() {
        // 优化性能，插入不会造成回流
        const fragment = document.createDocumentFragment();
        for (let i = 0; i < once; i++) {
          const li = document.createElement("li");
          li.innerText = Math.floor(Math.random() * total);
          fragment.appendChild(li);
        }
        ul.appendChild(fragment);
        countOfRender += 1;
        loop();
      }
      function loop() {
      }
    })
  </script>
</body>
</html>
```

```

        if (countOfRender < loopCount) {
            window.requestAnimationFrame(add);
        }
    }
    loop();
}, 0);
</script>
</body>
</html>

```

#67 希望获取到页面中所有的checkbox怎么做?

不使用第三方框架

```

var domList = document.getElementsByTagName('input')
var checkBoxList = [];
var len = domList.length; //缓存到局部变量
while (len--) { //使用while的效率会比for循环更高
    if (domList[len].type == 'checkbox') {
        checkBoxList.push(domList[len]);
    }
}

```

#68 怎样添加、移除、移动、复制、创建和查找节点

创建新节点

```

createDocumentFragment() //创建一个DOM片段
createElement() //创建一个具体的元素
createTextNode() //创建一个文本节点

```

添加、移除、替换、插入

```

appendChild() //添加
removeChild() //移除
replaceChild() //替换
insertBefore() //插入

```

查找

```

getElementsByName() //通过标签名称
getElementsByName() //通过元素的Name属性的值
getElementById() //通过元素Id, 唯一性

```

#69 正则表达式

正则表达式构造函数 var reg=new RegExp("xxx") 与正则表达字面量 var reg=/有什么不同?
匹配邮箱的正则表达式?

- 当使用 RegExp() 构造函数的时候，不仅需要转义引号（即 \" 表示），并且还需要双反斜杠（即 \\ 表示一个 \）。使用正则表达字面量的效率更高

邮箱的正则匹配：

```
var regMail = /^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+\.(a-zA-Z0-9_-){2,3}\{1,2\}$/;
```

#70 Javascript中callee和caller的作用?

- `caller`是返回一个对函数的引用，该函数调用了当前函数；
- `callee`是返回正在被执行的 `function` 函数，也就是所指定的 `function` 对象的正文

那么问题来了？如果一对兔子每月生一对兔子；一对新生兔，从第二个月起就开始生兔子；假定每对兔子都是一雌一雄，试问一对兔子，第n个月能繁殖成多少对兔子？（使用 `callee` 完成）

```
var result=[];
function fn(n){ //典型的斐波那契数列
    if(n==1){
        return 1;
    }else if(n==2){
        return 1;
    }else{
        if(result[n]){
            return result[n];
        }else{
            //argument.callee()表示fn()
            result[n]=arguments.callee(n-1)+arguments.callee(n-2);
            return result[n];
        }
    }
}
```

#71 window.onload和\$(document).ready

原生JS的 `window.onload` 与 Jquery 的 `$(document).ready(function() {})` 有什么不同？如何用原生JS实现jq的 `ready` 方法？

- `window.onload()` 方法是必须等到页面内包括图片的所有元素加载完毕后才能执行。
- `$(document).ready()` 是 DOM 结构绘制完毕后就执行，不必等到加载完毕

```
function ready(fn){
    if(document.addEventListener) { //标准浏览器
        document.addEventListener('DOMContentLoaded', function() {
            //注销事件，避免反复触发
            document.removeEventListener('DOMContentLoaded', arguments.callee,
false);
            fn(); //执行函数
        }, false);
    } else if(document.attachEvent) { //IE
        document.attachEvent('onreadystatechange', function() {
            if(document.readyState == 'complete') {
                document.detachEvent('onreadystatechange', arguments.callee);
                fn(); //函数执行
            }
        });
    }
};
```

#72 addEventListener()和attachEvent()的区别

- `addEventListener()` 是符合W3C规范的标准方法; `attachEvent()` 是IE低版本的非标准方法
- `addEventListener()` 支持事件冒泡和事件捕获; 而 `attachEvent()` 只支持事件冒泡
- `addEventListener()` 的第一个参数中,事件类型不需要添加 `on`; `attachEvent()` 需要添加 `'on'`
- 如果为同一个元素绑定多个事件, `addEventListener()` 会按照事件绑定的顺序依次执行, `attachEvent()` 会按照事件绑定的顺序倒序执行

~~#73 获取页面所有的checkbox~~

```
var resultArr = [];
var input = document.querySelectorAll('input');
for( var i = 0; i < input.length; i++ ) {
    if( input[i].type == 'checkbox' ) {
        resultArr.push( input[i] );
    }
}
//resultArr即中获取到了页面中的所有checkbox
```

#74 数组去重方法总结

方法一、利用ES6 Set去重 (ES6中最常用)

```
function unique (arr) {
    return Array.from(new Set(arr))
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

方法二、利用for嵌套for, 然后splice去重 (ES5中最常用)

```
function unique(arr){
    for(var i=0; i<arr.length; i++){
        for(var j=i+1; j<arr.length; j++){
            if(arr[i]==arr[j]){
                //第一个等同于第二个, splice方法删除第二个
                arr.splice(j,1);
                j--;
            }
        }
    }
    return arr;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
//[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {...}, {...}]      //NaN
和{}没有去重, 两个null直接消失了
```

- 双层循环, 外层循环元素, 内层循环时比较值。值相同时, 则删去这个值。
- 想快速学习更多常用的 ES6 语法

方法三、利用indexOf去重

```

function unique(arr) {
  if (!Array.isArray(arr)) {
    console.log('type error!')
    return
  }
  var array = [];
  for (var i = 0; i < arr.length; i++) {
    if (array.indexOf(arr[i]) === -1) {
      array.push(arr[i])
    }
  }
  return array;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
// [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a",
[...], {...}] //NaN、{}没有去重

```

新建一个空的结果数组，`for` 循环原数组，判断结果数组是否存在当前元素，如果有相同的值则跳过，不相同则 `push` 进数组

方法四、利用sort()

```

function unique(arr) {
  if (!Array.isArray(arr)) {
    console.log('type error!')
    return;
  }
  arr = arr.sort()
  var arrry= [arr[0]];
  for (var i = 1; i < arr.length; i++) {
    if (arr[i] !== arr[i-1]) {
      arrry.push(arr[i]);
    }
  }
  return arrry;
}
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
// [0, 1, 15, "NaN", NaN, NaN, {...}, {...}, "a", false, null, true, "true",
undefined] //NaN、{}没有去重

```

利用 `sort()` 排序方法，然后根据排序后的结果进行遍历及相邻元素比对

方法五、利用对象的属性不能相同的特点进行去重

```

function unique(arr) {
  if (!Array.isArray(arr)) {
    console.log('type error!')
    return
  }
  var arrry= [];
  var obj = {};
  for (var i = 0; i < arr.length; i++) {

```

```

        if (!obj[arr[i]]) {
            arry.push(arr[i])
            obj[arr[i]] = 1
        } else {
            obj[arr[i]]++
        }
    }
    return arry;
}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
//[1, "true", 15, false, undefined, null, NaN, 0, "a", {...}] //两个true直接去掉了，NaN和{}去重

```

方法六、利用includes

```

function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array =[];
    for(var i = 0; i < arr.length; i++) {
        if( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值
            array.push(arr[i]);
        }
    }
    return array
}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}, {...}]
//{}没有去重

```

方法七、利用hasOwnProperty

```

function unique(arr) {
    var obj = {};
    return arr.filter(function(item, index, arr){
        return obj.hasOwnProperty(typeof item + item) ? false : (obj[typeof item
+ item] = true)
    })
}

var arr = [1,1,'true','true',true,true,15,15,false,false,
undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {...}] //所有的都去重了

```

利用 hasOwnProperty 判断是否存在对象属性

方法八、利用filter

```

function unique(arr) {
    return arr.filter(function(item, index, arr) {
        //当前元素，在原始数组中的第一个索引==当前索引值，否则返回当前元素
        return arr.indexOf(item, 0) === index;
    });
}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {}, ...]

```

方法九、利用递归去重

```

function unique(arr) {
    var array= arr;
    var len = array.length;

    array.sort(function(a,b){    //排序后更加方便去重
        return a - b;
    })

    function loop(index){
        if(index >= 1){
            if(array[index] === array[index-1]){
                array.splice(index,1);
            }
            loop(index - 1);    //递归loop，然后数组去重
        }
    }
    loop(len-1);
    return array;
}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {}, null, NaN, NaN, "NaN", 0, "a", {}, ...,
undefined]

```

方法十、利用Map数据结构去重

```

function arrayNonRepeafy(arr) {
    let map = new Map();
    let array = new Array(); // 数组用于返回结果
    for (let i = 0; i < arr.length; i++) {
        if(map .has(arr[i])) { // 如果有该key值
            map .set(arr[i], true);
        } else {
            map .set(arr[i], false); // 如果没有该key值
            array .push(arr[i]);
        }
    }
    return array ;
}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];
console.log(unique(arr))

```

```
//[1, "a", "true", true, 15, false, 1, {}, null, NaN, NaN, "NaN", 0, "a", {}],  
undefined]
```

创建一个空 `Map` 数据结构，遍历需要去重的数组，把数组的每一个元素作为 `key` 存到 `Map` 中。由于 `Map` 中不会出现相同的 `key` 值，所以最终得到的就是去重后的结果

方法十一、利用reduce+includes

```
function unique(arr){  
    return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],  
[]);  
}  
var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined,  
null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{},{}];  
console.log(unique(arr));  
// [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

方法十二、[...new Set(arr)]

```
[...new Set(arr)]  
//代码就是这么少---- (其实，严格来说并不算是一种，相对于第一种方法来说只是简化了代码)
```

#75 (设计题) 想实现一个对页面某个节点的拖曳？如何做？(使用原生JS)

- 给需要拖拽的节点绑定 `mousedown`, `mousemove`, `mouseup` 事件
- `mousedown` 事件触发后，开始拖拽
- `mousemove` 时，需要通过 `event.clientX` 和 `clientY` 获取拖拽位置，并实时更新位置
- `mouseup` 时，拖拽结束
- 需要注意浏览器边界的情况

#76 Javascript全局函数和全局变量

全局变量

- `Infinity` 代表正的无穷大的数值。
- `Nan` 指示某个值是不是数字值。
- `undefined` 指示未定义的值。

全局函数

- `decodeURI()` 解码某个编码的 `URI`。
- `decodeURIComponent()` 解码一个编码的 `URI` 组件。
- `encodeURI()` 把字符串编码为 `URI`。
- `encodeURIComponent()` 把字符串编码为 `URI` 组件。
- `escape()` 对字符串进行编码。
- `eval()` 计算 `JavaScript` 字符串，并把它作为脚本代码来执行。
- `isFinite()` 检查某个值是否为有穷大的数。
- `isNaN()` 检查某个值是否是数字。
- `Number()` 把对象的值转换为数字。
- `parseFloat()` 解析一个字符串并返回一个浮点数。
- `parseInt()` 解析一个字符串并返回一个整数。
- `String()` 把对象的值转换为字符串。

- unescape() 对由 escape() 编码的字符串进行解码

#77 使用js实现一个持续的动画效果

定时器思路

```
var e = document.getElementById('e')
var flag = true;
var left = 0;
setInterval(() => {
    left == 0 ? flag = true : left == 100 ? flag = false : '';
    flag ? e.style.left = `${left++}px` : e.style.left = `${left--}px`;
}, 1000 / 60)
```

requestAnimationFrame

```
//兼容性处理
window.requestAnimFrame = (function(){
    return window.requestAnimationFrame ||
           window.webkitRequestAnimationFrame ||
           window.mozRequestAnimationFrame ||
           function(callback){
               window.setTimeout(callback, 1000 / 60);
           };
})();

var e = document.getElementById("e");
var flag = true;
var left = 0;

function render() {
    left == 0 ? flag = true : left == 100 ? flag = false : '';
    flag ? e.style.left = `${left++}px` :
          e.style.left = `${left--}px`;
}

(function animloop() {
    render();
    requestAnimFrame(animloop);
})();
```

使用css实现一个持续的动画效果

```
animation:mymove 5s infinite;

@keyframes mymove {
    from {top:0px;}
    to {top:200px;}
}
```

- animation-name 规定需要绑定到选择器的 keyframe 名称。
- animation-duration 规定完成动画所花费的时间，以秒或毫秒计。
- animation-timing-function 规定动画的速度曲线。
- animation-delay 规定在动画开始之前的延迟。
- animation-iteration-count 规定动画应该播放的次数。

- `animation-direction` 规定是否应该轮流反向播放动画

#78 封装一个函数，参数是定时器的时间，`.then`执行回调函数

```
function sleep (time) {
    return new Promise((resolve) => setTimeout(resolve, time));
}
```

#79 怎么判断两个对象相等？

```
obj={
  a:1,
  b:2
}
obj2={
  a:1,
  b:2
}
obj3={
  a:1,
  b:'2'
}
```

可以转换为字符串来判断

```
JSON.stringify(obj)==JSON.stringify(obj2); //true
JSON.stringify(obj)==JSON.stringify(obj3); //false
```

#80 项目做过哪些性能优化？

- 减少 `HTTP` 请求数
- 减少 `DNS` 查询
- 使用 `CDN`
- 避免重定向
- 图片懒加载
- 减少 `DOM` 元素数量
- 减少 `DOM` 操作
- 使用外部 `JavaScript` 和 `css`
- 压缩 `JavaScript`、`css`、字体、图片等
- 优化 `css sprite`
- 使用 `iconfont`
- 字体裁剪
- 多域名分发划分内容到不同域名
- 尽量减少 `iframe` 使用
- 避免图片 `src` 为空
- 把样式表放在 `link` 中
- 把 `javascript` 放在页面底部

#81 浏览器缓存

浏览器缓存分为强缓存和协商缓存。当客户端请求某个资源时，获取缓存的流程如下

- 先根据这个资源的一些 `http header` 判断它是否命中强缓存，如果命中，则直接从本地获取缓存资源，不会发请求到服务器；
- 当强缓存没有命中时，客户端会发送请求到服务器，服务器通过另一些 `request header` 验证这个资源是否命中协商缓存，称为 `http` 再验证，如果命中，服务器将请求返回，但不返回资源，而是告诉客户端直接从缓存中获取，客户端收到返回后就会从缓存中获取资源；
- 强缓存和协商缓存共同之处在于，如果命中缓存，服务器都不会返回资源；区别是，强缓存不对发送请求到服务器，但协商缓存会。
- 当协商缓存也没命中时，服务器就会将资源发送回客户端。
- 当 `ctrl+f5` 强制刷新网页时，直接从服务器加载，跳过强缓存和协商缓存；
- 当 `f5` 刷新网页时，跳过强缓存，但是会检查协商缓存；

强缓存

- `Expires` (该字段是 `http1.0` 时的规范，值为一个绝对时间的 `GMT` 格式的时间字符串，代表缓存资源的过期时间)
- `Cache-Control:max-age` (该字段是 `http1.1` 的规范，强缓存利用其 `max-age` 值来判断缓存资源的最大生命周期，它的值单位为秒)

协商缓存

- `Last-Modified` (值为资源最后更新时间，随服务器 `response` 返回)
- `If-Modified-Since` (通过比较两个时间来判断资源在两次请求期间是否有过修改，如果没有修改，则命中协商缓存)
- `ETag` (表示资源内容的唯一标识，随服务器 `response` 返回)
- `If-None-Match` (服务器通过比较请求头部的 `If-None-Match` 与当前资源的 `ETag` 是否一致来判断资源是否在两次请求之间有过修改，如果没有修改，则命中协商缓存)

#82 WebSocket

由于 `http` 存在一个明显的弊端（消息只能有客户端推送到服务器端，而服务器端不能主动推送到客户端），导致如果服务器如果有连续的变化，这时只能使用轮询，而轮询效率过低，并不适合。于是 `WebSocket` 被发明出来

相比与 `http` 具有以下有点

- 支持双向通信，实时性更强；
- 可以发送文本，也可以二进制文件；
- 协议标识符是 `ws`，加密后是 `wss`；
- 较少的控制开销。连接创建后，`ws` 客户端、服务端进行数据交换时，协议控制的数据包头部较小。在不包含头部的情况下，服务端到客户端的包头只有 2~10 字节（取决于数据包长度），客户端到服务端的话，需要加上额外的4字节的掩码。而 `HTTP` 协议每次通信都需要携带完整的头部；
- 支持扩展。`ws` 协议定义了扩展，用户可以扩展协议，或者实现自定义的子协议。（比如支持自定义压缩算法等）
- 无跨域问题。

实现比较简单，服务端库如 `socket.io`、`ws`，可以很好的帮助我们入门。而客户端也只需要参考 `api` 实现即可

#83 尽可能多的说出你对 Electron 的理解

最最重要的一点，electron 实际上是一个套了 chrome 的 nodeJS 程序

所以应该是从两个方面说开来

- Chrome (无各种兼容性问题)；
- NodeJS (NodeJS 能做的它也能做)

#84 深浅拷贝

浅拷贝

- Object.assign
- 或者展开运算符

深拷贝

- 可以通过 JSON.parse(JSON.stringify(object)) 来解决

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

该方法也是有局限性的

- 会忽略 undefined
- 不能序列化函数
- 不能解决循环引用的对象

#85 防抖/节流

防抖

在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。可以通过函数防抖动来实现

```
// 使用 underscore 的源码来解释防抖动

/**
 * underscore 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
 *
 * @param {function} func      回调函数
 * @param {number}   wait       表示时间窗口的间隔
 * @param {boolean} immediate  设置为ture时，是否立即调用函数
 * @return {function}           返回客户调用函数
 */
_.debounce = function(func, wait, immediate) {
  var timeout, args, context, timestamp, result;

  var later = function() {
    // 现在和上一次时间戳比较
    var last = _.now() - timestamp;
```

```

    // 如果当前间隔时间少于设定时间且大于0就重新设置定时器
    if (last < wait && last >= 0) {
        timeout = setTimeout(later, wait - last);
    } else {
        // 否则的话就是时间到了执行回调函数
        timeout = null;
        if (!immediate) {
            result = func.apply(context, args);
            if (!timeout) context = args = null;
        }
    }
};

return function() {
    context = this;
    args = arguments;
    // 获得时间戳
    timestamp = _.now();
    // 如果定时器不存在且立即执行函数
    var callNow = immediate && !timeout;
    // 如果定时器不存在就创建一个
    if (!timeout) timeout = setTimeout(later, wait);
    if (callNow) {
        // 如果需要立即执行函数的话 通过 apply 执行
        result = func.apply(context, args);
        context = args = null;
    }

    return result;
};
};

```

整体函数实现

对于按钮防点击来说的实现

- 开始一个定时器，只要我定时器还在，不管你怎麼点击都不会执行回调函数。一旦定时器结束并设置为 null，就可以再次点击了
- 对于延时执行函数来说的实现：每次调用防抖动函数都会判断本次调用和之前的时间间隔，如果小于需要的时间间隔，就会重新创建一个定时器，并且定时器的延时为设定时间减去之前的时间间隔。一旦时间到了，就会执行相应的回调函数

节流

防抖动和节流本质是不一样的。防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时间执行。

```

/**
 * underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 / wait
 *
 * @param {function} func      回调函数
 * @param {number}   wait      表示时间窗口的间隔
 * @param {object}  options    如果想忽略开始函数的调用，传入{leading: false}。
 *                            如果想忽略结尾函数的调用，传入{trailing: false}
 *
 * @return {function}          返回客户调用函数
 */
_.throttle = function(func, wait, options) {

```

```
var context, args, result;
var timeout = null;
// 之前的时间戳
var previous = 0;
// 如果 options 没传则设为空对象
if (!options) options = {};
// 定时器回调函数
var later = function() {
    // 如果设置了 leading, 就将 previous 设为 0
    // 用于下面函数的第一个 if 判断
    previous = options.leading === false ? 0 : _.now();
    // 置空一是为了防止内存泄漏, 二是为了下面的定时器判断
    timeout = null;
    result = func.apply(context, args);
    if (!timeout) context = args = null;
};

return function() {
    // 获得当前时间戳
    var now = _.now();
    // 首次进入前者肯定为 true
    // 如果需要第一次不执行函数
    // 就将上次时间戳设为当前的
    // 这样在接下来计算 remaining 的值时会大于0
    if (!previous && options.leading === false) previous = now;
    // 计算剩余时间
    var remaining = wait - (now - previous);
    context = this;
    args = arguments;
    // 如果当前调用已经大于上次调用时间 + wait
    // 或者用户手动调了时间
    // 如果设置了 trailing, 只会进入这个条件
    // 如果没有设置 leading, 那么第一次会进入这个条件
    // 还有一点, 你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
    // 其实还是会进入的, 因为定时器的延时
    // 并不是准确的时间, 很可能你设置了2秒
    // 但是他需要2.2秒才触发, 这时候就会进入这个条件
    if (remaining <= 0 || remaining > wait) {
        // 如果存在定时器就清理掉否则会调用二次回调
        if (timeout) {
            clearTimeout(timeout);
            timeout = null;
        }
        previous = now;
        result = func.apply(context, args);
        if (!timeout) context = args = null;
    } else if (!timeout && options.trailing !== false) {
        // 判断是否设置了定时器和 trailing
        // 没有的话就开启一个定时器
        // 并且不能同时设置 leading 和 trailing
        timeout = setTimeout(later, remaining);
    }
    return result;
};
};
```

#86 谈谈变量提升?

当执行 JS 代码时，会生成执行环境，只要代码不是写在函数中的，就是在全局执行环境中，函数中的代码会产生函数执行环境，只此两种执行环境

- 接下来让我们看一个老生常谈的例子，`var`

```
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
    console.log('call b')
}
```

变量提升

这是因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行环境时，会有两个阶段。第一个阶段是创建的阶段，JS 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 `undefined`，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用

在提升的过程中，相同的函数会覆盖上一个函数，并且函数优先于变量提升

```
b() // call b second

function b() {
    console.log('call b fist')
}
function b() {
    console.log('call b second')
}
var b = 'Hello world'
```

复制代码 `var` 会产生很多错误，所以在 ES6 中引入了 `let`。`let` 不能在声明前使用，但是这并不是常说的 `let` 不会提升，`let` 提升了，在第一阶段内存也已经为他开辟好了空间，但是因为这个声明的特性导致了并不能在声明前使用

#87 什么是单线程，和异步的关系

- 单线程 - 只有一个线程，只能做一件事
- 原因 - 避免 DOM 渲染的冲突
 - 浏览器需要渲染 DOM
 - JS 可以修改 DOM 结构
 - JS 执行的时候，浏览器 DOM 渲染会暂停
 - 两段 JS 也不能同时执行（都修改 DOM 就冲突了）
 - webworker 支持多线程，但是不能访问 DOM
- 解决方案 - 异步

#88 是否用过 jQuery 的 Deferred

jQuery 1.5 的变化 - 1.5 之前

```
var ajax = $.ajax({
    url: 'data.json',
    success: function () {
        console.log('success1')
        console.log('success2')
        console.log('success3')
    },
    error: function () {
        console.log('error')
    }
})
console.log.ajax) // 返回一个 XHR 对象
```

jQuery 1.5 的变化 - 1.5 之后

```
var ajax = $.ajax('data.json')
ajax.done(function () {
    console.log('success 1')
})
.fail(function () {
    console.log('error')
})
.done(function () {
    console.log('success 2')
})
console.log.ajax) // 返回一个 deferred 对象
```

jQuery 1.5 的变化 - 1.5 之后

```
// 很像 Promise 的写法
var ajax = $.ajax('data.json')
```

```
ajax.then(function () {
    console.log('success 1')
}, function () {
    console.log('error 1')
})
.then(function () {
    console.log('success 2')
}, function () {
    console.log('error 2')
})
```

jQuery 1.5 的变化

- 无法改变 JS 异步和单线程的本质
- 只能从写法上杜绝 callback 这种形式
- 它是一种语法糖形式，但是解耦了代码
- 很好的体现：开放封闭原则

使用 jQuery Deferred

```
// 给出一段非常简单的异步操作代码，使用setTimeout函数
var wait = function () {
    var task = function () {
        console.log('执行完成')
    }
    setTimeout(task, 2000)
}
wait()

// 新增需求：要在执行完成之后进行某些特别复杂的操作，代码可能会很多，而且分好几个步骤
```

使用 jQuery Deferred

```
function waitHandle() {
    var dtd = $.Deferred() // 创建一个 deferred 对象

    var wait = function (dtd) { // 要求传入一个 deferred 对象
        var task = function () {
            console.log('执行完成')
            dtd.resolve() // 表示异步任务已经完成
            // dtd.reject() // 表示异步任务失败或出错
        }
        setTimeout(task, 2000)
        return dtd // 要求返回 deferred 对象
    }

    // 注意，这里一定要有返回值
    return wait(dtd)
}
```

使用 jQuery Deferred

```
var w = waitHandle()
w.then(function () {
    console.log('ok 1')
}, function () {
    console.log('err 1')
}).then(function () {
    console.log('ok 2')
}, function () {
    console.log('err 2')
})

// 还有 w.done w.fail
```

使用 jQuery Deferred

总结，dtd 的 API 可分成两类，用意不同

第一类： `dtd.resolve` `dtd.reject`

第二类： `dtd.then` `dtd.done` `dtd.fail`

这两类应该分开，否则后果很严重！

使用 `dtd.promise()`

```
function waitHandle() {
    var dtd = $.Deferred()
    var wait = function (dtd) {
        var task = function () {
            console.log('执行完成')
            dtd.resolve()
        }
        setTimeout(task, 2000)
        return dtd.promise() // 注意，这里返回的是 promise
        // 而不是直接返回 deferred 对象
    }
    return wait(dtd)
}

var w = waitHandle() // 经过上面的改动，w 接收的就是一个 promise 对象
$.when(w)
    .then(function () {
        console.log('ok 1')
    })
    .then(function () {
        console.log('ok 2')
    })

// w.reject() // 执行这句话会直接报错
```

#89 前端面试之hybrid

<http://blog.poeties.top/2018/10/20/fe-interview-hybrid/>

#90 前端面试之组件化

<http://blog.poeties.top/2018/10/20/fe-interview-component/>

#91 前端面试之MVVM浅析

<http://blog.poeties.top/2018/10/20/fe-interview-mvvm/>

#92 实现效果，点击容器内的图标，图标边框变成border 1px solid red，点击空白处重置

```
const box = document.getElementById('box');
function isIcon(target) {
    return target.className.includes('icon');
}

box.onclick = function(e) {
    e.stopPropagation();
    const target = e.target;
    if (isIcon(target)) {
        target.style.border = '1px solid red';
    }
}
const doc = document;
doc.onclick = function(e) {
    const children = box.children;
    for(let i; i < children.length; i++) {
        if (isIcon(children[i])) {
            children[i].style.border = 'none';
        }
    }
}
```

#93 请简单实现双向数据绑定 mvvm

```
<input id="input"/>
```

```
const data = {};
const input = document.getElementById('input');
Object.defineProperty(data, 'text', {
    set(value) {
        input.value = value;
        this.value = value;
    }
});
input.onchange = function(e) {
    data.text = e.target.value;
}
```

#94 实现Storage，使得该对象为单例，并对localStorage进行封装设置值setItem(key,value)和getItem(key)

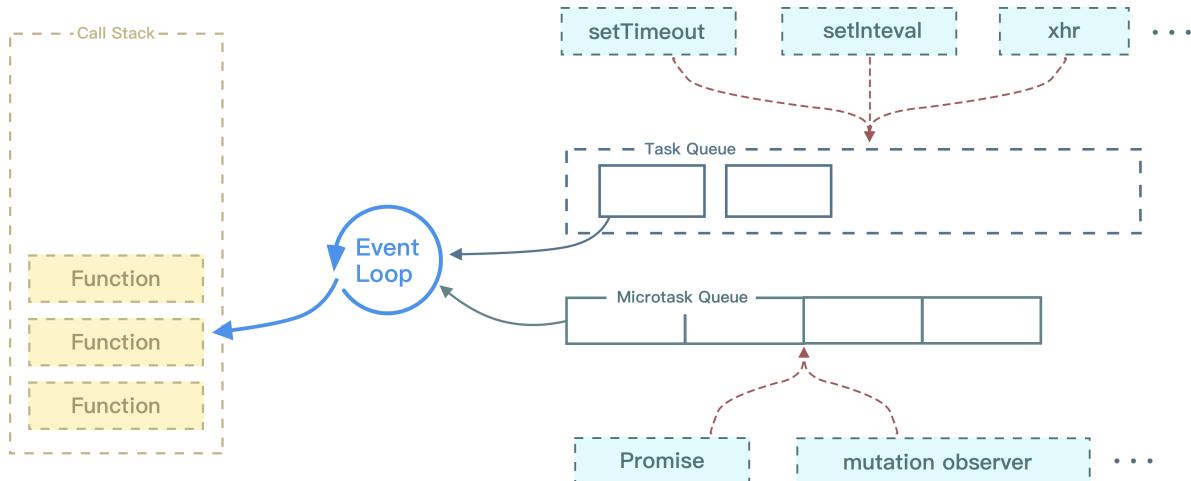
```
var instance = null;
class Storage {
  static getInstance() {
    if (!instance) {
      instance = new Storage();
    }
    return instance;
  }
  setItem = (key, value) => localStorage.setItem(key, value),
  getItem = key => localStorage.getItem(key)
}
```

#95 说说 event loop

首先，js是单线程的，主要的任务是处理用户的交互，而用户的交互无非就是响应DOM的增删改，使用事件队列的形式，一次事件循环只处理一个事件响应，使得脚本执行相对连续，所以有了事件队列，用来储存待执行的事件，那么事件队列的事件从哪里被push进来的呢。那就是另外一个线程叫事件触发线程做的事情了，他的作用主要是在定时触发器线程、异步HTTP请求线程满足特定条件下的回调函数push到事件队列中，等待js引擎空闲的时候去执行，当然js引擎执行过程中有优先级之分，首先js引擎在一次事件循环中，会先执行js线程的主任务，然后会去查找是否有微任务microtask(promise)，如果有那就优先执行微任务，如果没有，在去查找宏任务macrotask(timeout、setInterval）进行执行

众所周知JS是门非阻塞单线程语言，因为在最初JS就是为了和浏览器交互而诞生的。如果JS是门多线程的语言话，我们在多个线程中处理DOM就可能会发生问题（一个线程中新加节点，另一个线程中删除节点）

- JS在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到Task（有多种task）队列中。一旦执行栈为空，Event Loop就会从Task队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说JS中的异步还是同步行为



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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask）和 宏任务（macrotask）。在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task

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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
  console.log('Promise')
  resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout
```

以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务

微任务

- process.nextTick
- promise
- Object.observe
- MutationObserver

宏任务

- script
- setTimeout
- setInterval
- setImmediate
- I/O
- UI rendering

宏任务中包括了 script，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI

- 然后开始下一轮 `Event Loop`，执行宏任务中的异步代码

通过上述的 `Event Loop` 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 `DOM` 的话，为了更快的响应界面响应，我们可以把操作 `DOM` 放入微任务中

#96 说说事件流

事件流分为两种，捕获事件流和冒泡事件流

- 捕获事件流从根节点开始执行，一直往下子节点查找执行，直到查找执行到目标节点
- 冒泡事件流从目标节点开始执行，一直往上父节点冒泡查找执行，直到查到到根节点

事件流分为三个阶段，一个是捕获阶段，一个是处于目标节点阶段，一个是冒泡阶段

#97 JavaScript 对象生命周期的理解

- 当创建一个对象时，`Javascript` 会自动为该对象分配适当的内存
- 垃圾回收器定期扫描对象，并计算引用了该对象的其他对象的数量
- 如果被引用数量为 0，或唯一引用是循环的，那么该对象的内存即可回收

#98 我现在有一个 `canvas`，上面随机布着一些黑块，请实现方法，计算 `canvas` 上有多少个黑块

<https://www.jianshu.com/p/f54d265f7aa4>

#99 请手写实现一个 `promise`

<https://segmentfault.com/a/1190000013396601>

#100 说说从输入URL到看到页面发生的全过程，越详细越好

- 首先浏览器主进程接管，开了一个下载线程。
- 然后进行HTTP请求（DNS查询、IP寻址等等），中间会有三次握手，等待响应，开始下载响应报文。
- 将下载完的内容转交给Renderer进程管理。
- Renderer进程开始解析css rule tree和dom tree，这两个过程是并行的，所以一般我会把link标签放在页面顶部。
- 解析绘制过程中，当浏览器遇到link标签或者script、img等标签，浏览器会去下载这些内容，遇到时候缓存的使用缓存，不适用缓存的重新下载资源。
- css rule tree和dom tree生成完了之后，开始合成render tree，这个时候浏览器会进行layout，开始计算每一个节点的位置，然后进行绘制。
- 绘制结束后，关闭TCP连接，过程有四次挥手

#101 描述一下 `this`

`this`，函数执行的上下文，可以通过 `apply`，`call`，`bind` 改变 `this` 的指向。对于匿名函数或者直接调用的函数来说，`this` 指向全局上下文（浏览器为 `window`，NodeJS 为 `global`），剩下的函数调用，那就是谁调用它，`this` 就指向谁。当然还有 es6 的箭头函数，箭头函数的指向取决于该箭头函数声明的位置，在哪里声明，`this` 就指向哪里

#102 说一下浏览器的缓存机制

浏览器缓存机制有两种，一种为强缓存，一种为协商缓存

- 对于强缓存，浏览器在第一次请求的时候，会直接下载资源，然后缓存在本地，第二次请求的时候，直接使用缓存。
- 对于协商缓存，第一次请求缓存且保存缓存标识与时间，重复请求向服务器发送缓存标识和最后缓存时间，服务端进行校验，如果失效则使用缓存

协商缓存相关设置

- `Expires`：服务端的响应头，第一次请求的时候，告诉客户端，该资源什么时候会过期。
`Expires` 的缺陷是必须保证服务端时间和客户端时间严格同步。
- `Cache-control: max-age`：表示该资源多少时间后过期，解决了客户端和服务端时间必须同步的问题，
- `If-None-Match/ETag`：缓存标识，对比缓存时使用它来标识一个缓存，第一次请求的时候，服务端会返回该标识给客户端，客户端在第二次请求的时候会带上该标识与服务端进行对比并返回 `If-None-Match` 标识是否表示匹配。
- `Last-modified/If-Modified-since`：第一次请求的时候服务端返回 `Last-modified` 表明请求的资源上次的修改时间，第二次请求的时候客户端带上请求头 `If-Modified-since`，表示资源上次的修改时间，服务端拿到这两个字段进行对比

#103 现在要你完成一个Dialog组件，说说你设计的思路？它应该有什么功能？

- 该组件需要提供 `hook` 指定渲染位置，默认渲染在 `body` 下面。
- 然后改组件可以指定外层样式，如宽度等
- 组件外层还需要一层 `mask` 来遮住底层内容，点击 `mask` 可以执行传进来的 `onCancel` 函数关闭 `Dialog`。
- 另外组件是可控的，需要外层传入 `visible` 表示是否可见。
- 然后 `Dialog` 可能需要自定义头 `head` 和底部 `footer`，默认有头部和底部，底部有一个确认按钮和取消按钮，确认按钮会执行外部传进来的 `onok` 事件，然后取消按钮会执行外部传进来的 `onCancel` 事件。
- 当组件的 `visible` 为 `true` 时候，设置 `body` 的 `overflow` 为 `hidden`，隐藏 `body` 的滚动条，反之显示滚动条。
- 组件高度可能大于页面高度，组件内部需要滚动条。
- 只有组件的 `visible` 有变化且为 `true` 时候，才重渲染组件内的所有内容

#104 caller 和 callee 的区别

callee

`caller` 返回一个函数的引用，这个函数调用了当前的函数。

使用这个属性要注意

- 这个属性只有当函数在执行时才有用
- 如果在 `javascript` 程序中，函数是由顶层调用的，则返回 `null`

`functionName.caller: functionName` 是当前正在执行的函数。

```
function a() {  
    console.log(a.caller)  
}
```

callee

callee 放回正在执行的函数本身的引用，它是 arguments 的一个属性

使用 callee 时要注意：

- 这个属性只有在函数执行时才有效
- 它有一个 length 属性，可以用来获得形参的个数，因此可以用来比较形参和实参数是否一致，即比较 arguments.length 是否等于 arguments.callee.length
- 它可以用来递归匿名函数。

```
function a() {
  console.log(arguments.callee)
}
```

#105 ajax、axios、fetch区别

jQuery ajax

```
$.ajax({
  type: 'POST',
  url: url,
  data: data,
  dataType: dataType,
  success: function () {},
  error: function () {}
});
```

优缺点：

- 本身是针对 MVC 的编程，不符合现在前端 MVVM 的浪潮
- 基于原生的 XHR 开发，XHR 本身的架构不清晰，已经有了 fetch 的替代方案
- jQuery 整个项目太大，单纯使用 ajax 却要引入整个 jQuery 非常的不合理（采取个性化打包的方案又不能享受 CDN 服务）

axios

```
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
})
.then(function (response) {
  console.log(response);
})
.catch(function (error) {
  console.log(error);
});
```

优缺点：

- 从浏览器中创建 XMLHttpRequest
- 从 node.js 发出 http 请求

- 支持 Promise API
- 拦截请求和响应
- 转换请求和响应数据
- 取消请求
- 自动转换 JSON 数据
- 客户端支持防止 CSRF/XSRF

fetch

```
try {
  let response = await fetch(url);
  let data = response.json();
  console.log(data);
} catch(e) {
  console.log("Oops, error", e);
}
```

优缺点：

- `fetch` 只对网络请求报错，对 400, 500 都当做成功的请求，需要封装去处理
- `fetch` 默认不会带 cookie，需要添加配置项
- `fetch` 不支持 `abort`，不支持超时控制，使用 `setTimeout` 及 `Promise.reject` 的实现的超时控制并不能阻止请求过程继续在后台运行，造成了大量的浪费
- `fetch` 没有办法原生监测请求的进度，而 XHR 可以

#106 JavaScript 的组成

- JavaScript

由以下三部分组成：

- ECMAScript（核心）：JavaScript` 语言基础
- DOM（文档对象模型）：规定了访问 HTML 和 XML 的接口
- BOM（浏览器对象模型）：提供了浏览器窗口之间进行交互的对象和方法

#107 检测浏览器版本有哪些方式？

- 根据 `navigator.userAgent` `UA.toLowerCase().indexOf('chrome')`
- 根据 `window` 对象的成员 `'ActiveXObject' in window`

#108 介绍 JS 有哪些内置对象

- 数据封装类对象：`Object`、`Array`、`Boolean`、`Number`、`String`
- 其他对象：`Function`、`Arguments`、`Math`、`Date`、`RegExp`、`Error`
- ES6 新增对象：`Symbol`、`Map`、`Set`、`Promises`、`Proxy`、`Reflect`

#109 说几条写 JavaScript 的基本规范

- 代码缩进，建议使用“四个空格”缩进
- 代码段使用花括号 {} 包裹
- 语句结束使用分号；
- 变量和函数在使用前进行声明
- 以大写字母开头命名构造函数，全大写命名常量

- 规范定义 JSON 对象，补全双引号
- 用 {} 和 [] 声明对象和数组

#110 如何编写高性能的JavaScript

- 遵循严格模式: "use strict";
- 将js脚本放在页面底部，加快渲染页面
- 将js脚本将脚本成组打包，减少请求
- 使用非阻塞方式下载js脚本
- 尽量使用局部变量来保存全局变量
- 尽量减少使用闭包
- 使用 window 对象属性方法时，省略 window
- 尽量减少对象成员嵌套
- 缓存 DOM 节点的访问
- 通过避免使用 eval() 和 Function() 构造器
- 给 setTimeout() 和 setInterval() 传递函数而不是字符串作为参数
- 尽量使用直接量创建对象和数组
- 最小化重绘(repaint)和回流(reflow)

#111 描述浏览器的渲染过程，DOM树和渲染树的区别

- 浏览器的渲染过程：
 - 解析 HTML 构建 DOM (DOM树)，并行请求 css/image/js
 - CSS 文件下载完成，开始构建 CSSOM (CSS 树)
 - CSSOM 构建结束后，和 DOM 一起生成 Render Tree (渲染树)
 - 布局(Layout)：计算出每个节点在屏幕中的位置
 - 显示(Painting)：通过显卡把页面画到屏幕上
- DOM 树 和 渲染树 的区别：
 - DOM 树与 HTML 标签一一对应，包括 head 和隐藏元素
 - 渲染树不包括 head 和隐藏元素，大段文本的每一个行都是独立节点，每一个节点都有对应的 css 属性

#112 script 的位置是否会影响首屏显示时间

- 在解析 HTML 生成 DOM 过程中，js 文件的下载是并行的，不需要 DOM 处理到 script 节点。因此，script 的位置不影响首屏显示的开始时间。
- 浏览器解析 HTML 是自上而下的线性过程，script 作为 HTML 的一部分同样遵循这个原则
- 因此，script 会延迟 DomContentLoaded，只显示其上部分首屏内容，从而影响首屏显示的完成时间

#113 解释JavaScript中的作用域与变量声明提升

- JavaScript 作用域：
 - 在 Java、C 等语言中，作用域为 for 语句、if 语句或 {} 内的一块区域，称为作用域；
 - 而在 Javascript 中，作用域为 function(){} 内的区域，称为函数作用域。
- JavaScript 变量声明提升：
 - 在 Javascript 中，函数声明与变量声明经常被 Javascript 引擎隐式地提升到当前作用域的顶部。
 - 声明语句中的赋值部分并不会被提升，只有名称被提升
 - 函数声明的优先级高于变量，如果变量名跟函数名相同且未赋值，则函数声明会覆盖变量声明
 - 如果函数有多个同名参数，那么最后一个参数（即使没有定义）会覆盖前面的同名参数

#114 JavaScript有几种类型的值？，你能画一下他们的内存图吗

- 原始数据类型 (`Undefined`, `Null`, `Boolean`, `Number`、`String`) -- 栈
- 引用数据类型 (对象、数组和函数) -- 堆
- 两种类型的区别是：存储位置不同：
- 原始数据类型是直接存储在栈(`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；
- 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。

#115 JavaScript如何实现一个类，怎么实例化这个类

- 构造函数法（

```
this
```

```
+
```

```
prototype
```

```
) -- 用
```

```
new
```

关键字 `new` 生成实例对象

- 缺点：用到了 `this` 和 `prototype`，编写复杂，可读性差

```
function Mobile(name, price){  
    this.name = name;  
    this.price = price;  
}  
Mobile.prototype.sell = function(){  
    alert(this.name + ", 售价 $" + this.price);  
}  
var iPhone7 = new Mobile("iPhone7", 1000);  
iPhone7.sell();
```

- `Object.create` 法 -- 用 `Object.create()` 生成实例对象
- 缺点：不能实现私有属性和私有方法，实例对象之间也不能共享数据

```
var Person = {  
    firstname: "Mark",  
    lastname: "Yun",  
    age: 25,  
    introduce: function(){  
        alert('I am ' + Person.firstname + ' ' + Person.lastname);  
    }  
};  
  
var person = Object.create(Person);
```

```
person.introduce();

// object.create 要求 IE9+, 低版本浏览器可以自行部署:
if (!Object.create) {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

- 极简主义法 (消除

```
this
```

和

```
prototype
```

) -- 调用

```
createNew()
```

得到实例对象

- 优点: 容易理解, 结构清晰优雅, 符合传统的"面向对象编程"的构造

```
var Cat = {
    age: 3, // 共享数据 -- 定义在类对象内, createNew() 外
    createNew: function () {
        var cat = {};
        // var cat = Animal.createNew(); // 继承 Animal 类
        cat.name = "小咪";
        var sound = "喵喵喵"; // 私有属性--定义在 createNew() 内, 输出对象外
        cat.makeSound = function () {
            alert(sound); // 暴露私有属性
        };
        cat.changeAge = function (num) {
            cat.age = num; // 修改共享数据
        };
        return cat; // 输出对象
    }
};

var cat = Cat.createNew();
cat.makeSound();
```

| ES6 语法糖 `class` -- 用 `new` 关键字生成实例对象

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
    toString() {  
        return '(' + this.x + ', ' + this.y + ')';  
    }  
}  
  
var point = new Point(2, 3);
```

#116 Javascript如何实现继承

构造函数绑定：使用 `call` 或 `apply` 方法，将父对象的构造函数绑定在子对象上

```
function Cat(name,color){  
    Animal.apply(this, arguments);  
    this.name = name;  
    this.color = color;  
}
```

- 实例继承：将子对象的 `prototype` 指向父对象的一个实例

```
Cat.prototype = new Animal();  
Cat.prototype.constructor = Cat;
```

拷贝继承：如果把父对象的所有属性和方法，拷贝进子对象

```
function extend(child, Parent) {  
    var p = Parent.prototype;  
    var c = Child.prototype;  
    for (var i in p) {  
        c[i] = p[i];  
    }  
    c.uber = p;  
}
```

原型继承：将子对象的 `prototype` 指向父对象的 `prototype`

```
function extend(child, Parent) {  
    var F = function(){};  
    F.prototype = Parent.prototype;  
    Child.prototype = new F();  
    Child.prototype.constructor = Child;  
    Child.uber = Parent.prototype;  
}
```

ES6 语法糖 `extends: class ColorPoint extends Point {}`

```

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y); // 调用父类的constructor(x, y)
        this.color = color;
    }
    toString() {
        return this.color + ' ' + super.toString(); // 调用父类的toString()
    }
}

```

#117 Javascript作用链域

- 全局函数无法查看局部函数的内部细节，但局部函数可以查看其上层的函数细节，直至全局细节
- 如果当前作用域没有找到属性或方法，会向上层作用域查找，直至全局函数，这种形式就是作用域链

#118 介绍 DOM 的发展

- `DOM`：文档对象模型（`Document Object Model`），定义了访问HTML和XML文档的标准，与编程语言及平台无关
- `DOM0`：提供了查询和操作Web文档的内容API。未形成标准，实现混乱。如：
`document.forms['login']`
- `DOM1`：W3C提出标准化的DOM，简化了对文档中任意部分的访问和操作。如：`Javascript`中的`Document`对象
- `DOM2`：原来DOM基础上扩充了鼠标事件等细分模块，增加了对CSS的支持。如：
`getComputedStyle(elem, pseudo)`
- `DOM3`：增加了XPath模块和加载与保存（`Load and Save`）模块。如：`xPathEvaluator`

#119 介绍DOM0, DOM2, DOM3事件处理方式区别

- DOM0级事件处理方式：
 - `btn.onclick = func;`
 - `btn.onclick = null;`
- DOM2级事件处理方式：
 - `btn.addEventListener('click', func, false);`
 - `btn.removeEventListener('click', func, false);`
 - `btn.attachEvent("onclick", func);`
 - `btn.detachEvent("onclick", func);`
- DOM3级事件处理方式：
 - `eventUtil.addListener(input, "textInput", func);`
 - `eventUtil` 是自定义对象，`textInput` 是DOM3级事件

事件的三个阶段

- 捕获、目标、冒泡

#120 介绍事件“捕获”和“冒泡”执行顺序和事件的执行次数

- 按照W3C标准的事件：首先是进入捕获阶段，直到达到目标元素，再进入冒泡阶段
- 事件执行次数（`DOM2-addEventListener`）：元素上绑定事件的个数
 - 注意1：前提是事件被确实触发
 - 注意2：事件绑定几次就算几个事件，即使类型和功能完全一样也不会“覆盖”

- 事件执行顺序：判断的关键是否目标元素
 - 非目标元素：根据W3C的标准执行：捕获->目标元素->冒泡（不依据事件绑定顺序）
 - 目标元素：依据事件绑定顺序：先绑定的事件先执行（不依据捕获冒泡标准）
 - 最终顺序：父元素捕获->目标元素事件1->目标元素事件2->子元素捕获->子元素冒泡->父元素冒泡
 - 注意：子元素事件执行前提 事件确实“落”到子元素布局区域上，而不是简单的具有嵌套关系

在一个DOM上同时绑定两个点击事件：一个用捕获，一个用冒泡。事件会执行几次，先执行冒泡还是捕获？

- 该DOM上的事件如果被触发，会执行两次（执行次数等于绑定次数）
- 如果该DOM是目标元素，则按事件绑定顺序执行，不区分冒泡/捕获
- 如果该DOM是处于事件流中的非目标元素，则先执行捕获，后执行冒泡

事件的代理/委托

- 事件委托是指将事件绑定目标元素的到父元素上，利用冒泡机制触发该事件
 - 优点：
 - 可以减少事件注册，节省大量内存占用
 - 可以将事件应用于动态添加的子元素上
 - 缺点：使用不当会造成事件在不应该触发时触发
 - 示例：

```
ulEl.addEventListener('click', function(e){
  var target = event.target || event.srcElement;
  if(!target && target.nodeName.toUpperCase() === "LI"){
    console.log(target.innerHTML);
  }
}, false);
```

W3C事件的 target 与 currentTarget 的区别？

- `target` 只会出现在事件流的目标阶段
- `currentTarget` 可能出现在事件流的任何阶段
- 当事件流处在目标阶段时，二者的指向相同
- 当事件流处于捕获或冒泡阶段时：`currentTarget` 指向当前事件活动的对象(一般为父级)

如何派发事件(dispatchEvent)？（如何进行事件广播？）

- W3C: 使用 `dispatchEvent` 方法
- IE: 使用 `fireEvent` 方法

```
var fireEvent = function(element, event){
  if (document.createEventObject){
    var mockEvent = document.createEventObject();
    return element.fireEvent('on' + event, mockEvent)
  }else{
    var mockEvent = document.createEvent('HTMLEvents');
    mockEvent.initEvent(event, true, true);
    return !element.dispatchEvent(mockEvent);
  }
}
```

#121 什么是函数节流？介绍一下应用场景和原理？

- 函数节流(throttle)是指阻止一个函数在很短时间间隔内连续调用。只有当上一次函数执行后达到规定的时间间隔，才能进行下一次调用。但要保证一个累计最小调用间隔（否则拖拽类的节流都将无连续效果）
- 函数节流用于`onresize`, `onscroll`等短时间内会多次触发的事件
- 函数节流的原理：使用定时器做时间节流。当触发一个事件时，先用`setTimeout`让这个事件延迟一小段时间再执行。如果在这个时间间隔内又触发了事件，就`clearTimeout`原来的定时器，再`setTimeout`一个新的定时器重复以上流程。

函数节流简单实现：

```
function throttle(method, context) {  
    clearTimeout(method.tId);  
    method.tId = setTimeout(function() {  
        method.call(context);  
    }, 100); // 两次调用至少间隔 100ms  
}  
// 调用  
window.onresize = function() {  
    throttle(myFunc, window);  
}
```

#122 区分什么是“客户区坐标”、“页面坐标”、“屏幕坐标”

- 客户区坐标：鼠标指针在可视区中的水平坐标(`clientX`)和垂直坐标(`clientY`)
- 页面坐标：鼠标指针在页面布局中的水平坐标(`pageX`)和垂直坐标(`pageY`)
- 屏幕坐标：设备物理屏幕的水平坐标(`screenX`)和垂直坐标(`screenY`)

如何获得一个DOM元素的绝对位置？

- `elem.offsetLeft`: 返回元素相对于其定位父级左侧的距离
- `elem.offsetTop`: 返回元素相对于其定位父级顶部的距离
- `elem.getBoundingClientRect()`: 返回一个`DOMRect`对象，包含一组描述边框的只读属性，单位像素

#123 解释一下这段代码的意思

```
[].forEach.call($("#*"), function(el){  
    el.style.outline = "1px solid #" + (~Math.random()*  
(1<<24)).toString(16);  
})
```

- 解释：获取页面所有的元素，遍历这些元素，为它们添加1像素随机颜色的轮廓(outline)
- 1. `$$(sel)` // \$\$函数被许多现代浏览器命令行支持，等价于`document.querySelectorAll(sel)`
- 1. `[]`.`forEach.call(NodeLists)` // 使用`call`函数将数组遍历函数`forEach`应到节点元素列表
- 1. `el.style.outline = "1px solid #333"` // 样式`outline`位于盒模型之外，不影响元素布局位置
- 1. `(1<<24)` // `parseInt("fffff", 16) == 16777215 == 2^24 - 1 // 1<<24 == 2^24 == 16777216`
- 1. `Math.random()*(1<<24)` // 表示一个位于0到16777216之间的随机浮点数
- 1. `~~Math.random()*(1<<24)` // `~~`作用相当于`parseInt`取整
- 1. `(~~(Math.random()*(1<<24))).toString(16)` // 转换为一个十六进制-

#124 Javascript垃圾回收方法

- 标记清除 (mark and sweep)
 - 这是JavaScript最常见的垃圾回收方式，当变量进入执行环境的时候，比如函数中声明一个变量，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”
 - 垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了

引用计数(reference counting)

在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间

#125 请解释一下 JavaScript 的同源策略

- 概念:同源策略是客户端脚本（尤其是Javascript）的重要安全度量标准。它最早出自Netscape Navigator2.0，其目的是防止某个文档或脚本从多个不同源装载。这里的同源策略指的是：协议，域名，端口相同，同源策略是一种安全协议
- 指一段脚本只能读取来自同一来源的窗口和文档的属性

为什么要有同源限制？

- 我们举例说明：比如一个黑客程序，他利用iframe把真正的银行登录页面嵌到他的页面上，当你使用真实的用户名，密码登录时，他的页面就可以通过Javascript读取到你的表单中input中的内容，这样用户名，密码就轻松到手了。
- 缺点
 - 现在网站的JS都会进行压缩，一些文件用了严格模式，而另一些没有。这时这些本来是严格模式的文件，被 merge后，这个串就到了文件的中间，不仅没有指示严格模式，反而在压缩后浪费了字节

#126 如何删除一个cookie

- 将时间设为当前时间往前一点

```
var date = new Date();

date.setDate(date.getDate() - 1); //真正的删除
```

setDate() 方法用于设置一个月的某一天

- expires 的设置

```
document.cookie = 'user=' + encodeURIComponent('name') + ';expires = ' + new
Date(0)
```

#127 页面编码和被请求的资源编码如果不一致如何处理

- 后端响应头设置 `charset`
- 前端页面 `<meta>` 设置 `charset`

#128 把 `<script>` 放在 `</body>` 之前和之后有什么区别？浏览器会如何解析它们？

- 按照HTML标准，在 `</body>` 结束后出现 `<script>` 或任何元素的开始标签，都是解析错误
- 虽然不符合HTML标准，但浏览器会自动容错，使实际效果与写在 `</body>` 之前没有区别
- 浏览器的容错机制会忽略 `<script>` 之前的 `</body>`，视作 `<script>` 仍在 body 体内。省略 `</body>` 和 `</html>` 闭合标签符合HTML标准，服务器可以利用这一标准尽可能少输出内容

#129 JavaScript 中，调用函数有哪几种方式

- 方法调用模式 `Foo.foo(arg1, arg2);`
- 函数调用模式 `foo(arg1, arg2);`
- 构造器调用模式 `(new Foo())(arg1, arg2);`
- `call/apply` 调用模式 `Foo.foo.call(that, arg1, arg2);`
- `bind` 调用模式 `Foo.foo.bind(that)(arg1, arg2)();`

#130 简单实现 `Function.bind` 函数

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(that) {
        var func = this, args = arguments;
        return function() {
            return func.apply(that, Array.prototype.slice.call(args, 1));
        }
    }
}
// 只支持 bind 阶段的默认参数:
func.bind(that, arg1, arg2)();

// 不支持以下调用阶段传入的参数:
func.bind(that)(arg1, arg2);
```

#131 列举一下JavaScript数组和对象有哪些原生方法？

数组：

- `arr.concat(arr1, arr2, arrn);`
- `arr.join(",");`
- `arr.sort(func);`
- `arr.pop();`
- `arr.push(e1, e2, en);`
- `arr.shift();`
- `unshift(e1, e2, en);`
- `arr.reverse();`
- `arr.slice(start, end);`
- `arr.splice(index, count, e1, e2, en);`
- `arr.indexOf(e1);`

- `arr.includes(e1); // ES6`

对象:

- `object.hasOwnProperty(prop);`
- `object.propertyIsEnumerable(prop);`
- `object.valueOf();`
- `object.toString();`
- `object.toLocaleString();`
- `Class.prototype.isPrototypeOf(object);`

#132 Array.slice() 与 Array.splice() 的区别?

`slice`

“读取”数组指定的元素，不会对原数组进行修改

- 语法: `arr.slice(start, end)`
- `start` 指定选取开始位置 (含)
- `end` 指定选取结束位置 (不含)

`splice`

- “操作”数组指定的元素，会修改原数组，返回被删除的元素
- 语法: `arr.splice(index, count, [insert Elements])`
- `index` 是操作的起始位置
- `count = 0` 插入元素，`count > 0` 删除元素
- `[insert Elements]` 向数组新插入的元素

#133 MVVM

MVVM 由以下三个内容组成

- `View`: 界面
 - `Model`: 数据模型
 - `ViewModel`: 作为桥梁负责沟通 `View` 和 `Model`
- 在 JQuery 时期，如果需要刷新 UI 时，需要先取到对应的 DOM 再更新 UI，这样数据和业务的逻辑就和页面有强耦合
- 在 MVVM 中，UI 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 UI，UI 如果改变，也会改变对应的数据。这种方式就可以在业务处理中只关心数据的流转，而无需直接和页面打交道。ViewModel 只关心数据和业务的处理，不关心 View 如何处理数据，在这种情况下，View 和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 复用这个 ViewModel
- 在 MVVM 中，最核心的也就是数据双向绑定，例如 Angular 的脏数据检测，Vue 中的数据劫持

脏数据检测

- 当触发了指定事件后会进入脏数据检测，这时会调用 `digest` 循环遍历所有的数据观察者，判断当前值是否和先前的值有区别，如果检测到变化的话，会调用 `watch` 函数，然后再次调用 `$digest` 循环直到发现没有变化。循环至少为二次，至多为十次
- 脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 Vue 中的双向绑定是存在问题的。并且脏数据检测可以实现批量检测出更新的值，再去统一更新 UI，大大减少了操作 DOM 的次数

数据劫持

- Vue 内部使用了 `Object.defineProperty()` 来实现双向绑定，通过这个函数可以监听到 `set` 和 `get` 的事件

```

var data = { name: 'yck' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(data).forEach(key => {
    defineReactive(data, key, data[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}

```

以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的，还需要在适当的时候给属性添加发布订阅

```

<div>
  {{name}}
</div>

```

在解析如上模板代码时，遇到 `name` 就会给属性 `name` 添加发布订阅

```

// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
  addSub(sub) {
    // sub 是 Watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {

```

```

        sub.update()
    })
}
}

// 全局属性，通过该属性配置 Watcher
Dep.target = null

function update(value) {
    document.querySelector('div').innerText = value
}

class Watcher {
    constructor(obj, key, cb) {
        // 将 Dep.target 指向自己
        // 然后触发属性的 getter 添加监听
        // 最后将 Dep.target 置空
        Dep.target = this
        this.cb = cb
        this.obj = obj
        this.key = key
        this.value = obj[key]
        Dep.target = null
    }
    update() {
        // 获得新值
        this.value = this.obj[this.key]
        // 调用 update 方法更新 Dom
        this.cb(this.value)
    }
}
var data = { name: 'yck' }
observe(data)
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'

```

接下来，对 defineReactive 函数进行改造

```

function defineReactive(obj, key, val) {
    // 递归子属性
    observe(val)
    let dp = new Dep()
    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
            console.log('get value')
            // 将 Watcher 添加到订阅
            if (Dep.target) {
                dp.addSub(Dep.target)
            }
            return val
        },
        set: function reactiveSetter(newVal) {
            console.log('change value')
            val = newVal
        }
    })
}

```

```
// 执行 watcher 的 update 方法
dp.notify()
}
})
}
```

以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 getter 来实现发布订阅的添加

Proxy 与 Object.defineProperty 对比

- `Object.defineProperty`

虽然已经能够实现双向绑定了，但是他还是有缺陷的。

- 只能对属性进行数据劫持，所以需要深度遍历整个对象
- 对于数组不能监听到数据的变化

虽然 `Vue` 中确实能检测到数组数据的变化，但是其实是使用了 `hack` 的办法，并且也是有缺陷的

#134 WEB应用从服务器主动推送Data到客户端有那些方式

- `AJAX` 轮询
- `html5` 服务器推送事件 `(new EventSource(SERVER_URL)).addEventListener("message", func);`
- `html5 Websocket`
- `(new WebSocket(SERVER_URL)).addEventListener("message", func);`

#135 继承

- **原型链继承**，将父类的实例作为子类的原型，他的特点是实例是子类的实例也是父类的实例，父类新增的原型方法/属性，子类都能够访问，并且原型链继承简单易于实现，缺点是来自原型对象的所有属性被所有实例共享，无法实现多继承，无法向父类构造函数传参。
- **构造继承**，使用父类的构造函数来增强子类实例，即复制父类的实例属性给子类，构造继承可以向父类传递参数，可以实现多继承，通过 call 多个父类对象。但是构造继承只能继承父类的实例属性和方法，不能继承原型属性和方法，无法实现函数服用，每个子类都有父类实例函数的副本，影响性能
- **实例继承**，为父类实例添加新特性，作为子类实例返回，实例继承的特点是不限制调用方法，不管是 new 子类 () 还是子类 () 返回的对象具有相同的效果，缺点是实例是父类的实例，不是子类的实例，不支持多继承
- **拷贝继承**：特点：支持多继承，缺点：效率较低，内存占用高（因为要拷贝父类的属性）无法获取父类不可枚举的方法（不可枚举方法，不能使用 for in 访问到）
- **组合继承**：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
- **寄生组合继承**：通过寄生方式，砍掉父类的实例属性，这样，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

#136 this指向

1. this 指向有哪几种

- 默认绑定：全局环境中，`this` 默认绑定到 `window`
- 隐式绑定：一般地，被直接对象所包含的函数调用时，也称为方法调用，`this` 隐式绑定到该直接对象

- 隐式丢失：隐式丢失是指被隐式绑定的函数丢失绑定对象，从而默认绑定到 `window`。显式绑定：
通过 `call()`、`apply()`、`bind()` 方法把对象绑定到 `this` 上，叫做显式绑定

- `new`

绑定：如果函数或者方法调用之前带有关键字

`new`

，它就构成构造函数调用。对于

`this`

绑定来说，称为

`new`

绑定

- 构造函数通常不使用 `return` 关键字，它们通常初始化新对象，当构造函数的函数体执行完毕时，它会显式返回。在这种情况下，构造函数调用表达式的计算结果就是这个新对象的值
- 如果构造函数使用 `return` 语句但没有指定返回值，或者返回一个原始值，那么这时将忽略返回值，同时使用这个新对象作为调用结果
- 如果构造函数显式地使用 `return` 语句返回一个对象，那么调用表达式的值就是这个对象

2. 改变函数内部 `this` 指针的指向函数 (`bind`, `apply`, `call` 的区别)

- `apply`：调用一个对象的一个方法，用另一个对象替换当前对象。例如：`B.apply(A, arguments)`; 即 A 对象应用 B 对象的方法
- `call`：调用一个对象的一个方法，用另一个对象替换当前对象。例如：`B.call(A, args1, args2)`; 即 A 对象调用 B 对象的方法
- `bind` 除了返回是函数以外，它的参数和 `call` 一样

3. 箭头函数

- 箭头函数没有 `this`，所以需要通过查找作用域链来确定 `this` 的值，这就意味着如果箭头函数被非箭头函数包含，`this` 绑定的就是最近一层非箭头函数的 `this`，
- 箭头函数没有自己的 `arguments` 对象，但是可以访问外围函数的 `arguments` 对象
- 不能通过 `new` 关键字调用，同样也没有 `new.target` 值和原型

#137 判断是否是数组

- `Array.isArray(arr)`
- `Object.prototype.toString.call(arr) === '[Object Array]'`
- `arr instanceof Array`
- `array.constructor === Array`

#138 加载

1. 异步加载js的方法

- `defer`：只支持 IE。如果您的脚本不会改变文档的内容，可将 `defer` 属性加入到 `<script>` 标签中，以便加快处理文档的速度。因为浏览器知道它将能够安全地读取文档的剩余部分而不用执行脚本，它将推迟对脚本的解释，直到文档已经显示给用户为止

- `async`: `HTML5` 属性，仅适用于外部脚本；并且如果在IE中，同时存在 `defer` 和 `async`，那么 `defer` 的优先级比较高；脚本将在页面完成时执行

2. 图片的懒加载和预加载

- 预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。
- 懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数

两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

#139 垃圾回收

找出那些不再继续使用的变量，然后释放其占用的内存。为此，垃圾收集器会按照固定的时间间隔(或代码执行中预定的收集时间)，周期性地执行这一操作。

标记清除

先所有都加上标记，再把环境中引用到的变量去除标记。剩下的就是没用的了

引用计数

跟踪记录每个值被引用的次数。清除引用次数为0的变量 `△` 会有循环引用问题。循环引用如果大量存在就会导致内存泄露。

四、jQuery

#1 你觉得jQuery或zepto源码有哪些写的好地方

- `jquery` 源码封装在一个匿名函数的自执行环境中，有助于防止变量的全局污染，然后通过传入 `window` 对象参数，可以使 `window` 对象作为局部变量使用，好处是当 `jquery` 中访问 `window` 对象的时候，就不用将作用域链退回到顶层作用域了，从而可以更快的访问 `window` 对象。同样，传入 `undefined` 参数，可以缩短查找 `undefined` 时的作用域链

```
(function( window, undefined ) {  
  
    //用一个函数域包起来，就是所谓的沙箱  
  
    //在这里边var定义的变量，属于这个函数域内的局部变量，避免污染全局  
  
    //把当前沙箱需要的外部变量通过函数参数引入进来  
  
    //只要保证参数对内提供的接口的一致性，你还可以随意替换传进来的这个参数  
  
    window.jquery = window.$ = jquery;  
  
})( window );
```

- `jquery` 将一些原型属性和方法封装在了 `jquery.prototype` 中，为了缩短名称，又赋值给了 `jquery.fn`，这是很形象的写法
- 有一些数组或对象的方法经常能使用到，`jQuery` 将其保存为局部变量以提高访问速度
- `jquery` 实现的链式调用可以节约代码，所返回的都是同一个对象，可以提高代码效率

#2 jQuery 的实现原理

- `(function(window, undefined) {})(window);`
- `jQuery` 利用 `JS` 函数作用域的特性，采用立即调用表达式包裹了自身，解决命名空间和变量污染问题
- `window.jQuery = window.$ = jQuery;`
- 在闭包当中将 `jQuery` 和 `$` 绑定到 `window` 上，从而将 `jQuery` 和 `$` 暴露为全局变量

#3 `jQuery.fn` 的 `init` 方法返回的 `this` 指的是什么对象

- `jQuery.fn` 的 `init` 方法返回的 `this` 就是 `jQuery` 对象
- 用户使用 `jQuery()` 或 `$()` 即可初始化 `jQuery` 对象，不需要动态的去调用 `init` 方法

#4 `jQuery.extend` 与 `jQuery.fn.extend` 的区别

- `$.fn.extend()` 和 `$.extend()` 是 `jQuery` 为扩展插件提供了两个方法
- `$.extend(object); // 为jQuery添加“静态方法”（工具方法）`

```
$.extend({
    min: function(a, b) { return a < b ? a : b; },
    max: function(a, b) { return a > b ? a : b; }
});
$.min(2,3); // 2
$.max(4,5); // 5
```

- `$.extend([true,] targetObject, object1[, object2]); // 对target对象进行扩展`

```
var settings = {validate:false, limit:5};
var options = {validate:true, name:"bar"};
$.extend(settings, options); // 注意：不支持第一个参数传 false
// settings == {validate:true, limit:5, name:"bar"}
```

- `$.fn.extend(json); // 为jQuery添加“成员函数”（实例方法）`

```
$.fn.extend({
    alertValue: function() {
        $(this).click(function(){
            alert($(this).val());
        });
    }
});

$("#email").alertValue();
```

#5 jQuery 的属性拷贝(`extend`)的实现原理是什么，如何实现深拷贝

- 浅拷贝（只复制一份原始对象的引用） `var newObject = $.extend({}, oldObject);`
- 深拷贝（对原始对象属性所引用的对象进行递归拷贝） `var newObject = $.extend(true, {}, oldObject);`

#6 jQuery 的队列是如何实现的

- jQuery 核心中有一组队列控制方法，由 `queue()`/`dequeue()`/`clearQueue()` 三个方法组成。
- 主要应用于 `animate()`, `ajax`，其他要按时间顺序执行的事件中

```
var func1 = function(){alert('事件1');}
var func2 = function(){alert('事件2');}
var func3 = function(){alert('事件3');}
var func4 = function(){alert('事件4');}

// 入栈队列事件
$('#box').queue("queue1", func1); // push func1 to queue1
$('#box').queue("queue1", func2); // push func2 to queue1

// 替换队列事件
$('#box').queue("queue1", []); // delete queue1 with empty array
$('#box').queue("queue1", [func3, func4]); // replace queue1

// 获取队列事件（返回一个函数数组）
$('#box').queue("queue1"); // [func3(), func4()]

// 出栈队列事件并执行
$('#box').dequeue("queue1"); // return func3 and do func3
$('#box').dequeue("queue1"); // return func4 and do func4

// 清空整个队列
$('#box').clearQueue("queue1"); // delete queue1 with clearQueue
```

#7 jQuery 中的 `bind()`, `live()`, `delegate()`, `on()` 的区别

- `bind()` 直接绑定在目标元素上
- `live()` 通过冒泡传播事件，默认 `document` 上，支持动态数据
- `delegate()` 更精确的小范围使用事件代理，性能优于 live
- `on()` 是最新的 1.9 版本整合了之前的三种方式的新事件绑定机制

#8 是否知道自定义事件

- 事件即“发布/订阅”模式，自定义事件即“消息发布”，事件的监听即“订阅订阅”
- JS 原生支持自定义事件，示例：

```
document.createEvent(type); // 创建事件
event.initEvent(eventType, canBubble, prevent); // 初始化事件
target.addEventListener('dataavailable', handler, false); // 监听事件
target.dispatchEvent(e); // 触发事件
```

- jQuery 里的 `fire` 函数用于调用 jquery 自定义事件列表中的事件

#9 jQuery 通过哪个方法和 Sizzle 选择器结合的

- `sizzle` 选择器采取 `Right To Left` 的匹配模式，先搜寻所有匹配标签，再判断它的父节点
- `jquery` 通过 `$(selecter).find(selecter)` 和 `sizzle` 选择器结合

#10 jQuery 中如何将数组转化为 JSON 字符串，然后再转化回来

```
// 通过原生 JSON.stringify/JSON.parse 扩展 jQuery 实现
$.array2json = function(array) {
    return JSON.stringify(array);
}

$.json2array = function(array) {
    // $.parseJSON(array); // 3.0 开始，已过时
    return JSON.parse(array);
}

// 调用
var json = $.array2json(['a', 'b', 'c']);
var array = $.json2array(json);
```

#11 jQuery 一个对象可以同时绑定多个事件，这是如何实现的

```
$("#btn").on("mouseover mouseout", func);

$("#btn").on({
    mouseover: func1,
    mouseout: func2,
    click: func3
});
```

#12 针对 jQuery 的优化方法

- 缓存频繁操作 DOM 对象
- 尽量使用 id 选择器代替 class 选择器
- 总是从 #id 选择器来继承
- 尽量使用链式操作
- 使用时间委托 on 绑定事件
- 采用 jquery 的内部函数 data() 来存储数据
- 使用最新版本的 jquery

#13 jQuery 的 slideUp 动画，当鼠标快速连续触发，动画会滞后反复执行，该如何处理呢

- 在触发元素上的事件设置为延迟处理：使用 js 原生 setTimeout 方法
- 在触发元素的事件时预先停止所有的动画，再执行相应的动画事件：

```
$('.tab').stop().slideUp();
```

#14 jQuery UI 如何自定义组件

- 通过向 \$.widget() 传递组件名称和一个原型对象来完成
- \$.widget("ns.widgetName", [baseWidget], widgetPrototype);

#15 jQuery 与 jQuery UI、jQuery Mobile 区别

- `jquery` 是 `JS` 库，兼容各种PC浏览器，主要用作更方便地处理 `DOM`、事件、动画、`AJAX`
- `jquery UI` 是建立在 `jquery` 库上的一组用户界面交互、特效、小部件及主题
- `jquery Mobile` 以 `jquery` 为基础，用于创建“移动Web应用”的框架

#16 jQuery 和 Zepto 的区别？各自的使用场景

- `jquery` 主要目标是 `PC` 的网页中，兼容全部主流浏览器。在移动设备方面，单独推出 `jQuery Mobile`
- `zepto` 从一开始就定位移动设备，相对更轻量级。它的 API 基本兼容 `jQuery`，但对 `PC` 浏览器兼容不理想

#17 jQuery对象的特点

- 只有 `JQuery` 对象才能使用 `JQuery` 方法
- `JQuery` 对象是一个数组对象

五、Bootstrap

#1 什么是Bootstrap？以及为什么要使用Bootstrap？

`Bootstrap` 是一个用于快速开发 `Web` 应用程序和网站的前端框架。`Bootstrap` 是基于 `HTML`、`CSS`、`JAVASCRIPT` 的

- `Bootstrap` 具有移动设备优先、浏览器支持良好、容易上手、响应式设计等优点，所以 `Bootstrap` 被广泛应用

#2 使用Bootstrap时，要声明的文档类型是什么？以及为什么要这样声明？

- 使用 `Bootstrap` 时，需要使用 `HTML5` 文档类型（`Doctype`）。`<!DOCTYPE html>`
- 因为 `Bootstrap` 使用了一些 `HTML5` 元素和 `css` 属性，如果在 `Bootstrap` 创建的网页开头不使用 `HTML5` 的文档类型（`Doctype`），可能会面临一些浏览器显示不一致的问题，甚至可能面临一些特定情境下的不一致，以致于代码不能通过 `w3c` 标准的验证

#3 什么是Bootstrap网格系统

`Bootstrap` 包含了一个响应式的、移动设备优先的、不固定的网格系统，可以随着设备或视口大小的增加而适当地扩展到 `12` 列。它包含了用于简单的布局选项的预定义类，也包含了用于生成更多语义布局的功能强大的混合类

- 响应式网格系统随着屏幕或视口（`viewport`）尺寸的增加，系统会自动分为最多 `12` 列。

#4 Bootstrap 网格系统（Grid System）的工作原理

- (1) 行必须放置在 `.container class` 内，以便获得适当的对齐（`alignment`）和内边距（`padding`）。
- (2) 使用行来创建列的水平组。
- (3) 内容应该放置在列内，且唯有列可以是行的直接子元素。
- (4) 预定义的网格类，比如 `.row` 和 `.col-xs-4`，可用于快速创建网格布局。`LESS` 混合类可用于更多语义布局。

- (5) 列通过内边距 (padding) 来创建列内容之间的间隙。该内边距是通过 `.rows` 上的外边距 (margin) 取负，表示第一列和最后一列的行偏移。
- (6) 网格系统是通过指定您想要横跨的十二个可用的列来创建的。例如，要创建三个相等的列，则使用三个 `.col-xs-4`

#5 对于各类尺寸的设备，Bootstrap设置的class前缀分别是什么

- 超小设备手机 (`<768px`) : `.col-xs-*`
- 小型设备平板电脑 (`>=768px`) : `.col-sm-*`
- 中型设备台式电脑 (`>=992px`) : `.col-md-*`
- 大型设备台式电脑 (`>=1200px`) : `.col-lg-*`

#6 Bootstrap 网格系统列与列之间的间隙宽度是多少

间隙宽度为 `30px` (一个列的每边分别是 `15px`)

#7 如果需要在一个标题的旁边创建副标题，可以怎样操作

在元素两旁添加 `<small>`，或者添加 `.small` 的 class

#8 用Bootstrap，如何设置文字的对齐方式？

- `class="text-center"` 设置居中文本
- `class="text-right"` 设置向右对齐文本
- `class="text-left"` 设置向左对齐文本

#9 Bootstrap如何设置响应式表格？

增加 `class="table-responsive"`

#10 使用Bootstrap创建垂直表单的基本步骤？

- (1) 向父 `<form>` 元素添加 `role="form"`；
- (2) 把标签和控件放在一个带有 `class="form-group"` 的 `<div>` 中，这是获取最佳间距所必需的；
- (3) 向所有的文本元素 `<input>`、`<textarea>`、`<select>` 添加 `class="form-control"`

#11 使用Bootstrap创建水平表单的基本步骤？

- (1) 向父 `<form>` 元素添加 `class="form-horizontal"`；
- (2) 把标签和控件放在一个带有 `class="form-group"` 的 `<div>` 中；
- (3) 向标签添加 `class="control-label"`。

#12 使用Bootstrap如何创建表单控件的帮助文本？

增加 `class="help-block"` 的 `span` 标签或 `p` 标签。

#13 使用Bootstrap激活或禁用按钮要如何操作？

- 激活按钮：给按钮增加 `.active` 的 class
- 禁用按钮：给按钮增加 `disabled="disabled"` 的属性

#14 Bootstrap有哪些关于的class?

- (1) `.img-rounded` 为图片添加圆角
- (2) `.img-circle` 将图片变为圆形
- (3) `.img-thumbnail` 缩略图功能
- (4) `.img-responsive` 图片响应式 (将很好地扩展到父元素)

#15 Bootstrap中有关元素浮动及清除浮动的class?

- (1) `class="pull-left"` 元素浮动到左边
- (2) `class="pull-right"` 元素浮动到右边
- (3) `class="clearfix"` 清除浮动

#16 除了屏幕阅读器外，其他设备上隐藏元素的class?

``class="sr-only"```

#17 Bootstrap如何制作下拉菜单?

- (1) 将下拉菜单包裹在 `class="dropdown"` 的 `<div>` 中；
- (2) 在触发下拉菜单的按钮中添加: `class="btn dropdown-toggle" id="dropdownMenu1" data-toggle="dropdown"`
- (3) 在包裹下拉菜单的ul中添加: `class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu1"`
- (4) 在下拉菜单的列表项中添加: `role="presentation"`。其中，下拉菜单的标题要添加 `class="dropdown-header"`，选项部分要添加 `tabindex="-1"`。

#18 Bootstrap如何制作按钮组？以及水平按钮组和垂直按钮组的优先级？

- (1) 用 `class="btn-group"` 的 `<div>` 去包裹按钮组；`class="btn-group-vertical"` 可设置垂直按钮组。
- (2) `btn-group` 的优先级高于 `btn-group-vertical` 的优先级。

#19 Bootstrap如何设置按钮的下拉菜单？

在一个 `.btn-group` 中放置按钮和下拉菜单即可。

#20 Bootstrap中的输入框组如何制作？

- (1) 把前缀或者后缀元素放在一个带有 `class="input-group"` 中的 `<div>` 中
- (2) 在该 `<div>` 内，在 `class="input-group-addon"` 的 `` 里面放置额外的内容；
- (3) 把 `` 放在 `<input>` 元素的前面或后面。

#21 Bootstrap中的导航都有哪些？

- (1) 导航元素：有 `class="nav nav-tabs"` 的标签页导航，还有 `class="nav nav-pills"` 的胶囊式标签页导航；
- (2) 导航栏: `class="navbar navbar-default" role="navigation"`；
- (3) 面包屑导航: `class="breadcrumb"`

#22 Bootstrap中设置分页的class?

- 默认的分页: `class="pagination"`
- 默认的翻页: `class="pager"`

#23 Bootstrap中显示标签的class?

```
class="label"
```

#24 Bootstrap中如何制作徽章?

```
<span class="badge">26</span>
```

#25 Bootstrap中超大屏幕的作用是什么?

设置`class="jumbotron"`可以制作超大屏幕，该组件可以增加标题的大小并增加更多的外边距

六、微信小程序

#1 微信小程序有几个文件

- `WXSS (Weixin style sheets)`是一套样式语言，用于描述`WXML`的组件样式，`js`逻辑处理，网络请求`json`小程序设置，如页面注册，页面标题及`tabBar`。
- `app.json`必须要有这个文件，如果没有这个文件，项目无法运行，因为微信框架把这个作为配置文件入口，整个小程序的全局配置。包括页面注册，网络设置，以及小程序的`window`背景色，配置导航条样式，配置默认标题。
- `app.js`必须要有这个文件，没有也是会报错！但是这个文件创建一下就行 什么都不需要写以后我们可以在这个文件中监听并处理小程序的生命周期函数、声明全局变量。
- `app.wxss`配置全局`css`

#2 微信小程序怎样跟事件传值

给`HTML`元素添加`data-*`属性来传递我们需要的值，然后通过`e.currentTarget.dataset`或`onload`的`param`参数获取。但`data -`名称不能有大写字母和不可以存放对象

#3 小程序的wxss和css有哪些不一样的地方?

- `wxss`的图片引入需使用外链地址
- 没有`Body`；样式可直接使用`import`导入

#4 小程序关联微信公众号如何确定用户的唯一性

使用`wx.getUserInfo`方法`withCredentials`为`true`时可获取`encryptedData`，里面有`union_id`。后端需要进行对称解密

#5 微信小程序与vue区别

- 生命周期不一样，微信小程序生命周期比较简单
- 数据绑定也不同，微信小程序数据绑定需要使用`{}{}`，`vue`直接`:`就可以
- 显示与隐藏元素，`vue`中，使用`v-if`和`v-show`控制元素的显示和隐藏，小程序中，使用`wx-if`和`hidden`控制元素的显示和隐藏

- 事件处理不同，小程序中，全用 `bindtap(bind+event)`，或者 `catchtap(catch+event)` 绑定事件，`vue:` 使用 `v-on:event` 绑定事件，或者使用 `@event` 绑定事件
- 数据双向绑定也不不一样在 `vue` 中，只需要再表单元素上加上 `v-model`，然后再绑定 `data` 中对应的一个值，当表单元素内容发生变化时，`data` 中对应的值也会相应改变，这是 `vue` 非常 nice 的一点。微信小程序必须获取到表单元素，改变的值，然后再把值赋给一个 `data` 中声明的变量。

#七、webpack相关

#1 打包体积 优化思路

- 提取第三方库或通过引用外部文件的方式引入第三方库
- 代码压缩插件 `uglifyjsPlugin`
- 服务器启用 gzip 压缩
- 按需加载资源文件 `require.ensure`
- 优化 `devtool` 中的 `source-map`
- 剥离 `css` 文件，单独打包
- 去除不必要的插件，通常就是开发环境与生产环境用同一套配置文件导致

#2 打包效率

- 开发环境采用增量构建，启用热更新
- 开发环境不做无意义的工作如提取 `css` 计算文件hash等
- 配置 `devtool`
- 选择合适的 `loader`
- 个别 `loader` 开启 `cache` 如 `babel-loader`
- 第三方库采用引入方式
- 提取公共代码
- 优化构建时的搜索路径 指明需要构建目录及不需要构建目录
- 模块化引入需要的部分

#3 Loader

编写一个loader

`Loader` 就是一个 `node` 模块，它输出了一个函数。当某种资源需要用这个 `Loader` 转换时，这个函数会被调用。并且，这个函数可以通过提供给它的 `this` 上下文访问 `Loader API`。`reverse-txt-loader`

```
// 定义
module.exports = function(src) {
  //src是原文件内容 (abcde)，下面对内容进行处理，这里是反转
  var result = src.split('').reverse().join('');
  //返回JavaScript源码，必须是String或者Buffer
  return `module.exports = '${result}'`;
}

//使用
{
  test: /\.txt$/,
  use: [
    {
      './path/reverse-txt-loader'
    }
  ]
}
```

```
    ],
},
```

#4 说一下webpack的一些plugin，怎么使用webpack对项目进行优化

构建优化

- 减少编译体积 `contextReplacementPlugin`、`IgnorePlugin`、`babel-plugin-import`、`babel-plugin-transform-runtime`
- 并行编译 `happypack`、`thread-loader`、`uglifyjswebpackPlugin`开启并行
- 缓存 `cache-loader`、`hard-source-webpack-plugin`、`uglifyjswebpackPlugin`开启缓存、`babel-loader`开启缓存
- 预编译 `dllWebpackPlugin && DllReferencePlugin`、`auto-dll-webpack-plugin`

性能优化

- 减少编译体积 `Tree-shaking`、`Scope Hoisting`
- `hash`缓存 `webpack-md5-plugin`
- 拆包 `splitChunksPlugin`、`import()`、`require.ensure`

八、编程题

#1 写一个通用的事件侦听器函数

```
// event(事件)工具集，来源: github.com/markyun
markyun.Event = {

    // 视能力分别使用dom0|dom2|IE方式 来绑定事件
    // 参数: 操作的元素,事件名称 ,事件处理程序
    addEvent : function(element, type, handler) {
        if (element.addEventListener) {
            //事件类型、需要执行的函数、是否捕捉
            element.addEventListener(type, handler, false);
        } else if (element.attachEvent) {
            element.attachEvent('on' + type, function() {
                handler.call(element);
            });
        } else {
            element['on' + type] = handler;
        }
    },
    // 移除事件
    removeEvent : function(element, type, handler) {
        if (element.removeEventListener) {
            element.removeEventListener(type, handler, false);
        } else if (element.detachEvent) {
            element.detachEvent('on' + type, handler);
        } else {
            element['on' + type] = null;
        }
    },
    // 阻止事件 (主要是事件冒泡, 因为IE不支持事件捕获)
    stopPropagation : function(ev) {
```

```

        if (ev.stopPropagation) {
            ev.stopPropagation();
        } else {
            ev.cancelBubble = true;
        }
    },
    // 取消事件的默认行为
    preventDefault : function(event) {
        if (event.preventDefault) {
            event.preventDefault();
        } else {
            event.returnValue = false;
        }
    },
    // 获取事件目标
    getTarget : function(event) {
        return event.target || event.srcElement;
    }
}

```

#2 如何判断一个对象是否为数组

```

function isArray(arg) {
    if (typeof arg === 'object') {
        return Object.prototype.toString.call(arg) === '[object Array]';
    }
    return false;
}

```

#3 冒泡排序

- 每次比较相邻的两个数，如果后一个比前一个小，换位置

```

var arr = [3, 1, 4, 6, 5, 7, 2];

function bubbleSort(arr) {
    for (var i = 0; i < arr.length - 1; i++) {
        for(var j = 0; j < arr.length - i - 1; j++) {
            if(arr[j + 1] < arr[j]) {
                var temp;
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}

console.log(bubbleSort(arr));

```

#4 快速排序

采用二分法，取出中间数，数组每次和中间数比较，小的放到左边，大的放到右边

快速排序的思想很简单，整个排序过程只需要三步：

- (1) 在数据集之中，找一个基准点
- (2) 建立两个数组，分别存储左边和右边的数组
- (3) 利用递归进行下次比较

```
var arr = [3, 1, 4, 6, 5, 7, 2];

function quicksort(arr) {
    if(arr.length == 0) {
        return [];
    }

    var cIndex = Math.floor(arr.length / 2);
    var c = arr.splice(cIndex, 1);
    var l = [];
    var r = [];

    for (var i = 0; i < arr.length; i++) {
        if(arr[i] < c) {
            l.push(arr[i]);
        } else {
            r.push(arr[i]);
        }
    }

    return quicksort(l).concat(c, quicksort(r));
}

console.log(quicksort(arr));
```

#5 编写一个方法 求一个字符串的字节长度

- 假设：一个英文字符占用一个字节，一个中文字符占用两个字节

```
function GetBytes(str){

    var len = str.length;

    var bytes = len;

    for(var i=0; i<len; i++){

        if (str.charCodeAt(i) > 255) bytes++;

    }

    return bytes;

}

alert(GetBytes("你好,as"));
```

#6 bind的用法，以及如何实现bind的函数和需要注意的点

- `bind` 的作用与 `call` 和 `apply` 相同，区别是 `call` 和 `apply` 是立即调用函数，而 `bind` 是返回了一个函数，需要调用的时候再执行。一个简单的 `bind` 函数实现如下

```
Function.prototype.bind = function(ctx) {
    var fn = this;
    return function() {
        fn.apply(ctx, arguments);
    };
};
```

#7 实现一个函数clone

可以对 `JavaScript` 中的 5 种主要的数据类型，包括 `Number`、`String`、`Object`、`Array`、`Boolean` 进行值复

- 考察点1：对于基本数据类型和引用数据类型在内存中存放的是值还是指针这一区别是否清楚
- 考察点2：是否知道如何判断一个变量是什么类型的
- 考察点3：递归算法的设计

```
// 方法一：
Object.prototype.clone = function(){
    var o = this.constructor === Array ? [] : {};
    for(var e in this){
        o[e] = typeof this[e] === "object" ? this[e].clone() :
this[e];
    }
    return o;
}

//方法二：
/**
 * 克隆一个对象
 * @param Obj
 * @returns
 */
function clone(Obj) {
    var buf;
    if (Obj instanceof Array) {
        buf = []; // 创建一个空的数组
        var i = Obj.length;
        while (i--) {
            buf[i] = clone(Obj[i]);
        }
        return buf;
    }else if (Obj instanceof Object){
        buf = {};// 创建一个空对象
        for (var k in Obj) { // 为这个对象添加新的属性
            buf[k] = clone(Obj[k]);
        }
        return buf;
    }else{ // 普通变量直接赋值
        return Obj;
    }
}
```

#8 下面这个ul，如何点击每一列的时候alert其index

□考察闭包

```
<ul id="test">
    <li>这是第一条</li>
    <li>这是第二条</li>
    <li>这是第三条</li>
</ul>
```

```
// 方法一:
var lis=document.getElementById('2223').getElementsByTagName('li');
for(var i=0;i<3;i++){
    lis[i].index=i;
    lis[i].onclick=function(){
        alert(this.index);
    }
}

//方法二:
var lis=document.getElementById('2223').getElementsByTagName('li');
for(var i=0;i<3;i++){
    lis[i].index=i;
    lis[i].onclick=(function(a){
        return function() {
            alert(a);
        }
    })(i);
}
```

#9 定义一个log方法，让它可以代理console.log的方法

可行的方法一：

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

log("hello world!") // hello world!
```

如果要传入多个参数呢？显然上面的方法不能满足要求，所以更好的方法是：

```
function log(){
    console.log.apply(console, arguments);
};
```

#10 输出今天的日期

以`YYYY-MM-DD`的方式，比如今天是2014年9月26日，则输出2014-09-26

```
var d = new Date();
// 获取年, getFullYear()返回4位的数字
var year = d.getFullYear();
// 获取月, 月份比较特殊, 0是1月, 11是12月
var month = d.getMonth() + 1;
// 变成两位
month = month < 10 ? '0' + month : month;
// 获取日
var day = d.getDate();
day = day < 10 ? '0' + day : day;
alert(year + '-' + month + '-' + day);
```

#11 用js实现随机选取10-100之间的10个数字，存入一个数组，并排序

```
var iArray = [];
function getRandom(istart, iend){
    var ichoice = istart - iend +1;
    return Math.floor(Math.random() * ichoice + istart);
}
for(var i=0; i<10; i++){
    iArray.push(getRandom(10,100));
}
iArray.sort();
```

#12 写一段JS程序提取URL中的各个GET参数

有这样一个URL：`http://item.taobao.com/item.htm?a=1&b=2&c=&d=xxx&e`，请写一段JS程序提取URL中的各个GET参数(参数名和参数个数不确定)，将其按`key-value`形式返回到一个`json`结构中，如`{a:'1', b:'2', c:'', d:'xxx', e:undefined}`

```
function serializeUrl(url) {
    var result = {};
    url = url.split("?",1);
    var map = url.split("&");
    for(var i = 0, len = map.length; i < len; i++) {
        result[map[i].split("=")[0]] = map[i].split("=")[1];
    }
    return result;
}
```

#13 写一个`function`，清除字符串前后的空格

使用自带接口`trim()`，考虑兼容性：

```

if (!String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+/, "").replace(/\s+$/, "");
    }
}

// test the function
var str = " \t\n test string ".trim();
alert(str == "test string"); // alerts "true"

```

#14 实现每隔一秒钟输出1,2,3...数字

```

for(var i=0;i<10;i++){
(function(j){
    setTimeout(function(){
        console.log(j+1)
    },j*1000)
})(i)
}

```

#15 实现一个函数，判断输入是不是回文字符串

```

function run(input) {
    if (typeof input !== 'string') return false;
    return input.split('').reverse().join('') === input;
}

```

#16、数组扁平化处理

实现一个 `flatten` 方法，使得输入一个数组，该数组里面的元素也可以是数组，该方法会输出一个扁平化的数组

```

function flatten(arr){
    return arr.reduce(function(prev,item){
        return prev.concat(Array.isArray(item)?flatten(item):item);
    },[]);
}

```

#17、实现一个函数`clone`，可以对JavaScript中的5种主要的数据类型（包括Number、String、Object、Array、Boolean）进行值复制

```

Object.prototype.clone = function(){
    var o = this.constructor === Array ? [] : {};
    for(var e in this){
        o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];
    }
    return o;
}

```

#九、其他

#1 负载均衡

多台服务器共同协作，不让其中某一台或几台超额工作，发挥服务器的最大作用

- http 重定向负载均衡：调度者根据策略选择服务器以302响应请求，缺点只有第一次有效果，后续操作维持在该服务器
- dns负载均衡：解析域名时，访问多个 ip 服务器中的一个（可监控性较弱）
- 反向代理负载均衡：访问统一的服务器，由服务器进行调度访问实际的某个服务器，对统一的服务器要求大，性能受到 服务器群的数量

#2 CDN

内容分发网络，基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。

#3 内存泄漏

定义：程序中已动态分配的堆内存由于某种原因程序未释放或无法释放引发的各种问题。

js中可能出现的内存泄漏情况

结果：变慢，崩溃，延迟大等，原因：

- 全局变量
- dom 清空时，还存在引用
- ie 中使用闭包
- 定时器未清除
- 子元素存在引起的内存泄露

避免策略

- 减少不必要的全局变量，或者生命周期较长的对象，及时对无用的数据进行垃圾回收；
- 注意程序逻辑，避免“死循环”之类的；
- 避免创建过多的对象 原则：不用了的东西要及时归还。
- 减少层级过多的引用

#4 babel原理

ES6、7 代码输入 -> babylon 进行解析 -> 得到 AST (抽象语法树) -> plugin 用babel-traverse 对 AST 树进行遍历转译 -> 得到新的 AST 树-> 用 babel-generator 通过 AST 树生成 ES5 代码

#5 js自定义事件

三要素：document.createEvent() event.initEvent() element.dispatchEvent()

```
// (en:自定义事件名称, fn:事件处理函数, addEvent:为DOM元素添加自定义事件, triggerEvent:触发自定义事件)
window.onload = function(){
    var demo = document.getElementById("demo");
    demo.addEvent("test",function(){console.log("handler1")});
    demo.addEvent("test",function(){console.log("handler2")});
    demo.onclick = function(){
        this.triggerEvent("test");
    }
}
```

```

}
Element.prototype.addEvent = function(en,fn){
    this.pools = this.pools || {};
    if(en in this.pools){
        this.pools[en].push(fn);
    }else{
        this.pools[en] = [];
        this.pools[en].push(fn);
    }
}
Element.prototype.triggerEvent = function(en){
    if(en in this.pools){
        var fns = this.pools[en];
        for(var i=0,il=fn.length;i<il;i++){
            fns[i]();
        }
    }else{
        return;
    }
}

```

#6 前后端路由差别

- 后端每次路由请求都是重新访问服务器
- 前端路由实际上只是 JS 根据 URL 来操作 DOM 元素，根据每个页面需要的去服务端请求数据，返回数据后和模板进行组合

#十、综合

#1 谈谈你对重构的理解

- 网站重构：在不改变外部行为的前提下，简化结构、添加可读性，而在网站前端保持一致的行为。也就是说是在不改变UI的情况下，对网站进行优化，在扩展的同时保持一致的UI
- 对于传统的网站来说重构通常是：
 - 表格(table)布局改为 DIV+CSS
 - 使网站前端兼容于现代浏览器(针对于不合规范的 css、如对IE6有效的)
 - 对于移动平台的优化
 - 针对于 SEO 进行优化

#2 什么样的前端代码是好的

- 高复用低耦合，这样文件小，好维护，而且好扩展。
- 具有可用性、健壮性、可靠性、宽容性等特点
- 遵循设计模式的六大原则

#3 对前端工程师这个职位是怎么样理解的？它的前景会怎么样

- 前端是最贴近用户的程序员，比后端、数据库、产品经理、运营、安全都近
 - 实现界面交互
 - 提升用户体验
 - 基于NodeJS，可跨平台开发
- 前端是最贴近用户的程序员，前端的能力就是能让产品从 90 分进化到 100 分，甚至更好，

- 与团队成员，UI设计，产品经理的沟通；
- 做好的页面结构，页面重构和用户体验；

#4 你觉得前端工程的价值体现在哪

- 为简化用户使用提供技术支持（交互部分）
- 为多个浏览器兼容性提供支持
- 为提高用户浏览速度（浏览器性能）提供支持
- 为跨平台或者其他基于webkit或其他渲染引擎的应用提供支持
- 为展示数据提供支持（数据接口）

#5 平时如何管理你的项目

- 先期团队必须确定好全局样式（`globe.css`），编码模式(`utf-8`)等；
- 编写习惯必须一致（例如都是采用继承式的写法，单样式都写成一行）；
- 标注样式编写人，各模块都及时标注（标注关键样式调用的地方）；
- 页面进行标注（例如 页面 模块 开始和结束）；
- CSS 跟 HTML 分文件夹并行存放，命名都得统一（例如 `style.css`）；
- JS 分文件夹存放 命名以该 JS 功能为准的英文翻译。
- 图片采用整合的 `images.png png8` 格式文件使用 - 尽量整合在一起使用方便将来的管理
- 规定全局样式、公共脚本
- 严格要求代码注释(html/js/css)
- 严格要求静态资源存放路径
- Git 提交必须填写说明

#6 组件封装

目的：为了重用，提高开发效率和代码质量 注意：低耦合，单一职责，可复用性，可维护性 常用操作

- 分析布局
- 初步开发
- 化繁为简
- 组件抽象

#7 Web 前端开发的注意事项

- 特别设置 meta 标签 `viewport`
- 百分比布局宽度，结合 `box-sizing: border-box;`
- 使用 rem 作为计算单位。rem 只参照跟节点 html 的字体大小计算
- 使用 css3 新特性。弹性盒模型、多列布局、媒体查询等
- 多机型、多尺寸、多系统覆盖测试

#8 在设计 Web APP 时，应当遵循以下几点

- 简化不重要的动画/动效/图形文字样式
- 少用手势，避免与浏览器手势冲突
- 减少页面内容，页面跳转次数，尽量在当前页面显示
- 增强 `Loading` 趣味性，增强页面主次关系

#9 你怎么看待 Web App/hybrid App/Native App? (移动端前端和 Web 前端区别?)

- Web App(HTML5): 采用HTML5生存在浏览器中的应用，不需要下载安装
 - 优点: 开发成本低, 迭代更新容易, 不需用户升级, 跨多个平台和终端
 - 缺点: 消息推送不够及时, 支持图形和动画效果较差, 功能使用限制 (相机、GPS等)
- Hybrid App(混合开发): UI WebView, 需要下载安装
 - 优点: 接近 Native App 的体验, 部分支持离线功能
 - 缺点: 性能速度较慢, 未知的部署时间, 受限于技术尚不成熟
- Native App(原生开发): 依托于操作系统, 有很强的交互, 需要用户下载安装使用
 - 优点: 用户体验完美, 支持离线工作, 可访问本地资源 (通讯录, 相册)
 - 缺点: 开发成本高 (多系统), 开发成本高 (版本更新), 需要应用商店的审核

#10 页面重构怎么操作

网站重构: 不改变UI的情况下, 对网站进行优化, 在扩展的同时保持一致的UI。

- 页面重构可以考虑的方面:
 - 升级第三方依赖
 - 使用 `HTML5`、`CSS3`、`ES6` 新特性
 - 加入响应式布局
 - 统一代码风格规范
 - 减少代码间的耦合
 - 压缩/合并静态资源
 - 程序的性能优化
 - 采用 CDN 来加速资源加载
 - 对于 `JS DOM` 的优化
 - HTTP服务器的文件缓存

中级篇

#一、JS

#1 谈谈变量提升

当执行 `JS` 代码时, 会生成执行环境, 只要代码不是写在函数中的, 就是在全局执行环境中, 函数中的代码会产生函数执行环境, 只此两种执行环境。

```
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
  console.log('call b')
}
```

想必以上的输出大家肯定都已经明白了，这是因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行环境时，会有两个阶段。第一个阶段是创建的阶段，JS 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 `undefined`，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用

- 在提升的过程中，相同的函数会覆盖上一个函数，并且函数优先于变量提升

```
b() // call b second

function b() {
  console.log('call b fist')
}

function b() {
  console.log('call b second')
}

var b = 'Hello world'
```

`var` 会产生很多错误，所以在 ES6 中引入了 `let`。`let` 不能在声明前使用，但是这并不是常说的 `let` 不会提升，`let` 提升了，在第一阶段内存也已经为他开辟好了空间，但是因为这个声明的特性导致了并不能在声明前使用

#2 bind、call、apply 区别

- `call` 和 `apply` 都是为了解决改变 `this` 的指向。作用都是相同的，只是传参的方式不同。
- 除了第一个参数外，`call` 可以接收一个参数列表，`apply` 只接受一个参数数组

```
let a = {
  value: 1
}

function getValue(name, age) {
  console.log(name)
  console.log(age)
  console.log(this.value)
}

getValue.call(a, 'yck', '24')
getValue.apply(a, ['yck', '24'])
```

`bind` 和其他两个方法作用也是一致的，只是该方法会返回一个函数。并且我们可以通过 `bind` 实现柯里化

#3 如何实现一个 bind 函数

对于实现以下几个函数，可以从几个方面思考

- 不传入第一个参数，那么默认为 `window`
- 改变了 `this` 指向，让新的对象可以执行该函数。那么思路是否可以变成给新的对象添加一个函数，然后在执行完以后删除？

```
Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  var _this = this
  var args = [...arguments].slice(1)
```

```
// 返回一个函数
return function F() {
    // 因为返回了一个函数，我们可以 new F()，所以需要判断
    if (this instanceof F) {
        return new _this(...args, ...arguments)
    }
    return _this.apply(context, args.concat(...arguments))
}
```

#4 如何实现一个 call 函数

```
Function.prototype.myCall = function (context) {
    var context = context || window
    // 给 context 添加一个属性
    // getValue.call(a, 'yck', '24') => a.fn = getValue
    context.fn = this
    // 将 context 后面的参数取出来
    var args = [...arguments].slice(1)
    // getValue.call(a, 'yck', '24') => a.fn('yck', '24')
    var result = context.fn(...args)
    // 删除 fn
    delete context.fn
    return result
}
```

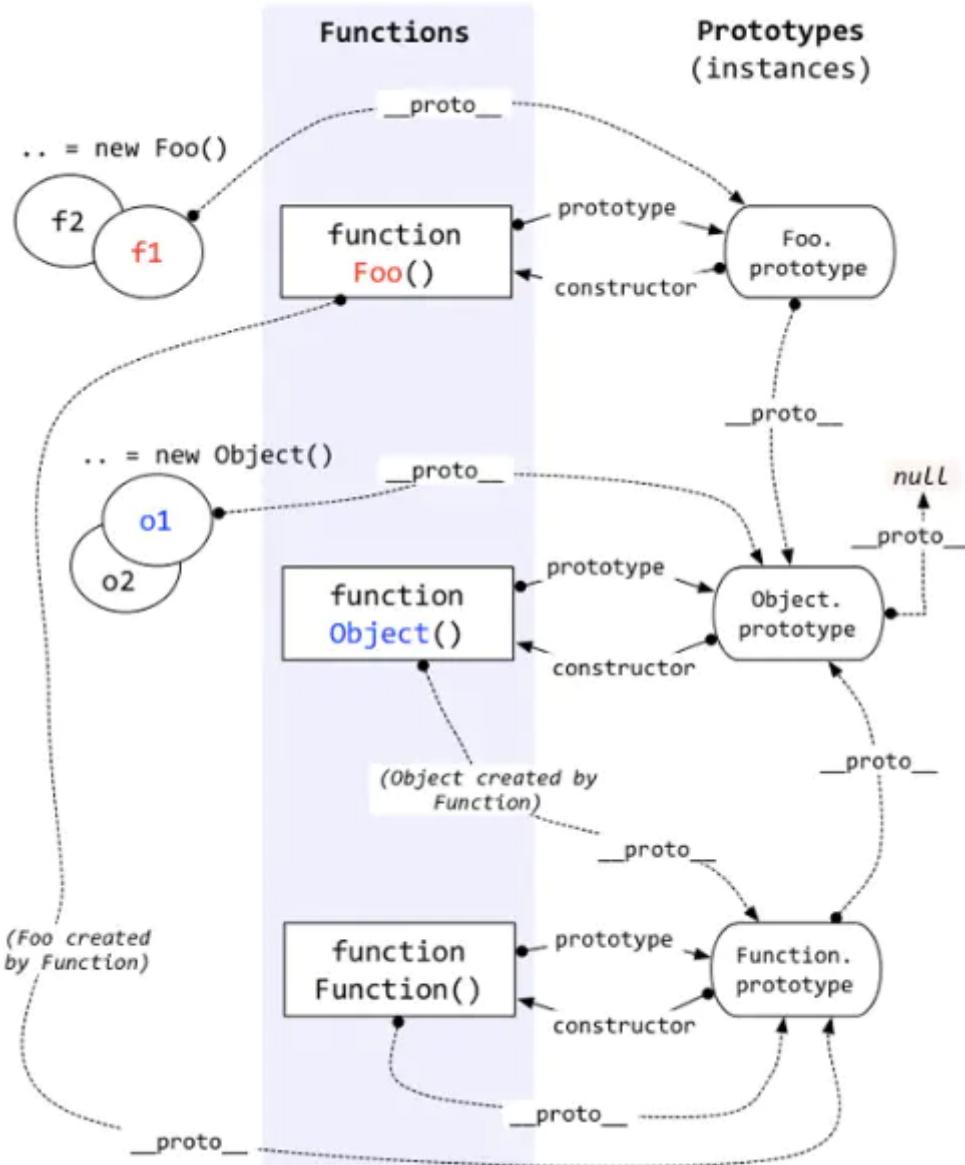
#5 如何实现一个 apply 函数

```
Function.prototype.myApply = function (context) {
    var context = context || window
    context.fn = this

    var result
    // 需要判断是否存储第二个参数
    // 如果存在，就将第二个参数展开
    if (arguments[1]) {
        result = context.fn(...arguments[1])
    } else {
        result = context.fn()
    }

    delete context.fn
    return result
}
```

#6 简单说下原型链？



- 每个函数都有 `prototype` 属性，除了 `Function.prototype.bind()`，该属性指向原型。
- 每个对象都有 `__proto__` 属性，指向了创建该对象的构造函数的原型。其实这个属性指向了 `[[prototype]]`，但是 `[[prototype]]` 是内部属性，我们并不能访问到，所以使用 `__proto__` 来访问。
- 对象可以通过 `__proto__` 来寻找不属于该对象的属性，`__proto__` 将对象连接起来组成了原型链。

#7 怎么判断对象类型

- 可以通过 `Object.prototype.toString.call(xx)`。这样我们就可以获得类似 `[object Type]` 的字符串。
- `instanceof` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 `prototype`

#8 箭头函数的特点

```
function a() {
  return () => {
    return () => {
      console.log(this)
    }
  }
}
console.log(a()())
```

箭头函数其实是没有 `this` 的，这个函数中的 `this` 只取决于他外面的第一个不是箭头函数的函数的 `this`。在这个例子中，因为调用 `a` 符合前面代码中的第一个情况，所以 `this` 是 `window`。并且 `this` 一旦绑定了上下文，就不会被任何代码改变

#9 This

```
function foo() {
  console.log(this.a)
}

var a = 1
foo()

var obj = {
  a: 2,
  foo: foo
}
obj.foo()
```

// 以上两者情况 `this` 只依赖于调用函数前的对象，优先级是第二个情况大于第一个情况

// 以下情况是优先级最高的，`this` 只会绑定在 `c` 上，不会被任何方式修改 `this` 指向

```
var c = new foo()
c.a = 3
console.log(c.a)
```

// 还有种就是利用 `call`, `apply`, `bind` 改变 `this`，这个优先级仅次于 `new`

#10 async、await 优缺点

`async` 和 `await` 相比直接使用 `Promise` 来说，优势在于处理 `then` 的调用链，能够更清晰准确的写出代码。缺点在于滥用 `await` 可能会导致性能问题，因为 `await` 会阻塞代码，也许之后的异步代码并不依赖于前者，但仍然需要等待前者完成，导致代码失去了并发性

下面来看一个使用 `await` 的代码。

```
var a = 0
var b = async () => {
  a = a + await 10
  console.log('2', a) // -> '2' 10
  a = (await 10) + a
  console.log('3', a) // -> '3' 20
}
b()
a++
console.log('1', a) // -> '1' 1
```

- 首先函数 b 先执行，在执行到 `await 10` 之前变量 `a` 还是 `0`，因为在 `await` 内部实现了 `generators`，`generators` 会保留堆栈中东西，所以这时候 `a = 0` 被保存了下来
- 因为 `await` 是异步操作，遇到 `await` 就会立即返回一个 `pending` 状态的 `Promise` 对象，暂时返回执行代码的控制权，使得函数外的代码得以继续执行，所以会先执行 `console.log('1', a)`
- 这时候同步代码执行完毕，开始执行异步代码，将保存下来的值拿出来使用，这时候 `a = 10`
- 然后后面就是常规执行代码了

#11 generator 原理

`Generator` 是 `ES6` 中新增的语法，和 `Promise` 一样，都可以用来异步编程

```
// 使用 * 表示这是一个 Generator 函数
// 内部可以通过 yield 暂停代码
// 通过调用 next 恢复执行
function* test() {
  let a = 1 + 2;
  yield 2;
  yield 3;
}
let b = test();
console.log(b.next()); // > { value: 2, done: false }
console.log(b.next()); // > { value: 3, done: false }
console.log(b.next()); // > { value: undefined, done: true }
```

从以上代码可以发现，加上 `*` 的函数执行后拥有了 `next` 函数，也就是说函数执行后返回了一个对象。每次调用 `next` 函数可以继续执行被暂停的代码。以下是 `Generator` 函数的简单实现

```
// cb 也就是编译过的 test 函数
function generator(cb) {
  return (function() {
    var object = {
      next: 0,
      stop: function() {}
    };

    return {
      next: function() {
        var ret = cb(object);
        if (ret === undefined) return { value: undefined, done: true };
        return {
          value: ret,
          done: false
        };
      }
    };
  })();
}

// 如果你使用 babel 编译后可以发现 test 函数变成了这样
function test() {
  var a;
  return generator(function(_context) {
    while (1) {
      switch ((_context.prev = _context.next)) {
        // 可以发现通过 yield 将代码分割成几块
        // 每次执行 next 函数就执行一块代码
      }
    }
  });
}
```

```

    // 并且表明下次需要执行哪块代码
    case 0:
        a = 1 + 2;
        _context.next = 4;
        return 2;
    case 4:
        _context.next = 6;
        return 3;
        // 执行完毕
    case 6:
        case "end":
            return _context.stop();
    }
}
);
}

```

#12 Promise

- Promise 是 ES6 新增的语法，解决了回调地狱的问题。
- 可以把 Promise 看成一个状态机。初始是 pending 状态，可以通过函数 resolve 和 reject，将状态转变为 resolved 或者 rejected 状态，状态一旦改变就不能再次变化。
- then 函数会返回一个 Promise 实例，并且该返回值是一个新的实例而不是之前的实例。因为 Promise 规范规定除了 pending 状态，其他状态是不可以改变的，如果返回的是一个相同实例的话，多个 then 调用就失去意义了。对于 then 来说，本质上可以把它看成是 flatMap

#13 如何实现一个 Promise

```

// 三种状态
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";
// promise 接收一个函数参数，该函数会立即执行
function MyPromise(fn) {
    let _this = this;
    _this.currentState = PENDING;
    _this.value = undefined;
    // 用于保存 then 中的回调，只有当 promise
    // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
    _this.resolvedCallbacks = [];
    _this.rejectedCallbacks = [];

    _this.resolve = function (value) {
        if (value instanceof MyPromise) {
            // 如果 value 是个 Promise，递归执行
            return value.then(_this.resolve, _this.reject)
        }
        setTimeout(() => { // 异步执行，保证执行顺序
            if (_this.currentState === PENDING) {
                _this.currentState = RESOLVED;
                _this.value = value;
                _this.resolvedCallbacks.forEach(cb => cb());
            }
        })
    }
};

```

```

    _this.reject = function (reason) {
      setTimeout(() => { // 异步执行，保证执行顺序
        if (_this.currentState === PENDING) {
          _this.currentState = REJECTED;
          _this.value = reason;
          _this.rejectedCallbacks.forEach(cb => cb());
        }
      })
    }

  // 用于解决以下问题
  // new Promise(() => throw Error('error'))
  try {
    fn(_this.resolve, _this.reject);
  } catch (e) {
    _this.reject(e);
  }
}

MyPromise.prototype.then = function (onResolved, onRejected) {
  var self = this;
  // 规范 2.2.7, then 必须返回一个新的 promise
  var promise2;
  // 规范 2.2.onResolved 和 onRejected 都为可选参数
  // 如果类型不是函数需要忽略，同时也实现了透传
  // Promise.resolve(4).then().then((value) => console.log(value))
  onResolved = typeof onResolved === 'function' ? onResolved : v => v;
  onRejected = typeof onRejected === 'function' ? onRejected : r => throw r;

  if (self.currentState === RESOLVED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      // 规范 2.2.4, 保证 onFulfilled, onRejected 异步执行
      // 所以用了 setTimeout 包裹下
      setTimeout(function () {
        try {
          var x = onResolved(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (reason) {
          reject(reason);
        }
      });
    }));
  }

  if (self.currentState === REJECTED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      setTimeout(function () {
        // 异步执行onRejected
        try {
          var x = onRejected(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (reason) {
          reject(reason);
        }
      });
    }));
  }

  if (self.currentState === PENDING) {

```

```

        return (promise2 = new MyPromise(function (resolve, reject) {
            self.resolvedCallbacks.push(function () {
                // 考虑到可能会有报错，所以使用 try/catch 包裹
                try {
                    var x = onResolved(self.value);
                    resolutionProcedure(promise2, x, resolve, reject);
                } catch (r) {
                    reject(r);
                }
            });
        });

        self.rejectedCallbacks.push(function () {
            try {
                var x = onRejected(self.value);
                resolutionProcedure(promise2, x, resolve, reject);
            } catch (r) {
                reject(r);
            }
        });
    });
};

// 规范 2.3
function resolutionProcedure(promise2, x, resolve, reject) {
    // 规范 2.3.1, x 不能和 promise2 相同，避免循环引用
    if (promise2 === x) {
        return reject(new TypeError("Error"));
    }

    // 规范 2.3.2
    // 如果 x 为 Promise, 状态为 pending 需要继续等待否则执行
    if (x instanceof MyPromise) {
        if (x.currentState === PENDING) {
            x.then(function (value) {
                // 再次调用该函数是为了确认 x resolve 的
                // 参数是什么类型，如果是基本类型就再次 resolve
                // 把值传给下个 then
                resolutionProcedure(promise2, value, resolve, reject);
            }, reject);
        } else {
            x.then(resolve, reject);
        }
        return;
    }

    // 规范 2.3.3.3
    // reject 或者 resolve 其中一个执行过得话，忽略其他的
    let called = false;

    // 规范 2.3.3, 判断 x 是否为对象或者函数
    if (x !== null && (typeof x === "object" || typeof x === "function")) {
        // 规范 2.3.3.2, 如果不能取出 then, 就 reject
        try {
            // 规范 2.3.3.1
            let then = x.then;
            // 如果 then 是函数，调用 x.then
            if (typeof then === "function") {
                // 规范 2.3.3.3
                then.call(
                    x,
                    y => {

```

```

        if (called) return;
        called = true;
        // 规范 2.3.3.3.1
        resolutionProcedure(promise2, y, resolve, reject);
    },
    e => {
        if (called) return;
        called = true;
        reject(e);
    }
);
} else {
    // 规范 2.3.3.4
    resolve(x);
}
} catch (e) {
    if (called) return;
    called = true;
    reject(e);
}
} else {
    // 规范 2.3.4, x 为基本类型
    resolve(x);
}
}
}

```

#14 == 和 ===区别，什么情况用 ==

这里来解析一道题目 `[] == ![] // -> true`，下面是这个表达式为何为 `true` 的步骤

```

// [] 转成 true, 然后取反变成 false
[] == false
// 根据第 8 条得出
[] == ToNumber(false)
[] == 0
// 根据第 10 条得出
ToPrimitive([]) == 0
// [].toString() -> ''
'' == 0
// 根据第 6 条得出
0 == 0 // -> true

```

`==` 用于判断两者类型和值是否相同。在开发中，对于后端返回的 `code`，可以通过 `==` 去判断

#15 基本数据类型和引用类型在存储上的差别

前者存储在线上，后者存储在堆上

#16 浏览器 Eventloop 和 Node 中的有什么区别

众所周知 JS 是一门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是一门多线程的语言话，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点），当然可以引入读写锁解决这个问题。

- JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task (有多种 task) 队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为

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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

- 以上代码虽然 setTimeout 延时为 0，其实还是异步。这是因为 HTML5 标准规定这个函数第二个参数不得小于 4 毫秒，不足会自动增加。所以 setTimeout 还是会在 script end 之后打印。
- 不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务 (microtask) 和 宏任务 (macrotask)。在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task。

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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
  console.log('Promise')
  resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout
```

- 以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务，所以会有以上的打印。
- 微任务包括** process.nextTick , promise , Object.observe , MutationObserver
- 宏任务包括** script , setTimeout , setInterval , setImmediate , I/O , UI rendering

很多人有个误区，认为微任务快于宏任务，其实是错误的。因为宏任务中包括了 script，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 Event loop，执行宏任务中的异步代码

通过上述的 Event loop 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话，为了更快的界面响应，我们可以把操作 DOM 放入微任务中

#17 setTimeout 倒计时误差

JS 是单线程的，所以 `setTimeout` 的误差其实是无法被完全解决的，原因有很多，可能是回调中的，有可能是浏览器中的各种事件导致。这也是为什么页面开久了，定时器会不准的原因，当然我们可以通过一定的办法去减少这个误差。

```
// 以下是一个相对准备的倒计时实现
var period = 60 * 1000 * 60 * 2
var startTime = new Date().getTime();
var count = 0
var end = new Date().getTime() + period
var interval = 1000
var currentInterval = interval

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

    setTimeout(loop, currentInterval)
}

setTimeout(loop, currentInterval)
```

#18 数组降维

```
[1, [2], 3].flatMap(v => v)
// -> [1, 2, 3]
```

如果想将一个多维数组彻底的降维，可以这样实现

```
const flattenDeep = (arr) => Array.isArray(arr)
? arr.reduce( (a, b) => [...a, ...flattenDeep(b)] , [])
: [arr]

flattenDeep([1, [[2], [3, [4]]], 5])
```

#19 深拷贝

这个问题通常可以通过 `JSON.parse(JSON.stringify(object))` 来解决

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

但是该方法也是有局限性的：

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能序列化函数
- 不能解决循环引用的对象

```
let obj = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  },
}
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)
复
```

在遇到函数、`undefined` 或者 `symbol` 的时候，该对象也不能正常的序列化

```
let a = {
  age: undefined,
  sex: Symbol('male'),
  jobs: function() {},
  name: 'yck'
}
let b = JSON.parse(JSON.stringify(a))
console.log(b) // {name: "yck"}
```

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题，并且该函数是内置函数中处理深拷贝性能最快的。当然如果你的数据中含有以上三种情况下，可以使用 `lodash` 的深拷贝函数

#20 `typeof` 于 `instanceof` 区别

`typeof` 对于基本类型，除了 `null` 都可以显示正确的类型

```
typeof 1 // 'number'  
typeof '1' // 'string'  
typeof undefined // 'undefined'  
typeof true // 'boolean'  
typeof symbol // 'symbol'  
typeof b // b 没有声明，但是还会显示 undefined
```

typeof` 对于对象，除了函数都会显示 `object`

```
typeof [] // 'object'  
typeof {} // 'object'  
typeof console.log // 'function'
```

对于 `null` 来说，虽然它是基本类型，但是会显示 `object`，这是一个存在很久了的 bug

```
typeof null // 'object'
```

`instanceof`` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 `prototype`

```
我们也可以试着实现一下 instanceof  
function instanceof(left, right) {  
    // 获得类型的原型  
    let prototype = right.prototype  
    // 获得对象的原型  
    left = left.__proto__  
    // 判断对象的类型是否等于类型的原型  
    while (true) {  
        if (left === null)  
            return false  
        if (prototype === left)  
            return true  
        left = left.__proto__  
    }  
}
```

#二、浏览器

#1 cookie和localStorage、session、indexDB 的区别

| 特性 | cookie | localStorage | sessionStorage | indexDB |
|--------|---------------------------|--------------|----------------|--------------|
| 数据生命周期 | 一般由服务器生成，可以设置过期时间 | 除非被清理，否则一直存在 | 页面关闭就清理 | 除非被清理，否则一直存在 |
| 数据存储大小 | 4K | 5M | 5M | 无限 |
| 与服务端通信 | 每次都会携带在 header 中，对于请求性能影响 | 不参与 | 不参与 | 不参与 |

从上表可以看到，`cookie` 已经不建议用于存储。如果没有大量数据存储需求的话，可以使用 `localStorage` 和 `sessionStorage`。对于不怎么改变的数据尽量使用 `localStorage` 存储，否则可以用 `sessionStorage` 存储。

对于 `cookie`，我们还需要注意安全性

| 属性 | 作用 |
|------------------------|---|
| <code>value</code> | 如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识 |
| <code>http-only</code> | 不能通过 JS 访问 <code>cookie</code> ，减少 XSS 攻击 |
| <code>secure</code> | 只能在协议为 <code>HTTPS</code> 的请求中携带 |
| <code>same-site</code> | 规定浏览器不能在跨域请求中携带 <code>Cookie</code> ，减少 CSRF 攻击 |

#2 怎么判断页面是否加载完成？

- `Load` 事件触发代表页面中的 `DOM`, `CSS`, `JS`，图片已经全部加载完毕。
- `DOMContentLoaded` 事件触发代表初始的 `HTML` 被完全加载和解析，不需要等待 `CSS`, `JS`，图片加载

#3 如何解决跨域

因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，`Ajax` 请求会失败。

我们可以通过以下几种常用方法解决跨域的问题

JSONP

`JSONP` 的原理很简单，就是利用 `<script>` 标签没有跨域限制的漏洞。通过 `<script>` 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时

```
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
  function jsonp(data) {
    console.log(data)
  }
</script>
```

`JSONP` 使用简单且兼容性不错，但是只限于 `get` 请求

- 在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的，这时候就需要自己封装一个 JSONP，以下是简单实现

```

function jsonp(url, jsonpCallback, success) {
  let script = document.createElement("script");
  script.src = url;
  script.async = true;
  script.type = "text/javascript";
  window[jjsonpCallback] = function(data) {
    success && success(data);
  };
  document.body.appendChild(script);
}

jsonp(
  "http://xxx",
  "callback",
  function(value) {
    console.log(value);
  }
);

```

CORS

- CORS 需要浏览器和后端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。
- 浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。
- 服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

document.domain

- 该方式只能用于二级域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式。
- 只需要给页面添加 document.domain = 'test.com' 表示二级域名都相同就可以实现跨域

postMessage

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

```

// 发送消息端
window.parent.postMessage('message', 'http://test.com');
// 接收消息端
var mc = new MessageChannel();
mc.addEventListener('message', (event) => {
  var origin = event.origin || event.originalEvent.origin;
  if (origin === 'http://test.com') {
    console.log('验证通过')
  }
});

```

#4 什么是事件代理

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```

<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector('#ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>

```

- 事件代理的方式相对于直接给目标注册事件来说，有以下优点
 - 节省内存
 - 不需要给子节点注销事件

#5 Service worker

service worker

`Service workers` 本质上充当Web应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。它们旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作。他们还允许访问推送通知和后台同步API

目前该技术通常用来做缓存文件，提高首屏速度，可以试着来实现这个功能

```

// index.js
if (navigator.serviceWorker) {
    navigator.serviceWorker
        .register("sw.js")
        .then(function(registration) {
            console.log("service worker 注册成功");
        })
        .catch(function(err) {
            console.log("servcie worker 注册失败");
        });
}
// sw.js
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener("install", e => {
    e.waitUntil(
        caches.open("my-cache").then(function(cache) {
            return cache.addAll(["./index.html", "./index.js"]);
        })
    );
});

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener("fetch", e => {
    e.respondWith(
        caches.match(e.request).then(function(response) {
            if (response) {

```

```

        return response;
    }
    console.log("fetch source");
}
);
);
}
);

```

打开页面，可以在开发者工具中的 Application 看到 Service worker 已经启动了

The screenshot shows the Chrome DevTools Application tab with the Service Workers section selected. A service worker named `sw.js` is listed under the address `127.0.0.1`. The status is shown as `#266 activated and is running` with a green dot icon. Below the status, it says `Received 2018/3/28 下午1:34:26`. Under the Clients section, it lists `http://127.0.0.1:5500/index.html` with the `focus` state. There are two buttons at the bottom: `Push` (with the placeholder `Test push message from DevTools.`) and `Sync` (with the placeholder `test-tag-from-devtools`).

#6 浏览器缓存

缓存对于前端性能优化来说是个很重要的点，良好的缓存策略可以降低资源的重复加载提高网页的整体加载速度。

- 通常浏览器缓存策略分为两种：强缓存和协商缓存。

强缓存

实现强缓存可以通过两种响应头实现：`Expires` 和 `Cache-Control`。强缓存表示在缓存期间不需要请求，`state code` 为 `200`

`Expires: wed, 22 Oct 2018 08:41:00 GMT`

`Expires` 是 `HTTP / 1.0` 的产物，表示资源会在 `wed, 22 oct 2018 08:41:00 gmt` 后过期，需要再次请求。并且 `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。

`Cache-control: max-age=30`

- `Cache-Control` 出现于 `HTTP / 1.1`，优先级高于 `Expires`。该属性表示资源会在 `30` 秒后过期，需要再次请求。

协商缓存

- 如果缓存过期了，我们就可以使用协商缓存来解决问题。协商缓存需要请求，如果缓存有效会返回 `304`。
- 协商缓存需要客户端和服务端共同实现，和强缓存一样，也有两种实现方式

Last-Modified 和 If-Modified-Since

- `Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来。
- 但是如果在本地打开缓存文件，就会造成 `Last-Modified` 被修改，所以在 `HTTP / 1.1` 出现了 `ETag`

ETag 和 If-None-Match

`ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来。并且 `ETag` 优先级比 `Last-Modified` 高

选择合适的缓存策略

对于大部分的场景都可以使用强缓存配合协商缓存解决，但是在一些特殊的地方可能需要选择特殊的缓存策略

- 对于某些不需要缓存的资源，可以使用 `Cache-control: no-store`，表示该资源不需要缓存
- 对于频繁变动的资源，可以使用 `Cache-Control: no-cache` 并配合 `ETag` 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新。
- 对于代码文件来说，通常使用 `Cache-Control: max-age=31536000` 并配合策略缓存使用，然后对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件

#7 浏览器性能问题

重绘 (Repaint) 和回流 (Reflow)

- 重绘和回流是渲染步骤中的一小节，但是这两个步骤对于性能影响很大。
- 重绘是当节点需要更改外观而不会影响布局的，比如改变 `color` 就称为重绘
- 回流是布局或者几何属性需要改变就称为回流。
- 回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流。

所以以下几个动作可能会导致性能问题：

- 改变 `window` 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

很多人不知道的是，重绘和回流其实和 Event loop 有关。

- 当 `Event loop` 执行完 `Microtasks` 后，会判断 `document` 是否需要更新。- 因为浏览器是 `60Hz` 的刷新率，每 `16ms` 才会更新一次。
- 然后判断是否有 `resize` 或者 `scroll`，有的话会去触发事件，所以 `resize` 和 `scroll` 事件也是至少 `16ms` 才会触发一次，并且自带节流功能。
- 判断是否触发了 `media query`
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 `requestAnimationFrame` 回调
- 执行 `IntersectionObserver` 回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好
- 更新界面
- 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 `requestIdleCallback` 回调。

减少重绘和回流

使用 `translate` 替代 `top`

```
<div class="test"></div>
<style>
```

```

    .test {
      position: absolute;
      top: 10px;
      width: 100px;
      height: 100px;
      background: red;
    }
</style>
<script>
  setTimeout(() => {
    // 引起回流
    document.querySelector('.test').style.top = '100px'
  }, 1000)
</script>

```

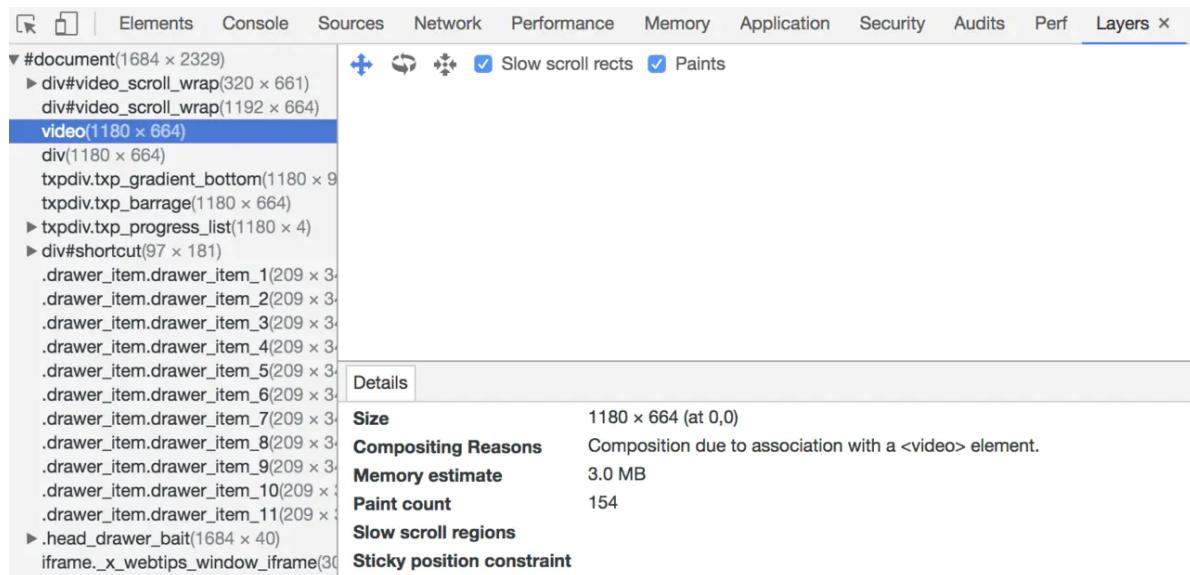
- 使用 `visibility` 替换 `display: none`，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 把 `DOM` 离线后修改，比如：先把 `DOM` 给 `display:none` (有一次 `Reflow`)，然后你修改 100 次，然后再把它显示出来
- 不要把 `DOM` 结点的属性值放在一个循环里当成循环里的变量

```

for(let i = 0; i < 1000; i++) {
  // 获取 offsetTop 会导致回流，因为需要去获取正确的值
  console.log(document.querySelector('.test').style.offsetTop)
}

```

- 不要使用 `table` 布局，可能很小的一个小改动会造成整个 `table` 的重新布局 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 `requestAnimationFrame`
- `CSS` 选择符从右往左匹配查找，避免 `DOM` 深度过深
- 将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 `video` 标签，浏览器会自动将该节点变为图层。



CDN

静态资源尽量使用 `CDN` 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 `CDN` 域名。对于 `CDN` 加载静态资源需要注意 `CDN` 域名要与主站不同，否则每次请求都会带上主站的 `Cookie`

使用 Webpack 优化项目

- 对于 `webpack4`，打包项目使用 `production` 模式，这样会自动开启代码压缩
- 使用 `ES6` 模块来开启 `tree shaking`，这个技术可以移除没有使用的代码
- 优化图片，对于小图可以使用 `base64` 的方式写入文件中
- 按照路由拆分代码，实现按需加载

#三、Webpack

□

#1 优化打包速度

- 减少文件搜索范围
 - 比如通过别名
 - `loader` 的 `test`, `include & exclude`
- `webpack4` 默认压缩并行
- `Happypack` 并发调用
- `babel` 也可以缓存编译

#2 Babel 原理

- 本质就是编译器，当代码转为字符串生成 `AST`，对 `AST` 进行转变最后再生成新的代码
- 分为三步：词法分析生成 `Token`，语法分析生成 `AST`，遍历 `AST`，根据插件变换相应的节点，最后把 `AST` 转换为代码

#3 如何实现一个插件

- 调用插件 `apply` 函数传入 `compiler` 对象
- 通过 `compiler` 对象监听事件

比如你想实现一个编译结束退出命令的插件

```
apply (compiler) {  
  const afterEmit = (compilation, cb) => {  
    cb()  
    setTimeout(function () {  
      process.exit(0)  
    }, 1000)  
  }  
  
  compiler.plugin('after-emit', afterEmit)  
}  
}  
  
module.exports = BuildEndPlugin
```

进阶篇

#一、JavaScript进阶

#1 内置类型

- JS 中分为七种内置类型，七种内置类型又分为两大类型：基本类型和对象（object）。
- 基本类型有六种：`null`, `undefined`, `boolean`, `number`, `string`, `symbol`。
- 其中 JS 的数字类型是浮点类型的，没有整型。并且浮点类型基于 IEEE 754 标准实现，在使用中会遇到某些 Bug。`Nan` 也属于 `number` 类型，并且 `Nan` 不等于自身。
- 对于基本类型来说，如果使用字面量的方式，那么这个变量只是个字面量，只有在必要的时候才会转换为对应的类型。

```
let a = 111 // 这只是字面量，不是 number 类型  
a.toString() // 使用时候才会转换为对象类型
```

对象（object）是引用类型，在使用过程中会遇到浅拷贝和深拷贝的问题。

```
let a = { name: 'FE' }  
let b = a  
b.name = 'EF'  
console.log(a.name) // EF
```

#2 Typeof

`typeof` 对于基本类型，除了 `null` 都可以显示正确的类型

```
typeof 1 // 'number'  
typeof '1' // 'string'  
typeof undefined // 'undefined'  
typeof true // 'boolean'  
typeof Symbol() // 'symbol'  
typeof b // b 没有声明，但是还会显示 undefined
```

`typeof`` 对于对象，除了函数都会显示 `object`

```
typeof [] // 'object'  
typeof {} // 'object'  
typeof console.log // 'function'
```

对于 `null` 来说，虽然它是基本类型，但是会显示 `object`，这是一个存在很久了的 Bug

```
typeof null // 'object'
```

PS：为什么会出现这种情况呢？因为在 JS 的最初版本中，使用的是 32 位系统，为了性能考虑使用低位存储了变量的类型信息，000 开头代表是对象，然而 `null` 表示为全零，所以将它错误的判断为 `object`。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

- 如果我们想获得一个变量的正确类型，可以通过 `Object.prototype.toString.call(xx)`。这样我们就可以获得类似 `[object Type]` 的字符串

```
let a
// 我们也可以这样判断 undefined
a === undefined
// 但是 undefined 不是保留字，能够在低版本浏览器被赋值
let undefined = 1
// 这样判断就会出错
// 所以可以用下面的方式来判断，并且代码量更少
// 因为 void 后面随便跟上一个组成表达式
// 返回就是 undefined
a === void 0
```

#3 类型转换

转Boolean

在条件判断时，除了 `undefined`, `null`, `false`, `Nan`, `''`, `0`, `-0`，其他所有值都转为 `true`，包括所有对象

对象转基本类型

对象在转换基本类型时，首先会调用 `valueOf` 然后调用 `toString`。并且这两个方法你是可以重写的

```
let a = {
  valueOf() {
    return 0
  }
}
```

四则运算符

只有当加法运算时，其中一方是字符串类型，就会把另一个也转为字符串类型。其他运算只要其中一方是数字，那么另一方就转为数字。并且加法运算会触发三种类型转换：将值转换为原始值，转换为数字，转换为字符串

```
1 + '1' // '11'
2 * '2' // 4
[1, 2] + [2, 1] // '1,22,1'
// [1, 2].toString() -> '1,2'
// [2, 1].toString() -> '2,1'
// '1,2' + '2,1' = '1,22,1'
```

对于加号需要注意这个表达式 `'a' + + 'b'`

```
'a' + + 'b' // -> "aNan"
// 因为 + 'b' -> NaN
// 你也许在一些代码中看到过 + '1' -> 1
```

== 操作符

比较运算`x==y`, 其中`x`和`y`是值, 产生`true`或者`false`。这样的比较按如下方式进行:

1. 若`Type(x)`与`Type(y)`相同, 则
 - a. 若`Type(x)`为`Undefined`, 返回`true`。
 - b. 若`Type(x)`为`Null`, 返回`true`。
 - c. 若`Type(x)`为`Number`, 则
 - i. 若`x`为`Nan`, 返回`false`。
 - ii. 若`y`为`Nan`, 返回`false`。
 - iii. 若`x`与`y`为相等数值, 返回`true`。
 - iv. 若`x`为`+0`且`y`为`-0`, 返回`true`。
 - v. 若`x`为`-0`且`y`为`+0`, 返回`true`。
 - vi. 返回`false`。
 - d. 若`Type(x)`为`String`, 则当`x`和`y`为完全相同的字符序列 (长度相等且相同字符在相同位置) 时返回`true`。否则, 返回`false`。
 - e. 若`Type(x)`为`Boolean`, 当`x`和`y`为同为`true`或者同为`false`时返回`true`。否则, 返回`false`。
 - f. 当`x`和`y`为引用同一对象时返回`true`。否则, 返回`false`。
2. 若`x`为`null`且`y`为`undefined`, 返回`true`。
3. 若`x`为`undefined`且`y`为`null`, 返回`true`。
4. 若`Type(x)`为`Number`且`Type(y)`为`String`, 返回`comparison x == ToNumber(y)`的结果。
5. 若`Type(x)`为`String`且`Type(y)`为`Number`,
6. 返回比较`ToNumber(x) == y`的结果。
7. 若`Type(x)`为`Boolean`, 返回比较`ToNumber(x) == y`的结果。
8. 若`Type(y)`为`Boolean`, 返回比较`x == ToNumber(y)`的结果。
9. 若`Type(x)`为`String`或`Number`, 且`Type(y)`为`Object`, 返回比较`x == ToPrimitive(y)`的结果。
10. 若`Type(x)`为`Object`且`Type(y)`为`String`或`Number`, 返回比较`ToPrimitive(x) == y`的结果。
11. 返回`false`。

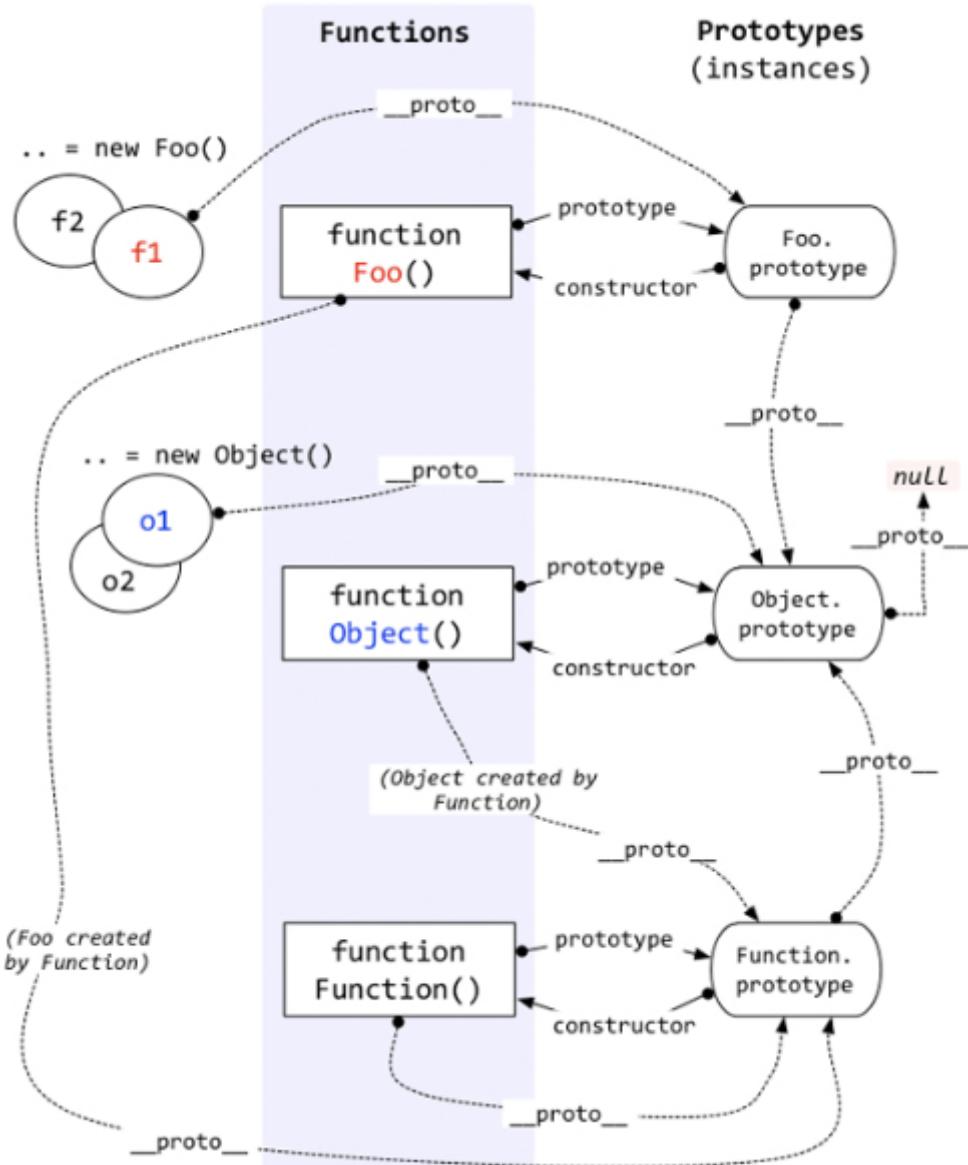
这里来解析一道题目`[] == ![] // -> true`, 下面是这个表达式为何为`true`的步骤

```
// [] 转成 true, 然后取反变成 false
[] == false
// 根据第 8 条得出
[] == ToNumber(false)
[] == 0
// 根据第 10 条得出
ToPrimitive([]) == 0
// [].toString() -> ''
'' == 0
// 根据第 6 条得出
0 == 0 // -> true
```

比较运算符

- 如果是对象, 就通过`toPrimitive`转换对象
- 如果是字符串, 就通过`unicode`字符索引来比较

#4 原型



- 每个函数都有 `prototype` 属性，除了 `Function.prototype.bind()`，该属性指向原型。
- 每个对象都有 `__proto__` 属性，指向了创建该对象的构造函数的原型。其实这个属性指向了 `[[prototype]]`，但是 `[[prototype]]` 是内部属性，我们并不能访问到，所以使用 `__proto__` 来访问。
- 对象可以通过 `__proto__` 来寻找不属于该对象的属性，`__proto__` 将对象连接起来组成了原型链

#5 new

- 新生成了一个对象
- 链接到原型
- 绑定 `this`
- 返回新对象

在调用 `new` 的过程中会发生以上四件事情，我们也可以试着来自己实现一个 `new`

```
function create() {
    // 创建一个空的对象
    let obj = new Object()
    // 获得构造函数
    let Con = [].shift.call(arguments)
    // 链接到原型
    obj.__proto__ = Con.prototype
    // 绑定 this, 执行构造函数
    let result = Con.apply(obj, arguments)
    // 确保 new 出来的是个对象
    return typeof result === 'object' ? result : obj
}
```

#6 instanceof

`instanceof` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类的 `prototype`

我们也可以试着实现一下 `instanceof`

```
function instanceof(left, right) {
    // 获得类型的原型
    let prototype = right.prototype
    // 获得对象的原型
    left = left.__proto__
    // 判断对象的类型是否等于类型的原型
    while (true) {
        if (left === null)
            return false
        if (prototype === left)
            return true
        left = left.__proto__
    }
}
```

#7 this

```
function foo() {
    console.log(this.a)
}
var a = 1
foo()

var obj = {
    a: 2,
    foo: foo
}
obj.foo()

// 以上两者情况 `this` 只依赖于调用函数前的对象，优先级是第二个情况大于第一个情况

// 以下情况是优先级最高的，`this` 只会绑定在 `c` 上，不会被任何方式修改 `this` 指向
var c = new foo()
c.a = 3
console.log(c.a)
```

```
// 还有种就是利用 call, apply, bind 改变 this, 这个优先级仅次于 new
```

看看箭头函数中的 `this`

```
function a() {
  return () => {
    return () => {
      console.log(this)
    }
  }
}
console.log(a()())
```

箭头函数其实是没有 `this` 的，这个函数中的 `this` 只取决于他外面的第一个不是箭头函数的函数的 `this`。在这个例子中，因为调用 `a` 符合前面代码中的第一个情况，所以 `this` 是 `window`。并且 `this` 一旦绑定了上下文，就不会被任何代码改变

#8 执行上下文

当执行 JS 代码时，会产生三种执行上下文

- 全局执行上下文
- 函数执行上下文
- `eval` 执行上下文

每个执行上下文中都有三个重要的属性

- 变量对象（`vo`），包含变量、函数声明和函数的形参，该属性只能在全局上下文中访问
- 作用域链（JS 采用词法作用域，也就是说变量的作用域是在定义时就决定了）
- `this`

```
var a = 10
function foo(i) {
  var b = 20
}
foo()
```

对于上述代码，执行栈中有两个上下文：全局上下文和函数 `foo` 上下文。

```
stack = [
  globalContext,
  fooContext
]
```

对于全局上下文来说，`vo` 大概是这样的

```
globalContext.vo === globe
globalContext.vo = {
  a: undefined,
  foo: <Function>,
}
```

对于函数 `foo` 来说，`vo` 不能访问，只能访问到活动对象（`ao`）

```
fooContext.VO === foo.AO
fooContext.AO {
    i: undefined,
    b: undefined,
    arguments: <>
}
// arguments 是函数独有的对象(箭头函数没有)
// 该对象是一个伪数组，有 `length` 属性且可以通过下标访问元素
// 该对象中的 `callee` 属性代表函数本身
// `caller` 属性代表函数的调用者
```

对于作用域链，可以把它理解成包含自身变量对象和上级变量对象的列表，通过 `[[Scope]]` 属性查找上级变量

```
fooContext.[[Scope]] = [
    globalContext.VO
]
fooContext.Scope = fooContext.[[Scope]] + fooContext.VO
fooContext.Scope = [
    fooContext.VO,
    globalContext.VO
]
```

接下来让我们看一个老生常谈的例子，`var`

```
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
    console.log('call b')
}
```

想必以上的输出大家肯定都已经明白了，这是因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行上下文时，会有两个阶段。第一个阶段是创建的阶段（具体步骤是创建 `VO`），`JS` 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 `undefined`，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用。

- 在提升的过程中，相同的函数会覆盖上一个函数，并且函数优先于变量提升

```
b() // call b second

function b() {
    console.log('call b fist')
}
function b() {
    console.log('call b second')
}
var b = 'Hello world'
```

`var` 会产生很多错误，所以在 ES6 中引入了 `let`。`let` 不能在声明前使用，但是这并不是常说的 `let` 不会提升，`let` 提升了声明但没有赋值，因为临时死区导致了并不能在声明前使用。

- 对于非匿名的立即执行函数需要注意以下一点

```
var foo = 1
(function foo() {
  foo = 10
  console.log(foo)
}()) // -> f foo() { foo = 10 ; console.log(foo) }
```

因为当 JS 解释器在遇到非匿名的立即执行函数时，会创建一个辅助的特定对象，然后将函数名称作为这个对象的属性，因此函数内部才可以访问到 `foo`，但是这个值又是只读的，所以对它的赋值并不生效，所以打印的结果还是这个函数，并且外部的值也没有发生更改。

```
specialObject = {};
Scope = specialObject + Scope;
foo = new FunctionExpression;
foo.:[[Scope]] = Scope;
specialObject.foo = foo; // {DontDelete}, {ReadOnly}
delete Scope[0]; // remove specialObject from the front of scope chain
```

#9 闭包

闭包的定义很简单：函数 A 返回了一个函数 B，并且函数 B 中使用了函数 A 的变量，函数 B 就被称为闭包。

```
function A() {
  let a = 1
  function B() {
    console.log(a)
  }
  return B
}
```

你是否会疑惑，为什么函数 A 已经弹出调用栈了，为什么函数 B 还能引用到函数 A 中的变量。

因为函数 A 中的变量这时候是存储在堆上的。现在的 JS 引擎可以通过逃逸分析辨别出哪些变量需要存储在堆上，哪些需要存储在栈上。

经典面试题，循环中使用闭包解决 var 定义函数的问题

```
for ( var i=1; i<=5; i++ ) {
  setTimeout( function timer() {
    console.log( i );
  }, i*1000 );
}
```

- 首先因为 `setTimeout` 是个异步函数，所有会先把循环全部执行完毕，这时候 `i` 就是 6 了，所以会输出一堆 6。
- 解决办法两种，第一种使用闭包

```
for (var i = 1; i <= 5; i++) {  
  (function(j) {  
    setTimeout(function timer() {  
      console.log(j);  
    }, j * 1000);  
  })(i);  
}
```

- 第二种就是使用 `setTimeout` 的第三个参数

```
for ( var i=1; i<=5; i++) {  
  setTimeout( function timer(j) {  
    console.log( j );  
  }, i*1000, i);  
}
```

第三种就是使用 `let` 定义 `i` 了

```
for ( let i=1; i<=5; i++) {  
  setTimeout( function timer() {  
    console.log( i );  
  }, i*1000 );  
}
```

因为对于 `let` 来说，他会创建一个块级作用域，相当于

```
{ // 形成块级作用域  
let i = 0  
{  
  let ii = i  
  setTimeout( function timer() {  
    console.log( ii );  
  }, ii*1000 );  
}  
i++  
{  
  let ii = i  
}  
i++  
{  
  let ii = i  
}  
...  
}
```

#10 深浅拷贝

```
let a = {  
  age : 1  
}  
let b = a  
a.age = 2  
console.log(b.age) // 2
```

- 从上述例子中我们可以发现，如果给一个变量赋值一个对象，那么两者的值会是同一个引用，其中一方改变，另一方也会相应改变。
- 通常在开发中我们不希望出现这样的问题，我们可以使用浅拷贝来解决这个问题

浅拷贝

首先可以通过 `Object.assign` 来解决这个问题

```
let a = {
  age: 1
}
let b = Object.assign({}, a)
a.age = 2
console.log(b.age) // 1
```

当然我们也可以通过展开运算符 `(...)` 来解决

```
let a = {
  age: 1
}
let b = {...a}
a.age = 2
console.log(b.age) // 1
```

通常浅拷贝就能解决大部分问题了，但是当我们遇到如下情况就需要使用到深拷贝了

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = {...a}
a.jobs.first = 'native'
console.log(b.jobs.first) // native
```

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，那么就又回到刚开始的话题了，两者享有相同的引用。要解决这个问题，我们需要引入深拷贝

深拷贝

这个问题通常可以通过 `JSON.parse(JSON.stringify(object))` 来解决

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

但是该方法也是有局限性的：

- 会忽略 `undefined`

- 不能序列化函数
- 不能解决循环引用的对象

```
let obj = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  },
}
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)
```

| 如果你有这么一个循环引用对象，你会发现你不能通过该方法深拷贝

- 在遇到函数或者 `undefined` 的时候，该对象也不能正常的序列化

```
let a = {
  age: undefined,
  jobs: function() {},
  name: 'poetries'
}
let b = JSON.parse(JSON.stringify(a))
console.log(b) // {name: "poetries"}
```

- 你会发现在上述情况中，该方法会忽略掉函数和``undefined``。
- 但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题，并且该函数是内置函数中处理深拷贝性能最快的。当然如果你的数据中含有以上三种情况下，可以使用 `lodash` 的深拷贝函数。

#11 模块化

| 在有 `Babel` 的情况下，我们可以直接使用 `ES6` 的模块化

```
// file a.js
export function a() {}
export function b() {}
// file b.js
export default function() {}

import {a, b} from './a.js'
import xxx from './b.js'
```

CommonJS

| `CommonJS` 是 `Node` 独有的规范，浏览器中使用就需要用到 `Browserify` 解析了。

```
// a.js
module.exports = {
  a: 1
}
// or
exports.a = 1

// b.js
var module = require('./a.js')
module.a // -> log 1
```

在上述代码中，`module.exports` 和 `exports` 很容易混淆，让我们来看看大致内部实现

```
var module = require('./a.js')
module.a
// 这里其实就是包装了一层立即执行函数，这样就不会污染全局变量了，
// 重要的是 module 这里，module 是 Node 独有的一个变量
module.exports = {
  a: 1
}
// 基本实现
var module = {
  exports: {} // exports 就是个空对象
}
// 这个是因为 exports 和 module.exports 用法相似的原因
var exports = module.exports
var load = function (module) {
  // 导出的东西
  var a = 1
  module.exports = a
  return module.exports
};
```

再来说说 `module.exports` 和 `exports`，用法其实是相似的，但是不能对 `exports` 直接赋值，不会有任何效果。

对于 `CommonJS` 和 `ES6` 中的模块化的两者区别是：

- 前者支持动态导入，也就是 `require(${path}/xx.js)`，后者目前不支持，但是已有提案，前者是同步导入，因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。
- 而后者是异步导入，因为用于浏览器，需要下载文件，如果也采用同步导入会对渲染有很大影响
- 前者在导出时都是值拷贝，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。
- 但是后者采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
- 后者会编译成 `require,exports` 来执行的

AMD

`AMD` 是由 `RequireJS` 提出的

```
// AMD
define(['./a', './b'], function(a, b) {
    a.do()
    b.do()
})
define(function(require, exports, module) {
    var a = require('./a')
    a.doSomething()
    var b = require('./b')
    b.doSomething()
})
```

#12 防抖

你是否在日常开发中遇到一个问题，在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。

- 这些需求都可以通过函数防抖动来实现。尤其是第一个需求，如果在频繁的事件回调中做复杂计算，很有可能导致页面卡顿，不如将多次计算合并为一次计算，只在一个精确点做操作
- PS：防抖和节流的作用都是防止函数多次调用。区别在于，假设一个用户一直触发这个函数，且每次触发函数的间隔小于 `wait`，防抖的情况下只会调用一次，而节流的情况会每隔一定时间（参数 `wait`）调用函数

```
// 这个是用来获取当前时间戳的
function now() {
    return +new Date()
}

/**
 * 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
 *
 * @param {function} func      回调函数
 * @param {number}   wait       表示时间窗口的间隔
 * @param {boolean} immediate  设置为ture时，是否立即调用函数
 * @return {function}          返回客户调用函数
 */
function debounce (func, wait = 50, immediate = true) {
    let timer, context, args

    // 延迟执行函数
    const later = () => setTimeout(() => {
        // 延迟函数执行完毕，清空缓存的定时器序号
        timer = null
        // 延迟执行的情况下，函数会在延迟函数中执行
        // 使用到之前缓存的参数和上下文
        if (!immediate) {
            func.apply(context, args)
            context = args = null
        }
    }, wait)

    // 这里返回的函数是每次实际调用的函数
    return function(...params) {
        // 如果没有创建延迟执行函数（later），就创建一个
        if (!timer) {
            timer = later()
            // 如果是立即执行，调用函数
        }
    }
}
```

```

    // 否则缓存参数和调用上下文
    if (immediate) {
      func.apply(this, params)
    } else {
      context = this
      args = params
    }
    // 如果已有延迟执行函数 (later)，调用的时候清除原来的并重新设定一个
    // 这样做延迟函数会重新计时
    } else {
      clearTimeout(timer)
      timer = later()
    }
  }
}

```

- 对于按钮防点击来说的实现：如果函数是立即执行的，就立即调用，如果函数是延迟执行的，就缓存上下文和参数，放到延迟函数中去执行。一旦我开始一个定时器，只要我定时器还在，你每次点击我都重新计时。一旦你点累了，定时器时间到，定时器重置为 `null`，就可以再次点击了。
- 对于延时执行函数来说的实现：清除定时器 `ID`，如果是延迟调用就调用函数

#13 节流

防抖动和节流本质是不一样的。防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时间执行

```

/**
 * underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 / wait
 *
 * @param {function} func      回调函数
 * @param {number}   wait       表示时间窗口的间隔
 * @param {object}  options    如果想忽略开始函数的调用，传入{leading: false}。
 *                            如果想忽略结尾函数的调用，传入{trailing: false}
 *                            两者不能共存，否则函数不能执行
 *
 * @return {function}          返回客户调用函数
 */
_.throttle = function(func, wait, options) {
  var context, args, result;
  var timeout = null;
  // 之前的时间戳
  var previous = 0;
  // 如果 options 没传则设为空对象
  if (!options) options = {};
  // 定时器回调函数
  var later = function() {
    // 如果设置了 leading，就将 previous 设为 0
    // 用于下面函数的第一个 if 判断
    previous = options.leading === false ? 0 : _.now();
    // 置空一是为了防止内存泄漏，二是为了下面的定时器判断
    timeout = null;
    result = func.apply(context, args);
    if (!timeout) context = args = null;
  };
  return function() {
    // 获得当前时间戳
    var now = _.now();
    
```

```

// 首次进入前者肯定为 true
// 如果需要第一次不执行函数
// 就将上次时间戳设为当前的
// 这样在接下来计算 remaining 的值时会大于0
if (!previous && options.leading === false) previous = now;
// 计算剩余时间
var remaining = wait - (now - previous);
context = this;
args = arguments;
// 如果当前调用已经大于上次调用时间 + wait
// 或者用户手动调了时间
// 如果设置了 trailing, 只会进入这个条件
// 如果没有设置 leading, 那么第一次会进入这个条件
// 还有一点, 你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
// 其实还是会进入的, 因为定时器的延时
// 并不是准确的时间, 很可能你设置了2秒
// 但是他需要2.2秒才触发, 这时候就会进入这个条件
if (remaining <= 0 || remaining > wait) {
    // 如果存在定时器就清理掉否则会调用二次回调
    if (timeout) {
        clearTimeout(timeout);
        timeout = null;
    }
    previous = now;
    result = func.apply(context, args);
    if (!timeout) context = args = null;
} else if (!timeout && options.trailing !== false) {
    // 判断是否设置了定时器和 trailing
    // 没有的话就开启一个定时器
    // 并且不能同时设置 leading 和 trailing
    timeout = setTimeout(later, remaining);
}
return result;
};
};


```

#14 继承

在 ES5 中, 我们可以使用如下方式解决继承的问题

```

function Super() {}
Super.prototype.getNumber = function() {
    return 1
}

function Sub() {}
let s = new Sub()
Sub.prototype = Object.create(Super.prototype, {
    constructor: {
        value: Sub,
        enumerable: false,
        writable: true,
        configurable: true
    }
})

```

- 以上继承实现思路就是将子类的原型设置为父类的原型
- 在 ES6 中，我们可以通过 class 语法轻松解决这个问题

```
class MyDate extends Date {
  test() {
    return this.getTime()
  }
}
let myDate = new MyDate()
myDate.test()
```

- 但是 ES6 不是所有浏览器都兼容，所以我们需要使用 Babel 来编译这段代码。
- 如果你使用编译过得代码调用 myDate.test() 你会惊奇地发现出现了报错

因为在 JS 底层有限制，如果不是由 Date 构造出来的实例的话，是不能调用 Date 里的函数的。所以这也侧面的说明了：ES6 中的 class 继承与 ES5 中的一般继承写法是不同的。

- 既然底层限制了实例必须由 Date 构造出来，那么我们可以改变下思路实现继承

```
function MyData() {

}
MyData.prototype.test = function () {
  return this.getTime()
}
let d = new Date()
Object.setPrototypeOf(d, MyData.prototype)
Object.setPrototypeOf(MyData.prototype, Date.prototype)
```

- 以上继承实现思路：先创建父类实例 => 改变实例原先的 __proto__ 转而连接到子类的 prototype => 子类的 prototype 的 __proto__ 改为父类的 prototype。
- 通过以上方法实现的继承就可以完美解决 JS 底层的这个限制

#15 call, apply, bind

- call 和 apply 都是为了解决改变 this 的指向。作用都是相同的，只是传参的方式不同。
- 除了第一个参数外，call 可以接收一个参数列表，apply 只接受一个参数数组

```
let a = {
  value: 1
}
function getValue(name, age) {
  console.log(name)
  console.log(age)
  console.log(this.value)
}
getValue.call(a, 'yck', '24')
getValue.apply(a, ['yck', '24'])
```

#16 Promise 实现

- 可以把 `Promise` 看成一个状态机。初始是 `pending` 状态，可以通过函数 `resolve` 和 `reject`，将状态转变为 `resolved` 或者 `rejected` 状态，状态一旦改变就不能再次变化。
- `then` 函数会返回一个 `Promise` 实例，并且该返回值是一个新的实例而不是之前的实例。因为 `Promise` 规范规定除了 `pending` 状态，其他状态是不可以改变的，如果返回的是一个相同实例的话，多个 `then` 调用就失去意义了。
- 对于 `then` 来说，本质上可以把它看成是 `flatMap`

```
// 三种状态
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";
// promise 接收一个函数参数，该函数会立即执行
function MyPromise(fn) {
  let _this = this;
  _this.currentState = PENDING;
  _this.value = undefined;
  // 用于保存 then 中的回调，只有当 promise
  // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
  _this.resolvedCallbacks = [];
  _this.rejectedCallbacks = [];

  _this.resolve = function (value) {
    if (value instanceof MyPromise) {
      // 如果 value 是个 Promise，递归执行
      return value.then(_this.resolve, _this.reject)
    }
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = RESOLVED;
        _this.value = value;
        _this.resolvedCallbacks.forEach(cb => cb());
      }
    })
  }

  _this.reject = function (reason) {
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = REJECTED;
        _this.value = reason;
        _this.rejectedCallbacks.forEach(cb => cb());
      }
    })
  }
}

// 用于解决以下问题
// new Promise(() => throw Error('error'))
try {
  fn(_this.resolve, _this.reject);
} catch (e) {
  _this.reject(e);
}
}

MyPromise.prototype.then = function (onResolved, onRejected) {
```

```
var self = this;
// 规范 2.2.7, then 必须返回一个新的 promise
var promise2;
// 规范 2.2.onResolved 和 onRejected 都为可选参数
// 如果类型不是函数需要忽略, 同时也实现了透传
// Promise.resolve(4).then().then((value) => console.log(value))
onResolved = typeof onResolved === 'function' ? onResolved : v => v;
onRejected = typeof onRejected === 'function' ? onRejected : r => throw r;

if (self.currentState === RESOLVED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
        // 规范 2.2.4, 保证 onFulfilled, onRejected 异步执行
        // 所以用了 setTimeout 包裹下
        setTimeout(function () {
            try {
                var x = onResolved(self.value);
                resolutionProcedure(promise2, x, resolve, reject);
            } catch (reason) {
                reject(reason);
            }
        });
    }));
}

if (self.currentState === REJECTED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
        setTimeout(function () {
            // 异步执行onRejected
            try {
                var x = onRejected(self.value);
                resolutionProcedure(promise2, x, resolve, reject);
            } catch (reason) {
                reject(reason);
            }
        });
    }));
}

if (self.currentState === PENDING) {
    return (promise2 = new MyPromise(function (resolve, reject) {
        self.resolvedCallbacks.push(function () {
            // 考虑到可能会有报错, 所以使用 try/catch 包裹
            try {
                var x = onResolved(self.value);
                resolutionProcedure(promise2, x, resolve, reject);
            } catch (r) {
                reject(r);
            }
        });
    }));
}

self.rejectedCallbacks.push(function () {
    try {
        var x = onRejected(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
    } catch (r) {
        reject(r);
    }
});
```

```

        });
    }
};

// 规范 2.3
function resolutionProcedure(promise2, x, resolve, reject) {
    // 规范 2.3.1, x 不能和 promise2 相同, 避免循环引用
    if (promise2 === x) {
        return reject(new TypeError("Error"));
    }
    // 规范 2.3.2
    // 如果 x 为 Promise, 状态为 pending 需要继续等待否则执行
    if (x instanceof MyPromise) {
        if (x.currentState === PENDING) {
            x.then(function (value) {
                // 再次调用该函数是为了确认 x resolve 的
                // 参数是什么类型, 如果是基本类型就再次 resolve
                // 把值传给下个 then
                resolutionProcedure(promise2, value, resolve, reject);
            }, reject);
        } else {
            x.then(resolve, reject);
        }
        return;
    }
    // 规范 2.3.3.3.3
    // reject 或者 resolve 其中一个执行过得话, 忽略其他的
    let called = false;
    // 规范 2.3.3, 判断 x 是否为对象或者函数
    if (x !== null && (typeof x === "object" || typeof x === "function")) {
        // 规范 2.3.3.2, 如果不能取出 then, 就 reject
        try {
            // 规范 2.3.3.1
            let then = x.then;
            // 如果 then 是函数, 调用 x.then
            if (typeof then === "function") {
                // 规范 2.3.3.3
                then.call(
                    x,
                    y => {
                        if (called) return;
                        called = true;
                        // 规范 2.3.3.3.1
                        resolutionProcedure(promise2, y, resolve, reject);
                    },
                    e => {
                        if (called) return;
                        called = true;
                        reject(e);
                    }
                );
            } else {
                // 规范 2.3.3.4
                resolve(x);
            }
        } catch (e) {
            if (called) return;
            called = true;
            reject(e);
        }
    }
}

```

```
        }
    } else {
        // 规范 2.3.4, x 为基本类型
        resolve(x);
    }
}
```

#17 Generator 实现

Generator 是 ES6 中新增的语法，和 Promise 一样，都可以用来异步编程

```
// 使用 * 表示这是一个 Generator 函数
// 内部可以通过 yield 暂停代码
// 通过调用 next 恢复执行
function* test() {
    let a = 1 + 2;
    yield 2;
    yield 3;
}
let b = test();
console.log(b.next()); // > { value: 2, done: false }
console.log(b.next()); // > { value: 3, done: false }
console.log(b.next()); // > { value: undefined, done: true }
```

从以上代码可以发现，加上 * 的函数执行后拥有了 next 函数，也就是说函数执行后返回了一个对象。每次调用 next 函数可以继续执行被暂停的代码。以下是 Generator 函数的简单实现

```
// cb 也就是编译过的 test 函数
function generator(cb) {
    return (function() {
        var object = {
            next: 0,
            stop: function() {}
        };

        return {
            next: function() {
                var ret = cb(object);
                if (ret === undefined) return { value: undefined, done: true };
                return {
                    value: ret,
                    done: false
                };
            }
        };
    })();
}

// 如果你使用 babel 编译后可以发现 test 函数变成了这样
function test() {
    var a;
    return generator(function(_context) {
        while (1) {
            switch (((_context.prev = _context.next)) {
                // 可以发现通过 yield 将代码分割成几块
                // 每次执行 next 函数就执行一块代码
                // 并且表明下次需要执行哪块代码
            })
        }
    });
}
```

```

        case 0:
            a = 1 + 2;
            _context.next = 4;
            return 2;
        case 4:
            _context.next = 6;
            return 3;
            // 执行完毕
        case 6:
        case "end":
            return _context.stop();
    }
}
});
}

```

#18 Proxy

`Proxy` 是 `ES6` 中新增的功能，可以用来自定义对象中的操作

```

let p = new Proxy(target, handler);
// `target` 代表需要添加代理的对象
// `handler` 用来自定义对象中的操作
可以很方便的使用 Proxy 来实现一个数据绑定和监听

let onwatch = (obj, setBind, getLogger) => {
    let handler = {
        get(target, property, receiver) {
            getLogger(target, property)
            return Reflect.get(target, property, receiver);
        },
        set(target, property, value, receiver) {
            setBind(value);
            return Reflect.set(target, property, value);
        }
    };
    return new Proxy(obj, handler);
};

let obj = { a: 1 }
let value
let p = onwatch(obj, (v) => {
    value = v
}, (target, property) => {
    console.log(`Get '${property}' = ${target[property]}`);
})
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2

```

#二、浏览器

#1 事件机制

事件触发三阶段

- `document` 往事件触发处传播，遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 `document` 传播，遇到注册的冒泡事件会触发

事件触发一般来说会按照上面的顺序进行，但是也有特例，如果给一个目标节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行

```
// 以下会先打印冒泡然后是捕获
node.addEventListener('click',(event) =>{
    console.log('冒泡')
},false);
node.addEventListener('click',(event) =>{
    console.log('捕获 ')
},true)
```

注册事件

- 通常我们使用 `addEventListener` 注册事件，该函数的第三个参数可以是布尔值，也可以是对象。对于布尔值 `useCapture` 参数来说，该参数默认值为 `false`。`useCapture` 决定了注册的事件是捕获事件还是冒泡事件
- 一般来说，我们只希望事件只触发在目标上，这时候可以使用 `stopPropagation` 来阻止事件的进一步传播。通常我们认为 `stopPropagation` 是用来阻止事件冒泡的，其实该函数也可以阻止捕获事件。`stopImmediatePropagation` 同样也能实现阻止事件，但是还能阻止该事件目标执行别的注册事件

```
node.addEventListener('click',(event) =>{
    event.stopImmediatePropagation()
    console.log('冒泡')
},false);
// 点击 node 只会执行上面的函数，该函数不会执行
node.addEventListener('click',(event) => {
    console.log('捕获 ')
},true)
```

事件代理

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```
<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector('#ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>
```

事件代理的方式相对于直接给目标注册事件来说，有以下优点

- 节省内存
- 不需要给子节点注销事件

#2 跨域

因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，`Ajax` 请求会失败

JSONP

`JSONP` 的原理很简单，就是利用 `<script>` 标签没有跨域限制的漏洞。通过 `<script>` 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时

```
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

- `JSONP` 使用简单且兼容性不错，但是只限于 `get` 请求

CORS

- CORS 需要浏览器和后端同时支持
- 浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。
- 服务端设置 `Access-Control-Allow-Origin` 就可以开启 CORS。该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源

document.domain

- 该方式只能用于二级域名相同的情况下，比如 `a.test.com` 和 `b.test.com` 适用于该方式。
- 只需要给页面添加 `document.domain = 'test.com'` 表示二级域名都相同就可以实现跨域

postMessage

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

```
// 发送消息端
window.parent.postMessage('message', 'http://blog.poeties.com');

// 接收消息端
var mc = new MessageChannel();
mc.addEventListener('message', (event) => {
    var origin = event.origin || event.originalEvent.origin;
    if (origin === 'http://blog.poeties.com') {
        console.log('验证通过')
    }
});
```

#3 Event loop

JS中的event loop

众所周知 JS 是一门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是一门多线程的语言话，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点）

- JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task（有多种 task）队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为

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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask）和宏任务（macrotask）。在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task

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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
  console.log('Promise')
  resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout
```

以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务

微任务

- process.nextTick
- promise
- Object.observe
- MutationObserver

宏任务

- script
- setTimeout

- `setInterval`
- `setImmediate`
- `I/O`
- `UI rendering`

宏任务中包括了 `script`，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

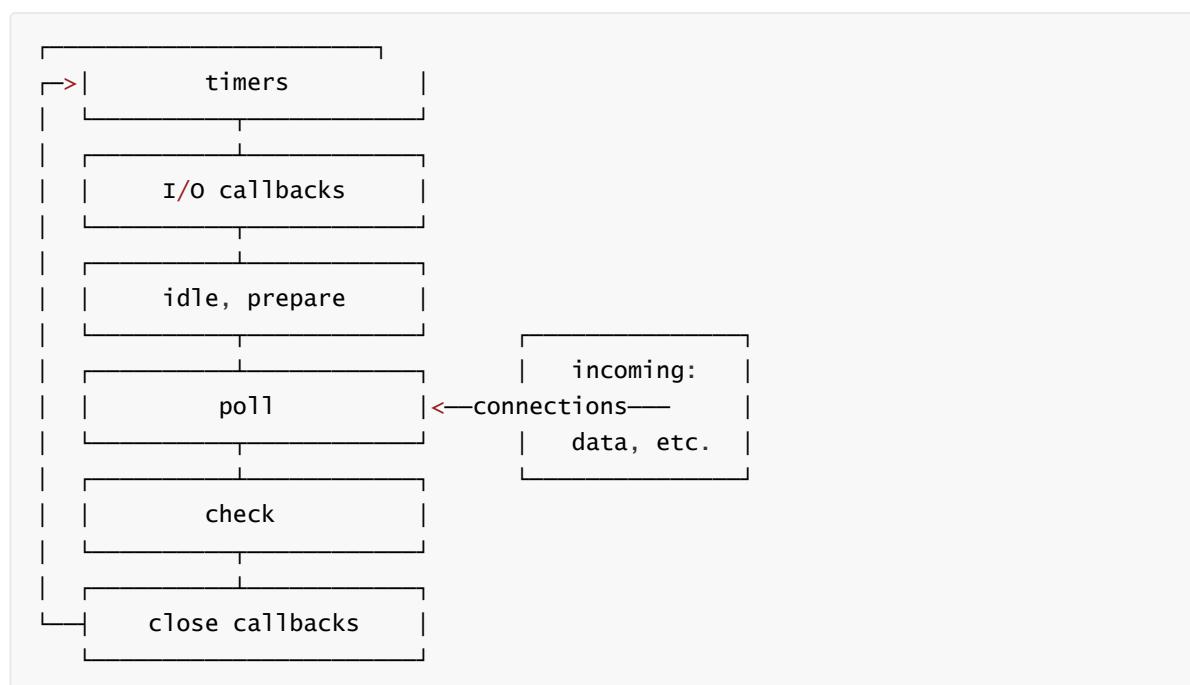
所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 `Event loop`，执行宏任务中的异步代码

通过上述的 `Event loop` 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 `DOM` 的话，为了更快的响应界面响应，我们可以把操作 `DOM` 放入微任务中

Node 中的 Event loop

- `Node` 中的 `Event loop` 和浏览器中的不相同。
- `Node` 的 `Event loop` 分为 6 个阶段，它们会按照顺序反复运行



timer

- `timers` 阶段会执行 `setTimeout` 和 `setInterval`
- 一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟

I/O

- `I/O` 阶段会执行除了 `close` 事件，定时器和 `setImmediate` 的回调

poll

- `poll` 阶段很重要，这一阶段中，系统会做两件事情
 - 执行到点的定时器
 - 执行 `poll` 队列中的事件
- 并且当 `poll` 中没有定时器的情况下，会发现以下两件事情

- 如果 `poll` 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
- 如果 `poll` 队列为空，会有两件事发生
- 如果有 `setImmediate` 需要执行，`poll` 阶段会停止并且进入到 `check` 阶段执行 `setImmediate`
- 如果没有 `setImmediate` 需要执行，会等待回调被加入到队列中并立即执行回调
- 如果有别的定时器需要被执行，会回到 `timer` 阶段执行回调。

check

- `check` 阶段执行 `setImmediate`

close callbacks

- `close callbacks` 阶段执行 `close` 事件
- 并且在 `Node` 中，有些情况下的定时器执行顺序是随机的

```
setTimeout(() => {
  console.log('setTimeout');
}, 0);
setImmediate(() => {
  console.log('setImmediate');
})
// 这里可能会输出 setTimeout, setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

上面介绍的都是 `macrotask` 的执行情况，`microtask` 会在以上每个阶段完成后立即执行

```
setTimeout(()=>{
  console.log('timer1')

  Promise.resolve().then(function() {
    console.log('promise1')
  })
}, 0)

setTimeout(()=>{
  console.log('timer2')

  Promise.resolve().then(function() {
    console.log('promise2')
  })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2
```

`Node` 中的 `process.nextTick` 会先于其他 `microtask` 执行

```

setTimeout(() => {
  console.log("timer1");

  Promise.resolve().then(function() {
    console.log("promise1");
  });
}, 0);

process.nextTick(() => {
  console.log("nextTick");
});
// nextTick, timer1, promise1

```

#4 Service Worker

[Service workers] 本质上充当Web应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。它们旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作。他们还允许访问推送通知和后台同步API

目前该技术通常用来做缓存文件，提高首屏速度

```

// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register("sw.js")
    .then(function(registration) {
      console.log("service worker 注册成功");
    })
    .catch(function(err) {
      console.log("service worker 注册失败");
    });
}

// sw.js
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener("install", e => {
  e.waitUntil(
    caches.open("my-cache").then(function(cache) {
      return cache.addAll(["./index.html", "./index.js"]);
    })
  );
});

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener("fetch", e => {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      if (response) {
        return response;
      }
      console.log("fetch source");
    })
  );
});

```

打开页面，可以在开发者工具中的 `Application` 看到 `Service worker` 已经启动了

在 Cache 中也可以发现我们所需的文件已被缓存

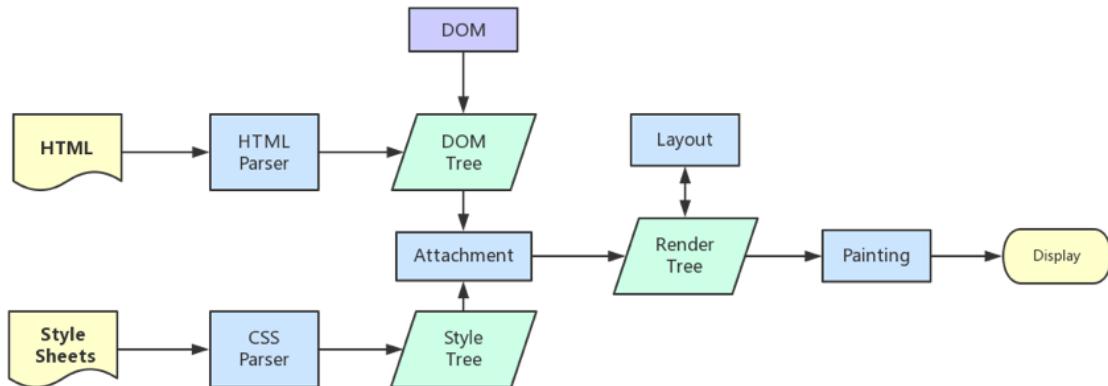
The screenshot shows the Chrome DevTools Application tab. On the left, there's a sidebar with sections for Application (Manifest, Service Workers, Clear storage), Storage (Local Storage, Session Storage, IndexedDB, Web SQL, Cookies), and Cache (Cache Storage, Application Cache). In the main area, under Cache Storage, a entry for "my-cache - http://127.0.0.1" is selected. The Path panel on the right shows "index.html" and "index.js" listed.

当我们重新刷新页面可以发现我们缓存的数据是从 `Service worker` 中读取的

#5 渲染机制

浏览器的渲染机制一般分为以下几个步骤

- 处理 `HTML` 并构建 `DOM` 树。
- 处理 `CSS` 构建 `CSSOM` 树。
- 将 `DOM` 与 `CSSOM` 合并成一个渲染树。
- 根据渲染树来布局，计算每个节点的位置。
- 调用 `GPU` 绘制，合成图层，显示在屏幕上



- 在构建 `CSSOM` 树时，会阻塞渲染，直至 `CSSOM` 树构建完成。并且构建 `CSSOM` 树是一个十分消耗性能的过程，所以应该尽量保证层级扁平，减少过度层叠，越是具体的 CSS 选择器，执行速度越慢

- 当 HTML 解析到 `script` 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件。并且 CSS 也会影响 JS 的执行，只有当解析完样式表才会执行 JS，所以也可以认为这种情况下，CSS 也会暂停构建 DOM

图层

一般来说，可以把普通文档流看成一个图层。特定的属性可以生成一个新的图层。不同的图层渲染互不影响，所以对于某些频繁需要渲染的建议单独生成一个新图层，提高性能。但也不能生成过多的图层，会引起反作用

- 通过以下几个常用属性可以生成新图层
 - `3D` 变换: `translate3d`、`translateZ`
 - `will-change`
 - `video`、`iframe` 标签
 - 通过动画实现的 `opacity` 动画转换
 - `position: fixed`

重绘 (Repaint) 和回流 (Reflow)

- 重绘是当节点需要更改外观而不会影响布局的，比如改变 `color` 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流

回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流

- 所以以下几个动作可能会导致性能问题

:

- 改变 `window` 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

很多人不知道的是，重绘和回流其实和 Event Loop 有关

- 当 `Event loop` 执行完 `Microtasks` 后，会判断 `document` 是否需要更新。因为浏览器是 `60Hz` 的刷新率，每 `16ms` 才会更新一次。
- 然后判断是否有 `resize` 或者 `scroll`，有的话会去触发事件，所以 `resize` 和 `scroll` 事件也是至少 `16ms` 才会触发一次，并且自带节流功能。
- 判断是否触发了 `media query`
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 `requestAnimationFrame` 回调
- 执行 `IntersectionObserver` 回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好
- 更新界面
- 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 `requestIdleCallback` 回调

减少重绘和回流

- 使用 `translate` 替代 `top`
- 使用 `visibility` 替换 `display: none`，因为前者只会引起重绘，后者会引发回流（改变了布局）

- 不要使用 `table` 布局，可能很小的一个小改动会造成整个 `table` 的重新布局
- 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 `requestAnimationFrame`
- `CSS` 选择符从右往左匹配查找，避免 `DOM` 深度过深
- 将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 `video` 标签，浏览器会自动将该节点变为图层

#三、性能

#1 DNS 预解析

- `DNS` 解析也是需要时间的，可以通过预解析的方式来预先获得域名所对应的 `IP`

```
<link rel="dns-prefetch" href="//blog.poetries.top">
```

#2 缓存

- 缓存对于前端性能优化来说是个很重要的点，良好的缓存策略可以降低资源的重复加载提高网页的整体加载速度
- 通常浏览器缓存策略分为两种：强缓存和协商缓存

强缓存

实现强缓存可以通过两种响应头实现：`Expires` 和 `Cache-Control`。强缓存表示在缓存期间不需要请求，`state code` 为 `200`

```
Expires: wed, 22 Oct 2018 08:41:00 GMT
```

`Expires` 是 `HTTP / 1.0` 的产物，表示资源会在 `wed, 22 Oct 2018 08:41:00 GMT` 后过期，需要再次请求。并且 `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

```
Cache-control: max-age=30
```

`Cache-Control` 出现于 `HTTP / 1.1`，优先级高于 `Expires`。该属性表示资源会在 `30` 秒后过期，需要再次请求

协商缓存

- 如果缓存过期了，我们就可以使用协商缓存来解决问题。协商缓存需要请求，如果缓存有效会返回 `304`
- 协商缓存需要客户端和服务端共同实现，和强缓存一样，也有两种实现方式

```
Last-Modified` 和 `If-Modified-Since`
```

- `Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来
- 但是如果在本地打开缓存文件，就会造成 `Last-Modified` 被修改，所以在 `HTTP / 1.1` 出现了 `ETag`

```
ETag` 和 `If-None-Match`
```

- `ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来。并且 `ETag` 优先级比 `Last-Modified` 高

选择合适的缓存策略

对于大部分的场景都可以使用强缓存配合协商缓存解决，但是在一些特殊的地方可能需要选择特殊的缓存策略

- 对于某些不需要缓存的资源，可以使用 `Cache-control: no-store`，表示该资源不需要缓存
- 对于频繁变动的资源，可以使用 `Cache-Control: no-cache` 并配合 `ETag` 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新。
- 对于代码文件来说，通常使用 `Cache-Control: max-age=31536000` 并配合策略缓存使用，然后对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件

#3 使用 HTTP / 2.0

- 因为浏览器会有并发请求限制，在 `HTTP / 1.1` 时代，每个请求都需要建立和断开，消耗了好几个 `RTT` 时间，并且由于 `TCP` 慢启动的原因，加载体积大的文件会需要更多的时间
- 在 `HTTP / 2.0` 中引入了多路复用，能够让多个请求使用同一个 `TCP` 链接，极大的加快了网页的加载速度。并且还支持 `Header` 压缩，进一步的减少了请求的数据大小

#4 预加载

- 在开发中，可能会遇到这样的情况。有些资源不需要马上用到，但是希望尽早获取，这时候就可以使用预加载
- 预加载其实是声明式的 `fetch`，强制浏览器请求资源，并且不会阻塞 `onload` 事件，可以使用以下代码开启预加载

```
<link rel="preload" href="http://example.com">
```

预加载可以一定程度上降低首屏的加载时间，因为可以将一些不影响首屏但重要的文件延后加载，唯一缺点就是兼容性不好

#5 预渲染

可以通过预渲染将下载的文件预先在后台渲染，可以使用以下代码开启预渲染

```
<link rel="prerender" href="http://poetries.com">
```

- 预渲染虽然可以提高页面的加载速度，但是要确保该页面百分百会被用户在之后打开，否则就白白浪费资源去渲染

#6 懒执行与懒加载

懒执行

- 懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒

懒加载

- 懒加载就是将不关键的资源延后加载

懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。对于图片来说，先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 `src` 属性，这样图片就会去下载资源，实现了图片懒加载

- 懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等

#7 文件优化

图片优化

对于如何优化图片，有 2 个思路

- 减少像素点
- 减少每个像素点能够显示的颜色

图片加载优化

- 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 `css` 去代替。
- 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 `CDN` 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片
- 小图使用 `base64` 格式
- 将多个图标文件整合到一张图片中（雪碧图）
- 选择正确的图片格式：
 - 对于能够显示 `WebP` 格式的浏览器尽量使用 `WebP` 格式。因为 `WebP` 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
 - 小图使用 `PNG`，其实对于大部分图标这类图片，完全可以使用 `SVG` 代替
 - 照片使用 `JPEG`

其他文件优化

- `CSS` 文件放在 `head` 中
- 服务端开启文件压缩功能
- 将 `script` 标签放在 `body` 底部，因为 `js` 文件执行会阻塞渲染。当然也可以把 `script` 标签放在任意位置然后加上 `defer`，表示该文件会并行下载，但是会放到 `HTML` 解析完成后顺序执行。对于没有任何依赖的 `js` 文件可以加上 `async`，表示加载和渲染后续文档元素的过程将和 `js` 文件的加载与执行并行无序进行。执行 `js` 代码过长会卡住渲染，对于需要很多时间计算的代码
- 可以考虑使用 `webworker`。`webworker` 可以让我们另开一个线程执行脚本而不影响渲染。

CDN

静态资源尽量使用 `CDN` 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 `CDN` 域名。对于 `CDN` 加载静态资源需要注意 `CDN` 域名要与主站不同，否则每次请求都会带上主站的 `Cookie`

#8 其他

使用 Webpack 优化项目

- 对于 `webpack4`，打包项目使用 `production` 模式，这样会自动开启代码压缩
- 使用 `ES6` 模块来开启 `tree shaking`，这个技术可以移除没有使用的代码
- 优化图片，对于小图可以使用 `base64` 的方式写入文件中
- 按照路由拆分代码，实现按需加载
- 给打包出来的文件名添加哈希，实现浏览器缓存文件

监控

对于代码运行错误，通常的办法是使用 `window.onerror` 拦截报错。该方法能拦截到大部分的详细报错信息，但是也有例外

- 对于跨域的代码运行错误会显示 `script error`。对于这种情况我们需要给 `script` 标签添加 `crossorigin` 属性
- 对于某些浏览器可能不会显示调用栈信息，这种情况可以通过 `arguments.callee.caller` 来做栈递归
- 对于异步代码来说，可以使用 `catch` 的方式捕获错误。比如 `Promise` 可以直接使用 `catch` 函数，`async await` 可以使用 `try catch`
- 但是要注意线上运行的代码都是压缩过的，需要在打包时生成 `sourceMap` 文件便于 `debug`。
- 对于捕获的错误需要上传给服务器，通常可以通过 `img` 标签的 `src` 发起一个请求

#四、安全

#1 XSS

跨网站指令码（英语：`Cross-site scripting`，通常简称为：`XSS`）是一种网站应用程式的安全漏洞攻击，是代码注入的一种。它允许恶意使用者将程式码注入到网页上，其他使用者在观看网页时就会受到影响。这类攻击通常包含了 `HTML` 以及使用者端脚本语言

`XSS` 分为三种：反射型，存储型和 `DOM-based`

如何攻击

- `XSS` 通过修改 `HTML` 节点或者执行 `JS` 代码来攻击网站。
- 例如通过 `URL` 获取某些参数

```
<!-- http://www.domain.com?name=<script>alert(1)</script> -->
<div>{{name}}</div>
```

上述 `URL` 输入可能会将 `HTML` 改为 `<div><script>alert(1)</script></div>`，这样页面中就凭空多了一段可执行脚本。这种攻击类型是反射型攻击，也可以说是 `DOM-based` 攻击

如何防御

最普遍的做法是转义输入输出的内容，对于引号，尖括号，斜杠进行转义

```
function escape(str) {
    str = str.replace(/&/g, "&amp;");
    str = str.replace(/</g, "&lt;");
    str = str.replace(/>/g, "&gt;");
    str = str.replace(/\"/g, "&quot;");
    str = str.replace(/\'/g, "&#39;");
    str = str.replace(/\^/g, "&#96;");
    str = str.replace(/\//g, "&#x2F;");
    return str
}
```

通过转义可以将攻击代码 `<script>alert(1)</script>` 变成

```
// -> <script>alert(1)</script>
escape('<script>alert(1)</script>')
```

对于显示富文本来说，不能通过上面的办法来转义所有字符，因为这样会把需要的格式也过滤掉。这种情况通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式

```
var xss = require("xss");
var html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>');
// -> <h1>XSS Demo</h1>&lt;script&ampgtalert("xss");&lt;/script&ampgt;
console.log(html);
```

以上示例使用了 `js-xss` 来实现。可以看到在输出中保留了 `h1` 标签且过滤了 `script` 标签

#2 CSRF

跨站请求伪造（英语：`Cross-site request forgery`），也被称为 `one-click attack` 或者 `session riding`，通常缩写为 `CSRF` 或者 `XSRF`，是一种挟制用户在当前已登录的 `web` 应用程序上执行非本意的操作的攻击方法

`CSRF` 就是利用用户的登录态发起恶意请求

如何攻击

假设网站中有一个通过 `Get` 请求提交用户评论的接口，那么攻击者就可以在钓鱼网站中加入一个图片，图片的地址就是评论接口

```

```

如何防御

- `Get` 请求不对数据进行修改
- 不让第三方网站访问到用户 `Cookie`
- 阻止第三方网站请求接口
- 请求时附带验证信息，比如验证码或者 `token`

#3 密码安全

加盐

对于密码存储来说，必然是不能明文存储在数据库中的，否则一旦数据库泄露，会对用户造成很大的损失。并且不建议只对密码单纯通过加密算法加密，因为存在彩虹表的关系

- 通常需要对密码加盐，然后进行几次不同加密算法的加密

```
// 加盐也就是给原密码添加字符串，增加原密码长度
sha256(sha1(md5(salt + password + salt)))
```

但是加盐并不能阻止别人盗取账号，只能确保即使数据库泄露，也不会暴露用户的真实密码。一旦攻击者得到了用户的账号，可以通过暴力破解的方式破解密码。对于这种情况，通常使用验证码增加延时或者限制尝试次数的方式。并且一旦用户输入了错误的密码，也不能直接提示用户输错密码，而应该提示账号或密码错误

前端加密

虽然前端加密对于安全防护来说意义不大，但是在遇到中间人攻击的情况下，可以避免明文密码被第三方获取

#五、小程序

#1 登录

unionid和openid

了解小程序登陆之前，我们先了解下小程序/公众号登录涉及到两个最关键的用户标识：

- `OpenID` 是一个用户对于一个小程序 / 公众号的标识，开发者可以通过这个标识识别出用户。
- `UnionId` 是一个用户对于同主体微信小程序 / 公众号 / APP 的标识，开发者需要在微信开放平台下绑定相同账号的主体。开发者可通过 `UnionId`，实现多个小程序、公众号、甚至APP 之间的数据互通了。

关键Api

- `wx.login` 官方提供的登录能力
- `wx.checkSession` 校验用户当前的 `session_key` 是否有效
- `wx.authorize` 提前向用户发起授权请求
- `wx.getUserInfo` 获取用户基本信息

登录流程设计

- **利用现有登录体系**

直接复用现有系统的登录体系，只需要在小程序端设计用户名，密码/验证码输入页面，便可以简便的实现登录，只需要保持良好的用户体验即可

- **利用 `OpenID` 创建用户体系**

`OpenID` 是一个小程序对于一个用户的标识，利用这一点我们可以轻松的实现一套基于小程序的用户体系，值得一提的是这种用户体系对用户的打扰最低，可以实现静默登录。具体步骤如下

- 小程序客户端通过 `wx.login` 获取 `code`
- 传递 `code` 向服务端，服务端拿到 `code` 调用微信登录凭证校验接口，微信服务器返回 `openid` 和会话密钥 `session_key`，此时开发者服务端便可以利用 `openid` 生成用户入库，再向小程序客户端返回自定义登录态
- 小程序客户端缓存（通过 `storage`）自定义登录态（`token`），后续调用接口时携带该登录态作为用户身份标识即可

利用 `Unionid` 创建用户体系

如果想实现多个小程序，公众号，已有登录系统的数据互通，可以通过获取到用户 `unionid` 的方式建立用户体系。因为 `unionid` 在同一开放平台下的所有应用都是相同的，通过 `unionid` 建立的用户体系即可实现全平台数据的互通，更方便的接入原有的功能，那如何获取 `unionid` 呢，有以下两种方式

- 如果户关注了某个相同主体公众号，或曾经在某个相同主体 App、公众号上进行过微信登录授权，通过 `wx.login` 可以直接获取到 `unionid`
- 结合

```
wx.getUserInfo
```

和

```
<button open-type="getUserInfo"></button>
```

这两种方式引导用户主动授权，主动授权后通过返回的信息和服务端交互(这里有一步需要服务端解密数据的过程，很简单，微信提供了示例代码)即可拿到

```
unionid
```

建立用户体系，然后由服务端返回登录态，本地记录即可实现登录，附上微信提供的最佳实践

- 调用 `wx.login` 获取 `code`，然后从微信后端换取到 `session_key`，用于解密 `userInfo` 返回的敏感数据
- 使用

```
wx.getSetting
```

获取用户的授权情况

- 如果用户已经授权，直接调用 API `wx.getUserInfo` 获取用户最新的信息；
- 用户未授权，在界面中显示一个按钮提示用户登入，当用户点击并授权后就获取到用户的最新信息
- 获取到用户数据后可以进行展示或者发送给自己的后端。

注意事项

- 需要获取 `unionid` 形式的登录体系，在以前（18年4月之前）是通过以下这种方式来实现，但后续微信做了调整（因为一进入小程序，主动弹起各种授权弹窗的这种形式，比较容易导致用户流失），调整为必须使用按钮引导用户主动授权的方式，这次调整对开发者影响较大，开发者需要注意遵守微信的规则，并及时和业务方沟通业务形式，不要存在侥幸心理，以防造成小程序不过审等情况

```
wx.login(获取code) ==> wx.getUserInfo(用户授权) ==> 获取 unionid
```

- 因为小程序不存在 `cookie` 的概念，登录态必须缓存在本地，因此强烈建议为登录态设置过期时间
- 值得一提的是如果需要支持风控安全校验，多平台登录等功能，可能需要加入一些公共参数，例如 `platform`, `channel`, `deviceParam` 等参数。在和服务端确定方案时，作为前端同学应该及时提出这些合理的建议，设计合理的系统。
- `openid`, `unionid` 不要在接口中明文传输，这是一种危险的行为，同时也很不专业

#2 图片导出

这是一种常见的引流方式，一般同时会在图片中附加一个小程序二维码。

基本原理

- 借助 `canvas` 元素，将需要导出的样式首先在 `canvas` 画布上绘制出来（`api` 基本和 `h5` 保持一致，但有轻微差异，使用时注意即可）
- 借助微信提供的 `canvasToTempFilePath` 导出图片，最后再使用 `saveImageToPhotosAlbum`（需要授权）保存图片到本地

如何优雅实现

- 绘制出需要的样式这一步是省略不掉的。但是我们可以封装一个绘制库，包含常见图形的绘制，例如矩形，圆角矩形，圆，扇形，三角形，文字，图片减少绘制代码，只需要提炼出样式信息，便可以轻松的绘制，最后导出图片存入相册。笔者觉得以下这种方式绘制更为优雅清晰一些，其实也可以使用加入一个type参数来指定绘制类型，传入的一个是样式数组，实现绘制。
- 结合上一步的实现，如果对于同一类型的卡片有多次导出需求的场景，也可以使用自定义组件的方式，封装同一类型的卡片为一个通用组件，在需要导出图片功能的地方，引入该组件即可。

```

class Canvaskit {
  constructor() {
  }
  drawImg(option = {}) {
    ...
    return this
  }
  drawRect(option = {}) {
    return this
  }
  drawText(option = {}) {
    ...
    return this
  }
  static exportImg(option = {}) {
    ...
  }
}

let drawer = new Canvaskit('canvasId').drawImg(styleObj1).drawText(styleObj2)
drawer.exportImg()

```

注意事项

- 小程序中无法绘制网络图片到 canvas 上，需要通过 `downLoadFile` 先下载图片到本地临时文件才可以绘制
- 通常需要绘制二维码到导出的图片上，有一种方式导出二维码时，需要携带的参数必须做编码，而且有具体的长度（32可见字符）限制，可以借助服务端生成 短链接 的方式来解决

#3 数据统计

数据统计作为目前一种常用的分析用户行为的方式，小程序端也是必不可少的。小程序采取的曝光，点击数据埋点其实和h5原理是一样的。但是埋点作为一个和业务逻辑不相关的需求，我们如果在每一个点击事件，每一个生命周期加入各种埋点代码，则会干扰正常的业务逻辑，和使代码变的臃肿，笔者提供以下几种思路来解决数据埋点

设计一个埋点sdk

小程序的代码结构是，每一个 `Page` 中都有一个 `Page` 方法，接受一个包含生命周期函数，数据的业务逻辑对象 包装这层数据，借助小程序的底层逻辑实现页面的业务逻辑。通过这个我们可以想到思路，对 `Page` 进行一次包装，篡改它的生命周期和点击事件，混入埋点代码，不干扰业务逻辑，只要做一些简单的配置即可埋点，简单的代码实现如下

```

// 代码仅供理解思路
page = function(params) {
  let keys = params.keys()
  keys.forEach(v => {
    if (v === 'onLoad') {
      params[v] = function(options) {

```

```
    stat() // 曝光埋点代码
    params[v].call(this, options)
}
}
else if (v.includes('click')) {
  params[v] = funciton(event) {
    let data = event.dataset.config
    stat(data) // 点击埋点
    param[v].call(this)
  }
}
})
}
```

这种思路不光适用于埋点，也可以用来作全局异常处理，请求的统一处理等场景。

分析接口

对于特殊的一些业务，我们可以采取 接口埋点，什么叫接口埋点呢？很多情况下，我们有的 api 并不是多处调用的，只会在某一个特定的页面调用，通过这个思路我们可以分析出，该接口被请求，则这个行为被触发了，则完全可以通过服务端日志得出埋点数据，但是这种方式局限性较大，而且属于分析结果得出过程，可能存在误差，但可以作为一种思路了解一下。

微信自定义数据分析

微信本身提供的数据分析能力，微信本身提供了常规分析和自定义分析两种数据分析方式，在小程序后台配置即可。借助小程序数据助手这款小程序可以很方便的查看

#4 工程化

工程化做什么

目前的前端开发过程，工程化是必不可少的一环，那小程序工程化都需要做些什么呢，先看下目前小程序开发当中存在哪些问题需要解决：

- 不支持 css 预编译器，作为一种主流的 css 解决方案，不论是 less, sass, stylus 都可以提升 css 效率
- 不支持引入 npm 包（这一条，从微信公开课中听闻，微信准备支持）
- 不支持 ES7 等后续的 js 特性，好用的 async await 等特性都无法使用
- 不支持引入外部字体文件，只支持 base64
- 没有 eslint 等代码检查工具

方案选型

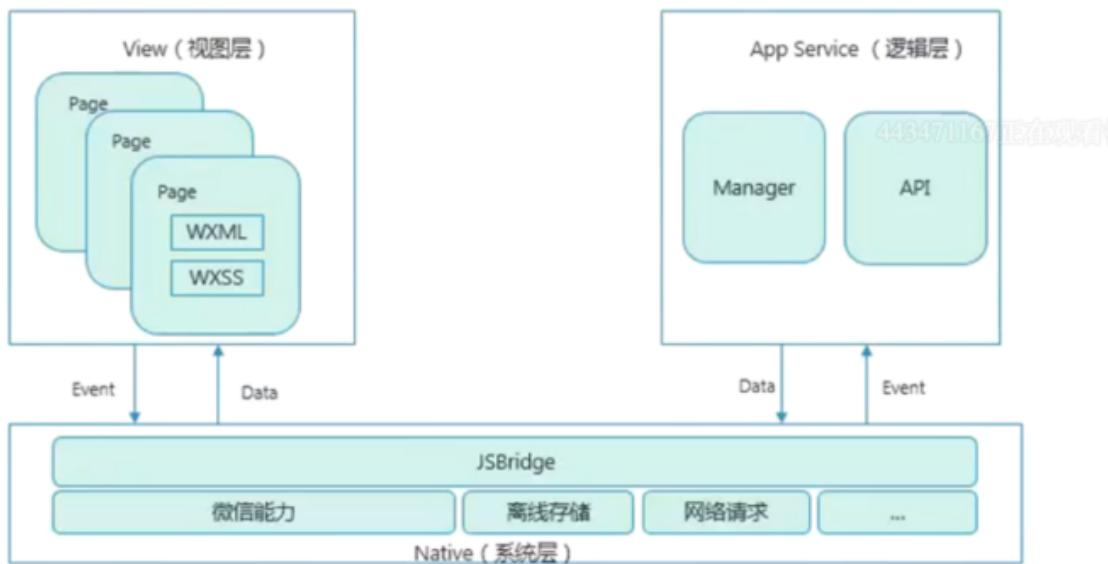
对于目前常用的工程化方案，webpack, rollup, parcel 等来看，都常用与单页应用的打包和处理，而小程序天生是“多页应用”并且存在一些特定的配置。根据要解决的问题来看，无非是文件的编译，修改，拷贝这些处理，对于这些需求，我们想到基于流的 gulp 非常的适合处理，并且相对于 webpack 配置多页应用更加简单。所以小程序工程化方案推荐使用 gulp

具体开发思路

通过 gulp 的 task 实现：

- 实时编译 less 文件至相应目录
- 引入支持 async, await 的运行时文件
- 编译字体文件为 base64 并生成相应 css 文件，方便使用
- 依赖分析哪些地方引用了 npm 包，将 npm 包打成一个文件，拷贝至相应目录
- 检查代码规范

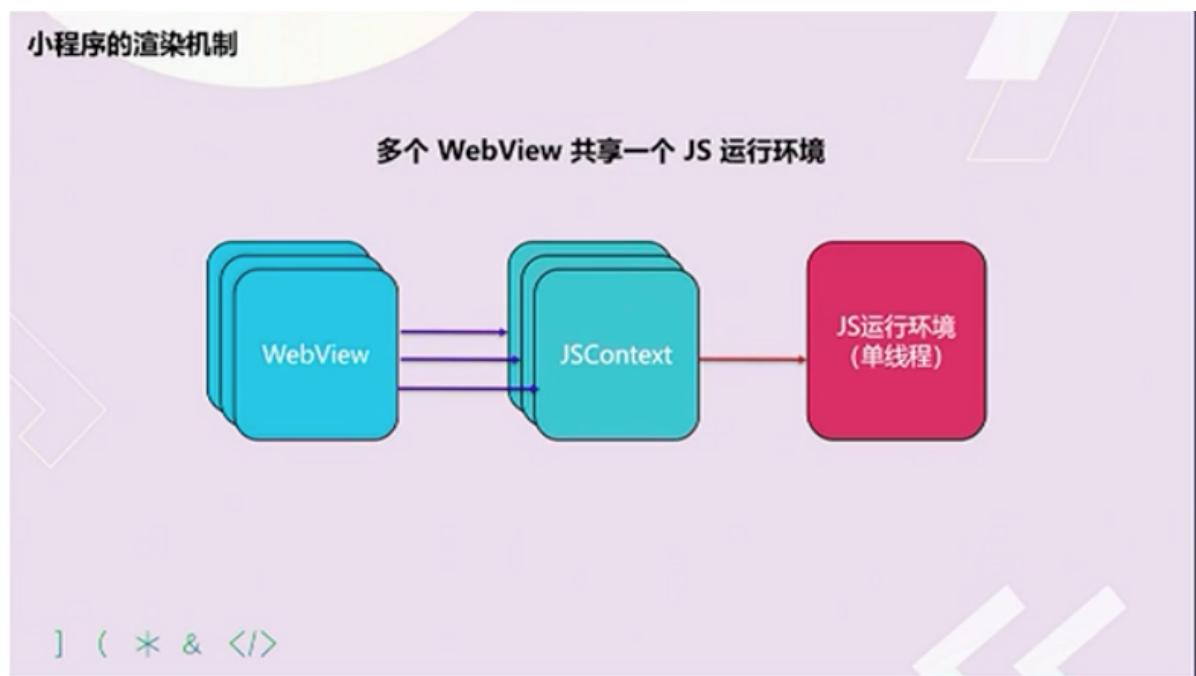
#5 小程序架构



微信小程序的框架包含两部分 `view` 视图层、`App Service` 逻辑层。`view` 层用来渲染页面结构，`AppService` 层用来逻辑处理、数据请求、接口调用。

它们在两个线程里运行。

视图层和逻辑层通过系统层的 `JSBridge` 进行通信，逻辑层把数据变化通知到视图层，触发视图层页面更新，视图层把触发的事件通知到逻辑层进行业务处理



- 视图层使用 `webview` 渲染，`ios` 中使用自带 `wkwebview`，在 `Android` 使用腾讯的 `x5` 内核（基于 `blink`）运行。
- 逻辑层使用在 `ios` 中使用自带的 `jscore` 运行，在 `Android` 中使用腾讯的 `x5` 内核（基于 `blink`）运行。
- 开发工具使用 `nw.js` 同时提供了视图层和逻辑层的运行环境。

#6 WXML && WXSS

WXML

- 支持数据绑定
- 支持逻辑算术、运算
- 支持模板、引用
- 支持添加事件 (`bindtap`)
- `wxml` 编译器: `wcc` 把 `wxml` 文件转为 `js`
- 执行方式: `wcc index.wxml`
- 使用 `virtual DOM`, 进行局部更新

WXSS

- `WXSS` 编译器: `wcsc` 把 `wxss` 文件转化为 `js`
- 执行方式: `wcsc index.wxss`

尺寸单位 rpx

`rpx (responsive pixel)` : 可以根据屏幕宽度进行自适应。规定屏幕宽为 `750rpx`。公式:

```
const dswidth = 750

export const screenHeightOfRpx = function () {
  return 750 / env.screenWidth * env.screenHeight
}

export const rpxToPx = function (rpx) {
  return env.screenWidth / 750 * rpx
}

export const pxToRpx = function (px) {
  return 750 / env.screenWidth * px
}
```

样式导入

使用 `@import` 语句可以导入外联样式表, `@import` 后跟需要导入的外联样式表的相对路径, 用 `;` 表示语句结束

内联样式

静态的样式统一写到 `class` 中。`style` 接收动态的样式, 在运行时会进行解析, 请尽量避免将静态的样式写进 `style` 中, 以免影响渲染速度

全局样式与局部样式

定义在 `app.wxss` 中的样式为全局样式, 作用于每一个页面。在 `page` 的 `wxss` 文件中定义的样式为局部样式, 只作用在对应的页面, 并会覆盖 `app.wxss` 中相同的选择器

#7 小程序的问题

- 小程序仍然使用 `webview` 渲染, 并非原生渲染。 (部分原生)
- 服务端接口返回的头无法执行, 比如: `Set-Cookie`。
- 依赖浏览器环境的 `JS` 库不能使用。
- 不能使用 `npm`, 但是可以自搭构建工具或者使用 `mpvue`。 (未来官方有计划支持)

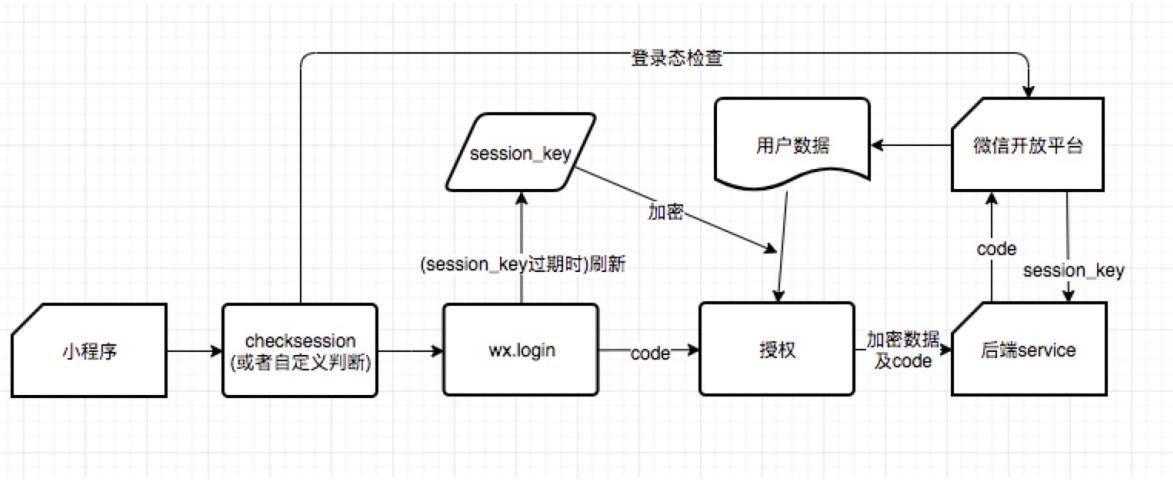
- 不能使用 `ES7`，可以自己用 `babel+webpack` 自搭或者使用 `mpvue`。
- 不支持使用自己的字体（未来官方计划支持）。
- 可以用 `base64` 的方式来使用 `iconfont`。
- 小程序不能发朋友圈（可以通过保存图片到本地，发图片到朋友前。二维码可以使用B接口）。
- 获取二维码/小程序接口的限制
- 程序推送只能使用“服务通知”而且需要用户主动触发提交 `formId`，`formId` 只有7天有效期。
(现在的做法是在每个页面都放入 `form` 并且隐藏以此获取更多的 `formId`。后端使用原则为：优先使用有效期最短的)
- 小程序大小限制 2M，分包总计不超过 8M
- 转发（分享）小程序不能拿到成功结果，原来可以。链接（小游戏造的孽）
- 拿到相同的

`unionId`

必须绑在同一个开放平台下。开放平台绑定限制：

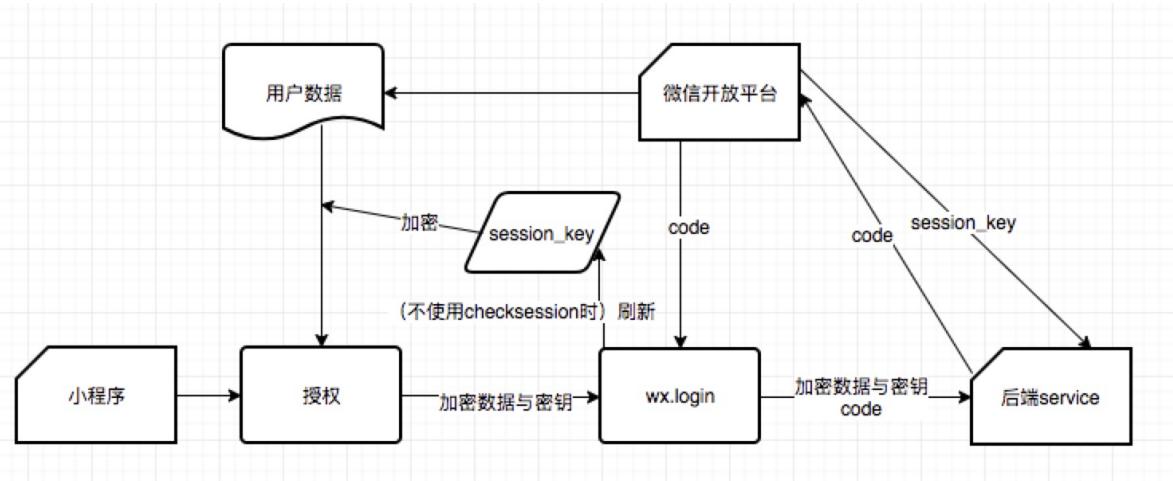
- 50 个移动应用
- 10 个网站
- 50 个同主体公众号
- 5 个不同主体公众号
- 50 个同主体小程序
- 5 个不同主体小程序
- 公众号关联小程序
 - 所有公众号都可以关联小程序。
 - 一个公众号可关联10个同主体的小程序，3个不同主体的小程序。
 - 一个小程序可关联500个公众号。
 - 公众号一个月可新增关联小程序13次，小程序一个月可新增关联500次。
- 一个公众号关联的10个同主体小程序和3个非同主体小程序可以互相跳转
- 品牌搜索不支持金融、医疗
- 小程序授权需要用户主动点击
- 小程序不提供测试 `access_token`
- 安卓系统下，小程序授权获取用户信息之后，删除小程序再重新获取，并重新授权，得到旧签名，导致第一次授权失败
- 开发者工具上，授权获取用户信息之后，如果清缓存选择全部清除，则即使使用了 `wx.checkSession`，并且在 `session_key` 有效期内，授权获取用户信息也会得到新的 `session_key`

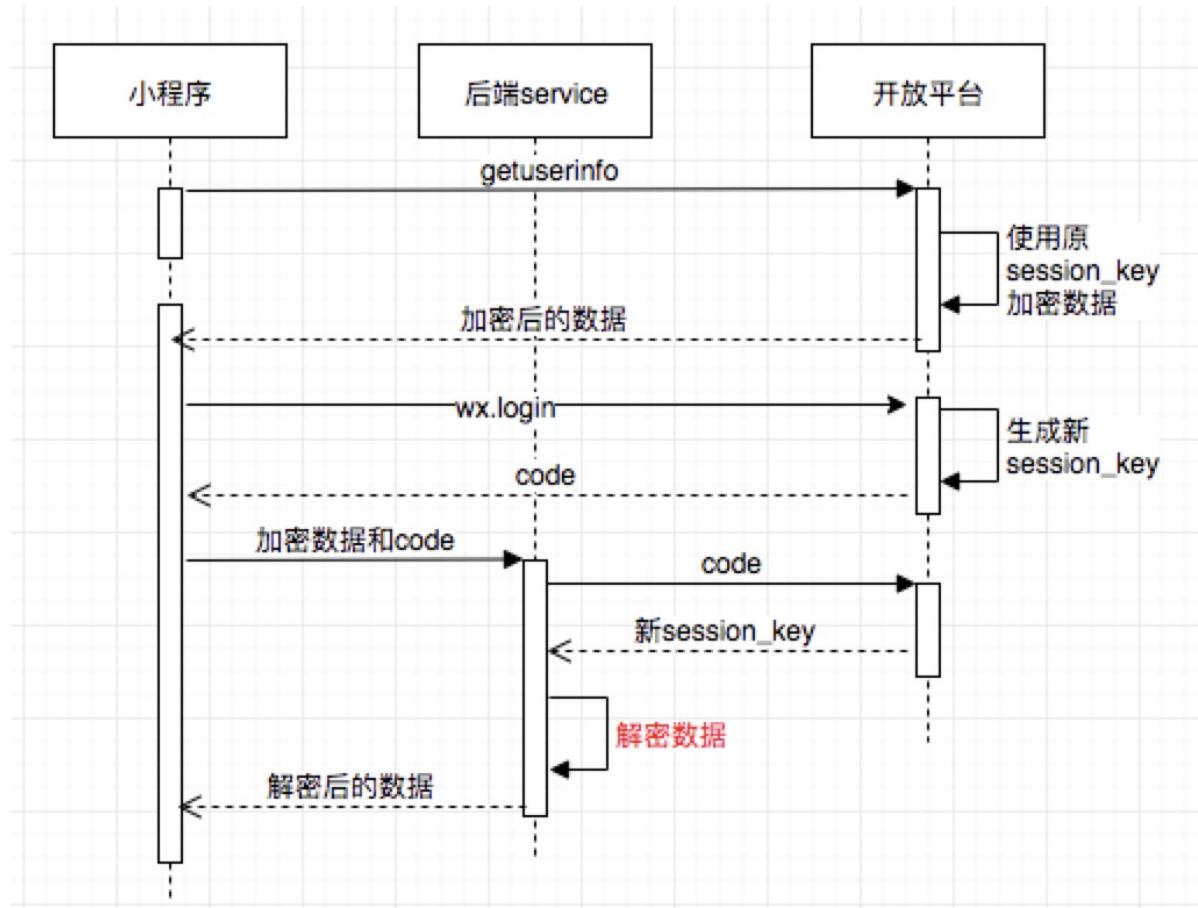
#8 授权获取用户信息流程



- `session_key` 有有效期，有效期并没有被告知开发者，只知道用户越频繁使用小程序，`session_key` 有效期越长
- 在调用 `wx.login` 时会直接更新 `session_key`，导致旧 `session_key` 失效
- 小程序内先调用 `wx.checkSession` 检查登录态，并保证没有过期的 `session_key` 不会被更新，再调用 `wx.login` 获取 `code`。接着用户授权小程序获取用户信息，小程序拿到加密后的用户数据，把加密数据和 `code` 传给后端服务。后端通过 `code` 拿到 `session_key` 并解密数据，将解密后的用户信息返回给小程序

面试题：先授权获取用户信息再 `login` 会发生什么？





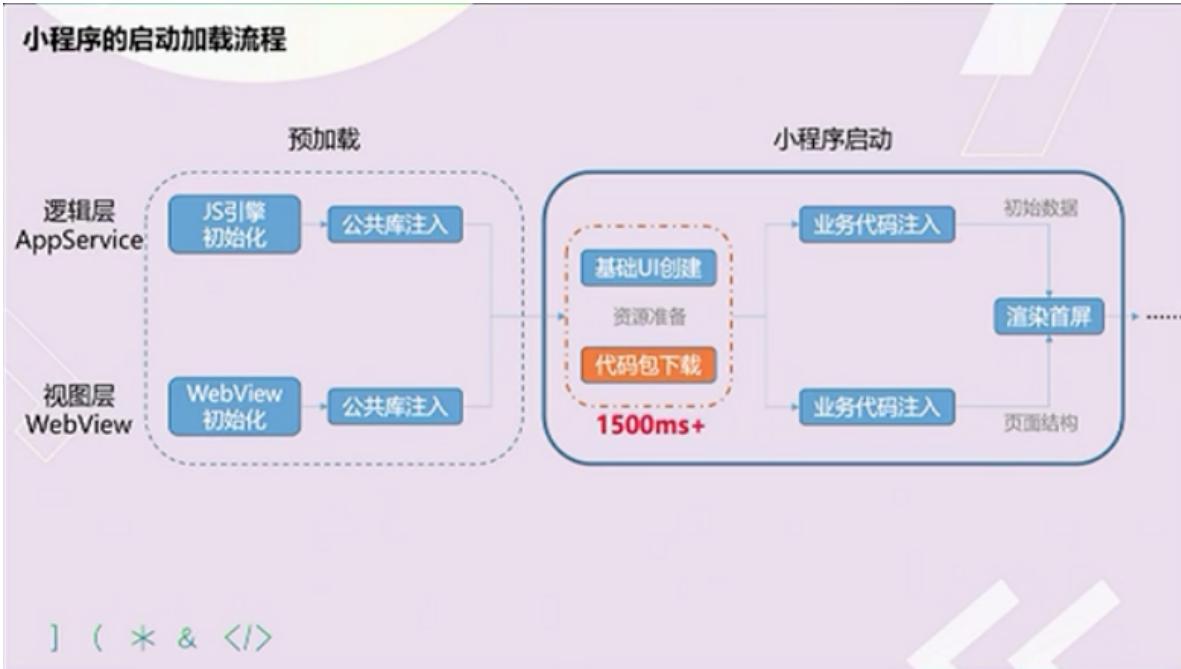
- 用户授权时，开放平台使用旧的 `session_key` 对用户信息进行加密。调用 `wx.login` 重新登录，会刷新 `session_key`，这时后端服务从开放平台获取到新 `session_key`，但是无法对老 `session_key` 加密过的数据解密，用户信息获取失败
- 在用户信息授权之前先调用 `wx.checkSession` 呢？`wx.checkSession` 检查登录态，并且保证 `wx.login` 不会刷新 `session_key`，从而让后端服务正确解密数据。但是这里存在一个问题，如果小程序较长时间不用导致 `session_key` 过期，则 `wx.login` 必定会重新生成 `session_key`，从而再一次导致用户信息解密失败

#9 性能优化

我们知道 `view` 部分是运行在 `webview` 上的，所以前端领域的大多数优化方式都有用

加载优化

小程序的启动加载流程



代码包的大小是最直接影响小程序加载启动速度的因素。代码包越大不仅下载速度时间长，业务代码注入时间也会变长。所以最好的优化方式就是减少代码包的大小

小程序加载的三个阶段的表示



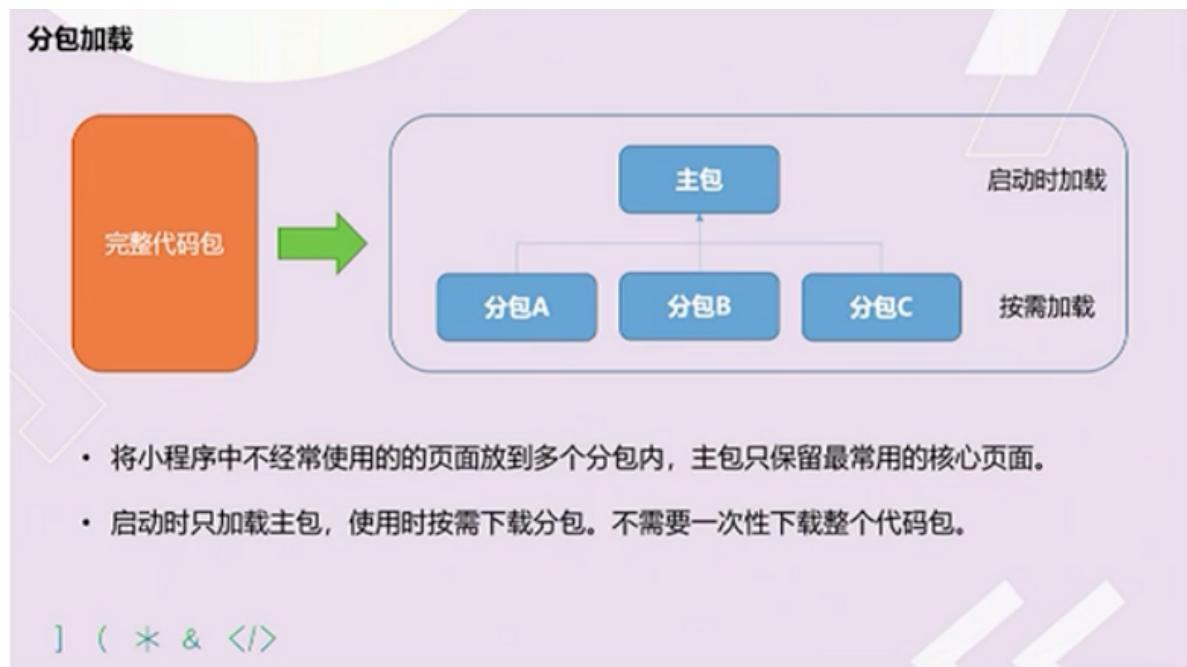
优化方式

- 代码压缩。
- 及时清理无用代码和资源文件。
- 减少代码包中的图片等资源文件的大小和数量。
- 分包加载。

首屏加载的体验优化建议

- 提前请求: 异步数据请求不需要等待页面渲染完成。
- 利用缓存: 利用 `storage API` 对异步请求数据进行缓存, 二次启动时先利用缓存数据渲染页面, 在进行后台更新。
- 避免白屏: 先展示页面骨架页和基础内容。
- 及时反馈: 即时地对需要用户等待的交互操作给出反馈, 避免用户以为小程序无响应

使用分包加载优化



- 在构建小程序分包项目时，构建会输出一个或多个功能的分包，其中每个分包小程序必定含有一个主包，所谓的主包，即放置默认启动页面/TabBar 页面，以及一些所有分包都需用到公共资源/JS 脚本，而分包则是根据开发者的配置进行划分
- 在小程序启动时，默认会下载主包并启动主包内页面，如果用户需要打开分包内某个页面，客户端会把对应分包下载下来，下载完成后再进行展示。

优点：

- 对开发者而言，能使小程序有更大的代码体积，承载更多的功能与服务
- 对用户而言，可以更快地打开小程序，同时在不影响启动速度前提下使用更多功能

限制

- 整个小程序所有分包大小不超过 8M
- 单个分包/主包大小不能超过 2M
- 原生分包加载的配置 假设支持分包的小程序目录结构如下

```
|── app.js  
|── app.json  
|── app.wxss  
|── packageA  
|   └── pages  
|       └── cat  
|           └── dog  
|── packageB  
|   └── pages  
|       └── apple  
|           └── banana  
└── pages  
    └── index  
    └── logs  
└── utils
```

开发者通过在 `app.json` `subPackages` 字段声明项目分包结构

{

```
"pages": [
  "pages/index",
  "pages/logs"
],
"subPackages": [
  {
    "root": "packageA",
    "pages": [
      "pages/cat",
      "pages/dog"
    ]
  },
  {
    "root": "packageB",
    "pages": [
      "pages/apple",
      "pages/banana"
    ]
  }
]
```

分包原则

- 声明 `subPackages` 后，将按 `subPackages` 配置路径进行打包，`subPackages` 配置路径外的目录将被打包到 `app`（主包）中
- `app`（主包）也可以有自己的 `pages`（即最外层的 `pages` 字段）
- `subPackage` 的根目录不能是另外一个 `subPackage` 内的子目录
- 首页的 TAB 页面必须在 `app`（主包）内

引用原则

- `packageA` 无法 `require packageB` JS 文件，但可以 `require app`、自己 `package` 内的 JS 文件
- `packageA` 无法 `import packageB` 的 `template`，但可以 `require app`、自己 `package` 内的 `template`
- `packageA` 无法使用 `packageB` 的资源，但可以使用 `app`、自己 `package` 内的资源

官方即将推出 分包预加载

分包加载——分包预下载（即将推出）

- 现状：用户首次进入分包页面时需要进行分包的下载和注入，造成页面切换的延迟。



- 分包预下载——开发者预先配置页面可能会跳转到的分包，框架在进入页面后根据配置进行预下载。



】 (* & </>

独立分包

分包加载——独立分包（即将推出）

- 现状：从分包页面启动时，必须要先依赖于主包的下载和注入。启动速度受主包限制。



- 独立分包——可以不依赖主包，独立下载和运行的分包。从独立分包页面启动，只下载和注入分包就可以打开页面。

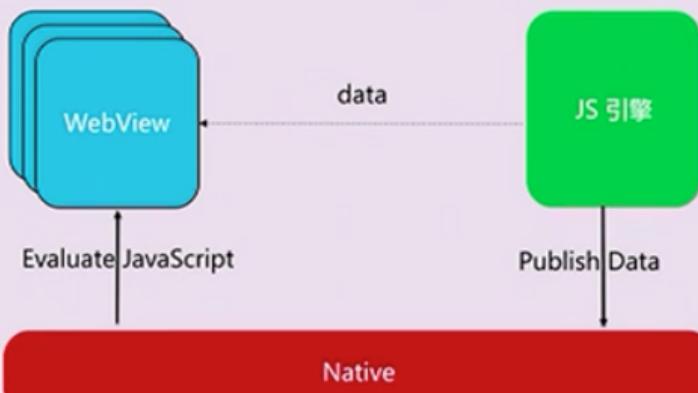


] (* & </>

渲染性能优化

小程序的渲染机制

逻辑层调用 setData 更新视图层的页面内容



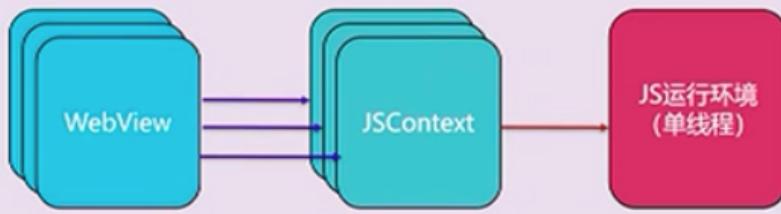
] (* & </>

- 每次 setData 的调用都是一次进程间通信过程，通信开销与 setData 的数据量正相关。
- setData 会引发视图层页面内容的更新，这一耗时操作一定时间中会阻塞用户交互。
- setData 是小程序开发使用最频繁，也是最容易引发性能问题的

避免不当使用 setData

- 使用 data 在方法间共享数据，可能增加 setData 传输的数据量。。 data 应仅包括与页面渲染相关的数据。
- 使用 setData 传输大量数据，通讯耗时与数据正相关，页面更新延迟可能造成页面更新开销增加。仅传输页面中发生变化的数据，使用 setData 的特殊 key 实现局部更新。
- 短时间内频繁调用 setData，操作卡顿，交互延迟，阻塞通信，页面渲染延迟。避免不必要的 setData，对连续的 setData 调用进行合并。
- 在后台页面进行 setData，抢占前台页面的渲染资源。页面切入后台后的 setData 调用，延迟到页面重新展示时执行。

多个 WebView 共享一个 JS 运行环境



] (* & </>

避免不当使用onPageScroll

- 只在必要的时候监听 `pagescroll` 事件。不监听，则不会派发。
- 避免在 `onPageScroll` 中执行复杂逻辑
- 避免在 `onPageScroll` 中频繁调用 `setData`
- 避免滑动时频繁查询节点信息 (`selectQuery`) 用以判断是否显示，部分场景建议使用节点布局橡胶状态监听 (`intersectionobserver`) 替代

使用自定义组件

在需要频繁更新的场景下，自定义组件的更新只在组件内部进行，不受页面其他部分内容复杂性影响

#10 wepy vs mpvue

数据流管理

相比传统的小程序框架，这个一直是我们作为资深开发者比较期望去解决的，在 `web` 开发中，随着 `Flux`、`Redux`、`Vuex` 等多个数据流工具出现，我们也期望在业务复杂的小程序中使用

- `Wepy` 默认支持 `Redux`，在脚手架生成项目的时候可以内置
- `Mpvue` 作为 `Vue` 的移植版本，当然支持 `Vuex`，同样在脚手架生成项目的时候可以内置

组件化

- `Wepy` 类似 `Vue` 实现了单文件组件，最大的差别是文件后缀 `.wepy`，只是写法上会有差异

```
export default class Index extends wepy.page {}
```

- `Mpvue` 作为 `Vue` 的移植版本，支持单文件组件，`template`、`script` 和 `style` 都在一个 `.vue` 文件中，和 `Vue` 的写法类似，所以对 `Vue` 开发熟悉的同学会比较适应

工程化

所有的小程序开发依赖官方提供的开发者工具。开发者工具简单直观，对调试小程序很有帮助，现在也支持腾讯云（目前我们还没有使用，但是对新的一年还是有帮助的），可以申请测试报告查看小程序在真实的移动设备上运行性能和运行效果，但是它本身没有类似前端工程化中的概念和工具

- `wepy` 内置了构建，通过 `wepy init` 命令初始化项目，大致流程如下：
 - `wepy-cli` 会判断模版是在远程仓库还是在本地，如果在本地则会立即跳到第 3 步，反之继续进行。
 - 会从远程仓库下载模版，并保存到本地。
 - 询问开发者 `Project name` 等问题，依据开发者的回答，创建项目
- `mpvue` 沿用了 `vue` 中推崇的 `webpack` 作为构建工具，但同时提供了一些自己的插件以及配置文件的一些修改，比如
 - 不再需要 `html-webpack-plugin`
 - 基于 `webpack-dev-middleware` 修改成 `webpack-dev-middleware-hard-disk`
 - 最大的变化是基于 `webpack-loader` 修改成 `mpvue-loader`
 - 但是配置方式还是类似，分环境配置文件，最终都会编译成小程序支持的目录结构和文件后缀

#11 mpvue

mpvue

`vue.js` 小程序版，`fork` 自 `vuejs/vue@2.4.1`，保留了 `vue runtime` 能力，添加了小程序平台的支持。`mpvue` 是一个使用 `vue.js` 开发小程序的前端框架。框架基于 `vue.js` 核心，`mpvue` 修改了 `vue.js` 的 `runtime` 和 `compiler` 实现，使其可以运行在小程序环境中，从而为小程序开发引入了整套 `vue.js` 开发体验

框架原理

两个大方向

- 通过 `mpvue` 提供 `mp` 的 `runtime` 适配小程序
- 通过 `mpvue-loader` 产出微信小程序所需要的文件结构和模块内容

七个具体问题

- 要了解 `mpvue` 原理必然要了解 `vue` 原理，这是大前提

现在假设您对 `Vue` 原理有个大概的了解

- 由于 `Vue` 使用了 `Virtual DOM`，所以 `Virtual DOM` 可以在任何支持 `Javascript` 语言的平台上操作，譬如说目前 `Vue` 支持浏览器平台或 `Weex`，也可以是 `mp`（小程序）。那么最后 `Virtual DOM` 如何映射到真实的 `DOM` 节点上呢？`Vue` 为平台做了一层适配层，浏览器平台见 `runtime/node-ops.js`、`Weex` 平台见 `runtime/node-ops.js`，小程序见 `runtime/node-ops.js`。不同平台之间通过适配层对外提供相同的接口，`Virtual DOM` 进行操作 `Real DOM` 节点的时候，只需要调用这些适配层的接口即可，而内部实现则不需要关心，它会根据平台的改变而改变
- 所以思路肯定是往增加一个 `mp` 平台的 `runtime` 方向走。但问题是小程序不能操作 `DOM`，所以 `mp` 下的 `node-ops.js` 里面的实现都是直接 `return obj`
- 新 `Virtual DOM` 和旧 `Virtual DOM` 之间需要做一个 `patch`，找出 `diff`。`patch` 完了之后的 `diff` 怎么更新视图，也就是如何给这些 `DOM` 加入 `attr`、`class`、`style` 等 `DOM` 属性呢？`Vue` 中有 `nextTick` 的概念用以更新视图，`mpvue` 这块对于小程序的 `setData` 应该怎么处理呢？
- 另外个问题在于小程序的 `Virtual DOM` 怎么生成？也就是怎么将 `template` 编译成 `render function`。这当中还涉及到运行时编译器-`vs`-只包含运行时，显然如果要提高性能、减少包大小、输出 `wxml`、`mpvue` 也要提供预编译的能力。因为要预输出 `wxml` 且没法动态改变 `DOM`，所以动态组件，自定义 `render`，和 `<script type="text/x-template">` 字符串模版等都不支持

另外还有一些其他问题，最后总结一下

- 1. 如何预编译生成 `render function`

- 2.如何预编译生成 `wxml`, `wxss`, `wxs`
- 3.如何 `patch` 出 `diff`
- 4.如何更新视图
- 5.如何建立小程序事件代理机制，在事件代理函数中触发与之对应的 vue 组件事件响应
- 6.如何建立 vue 实例与小程序 Page 实例关联
- 7.如何建立小程序和 vue 生命周期映射关系，能在小程序生命周期中触发 vue 生命周期

`platform/mp` 的目录结构

```
.
├── compiler //解决问题1, mpvue-template-compiler源码部分
├── runtime //解决问题3 4 5 6 7
├── util //工具方法
├── entry-compiler.js //mpvue-template-compiler的入口。package.json相关命令会自动生成
  mpvue-template-compiler这个package。
├── entry-runtime.js //对外提供Vue对象，当然是mpvue
└── join-code-in-build.js //编译出SDK时的修复
```

mpvue-loader

`mpvue-loader` 是 `vue-loader` 的一个扩展延伸版，类似于超集的关系，除了`vue-loader`本身所具备的能力之外，它还会利用`mpvue-template-compiler`生成`render function``

entry

- 它会从 `webpack` 的配置中的 `entry` 开始，分析依赖模块，并分别打包。在 `entry` 中 `app` 属性及其内容会被打包为微信小程序所需要的 `app.js / app.json / app.wxss`，其余的会生成对应的
- 页面 `page.js / page.json / page.wxml / page.wxss`，如示例的 `entry` 将会生成如下这些文件，文件内容下文慢慢讲来：

```
// webpack.config.js
{
  // ...
  entry: {
    app: resolve('./src/main.js'),           // app 字段被识别为 app 类型
    index: resolve('./src/pages/index/main.js'), // 其余字段被识别为 page 类型
    'news/home': resolve('./src/pages/news/home/index.js')
  }
}

// 产出文件的结构

.
├── app.js
├── app.json
├── app.wxss
├── components
|   ├── card$74bfae61.wxml
|   ├── index$023eef02.wxml
|   └── news$0699930b.wxml
└── news
    ├── home.js
    ├── home.wxml
    └── home.wxss
└── pages
    └── index
        └── index.js
```

```

|   └── index.wxml
|       └── index.wxss
└── static
    ├── css
    |   ├── app.wxss
    |   ├── index.wxss
    |   └── news
    |       └── home.wxss
    └── js
        ├── app.js
        ├── index.js
        ├── manifest.js
        ├── news
        |   └── home.js
        └── vendor.js

```

wxml 每一个 `.vue` 的组件都会被生成成为一个 `wxml` 规范的 `template`，然后通过 `wxml` 规范的 `import` 语法来达到一个复用，同时组件如果涉及到 `props` 的 `data` 数据，我们也会做相应的处理，举个实际的例子：

```

<template>
  <div class="my-component" @click="test">
    <h1>{{msg}}</h1>
    <other-component :msg="msg"></other-component>
  </div>
</template>
<script>
  import otherComponent from './otherComponent.vue'

  export default {
    components: { otherComponent },
    data () {
      return { msg: 'Hello Vue.js!' }
    },
    methods: {
      test() {}
    }
  }
</script>

```

这样一个 `Vue` 的组件的模版部分会生成相应的 `wxml`

```

<import src="components/other-component$hash.wxml" />
<template name="component$hash">
  <view class="my-component" bindtap="handleProxy">
    <view class="_h1">{{msg}}</view>
    <template is="other-component$hash" wx:if="{{ $c[0] }}" data="{{ ...$c[0] }}"></template>
  </view>
</template>

```

可能已经注意到了 `other-component(:msg="msg")` 被转化成了。`mpvue` 在运行时会从根组件开始把所有的组件实例数据合并成一个树形的数据，然后通过 `setData` 到 `appData`，`$c` 是 `$children` 的缩写。至于那个 `[0]` 则是我们的 `compiler` 处理过后的一个标记，会为每一个子组件打一个特定的不重复的标记。树形数据结构如下

```
// 这儿数据结构是一个数组，index 是动态的
{
  $child: {
    '0'{
      // ... root data
      $child: {
        '0': {
          // ... data
          msg: 'Hello vue.js!',
          $child: {
            // ...data
          }
        }
      }
    }
  }
}
```

WXSS

这个部分的处理同 web 的处理差异不大，唯一不同在于通过配置生成 .css 为 .wxss，其中的对于 css 的若干处理，在 postcss-mpvue-wxss 和 px2rpx-loader 这两部分的文档中又详细的介绍

- 推荐和小程序一样，将 app.json / page.json 放到页面入口处，使用 copy-webpack-plugin copy 到对应的生成位置。

这部分内容来源于 app 和 page 的 entry 文件，通常习惯是 main.js，你需要在你的入口文件中 export default { config: {} }，这才能被我们的 loader 识别为这是一个配置，需要写成 json 文件

```
import Vue from 'vue';
import App from './app';

const vueApp = new Vue(App);
vueApp.$mount();

// 这个是我们约定的额外的配置
export default {
  // 这个字段下的数据会被填充到 app.json / page.json
  config: {
    pages: ['static/calendar/calendar', '^pages/list/list'], // will be
filled in webpack
    window: {
      backgroundTextStyle: 'light',
      navigationBarBackgroundColor: '#455A73',
      navigationBarTitleText: '美团汽车票',
      navigationBarTextStyle: 'fff'
    }
  };
};
```

#六、React

#1、React 中 keys 的作用是什么？

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识

- 在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。在 React Diff 算法中 React 会借助元素的 key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染。此外，React 还需要借助 key 值来判断元素与本地状态的关联关系，因此我们绝不可忽视转换函数中 key 的重要性

#2、传入 setState 函数的第二个参数的作用是什么？

该函数会在 setState 函数调用完成并且组件开始重渲染的时候被调用，我们可以用该函数来监听渲染是否完成：

```
this.setState(  
  { username: 'tylermcginnis33' },  
  () => console.log('setState has finished and the component has re-rendered.')  
)  
this.setState((prevState, props) => {  
  return {  
    streak: prevState.streak + props.count  
  }  
})
```

#3、React 中 refs 的作用是什么

- Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄
- 可以为元素添加 ref 属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回

#4、在生命周期中的哪一步你应该发起 AJAX 请求

我们应当将AJAX 请求放到 componentDidMount 函数中执行，主要原因有下

- React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能，其会影响到 componentWillMount 的触发次数。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定，React 可能会多次频繁调用 componentWillMount。如果我们将 AJAX 请求放到 componentWillMount 函数中，那么显而易见其会被触发多次，自然也就不是好的选择。
- 如果我们将 AJAX 请求放置在生命周期的其他函数中，我们并不能保证请求仅在组件挂载完毕后才会要求响应。如果我们的数据请求在组件挂载之前就完成，并且调用了 setState 函数将数据添加到组件状态中，对于未挂载的组件则会报错。而在 componentDidMount 函数中进行 AJAX 请求则能有效避免这个问题

#5、shouldComponentUpdate 的作用

shouldComponentUpdate 允许我们手动地判断是否要进行组件更新，根据组件的应用场景设置函数的合理返回值能够帮我们避免不必要的更新

#6、如何告诉 React 它应该编译生产环境版

通常情况下我们会使用 Webpack 的 DefinePlugin 方法来将 NODE_ENV 变量值设置为 production。编译版本中 React 会忽略 propTypes 验证以及其他告警信息，同时还会降低代码库的大小，React 使用了 uglify 插件来移除生产环境下不必要的注释等信息

#7、概述下 React 中的事件处理逻辑

为了解决跨浏览器兼容性问题，`React` 会将浏览器原生事件（`Browser Native Event`）封装为合成事件（`SyntheticEvent`）传入设置的事件处理器中。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性。另外有意思的是，`React` 并没有直接将事件附着到子元素上，而是以单一事件监听器的方式将所有的事件发送到顶层进行处理。这样 `React` 在更新 `DOM` 的时候就不需要考虑如何去处理附着在 `DOM` 上的事件监听器，最终达到优化性能的目的

#8、createElement 与 cloneElement 的区别是什么

`createElement` 函数是 `JSX` 编译之后使用的创建 `React Element` 的函数，而 `cloneElement` 则是用于复制某个元素并传入新的 `Props`

#9、redux中间件

中间件提供第三方插件的模式，自定义拦截 `action` -> `reducer` 的过程。变为 `action` -> `middlewares` -> `reducer`。这种机制可以让我们改变数据流，实现如异步 `action`，`action` 过滤，日志输出，异常报告等功能

- `redux-logger`：提供日志输出
- `redux-thunk`：处理异步操作
- `redux-promise`：处理异步操作，`actionCreator` 的返回值是 `promise`

#10、redux有什么缺点

- 一个组件所需要的数据，必须由父组件传过来，而不能像 `flux` 中直接从 `store` 取。
- 当一个组件相关数据更新时，即使父组件不需要用到这个组件，父组件还是会重新 `render`，可能会有效率影响，或者需要写复杂的 `shouldComponentUpdate` 进行判断。

#11、react组件的划分业务组件技术组件？

- 根据组件的职责通常把组件分为UI组件和容器组件。
- UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。
- 两者通过 `React-Redux` 提供 `connect` 方法联系起来

#12、react旧版生命周期函数

初始化阶段

- `get defaultProps`：获取实例的默认属性
- `getInitialState`：获取每个实例的初始化状态
- `componentWillMount`：组件即将被装载、渲染到页面上
- `render`：组件在这里生成虚拟的 `DOM` 节点
- `componentDidMount`：组件真正在被装载之后

运行中状态

- `componentWillReceiveProps`：组件将要接收到属性的时候调用
- `shouldComponentUpdate`：组件接受到新属性或者新状态的时候（可以返回`false`，接收数据后不更新，阻止 `render` 调用，后面的函数不会被继续执行了）
- `componentWillUpdate`：组件即将更新不能修改属性和状态
- `render`：组件重新描绘
- `componentDidUpdate`：组件已经更新

销毁阶段

- `componentWillUnmount`: 组件即将销毁

#新版生命周期

在新版本中，React 官方对生命周期有了新的变动建议：

- 使用 `getDerivedStateFromProps` 替换 `componentWillMount`；
- 使用 `getSnapshotBeforeUpdate` 替换 `componentWillUpdate`；
- 避免使用 `componentWillReceiveProps`；

其实该变动的原因，正是由于上述提到的 `Fiber`。首先，从上面我们知道 React 可以分成 `reconciliation` 与 `commit` 两个阶段，对应的生命周期如下：

reconciliation

- `componentWillMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`

commit

- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`

在 `Fiber` 中，`reconciliation` 阶段进行了任务分割，涉及到暂停和重启，因此可能会导致 `reconciliation` 中的生命周期函数在一次更新渲染循环中被多次调用的情况，产生一些意外错误

新版的建议生命周期如下：

```
class Component extends React.Component {  
  // 替换 `componentWillReceiveProps`，  
  // 初始化和 update 时被调用  
  // 静态函数，无法使用 this  
  static getDerivedStateFromProps(nextProps, prevState) {}  
  
  // 判断是否需要更新组件  
  // 可以用于组件性能优化  
  shouldComponentUpdate(nextProps, nextState) {}  
  
  // 组件被挂载后触发  
  componentDidMount() {}  
  
  // 替换 componentWillMount  
  // 可以在更新之前获取最新 dom 数据  
  getSnapshotBeforeUpdate() {}  
  
  // 组件更新后调用  
  componentDidUpdate() {}  
  
  // 组件即将销毁  
  componentWillUnmount() {}  
  
  // 组件已销毁
```

```
componentDidUnmount() {}  
}
```

使用建议:

- 在 `constructor` 初始化 `state`;
- 在 `componentDidMount` 中进行事件监听，并在 `componentWillUnmount` 中解绑事件；
- 在 `componentDidMount` 中进行数据的请求，而不是在 `componentWillMount`；
- 需要根据

props

更新

state

时，使用

```
getDerivedStateFromProps(nextProps, prevState)
```

;

- 旧 props 需要自己存储，以便比较；

```
public static getDerivedStateFromProps(nextProps, prevState) {  
    // 当新 props 中的 data 发生变化时，同步更新到 state 上  
    if (nextProps.data !== prevState.data) {  
        return {  
            data: nextProps.data  
        }  
    } else {  
        return null  
    }  
}
```

可以在 `componentDidUpdate` 监听 `props` 或者 `state` 的变化，例如：

```
componentDidUpdate(prevProps) {  
    // 当 id 发生变化时，重新获取数据  
    if (this.props.id !== prevProps.id) {  
        this.fetchData(this.props.id);  
    }  
}
```

- 在 `componentDidUpdate` 使用 `setState` 时，必须加条件，否则将进入死循环；
- `getSnapshotBeforeUpdate(prevProps, prevState)` 可以在更新之前获取最新的渲染数据，它的调用是在 `render` 之后，`update` 之前；
- `shouldComponentUpdate`: 默认每次调用 `setState`，一定会最终走到 `diff` 阶段，但可以通过 `shouldComponentUpdate` 的生命钩子返回 `false` 来直接阻止后面的逻辑执行，通常是用于做条件渲染，优化渲染的性能。

#13、react性能优化是哪个周期函数

`shouldComponentUpdate` 这个方法用来判断是否需要调用`render`方法重新描绘dom。因为dom的描绘非常消耗性能，如果我们能在`shouldComponentUpdate`方法中能够写出更优化的`dom diff`算法，可以极大的提高性能

#14、为什么虚拟dom会提高性能

虚拟`dom`相当于在`js`和真实`dom`中间加了一个缓存，利用`dom diff`算法避免了没有必要的`dom`操作，从而提高性能

具体实现步骤如下

- 用`JavaScript`对象结构表示`DOM`树的结构；然后用这个树构建一个真正的`DOM`树，插到文档当中
- 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异
- 把2所记录的差异应用到步骤1所构建的真正的`DOM`树上，视图就更新

#15、diff算法？

- 把树形结构按照层级分解，只比较同级元素。
- 给列表结构的每个单元添加唯一的`key`属性，方便比较。
- `React` 只会匹配相同`class`的`component`（这里面的`class`指的是组件的名字）
- 合并操作，调用`component`的`setState`方法的时候，`React`将其标记为`-dirty`。到每一个事件循环结束，`React`检查所有标记`dirty`的`component`重新绘制。
- 选择性子树渲染。开发人员可以重写`shouldComponentUpdate`提高`diff`的性能

#16、react性能优化方案

- 重写`shouldComponentUpdate`来避免不必要的`dom`操作
- 使用`production`版本的`react.js`
- 使用`key`来帮助`React`识别列表中所有子组件的最小变化

#16、简述flux思想

`Flux` 的最大特点，就是数据的“单向流动”。

- 用户访问`view`
- `view`发出用户的`Action`
- `Dispatcher`收到`Action`，要求`store`进行相应的更新
- `store`更新后，发出一个“change”事件
- `view`收到“change”事件后，更新页面

#17、说说你用react有什么坑点？

1. JSX做表达式判断时候，需要强转为boolean类型

如果不使用`!!b`进行强转数据类型，会在页面里面输出`0`。

```
render() {
  const b = 0;
  return <div>
  {
    !!b && <div>这是一段文本</div>
  }
</div>
}
```

2. 尽量不要在 `componentWillReceiveProps` 里使用 `setState`, 如果一定要使用, 那么需要判断结束条件, 不然会出现无限重渲染, 导致页面崩溃

3. 给组件添加ref时候, 尽量不要使用匿名函数, 因为当组件更新的时候, 匿名函数会被当做新的prop处理, 让ref属性接受到新函数的时候, react内部会先清空ref, 也就是会以null为回调参数先执行一次ref这个props, 然后在以该组件的实例执行一次ref, 所以用匿名函数做ref的时候, 有的时候去ref赋值后的属性会取到null

4. 遍历子节点的时候, 不要用 index 作为组件的 key 进行传入

#18、我现在有一个button, 要用react在上面绑定点击事件, 要怎么做?

```
class Demo {
  render() {
    return <button onClick={() => {
      alert('我点击了按钮')
    }}>
      按钮
    </button>
  }
}
```

你觉得你这样设置点击事件会有什么问题吗?

由于 `onClick` 使用的是匿名函数, 所有每次重渲染的时候, 会把该 `onClick` 当做一个新的 prop 来处理, 会将内部缓存的 `onClick` 事件进行重新赋值, 所以相对直接使用函数来说, 可能有一点的性能下降

修改

```
class Demo {
  onClick = (e) => {
    alert('我点击了按钮')
  }

  render() {
    return <button onClick={this.onClick}>
      按钮
    </button>
  }
}
```

#19、react 的虚拟dom是怎么实现的

首先说说为什么要使用 Virtual DOM，因为操作真实 DOM 的耗费的性能代价太高，所以 react 内部使用 js 实现了一套 dom 结构，在每次操作在和真实 dom 之前，使用实现好的 diff 算法，对虚拟 dom 进行比较，递归找出有变化的 dom 节点，然后对其进行更新操作。为了实现虚拟 DOM，我们需要把每一种节点类型抽象成对象，每一种节点类型有自己的属性，也就是 prop，每次进行 diff 的时候，react 会先比较该节点类型，假如节点类型不一样，那么 react 会直接删除该节点，然后直接创建新的节点插入到其中，假如节点类型一样，那么会比较 prop 是否有更新，假如有 prop 不一样，那么 react 会判定该节点有更新，那么重渲染该节点，然后在对其子节点进行比较，一层一层往下，直到没有子节点

#20、react 的渲染过程中，兄弟节点之间是怎么处理的？也就是 key值不一样的时候

通常我们输出节点的时候都是 map 一个数组然后返回一个 `ReactNode`，为了方便 react 内部进行优化，我们必须给每一个 `reactNode` 添加 `key`，这个 `key prop` 在设计值处不是给开发者用的，而是给 react 用的，大概的作用就是给每一个 `reactNode` 添加一个身份标识，方便 react 进行识别，在重渲染过程中，如果 `key` 一样，若组件属性有所变化，则 react 只更新组件对应的属性；没有变化则不更新，如果 `key` 不一样，则 react 先销毁该组件，然后重新创建该组件

#21、介绍一下react

1. 以前我们没有 jquery 的时候，我们大概的流程是从后端通过 ajax 获取到数据然后使用 jquery 生成 dom 结果然后更新到页面当中，但是随着业务发展，我们的项目可能会越来越复杂，我们每次请求到数据，或则数据有更改的时候，我们又需要重新组装一次 dom 结构，然后更新页面，这样我们手动同步 dom 和数据的成本就越来越高，而且频繁的操作 dom，也使我们页面的性能慢慢的降低。
2. 这个时候 mvvm 出现了，mvvm 的双向数据绑定可以让我们在数据修改的同时同步 dom 的更新，dom 的更新也可以直接同步我们数据的更改，这个特定可以大大降低我们手动去维护 dom 更新的成本，mvvm 为 react 的特性之一，虽然 react 属于单项数据流，需要我们手动实现双向数据绑定。
3. 有了 mvvm 还不够，因为如果每次有数据做了更改，然后我们都全量更新 dom 结构的话，也没办法解决我们频繁操作 dom 结构(降低了页面性能)的问题，为了解决这个问题，react 内部实现了一套虚拟 dom 结构，也就是用 js 实现的一套 dom 结构，他的作用是讲真实 dom 在 js 中做一套缓存，每次有数据更改的时候，react 内部先使用算法，也就是鼎鼎有名的 diff 算法对 dom 结构进行对比，找到那些我们需要新增、更新、删除的 dom 节点，然后一次性对真实 DOM 进行更新，这样就大大降低了操作 dom 的次数。那么 diff 算法是怎么运作的呢，首先，diff 针对类型不同的节点，会直接判定原来节点需要卸载并且用新的节点来装载卸载的节点的位置；针对于节点类型相同的节点，会对比这个节点的所有属性，如果节点的所有属性相同，那么判定这个节点不需要更新，如果节点属性不相同，那么会判定这个节点需要更新，react 会更新并重渲染这个节点。
4. react 设计之初是主要负责 UI 层的渲染，虽然每个组件有自己的 state，state 表示组件的状态，当状态需要变化的时候，需要使用 `setState` 更新我们的组件，但是，我们想通过一个组件重渲染它的兄弟组件，我们就需要将组件的状态提升到父组件当中，让父组件的状态来控制这两个组件的重渲染，当我们组件的层次越来越深的时候，状态需要一直往下传，无疑加大了我们代码的复杂度，我们需要一个状态管理中心，来帮我们管理我们状态 state。
5. 这个时候，redux 出现了，我们可以将所有的 state 交给 redux 去管理，当我们的某一个 state 有变化的时候，依赖到这个 state 的组件就会进行一次重渲染，这样就解决了我们的我们需要一直把 state 往下传的问题。redux 有 action、reducer 的概念，action 为唯一修改 state 的来源，reducer 为唯一确定 state 如何变化的入口，这使得 redux 的数据流非常规范，同时也暴露出了 redux 代码的复杂，本来那么简单的功能，却需要完成那么多的代码。
6. 后来，社区就出现了另外一套解决方案，也就是 mobx，它推崇代码简约易懂，只需要定义一个可观测的对象，然后哪个组件使用到这个可观测的对象，并且这个对象的数据有更改，那么这个组件就会重渲染，而且 mobx 内部也做好了是否重渲染组件的生命周期 `shouldUpdateComponent`，不建议开发者进行更改，这使得我们使用 mobx 开发项目的时候可以简单快速的完成很多功能，连

redux的作者也推荐使用mobx进行项目开发。但是，随着项目的不断变大，mobx也不断暴露出了它的缺点，就是数据流太随意，出了bug之后不好追溯数据的流向，这个缺点正好体现出了redux的优点所在，所以针对于小项目来说，社区推荐使用mobx，对大项目推荐使用redux

#22、React怎么做数据的检查和变化

Model`改变之后（可能是调用了``setState``），触发了``virtual dom``的更新，再用``diff``算法来`把`virtual DOM``比较``real DOM``，看看是哪个``dom``节点更新了，再渲染``real dom`

#23、react-router里的`<Link>`标签和`<a>`标签有什么区别

对比`<a>`,`Link`组件避免了不必要的重渲染

#24、connect原理

- 首先 connect之所以会成功，是因为 Provider 组件：
- 在原应用组件上包裹一层，使原来整个应用成为 Provider 的子组件 接收 Redux 的 store 作为 props，通过 context 对象传递给子孙组件上的 connect

`connect`做了些什么。它真正连接`Redux` 和 `React`，它包在我们的容器组件的外一层，它接收上面`Provider`提供的`store`里面的`state`和`dispatch`，传给一个构造函数，返回一个对象，以属性形式传给我们的容器组件

- `connect`是一个高阶函数，首先传入`mapStateToProps`、`mapDispatchToProps`，然后返回一个生产`Component`的函数(`wrapWithConnect`)，然后再将真正的`Component`作为参数传入`wrapWithConnect`，这样就生产出一个经过包裹的`connect`组件，

该组件具有如下特点

- 通过`props.store`获取祖先`Component`的`store props`包括`stateProps`、`dispatchProps`、`parentProps`合并在一起得到`nextState`，作为`props`传给真正的`Component`
`componentDidMount`时，添加事件`this.store.subscribe(this.handleChange)`，实现页面交互
- `shouldComponentUpdate`时判断是否有避免进行渲染，提升页面性能，并得到`nextState`
`componentWillUnmount`时移除注册的事件`this.handleChange`

由于`connect`的源码过长，我们只看主要逻辑

```
export default function connect(mapStateToProps, mapDispatchToProps, mergeProps,
options = {}) {
  return function wrapWithConnect(wrappedComponent) {
    class Connect extends Component {
      constructor(props, context) {
        // 从祖先Component处获得store
        this.store = props.store || context.store
        this.stateProps = computeStateProps(this.store, props)
        this.dispatchProps = computeDispatchProps(this.store, props)
        this.state = { storeState: null }
        // 对stateProps、dispatchProps、parentProps进行合并
        this.updateState()
      }
      shouldComponentUpdate(nextProps, nextState) {
        // 进行判断，当数据发生改变时，Component重新渲染
        if (propsChanged || mapStateProducedChange || dispatchPropsChanged) {
          this.updateState(nextProps)
        }
      }
    }
    return (props) =>
      <div>
        <Connect {...props} />
        <wrappedComponent {...props} />
      </div>
    )
  }
}
```

```

        return true
    }
}

componentDidMount() {
    // 改变Component的state
    this.store.subscribe(() => {
        this.setState({
            storeState: this.store.getState()
        })
    })
}

render() {
    // 生成包裹组件Connect
    return (
        <WrappedComponent {...this.nextState} />
    )
}
}

Connect.contextTypes = {
    store: storeShape
}

return Connect;
}

}

```

#25、Redux实现原理解析

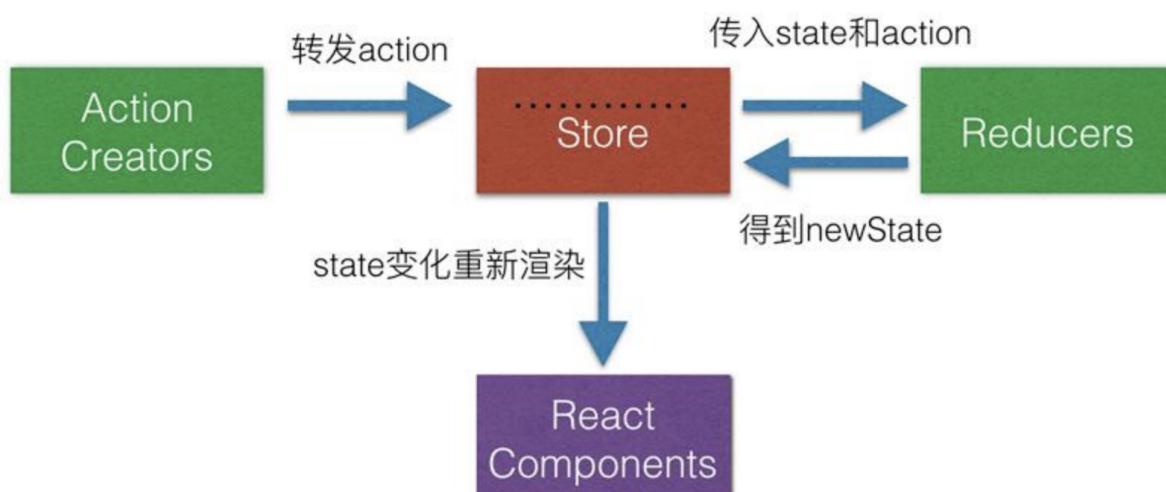
为什么要用redux

在React中，数据在组件中是单向流动的，数据从一个方向父组件流向子组件（通过props），所以，两个非父子组件之间通信就相对麻烦，redux的出现就是为了解决state里面的数据问题

Redux设计理念

Redux是将整个应用状态存储到一个地方上称为store，里面保存着一个状态树store tree，组件可以派发(dispatch)行为(action)给store，而不是直接通知其他组件，组件内部通过订阅store中的状态state来刷新自己的视图

Redux工作流



Redux三大原则

- 唯一数据源

整个应用的state都被存储到一个状态树里面，并且这个状态树，只存在于唯一的store中

- 保持只读状态

`state`是只读的，唯一改变`state`的方法就是触发`action`，`action`是一个用于描述以发生时间的普通对象

- 数据改变只能通过纯函数来执行

使用纯函数来执行修改，为了描述`action`如何改变`state`的，你需要编写`reducers`

Redux源码

```
let createStore = (reducer) => {
  let state;
  //获取状态对象
  //存放所有的监听函数
  let listeners = [];
  let getState = () => state;
  //提供一个方法供外部调用派发action
  let dispatch = (action) => {
    //调用管理员reducer得到新的state
    state = reducer(state, action);
    //执行所有的监听函数
    listeners.forEach(l => l());
  }
  //订阅状态变化事件，当状态改变发生之后执行监听函数
  let subscribe = (listener) => {
    listeners.push(listener);
  }
  dispatch();
  return {
    getState,
    dispatch,
    subscribe
  }
}
let combineReducers=(reducers)=>{
  //传入一个reducers管理组，返回的是一个reducer
  return function(state={},action=[]){
    let newState={};
    for(var attr in reducers){
      newState[attr]=reducers[attr](state[attr],action)

    }
    return newState;
  }
}
export {createStore,combineReducers};
```

#26、pureComponent和FunctionComponent区别

`PureComponent`和`Component`完全相同，但是在`shouldComponentUpdate`实现中，

`PureComponent`使用了`props`和`state`的浅比较。主要作用是用来提高某些特定场景的性能

#27 react hooks, 它带来了那些便利

- 代码逻辑聚合，逻辑复用
- HOC嵌套地狱
- 代替class

React 中通常使用类定义或者函数定义创建组件:

在类定义中，我们可以使用到许多 React 特性，例如 state、各种组件生命周期钩子等，但是在函数定义中，我们却无能为力，因此 React 16.8 版本推出了一个新功能 (React Hooks)，通过它，可以更好的在函数定义组件中使用 React 特性。

好处:

1. 跨组件复用: 其实 render props / HOC 也是为了复用，相比于它们，Hooks 作为官方的底层 API，最为轻量，而且改造成本小，不会影响原来的组件层次结构和传说中的嵌套地狱；
2. 类定义更为复杂
 - 不同的生命周期会使逻辑变得分散且混乱，不易维护和管理；
 - 时刻需要关注this的指向问题；
 - 代码复用代价高，高阶组件的使用经常会使整个组件树变得臃肿；
1. 状态与UI隔离: 正是由于 Hooks 的特性，状态逻辑会变成更小的粒度，并且极容易被抽象成一个自定义 Hooks，组件中的状态和 UI 变得更为清晰和隔离。

注意:

- 避免在循环/条件判断/嵌套函数 中调用 hooks，保证调用顺序的稳定；
- 只有函数定义组件 和 hooks 可以调用 hooks，避免在类组件 或者 普通函数 中调用；
- 不能在useEffect中使用useState，React 会报错提示；
- 类组件不会被替换或废弃，不需要强制改造类组件，两种方式能并存；

重要钩子

1. 状态钩子 (useState): 用于定义组件的 State，其到类定义中this.state的功能；

```
// useState 只接受一个参数：初始状态
// 返回的是组件名和更改该组件对应的函数
const [flag, setFlag] = useState(true);
// 修改状态
setFlag(false)

// 上面的代码映射到类定义中：
this.state = {
  flag: true
}
const flag = this.state.flag
const setFlag = (bool) => {
  this.setState({
    flag: bool,
  })
}
```

1. 生命周期钩子 (useEffect):

类定义中有许多生命周期函数，而在 React Hooks 中也提供了一个相应的函数 (useEffect)，这里可以看做componentDidMount、componentDidUpdate和componentWillUnmount的结合。

useEffect(callback, [source])接受两个参数

- callback: 钩子回调函数;
- source: 设置触发条件, 仅当 source 发生改变时才会触发;
- useEffect钩子在没有传入[source]参数时, 默认在每次 render 时都会优先调用上次保存的回调中返回的函数, 后再重新调用回调;

```
useEffect(() => {
    // 组件挂载后执行事件绑定
    console.log('on')
    addEventListener()

    // 组件 update 时会执行事件解绑
    return () => {
        console.log('off')
        removeEventListener()
    }
}, [source]);

// 每次 source 发生改变时, 执行结果(以类定义的生命周期, 便于大家理解):
// --- DidMount ---
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- Didupdate ---
// 'off'
// 'on'
// --- willUnmount ---
// 'off'
```

通过第二个参数, 我们便可模拟出几个常用的生命周期:

- componentDidMount: 传入[]时, 就只会在初始化时调用一次

```
const useMount = (fn) => useEffect(fn, [])
```

- componentWillUnmount: 传入[], 回调中的返回的函数也只会被最终执行一次

```
const useUnmount = (fn) => useEffect(() => fn, [])
```

- mounted: 可以使用 useState 封装成一个高度可复用的 mounted 状态;

```
const useMounted = () => {
    const [mounted, setMounted] = useState(false);
    useEffect(() => {
        !mounted && setMounted(true);
        return () => setMounted(false);
    }, []);
    return mounted;
}
```

- componentDidUpdate: useEffect每次均会执行, 其实就是排除了 DidMount 后即可;

```
const mounted = useMounted()
useEffect(() => {
  mounted && fn()
})
```

1. 其它内置钩子:

- `useContext`: 获取 context 对象

- `useReducer`

: 类似于 Redux 思想的实现，但其并不足以替代 Redux，可以理解成一个组件内部的 redux:

- 并不是持久化存储，会随着组件被销毁而销毁；
- 属于组件内部，各个组件是相互隔离的，单纯用它并无法共享数据；
- 配合`useContext``的全局性，可以完成一个轻量级的 Redux；(easy-peasy)
- `useCallback`: 缓存回调函数，避免传入的回调每次都是新的函数实例而导致依赖组件重新渲染，具有性能优化的效果；
- `useMemo`: 用于缓存传入的 props，避免依赖的组件每次都重新渲染；
- `useRef`: 获得组件的真实节点；

- `useLayoutEffect`

- DOM更新同步钩子。用法与`useEffect`类似，只是区别于执行时间点的不同
- `useEffect`属于异步执行，并不会等待 DOM 真正渲染后执行，而`useLayoutEffect`则会真正渲染后才触发；
- 可以获取更新后的 state；

1. 自定义钩子(useXXXX): 基于 Hooks 可以引用其它 Hooks 这个特性，我们可以编写自定义钩子，如上面的`useMounted`。又例如，我们需要每个页面自定义标题：

```
function useTitle(title) {
  useEffect(
    () => {
      document.title = title;
    }
  )

  // 使用：
  function Home() {
    const title = '我是首页'
    useTitle(title)

    return (
      <div>{title}</div>
    )
  }
}
```

#28、React Portal 有哪些使用场景

- 在以前，react 中所有的组件都会位于 #app 下，而使用 Portals 提供了一种脱离 #app 的组件
- 因此 Portals 适合脱离文档流(out of flow) 的组件，特别是 position: absolute 与 position: fixed 的组件。比如模态框，通知，警告，goTop 等。

以下是官方一个模态框的示例，可以在以下地址中测试效果

```

<html>
  <body>
    <div id="app"></div>
    <div id="modal"></div>
    <div id="gotop"></div>
    <div id="alert"></div>
  </body>
</html>
const modalRoot = document.getElementById('modal');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el,
    );
  }
}

```

React Hooks当中的useEffect是如何区分生命周期钩子的

useEffect可以看成是 componentDidMount， componentDidUpdate 和 componentWillUnmount 三者的结合。useEffect(callback, [source])接收两个参数，调用方式如下

```

useEffect(() => {
  console.log('mounted');

  return () => {
    console.log('willUnmount');
  }
}, [source]);

```

生命周期函数的调用主要是通过第二个参数[source]来进行控制，有如下几种情况：

- [source] 参数不传时，则每次都会优先调用上次保存的函数中返回的那个函数，然后再调用外部那个函数；
- [source] 参数传[]时，则外部的函数只会在初始化时调用一次，返回的那个函数也只会最终在组件卸载时调用一次；
- [source] 参数有值时，则只会监听到数组中的值发生变化后才优先调用返回的那个函数，再调用外部的函数。

#29、react和vue的区别

相同点：

1. 数据驱动页面，提供响应式的试图组件
2. 都有virtual DOM,组件化的开发，通过props参数进行父子之间组件传递数据，都实现了webComponents规范
3. 数据流动单向，都支持服务器的渲染SSR
4. 都有支持native的方法，react有React native， vue有wexx

不同点：

1. 数据绑定：Vue实现了双向的数据绑定，react数据流动是单向的
2. 数据渲染：大规模的数据渲染，react更快
3. 使用场景：React配合Redux架构适合大规模多人协作复杂项目，Vue适合小快的项目
4. 开发风格：react推荐做法js + inline style把html和css都写在js了

vue是采用webpack +vue-loader单文件组件格式，html, js, css同一个文件

#30、什么是高阶组件(HOC)

- 高阶组件(Higher Order Component)本身其实不是组件，而是一个函数，这个函数接收一个元组件作为参数，然后返回一个新的增强组件，高阶组件的出现本身也是为了逻辑复用，举个例子

```
function withLoginAuth(WrappedComponent) {
  return class extends React.Component {

    constructor(props) {
      super(props);
      this.state = {
        isLogin: false
      };
    }

    async componentDidMount() {
      const isLogin = await getLoginStatus();
      this.setState({ isLogin });
    }

    render() {
      if (this.state.isLogin) {
        return <WrappedComponent {...this.props} />;
      }

      return (<div>您还未登录...</div>);
    }
  }
}
```

#31、React实现的移动应用中，如果出现卡顿，有哪些可以考虑的优化方案

- 增加 `shouldComponentUpdate` 钩子对新旧 `props` 进行比较，如果值相同则阻止更新，避免不必要的渲染，或者使用 `PureReactComponent` 替代 `Component`，其内部已经封装了 `shouldComponentUpdate` 的浅比较逻辑
- 对于列表或其他结构相同的节点，为其中的每一项增加唯一 `key` 属性，以方便 React 的 `diff` 算法中对该节点的复用，减少节点的创建和删除操作
- `render` 函数中减少类似 `onClick={() => {doSomething()}}` 的写法，每次调用 `render` 函数时均会创建一个新的函数，即使内容没有发生任何变化，也会导致节点没必要的重渲染，建议将函数保存在组件的成员对象中，这样只会创建一次
- 组件的 `props` 如果需要经过一系列运算后才能拿到最终结果，则可以考虑使用 `reselect` 库对结果进行缓存，如果 `props` 值未发生变化，则结果直接从缓存中拿，避免高昂的运算代价
- `webpack-bundle-analyzer` 分析当前页面的依赖包，是否存在不合理性，如果存在，找到优化点并进行优化

#32、Fiber

React 的核心流程可以分为两个部分：

- reconciliation

(调度算法，也可称为

`render`

)

- 更新 `state` 与 `props`；
- 调用生命周期钩子；
- 生成

`virtual dom`

- 这里应该称为 `Fiber Tree` 更为符合；
- 通过新旧 vdom 进行 `diff` 算法，获取 vdom change
- 确定是否需要重新渲染

- commit

- 如需要，则操作 `dom` 节点更新

要了解 Fiber，我们首先来看为什么需要它

- **问题：**随着应用变得越来越庞大，整个更新渲染的过程开始变得吃力，大量的组件渲染会导致主线程长时间被占用，导致一些动画或高频操作出现卡顿和掉帧的情况。而关键点，便是 同步阻塞。在之前的调度算法中，React 需要实例化每个类组件，生成一颗组件树，使用 同步递归 的方式进行遍历渲染，而这个过程最大的问题就是无法暂停和恢复。
- **解决方案：**解决同步阻塞的方法，通常有两种：异步 与 任务分割。而 React Fiber 便是为了实现任务分割而诞生的
- 简述

- 在 React v16 将调度算法进行了重构，将之前的 stack reconciler 重构成新版的 fiber reconciler，变成了具有链表和指针的单链表树遍历算法。通过指针映射，每个单元都记录着遍历当下的上一步与下一步，从而使遍历变得可以被暂停和重启
- 这里我理解为是一种任务分割调度算法，主要是将原先同步更新渲染的任务分割成一个个独立的小任务单位，根据不同的优先级，将小任务分散到浏览器的空闲时间执行，充分利用主进程的事件循环机制
- 核心
 - Fiber 这里可以具象为一个数据结构

```
class Fiber {
  constructor(instance) {
    this.instance = instance
    // 指向第一个 child 节点
    this.child = child
    // 指向父节点
    this.parent = parent
    // 指向第一个兄弟节点
    this.sibling = previous
  }
}
```

- 链表树遍历算法
 - 通过节点保存与映射，便能够随时地进行停止和重启，这样便能达到实现任务分割的基本前提
 - 首先通过不断遍历子节点，到树末尾；
 - 开始通过 sibling 遍历兄弟节点；
 - return 返回父节点，继续执行2；
 - 直到 root 节点后，跳出遍历；
- 任务分割
 - React 中的渲染更新可以分成两个阶段
 - reconciliation 阶段**: vdom 的数据对比，是个适合拆分的阶段，比如对比一部分树后，先暂停执行个动画调用，待完成后再回来继续比对
 - Commit 阶段**: 将 change list 更新到 dom 上，并不适合拆分，才能保持数据与 UI 的同步。否则可能由于阻塞 UI 更新，而导致数据更新和 UI 不一致的情况
- 分散执行:

任务分割后，就可以把小任务单元分散到浏览器的空闲期间去排队执行，而实现的关键是两个新 API:

requestIdleCallback

与

requestAnimationFrame

- 低优先级的任务交给 requestIdleCallback 处理，这是个浏览器提供的事件循环空闲期的回调函数，需要 polyfill，而且拥有 deadline 参数，限制执行事件，以继续切分任务；
- 高优先级的任务交给 requestAnimationFrame 处理；

```
// 类似于这样的方式
requestIdleCallback((deadline) => {
  // 当有空闲时间时，我们执行一个组件渲染;
  // 把任务塞到一个个碎片时间中去;
  while ((deadline.timeRemaining() > 0 || deadline.didTimeout) &&
nextComponent) {
    nextComponent = performWork(nextComponent);
  }
});
```

- **优先级策略:** 文本框输入 > 本次调度结束需完成的任务 > 动画过渡 > 交互反馈 > 数据更新 > 不会显示但以防将来会显示的任务

- Fiber 其实可以算是一种编程思想，在其它语言中也有许多应用(Ruby Fiber)。
- 核心思想是任务拆分和协同，主动把执行权交给主线程，使主线程有时间空挡处理其他高优先级任务。
- 当遇到进程阻塞的问题时，任务分割、异步调用 和 缓存策略 是三个显著的解决思路。

#33、`setState`

在了解`setState`之前，我们先来简单了解下 React 一个包装结构: Transaction:

事务 (Transaction)

是 React 中的一个调用结构，用于包装一个方法，结构为: initialize - perform(method) - close。
通过事务，可以统一管理一个方法的开始与结束；处于事务流中，表示进程正在执行一些操作

- `setState`: React 中用于修改状态，更新视图。它具有以下特点:

异步与同步: `setState`并不是单纯的异步或同步，这其实与调用时的环境相关:

- 在

合成事件

和

生命周期钩子

(除 `componentDidUpdate`) 中，`setState`是"异步"的；

- 原因: 因为在`setState`的实现中，有一个判断: 当更新策略正在事务流的执行中时，该组件更新会被推入`dirtyComponents`队列中等待执行；否则，开始执行`batchedUpdates`队列更新；
 - 在生命周期钩子调用中，更新策略都处于更新之前，组件仍处于事务流中，而 `componentDidUpdate`是在更新之后，此时组件已经不在事务流中了，因此则会同步执行；
 - 在合成事件中，React 是基于 事务流完成的事件委托机制 实现，也是处于事务流中；
- 问题: 无法在`setState`后马上从`this.state`上获取更新后的值。
- 解决: 如果需要马上同步去获取新值，`setState`其实是可以传入第二个参数的。
`setState(updater, callback)`，在回调中即可获取最新值；

- 在

原生事件

和 `setTimeout` 中，`setState`是同步的，可以马上获取更新后的值；

- 原因: 原生事件是浏览器本身的实现，与事务流无关，自然是同步；而`setTimeout`是放置于定时器线程中延后执行，此时事务流已结束，因此也是同步；

- **批量更新:** 在 合成事件 和 生命周期钩子 中，`setState`更新队列时，存储的是 合并状态 (`Object.assign`)。因此前面设置的 key 值会被后面所覆盖，最终只会执行一次更新；

- 函数式

:由于 Fiber 及 合并 的问题，官方推荐可以传入 函数 的形式。`setState(fn)`，在fn中返回新的state 对象即可，例如`this.setState((state, props) => newState)`；

- 使用函数式，可以用于避免`setState`的批量更新的逻辑，传入的函数将会被 顺序调用；

注意事项:

- `setState` 合并，在 合成事件 和 生命周期钩子 中多次连续调用会被优化为一次；
- 当组件已被销毁，如果再次调用`setState`，React 会报错警告，通常有两种解决办法
 - 将数据挂载到外部，通过 `props` 传入，如放到 Redux 或 父级中；
 - 在组件内部维护一个状态量 (`isUnmounted`)，`componentWillUnmount`中标记为 `true`，在 `setState`前进行判断；

#34、HOC(高阶组件)

HOC(Higher Order Component) 是在 React 机制下社区形成的一种组件模式，在很多第三方开源库中表现强大。

简述:

- 高阶组件不是组件，是 增强函数，可以输入一个元组件，返回出一个新的增强组件；
- 高阶组件的主要作用是 代码复用，操作 状态和参数；

用法:

- 属性代理 (Props Proxy): 返回出一个组件，它基于被包裹组件进行 功能增强；
 1. 默认参数: 可以为组件包裹一层默认参数；

```
function proxyHoc(Comp) {
  return class extends React.Component {
    render() {
      const newProps = {
        name: 'tayde',
        age: 1,
      }
      return <Comp {...this.props} {...newProps} />
    }
  }
}
```

1. 提取状态: 可以通过 `props` 将被包裹组件中的 `state` 依赖外层，例如用于转换受控组件：

```
function withOnChange(Comp) {
  return class extends React.Component {
    constructor(props) {
      super(props)
      this.state = {
        name: '',
      }
    }
    onChangeName = () => {
      this.setState({
        name: 'dongdong',
      })
    }
  }
}
```

```

    render() {
      const newProps = {
        value: this.state.name,
        onChange: this.onChangeName,
      }
      return <Comp {...this.props} {...newProps} />
    }
}

```

使用姿势如下，这样就能非常快速的将一个 Input 组件转化成受控组件。

```

const NameInput = props => (<input name="name" {...props} />)
export default withOnChange(NameInput)

```

包裹组件: 可以为被包裹元素进行一层包装,

```

function withMask(Comp) {
  return class extends React.Component {
    render() {
      return (
        <div>
          <Comp {...this.props} />
          <div style={{{
            width: '100%',
            height: '100%',
            backgroundColor: 'rgba(0, 0, 0, .6)'
          }}}
        </div>
      )
    }
  }
}

```

反向继承 (Inheritance Inversion): 返回出一个组件，继承于被包裹组件，常用于以下操作

```

function IIHoc(Comp) {
  return class extends Comp {
    render() {
      return super.render();
    }
  };
}

```

渲染劫持 (Render Highjacking)

条件渲染: 根据条件，渲染不同的组件

```

function withLoading(Comp) {
  return class extends Comp {
    render() {
      if(this.props.isLoading) {
        return <Loading />
      } else {
        return super.render()
      }
    }
  };
}

```

可以直接修改被包裹组件渲染出的 React 元素树

操作状态 (Operate State): 可以直接通过 this.state 获取到被包裹组件的状态，并进行操作。但这样的操作容易使 state 变得难以追踪，不易维护，谨慎使用。

应用场景：

权限控制，通过抽象逻辑，统一对页面进行权限判断，按不同的条件进行页面渲染：

```

function withAdminAuth(WrappedComponent) {
  return class extends React.Component {
    constructor(props){
      super(props)
      this.state = {
        isAdmin: false,
      }
    }
    async componentWillMount() {
      const currentRole = await getCurrentUserRole();
      this.setState({
        isAdmin: currentRole === 'Admin',
      });
    }
    render() {
      if (this.state.isAdmin) {
        return <Comp {...this.props} />;
      } else {
        return (<div>您没有权限查看该页面，请联系管理员！</div>);
      }
    }
  };
}

```

性能监控，包裹组件的生命周期，进行统一埋点：

```

function withTiming(Comp) {
  return class extends Comp {
    constructor(props) {
      super(props);
      this.start = Date.now();
      this.end = 0;
    }
    componentDidMount() {
      super.componentDidMount && super.componentDidMount();
      this.end = Date.now();
    }
  };
}

```

```
        console.log(`\$ {WrappedComponent.name} 组件渲染时间为 \$ {this.end - this.start} ms`);  
    }  
    render() {  
        return super.render();  
    }  
};  
}
```

代码复用，可以将重复的逻辑进行抽象。

使用注意:

- 纯函数: 增强函数应为纯函数，避免侵入修改元组件；
- 避免用法污染: 理想状态下，应透传元组件的无关参数与事件，尽量保证用法不变；
- 命名空间: 为 HOC 增加特异性的组件名称，这样能便于开发调试和查找问题；
- 引用传递: 如果需要传递元组件的 refs 引用，可以使用React.forwardRef；
- 静态方法
 - 元组件上的静态方法并无法被自动传出，会导致业务层无法调用；解决：
 - 函数导出
 - 静态方法赋值
- 重新渲染: 由于增强函数每次调用是返回一个新组件，因此如果在 Render 中使用增强函数，就会导致每次都重新渲染整个HOC，而且之前的状态会丢失；

#35、React如何进行组件/逻辑复用？

抛开已经被官方弃用的Mixin,组件抽象的技术目前有三种比较主流:

- 高阶组件:
 - 属性代理
 - 反向继承
- 渲染属性
- react-hooks

#36、你对 Time Slice的理解？

时间分片

- React 在渲染 (render) 的时候，不会阻塞现在的线程
- 如果你的设备足够快，你会感觉渲染是同步的
- 如果你设备非常慢，你会感觉还算是灵敏的
- 虽然是异步渲染，但是你将会看到完整的渲染，而不是一个组件一行行的渲染出来
- 同样书写组件的方式

也就是说，这是React背后在做的事情，对于我们开发者来说，是透明的，具体是什么样的效果呢？

#37、setState到底异步还是同步？

先给出答案: 有时表现出异步,有时表现出同步

- `setState` 只在合成事件和钩子函数中是“异步”的，在原生事件和 `setTimeout` 中都是同步的
- `setState` 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的

值，形成了所谓的“异步”，当然可以通过第二个参数 `setState(partialState, callback)` 中的 `callback` 拿到更新后的结果

- `setState` 的批量更新优化也是建立在“异步”（合成事件、钩子函数）之上的，在原生事件和 `setTimeout` 中不会批量更新，在“异步”中如果对同一个值进行多次 `setState`，`setState` 的批量更新策略会对其进行覆盖，取最后一次的执行，如果是同时 `setState` 多个不同的值，在更新时会对其进行合并批量更新

#七、Vue

#1 对于MVVM的理解

MVVM是Model-View-ViewModel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。

- MVVM 是 Model-View-ViewModel 的缩写
- Model: 代表数据模型，也可以在Model中定义数据修改和操作的业务逻辑。我们可以把Model称为数据层，因为它仅仅关注数据本身，不关心任何行为
- View: 用户操作界面。当ViewModel对Model进行更新的时候，会通过数据绑定更新到View
- ViewModel: 业务逻辑层，View需要什么数据，ViewModel要提供这个数据；View有某些操作，ViewModel就要响应这些操作，所以可以说它是Model for View.
- **总结：** MVVM模式简化了界面与业务的依赖，解决了数据频繁更新。MVVM 在使用当中，利用双向绑定技术，使得 Model 变化时，ViewModel 会自动更新，而 ViewModel 变化时，View 也会自动变化。

#2 请详细说下你对vue生命周期的理解

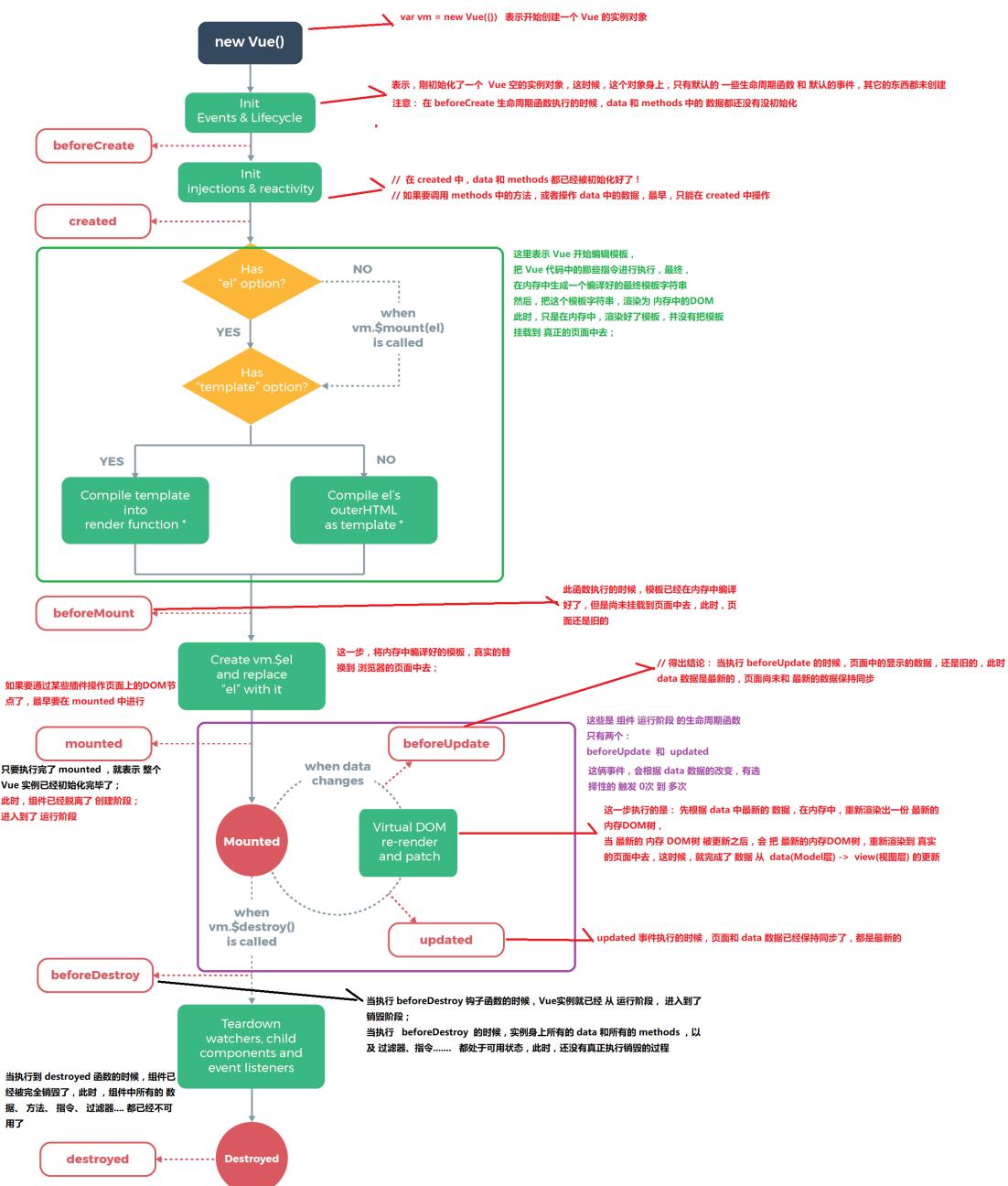
答：总共分为8个阶段创建前/后，载入前/后，更新前/后，销毁前/后

生命周期是什么

Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模版、挂载Dom -> 渲染、更新 -> 渲染、卸载等一系列过程，我们称这是Vue的生命周期

各个生命周期的作用

| 生命周期 | 描述 |
|---------------|---------------------------------------|
| beforeCreate | 组件实例被创建之初，组件的属性生效之前 |
| created | 组件实例已经完全创建，属性也绑定，但真实dom还没有生成，\$el还不可用 |
| beforeMount | 在挂载开始之前被调用：相关的 render 函数首次被调用 |
| mounted | el 被新创建的 vm.\$el 替换，并挂载到实例上去之后调用该钩子 |
| beforeUpdate | 组件数据更新之前调用，发生在虚拟 DOM 打补丁之前 |
| update | 组件数据更新之后 |
| activated | keep-alive专属，组件被激活时调用 |
| deactivated | keep-alive专属，组件被销毁时调用 |
| beforeDestory | 组件销毁前调用 |
| destoryed | 组件销毁后调用 |



由于Vue会在初始化实例时对属性执行 getter/setter 转化，所以属性必须在 data 对象上存在才能让 vue 将它转换为响应式的。Vue提供了 \$set 方法用来触发视图更新

```
export default {
  data() {
    return {
      obj: {
        name: 'fei'
      }
    }
  },
  mounted() {
    this.$set(this.obj, 'sex', 'man')
  }
}
```

什么是vue生命周期？

- 答：Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载 Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。

vue生命周期的作用是什么？

- 答：它的生命周期中有多个事件钩子，让我们在控制整个Vue实例的过程时更容易形成好的逻辑。

vue生命周期总共有几个阶段？

- 答：它可以总共分为 8 个阶段：创建前/后、载入前/后、更新前/后、销毁前/销毁后。

第一次页面加载会触发哪几个钩子？

- 答：会触发下面这几个 beforeCreate、created、beforeMount、mounted 。

DOM 渲染在哪个周期中就已经完成？

- 答：DOM 渲染在 mounted 中就已经完成了

#3 Vue实现数据双向绑定的原理：Object.defineProperty()

- vue 实现数据双向绑定主要是：采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty() 来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应监听回调。当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty() 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让 vue 追踪依赖，在属性被访问和修改时通知变化。
- vue的数据双向绑定 将 MVVM 作为数据绑定的入口，整合 Observer，Compile 和 Watcher 三者，通过 observer 来监听自己的 model 的数据变化，通过 compile 来解析编译模板指令（vue 中是用来解析 {{}}），最终利用 watcher 搭起 observer 和 compile 之间的通信桥梁，达到数据变化 → 视图更新；视图交互变化（input）→ 数据 model 变更双向绑定效果。

#4 Vue组件间的参数传递

父组件与子组件传值

父组件传给子组件：子组件通过 props 方法接受数据；

- 子组件传给父组件： \$emit 方法传递参数

非父子组件间的数据传递，兄弟组件传值

`eventBus`，就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。项目比较小时，用这个比较合适（虽然也有不少人推荐直接用`VUEX`，具体来说看需求）

#5 Vue的路由实现：hash模式 和 history模式

- `hash` 模式：在浏览器中符号“#”，#以及#后面的字符称之为 hash，用 `window.location.hash` 读取。特点：`hash` 虽然在 URL 中，但不被包括在 HTTP 请求中；用来指导浏览器动作，对服务端安全无用，`hash` 不会重加载页面。
- `history` 模式：`history` 采用 HTML5 的新特性；且提供了两个新方法：`pushState()`，`replaceState()` 可以对浏览器历史记录栈进行修改，以及 `popstate` 事件的监听到状态变更

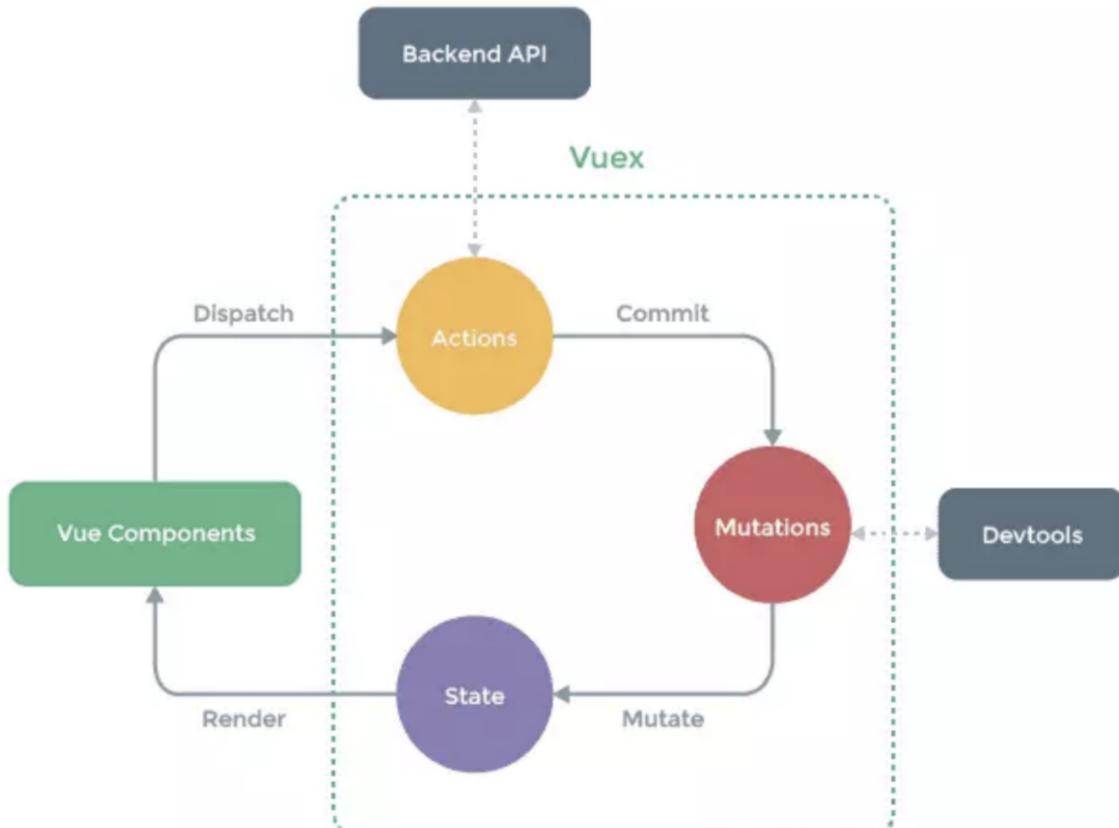
#5 vue路由的钩子函数

首页可以控制导航跳转，`beforeEach`，`afterEach` 等，一般用于页面 `title` 的修改。一些需要登录才能调整页面的重定向功能。

- `beforeEach` 主要有3个参数 `to`，`from`，`next`。
- `to`：`route` 即将进入的目标路由对象。
- `from`：`route` 当前导航正要离开的路由。
- `next`：`function`一定要调用该方法 `resolve` 这个钩子。执行效果依赖 `next` 方法的调用参数。可以控制网页的跳转

#6 vuex是什么？怎么使用？哪种功能场景使用它？

- 只用来读取的状态集中放在 `store` 中；改变状态的方式是提交 `mutations`，这是个同步的事物；异步逻辑应该封装在 `action` 中。
- 在 `main.js` 引入 `store`，注入。新建了一个目录 `store`，... `export`
- 场景有：单页应用中，组件之间的状态、音乐播放、登录状态、加入购物车



- `state`: `Vuex` 使用单一状态树,即每个应用将仅仅包含一个 `store` 实例, 但单一状态树和模块化并不冲突。存放的数据状态, 不可以直接修改里面的数据。
- `mutations`: `mutations` 定义的方法动态修改 `Vuex` 的 `store` 中的状态或数据
- `getters`: 类似 `vue` 的计算属性, 主要用来过滤一些数据。
- `action`: `actions` 可以理解为通过将 `mutations` 里面处理数据的方法变成可异步的处理数据的方法, 简单的说就是异步操作数据。`view` 层通过 `store.dispatch` 来分发 `action`

```
const store = new Vuex.Store({ //store实例
  state: {
    count: 0
  },
  mutations: {
    increment (state) {
      state.count++
    }
  },
  actions: {
    increment (context) {
      context.commit('increment')
    }
  }
})
```

`modules`: 项目特别复杂的时候, 可以让每一个模块拥有自己的 `state`、`mutation`、`action`、`getters`, 使得结构非常清晰, 方便管理

```
const moduleA = {
  state: { ... },
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: { ... },
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})
```

#7 v-if 和 v-show 区别

- 答: `v-if` 按照条件是否渲染, `v-show` 是 `display` 的 `block` 或 `none`;

#8 \$route 和 \$router 的区别

- `$route` 是“路由信息对象”, 包括 `path`, `params`, `hash`, `query`, `fullPath`, `matched`, `name` 等路由信息参数。
- 而 `$router` 是“路由实例”对象包括了路由的跳转方法, 钩子函数等

#9 如何让CSS只在当前组件中起作用?

将当前组件的 `<style>` 修改为 `<style scoped>`

#10 <keep-alive></keep-alive> 的作用是什么？

- keep-alive可以实现组件缓存，当组件切换时不会对当前组件进行卸载
- <keep-alive></keep-alive> 包裹动态组件时，会缓存不活动的组件实例，主要用于保留组件状态或避免重新渲染

比如有一个列表和一个详情，那么用户就会经常执行打开详情=>返回列表=>打开详情...这样的话列表和详情都是一个频率很高的页面，那么就可以对列表组件使用<keep-alive></keep-alive>进行缓存，这样用户每次返回列表的时候，都能从缓存中快速渲染，而不是重新渲染

- 常用的两个属性 `include/exclude`，允许组件有条件的进行缓存
- 两个生命周期 `activated/deactivated`，用来得知当前组件是否处于活跃状态

#11 指令v-el的作用是什么？

提供一个在页面上已存在的 DOM 元素作为 vue 实例的挂载目标。可以是 CSS 选择器，也可以是一个 `HTMLElement` 实例。

#12 在Vue中使用插件的步骤

- 采用 ES6 的 `import ... from ...` 语法或 commonJS 的 `require()` 方法引入插件
- 使用全局方法 `vue.use(plugin)` 使用插件，可以传入一个选项对象 `vue.use(MyPlugin, { someOption: true })`

#13 请列举出3个Vue中常用的生命周期钩子函数？

- `created`：实例已经创建完成之后调用，在这一步，实例已经完成数据观测，属性和方法的运算，`watch/event` 事件回调。然而，挂载阶段还没有开始，`$el` 属性目前还不可见
- `mounted`：`el` 被新创建的 `vm.$el` 替换，并挂载到实例上去之后调用该钩子。如果 `root` 实例挂载了一个文档内元素，当 `mounted` 被调用时 `vm.$el` 也在文档内。
- `activated`：`keep-alive` 组件激活时调用

#14 vue-cli 工程技术集合介绍

问题一：构建的 vue-cli 工程都到了哪些技术，它们的作用分别是什么？

- `vue.js`：vue-cli 工程的核心，主要特点是双向数据绑定和组件系统。
- `vue-router`：vue 官方推荐使用的路由框架。
- `vuex`：专为 `vue.js` 应用项目开发的状态管理器，主要用于维护 `vue` 组件间共用的一些变量和方法。
- `axios`（或者 `fetch`、`ajax`）：用于发起 `GET`、或 `POST` 等 `http` 请求，基于 `Promise` 设计。
- `vuex` 等：一个专为 `vue` 设计的移动端UI组件库。
- 创建一个 `emit.js` 文件，用于 `vue` 事件机制的管理。
- `webpack`：模块加载和 `vue-cli` 工程打包器。

问题二：vue-cli 工程常用的 npm 命令有哪些？

- 下载 `node_modules` 资源包的命令：

```
npm install
```

- 启动 `vue-cli` 开发环境的 npm 命令：

```
npm run dev
```

- `vue-cli` 生成生产环境部署资源的 `npm` 命令：

```
npm run build
```

- 用于查看 `vue-cli` 生产环境部署资源文件大小的 `npm` 命令：

```
npm run build --report
```

在浏览器上自动弹出一个展示 `vue-cli` 工程打包后 `app.js`、`manifest.js`、`vendor.js` 文件里面所包含代码的页面。可以据此优化 `vue-cli` 生产环境部署的静态资源，提升页面的加载速度

#15 NextTick

`nextTick` 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM

#16 vue的优点是什么？

- 低耦合。视图（view）可以独立于 Model 变化和修改，一个 viewModel 可以绑定到不同的 "view" 上，当 View 变化的时候 Model 可以不变，当 Model 变化的时候 view 也可以不变
- 可重用性。你可以把一些视图逻辑放在一个 viewModel 里面，让很多 view 重用这段视图逻辑
- 可测试。界面本来是比较难于测试的，而现在测试可以针对 viewModel 来写

#17 路由之间跳转？

声明式（标签跳转）

```
<router-link :to="index">
```

编程式（js跳转）

```
router.push('index')
```

#18 实现 Vue SSR

其基本实现原理

- `app.js` 作为客户端与服务端的公用入口，导出 `Vue` 根实例，供客户端 `entry` 与服务端 `entry` 使用。客户端 `entry` 主要作用挂载到 `DOM` 上，服务端 `entry` 除了创建和返回实例，还进行路由匹配与数据预获取。
- `webpack` 为客户端打包一个 `Client Bundle`，为服务端打包一个 `Server Bundle`。
- 服务器接收请求时，会根据 `url`，加载相应组件，获取和解析异步数据，创建一个读取 `Server Bundle` 的 `BundleRenderer`，然后生成 `html` 发送给客户端。
- 客户端混合，客户端收到从服务端传来的 `DOM` 与自己的生成的 `DOM` 进行对比，把不相同的 `DOM` 激活，使其能够响应后续变化，这个过程称为客户端激活。为确保混合成功，客户端与服务器端需要共享同一套数据。在服务端，可以在渲染之前获取数据，填充到 `store` 里，这样，在客户端挂载到 `DOM` 之前，可以直接从 `store` 里取数据。首屏的动态数据通过 `window.__INITIAL_STATE__` 发送到客户端

`Vue SSR` 的实现，主要就是把 `vue` 的组件输出成一个完整 `HTML`，`vue-server-renderer` 就是干这事的

- `Vue SSR` 需要做的事多点（输出完整 `HTML`），除了 `complier -> vnode`，还需如数据获取填充至 `HTML`、客户端混合（`hydration`）、缓存等等。相比于其他模板引擎（`ejs, jade` 等），最终要实现的目的是一样的，性能上可能要差点

#19 Vue 组件 data 为什么必须是函数

- 每个组件都是 `Vue` 的实例。
- 组件共享 `data` 属性，当 `data` 的值是同一个引用类型的值时，改变其中一个会影响其他

#20 Vue computed 实现

- 建立与其他属性（如：`data`、`store`）的联系；
- 属性改变后，通知计算属性重新计算

实现时，主要如下

- 初始化 `data`，使用 `Object.defineProperty` 把这些属性全部转为 `getter/setter`。
- 初始化 `computed`，遍历 `computed` 里的每个属性，每个 `computed` 属性都是一个 `watch` 实例。每个属性提供的函数作为属性的 `getter`，使用 `Object.defineProperty` 转化。
- `Object.defineProperty` `getter` 依赖收集。用于依赖发生变化时，触发属性重新计算。
- 若出现当前 `computed` 计算属性嵌套其他 `computed` 计算属性时，先进行其他的依赖收集

#21 Vue complier 实现

- 模板解析这种事，本质是将数据转化为一段 `html`，最开始出现在后端，经过各种处理吐给前端。随着各种 `mv*` 的兴起，模板解析交由前端处理。
- 总的来说，`Vue complier` 是将 `template` 转化成一个 `render` 字符串。

可以简单理解成以下步骤：

- `parse` 过程，将 `template` 利用正则转化成 `AST` 抽象语法树。
- `optimize` 过程，标记静态节点，后 `diff` 过程跳过静态节点，提升性能。
- `generate` 过程，生成 `render` 字符串

#22 怎么快速定位哪个组件出现性能问题

用 `timeline` 工具。大意是通过 `timeline` 来查看每个函数的调用时常，定位出哪个函数的问题，从而能判断哪个组件出了问题

#23 开发中常用的指令有哪些

- `v-model`：一般用在表达输入，很轻松的实现表单控件和数据的双向绑定
- `v-html`：更新元素的 `innerHTML`
- `v-show` 与 `v-if`：条件渲染，注意二者区别

使用了 `v-if` 的时候，如果值为 `false`，那么页面将不会有这个 `html` 标签生成。`v-show` 则是不管值为 `true` 还是 `false`，`html` 元素都会存在，只是 `CSS` 中的 `display` 显示或隐藏

- `v-on :click`：可以简写为 `@click`，绑定一个事件。如果事件触发了，就可以指定事件的处理函数
- `v-for`：基于源数据多次渲染元素或模板块
- `v-bind`：当表达式的值改变时，将其产生的连带影响，响应式地作用于 `DOM`

语法: `v-bind:title="msg"` 简写: `:title="msg"`

#24 Proxy 相比于 `defineProperty` 的优势

`Object.defineProperty()` 的问题主要有三个:

- 不能监听数组的变化
- 必须遍历对象的每个属性
- 必须深层遍历嵌套的对象

`Proxy` 在 ES2015 规范中被正式加入, 它有以下几个特点

- 针对对象: 针对整个对象, 而不是对象的某个属性, 所以也就不需要对 keys 进行遍历。这解决了上述 `Object.defineProperty()` 第二个问题
- 支持数组: `Proxy` 不需要对数组的方法进行重载, 省去了众多 hack, 减少代码量等于减少了维护成本, 而且标准的就是最好的。

除了上述两点之外, `Proxy` 还拥有以下优势:

- `Proxy` 的第二个参数可以有 13 种拦截方法, 这比起 `Object.defineProperty()` 要更加丰富
- `Proxy` 作为新标准受到浏览器厂商的重点关注和性能优化, 相比之下 `Object.defineProperty()` 是一个已有的老方法。

#25 vue-router 有哪几种导航守卫?

- 全局守卫
- 路由独享守卫
- 路由组件内的守卫

全局守卫

`vue-router` 全局有三个守卫

- `router.beforeEach` 全局前置守卫 进入路由之前
- `router.beforeResolve` 全局解析守卫(2.5.0+) 在 `beforeRouteEnter` 调用之后调用
- `router.afterEach` 全局后置钩子 进入路由之后

```
// main.js 入口文件
import router from './router'; // 引入路由
router.beforeEach((to, from, next) => {
  next();
});
router.beforeResolve((to, from, next) => {
  next();
});
router.afterEach((to, from) => {
  console.log(' afterEach 全局后置钩子');
});
```

路由独享守卫

如果你不想全局配置守卫的话, 你可以为某些路由单独配置守卫

```

const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // 参数用法什么的都一样，调用顺序在全局前置守卫后面，所以不会被全局守卫覆盖
        // ...
      }
    }
  ]
})

```

路由组件内的守卫

- beforeRouteEnter 进入路由前，在路由独享守卫后调用 不能 获取组件实例 this，组件实例还没被创建
- beforeRouteUpdate (2.2) 路由复用同一个组件时，在当前路由改变，但是该组件被复用时调用 可以访问组件实例 this
- beforeRouteLeave 离开当前路由时，导航离开该组件的对应路由时调用，可以访问组件实例 this

#26 组件之间的传值通信

组件之间通讯分为三种：父传子、子传父、兄弟组件之间的通讯

1. 父组件给子组件传值

- 使用 `props`，父组件可以使用 `props` 向子组件传递数据。
- 父组件 vue 模板 `father.vue`：

```

<template>
  <child :msg="message"></child>
</template>

<script>
import child from './child.vue';
export default {
  components: {
    child
  },
  data () {
    return {
      message: 'father message';
    }
  }
}
</script>

```

子组件vue模板`child.vue`：

```

<template>
  <div>{{msg}}</div>
</template>

<script>
export default {

```

```
props: {
  msg: {
    type: String,
    required: true
  }
}
</script>
```

2. 子组件向父组件通信

父组件向子组件传递事件方法，子组件通过 \$emit 触发事件，回调给父组件

父组件vue模板father.vue:

```
<template>
  <child @msgFunc="func"></child>
</template>

<script>
import child from './child.vue';
export default {
  components: {
    child
  },
  methods: {
    func (msg) {
      console.log(msg);
    }
  }
}
</script>
```

子组件vue模板child.vue:

```
<template>
  <button @click="handleClick">点我</button>
</template>

<script>
export default {
  props: {
    msg: {
      type: String,
      required: true
    }
  },
  methods () {
    handleClick () {
      //.....
      this.$emit('msgFunc');
    }
  }
}
</script>
```

3. 非父子, 兄弟组件之间通信

vue2中废弃了broadcast广播和分发事件的方法。父子组件中可以用props和\$emit()。如何实现非父子组件间的通信，可以通过实例一个vue实例Bus作为媒介，要相互通信的兄弟组件之中，都引入Bus，然后通过分别调用Bus事件触发和监听来实现通信和参数传递。Bus.js可以是这样：

```
import Vue from 'vue'  
export default new Vue()
```

在需要通信的组件都引入Bus.js：

```
<template>  
  <button @click="toBus">子组件传给兄弟组件</button>  
</template>  
  
<script>  
import Bus from '../common/js/bus.js'  
export default{  
  methods: {  
    toBus () {  
      Bus.$emit('on', '来自兄弟组件')  
    }  
  }  
}</script>
```

另一个组件也import Bus.js 在钩子函数中监听on事件

```
import Bus from '../common/js/bus.js'  
export default {  
  data() {  
    return {  
      message: ''  
    },  
    mounted() {  
      Bus.$on('on', (msg) => {  
        this.message = msg  
      })  
    }  
  }  
}
```

#27 Vue与Angular以及React的区别？

Vue与AngularJS的区别

- Angular采用TypeScript开发, 而Vue可以使用javascript也可以使用TypeScript
- AngularJS依赖对数据做脏检查，所以Watcher越多越慢；Vue.js使用基于依赖追踪的观察并且使用异步队列更新，所有的数据都是独立触发的。
- AngularJS社区完善, Vue的学习成本较小

Vue与React的区别

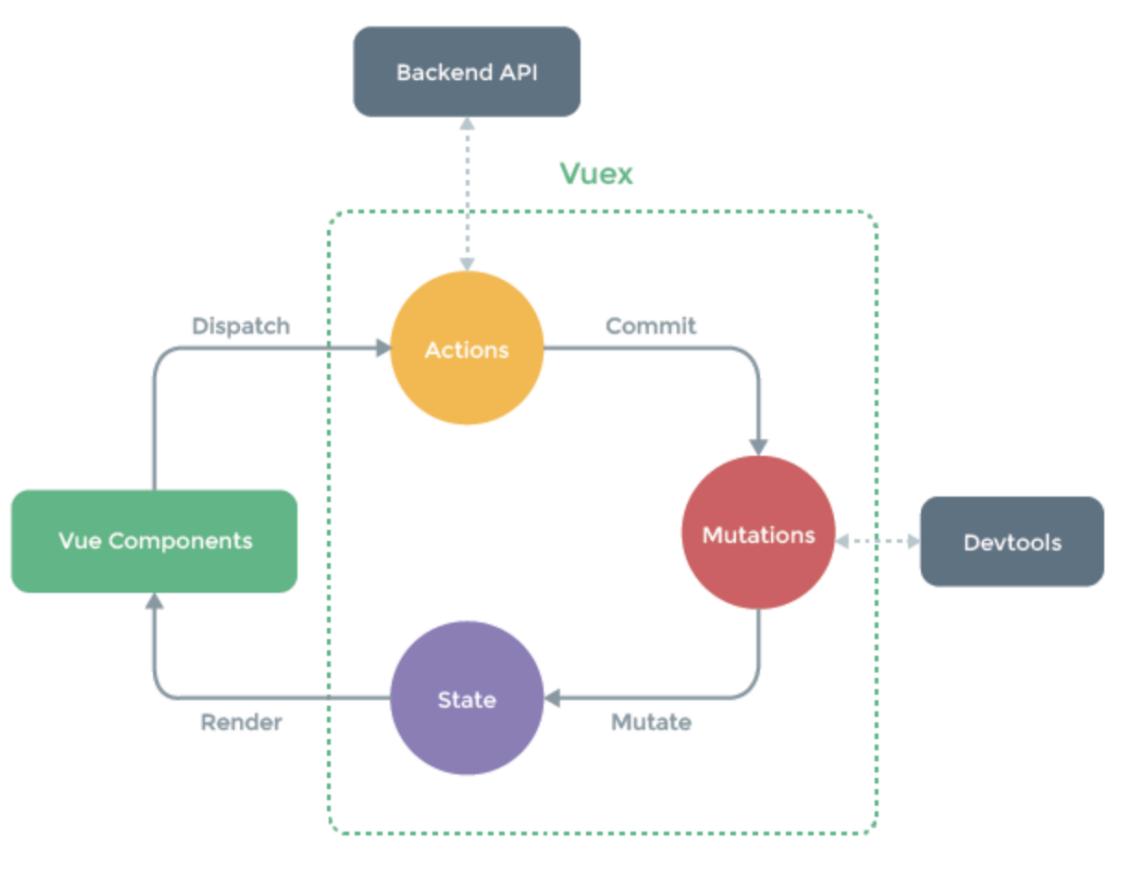
- vue组件分为全局注册和局部注册，在react中都是通过import相应组件，然后模版中引用；
- props是可以动态变化的，子组件也实时更新，在react中官方建议props要像纯函数那样，输入输出一致对应，而且不太建议通过props来更改视图；

- 子组件一般要显示地调用props选项来声明它期待获得的数据。而在react中不必需，另两者都有props校验机制；
- 每个Vue实例都实现了事件接口，方便父子组件通信，小型项目中不需要引入状态管理机制，而react必需自己实现；
- 使用插槽分发内容，使得可以混合父组件的内容与子组件自己的模板；
- 多了指令系统，让模版可以实现更丰富的功能，而React只能使用JSX语法；
- Vue增加的语法糖computed和watch，而在React中需要自己写一套逻辑来实现；
- react的思路是all in js，通过js来生成html，所以设计了jsx，还有通过js来操作css，社区的styled-component、jss等；而vue是把html，css，js组合到一起，用各自的处理方式，vue有单文件组件，可以把html、css、js写到一个文件中，html提供了模板引擎来处理。
- react做的事情很少，很多都交给社区去做，vue很多东西都是内置的，写起来确实方便一些，比如redux的combineReducer就对应vuex的modules，比如reselect就对应vuex的getter和vue组件的computed，vuex的mutation是直接改变的原始数据，而redux的reducer是返回一个全新的state，所以redux结合immutable来优化性能，vue不需要。
- react是整体的思路的就是函数式，所以推崇纯组件，数据不可变，单向数据流，当然需要双向的地方也可以做到，比如结合redux-form，组件的横向拆分一般是通过高阶组件。而vue是数据可变的，双向绑定，声明式的写法，vue组件的横向拆分很多情况下用mixin

#28 vuex是什么？怎么使用？哪种功能场景使用它？

- vuex就是一个仓库，仓库里放了很多对象。其中state就是数据源存放地，对应于一般vue对象里面的data
- state里面存放的数据是响应式的，vue组件从store读取数据，若是store中的数据发生改变，依赖这组数据的组件也会发生更新
- 它通过mapState把全局的state和getters映射到当前组件的computed计算属性

vuex的使用借助官方提供的一张图来说明：



Vuex有5种属性：分别是state、getter、mutation、action、module；

state

vuex 使用单一状态树,即每个应用将仅仅包含一个store 实例,但单一状态树和模块化并不冲突。
存放的数据状态,不可以直接修改里面的数据

mutations

mutations 定义的方法动态修改Vuex 的 store 中的状态或数据。

getters

类似vue的计算属性,主要用来过滤一些数据

action

- actions可以理解为通过将mutations里面处理数据的方法变成可异步的处理数据的方法,简单的说就是异步操作数据。view 层通过 store.dispatch 来分发 action。
- vuex一般用于中大型 web 单页应用中对应用的状态进行管理,对于一些组件间关系较为简单的小型应用,使用 vuex 的必要性不是很大,因为完全可以用组件 prop 属性或者事件来完成父子组件之间的通信,vuex 更多地用于解决跨组件通信以及作为数据中心集中式存储数据。
- 使用Vuex解决非父子组件之间通信问题 vuex 是通过将 state 作为数据中心、各个组件共享 state 实现跨组件通信的,此时的数据完全独立于组件,因此将组件间共享的数据置于 State 中能有效解决多层级组件嵌套的跨组件通信问题
- vuex 作为数据存储中心 vuex 的 State 在单页应用的开发中本身具有一个“数据库”的作用,可以将组件中用到的数据存储在 State 中,并在 Action 中封装数据读写的逻辑。这时候存在一个问题,一般什么样的数据会放在 State 中呢? 目前主要有两种数据会使用 vuex 进行管理: 1、组件之间全局共享的数据 2、通过后端异步请求的数据 比如做加入购物车、登录状态等都可以使用Vuex来管理数据状态

一般面试官问到这里vue基本知识就差不多了,如果更深入的研究就是和你探讨关于vue的底层源码;或者是具体在项目中遇到的问题,下面列举几个项目中可能遇到的问题:

- 开发时,改变数组或者对象的数据,但是页面没有更新如何解决?
- vue弹窗后如何禁止滚动条滚动?
- 如何在 vue 项目里正确地引用 jquery 和 jquery-ui的插件

#28 watch与computed的区别

computed:

- computed是计算属性,也就是计算值,它更多用于计算值的场景
- computed具有缓存性,computed的值在getter执行后是会缓存的,只有在它依赖的属性值改变之后,下一次获取computed的值时才会重新调用对应的getter来计算 computed适用于计算比较消耗性能的计算场景

watch:

- 更多的是「观察」的作用,类似于某些数据的监听回调,用于观察props \$emit或者本组件的值,当数据变化时来执行回调进行后续操作
- 无缓存性,页面重新渲染时值不变化也会执行

小结:

- 当我们要进行数值计算,而且依赖于其他数据,那么把这个数据设计为computed
- 如果你需要在某个数据变化时做一些事情,使用watch来观察这个数据变化

#29、Vue是如何实现双向绑定的？

利用Object.defineProperty劫持对象的访问器，在属性值发生变化时我们可以获取变化，然后根据变化进行后续响应，在vue3.0中通过Proxy代理对象进行类似的操作。

```
// 这是将要被劫持的对象
const data = {
  name: '',
};

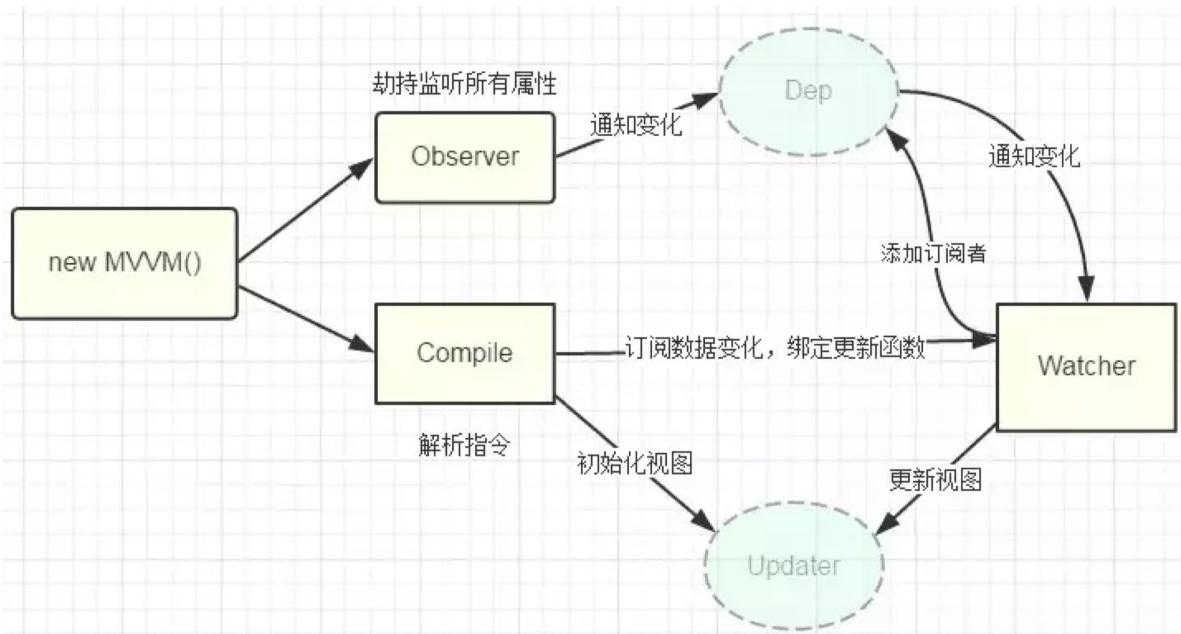
function say(name) {
  if (name === '古天乐') {
    console.log('给大家推荐一款超好玩的游戏');
  } else if (name === '渣渣辉') {
    console.log('戏我演过很多，可游戏我只玩贪玩懒月');
  } else {
    console.log('来做我的兄弟');
  }
}

// 遍历对象，对其属性值进行劫持
Object.keys(data).forEach(function(key) {
  Object.defineProperty(data, key, {
    enumerable: true,
    configurable: true,
    get: function() {
      console.log('get');
    },
    set: function(newVal) {
      // 当属性值发生变化时我们可以进行额外操作
      console.log(`大家好，我系${newVal}`);
      say(newVal);
    },
  });
});

data.name = '渣渣辉';
//大家好，我系渣渣辉
//戏我演过很多，可游戏我只玩贪玩懒月
```

#29 Vue2.x 响应式原理

Vue 采用数据劫持结合发布一订阅模式的方法，通过 Object.defineProperty() 来劫持各个属性的 setter, getter，在数据变动时发布消息给订阅者，触发相应的监听回调。



- `Observer` 遍历数据对象，给所有属性加上 `setter` 和 `getter`，监听数据的变化
- `compile` 解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

`watcher` 订阅者是 `Observer` 和 `Compile` 之间通信的桥梁，主要做的事情

- 在自身实例化时往属性订阅器 (`dep`) 里面添加自己
- 待属性变动 `dep.notice()` 通知时，调用自身的 `update()` 方法，并触发 `Compile` 中绑定的回调

Vue3.x响应式数据原理

`Vue3.x` 改用 `Proxy` 替代 `Object.defineProperty`。因为 `Proxy` 可以直接监听对象和数组的变化，并且有多达 13 种拦截方法。并且作为新标准将受到浏览器厂商重点持续的性能优化。

`Proxy` 只会代理对象的第一层，那么 `vue3` 又是怎样处理这个问题的呢？

判断当前 `Reflect.get` 的返回值是否为 `Object`，如果是则再通过 `reactive` 方法做代理，这样就实现了深度观测。

监测数组的时候可能触发多次get/set，那么如何防止触发多次呢？

我们可以判断 `key` 是否为当前被代理对象 `target` 自身属性，也可以判断旧值与新值是否相等，只有满足以上两个条件之一时，才有可能执行 `trigger`

#30 v-model 双向绑定原理

`v-model` 本质上是语法糖，`v-model` 在内部为不同的输入元素使用不同的属性并抛出不同的事件

- `text` 和 `textarea` 元素使用 `value` 属性和 `input` 事件
- `checkbox` 和 `radio` 使用 `checked` 属性和 `change` 事件
- `select` 字段将 `value` 作为 `prop` 并将 `change` 作为事件

所以我们可以 `v-model` 进行如下改写：

```
<input v-model="sth" />
// 等同于
<input :value="sth" @input="sth = $event.target.value" />
```

- 这个语法糖必须是固定的，也就是说属性必须为 `value`，方法名必须为：`input`。

- 知道了 `v-model` 的原理，我们可以在自定义组件上实现 `v-model`

```
//Parent
<template>
  {{num}}
  <child v-model="num">
</template>
export default {
  data(){
    return {
      num: 0
    }
  }
}

//Child
<template>
  <div @click="add">Add</div>
</template>
export default {
  props: ['value'],
  methods:{
    add(){
      this.$emit('input', this.value + 1)
    }
  }
}
```

#31 scoped样式穿透

`scoped` 虽然避免了组件间样式污染，但是很多时候我们需要修改组件中的某个样式，但是又不想去除 `scoped` 属性

1. 使用 `/deep/`

```
//Parent
<template>
  <div class="wrap">
    <child />
  </div>
</template>

<style lang="scss" scoped>
  .wrap /deep/ .box{
    background: red;
  }
</style>

//Child
<template>
  <div class="box"></div>
</template>
```

1. 使用两个 `style` 标签

```
//Parent
```

```

<template>
<div class="wrap">
  <child />
</div>
</template>

<style lang="scss" scoped>
//其他样式
</style>
<style lang="scss">
.wrap .box{
  background: red;
}
</style>

//child
<template>
  <div class="box"></div>
</template>

```

#32 ref的作用

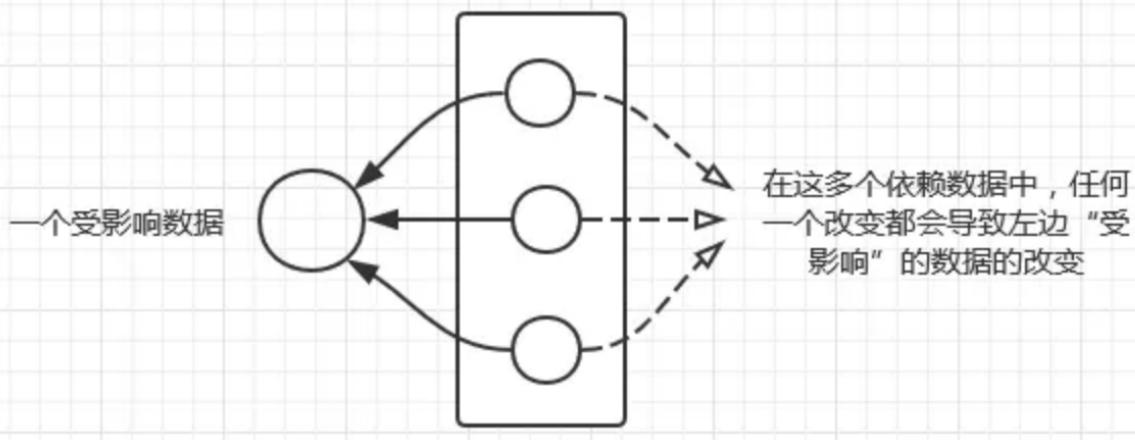
- 获取 dom 元素 `this.$refs.box`
- 获取子组件中的 data `this.$refs.box.msg`
- 调用子组件中的方法 `this.$refs.box.open()`

#33 computed和watch区别

1. 当页面中有某些数据依赖其他数据进行变动的时候，可以使用计算属性computed

`Computed` 本质是一个具备缓存的 `watcher`，依赖的属性发生变化就会更新视图。适用于计算比较消耗性能的计算场景。当表达式过于复杂时，在模板中放入过多逻辑会让模板难以维护，可以将复杂的逻辑放入计算属性中处理

computed擅长处理的情景：一个数据受多个数据影响



```

<template>{{fullName}}</template>
export default {
  data() {
    return {
      firstName: 'xie',
      lastName: 'yu fei',
    }
  }
}

```

```

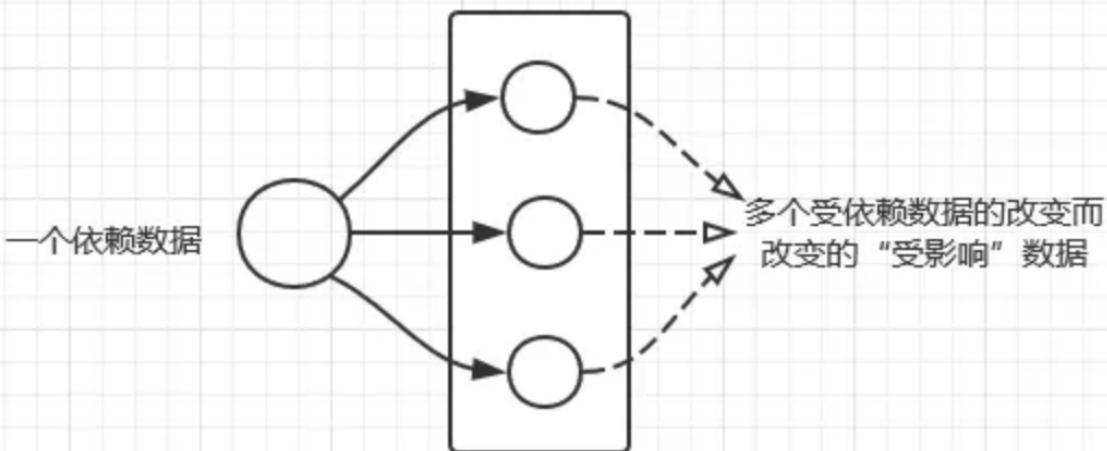
        }
    },
    computed:{
        fullName: function(){
            return this.firstName + ' ' + this.lastName
        }
    }
}

```

1. `watch` 用于观察和监听页面上的vue实例，如果要在数据变化的同时进行异步操作或者是比较大的开销，那么 `watch` 为最佳选择

`watch` 没有缓存性，更多的是观察的作用，可以监听某些数据执行回调。当我们需要深度监听对象中的属性时，可以打开 `deep: true` 选项，这样便会对对象中的每一项进行监听。这样会带来性能问题，优化的话可以使用字符串形式监听，如果没有写到组件中，不要忘记使用 `unwatch` 手动注销

watch擅长处理的情景：一个数据影响多个数据



```

<template>{{fullName}}</template>
export default {
    data(){
        return {
            firstName: 'xie',
            lastName: 'xiao fei',
            fullName: 'xie xiao fei'
        }
    },
    watch:{
        firstName(val) {
            this.fullName = val + ' ' + this.lastName
        },
        lastName(val) {
            this.fullName = this.firstName + ' ' + val
        }
    }
}

```

#34 vue-router守卫

导航守卫 `router.beforeEach` 全局前置守卫

- `to: Route`: 即将要进入的目标 (路由对象)
- `from: Route`: 当前导航正要离开的路由
- `next: Function`: 一定要调用该方法来 `resolve` 这个钩子。 (一定要用这个函数才能去到下一个路由, 如果不用就拦截)
- 执行效果依赖 `next` 方法的调用参数。
- `next()`: 进行管道中的下一个钩子。如果全部钩子执行完了, 则导航的状态就是 `confirmed` (确认的)。
- `next(false)`: 取消进入路由, url地址重置为from路由地址(也就是将要离开的路由地址)

```
// main.js 入口文件
import router from './router'; // 引入路由
router.beforeEach((to, from, next) => {
  next();
});
router.beforeResolve((to, from, next) => {
  next();
});
router.afterEach((to, from) => {
  console.log('afterEach 全局后置钩子');
});
```

路由独享的守卫 你可以在路由配置上直接定义 `beforeEnter` 守卫

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

组件内的守卫 你可以在路由组件内直接定义以下路由导航守卫

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不! 能! 获取组件实例 `this`
    // 因为当守卫执行前, 组件实例还没被创建
  },
  beforeRouteUpdate (to, from, next) {
    // 在当前路由改变, 但是该组件被复用时调用
    // 举例来说, 对于一个带有动态参数的路径 /foo/:id, 在 /foo/1 和 /foo/2 之间跳转的时候,
    // 由于会渲染同样的 Foo 组件, 因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
    // 可以访问组件实例 `this`
  },
  beforeRouteLeave (to, from, next) {
```

```
// 导航离开该组件的对应路由时调用，我们用它来禁止用户离开
// 可以访问组件实例 `this`
// 比如还未保存草稿，或者在用户离开前，
将setInterval销毁，防止离开之后，定时器还在调用。
}
}
```

#35 vue修饰符

- `stop`: 阻止事件的冒泡
- `prevent`: 阻止事件的默认行为
- `once`: 只触发一次
- `self`: 只触发自己的事件行为时，才会执行

#36 vue项目中的性能优化

- 不要在模板里面写过多表达式
- 循环调用子组件时添加key
- 频繁切换的使用v-show，不频繁切换的使用v-if
- 尽量少用float，可以用flex
- 按需加载，可以用require或者import()按需加载需要的组件
- 路由懒加载

#37 vue.extend和vue.component

- `extend` 是构造一个组件的语法器。然后这个组件你可以作用到Vue.component这个全局注册方法里还可以在任意vue模板里使用组件。也可以作用到vue实例或者某个组件中的components属性中并在内部使用apple组件。
- `vue.component` 你可以创建，也可以取组件。

#38 Vue的SPA 如何优化加载速度

- 减少入口文件体积
- 静态资源本地缓存
- 开启Gzip压缩
- 使用SSR,nuxt.js

#39 移动端如何设计一个比较友好的Header组件？

当时的思路是头部(Header)一般分为左、中、右三个部分，分为三个区域来设计，中间为主标题，每个页面的标题肯定不同，所以可以通过vue props的方式做成可配置对外进行暴露，左侧大部分页面可能都是回退按钮，但是样式和内容不尽相同，右侧一般都是具有功能性的操作按钮，所以左右两侧可以通过vue slot插槽的方式对外暴露以实现多样化，同时也可提供default slot默认插槽来统一页面风格

#40 Proxy与Object.defineProperty的优劣对比？

Proxy的优势如下：

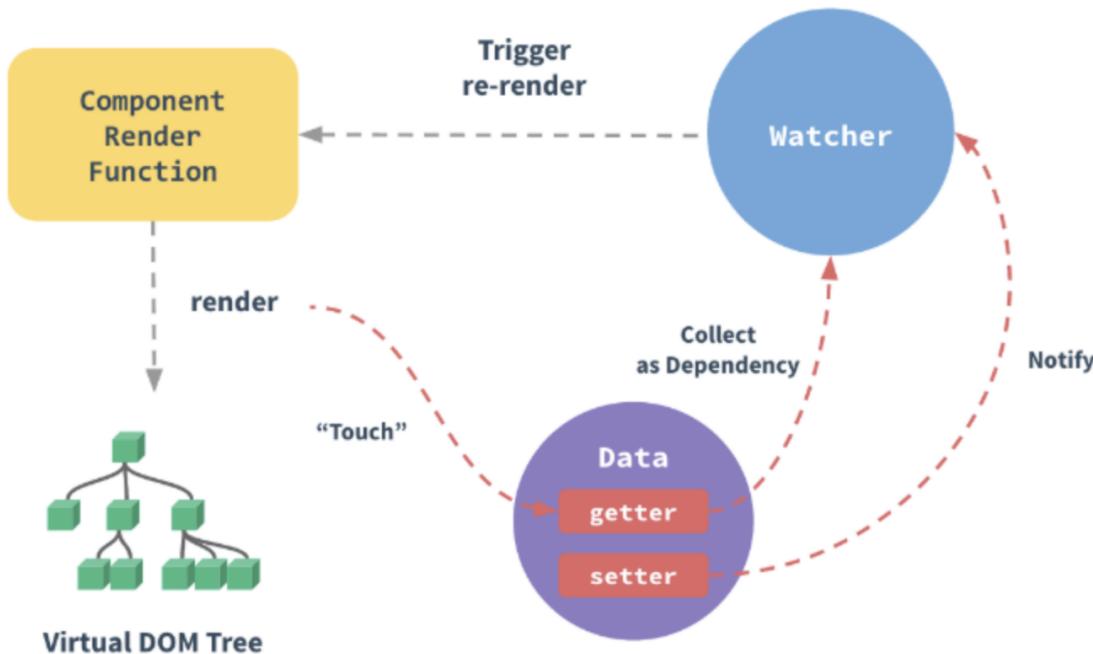
- Proxy可以直接监听对象而非属性
- Proxy可以直接监听数组的变化
- Proxy有多达13种拦截方法,不限于apply、ownKeys、deleteProperty、has等等是Object.defineProperty不具备的

- Proxy返回的是一个新对象,我们可以只操作新的对象达到目的,而Object.defineProperty只能遍历对象属性直接修改
- Proxy作为新标准将受到浏览器厂商重点持续的性能优化,也就是传说中的新标准的性能红利

Object.defineProperty的优势如下:

兼容性好,支持IE9

#41 你是如何理解Vue的响应式系统的?



响应式系统简述:

- 任何一个 Vue Component 都有一个与之对应的 Watcher 实例。
- Vue 的 data 上的属性会被添加 getter 和 setter 属性。
- 当 Vue Component render 函数被执行的时候, data 上会被触碰(touch), 即被读, getter 方法会被调用, 此时 Vue 会去记录此 Vue component 所依赖的所有 data。(这一过程被称为依赖收集)
- data 被改动时 (主要是用户操作), 即被写, setter 方法会被调用, 此时 Vue 会去通知所有依赖于此 data 的组件去调用他们的 render 函数进行更新。

#42 既然Vue通过数据劫持可以精准探测数据变化,为什么还需要虚拟DOM进行diff检测差异?

现代前端框架有两种方式侦测变化,一种是pull一种是push

- pull: 其代表为React,我们可以回忆一下React是如何侦测到变化的,我们通常会用setStateAPI显式更新,然后React会进行一层层的Virtual Dom Diff操作找出差异,然后Patch到DOM上,React从一开始就不知道到底是哪发生了变化,只是知道「有变化了」,然后再进行比较暴力的Diff操作查找「哪发生了变化了」, 另外一个代表就是Angular的脏检查操作。
- push: Vue的响应式系统则是push的代表,当Vue程序初始化的时候就会对数据data进行依赖的收集,一旦数据发生变化,响应式系统就会立刻得知,因此Vue是一开始就知道是「在哪发生了变化了」,但是这又会产生一个问题,如果你熟悉Vue的响应式系统就知道,通常一个绑定一个数据就需要一个Watcher,一但我们的绑定细粒度过高就会产生大量的Watcher,这会带来内存以及依赖追踪的开销,而细粒度过低会无法精准侦测变化,因此Vue的设计是选择中等细粒度的方案,在组件级别进行push侦测的方式,也就是那套响应式系统,通常我们会第一时间侦测到发生变化的组件,然后在组件内部进行Virtual Dom Diff获取更加具体的差异,而Virtual Dom Diff则是pull操作,Vue是push+pull结合的方式进行变化侦测的

#43 Vue为什么没有类似于React中shouldComponentUpdate的生命周期?

考点: Vue的变化侦测原理

前置知识: 依赖收集、虚拟DOM、响应式系统

根本原因是Vue与React的变化侦测方式有所不同

- React是pull的方式侦测变化,当React知道发生变化后,会使用Virtual Dom Diff进行差异检测,但是很多组件实际上是肯定不会发生变化的,这个时候需要用shouldComponentUpdate进行手动操作来减少diff,从而提高程序整体的性能.
- Vue是pull+push的方式侦测变化的,在一开始就知道那个组件发生了变化,因此在push的阶段并不需要手动控制diff,而组件内部采用的diff方式实际上是可以引入类似于shouldComponentUpdate相关生命周期的,但是通常合理大小的组件不会有过量的diff,手动优化的价值有限,因此目前Vue并没有考虑引入shouldComponentUpdate这种手动优化的生命周期.

#44 Vue中的key到底有什么用?

- key是为Vue中的vnode标记的唯一id,通过这个key,我们的diff操作可以更准确、更快速
- diff算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的key与旧节点进行比对,然后超出差异.

diff程可以概括为: oldCh和newCh各有两个头尾的变量StartIdx和EndIdx, 它们的2个变量相互比较, 一共有4种比较方式。如果4种比较都没匹配, 如果设置了key, 就会用key进行比较, 在比较的过程中, 变量会往中间靠, 一旦StartIdx>EndIdx表明oldCh和newCh至少有一个已经遍历完了, 就会结束比较,这四种比较方式就是首、尾、旧尾新头、旧头新尾.

准确: 如果不加key,那么vue会选择复用节点(Vue的就地更新策略),导致之前节点的状态被保留下来,会产生一系列的bug. 快速: key的唯一性可以被Map数据结构充分利用,相比于遍历查找的时间复杂度 $O(n)$, Map 的时间复杂度仅为 $O(1)$.

```
60  function createKeyToOldIdx (children, beginIdx, endIdx) {  
61    let i, key  
62    const map = {}  
63    for (i = beginIdx; i <= endIdx; ++i) {  
64      key = children[i].key  
65      if (isDef(key)) map[key] = i  
66    }  
67    return map  
68  }
```

#45 vue 项目性能优化

代码层面:

- 合理使用 `v-if` 和 `v-show`
- 区分 `computed` 和 `watch` 的使用
- `v-for` 遍历为 `item` 添加 `key`
- `v-for` 遍历避免同时使用 `v-if`
- 通过 `addEventListener` 添加的事件在组件销毁时要用 `removeEventListener` 手动移除这些事件的监听
- 图片懒加载
- 路由懒加载
- 第三方插件按需引入
- `SSR` 服务端渲染, 首屏加载速度快, `SEO` 效果好

Webpack 层面优化:

- 对图片进行压缩
- 使用 `CommonsChunkPlugin` 插件提取公共代码
- 提取组件的 CSS
- 优化 `sourceMap`
- 构建结果输出分析，利用 `webpack-bundle-analyzer` 可视化分析工具

#46 nextTick

`nextTick` 可以让我们在下次 `DOM` 更新循环结束之后执行延迟回调，用于获得更新后的 `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。

#47 说一下vue2.x中如何监测数组变化

使用了函数劫持的方式，重写了数组的方法，`Vue` 将 `data` 中的数组进行了原型链重写，指向了自己定义的数组原型方法。这样当调用数组api时，可以通知依赖更新。如果数组中包含着引用类型，会对数组中的引用类型再次递归遍历进行监控。这样就实现了监测数组变化。

#48 你的接口请求一般放在哪个生命周期中

接口请求一般放在 `mounted` 中，但需要注意的是服务端渲染时不支持 `mounted`，需要放到 `created` 中。

#49 组件中的data为什么是一个函数

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。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。

#50 说一下v-model的原理

`v-model` 本质就是一个语法糖，可以看成是 `value + input` 方法的语法糖。可以通过 `model` 属性的 `prop` 和 `event` 属性来进行自定义。原生的 `v-model`，会根据标签的不同生成不同的事件和属性。

#51 Vue事件绑定原理说一下

原生事件绑定是通过 `addEventListener` 绑定给真实元素的，组件事件绑定是通过 `Vue` 自定义的 `$on` 实现的。

#52 Vue模版编译原理知道吗，能简单说一下吗？

简单说，`Vue` 的编译过程就是将 `template` 转化为 `render` 函数的过程。会经历以下阶段：

- 生成 `AST` 树
- 优化
- `codegen`

- 首先解析模版，生成 AST 语法树(一种用 javascript 对象的形式来描述整个模板)。使用大量的正则表达式对模板进行解析，遇到标签、文本的时候都会执行对应的钩子进行相关处理。
- vue 的数据是响应式的，但其实模板中并不是所有的数据都是响应式的。有一些数据首次渲染后就不会再变化，对应的DOM也不会变化。那么优化过程就是深度遍历AST树，按照相关条件对树节点进行标记。这些被标记的节点(静态节点)我们就可以跳过对它们的比对，对运行时的模板起到很大的优化作用。
- 编译的最后一步是将优化后的 AST 树转换为可执行的代码

#53 Vue2.x和Vue3.x渲染器的diff算法分别说一下

简单来说，diff 算法有以下过程

- 同级比较，再比较子节点
- 先判断一方有子节点一方没有子节点的情况(如果新的 children 没有子节点，将旧的子节点移除)
- 比较都有子节点的情况(核心 diff)
- 递归比较子节点
- 正常 diff 两个树的时间复杂度是 $O(n^3)$ ，但实际情况下我们很少会进行跨层级的移动 DOM，所以 vue 将 diff 进行了优化，从 $O(n^3) \rightarrow O(n)$ ，只有当新旧 children 都为多个子节点时才需要用核心的 diff 算法进行同层级比较。
- vue2 的核心 diff 算法采用了双端比较的算法，同时从新旧 children 的两端开始进行比较，借助 key 值找到可复用的节点，再进行相关操作。相比 React 的 diff 算法，同样情况下可以减少移动节点次数，减少不必要的性能损耗，更加的优雅
- 在创建 VNode 时就确定其类型，以及在 mount/patch 的过程中采用位运算来判断一个 VNode 的类型，在这个基础之上再配合核心的 diff 算法，使得性能上较 vue2.x 有了提升

#54 再说一下虚拟Dom以及key属性的作用

- 由于在浏览器中操作 DOM 是很昂贵的。频繁的操作 DOM，会产生一定的性能问题。这就是虚拟 Dom 的产生原因
- virtual DOM 本质就是用一个原生的 JS 对象去描述一个 DOM 节点。是对真实 DOM 的一层抽象
- virtual DOM 映射到真实 DOM 要经历 VNode 的 create、diff、patch 等阶段

key 的作用是尽可能的复用 DOM 元素

- 新旧 children 中的节点只有顺序是不同的时候，最佳的操作应该是通过移动元素的位置来达到更新的目的
- 需要在新旧 children 的节点中保存映射关系，以便能够在旧 children 的节点中找到可复用的节点。key 也就是 children 中节点的唯一标识

#55 Vue 中组件生命周期调用顺序说一下

- 组件的调用顺序都是先父后子，渲染完成的顺序是先子后父。
- 组件的销毁操作是先父后子，销毁完成的顺序是先子后父。

加载渲染过程

```
父beforeCreate`->`父created`->`父beforeMount`->`子beforeCreate`->`子created`->`子beforeMount`->`子mounted`->`父mounted`
```

子组件更新过程

```
父beforeUpdate`->`子beforeUpdate`->`子updated`->`父updated`
```

父组件更新过程

父 beforeUpdate` -> `父 updated

销毁过程

父beforeDestroy`->`子beforeDestroy`->`子destroyed`->`父destroyed

#56 SSR了解吗

SSR 也就是服务端渲染，也就是将 vue 在客户端把标签渲染成HTML的工作放在服务端完成，然后
再把html直接返回给客户端

SSR 有着更好的 SEO、并且首屏加载速度更快等优点。不过它也有一些缺点，比如我们的开发条件会受到限制，服务器端渲染只支持 beforeCreate 和 created 两个钩子，当我们需要一些外部扩展库时需要特殊处理，服务端渲染应用程序也需要处于 Node.js 的运行环境。还有就是服务器会有更大的负载需求

#57 你都做过哪些Vue的性能优化

编码阶段

- 尽量减少 data 中的数据， data 中的数据都会增加 getter 和 setter，会收集对应的 watcher
- v-if 和 v-for 不能连用
- 如果需要使用 v-for 给每项元素绑定事件时使用事件代理
- SPA 页面采用 keep-alive 缓存组件
- 在更多的情况下，使用 v-if 替代 v-show
- key 保证唯一
- 使用路由懒加载、异步组件
- 防抖、节流
- 第三方模块按需导入
- 长列表滚动到可视区域动态加载
- 图片懒加载

SEO优化

- 预渲染
- 服务端渲染 SSR

打包优化

- 压缩代码
- Tree Shaking/Scope Hoisting
- 使用 cdn 加载第三方模块
- 多线程打包 happypack
- splitChunks 抽离公共文件
- sourceMap 优化

用户体验

- 骨架屏
- PWA

还可以使用缓存(客户端缓存、服务端缓存)优化、服务端开启 gzip 压缩等。

#58 Vue.js特点

- 简洁：页面由 `HTML` 模板+`Json`数据+`Vue` 实例组成
- 数据驱动：自动计算属性和追踪依赖的模板表达式
- 组件化：用可复用、解耦的组件来构造页面
- 轻量：代码量小，不依赖其他库
- 快速：精确有效批量DOM更新
- 模板友好：可通过`npm`, `bower`等多种方式安装，很容易融入

#59 请说出vue.cli项目中src目录每个文件夹和文件的用法

- `assets` 文件夹是放静态资源；
- `components` 是放组件；
- `router` 是定义路由相关的配置；
- `view` 视图；
- `app.vue` 是一个应用主组件；
- `main.js` 是入口文件

#60 vue路由传参数

- 使用 `query` 方法传入的参数使用 `this.$route.query` 接受
- 使用 `params` 方式传入的参数使用 `this.$route.params` 接受

#61 vuex 是什么？有哪几种属性？

- `Vuex` 是一个专为 `Vue.js` 应用程序开发的状态管理模式。
- 有 5 种，分别是 `state`、`getter`、`mutation`、`action`、`module`
- `Vuex` 是一个专为 `Vue.js` 应用程序开发的状态管理模式。
- 有 5 种，分别是 `state`、`getter`、`mutation`、`action`、`module`
- `vuex` 的 `store` 是什么？
- `vuex` 就是一个仓库，仓库里放了很多对象。其中 `state` 就是数据源存放地，对于一般 `Vue` 对象里面的 `data` 里面存放的数据是响应式的，`Vue` 组件从 `store` 读取数据，若是 `store` 中的数据发生改变，依赖这组数据的组件也会发生更新它通过 `mapState` 把全局的 `state` 和 `getters` 映射到当前组件的 `computed` 计算属性

vuex 的 getter 是什么？

- `getter` 可以对 `state` 进行计算操作，它就是 `store` 的计算属性虽然在组件内也可以做计算属性，但是 `getters` 可以在多组件之间复用如果一个状态只在一个组件内使用，是可以不用 `getters`

vuex 的 mutation 是什么？

- 更改 `vuex` 的 `store` 中的状态的唯一方法是提交 `mutation`

vuex 的 action 是什么？

- `action` 类似于 `mutation`，不同在于：`action` 提交的是 `mutation`，而不是直接变更状态 `action` 可以包含任意异步操作
- `Vue` 中 `ajax` 请求代码应该写在组件的 `methods` 中还是 `vuex` 的 `action` 中
- `vuex` 的 `module` 是什么？

面对复杂的应用程序，当管理的状态比较多时；我们需要将 `vuex` 的 `store` 对象分割成模块 (`modules`)。

如果请求来的数据不是要被其他组件公用，仅仅在请求的组件内使用，就不需要放入 vuex 的 state 里如果被其他地方复用，请将请求放入 action 里，方便复用，并包装成 promise 返回

#62 如何让CSS只在当前组件中起作用？

将当前组件的 `<style>` 修改为 `<style scoped>`

#63 delete和Vue.delete删除数组的区别？

- `delete` 只是被删除的元素变成了 `empty/undefined` 其他的元素的键值还是不变。
- `Vue.delete` 直接删除了数组 改变了数组的键值。

```
var a=[1,2,3,4]
var b=[1,2,3,4]
delete a[0]
console.log(a) // [empty,2,3,4]
this.$delete(b,0)
console.log(b) // [2,3,4]
```

#64 v-on可以监听多个方法吗？

可以

```
<input type="text" :value="name" @input="onInput" @focus="onFocus"
@blur="onBlur" />
```

v-on 常用修饰符

- `.stop` 该修饰符将阻止事件向上冒泡。同理于调用 `event.stopPropagation()` 方法
- `.prevent` 该修饰符会阻止当前事件的默认行为。同理于调用 `event.preventDefault()` 方法
- `.self` 该指令只当事件是從事件绑定的元素本身触发时才触发回调
- `.once` 该修饰符表示绑定的事件只会被触发一次

#65 Vue子组件调用父组件的方法

- 第一种方法是直接在子组件中通过 `this.$parent.event` 来调用父组件的方法
- 第二种方法是在子组件里用 `$emit` 向父组件触发一个事件，父组件监听这个事件就行了。

#66 vue如何兼容ie的问题

babel-polyfill插件

#67 Vue 改变数组触发视图更新

以下方法调用会改变原始数组: `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `sort()`,
`reverse()`, `Vue.set(target, key, value)`

- 调用方法:

```
Vue.set( target, key, value )
```

- `target` : 要更改的数据源(可以是对象或者数组)
- `key` : 要更改的具体数据
- `value` : 重新赋的值

#68 DOM 渲染在哪个周期中就已经完成?

在 `mounted`

注意 `mounted` 不会承诺所有的子组件也都一起被挂载。如果你希望等到整个视图都渲染完毕，可以用 `vm.$nextTick` 替换掉 `mounted`

```
mounted: function () {
  this.$nextTick(function () {
    // Code that will run only after the
    // entire view has been rendered
  })
}
```

#69 简述每个周期具体适合哪些场景

- `beforeCreate` : 可以在这加个 `loading` 事件，在加载实例时触发
- `created` : 初始化完成时的事件写在这里，如在这结束 `loading` 事件，异步请求也适宜在这里调用
- `mounted` : 挂载元素，获取到DOM节点
- `updated` : 如果对数据统一处理，在这里写上相应函数
- `beforeDestroy` : 可以做一个确认停止事件的确认框

第一次加载会触发哪几个钩子

会触发 `beforeCreate`, `created`, `beforeMount`, `mounted`

#70 动态绑定class

`active` `classname`, `isActive` 变量

```
<div :class="{ active: isActive }"></div>
```

#八、框架通识

#1 MVVM

`MVVM` 由以下三个内容组成

- `View`: 界面
- `Model`: 数据模型
- `ViewModel`: 作为桥梁负责沟通 `View` 和 `Model`

在 `JQuery` 时期，如果需要刷新 `UI` 时，需要先取到对应的 `DOM` 再更新 `UI`，这样数据和业务的逻辑就和页面有强耦合。

`MVVM`

在 `MVVM` 中，`UI` 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 `UI`，`UI` 如果改变，也会改变对应的数据。这种方式就可以在业务处理中只关心数据的流转，而无需直接和页面打交道。

`ViewModel` 只关心数据和业务的处理，不关心 `View` 如何处理数据，在这种情况下，`View` 和 `Model` 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 `ViewModel` 中，让多个 `View` 复用这个 `ViewModel`。

- 在 `MVVM` 中，最核心的也就是数据双向绑定，例如 `Angular` 的脏数据检测，`Vue` 中的数据劫持。

脏数据检测

当触发了指定事件后会进入脏数据检测，这时会调用 `$digest` 循环遍历所有的数据观察者，判断当前值是否和先前的值有区别，如果检测到变化的话，会调用 `$watch` 函数，然后再次调用 `$digest` 循环直到发现没有变化。循环至少为二次，至多为十次。

脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 `Vue` 中的双向绑定是存在问题的。并且脏数据检测可以实现批量检测出更新的值，再去统一更新 `UI`，大大减少了操作 `DOM` 的次数。所以低效也是相对的，这就仁者见仁智者见智了。

数据劫持

`Vue` 内部使用了 `Object.defineProperty()` 来实现双向绑定，通过这个函数可以监听到 `set` 和 `get` 的事件。

```
var data = { name: 'yck' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(obj).forEach(key => {
    defineReactive(obj, key, obj[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}
```

以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的，还需要在适当的时候给属性添加发布订阅。

```
<div>
  {{name}}
</div>
```

在解析如上模板代码时，遇到 `{name}` 就会给属性 `name` 添加发布订阅。

```

// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
  addSub(sub) {
    // sub 是 watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 watcher
Dep.target = null

function update(value) {
  document.querySelector('div').innerText = value
}

class watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
var data = { name: 'yck' }
observe(data)
// 模拟解析到 `{{name}}` 触发的操作
new watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'

```

接下来，对 `defineReactive` 函数进行改造

```

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get() {
      return val
    },
    set(newVal) {
      val = newVal
      dp.notify()
    }
  })
}

```

```

get: function reactiveGetter() {
  console.log('get value')
  // 将 watcher 添加到订阅
  if (Dep.target) {
    dp.addSub(Dep.target)
  }
  return val
},
set: function reactiveSetter(newVal) {
  console.log('change value')
  val = newVal
  // 执行 watcher 的 update 方法
  dp.notify()
}
}
}

```

以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 `getter` 来实现发布订阅的添加

Proxy 与 `Object.defineProperty` 对比

`Object.defineProperty` 虽然已经能够实现双向绑定了，但是他还是有缺陷的。

- 只能对属性进行数据劫持，所以需要深度遍历整个对象 对于数组不能监听到数据的变化
- 虽然 `vue` 中确实能检测到数组数据的变化，但是其实是使用了 `hack` 的办法，并且也是有缺陷的。

```

const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// hack 以下几个函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 获得原生函数
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    // 调用原生函数
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
  })
})

```

```

    if (inserted) ob.observeArray(inserted)
    // 触发更新
    ob.dep.notify()
    return result
  })
}

```

反观 `Proxy` 就没以上的问题，原生支持监听数组变化，并且可以直接对整个对象进行拦截，所以 `Vue` 也将在下个大版本中使用 `Proxy` 替换 `Object.defineProperty`

```

let onwatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};

let obj = { a: 1 }
let value
let p = onwatch(obj, (v) => {
  value = v
}, (target, property) => {
  console.log(`Get '${property}' = ${target[property]}`);
})
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2

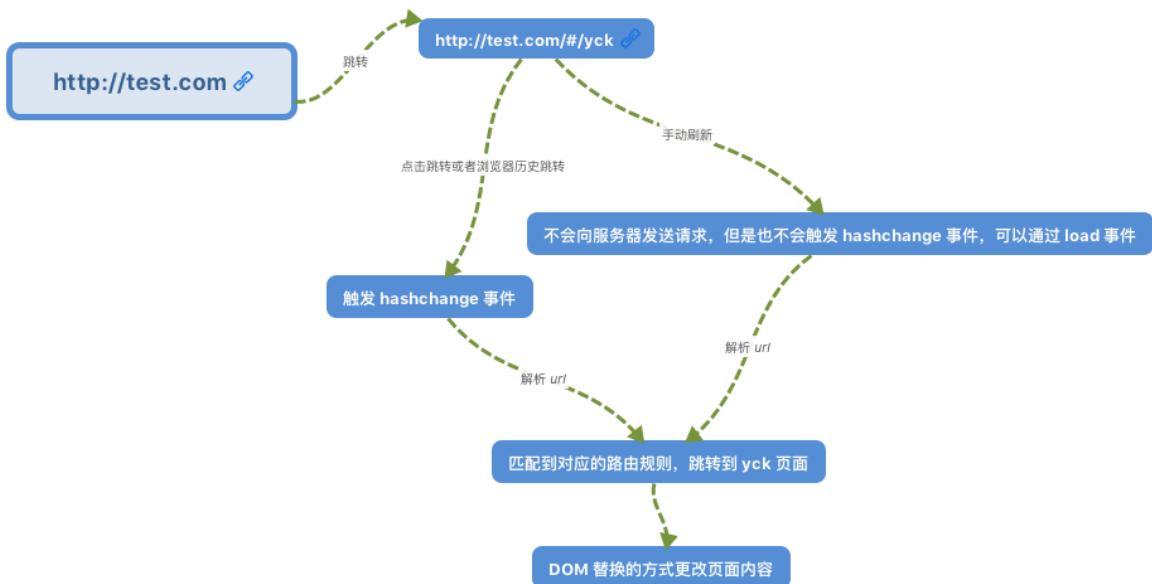
```

#2 路由原理

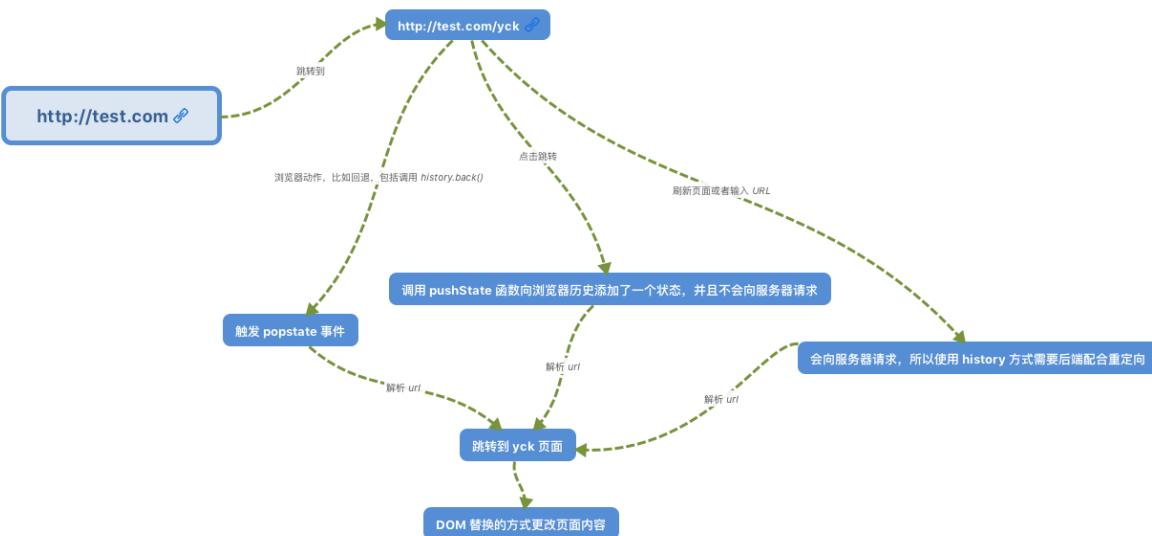
前端路由实现起来其实很简单，本质就是监听 `URL` 的变化，然后匹配路由规则，显示相应的页面，并且无须刷新。目前单页面使用的路由就只有两种实现方式

- `hash` 模式
- `history` 模式

`www.test.com/#/` 就是 `Hash URL`，当 `##` 后面的哈希值发生变化时，不会向服务器请求数据，可以通过 `hashchange` 事件来监听到 `URL` 的变化，从而进行跳转页面。



History 模式是 HTML5 新推出的功能，比之 Hash URL 更加美观



#3 Virtual Dom

为什么需要 Virtual Dom

众所周知，操作 DOM 是很耗费性能的一件事情，既然如此，我们可以考虑通过 JS 对象来模拟 DOM 对象，毕竟操作 JS 对象比操作 DOM 省时的多

```
// 假设这里模拟一个 ul，其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

从上述例子中，我们一眼就可以看出先前的 `ul` 中的第三个 `li` 被移除了，四五替换了位置。

- 如果以上操作对应到 DOM 中，那么就是以下代码

```

// 删除第三个 li
ul.childNodes[2].remove()
// 将第四个 li 和第五个交换位置
let fromNode = ul.childNodes[4]
let toNode = node.childNodes[3]
let cloneFromNode = fromNode.cloneNode(true)
let cloenToNode = toNode.cloneNode(true)
ul.replaceChild(cloneFromNode, toNode)
ul.replaceChild(cloenToNode, fromNode)

```

当然在实际操作中，我们还需要给每个节点一个标识，作为判断是同一个节点的依据。所以这也是 `Vue` 和 `React` 中官方推荐列表里的节点使用唯一的 `key` 来保证性能。

- 那么既然 `DOM` 对象可以通过 `JS` 对象来模拟，反之也可以通过 `JS` 对象来渲染出对应的 `DOM`
- 以下是一个 `JS` 对象模拟 `DOM` 对象的简单实现

```

export default class Element {
    /**
     * @param {String} tag 'div'
     * @param {Object} props { class: 'item' }
     * @param {Array} children [ Element1, 'text' ]
     * @param {String} key option
     */
    constructor(tag, props, children, key) {
        this.tag = tag
        this.props = props
        if (Array.isArray(children)) {
            this.children = children
        } else if (isString(children)) {
            this.key = children
            this.children = null
        }
        if (key) this.key = key
    }
    // 渲染
    render() {
        let root = this._createElement(
            this.tag,
            this.props,
            this.children,
            this.key
        )
        document.body.appendChild(root)
        return root
    }
    create() {
        return this._createElement(this.tag, this.props, this.children, this.key)
    }
    // 创建节点
    _createElement(tag, props, child, key) {
        // 通过 tag 创建节点
        let el = document.createElement(tag)
        // 设置节点属性
        for (const key in props) {
            if (props.hasOwnProperty(key)) {
                const value = props[key]

```

```

        e1.setAttribute(key, value)
    }
}
if (key) {
    e1.setAttribute('key', key)
}
// 递归添加子节点
if (child) {
    child.forEach(element => {
        let child
        if (element instanceof Element) {
            child = this._createElement(
                element.tag,
                element.props,
                element.children,
                element.key
            )
        } else {
            child = document.createTextNode(element)
        }
        e1.appendChild(child)
    })
}
return e1
}
}

```

Virtual Dom 算法简述

- 既然我们已经通过 JS 来模拟实现了 DOM，那么接下来的难点就在于如何判断旧的对象和新的对象之间的差异。
- DOM 是多叉树的结构，如果需要完整的对比两颗树的差异，那么需要的时间复杂度会是 $O(n^3)$ ，这个复杂度肯定是不能接受的。于是 React 团队优化了算法，实现了 $O(n)$ 的复杂度来对比差异。
- 实现 $O(n)$ 复杂度的关键就是只对比同层的节点，而不是跨层对比，这也是考虑到在实际业务中很少会去跨层的移动 DOM 元素

所以判断差异的算法就分为了两步

- 首先从上至下，从左往右遍历对象，也就是树的深度遍历，这一步中会给每个节点添加索引，便于最后渲染差异
- 一旦节点有子元素，就去判断子元素是否有不同

Virtual Dom 算法实现

树的递归

- 首先我们来实现树的递归算法，在实现该算法前，先来考虑下两个节点对比会有几种情况
- 新的节点的 `tagName` 或者 `key` 和旧的不同，这种情况代表需要替换旧的节点，并且也不再需要遍历新旧节点的子元素了，因为整个旧节点都被删掉了
- 新的节点的 `tagName` 和 `key`（可能都没有）和旧的相同，开始遍历子树
- 没有新的节点，那么什么都不用做

```

import { StateEnums, isString, move } from './util'
import Element from './element'

export default function diff(oldDomTree, newDomTree) {

```

```

// 用于记录差异
let pathchs = []
// 一开始的索引为 0
dfs(oldDomTree, newDomTree, 0, pathchs)
return pathchs
}

function dfs(oldNode, newNode, index, patches) {
// 用于保存子树的更改
let curPatches = []
// 需要判断三种情况
// 1.没有新的节点，那么什么都不用做
// 2.新的节点的 tagName 和 `key` 和旧的不同，就替换
// 3.新的节点的 tagName 和 key（可能都没有）和旧的相同，开始遍历子树
if (!newNode) {
} else if (newNode.tag === oldNode.tag && newNode.key === oldNode.key) {
// 判断属性是否变更
let props = diffProps(oldNode.props, newNode.props)
if (props.length) curPatches.push({ type: StateEnums.ChangeProps, props })
// 遍历子树
diffChildren(oldNode.children, newNode.children, index, patches)
} else {
// 节点不同，需要替换
curPatches.push({ type: StateEnums.Replace, node: newNode })
}

if (curPatches.length) {
if (patches[index]) {
patches[index] = patches[index].concat(curPatches)
} else {
patches[index] = curPatches
}
}
}
}

```

判断属性的更改

判断属性的更改也分三个步骤

- 遍历旧的属性列表，查看每个属性是否还存在于新的属性列表中
- 遍历新的属性列表，判断两个列表中都存在的属性的值是否有变化
- 在第二步中同时查看是否有属性不存在与旧的属性列表中

```

function diffProps(oldProps, newProps) {
// 判断 Props 分以下三步骤
// 先遍历 oldProps 查看是否存在删除的属性
// 然后遍历 newProps 查看是否有属性值被修改
// 最后查看是否有属性新增
let change = []
for (const key in oldProps) {
if (oldProps.hasOwnProperty(key) && !newProps[key]) {
change.push({
prop: key
})
}
}
for (const key in newProps) {

```

```

if (newProps.hasOwnProperty(key)) {
  const prop = newProps[key]
  if (oldProps[key] && oldProps[key] !== newProps[key]) {
    change.push({
      prop: key,
      value: newProps[key]
    })
  } else if (!oldProps[key]) {
    change.push({
      prop: key,
      value: newProps[key]
    })
  }
}
}

return change
}

```

判断列表差异算法实现

这个算法是整个 `virtual dom` 中最核心的算法，且让我——为你道来。这里的主要步骤其实和判断属性差异是类似的，也是分为三步

- 遍历旧的节点列表，查看每个节点是否还存在于新的节点列表中
- 遍历新的节点列表，判断是否有新的节点
- 在第二步中同时判断节点是否有移动

PS：该算法只对有 `key` 的节点做处理

```

function listDiff(oldList, newList, index, patches) {
  // 为了遍历方便，先取出两个 list 的所有 keys
  let oldKeys = getKeys(oldList)
  let newKeys = getKeys(newList)
  let changes = []

  // 用于保存变更后的节点数据
  // 使用该数组保存有以下好处
  // 1.可以正确获得被删除节点索引
  // 2.交换节点位置只需要操作一遍 DOM
  // 3.用于 `diffchildren` 函数中的判断，只需要遍历
  // 两个树中都存在的节点，而对于新增或者删除的节点来说，完全没必要
  // 再去判断一遍
  let list = []
  oldList &&
    oldList.forEach(item => {
      let key = item.key
      if (isString(item)) {
        key = item
      }
      // 寻找新的 children 中是否含有当前节点
      // 没有的话需要删除
      let index = newKeys.indexOf(key)
      if (index === -1) {
        list.push(null)
      } else list.push(key)
    })
  // 遍历变更后的数组

```

```

let length = list.length
// 因为删除数组元素是会更改索引的
// 所有从后往前删可以保证索引不变
for (let i = length - 1; i >= 0; i--) {
    // 判断当前元素是否为空，为空表示需要删除
    if (!list[i]) {
        list.splice(i, 1)
        changes.push({
            type: StateEnums.Remove,
            index: i
        })
    }
}

// 遍历新的 list，判断是否有节点新增或移动
// 同时也对 `list` 做节点新增和移动节点的操作
newList &&
newList.forEach((item, i) => {
    let key = item.key
    if (isString(item)) {
        key = item
    }

    // 寻找旧的 children 中是否含有当前节点
    let index = list.indexOf(key)
    // 没找到代表新节点，需要插入
    if (index === -1 || key == null) {
        changes.push({
            type: StateEnums.Insert,
            node: item,
            index: i
        })
        list.splice(i, 0, key)
    } else {
        // 找到了，需要判断是否需要移动
        if (index !== i) {
            changes.push({
                type: StateEnums.Move,
                from: index,
                to: i
            })
            move(list, index, i)
        }
    }
})
return { changes, list }
}

function getKeys(list) {
    let keys = []
    let text
    list &&
    list.forEach(item => {
        let key
        if (isString(item)) {
            key = [item]
        } else if (item instanceof Element) {
            key = item.key
        }
        keys.push(key)
    })
}

```

```
        }
    return keys
}
```

遍历子元素打标识

对于这个函数来说，主要功能就两个

- 判断两个列表差异
 - 给节点打上标记
 - 总体来说，该函数实现的功能很简单

```
function diffChildren(oldChild, newChild, index, patches) {
let { changes, list } = listDiff(oldChild, newChild, index, patches)
if (changes.length) {
    if (patches[index]) {
        patches[index] = patches[index].concat(changes)
    } else {
        patches[index] = changes
    }
}
// 记录上一个遍历过的节点
let last = null
oldChild &&
oldChild.forEach((item, i) => {
    let child = item && item.children
    if (child) {
        index =
            last && last.children ? index + last.children.length + 1 : index + 1
        let keyIndex = list.indexOf(item.key)
        let node = newChild[keyIndex]
        // 只遍历新旧中都存在的节点，其他新增或者删除的没必要遍历
        if (node) {
            dfs(item, node, index, patches)
        }
    } else index += 1
    last = item
})
}
```

渲染差异

通过之前的算法，我们已经可以得出两个树的差异了。既然知道了差异，就需要局部去更新 DOM 了，下面就让我们来看看 virtual Dom 算法的最后一步骤

这个函数主要两个功能

- 深度遍历树，将需要做变更操作的取出来
- 局部更新 DOM

```
let index = 0
export default function patch(node, patchs) {
    let changes = patchs[index]
    let childNodes = node && node.childNodes
    // 这里的深度遍历和 diff 中是一样的
    if (!childNodes) index += 1
    if (changes && changes.length && patchs[index]) {
```

```

changeDom(node, changes)
}

let last = null
if (childNodes && childNodes.length) {
  childNodes.forEach((item, i) => {
    index =
      last && last.children ? index + last.children.length + 1 : index + 1
    patch(item, patchs)
    last = item
  })
}
}

function changeDom(node, changes, nochild) {
  changes &&
  changes.forEach(change => {
    let { type } = change
    switch (type) {
      case StateEnums.ChangeProps:
        let { props } = change
        props.forEach(item => {
          if (item.value) {
            node.setAttribute(item.prop, item.value)
          } else {
            node.removeAttribute(item.prop)
          }
        })
        break
      case StateEnums.Remove:
        node.childNodes[change.index].remove()
        break
      case StateEnums.Insert:
        let dom
        if (isString(change.node)) {
          dom = document.createTextNode(change.node)
        } else if (change.node instanceof Element) {
          dom = change.node.create()
        }
        node.insertBefore(dom, node.childNodes[change.index])
        break
      case StateEnums.Replace:
        node.parentNode.replaceChild(change.node.create(), node)
        break
      case StateEnums.Move:
        let fromNode = node.childNodes[change.from]
        let toNode = node.childNodes[change.to]
        let cloneFromNode = fromNode.cloneNode(true)
        let cloenToNode = toNode.cloneNode(true)
        node.replaceChild(cloneFromNode, toNode)
        node.replaceChild(cloenToNode, fromNode)
        break
      default:
        break
    }
  })
}
}

```

Virtual Dom 算法的实现也就是以下三步

- 通过 `JS` 来模拟创建 `DOM` 对象
- 判断两个对象的差异
- 渲染差异

```
let test4 = new Element('div', { class: 'my-div' }, ['test4'])
let test5 = new Element('ul', { class: 'my-div' }, ['test5'])

let test1 = new Element('div', { class: 'my-div' }, [test4])

let test2 = new Element('div', { id: '11' }, [test5, test4])

let root = test1.render()

let patchchs = diff(test1, test2)
console.log(patchchs)

setTimeout(() => {
  console.log('开始更新')
  patch(root, patchchs)
  console.log('结束更新')
}, 1000)
```

高频考点

#1 `typeof`类型判断

`typeof` 是否能正确判断类型? `instanceof` 能正确判断对象的原理是什么

- `typeof` 对于原始类型来说, 除了 `null` 都可以显示正确的类型

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
typeof true // 'boolean'
typeof Symbol() // 'symbol'
```

`typeof` 对于对象来说, 除了函数都会显示 `object`, 所以说 `typeof` 并不能准确判断变量到底是什么类型

```
typeof [] // 'object'
typeof {} // 'object'
typeof console.log // 'function'
```

如果我们想判断一个对象的正确类型, 这时候可以考虑使用 `instanceof`, 因为内部机制是通过原型链来判断的

```

const Person = function() {}
const p1 = new Person()
p1 instanceof Person // true

var str = 'hello world'
str instanceof String // false

var str1 = new String('hello world')
str1 instanceof String // true

```

对于原始类型来说，你想直接通过 `instanceof` 来判断类型是不行的

#2 类型转换

首先我们要知道，在 JS 中类型转换只有三种情况，分别是：

- 转换为布尔值
- 转换为数字
- 转换为字符串

| 原始值 | 转换目标 | 结果 |
|--|------|------------------------------|
| <code>number</code> | 布尔值 | 除了 0、-0、NaN 都为 true |
| <code>string</code> | 布尔值 | 除了空串都为 true |
| <code>undefined</code> 、 <code>null</code> | 布尔值 | FALSE |
| 引用类型 | 布尔值 | TRUE |
| <code>number</code> | 字符串 | 5 => '5' |
| <code>Boolean</code> 、 <code>函数</code> 、 <code>Symbol</code> | 字符串 | 'true' |
| 数组 | 字符串 | [1,2] => '1,2' |
| 对象 | 字符串 | '[object Object]' |
| <code>string</code> | 数字 | '1' => 1, 'a' => NaN |
| 数组 | 数字 | 空数组为0，存在一个元素且为数字转数字，其他情况 NaN |
| <code>null</code> | 数字 | 0 |
| 除了数组的引用类型 | 数字 | NaN |
| <code>Symbol</code> | 数字 | 抛错 |

转 Boolean

在条件判断时，除了 `undefined`、`null`、`false`、`NaN`、`''`、`0`、`-0`，其他所有值都转为 `true`，包括所有对象

对象转原始类型

对象在转换类型的时候，会调用内置的 `[[ToPrimitive]]` 函数，对于该函数来说，算法逻辑一般说来如下

- 如果已经是原始类型了，那就不需要转换了
- 调用 `x.valueOf()`，如果转换为基础类型，就返回转换的值

- 调用 `x.toString()`，如果转换为基础类型，就返回转换的值
- 如果都没有返回原始类型，就会报错

当然你也可以重写 `Symbol.toPrimitive`，该方法在转原始类型时调用优先级最高。

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  },
  [Symbol.toPrimitive]() {
    return 2
  }
}
1 + a // => 3
```

四则运算符

它有以下几个特点：

- 运算中其中一方为字符串，那么就会把另一方也转换为字符串
- 如果一方不是字符串或者数字，那么会将它转换为数字或者字符串

```
1 + '1' // '11'
true + true // 2
4 + [1,2,3] // "41,2,3"
```

- 对于第一行代码来说，触发特点一，所以将数字 `1` 转换为字符串，得到结果 `'11'`
- 对于第二行代码来说，触发特点二，所以将 `true` 转为数字 `1`
- 对于第三行代码来说，触发特点二，所以将数组通过 `toString` 转为字符串 `1,2,3`，得到结果 `41,2,3`

另外对于加法还需要注意这个表达式 `'a' + + 'b'`

```
'a' + + 'b' // -> "aNan"
```

- 因为 `+ 'b'` 等于 `Nan`，所以结果为 `"aNan"`，你可能也会在一些代码中看到过 `+ '1'` 的形式来快速获取 `number` 类型。
- 那么对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字

```
4 * '3' // 12
4 * [] // 0
4 * [1, 2] // NaN
```

比较运算符

- 如果是对象，就通过 `toPrimitive` 转换对象
- 如果是字符串，就通过 `unicode` 字符索引来比较

```
let a = {
  valueOf() {
    return 0
  },
  toString() {
    return '1'
  }
}
a > -1 // true
```

在以上代码中，因为 `a` 是对象，所以会通过 `valueOf` 转换为原始类型再比较值。

#3 This

我们先来看几个函数调用的场景

```
function foo() {
  console.log(this.a)
}

var a = 1
foo()

const obj = {
  a: 2,
  foo: foo
}
obj.foo()

const c = new foo()
```

- 对于直接调用 `foo` 来说，不管 `foo` 函数被放在了什么地方，`this` 一定是 `window`
- 对于 `obj.foo()` 来说，我们只需要记住，谁调用了函数，谁就是 `this`，所以在这个场景下 `foo` 函数中的 `this` 就是 `obj` 对象
- 对于 `new` 的方式来说，`this` 被永远绑定在了 `c` 上面，不会被任何方式改变 `this`

说完了以上几种情况，其实很多代码中的 `this` 应该就没什么问题了，下面让我们看看箭头函数中的 `this`

```
function a() {
  return () => {
    return () => {
      console.log(this)
    }
  }
}
console.log(a())
```

- 首先箭头函数其实是没有 `this` 的，箭头函数中的 `this` 只取决于包裹箭头函数的第一个普通函数的 `this`。在这个例子中，因为包裹箭头函数的第一个普通函数是 `a`，所以此时的 `this` 是 `window`。另外对箭头函数使用 `bind` 这类函数是无效的。
- 最后种情况也就是 `bind` 这些改变上下文的 API 了，对于这些函数来说，`this` 取决于第一个参数，如果第一个参数为空，那么就是 `window`。

- 那么说到 `bind`, 不知道大家是否考虑过, 如果对一个函数进行多次 `bind`, 那么上下文会是什么呢?

```
let a = []
let fn = function () { console.log(this) }
fn.bind().bind(a) // => ?
```

如果你认为输出结果是 `a`, 那么你就错了, 其实我们可以把上述代码转换成另一种形式

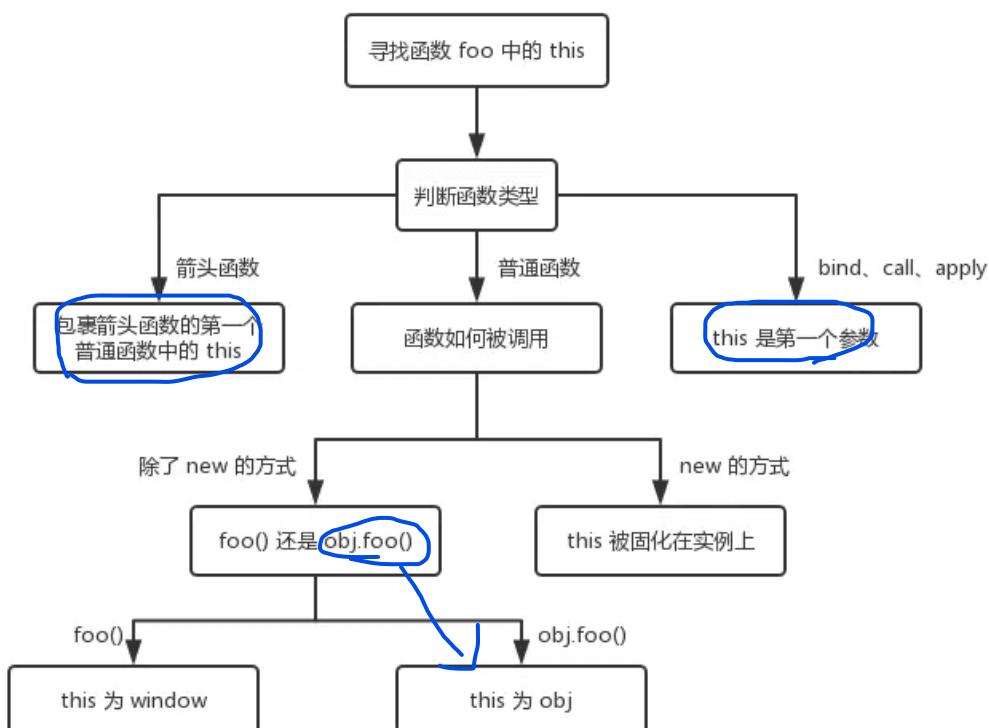
```
// fn.bind().bind(a) 等于
let fn2 = function fn1() {
  return function() {
    return fn.apply()
  }.apply(a)
}
fn2()
```

可以从上述代码中发现, 不管我们给函数 `bind` 几次, `fn` 中的 `this` 永远由第一次 `bind` 决定, 所以结果永远是 `window`

```
let a = { name: 'poetries' }
function foo() {
  console.log(this.name)
}
foo.bind(a) // => 'poetries'
```

以上就是 `this` 的规则了, 但是可能会发生多个规则同时出现的情况, 这时候不同的规则之间会根据优先级最高的来决定 `this` 最终指向哪里。

首先, `new` 的方式优先级最高, 接下来是 `bind` 这些函数, 然后是 `obj.foo()` 这种调用方式, 最后是 `foo` 这种调用方式, 同时, 箭头函数的 `this` 一旦被绑定, 就不会再被任何方式所改变。



#4 == 和 === 有什么区别

对于 `==` 来说，如果对比双方的类型不一样的话，就会进行类型转换

假如我们需要对比 `x` 和 `y` 是否相同，就会进行如下判断流程

- 首先会判断两者类型是否相同。相同的话就是比大小了
- 类型不相同的话，那么就会进行类型转换
- 会先判断是否在对比 `null` 和 `undefined`，是的话就会返回 `true`
- 判断两者类型是否为 `string` 和 `number`，是的话就会将字符串转换为 `number`

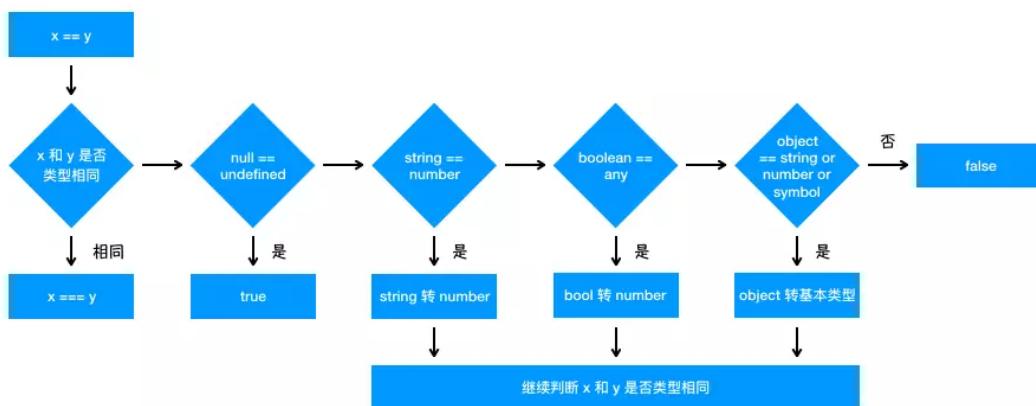
```
1 == '1'  
↓  
1 == 1
```

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

```
'1' == true  
↓  
'1' == 1  
↓  
1 == 1
```

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

```
'1' == { name: 'yck' }  
↓  
'1' == '[object Object]'
```



对于 `===` 来说就简单多了，就是判断两者类型和值是否相同

#5 闭包

闭包的定义其实很简单：函数 `A` 内部有一个函数 `B`，函数 `B` 可以访问到函数 `A` 中的变量，那么函数 `B` 就是闭包

```
function A() {
  let a = 1
  window.B = function () {
    console.log(a)
  }
}
A()
B() // 1
```

闭包存在的意义就是让我们可以间接访问函数内部的变量

经典面试题，循环中使用闭包解决 `var` 定义函数的问题

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

首先因为 `setTimeout` 是个异步函数，所以会先把循环全部执行完毕，这时候 `i` 就是 6 了，所以会输出一堆 6

解决办法有三种

1. 第一种是使用闭包的方式

```
for (var i = 1; i <= 5; i++) {
  ;(function(j) {
    setTimeout(function timer() {
      console.log(j)
    }, j * 1000)
  })(i)
}
```

在上述代码中，我们首先使用了立即执行函数将 `i` 传入函数内部，这个时候值就被固定在了参数 `j` 上面不会改变，当下次执行 `timer` 这个闭包的时候，就可以使用外部函数的变量 `j`，从而达到目的

1. 第二种就是使用 `setTimeout` 的第三个参数，这个参数会被当成 `timer` 函数的参数传入

```
for (var i = 1; i <= 5; i++) {
  setTimeout(
    function timer(j) {
      console.log(j)
    },
    i * 1000,
    i
  )
}
```

1. 第三种就是使用 `let` 定义 `i` 了来解决问题了，这个也是最为推荐的方式

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

#6 深浅拷贝

浅拷贝

首先可以通过 `Object.assign` 来解决这个问题，很多人认为这个函数是用来深拷贝的。其实并不是，`Object.assign` 只会拷贝所有的属性值到新的对象中，如果属性值是对象的话，拷贝的是地址，所以并不是深拷贝

```
let a = {
  age: 1
}
let b = Object.assign({}, a)
a.age = 2
console.log(b.age) // 1
```

另外我们还可以通过展开运算符 `...` 来实现浅拷贝

```
let a = {
  age: 1
}
let b = { ...a }
a.age = 2
console.log(b.age) // 1
```

通常浅拷贝就能解决大部分问题了，但是当我们遇到如下情况就可能需要使用到深拷贝了

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = { ...a }
a.jobs.first = 'native'
console.log(b.jobs.first) // native
```

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，那么就又回到最开始的话题了，两者享有相同的地址。要解决这个问题，我们就得使用深拷贝了。

深拷贝

这个问题通常可以通过 `JSON.parse(JSON.stringify(object))` 来解决。

```
let a = {
  age: 1,
  jobs: {
    first: 'FE'
  }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

但是该方法也是有局限性的：

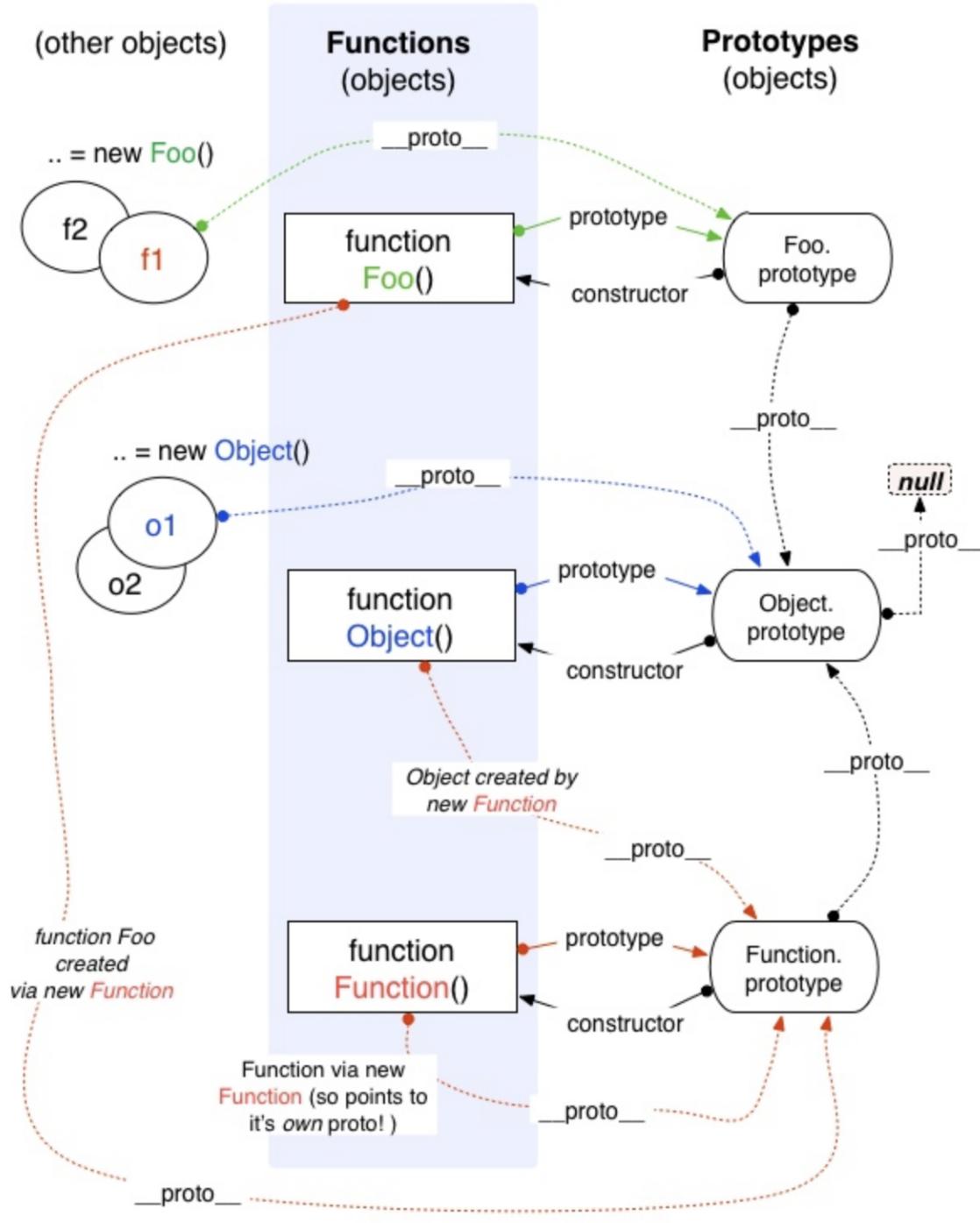
- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能序列化函数
- 不能解决循环引用的对象

```
let obj = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  },
}
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)
```

更多详情 <https://www.jianshu.com/p/2d8a26b3958f>

#7 原型

原型链就是多个对象通过 `__proto__` 的方式连接了起来。为什么 `obj` 可以访问到 `valueOf` 函数，就是因为 `obj` 通过原型链找到了 `valueOf` 函数



- `Object` 是所有对象的爸爸，所有对象都可以通过 `__proto__` 找到它
- `Function` 是所有函数的爸爸，所有函数都可以通过 `__proto__` 找到它
- 函数的 `prototype` 是一个对象
- 对象的 `__proto__` 属性指向原型，`__proto__` 将对象和原型连接起来组成了原型链

#8 var、let 及 const 区别

涉及面试题：什么是提升？什么是暂时性死区？var、let 及 const 区别？

- 函数提升优先于变量提升，函数提升会把整个函数挪到作用域顶部，变量提升只会把声明挪到作用域顶部
- `var` 存在提升，我们能在声明之前使用。`let`、`const` 因为暂时性死区的原因，不能在声明前使用

- `var` 在全局作用域下声明变量会导致变量挂载在 `window` 上，其他两者不会
- `let` 和 `const` 作用基本一致，但是后者声明的变量不能再次赋值

#9 原型继承和 Class 继承

涉及面试题：原型如何实现继承？`class` 如何实现继承？`class` 本质是什么？

首先先来讲下 `class`，其实在 JS 中并不存在类，`class` 只是语法糖，本质还是函数

```
class Person {}  
Person instanceof Function // true
```

组合继承

组合继承是最常用的继承方式

```
function Parent(value) {  
    this.val = value  
}  
Parent.prototype.getValue = function() {  
    console.log(this.val)  
}  
function Child(value) {  
    Parent.call(this, value)  
}  
Child.prototype = new Parent()  
  
const child = new Child(1)  
  
child.getValue() // 1  
child instanceof Parent // true
```

- 以上继承的方式核心是在子类的构造函数中通过 `Parent.call(this)` 继承父类的属性，然后改变子类的原型为 `new Parent()` 来继承父类的函数。
- 这种继承方式优点在于构造函数可以传参，不会与父类引用属性共享，可以复用父类的函数，但是也存在一个缺点就是在继承父类函数的时候调用了父类构造函数，导致子类的原型上多了不需要的父类属性，存在内存上的浪费

寄生组合继承

这种继承方式对组合继承进行了优化，组合继承缺点在于继承父类函数时调用了构造函数，我们只需要优化掉这点就行了

```
function Parent(value) {  
    this.val = value  
}  
Parent.prototype.getValue = function() {  
    console.log(this.val)  
}  
  
function Child(value) {  
    Parent.call(this, value)  
}  
Child.prototype = Object.create(Parent.prototype, {
```

```
constructor: {
  value: Child,
  enumerable: false,
  writable: true,
  configurable: true
}
})

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

以上继承实现的核心就是将父类的原型赋值给了子类，并且将构造函数设置为子类，这样既解决了无用的父类属性问题，还能正确的找到子类的构造函数。

Class 继承

以上两种继承方式都是通过原型去解决的，在ES6中，我们可以使用class去实现继承，并且实现起来很简单

```
class Parent {
  constructor(value) {
    this.val = value
  }
  getValue() {
    console.log(this.val)
  }
}
class Child extends Parent {
  constructor(value) {
    super(value)
    this.val = value
  }
}
let child = new Child(1)
child.getValue() // 1
child instanceof Parent // true
```

class实现继承的核心在于使用extends表明继承自哪个父类，并且在子类构造函数中必须调用super，因为这段代码可以看成Parent.call(this, value)。

#10 模块化

涉及面试题：为什么要使用模块化？都有哪几种方式可以实现模块化，各有什么特点？

使用一个技术肯定是有原因的，那么使用模块化可以给我们带来以下好处

- 解决命名冲突
- 提供复用性
- 提高代码可维护性

立即执行函数

在早期，使用立即执行函数实现模块化是常见的手段，通过函数作用域解决了命名冲突、污染全局作用域的问题

```
(function(globalvariable){  
    globalvariable.test = function() {}  
    // ... 声明各种变量、函数都不会污染全局作用域  
})(globalvariable)
```

AMD 和 CMD

鉴于目前这两种实现方式已经很少见到，所以不再对具体特性细聊，只需要了解这两者是如何使用的。

```
// AMD  
define(['./a', './b'], function(a, b) {  
    // 加载模块完毕可以使用  
    a.do()  
    b.do()  
})  
  
// CMD  
define(function(require, exports, module) {  
    // 加载模块  
    // 可以把 require 写在函数体的任意地方实现延迟加载  
    var a = require('./a')  
    a.doSomething()  
})
```

CommonJS

CommonJS 最早是 Node 在使用，目前也仍然广泛使用，比如在 webpack 中你就能见到它，当然目前在 Node 中的模块管理已经和 CommonJS 有一些区别了

```
// a.js  
module.exports = {  
    a: 1  
}  
// or  
exports.a = 1  
  
// b.js  
var module = require('./a.js')  
module.a // -> log 1  
var module = require('./a.js')  
module.a  
// 这里其实就是包装了一层立即执行函数，这样就不会污染全局变量了，  
// 重要的是 module 这里，module 是 Node 独有的一个变量  
module.exports = {  
    a: 1  
}  
// module 基本实现  
var module = {  
    id: 'xxxx', // 我总得知道怎么去找到他吧  
    exports: {} // exports 就是个空对象  
}  
// 这个是因为 exports 和 module.exports 用法相似的原因  
var exports = module.exports  
var load = function (module) {  
    // 导出的东西  
    var a = 1
```

```
module.exports = a
return module.exports
};

// 然后当我 require 的时候去找到独特的
// id, 然后将要使用的东西用立即执行函数包装下, over
```

另外虽然 `exports` 和 `module.exports` 用法相似，但是不能对 `exports` 直接赋值。因为 `var exports = module.exports` 这句代码表明了 `exports` 和 `module.exports` 享有相同地址，通过改变对象的属性值会对两者都起效，但是如果直接对 `exports` 赋值就会导致两者不再指向同一个内存地址，修改并不会对 `module.exports` 起效

ES Module

`ES Module` 是原生实现的模块化方案，与 `CommonJS` 有以下几个区别

1. `CommonJS` 支持动态导入，也就是 `require(${path}/xx.js)`，后者目前不支持，但是已有提案
2. `CommonJS` 是同步导入，因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。而后者是异步导入，因为用于浏览器，需要下载文件，如果也采用同步导入会对渲染有很大影响
3. `CommonJS` 在导出时都是值拷贝，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。但是 `ES Module` 采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
4. `ES Module` 会编译成 `require,exports` 来执行的

```
// 引入模块 API
import XXX from './a.js'
import { XXX } from './a.js'

// 导出模块 API
export function a() {}
export default function() {}
```

#11 实现一个简洁版的promise

```
// 三个常量用于表示状态
const PENDING = 'pending'
const RESOLVED = 'resolved'
const REJECTED = 'rejected'

function MyPromise(fn) {
    const that = this
    this.state = PENDING

    // value 变量用于保存 resolve 或者 reject 中传入的值
    this.value = null

    // 用于保存 then 中的回调，因为当执行完 Promise 时状态可能还是等待中，这时候应该把 then 中的回调保存起来用于状态改变时使用
    that.resolvedCallbacks = []
    that.rejectedCallbacks = []

    function resolve(value) {
        // 首先两个函数都得判断当前状态是否为等待中
        if(that.state === PENDING) {
```

```

        that.state = RESOLVED
        that.value = value

        // 遍历回调数组并执行
        that.resolvedCallbacks.map(cb=>cb(that.value))
    }
}

function reject(value) {
    if(that.state === PENDING) {
        that.state = REJECTED
        that.value = value
        that.rejectedCallbacks.map(cb=>cb(that.value))
    }
}

// 完成以上两个函数以后，我们就该实现如何执行 Promise 中传入的函数了
try {
    fn(resolve,reject)
}catch(e){
    reject(e)
}
}

// 最后我们来实现较为复杂的 then 函数
MyPromise.prototype.then = function(onFulfilled,onRejected){
    const that = this

    // 判断两个参数是否为函数类型，因为这两个参数是可选参数
    onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v=>v
    onRejected = typeof onRejected === 'function' ? onRejected : e=>throw e

    // 当状态不是等待态时，就去执行相对应的函数。如果状态是等待态的话，就往回调函数中 push 函数
    if(this.state === PENDING) {
        this.resolvedCallbacks.push(onFulfilled)
        this.rejectedCallbacks.push(onRejected)
    }
    if(this.state === RESOLVED) {
        onFulfilled(that.value)
    }
    if(this.state === REJECTED) {
        onRejected(that.value)
    }
}
}

```

#12 Event Loop

#12.1 进程与线程

涉及面试题：进程与线程区别？JS 单线程带来的好处？

- JS 是单线程执行的，但是你是否疑惑过什么是线程？
- 讲到线程，那么肯定也得说一下进程。本质上来说，两个名词都是 CPU 工作时间片的一个描述。
- 进程描述了 CPU 在运行指令及加载和保存上下文所需的时间，放在应用上来说就代表了一个程序。线程是进程中的更小单位，描述了执行一段指令所需的时间

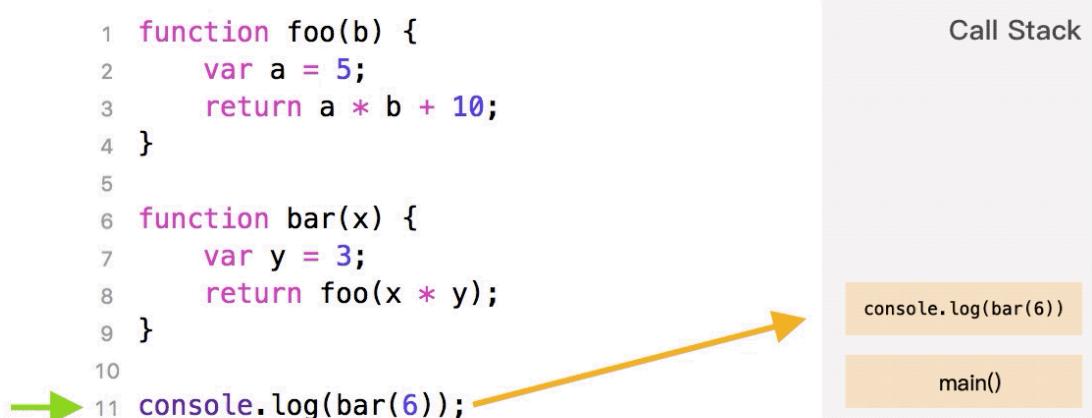
把这些概念拿到浏览器中来说，当你打开一个 Tab 页时，其实就是一个进程，一个进程中可以有多个线程，比如渲染线程、JS 引擎线程、HTTP 请求线程等等。当你发起一个请求时，其实就是一个线程，当请求结束后，该线程可能就会被销毁

- 上文说到了 JS 引擎线程和渲染线程，大家应该都知道，在 JS 运行的时候可能会阻止 UI 渲染，这说明了两个线程是互斥的。这其中的原因是因为 JS 可以修改 DOM，如果在 JS 执行的时候 UI 线程还在工作，就可能导致不安全的渲染 UI。这其实也是一个单线程的好处，得益于 JS 是单线程运行的，可以达到节省内存，节约上下文切换时间，没有锁的问题的好处

#12.2 执行栈

涉及面试题：什么是执行栈？

可以把执行栈认为是一个存储函数调用的栈结构，遵循先进后出的原则

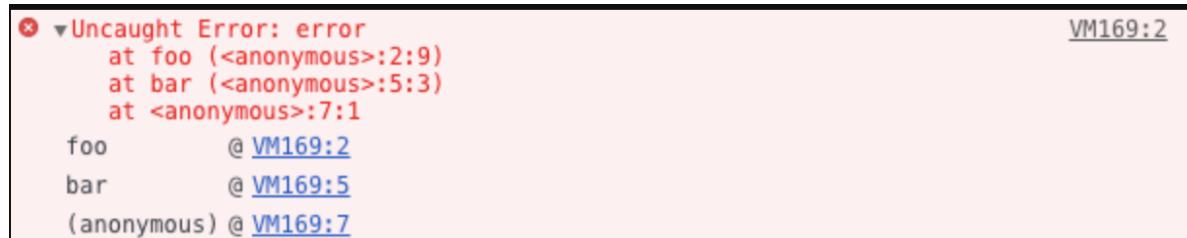


输出：

当开始执行 JS 代码时，首先会执行一个 main 函数，然后执行我们的代码。根据先进后出的原则，后执行的函数会先弹出栈，在图中我们也可以发现，foo 函数后执行，当执行完毕后就从栈中弹出了

在开发中，大家也可以在报错中找到执行栈的痕迹

```
function foo() {  
    throw new Error('error')  
}  
function bar() {  
    foo()  
}  
bar()
```

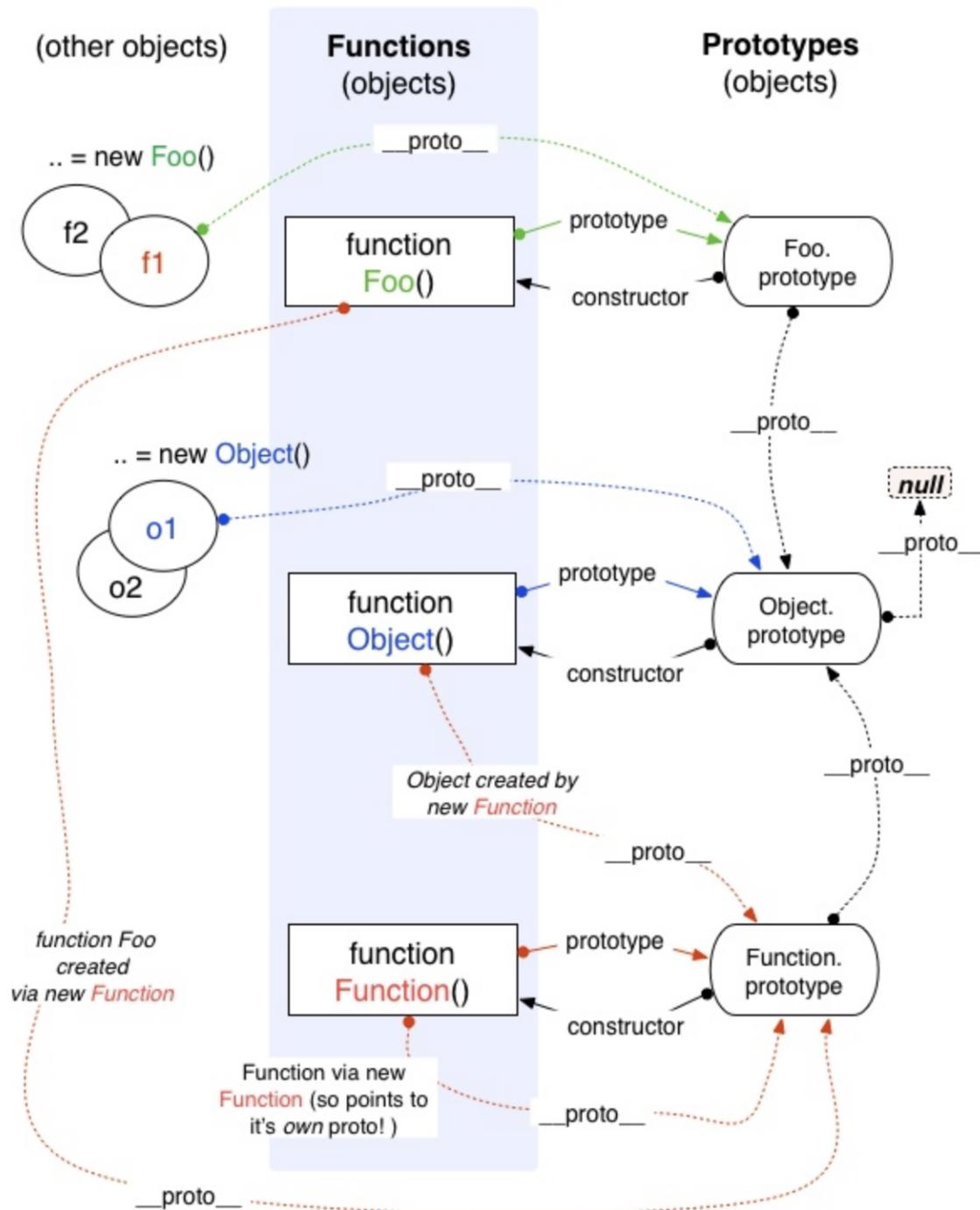


大家可以在上图清晰的看到报错在 foo 函数，foo 函数又是在 bar 函数中调用的

当我们使用递归的时候，因为栈可存放的函数是有限制的，一旦存放了过多的函数且没有得到释放的话，就会出现爆栈的问题

```
function bar() {  
    bar()  
}  
bar()
```

JavaScript Object Layout [Hrush Jain/mollypages.org]

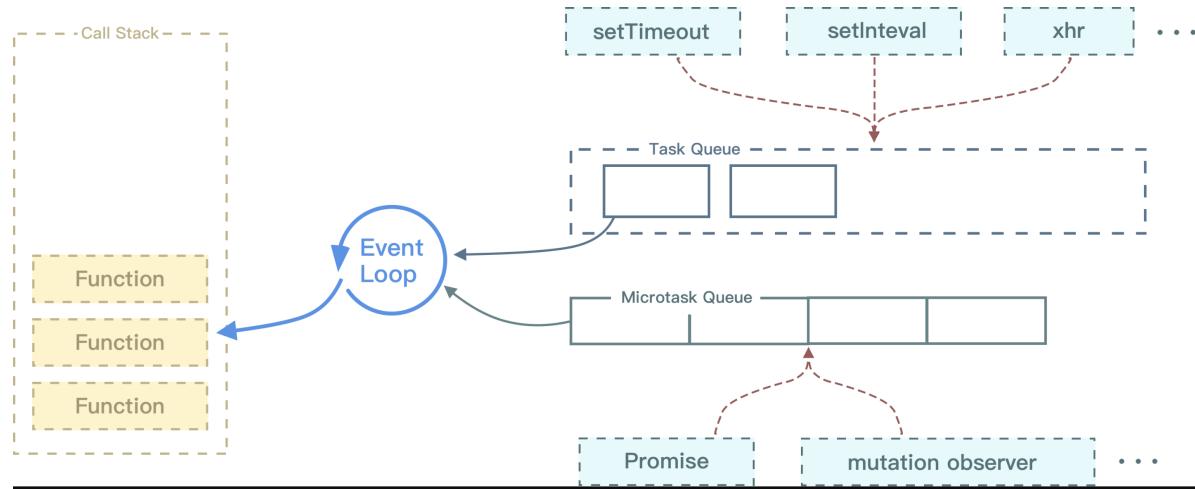


#12.3 浏览器中的 Event Loop

涉及面试题：异步代码执行顺序？解释一下什么是 Event Loop ？

众所周知 JS 是一门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是一门多线程的语言话，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点）

- JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task（有多种 task）队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为



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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask）和宏任务（macrotask）。在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task

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

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
  console.log('Promise')
  resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout
```

以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务

微任务

- `process.nextTick`
- `promise`
- `Object.observe`
- `MutationObserver`

宏任务

- `script`
- `setTimeout`
- `setInterval`
- `setImmediate`
- `I/O`
- `UI rendering`

宏任务中包括了 `script`，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

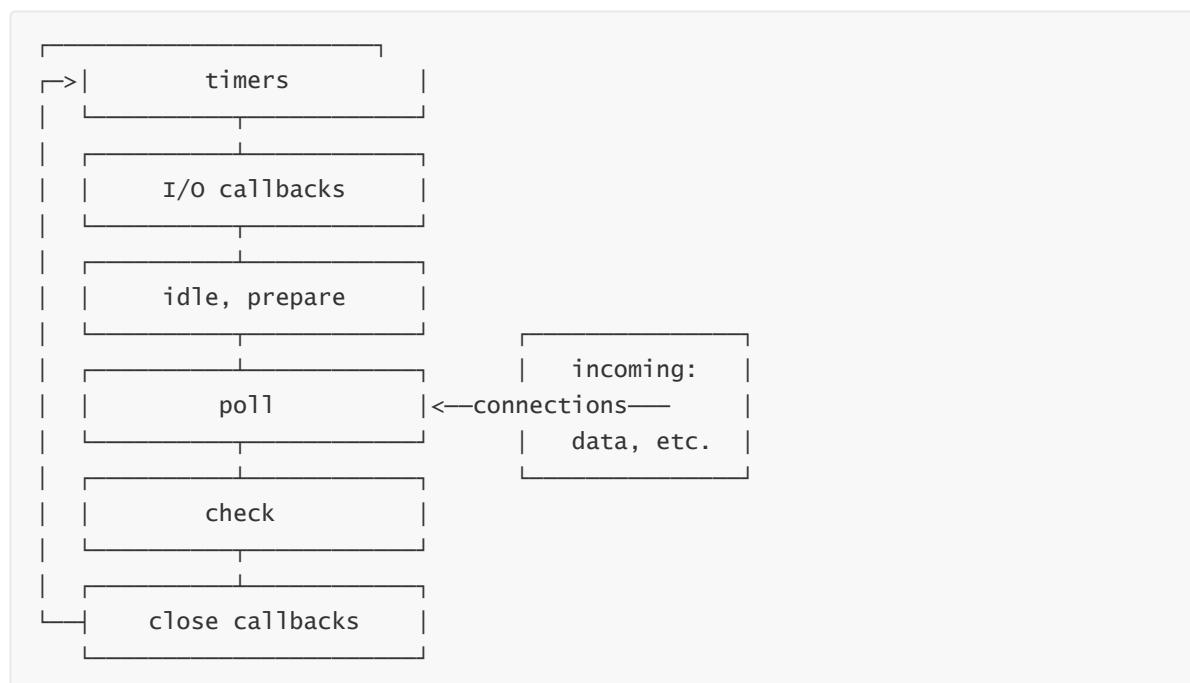
所以正确的一次 Event loop 顺序是这样的

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 `Event loop`，执行宏任务中的异步代码

通过上述的 `Event loop` 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 `DOM` 的话，为了更快的响应界面响应，我们可以把操作 `DOM` 放入微任务中

#12.4 Node 中的 Event loop

- `Node` 中的 `Event loop` 和浏览器中的不相同。
- `Node` 的 `Event loop` 分为 6 个阶段，它们会按照顺序反复运行



timer

- `timers` 阶段会执行 `setTimeout` 和 `setInterval`

- 一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟

I/O

- I/O 阶段会执行除了 close 事件，定时器和 setImmediate 的回调

poll

- poll 阶段很重要，这一阶段中，系统会做两件事情
 - 执行到点的定时器
 - 执行 poll 队列中的事件
- 并且当 poll 中没有定时器的情况下，会发现以下两件事情
 - 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
 - 如果 poll 队列为空，会有两件事发生
 - 如果有 setImmediate 需要执行， poll 阶段会停止并且进入到 check 阶段执行 setImmediate
 - 如果没有 setImmediate 需要执行，会等待回调被加入到队列中并立即执行回调
 - 如果有别的定时器需要被执行，会回到 timer 阶段执行回调。

check

- check 阶段执行 setImmediate

close callbacks

- close callbacks 阶段执行 close 事件
- 并且在 Node 中，有些情况下的定时器执行顺序是随机的

```
setTimeout(() => {
  console.log('setTimeout');
}, 0);
setImmediate(() => {
  console.log('setImmediate');
})
// 这里可能会输出 setTimeout, setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

上面介绍的都是 macrotask 的执行情况，microtask 会在以上每个阶段完成后立即执行

```
setTimeout(()=>{
  console.log('timer1')

  Promise.resolve().then(function() {
    console.log('promise1')
  })
}, 0)

setTimeout(()=>{
  console.log('timer2')

  Promise.resolve().then(function() {
    console.log('promise2')
  })
})
```

```
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2
```

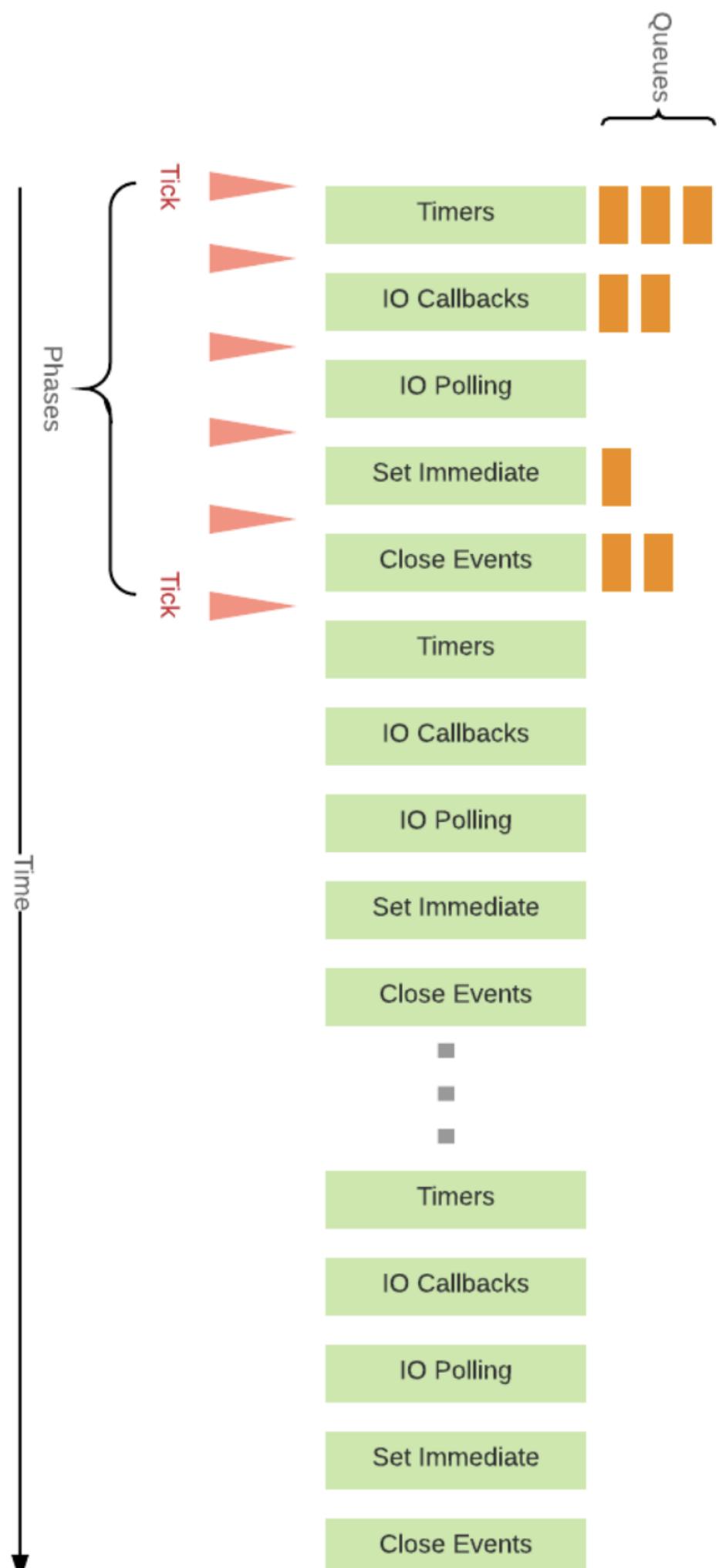
Node 中的 `process.nextTick` 会先于其他 `microtask` 执行

```
setTimeout(() => {
  console.log("timer1");

  Promise.resolve().then(function() {
    console.log("promise1");
  });
}, 0);

process.nextTick(() => {
  console.log("nextTick");
});
// nextTick, timer1, promise1
```

对于 `microtask` 来说，它会在以上每个阶段完成前清空 `microtask` 队列，下图中的 `Tick` 就代表了 `microtask`



#13 手写 call、apply 及 bind 函数

首先从以下几点来考虑如何实现这几个函数

- 不传入第一个参数，那么上下文默认为 `window`
- 改变了 `this` 指向，让新的对象可以执行该函数，并能接受参数

实现 call

- 首先 `context` 为可选参数，如果不传的话默认上下文为 `window`
- 接下来给 `context` 创建一个 `fn` 属性，并将值设置为需要调用的函数
- 因为 `call` 可以传入多个参数作为调用函数的参数，所以需要将参数剥离出来
- 然后调用函数并将对象上的函数删除

```
Function.prototype.myCall = function(context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this
  const args = [...arguments].slice(1)
  const result = context.fn(...args)
  delete context.fn
  return result
}
```

apply实现

`apply` 的实现也类似，区别在于对参数的处理

```
Function.prototype.myApply = function(context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this
  let result
  // 处理参数和 call 有区别
  if (arguments[1]) {
    result = context.fn(...arguments[1])
  } else {
    result = context.fn()
  }
  delete context.fn
  return result
}
```

bind 的实现

`bind` 的实现对比其他两个函数略微地复杂了一点，因为 `bind` 需要返回一个函数，需要判断一些边界问题，以下是 `bind` 的实现

- `bind` 返回了一个函数，对于函数来说有两种方式调用，一种是直接调用，一种是通过 `new` 的方式，我们先来说直接调用的方式
- 对于直接调用来说，这里选择了 `apply` 的方式实现，但是对于参数需要注意以下情况：因为 `bind` 可以实现类似这样的代码 `f.bind(obj, 1)(2)`，所以我们需要将两边的参数拼接起来，于是就有了这样的实现 `args.concat(...arguments)`
- 最后来说通过 `new` 的方式，在之前的章节中我们学习过如何判断 `this`，对于 `new` 的情况来说，不会被任何方式改变 `this`，所以对于这种情况我们需要忽略传入的 `this`

```
Function.prototype.myBind = function (context) {  
    if (typeof this !== 'function') {  
        throw new TypeError('Error')  
    }  
    const _this = this  
    const args = [...arguments].slice(1)  
    // 返回一个函数  
    return function F() {  
        // 因为返回了一个函数，我们可以 new F()，所以需要判断  
        if (this instanceof F) {  
            return new _this(...args, ...arguments)  
        }  
        return _this.apply(context, args.concat(...arguments))  
    }  
}
```

#14 new

涉及面试题：`new` 的原理是什么？通过 `new` 的方式创建对象和通过字面量创建有什么区别？

在调用 `new` 的过程中会发生四件事情

- 新生成了一个对象
- 链接到原型
- 绑定 `this`
- 返回新对象

根据以上几个过程，我们也可以试着来自己实现一个 `new`

- 创建一个空对象
- 获取构造函数
- 设置空对象的原型
- 绑定 `this` 并执行构造函数
- 确保返回值为对象

```
function create() {  
    let obj = {}  
    let Con = [].shift.call(arguments)  
    obj.__proto__ = Con.prototype  
    let result = Con.apply(obj, arguments)  
    return result instanceof Object ? result : obj  
}
```

- 对于对象来说，其实都是通过 `new` 产生的，无论是 `function Foo()` 还是 `let a = { b : 1 }`。
- 对于创建一个对象来说，更推荐使用字面量的方式创建对象（无论性能上还是可读性）。因为你使用 `new Object()` 的方式创建对象需要通过作用域链一层层找到 `Object`，但是你使用字面量的方式就没这个问题

```
function Foo() {}
// function 就是个语法糖
// 内部等同于 new Function()
let a = { b: 1 }
// 这个字面量内部也是使用了 new Object()
```

#15 instanceof 的原理

涉及面试题：`instanceof` 的原理是什么？

`instanceof` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 `prototype`

实现一下 `instanceof`

- 首先获取类型的原型
- 然后获得对象的原型
- 然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为 `null`，因为原型链最终为 `null`

```
function myInstanceof(left, right) {
  let prototype = right.prototype
  left = left.__proto__
  while (true) {
    if (left === null || left === undefined)
      return false
    if (prototype === left)
      return true
    left = left.__proto__
  }
}
```

#16 为什么 $0.1 + 0.2 \neq 0.3$

涉及面试题：为什么 $0.1 + 0.2 \neq 0.3$ ？如何解决这个问题？

原因，因为 JS 采用 IEEE 754 双精度版本（64位），并且只要采用 IEEE 754 的语言都有该问题

我们都应该知道计算机是通过二进制来存储东西的，那么 `0.1` 在二进制中会表示为

```
// (0011) 表示循环
0.1 = 2^-4 * 1.10011(0011)
```

我们可以发现，`0.1` 在二进制中是无限循环的一些数字，其实不只是 `0.1`，其实很多十进制小数用二进制表示都是无限循环的。这样其实没什么问题，但是 JS 采用的浮点数标准却会裁剪掉我们的数字。

IEEE 754 双精度版本（64位）将 64 位分为了三段

- 第一位用来表示符号
- 接下去的 `11` 位用来表示指数
- 其他的位数用来表示有效位，也就是用二进制表示 `0.1` 中的 `10011(0011)`

那么这些循环的数字被裁剪了，就会出现精度丢失的问题，也就造成了 `0.1` 不再是 `0.1` 了，而是变成了 `0.10000000000000002`

```
0.10000000000000002 === 0.1 // true
```

那么同样的，`0.2` 在二进制也是无限循环的，被裁剪后也失去了精度变成了

```
0.20000000000000002
```

```
0.20000000000000002 === 0.2 // true
```

所以这两者相加不等于 `0.3` 而是 `0.30000000000000004`

```
0.1 + 0.2 === 0.30000000000000004 // true
```

那么可能你又会有一个疑问，既然 `0.1` 不是 `0.1`，那为什么 `console.log(0.1)` 却是正确的呢？

因为在输入内容的时候，二进制被转换为了十进制，十进制又被转换为了字符串，在这个转换的过程中发生了取近似值的过程，所以打印出来的其实是一个近似值，你也可以通过以下代码来验证

```
console.log(0.10000000000000002) // 0.1
```

解决

```
parseFloat((0.1 + 0.2).toFixed(10)) === 0.3 // true
```

#17 事件机制

涉及面试题：事件的触发过程是怎么样的？知道什么是事件代理嘛？

#17.1 事件触发三阶段

事件触发有三个阶段：

- `window` 往事件触发处传播，遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 `window` 传播，遇到注册的冒泡事件会触发

事件触发一般来说会按照上面的顺序进行，但是也有特例，如果给一个 `body` 中的子节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行

```
// 以下会先打印冒泡然后是捕获
```

```
node.addEventListener(  
  'click',  
  event => {  
    console.log('冒泡')  
  },  
  false  
)  
node.addEventListener(  
  'click',  
  event => {  
    console.log('捕获 ')  
  },  
  true  
)
```

#17.2 注册事件

通常我们使用 `addEventListener` 注册事件，该函数的第三个参数可以是布尔值，也可以是对象。对于布尔值 `useCapture` 参数来说，该参数默认值为 `false`，`useCapture` 决定了注册的事件是捕获事件还是冒泡事件。对于对象参数来说，可以使用以下几个属性

- `capture`: 布尔值，和 `useCapture` 作用一样
- `once`: 布尔值，值为 `true` 表示该回调只会调用一次，调用后会移除监听
- `passive`: 布尔值，表示永远不会调用 `preventDefault`

一般来说，如果我们只希望事件只触发在目标上，这时候可以使用 `stopPropagation` 来阻止事件的进一步传播。通常我们认为 `stopPropagation` 是用来阻止事件冒泡的，其实该函数也可以阻止捕获事件。`stopImmediatePropagation` 同样也能实现阻止事件，但是还能阻止该事件目标执行别的注册事件。

```
node.addEventListener(  
  'click',  
  event => {  
    event.stopImmediatePropagation()  
    console.log('冒泡')  
  },  
  false  
)  
// 点击 node 只会执行上面的函数，该函数不会执行  
node.addEventListener(  
  'click',  
  event => {  
    console.log('捕获 ')  
  },  
  true  
)
```

#17.3 事件代理

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```

<ul id="u1">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let u1 = document.querySelector('#u1')
    u1.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>

```

事件代理的方式相较于直接给目标注册事件来说，有以下优点：

- 节省内存
- 不需要给子节点注销事件

#18 跨域

涉及面试题：什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？了解预检请求嘛？

- 因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，`Ajax` 请求会失败。
- 那么是出于什么安全考虑才会引入这种机制呢？其实主要是用来防止 `CSRF` 攻击的。简单点说，`CSRF` 攻击是利用用户的登录态发起恶意请求。
- 也就是说，没有同源策略的情况下，`A` 网站可以被任意其他来源的 `Ajax` 访问到内容。如果你当前 `A` 网站还存在登录态，那么对方就可以通过 `Ajax` 获得你的任何信息。当然跨域并不能完全阻止 `CSRF`。

然后我们来考虑一个问题，请求跨域了，那么请求到底发出去没有？请求必然是发出去了，但是浏览器拦截了响应。你可能会疑问明明通过表单的方式可以发起跨域请求，为什么 `Ajax` 就不会。因为归根结底，跨域是为了阻止用户读取到另一个域名下的内容，`Ajax` 可以获取响应，浏览器认为这不安全，所以拦截了响应。但是表单并不会获取新的内容，所以可以发起跨域请求。同时也说明了跨域并不能完全阻止 `CSRF`，因为请求毕竟是发出去了。

接下来我们将来学习几种常见的方法来解决跨域的问题

#18.1 JSONP

`JSONP` 的原理很简单，就是利用 `<script>` 标签没有跨域限制的漏洞。通过 `<script>` 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时

```

<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>

```

`JSONP` 使用简单且兼容性不错，但是只限于 `get` 请求。

在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的，这时候就需要自己封装一个 JSONP，以下是简单实现

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jjsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}
jsonp('http://xxx', 'callback', function(value) {
  console.log(value)
})
```

#18.2 CORS

- CORS 需要浏览器和后端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。
- 浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。
- 服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。虽然设置 CORS 和前端没什么关系，但是通过这种方式解决跨域问题的话，会在发送请求时出现两种情况，分别为简单请求和复杂请求。

简单请求

以 Ajax 为例，当满足以下条件时，会触发简单请求

1. 使用下列方法之一：

- GET
- HEAD
- POST

1. Content-Type 的值仅限于下列三者之一：

- text/plain
- multipart/form-data
- application/x-www-form-urlencoded

请求中的任意 XMLHttpRequestUpload 对象均没有注册任何事件监听器；

XMLHttpRequestUpload 对象可以使用 XMLHttpRequest.upload 属性访问

复杂请求

对于复杂请求来说，首先会发起一个预检请求，该请求是 option 方法的，通过该请求来知道服务器是否允许跨域请求。

对于预检请求来说，如果你使用过 Node 来设置 CORS 的话，可能会遇到过这么一个坑。

以下以 express 框架举例

```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*')
  res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS')
  res.header(
    'Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept, Authorization, Access-
Control-Allow-Credentials'
  )
  next()
})
```

- 该请求会验证你的 `Authorization` 字段，没有的话就会报错。
- 当前端发起了复杂请求后，你会发现就算你代码是正确的，返回结果也永远是报错的。因为预检请求也会进入回调中，也会触发 `next` 方法，因为预检请求并不包含 `Authorization` 字段，所以服务端会报错。

想解决这个问题很简单，只需要在回调中过滤 `option` 方法即可

```
res.statusCode = 204
res.setHeader('Content-Length', '0')
res.end()
```

#18.3 document.domain

- 该方式只能用于主域名相同的情况下，比如 `a.test.com` 和 `b.test.com` 适用于该方式。
- 只需要给页面添加 `document.domain = 'test.com'` 表示主域名都相同就可以实现跨域

#18.4 postMessage

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

```
// 发送消息端
window.parent.postMessage('message', 'http://test.com')
// 接收消息端
var mc = new MessageChannel()
mc.addEventListener('message', event => {
  var origin = event.origin || event.originalEvent.origin
  if (origin === 'http://test.com') {
    console.log('验证通过')
  }
})
```

#19 存储

涉及面试题：有几种方式可以实现存储功能，分别有什么优缺点？什么是 `service worker`？

`cookie, localStorage, sessionStorage, indexDB`

| 特性 | cookie | localStorage | sessionStorage | indexDB |
|--------|---------------------------|--------------|----------------|--------------|
| 数据生命周期 | 一般由服务器生成，可以设置过期时间 | 除非被清理，否则一直存在 | 页面关闭就清理 | 除非被清理，否则一直存在 |
| 数据存储大小 | 4K | 5M | 5M | 无限 |
| 与服务端通信 | 每次都会携带在 header 中，对于请求性能影响 | 不参与 | 不参与 | 不参与 |

从上表可以看到，`cookie` 已经不建议用于存储。如果没有大量数据存储需求的话，可以使用 `localStorage` 和 `sessionStorage`。对于不怎么改变的数据尽量使用 `localStorage` 存储，否则可以用 `sessionStorage` 存储。

对于 `cookie` 来说，我们还需要注意安全性。

| 属性 | 作用 |
|------------------------|---|
| <code>value</code> | 如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识 |
| <code>http-only</code> | 不能通过 <code>JS</code> 访问 <code>Cookie</code> ，减少 <code>xss</code> 攻击 |
| <code>secure</code> | 只能在协议为 <code>HTTPS</code> 的请求中携带 |
| <code>same-site</code> | 规定浏览器不能在跨域请求中携带 <code>Cookie</code> ，减少 <code>CSRF</code> 攻击 |

Service Worker

- `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`。因为 `Service worker` 中涉及到请求拦截，所以必须使用 `HTTPS` 协议来保障安全
- `Service worker` 实现缓存功能一般分为三个步骤：首先需要先注册 `Service worker`，然后监听到 `install` 事件以后就可以缓存需要的文件，那么在下次用户访问的时候就可以通过拦截请求的方式查询是否存在缓存，存在缓存的话就可以直接读取缓存文件，否则就去请求数据。以下是这个步骤的实现：

```
// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register('sw.js')
    .then(function(registration) {
      console.log('service worker 注册成功')
    })
    .catch(function(err) {
      console.log('servcie worker 注册失败')
    })
}
// sw.js
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener('install', e => {
  e.waitUntil(
    caches.open('my-cache').then(function(cache) {
      return cache.addAll(['./index.html', './index.js'])
    })
  )
})
```

```

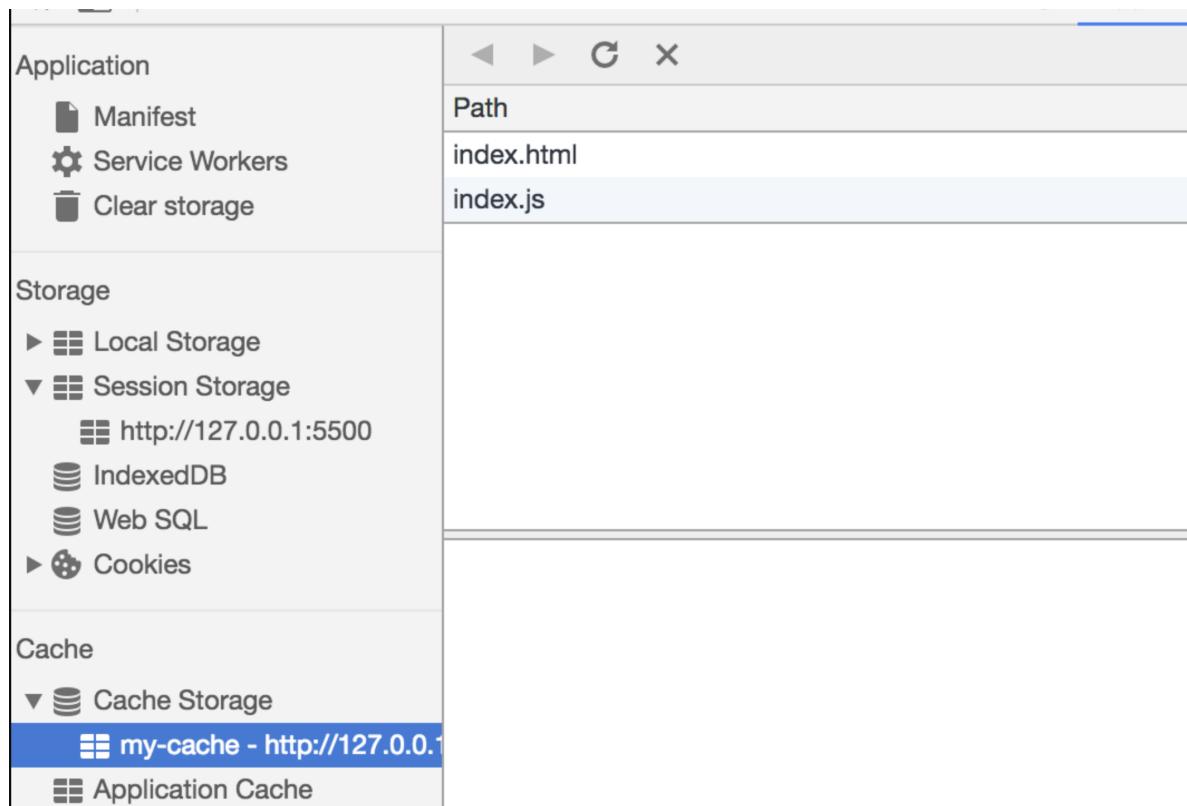
        }
    )
}

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener('fetch', e => {
    e.respondWith(
        caches.match(e.request).then(function(response) {
            if (response) {
                return response
            }
            console.log('fetch source')
        })
    )
}
)

```

打开页面，可以在开发者工具中的 `Application` 看到 `Service Worker` 已经启动了

在 `Cache` 中也可以发现我们所需的文件已被缓存



当我们重新刷新页面可以发现我们缓存的数据是从 `Service Worker` 中读取的

#20 浏览器缓存机制

注意：该知识点属于性能优化领域，并且整一章节都是一个面试题

- 缓存可以说是性能优化中简单高效的一种优化方式了，它可以显著减少网络传输所带来的损耗。
- 对于一个数据请求来说，可以分为发起网络请求、后端处理、浏览器响应三个步骤。浏览器缓存可以帮助我们在第一和第三步骤中优化性能。比如说直接使用缓存而不发起请求，或者发起了请求但后端存储的数据和前端一致，那么就没有必要再将数据回传回来，这样就减少了响应数据。

接下来的内容中我们将通过以下几个部分来探讨浏览器缓存机制：

- 缓存位置
- 缓存策略
- 实际场景应用缓存策略

#20.1 缓存位置

从缓存位置上来说分为四种，并且各自有优先级，当依次查找缓存且都没有命中的时候，才会去请求网络

1. Service Worker
2. Memory Cache
3. Disk Cache
4. Push Cache
5. 网络请求

1. Service Worker

- service worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的。
- 当 service worker 没有命中缓存的时候，我们需要去调用 fetch 函数获取数据。也就是说，如果我们没有在 service worker 命中缓存的话，会根据缓存查找优先级去查找数据。但是不管我们是从 Memory Cache 中还是从网络请求中获取的数据，浏览器都会显示我们是从 service worker 中获取的内容。

2. Memory Cache

- Memory Cache 也就是内存中的缓存，读取内存中的数据肯定比磁盘快。但是内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。
- 当我们访问过页面以后，再次刷新页面，可以发现很多数据都来自于内存缓存

| Name | Status | Type | Initiator | Size | ... | Waterfall |
|----------------------------------|----------------|----------|--------------------------|---------------------|-----|-----------|
| timeline | 200 | document | Other | 12.3 KB | ... | |
| manifest.af6996a28865b94880dd.js | 200 | script | timeline | (from memory cache) | ... | |
| vendor.1020ec95ad777daa7497.js | 200 | script | timeline | (from memory cache) | ... | |
| app.e4986bf68ccecea5526a.js | 200 | script | timeline | (from memory cache) | ... | |
| adsbygoogle.js | (blocked:0...) | script | timeline | 0 B | ... | |
| icon.a87e5ae.svg | 200 | svg+xml | timeline | (from memory cache) | ... | |

那么既然内存缓存这么高效，我们是不是能让数据都存放在内存中呢？

- 先说结论，这是不可能的。首先计算机中的内存一定比硬盘容量小得多，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。内存中其实可以存储大部分的文件，比如说 JS、HTML、CSS、图片等等
- 当然，我通过一些实践和猜测也得出了一些结论：
- 对于大文件来说，大概率是不存储在内存中的，反之优先当前系统内存使用率高的话，文件优先存储进硬盘

3. Disk Cache

- Disk Cache 也就是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。
- 在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 ·HTTP Header· 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据

4. Push Cache

- `Push Cache` 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。并且缓存时间也很短暂，只在会话（`session`）中存在，一旦会话结束就被释放。
- `Push Cache` 在国内能够查到的资料很少，也是因为 `HTTP/2` 在国内不够普及，但是 `HTTP/2` 将会是日后的一个趋势

结论

- 所有的资源都能被推送，但是 `Edge` 和 `Safari` 浏览器兼容性不怎么好
- 可以推送 `no-cache` 和 `no-store` 的资源
- 一旦连接被关闭，`Push Cache` 就被释放
- 多个页面可以使用相同的 `HTTP/2` 连接，也就是说能使用同样的缓存
- `Push Cache` 中的缓存只能被使用一次
- 浏览器可以拒绝接受已经存在的资源推送
- 你可以给其他域名推送资源

5. 网络请求

- 如果所有缓存都没有命中的话，那么只能发起请求来获取资源了。
- 那么为了性能上的考虑，大部分的接口都应该选择好缓存策略，接下来我们就来学习缓存策略这部分的内容

#20.2 缓存策略

通常浏览器缓存策略分为两种：强缓存和协商缓存，并且缓存策略都是通过设置 `HTTP Header` 来实现的

#20.2.1 强缓存

强缓存可以通过设置两种 `HTTP Header` 实现：`Expires` 和 `Cache-Control`。强缓存表示在缓存期间不需要请求，`state code` 为 `200`

Expires

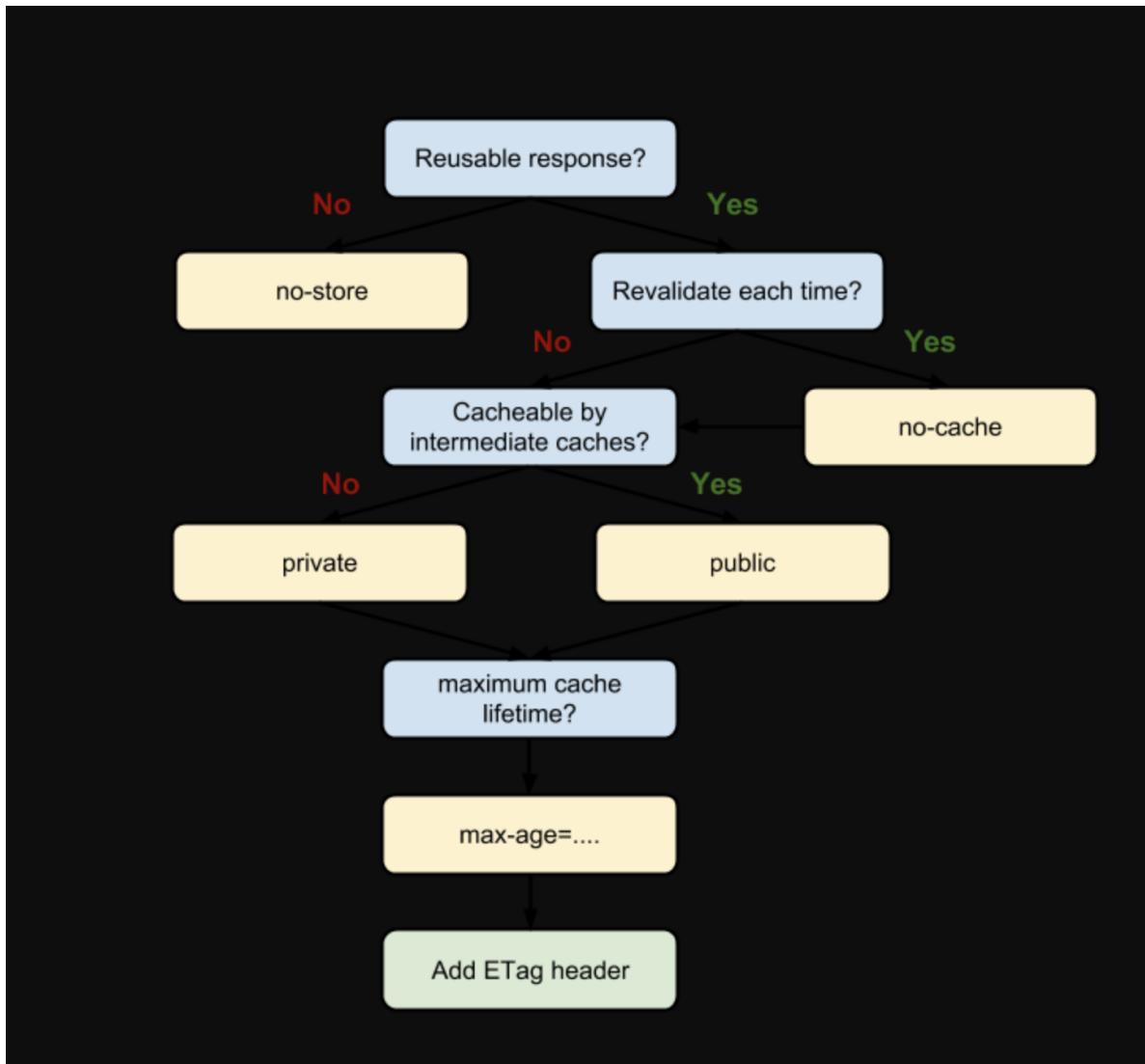
```
Expires: Wed, 22 Oct 2018 08:41:00 GMT
```

`Expires` 是 `HTTP/1` 的产物，表示资源会在 `wed, 22 oct 2018 08:41:00 gmt` 后过期，需要再次请求。并且 `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。

Cache-control

```
Cache-control: max-age=30
```

- `Cache-Control` 出现于 `HTTP/1.1`，优先级高于 `Expires`。该属性值表示资源会在 30 秒后过期，需要再次请求。
- `Cache-Control` 可以在请求头或者响应头中设置，并且可以组合使用多种指令



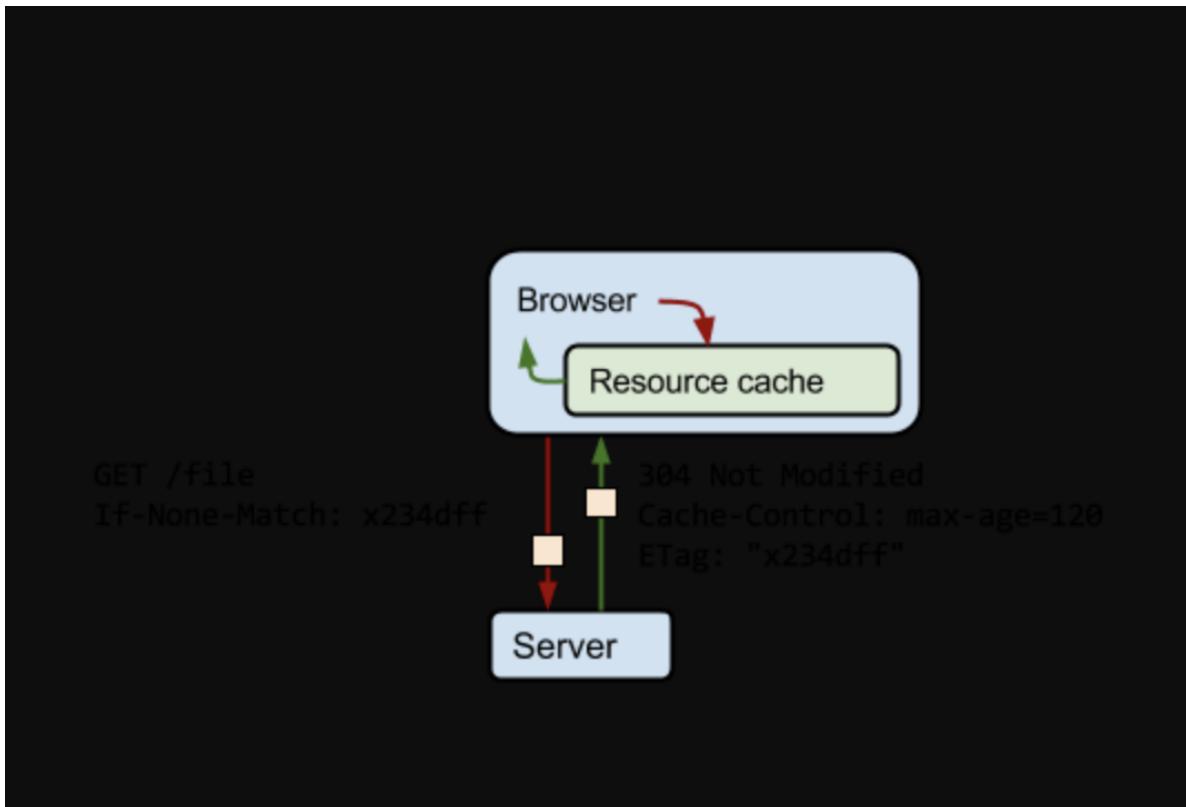
从图中我们可以看到，我们可以将多个指令配合起来一起使用，达到多个目的。比如说我们希望资源能被缓存下来，并且是客户端和代理服务器都能缓存，还能设置缓存失效时间等

一些常见指令的作用

| 指令 | 作用 |
|---------------------------|--|
| <code>public</code> | 表示响应可以被客户端和代理服务器缓存 |
| <code>private</code> | 表示响应只可以被客户端缓存 |
| <code>max-age=30</code> | 缓存 30 秒后就过期，需要重新请求 |
| <code>s-maxage=30</code> | 覆盖 <code>max-age</code> ，作用一样，只在代理服务器中生效 |
| <code>no-store</code> | 不缓存任何响应 |
| <code>no-cache</code> | 资源被缓存，但是立即失效，下次会发起请求验证资源是否过期 |
| <code>max-stale=30</code> | 30 秒内，即使缓存过期，也使用该缓存 |
| <code>min-fresh=30</code> | 希望在 30 秒内获取最新的响应 |

#20.2.2 协商缓存

- 如果缓存过期了，就需要发起请求验证资源是否有更新。协商缓存可以通过设置两种 `HTTP Header` 实现：`Last-Modified` 和 `ETag`
- 当浏览器发起请求验证资源时，如果资源没有做改变，那么服务端就会返回 `304` 状态码，并且更新浏览器缓存有效期。



Last-Modified 和 If-Modified-Since

`Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来，否则返回 `304` 状态码。

但是 `Last-Modified` 存在一些弊端：

- 如果本地打开缓存文件，即使没有对文件进行修改，但还是会造造成 `Last-Modified` 被修改，服务端不能命中缓存导致发送相同的资源
- 因为 `Last-Modified` 只能以秒计时，如果在不可感知的时间内修改完成文件，那么服务端会认为资源还是命中了，不会返回正确的资源。因为以上这些弊端，所以在 `HTTP / 1.1` 出现了 `ETag`

ETag 和 If-None-Match

- `ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来。并且 `ETag` 优先级比 `Last-Modified` 高。

以上就是缓存策略的所有内容了，看到这里，不知道你是否存在这样一个疑问。如果什么缓存策略都没设置，那么浏览器会怎么处理？

对于这种情况，浏览器会采用一个启发式的算法，通常会取响应头中的 `Date` 减去 `Last-Modified` 值的 `10%` 作为缓存时间。

#20.3 实际场景应用缓存策略

频繁变动的资源

对于频繁变动的资源，首先需要使用 `Cache-Control: no-cache` 使浏览器每次都请求服务器，然后配合 `ETag` 或者 `Last-Modified` 来验证资源是否有效。这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。

代码文件

这里特指除了 `HTML` 外的代码文件，因为 `HTML` 文件一般不缓存或者缓存时间很短。

一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理，只有当代码修改后才会生成新的文件名。基于此，我们就可以给代码文件设置缓存有效期一年 `Cache-Control: max-age=31536000`，这样只有当 `HTML` 文件中引入的文件名发生了改变才会去下载最新的代码文件，否则就一直使用缓存

更多缓存知识详解 <http://blog.poeties.top/2019/01/02/browser-cache>

#21 浏览器渲染原理

注意：该章节都是一个面试题。

#21.1 渲染过程

1. 浏览器接收到 HTML 文件并转换为 DOM 树

当我们打开一个网页时，浏览器都会去请求对应的 `HTML` 文件。虽然平时我们写代码时都会分为 `JS`、`CSS`、`HTML` 文件，也就是字符串，但是计算机硬件是不理解这些字符串的，所以在网络中传输的内容其实都是 `0` 和 `1` 这些字节数据。当浏览器接收到这些字节数据以后，它会将这些字节数据转换为字符串，也就是我们写的代码。

字节数据 => 字符串

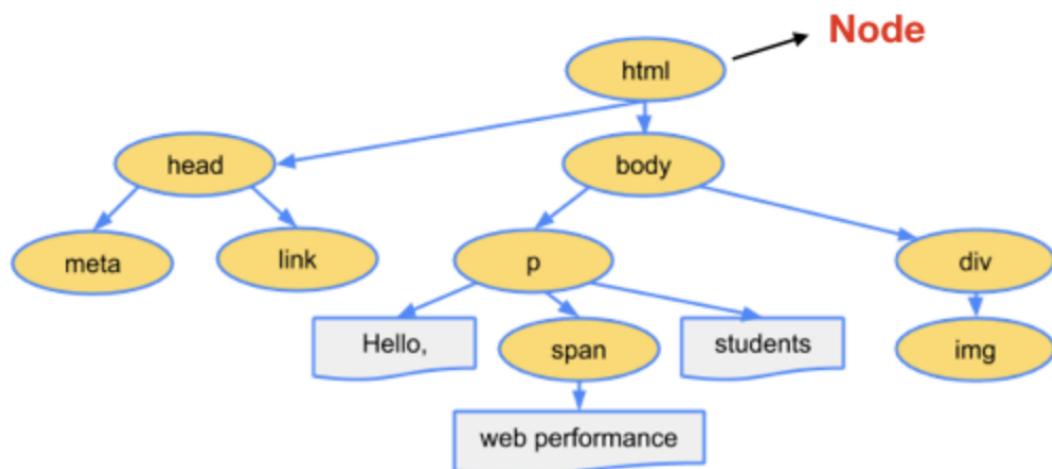
当数据转换为字符串以后，浏览器会先将这些字符串通过词法分析转换为标记（`token`），这一过程在词法分析中叫做标记化（`tokenization`）

字节数据 => 字符串 => Token

那么什么是标记呢？这其实属于编译原理这一块的内容了。简单来说，标记还是字符串，是构成代码的最小单位。这一过程会将代码分拆成一块块，并给这些内容打上标记，便于理解这些最小单位的代码是什么意思



当结束标记化后，这些标记会紧接着转换为 `Node`，最后这些 `Node` 会根据不同 `Node` 之前的联系构建为一颗 `DOM` 树



以上就是浏览器从网络中接收到 `HTML` 文件然后一系列的转换过程

字节数据 => 字符串 => Token => Node => DOM

当然，在解析 `HTML` 文件的时候，浏览器还会遇到 `css` 和 `js` 文件，这时候浏览器也会去下载并解析这些文件，接下来就让我们先来学习浏览器如何解析 `css` 文件

2. 将 CSS 文件转换为 CSSOM 树

其实转换 `css` 到 `CSSOM` 树的过程和上一小节的过程是极其类似的

字节数据 => 字符串 => Token => Node => CSSOM

- 在这一过程中，浏览器会确定下每一个节点的样式到底是什么，并且这一过程其实是很消耗资源的。因为样式你可以自行设置给某个节点，也可以通过继承获得。在这一过程中，浏览器得递归 `CSSOM` 树，然后确定具体的元素到底是什么样式。

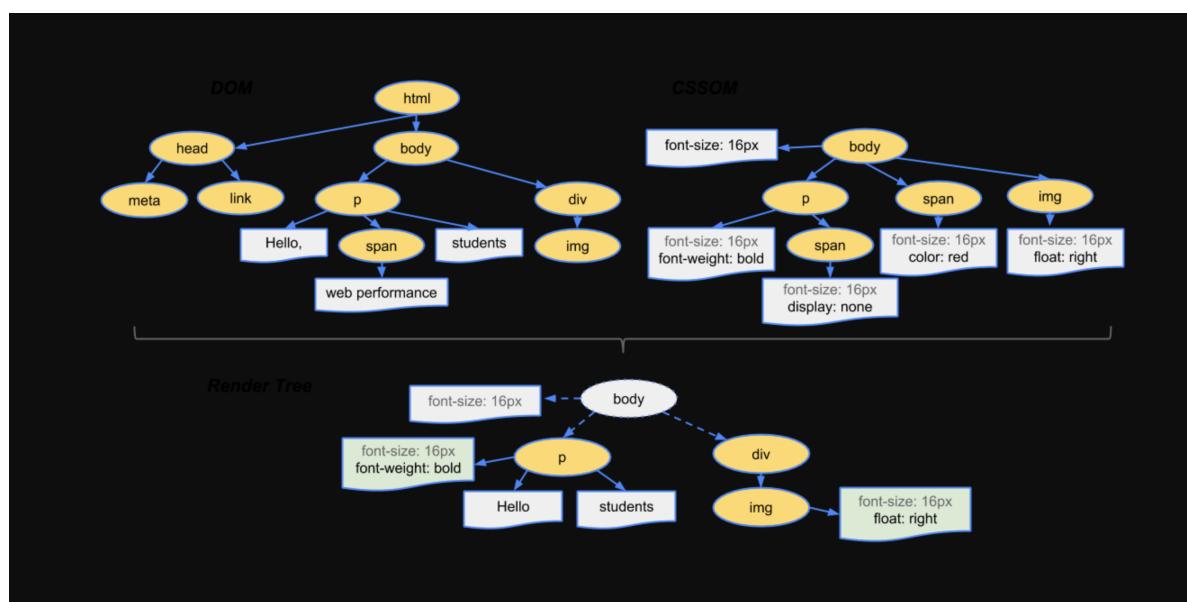
如果你有点不理解为什么会消耗资源的话，我这里举个例子

```
<div>
  <a> <span></span> </a>
</div>
<style>
  span {
    color: red;
  }
  div > a > span {
    color: red;
  }
</style>
```

对于第一种设置样式的方式来说，浏览器只需要找到页面中所有的 `span` 标签然后设置颜色，但是对于第二种设置样式的方式来说，浏览器首先需要找到所有的 `span` 标签，然后找到 `span` 标签上的 `a` 标签，最后再去找到 `div` 标签，然后给符合这种条件的 `span` 标签设置颜色，这样的递归过程就很复杂。所以我们应该尽可能的避免写过于具体的 `CSS` 选择器，然后对于 `HTML` 来说也尽量少的添加无意义标签，保证层级扁平

3. 生成渲染树

当我们生成 `DOM` 树和 `CSSOM` 树以后，就需要将这两棵树组合为渲染树



- 在这一过程中，不是简单的将两者合并就行了。渲染树只会包括需要显示的节点和这些节点的样式信息，如果某个节点是 `display: none` 的，那么就不会在渲染树中显示。
- 当浏览器生成渲染树以后，就会根据渲染树来进行布局（也可以叫做回流），然后调用 GPU 绘制，合成图层，显示在屏幕上。对于这一部分的内容因为过于底层，还涉及到了硬件相关的知识，这里就不再继续展开内容了。

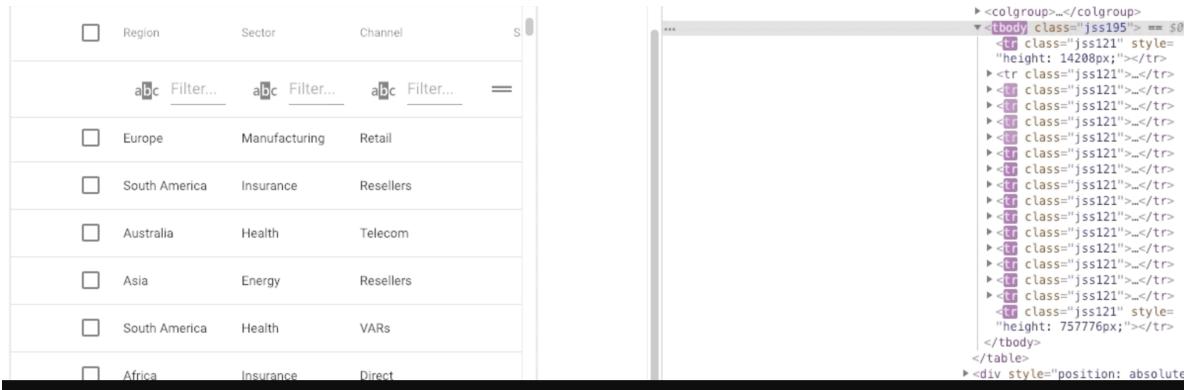
#21.2 为什么操作 DOM 慢

想必大家都听过操作 DOM 性能很差，但是这其中的原因是什么呢？

- 因为 `DOM` 是属于渲染引擎中的东西，而 `JS` 又是 `JS` 引擎中的东西。当我们通过 `JS` 操作 `DOM` 的时候，其实这个操作涉及到了两个线程之间的通信，那么势必会带来一些性能上的损耗。操作 `DOM` 次数一多，也就等同于一直在进行线程之间的通信，并且操作 `DOM` 可能还会带来重绘回流的情况，所以也就导致了性能上的问题。

经典面试题：插入几万个 DOM，如何实现页面不卡顿？

- 对于这道题目来说，首先我们肯定不能一次性把几万个 DOM 全部插入，这样肯定会卡顿，所以解决问题的重点应该是如何分批次部分渲染 DOM。大部分人应该可以想到通过 `requestAnimationFrame` 的方式去循环的插入 DOM，其实还有种方式去解决这个问题：虚拟滚动（virtualized scroller）。
- 这种技术的原理就是只渲染可视区域内的内容，非可见区域的那就完全不渲染了，当用户在滚动的时候就实时去替换渲染的内容



从上图中我们可以发现，即使列表很长，但是渲染的 DOM 元素永远只有那么几个，当我们滚动页面的时候就会实时去更新 DOM，这个技术就能顺利解决这道经典面试题

#21.3 什么情况阻塞渲染

- 首先渲染的前提是生成渲染树，所以 HTML 和 CSS 肯定会阻塞渲染。如果你想渲染的越快，你越应该降低一开始需要渲染的文件大小，并且扁平层级，优化选择器。
- 然后当浏览器在解析到 script 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件，这也是都建议将 script 标签放在 body 标签底部的原因。
- 当然在当下，并不是说 script 标签必须放在底部，因为你可以给 script 标签添加 defer 或者 async 属性。
- 当 script 标签加上 defer 属性以后，表示该 JS 文件会并行下载，但是会放到 HTML 解析完成后顺序执行，所以对于这种情况你可以把 script 标签放在任意位置。
- 对于没有任何依赖的 JS 文件可以加上 async 属性，表示 JS 文件下载和解析不会阻塞渲染。

#21.4 重绘 (Repaint) 和回流 (Reflow)

重绘和回流会在我们设置节点样式时频繁出现，同时也会很大程度上影响性能。

- 重绘是当节点需要更改外观而不会影响布局的，比如改变 color 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流。
- 回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变父节点里的子节点很可能导致父节点的一系列回流。

以下几个动作可能会导致性能问题：

- 改变 window 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

并且很多人不知道的是，重绘和回流其实也和 Event loop 有关。

- 当 Eventloop 执行完 Microtasks 后，会判断 document 是否需要更新，因为浏览器是 60Hz 的刷新率，每 16.6ms 才会更新一次。
- 然后判断是否有 resize 或者 scroll 事件，有的话会去触发事件，所以 resize 和 scroll 事件也是至少 16ms 才会触发一次，并且自带节流功能。
- 判断是否触发了 media query
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 requestAnimationFrame 回调
- 执行 IntersectionObserver 回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好 更新界面
- 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 requestIdleCallback 回调

#21.5 减少重绘和回流

1. 使用 transform 替代 top

```
<div class="test"></div>
<style>
.test {
  position: absolute;
  top: 10px;
  width: 100px;
  height: 100px;
  background: red;
}
</style>
<script>
setTimeout(() => {
  // 引起回流
  document.querySelector('.test').style.top = '100px'
}, 1000)
</script>
```

1. 使用 visibility 替换 display: none，因为前者只会引起重绘，后者会引发回流（改变了布局）

2. 不要把节点的属性值放在一个循环里当成循环里的变量

```
for(let i = 0; i < 1000; i++) {
  // 获取 offsetTop 会导致回流，因为需要去获取正确的值
  console.log(document.querySelector('.test').style.offsetTop)
}
```

1. 不要使用 table 布局，可能很小的一个小改动会造成整个 table 的重新布局

2. 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用

requestAnimationFrame

3. CSS 选择符从右往左匹配查找，避免节点层级过多

4. 将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点。比如对于 video 标签来说，浏览器会自动将该节点变为图层。

设置节点为图层的方式有很多，我们可以通过以下几个常用属性可以生成新图层

- will-change
- video、iframe 标签

#22 安全防范

#22.1 XSS

涉及面试题：什么是 `xss` 攻击？如何防范 `xss` 攻击？什么是 `CSP`？

- `xss` 简单点来说，就是攻击者想尽一切办法将可以执行的代码注入到网页中。
- `xss` 可以分为多种类型，但是总体上我认为分为两类：持久型和非持久型。
- 持久型也就是攻击的代码被服务端写入进数据库中，这种攻击危害性很大，因为如果网站访问量很大的话，就会导致大量正常访问页面的用户都受到攻击。

举个例子，对于评论功能来说，就得防范持久型 `xss` 攻击，因为我可以在评论中输入以下内容

```
<script>alert(1)</script>
```



😊 表情

Ctrl or ⌘ + Enter

评论

- 这种情况如果前后端没有做好防御的话，这段评论就会被存储到数据库中，这样每个打开该页面的用户都会被攻击到。
- 非持久型相比于前者危害就小的多了，一般通过修改 `URL` 参数的方式加入攻击代码，诱导用户访问链接从而进行攻击。

举个例子，如果页面需要从 `URL` 中获取某些参数作为内容的话，不经过过滤就会导致攻击代码被执行

```
<!-- http://www.domain.com?name=<script>alert(1)</script> -->
<div>{{name}}</div>
```

但是对于这种攻击方式来说，如果用户使用 `Chrome` 这类浏览器的话，浏览器就能自动帮助用户防御攻击。但是我们不能因此就不防御此类攻击了，因为我不能确保用户都使用了该类浏览器。



该网页无法正常运作

Chrome 在此网页上检测到了异常代码。为保护您的个人信息（例如密码、电话号码和信用卡信息），Chrome 已将该网页拦截。

请尝试[访问该网站的首页](#)。

ERR_BLOCKED_BY_XSS_AUDITOR

对于 `xss` 攻击来说，通常有两种方式可以用来防御。

1. 转义字符

首先，对于用户的输入应该是永远不信任的。最普遍的做法就是转义输入输出的内容，对于引号、尖括号、斜杠进行转义

```
function escape(str) {
    str = str.replace(/&/g, '&amp;')
    str = str.replace(/</g, '&lt;')
    str = str.replace(/>/g, '&gt;')
    str = str.replace(/\"/g, '&quot;')
    str = str.replace(/\'/g, '&#39;')
    str = str.replace(/\`/g, '&#96;')
    str = str.replace(/\//g, '&x2F;')
    return str
}
```

通过转义可以将攻击代码 `<script>alert(1)</script>` 变成

```
// -> &lt;script&gt;alert(1)&lt;&#x2F;script&gt;
escape('<script>alert(1)</script>')
```

但是对于显示富文本来说，显然不能通过上面的办法来转义所有字符，因为这样会把需要的格式也过滤掉。对于这种情况，通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式

```
const xss = require('xss')
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html)
```

以上示例使用了 `js-xss` 来实现，可以看到在输出中保留了 `h1` 标签且过滤了 `script` 标签

1. CSP

`CSP` 本质上就是建立白名单，开发者明确告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截是由浏览器自己实现的。我们可以通过这种方式来尽量减少 `xss` 攻击。

通常可以通过两种方式来开启 CSP：

- 设置 `HTTP Header` 中的 `Content-Security-Policy`
- 设置 `meta` 标签的方式 `<meta http-equiv="Content-Security-Policy">`

这里以设置 `HTTP Header` 来举例

只允许加载本站资源

```
Content-Security-Policy: default-src 'self'
```

只允许加载 HTTPS 协议图片

```
Content-Security-Policy: img-src https://*
```

允许加载任何来源框架

```
Content-Security-Policy: child-src 'none'
```

当然可以设置的属性远不止这些，你可以通过查阅 [文档](#) 的方式来学习，这里就不过多赘述其他的属性了。

对于这种方式来说，只要开发者配置了正确的规则，那么即使网站存在漏洞，攻击者也不能执行它的攻击代码，并且 `CSP` 的兼容性也不错。



#22.2 CSRF

涉及面试题：什么是 `CSRF` 攻击？如何防范 `CSRF` 攻击？

`CSRF` 中文名为跨站请求伪造。原理就是攻击者构造出一个后端请求地址，诱导用户点击或者通过某些途径自动发起请求。如果用户是在登录状态下的话，后端就以为是用户在操作，从而进行相应的逻辑。

举个例子，假设网站中有一个通过 `GET` 请求提交用户评论的接口，那么攻击者就可以在钓鱼网站中加入一个图片，图片的地址就是评论接口

```

```

那么你是否会想到使用 `POST` 方式提交请求是不是就没有这个问题了呢？其实并不是，使用这种方式也不是百分百安全的，攻击者同样可以诱导用户进入某个页面，在页面中通过表单提交 `POST` 请求。

如何防御

- `Get` 请求不对数据进行修改
- 不让第三方网站访问到用户 `Cookie`
- 阻止第三方网站请求接口
- 请求时附带验证信息，比如验证码或者 `Token`

SameSite

可以对 `Cookie` 设置 `SameSite` 属性。该属性表示 `Cookie` 不随着跨域请求发送，可以很大程度减少 `CSRF` 的攻击，但是该属性目前并不是所有浏览器都兼容。

验证 Referer

对于需要防范 `CSRF` 的请求，我们可以通过验证 `Referer` 来判断该请求是否为第三方网站发起的。

Token

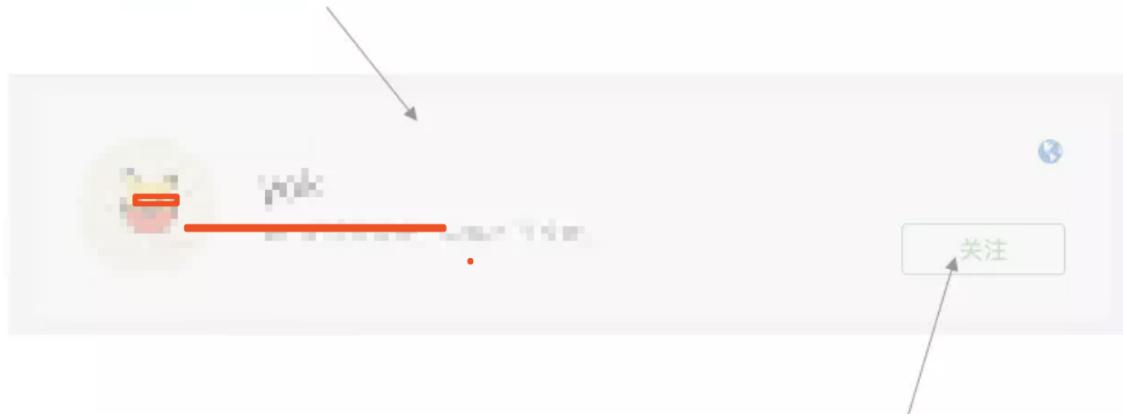
服务器下发一个随机 `Token`，每次发起请求时将 `Token` 携带上，服务器验证 `Token` 是否有效

#22.3 点击劫持

涉及面试题：什么是点击劫持？如何防范点击劫持？

点击劫持是一种视觉欺骗的攻击手段。攻击者将需要攻击的网站通过 `iframe` 嵌套的方式嵌入自己的网页中，并将 `iframe` 设置为透明，在页面中透出一个按钮诱导用户点击

页面设置为透明 `iframe`



在关注按钮位置放置一个自身的按钮，诱导用户点击

对于这种攻击方式，推荐防御的方法有两种

1. X-FRAME-OPTIONS

`X-FRAME-OPTIONS` 是一个 `HTTP` 响应头，在现代浏览器有一个很好的支持。这个 `HTTP` 响应头就是为了防御用 `iframe` 嵌套的点击劫持攻击。

该响应头有三个值可选，分别是

- `DENY`，表示页面不允许通过 `iframe` 的方式展示
- `SAMEORIGIN`，表示页面可以在相同域名下通过 `iframe` 的方式展示
- `ALLOW-FROM`，表示页面可以在指定来源的 `iframe` 中展示

2. JS 防御

对于某些远古浏览器来说，并不能支持上面的这种方式，那我们只有通过 `JS` 的方式来防御点击劫持了。

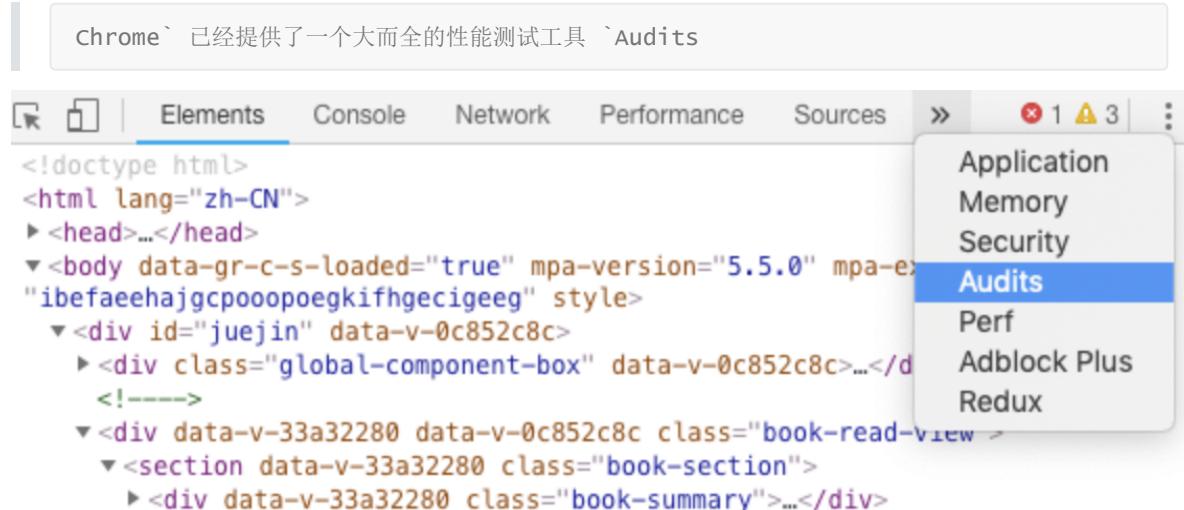
```
<head>
<style id="click-jack">
  html {
    display: none !important;
  }
</style>
</head>
<body>
<script>
  if (self == top) {
    var style = document.getElementById('click-jack')
    document.body.removeChild(style)
  } else {
    top.location = self.location
  }
</script>
</body>
```

以上代码的作用就是当通过 `iframe` 的方式加载页面时，攻击者的网页直接不显示所有内容了

#23 从 V8 中看 JS 性能优化

注意：该知识点属于性能优化领域。

#23.1 测试性能工具



点我们点击 `Audits` 后，可以看到如下的界面



Audits

Identify and fix common problems that affect your site's performance, accessibility, and user experience. [Learn more](#)

Device

Mobile

Desktop

Audits

Performance

Progressive Web App

Best practices

Accessibility

SEO

Throttling

Simulated Fast 3G, 4x CPU Slowdown

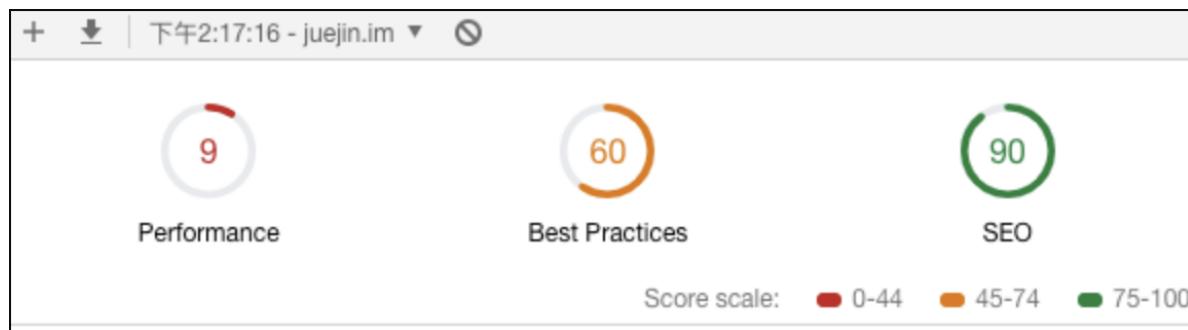
Applied Fast 3G, 4x CPU Slowdown

No throttling

Clear storage

Run audits

在这个界面中，我们可以选择想测试的功能然后点击 `Run audits`，工具就会自动运行帮助我们测试问题并且给出一个完整的报告



上图是给掘金首页测试性能后给出的一个报告，可以看到报告中分别为性能、体验、SEO 都给出了打分，并且每一个指标都有详细的评估

Performance

9

Metrics

First Contentful Paint 5,460 ms ▲ First Meaningful Paint 6,970 ms ▲

Speed Index 11,970 ms ▲ First CPU Idle 14,590 ms ▲

Time to Interactive 15,740 ms ▲ Estimated Input Latency 1,982 ms ▲

[View Trace](#)

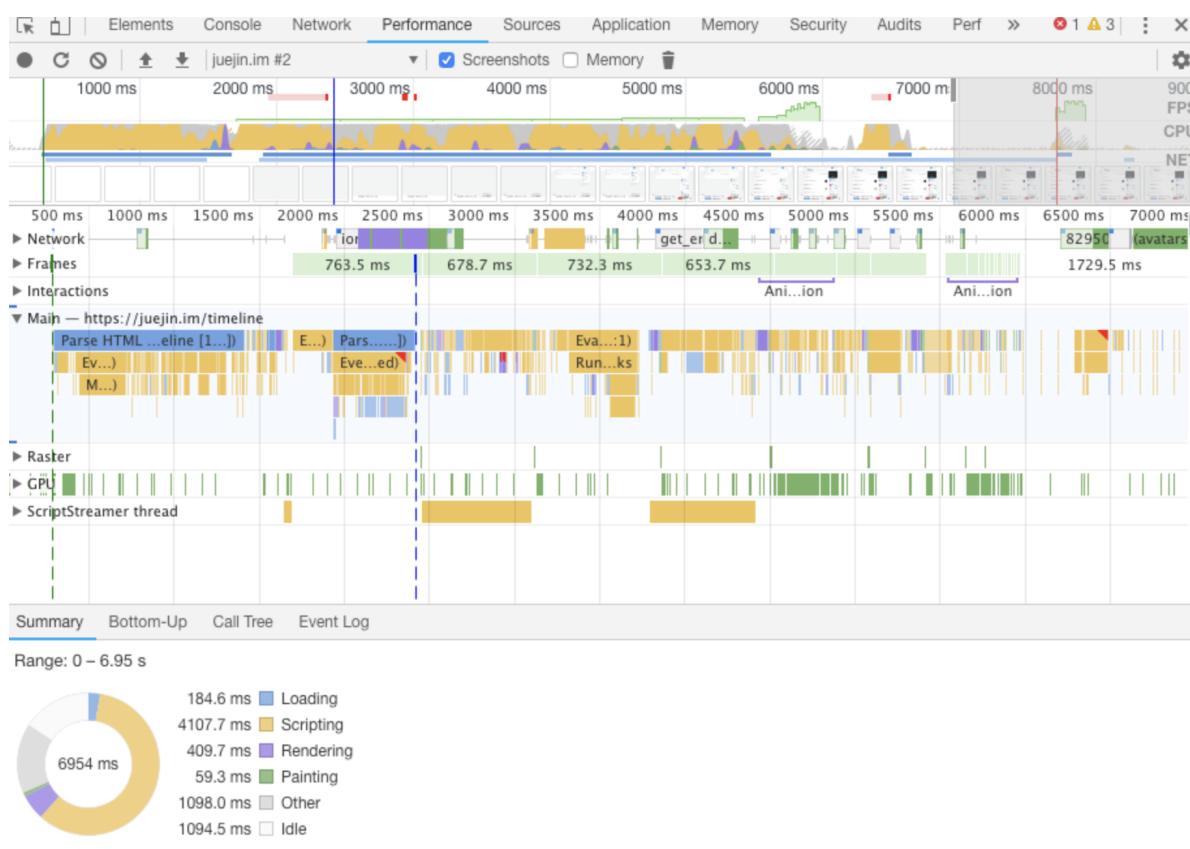
Values are estimated and may vary.



评估结束后，工具还提供了一些建议便于我们提高这个指标的分数

我们只需要一条条根据建议去优化性能即可。

除了 Audits 工具之外，还有一个 Performance 工具也可以供我们使用。



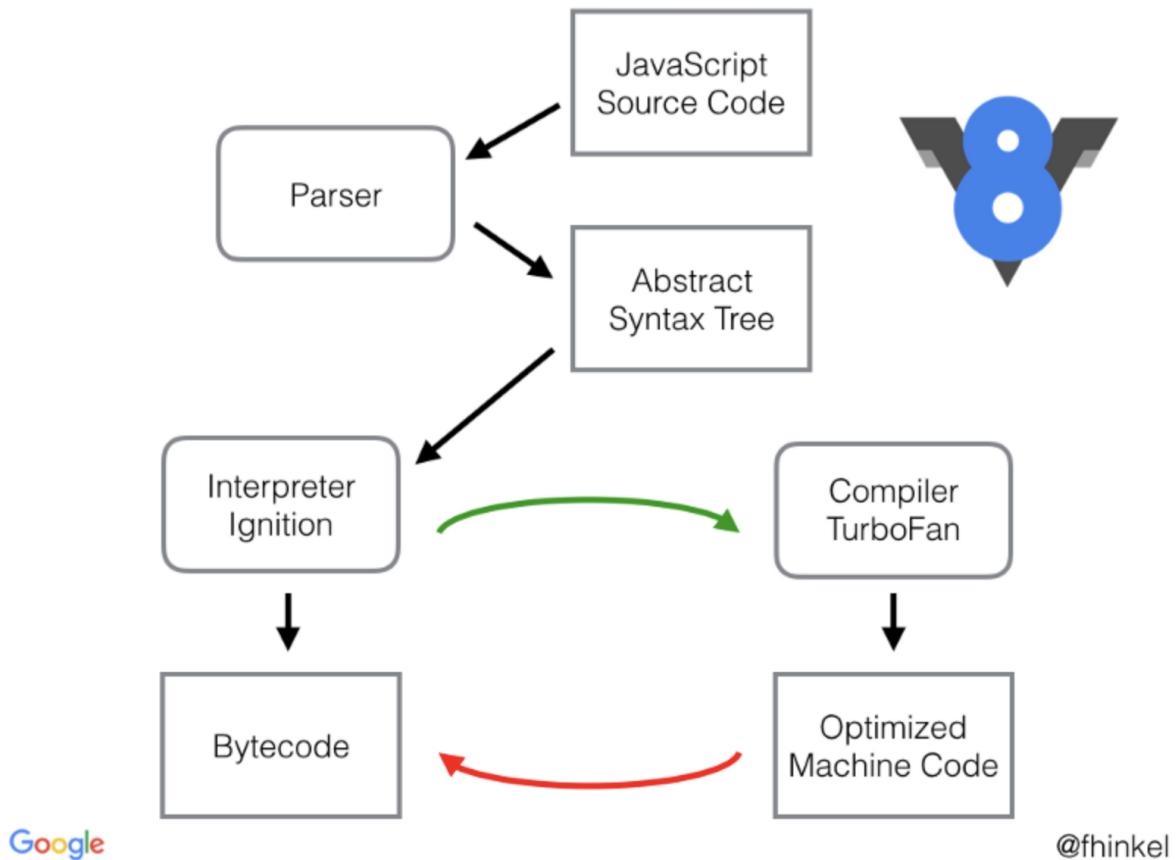
在这张图中，我们可以详细的看到每个时间段中浏览器在处理什么事情，哪个过程最消耗时间，便于我们更加详细的了解性能瓶颈

#23.2 JS 性能优化

JS 是编译型还是解释型语言其实并不固定。首先 JS 需要有引擎才能运行起来，无论是浏览器还是在 Node 中，这是解释型语言的特性。但是在 V8 引擎下，又引入了 TurboFan 编译器，它会在特定的情况下进行优化，将代码编译成执行效率更高的 Machine Code，当然这个编译器并不是 JS 必须需要的，只是为了提高代码执行性能，所以总的来说 JS 更偏向于解释型语言。

那么这一小节的内容主要会针对于 `Chrome` 的 `v8` 引擎来讲解。

在这一过程中，`JS` 代码首先会解析为抽象语法树（`AST`），然后会通过解释器或者编译器转化为 `Bytecode` 或者 `Machine Code`



从上图中我们可以发现，`JS` 会首先被解析为 `AST`，解析的过程其实是略慢的。代码越多，解析的过程也就耗费越长，这也是我们需要压缩代码的原因之一。另外一种减少解析时间的方式是预解析，会作用于未执行的函数，这个我们下面再谈

这里需要注意一点，对于函数来说，应该尽可能避免声明嵌套函数（类也是函数），因为这样会造成函数的重复解析

```
function test1() {  
    // 会被重复解析  
    function test2() {}  
}
```

然后 `Ignition` 负责将 `AST` 转化为 `Bytecode`，`TurboFan` 负责编译出优化后的 `Machine Code`，并且 `Machine Code` 在执行效率上优于 `Bytecode`

那么我们就产生了一个疑问，什么情况下代码会编译为 `Machine Code`？

`JS` 是一门动态类型的语言，并且还有一大堆的规则。简单的加法运算代码，内部就需要考虑好几种规则，比如数字相加、字符串相加、对象和字符串相加等等。这样的情况也就势必导致了内部要增加很多判断逻辑，降低运行效率。

```
function test(x) {
    return x + x
}

test(1)
test(2)
test(3)
test(4)
```

- 对于以上代码来说，如果一个函数被多次调用并且参数一直传入 `number` 类型，那么 `v8` 就会认为该段代码可以编译为 `Machine Code`，因为你固定了类型，不需要再执行很多判断逻辑了。
- 但是如果一旦我们传入的参数类型改变，那么 `Machine Code` 就会被 `Deoptimized` 为 `Bytecode`，这样就有性能上的一个损耗了。所以如果我们希望代码能多的编译为 `Machine Code` 并且 `deoptimized` 的次数减少，就应该尽可能保证传入的类型一致。
- 那么你可能会有一个疑问，到底优化前后有多少的提升呢，接下来我们就来实践测试一下到底有多少的提升

```
const { performance, PerformanceObserver } = require('perf_hooks')

function test(x) {
    return x + x
}
// node 10 中才有 PerformanceObserver
// 在这之前的 node 版本可以直接使用 performance 中的 API
const obs = new PerformanceObserver((list, observer) => {
    console.log(list.getEntries())
    observer.disconnect()
})
obs.observe({ entryTypes: ['measure'], buffered: true })

performance.mark('start')

let number = 10000000
// 不优化代码
%NeverOptimizeFunction(test)

while (number--) {
    test(1)
}

performance.mark('end')
performance.measure('test', 'start', 'end')
```

以上代码中我们使用了 `performance API`，这个 `API` 在性能测试上十分好用。不仅可以用来测量代码的执行时间，还能用来测量各种网络连接中的时间消耗等等，并且这个 `API` 也可以在浏览器中使

从上图中我们可以发现，优化过的代码执行时间只需要 `9ms`，但是不优化过的代码执行时间却是前者的二十倍，已经接近 `200ms` 了。在这个案例中，我相信大家已经看到了 `v8` 的性能优化到底有多强，只需要我们符合一定的规则书写代码，引擎底层就能帮助我们自动优化代码。

另外，编译器还有个骚操作 `Lazy-Compile`，当函数没有被执行的时候，会对函数进行一次预解析，直到代码被执行以后才会被解析编译。对于上述代码来说，`test` 函数需要被预解析一次，然后在调用的时候再被解析编译。但是对于这种函数马上就被调用的情况来说，预解析这个过程其实是多余的，那么有什么办法能够让代码不被预解析呢？

```
(function test(obj) {  
    return x + x  
})
```

但是不可能我们为了性能优化，给所有的函数都去套上括号，并且也不是所有函数都需要这样做。我们可以通过 `optimize-js` 实现这个功能，这个库会分析一些函数的使用情况，然后给需要的函数添加括号，当然这个库很久没人维护了，如果需要使用的话，还是需要测试过相关内容的。

其实很简单，我们只需要给函数套上括号就可以了

#24 性能优化

总的来说性能优化这个领域的很多内容都很碎片化，这一章节我们将来学习这些碎片化的内容。

#24.1 图片优化

计算图片大小

对于一张 `100 * 100` 像素的图片来说，图像上有 `10000` 个像素点，如果每个像素的值是 `RGBA` 存储的话，那么也就是说每个像素有 `4` 个通道，每个通道 `1` 个字节 (`8 位 = 1个字节`)，所以该图片大小大概为 `39KB` (`10000 * 1 * 4 / 1024`)。

- 但是在实际项目中，一张图片可能并不需要使用那么多颜色去显示，我们可以通过减少每个像素的调色板来相应缩小图片的大小。
- 了解了如何计算图片大小的知识，那么对于如何优化图片，想必大家已经有 2 个思路了：
 - 减少像素点
 - 减少每个像素点能够显示的颜色

#24.2 图片加载优化

- 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 `css` 去代替。
- 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 `CDN` 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
- 小图使用 `base64` 格式
- 将多个图标文件整合到一张图片中（雪碧图）
- 选择正确的图片格式：
 - 对于能够显示 `WebP` 格式的浏览器尽量使用 `WebP` 格式。因为 `WebP` 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
 - 小图使用 `PNG`，其实对于大部分图标这类图片，完全可以使用 `SVG` 代替
 - 照片使用 `JPEG`

#24.3 DNS 预解析

`DNS` 解析也是需要时间的，可以通过预解析的方式来预先获得域名所对应的 `IP`。

```
<link rel="dns-prefetch" href="//blog.poetries.top">
```

#24.4 节流

考虑一个场景，滚动事件中会发起网络请求，但是我们并不希望用户在滚动过程中一直发起请求，而是隔一段时间发起一次，对于这种情况我们就可以使用节流。

理解了节流的用途，我们就来实现下这个函数

```
// func是用户传入需要防抖的函数
// wait是等待时间
const throttle = (func, wait = 50) => {
  // 上一次执行该函数的时间
  let lastTime = 0
  return function(...args) {
    // 当前时间
    let now = +new Date()
    // 将当前时间和上一次执行函数时间对比
    // 如果差值大于设置的等待时间就执行函数
    if (now - lastTime > wait) {
      lastTime = now
      func.apply(this, args)
    }
  }
}

setInterval(
  throttle(() => {
    console.log(1)
  }, 500),
  1
)
```

#24.5 防抖

考虑一个场景，有一个按钮点击会触发网络请求，但是我们并不希望每次点击都发起网络请求，而是当用户点击按钮一段时间后没有再次点击的情况下才去发起网络请求，对于这种情况我们就可以使用防抖。

理解了防抖的用途，我们就来实现下这个函数

```
// func是用户传入需要防抖的函数
// wait是等待时间
const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设定过定时器了就清空上一次的定时器
  // 开始一个新的定时器，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
```

#24.6 预加载

- 在开发中，可能会遇到这样的情况。有些资源不需要马上用到，但是希望尽早获取，这时候就可以使用预加载。
- 预加载其实是声明式的 `fetch`，强制浏览器请求资源，并且不会阻塞 `onload` 事件，可以使用以下代码开启预加载

```
<link rel="preload" href="http://blog.poeties.top">
```

预加载可以一定程度上降低首屏的加载时间，因为可以将一些不影响首屏但重要的文件延后加载，唯一缺点就是兼容性不好。

#24.7 预渲染

可以通过预渲染将下载的文件预先在后台渲染，可以使用以下代码开启预渲染

```
<link rel="prerender" href="http://blog.poeties.top">
```

预渲染虽然可以提高页面的加载速度，但是要确保该页面大概率会被用户在之后打开，否则就是白白浪费资源去渲染。

#24.8 懒执行

懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

#24.9 懒加载

- 懒加载就是将不关键的资源延后加载。
- 懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。对于图片来说，先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 `src` 属性，这样图片就会去下载资源，实现了图片懒加载。
- 懒加载不仅可以用在图片，也可以用在别的资源上。比如进入可视区域才开始播放视频等等。

#24.10 CDN

CDN 的原理是尽可能的在各个地方分布机房缓存数据，这样即使我们的根服务器远在国外，在国内的用户也可以通过国内的机房迅速加载资源。

因此，我们可以将静态资源尽量使用 `CDN` 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 `CDN` 域名。并且对于 `CDN` 加载静态资源需要注意 `CDN` 域名要与主站不同，否则每次请求都会带上主站的 `Cookie`，白白消耗流量。

#25 Webpack 性能优化

在这部分的内容中，我们会聚焦于以下两个知识点，并且每一个知识点都属于高频考点：

- 有哪些方式可以减少 `webpack` 的打包时间
- 有哪些方式可以让 `webpack` 打出来的包更小

#25.1 减少 Webpack 打包时间

1. 优化 Loader

对于 `Loader` 来说，影响打包效率首当其冲必属 `Babel` 了。因为 `Babel` 会将代码转为字符串生成 `AST`，然后对 `AST` 继续进行转变最后再生成新的代码，项目越大，转换代码越多，效率就越低。当然了，我们是有办法优化的

首先我们可以优化 `Loader` 的文件搜索范围

```
module.exports = {
  module: {
    rules: [
      {
        // js 文件才使用 babel
        test: /\.js$/,
        loader: 'babel-loader',
        // 只在 src 文件夹下查找
        include: [resolve('src')],
        // 不会去查找的路径
        exclude: /node_modules/
      }
    ]
  }
}
```

对于 `Babel` 来说，我们肯定是希望只作用在 `js` 代码上的，然后 `node_modules` 中使用的代码都是编译过的，所以我们也完全没有必要再去处理一遍

- 当然这样做还不够，我们还可以将 `Babel` 编译过的文件缓存起来，下次只需要编译更改过的代码文件即可，这样可以大幅度加快打包时间

```
loader: 'babel-loader?cacheDirectory=true'
```

2. HappyPack

受限于 `Node` 是单线程运行的，所以 `webpack` 在打包的过程中也是单线程的，特别是在执行 `Loader` 的时候，长时间编译的任务很多，这样就会导致等待的情况。

`HappyPack` 可以将 `Loader` 的同步执行转换为并行的，这样就能充分利用系统资源来加快打包效率了

```
module: {
  loaders: [
    {
      test: /\.js$/,
      include: [resolve('src')],
      exclude: /node_modules/,
      // id 后面的内容对应下面
      loader: 'happypack/loader?id=happybabel'
    }
  ],
  plugins: [
    new HappyPack({
      id: 'happybabel',
    })
  ]
},
```

```
loaders: ['babel-loader?cacheDirectory'],
// 开启 4 个线程
threads: 4
})
]
```

3. DllPlugin

`DllPlugin` 可以将特定的类库提前打包然后引入。这种方式可以极大的减少打包类库的次数，只有当类库更新版本才有需要重新打包，并且也实现了将公共代码抽离成单独文件的优化方案。

接下来我们就来学习如何使用 `DllPlugin`

```
// 单独配置在一个文件中
// webpack.dll.conf.js
const path = require('path')
const webpack = require('webpack')
module.exports = {
  entry: {
    // 想统一打包的类库
    vendor: ['react']
  },
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].dll.js',
    library: '[name]-[hash]'
  },
  plugins: [
    new webpack.DllPlugin({
      // name 必须和 output.library 一致
      name: '[name]-[hash]',
      // 该属性需要与 DllReferencePlugin 中一致
      context: __dirname,
      path: path.join(__dirname, 'dist', '[name]-manifest.json')
    })
  ]
}
```

然后我们需要执行这个配置文件生成依赖文件，接下来我们需要使用 `DllReferencePlugin` 将依赖文件引入项目中

```
// webpack.conf.js
module.exports = {
  // ...省略其他配置
  plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      // manifest 就是之前打包出来的 json 文件
      manifest: require('./dist/vendor-manifest.json'),
    })
  ]
}
```

4. 代码压缩

在 `webpack3` 中，我们一般使用 `uglifyjs` 来压缩代码，但是这个是单线程运行的，为了加快效率，我们可以使用 `webpack-parallel-uglify-plugin` 来并行运行 `uglifyjs`，从而提高效率。

在 `Webpack4` 中，我们就不需要以上这些操作了，只需要将 `mode` 设置为 `production` 就可以默认开启以上功能。代码压缩也是我们必做的性能优化方案，当然我们不止可以压缩 `JS` 代码，还可以压缩 `HTML`、`CSS` 代码，并且在压缩 `JS` 代码的过程中，我们还可以通过配置实现比如删除 `console.log` 这类代码的功能。

5. 一些小的优化点

我们还可以通过一些小的优化点来加快打包速度

- `resolve.extensions`：用来表明文件后缀列表，默认查找顺序是 `['.js', '.json']`，如果你的导入文件没有添加后缀就会按照这个顺序查找文件。我们应该尽可能减少后缀列表长度，然后将出现频率高的后缀排在前面
- `resolve.alias`：可以通过别名的方式来映射一个路径，能让 `Webpack` 更快找到路径
- `module.noParse`：如果你确定一个文件下没有其他依赖，就可以使用该属性让 `Webpack` 不扫描该文件，这种方式对于大型的类库很有帮助

#25.2 减少 Webpack 打包后的文件体积

1. 按需加载

想必大家在开发 `SPA` 项目的时候，项目中都会存在十几甚至更多的路由页面。如果我们将这些页面全部打包进一个 `JS` 文件的话，虽然将多个请求合并了，但是同样也加载了很多并不需要的代码，耗费了更长的时间。那么为了首能在更快地呈现给用户，我们肯定是希望首能在加载的文件体积越小越好，这时候我们就可以使用按需加载，将每个路由页面单独打包为一个文件。当然不仅仅路由可以按需加载，对于 `lodash` 这种大型类库同样可以使用这个功能。

按需加载的代码实现这里就不详细展开了，因为鉴于用的框架不同，实现起来都是不一样的。当然了，虽然他们的用法可能不同，但是底层的机制都是一样的。都是当使用的时候再去下载对应文件，返回一个 `Promise`，当 `Promise` 成功以后去执行回调。

2. Scope Hoisting

`Scope Hoisting` 会分析出模块之间的依赖关系，尽可能的把打包出来的模块合并到一个函数中去。

比如我们希望打包两个文件

```
// test.js
export const a = 1

// index.js
import { a } from './test.js'
```

对于这种情况，我们打包出来的代码会类似这样

```
[  
  /* 0 */  
  function (module, exports, require) {  
    //...  
  },  
  /* 1 */  
  function (module, exports, require) {  
    //...  
  }  
]
```

但是如果我们使用 `Scope Hoisting` 的话，代码就会尽可能的合并到一个函数中去，也就变成了这样的类似代码

```
[  
  /* 0 */  
  function (module, exports, require) {  
    //...  
  }  
]
```

这样的打包方式生成的代码明显比之前的少多了。如果在 `webpack4` 中你希望开启这个功能，只需要启用 `optimization.concatenateModules` 就可以了。

```
module.exports = {  
  optimization: {  
    concatenateModules: true  
  }  
}
```

3. Tree Shaking

`Tree Shaking` 可以实现删除项目中未被引用的代码，比如

```
// test.js  
export const a = 1  
export const b = 2  
// index.js  
import { a } from './test.js'
```

- 对于以上情况，`test` 文件中的变量 `b` 如果没有在项目中使用到的话，就不会被打包到文件中。
- 如果你使用 `Webpack 4` 的话，开启生产环境就会自动启动这个优化功能。

#26 实现小型打包工具

该工具可以实现以下两个功能

- 将 `ES6` 转换为 `ES5`
- 支持在 `JS` 文件中 `import CSS` 文件

通过这个工具的实现，大家可以理解到打包工具的原理到底是什么

实现

因为涉及到 ES6 转 ES5，所以我们首先需要安装一些 Babel 相关的工具

```
yarn add babylon babel-traverse babel-core babel-preset-env
```

接下来我们将这些工具引入文件中

```
const fs = require('fs')
const path = require('path')
const babylon = require('babylon')
const traverse = require('babel-traverse').default
const { transformFromAst } = require('babel-core')
```

首先，我们先来实现如何使用 Babel 转换代码

```
function readCode(filePath) {
  // 读取文件内容
  const content = fs.readFileSync(filePath, 'utf-8')
  // 生成 AST
  const ast = babylon.parse(content, {
    sourceType: 'module'
  })
  // 寻找当前文件的依赖关系
  const dependencies = []
  traverse(ast, {
    ImportDeclaration: ({ node }) => {
      dependencies.push(node.source.value)
    }
  })
  // 通过 AST 将代码转为 ES5
  const { code } = transformFromAst(ast, null, {
    presets: ['env']
  })
  return {
    filePath,
    dependencies,
    code
  }
}
```

- 首先我们传入一个文件路径参数，然后通过 `fs` 将文件中的内容读取出来
- 接下来我们通过 `babylon` 解析代码获取 `AST`，目的是为了分析代码中是否还引入了别的文件
- 通过 `dependencies` 来存储文件中的依赖，然后再将 `AST` 转换为 `ES5` 代码
- 最后函数返回了一个对象，对象中包含了当前文件路径、当前文件依赖和当前文件转换后的代码

接下来我们需要实现一个函数，这个函数的功能有以下几点

- 调用 `readCode` 函数，传入入口文件
- 分析入口文件的依赖
- 识别 `js` 和 `css` 文件

```
function getDependencies(entry) {
  // 读取入口文件
  const entryObject = readCode(entry)
  const dependencies = [entryObject]
  // 遍历所有文件依赖关系
```

```

for (const asset of dependencies) {
  // 获得文件目录
  const dirname = path.dirname(asset.filePath)
  // 遍历当前文件依赖关系
  asset.dependencies.forEach(relativePath => {
    // 获得绝对路径
    const absolutePath = path.join(dirname, relativePath)
    // CSS 文件逻辑就是将代码插入到 `style` 标签中
    if (/\.css$/.test(absolutePath)) {
      const content = fs.readFileSync(absolutePath, 'utf-8')
      const code =
        const style = document.createElement('style')
        style.innerText = `${JSON.stringify(content).replace(/\r\n/g, '')}`
        document.head.appendChild(style)
    }

    dependencies.push({
      filePath: absolutePath,
      relativePath,
      dependencies: [],
      code
    })
  } else {
    // JS 代码需要继续查找是否有依赖关系
    const child = readCode(absolutePath)
    child.relativePath = relativePath
    dependencies.push(child)
  }
})
}

return dependencies
}

```

- 首先我们读取入口文件，然后创建一个数组，该数组的目的是存储代码中涉及到的所有文件
- 接下来我们遍历这个数组，一开始这个数组中只有入口文件，在遍历的过程中，如果入口文件有依赖其他的文件，那么就会被 `push` 到这个数组中
- 在遍历的过程中，我们先获得该文件对应的目录，然后遍历当前文件的依赖关系
- 在遍历当前文件依赖关系的过程中，首先生成依赖文件的绝对路径，然后判断当前文件是 `css` 文件还是 `js` 文件
- 如果是 `css` 文件的话，我们就不能用 `Babel` 去编译了，只需要读取 `css` 文件中的代码，然后创建一个 `style` 标签，将代码插入进标签并且放入 `head` 中即可
- 如果是 `js` 文件的话，我们还需要分析 `js` 文件是否还有别的依赖关系
- 最后将读取文件后的对象 `push` 进数组中
- 现在我们已经获取到了所有的依赖文件，接下来就是实现打包的功能了

```

function bundle(dependencies, entry) {
  let modules = ''
  // 构建函数参数，生成的结构为
  // { './entry.js': function(module, exports, require) { 代码 } }
  dependencies.forEach(dep => {
    const filePath = dep.relativePath || entry
    modules += `${filePath}: (` +
      function (module, exports, require) { ${dep.code} }
    ),` +
  })
  // 构建 require 函数，目的是为了获取模块暴露出来的内容
}

```

```

const result = `
(function(modules) {
    function require(id) {
        const module = { exports: {} }
        modules[id](module, module.exports, require)
        return module.exports
    }
    require('${entry}')
})(${modules})
` // 生成的内容写入到文件中
fs.writeFileSync('./bundle.js', result)
}

```

这段代码需要结合着 `Babel` 转换后的代码来看，这样大家就能理解为什么需要这样写了

```

// entry.js
var _a = require('./a.js')
var _a2 = _interopRequireDefault(_a)
function _interopRequireDefault(obj) {
    return obj && obj.__esModule ? obj : { default: obj }
}
console.log(_a2.default)
// a.js
Object.defineProperty(exports, '__esModule', {
    value: true
})
var a = 1
exports.default = a

```

`Babel` 将我们 `ES6` 的模块化代码转换为了 `CommonJS` 的代码，但是浏览器是不支持 `CommonJS` 的，所以如果这段代码需要在浏览器环境下运行的话，我们需要自己实现 `CommonJS` 相关的代码，这就是 `bundle` 函数做的大部分事情。

接下来我们再来逐行解析 `bundle` 函数

- 首先遍历所有依赖文件，构建出一个函数参数对象
- 对象的属性就是当前文件的相对路径，属性值是一个函数，函数体是当前文件下的代码，函数接受三个参数

module

,

exports

,

require

- `module` 参数对应 `CommonJS` 中的 `module`
- `exports` 参数对应 `CommonJS` 中的 `module.export`
- `require` 参数对应我们自己创建的 `require` 函数

- 接下来就是构造一个使用参数的函数了，函数做的事情很简单，就是内部创建一个 `require` 函数，然后调用 `require(entry)`，也就是 `require('./entry.js')`，这样就会从函数参数中找到 `./entry.js` 对应的函数并执行，最后将导出的内容通过 `module.exports` 的方式让外部获取到
- 最后再将打包出来的内容写入到单独的文件中

| 如果你对于上面的实现还有疑惑的话，可以阅读下打包后的部分简化代码

```
; (function(modules) {
  function require(id) {
    // 构造一个 CommonJS 导出代码
    const module = { exports: {} }
    // 去参数中获取文件对应的函数并执行
    modules[id](module, module.exports, require)
    return module.exports
  }
  require('./entry.js')
})({
  './entry.js': function(module, exports, require) {
    // 这里继续通过构造的 require 去找到 a.js 文件对应的函数
    var _a = require('./a.js')
    console.log(_a.default)
  },
  './a.js': function(module, exports, require) {
    var a = 1
    // 将 require 函数中的变量 module 变成了这样的结构
    // module.exports = 1
    // 这样就能在外部取到导出的内容了
    exports.default = a
  }
  // 省略
})
```

| 虽然实现这个工具只写了不到 100 行的代码，但是打包工具的核心原理就是这些了

- 找出入口文件所有的依赖关系
- 然后通过构建 `CommonJS` 代码来获取 `exports` 导出的内容

#27 MVVM/虚拟DOM/前端路由

#27.1 MVVM

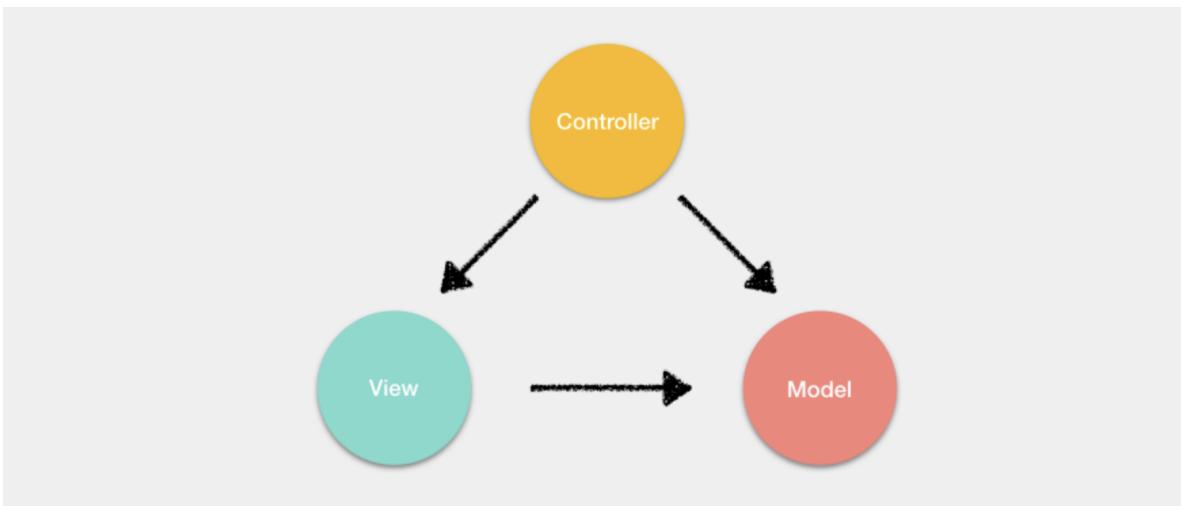
| 涉及面试题：什么是 `MVVM`？比之 `MVC` 有什么区别？

首先先来说下 View 和 Model

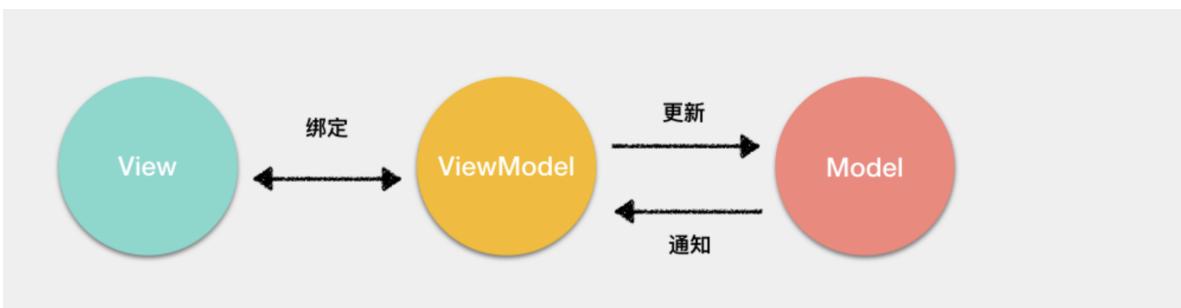
- `View` 很简单，就是用户看到的视图
- `Model` 同样很简单，一般就是本地数据和数据库中的数据

| 基本上，我们写的产品就是通过接口从数据库中读取数据，然后将数据经过处理展现到用户看到的视图上。当然我们还可以从视图上读取用户的输入，然后又将用户的输入通过接口写入到数据库中。但是，如何将数据展示到视图上，然后又如何将用户的输入写入到数据中，不同的人就产生了不同的看法，从此出现了很多种架构设计。

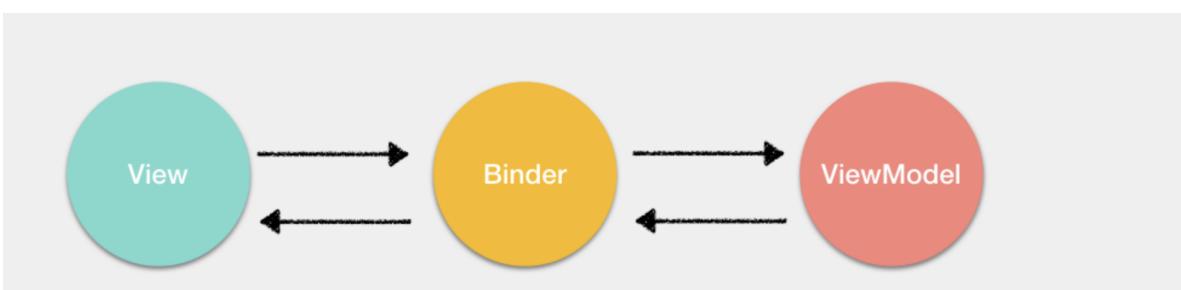
传统的 `MVC` 架构通常是使用控制器更新模型，视图从模型中获取数据去渲染。当用户有输入时，会通过控制器去更新模型，并且通知视图进行更新



- 但是 `MVC` 有一个巨大的缺陷就是控制器承担的责任太大了，随着项目愈加复杂，控制器中的代码会越来越臃肿，导致出现不利于维护的情况。
- 在 `MVVM` 架构中，引入了 `viewModel` 的概念。`viewModel` 只关心数据和业务的处理，不关心 `View` 如何处理数据，在这种情况下，`View` 和 `Model` 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 `viewModel` 中，让多个 `View` 复用这个 `viewModel`。



- 以 `vue` 框架来举例，`viewModel` 就是组件的实例。`view` 就是模板，`Model` 的话在引入 `vuex` 的情况下是完全可以和组件分离的。
- 除了以上三个部分，其实在 `MVVM` 中还引入了一个隐式的 `Binder` 层，实现了 `View` 和 `ViewModel` 的绑定



- 同样以 `vue` 框架来举例，这个隐式的 `Binder` 层就是 `vue` 通过解析模板中的插值和指令从而实现 `View` 与 `ViewModel` 的绑定。
- 对于 `MVVM` 来说，其实最重要的并不是通过双向绑定或者其他的方式将 `View` 与 `ViewModel` 绑定起来，而是通过 `viewModel` 将视图中的状态和用户的行为分离出一个抽象，这才是 `MVVM` 的精髓

#27.2 Virtual DOM

涉及面试题：什么是 `Virtual DOM`？为什么 `Virtual DOM` 比原生 `DOM` 快？

- 大家都知道操作 `DOM` 是很慢的，为什么慢的原因以及在「浏览器渲染原理」章节中说过，这里就不再赘述了。那么相较于 `DOM` 来说，操作 `JS` 对象会快很多，并且我们也可以通过 `JS` 来模拟 `DOM`。

```
const ul = {
  tag: 'ul',
  props: {
    class: 'list'
  },
  children: {
    tag: 'li',
    children: '1'
  }
}
```

上述代码对应的 `DOM` 就是

```
<ul class='list'>
  <li>1</li>
</ul>
```

- 那么既然 `DOM` 可以通过 `JS` 对象来模拟，反之也可以通过 `JS` 对象来渲染出对应的 `DOM`。当然了，通过 `JS` 来模拟 `DOM` 并且渲染对应的 `DOM` 只是第一步，难点在于如何判断新旧两个 `JS` 对象的最小差异并且实现局部更新 `DOM`。

首先 `DOM` 是一个多叉树的结构，如果需要完整的对比两棵树的差异，那么需要的时间复杂度会是 $O(n^3)$ ，这个复杂度肯定是不能接受的。于是 `React` 团队优化了算法，实现了 $O(n)$ 的复杂度来对比差异。实现 $O(n)$ 复杂度的关键就是只对比同层的节点，而不是跨层对比，这也是考虑到在实际业务中很少会去跨层的移动 `DOM` 元素。所以判断差异的算法就分为了两步。

- 首先从上至下，从左往右遍历对象，也就是树的深度遍历，这一步中会给每个节点添加索引，便于最后渲染差异。
- 一旦节点有子元素，就去判断子元素是否有不同。

在第一步算法中我们需要判断新旧节点的 `tagName` 是否相同，如果不相同的话就代表节点被替换了。如果没有更改 `tagName` 的话，就需要判断是否有子元素，有的话就进行第二步算法。

在第二步算法中，我们需要判断原本的列表中是否有节点被移除，在新的列表中需要判断是否有新的节点加入，还需要判断节点是否有移动。

举个例子来说，假设页面中只有一个列表，我们对列表中的元素进行了变更

```
// 假设这里模拟一个 ul，其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

从上述例子中，我们一眼就可以看出先前的 `ul` 中的第三个 `li` 被移除了，四五替换了位置。

那么在实际的算法中，我们如何去识别改动的是哪个节点呢？这就引入了 `key` 这个属性，想必大家在 `Vue` 或者 `React` 的列表中都用过这个属性。这个属性是用来给每一个节点打标志的，用于判断是否是同一个节点。

- 当然在判断以上差异的过程中，我们还需要判断节点的属性是否有变化等等。
- 当我们判断出以上的差异后，就可以把这些差异记录下来。当对比完两棵树以后，就可以通过差异去局部更新 `DOM`，实现性能的最优化。

当然了 `virtual DOM` 提高性能是其中一个优势，其实最大的优势还是在于：

- 将 `virtual DOM` 作为一个兼容层，让我们还能对接非 `web` 端的系统，实现跨端开发。
- 同样的，通过 `virtual DOM` 我们可以渲染到其他的平台，比如实现 `SSR`、同构渲染等等。
- 实现组件的高度抽象化

#27.3 路由原理

涉及面试题：前端路由原理？两种实现方式有什么区别？

前端路由实现起来其实很简单，本质就是监听 `URL` 的变化，然后匹配路由规则，显示相应的页面，并且无须刷新页面。目前前端使用的路由就只有两种实现方式

- `Hash` 模式
- `History` 模式

1. Hash 模式

`www.test.com/#/`` 就是 `Hash URL`，当 `#` 后面的哈希值发生变化时，可以通过 `hashchange` 事件来监听到 `URL` 的变化，从而进行跳转页面，并且无论哈希值如何变化，服务端接收到的 `URL` 请求永远是 `www.test.com`

```
window.addEventListener('hashchange', () => {  
    // ... 具体逻辑  
})
```

`Hash` 模式相对来说更简单，并且兼容性也更好

2. History 模式

`History` 模式是 `HTML5` 新推出的功能，主要使用 `history.pushState` 和 `history.replaceState` 改变 `URL`

- 通过 `History` 模式改变 `URL` 同样不会引起页面的刷新，只会更新浏览器的历史记录。

```
// 新增历史记录  
history.pushState(stateObject, title, URL)  
// 替换当前历史记录  
history.replaceState(stateObject, title, URL)
```

当用户做出浏览器动作时，比如点击后退按钮时会触发 `popstate` 事件

```
window.addEventListener('popstate', e => {  
    // e.state 就是 pushState(stateObject) 中的 stateObject  
    console.log(e.state)  
})
```

两种模式对比

- `Hash` 模式只可以更改 `#` 后面的内容，`History` 模式可以通过 API 设置任意的同源 `URL`

- `History` 模式可以通过 API 添加任意类型的数据到历史记录中，`Hash` 模式只能更改哈希值，也就是字符串
- `Hash` 模式无需后端配置，并且兼容性好。`History` 模式在用户手动输入地址或者刷新页面的时候会发起 URL 请求，后端需要配置 `index.html` 页面用于匹配不到静态资源的时候

#27.4 Vue 和 React 之间的区别

- `Vue` 的表单可以使用 `v-model` 支持双向绑定，相比于 `React` 来说开发上更加方便，当然了 `v-model` 其实就是个语法糖，本质上和 `React` 写表单的方式没什么区别
- 改变数据方式不同，`Vue` 修改状态相比来说要简单许多，`React` 需要使用 `setState` 来改变状态，并且使用这个 API 也有一些坑点。并且 `Vue` 的底层使用了依赖追踪，页面更新渲染已经是最优的了，但是 `React` 还是需要用户手动去优化这方面的问题。
- `React 16` 以后，有些钩子函数会执行多次，这是因为引入 `Fiber` 的原因
- `React` 需要使用 `JSX`，有一定的上手成本，并且需要一整套的工具链支持，但是完全可以通过 `JS` 来控制页面，更加灵活。`Vue` 使用了模板语法，相比于 `JSX` 来说没有那么灵活，但是完全可以脱离工具链，通过直接编写 `render` 函数就能在浏览器中运行。
- 在生态上来说，两者其实没多大的差距，当然 `React` 的用户是远远高于 `Vue` 的

#28 Vue 常考知识点

#28.1 生命周期钩子函数

- 在 `beforeCreate` 钩子函数调用的时候，是获取不到 `props` 或者 `data` 中的数据的，因为这些数据的初始化都在 `initState` 中。
- 然后会执行 `created` 钩子函数，在这一步的时候已经可以访问到之前不能访问到的数据，但是这时候组件还没被挂载，所以是看不到的。
- 接下来会先执行 `beforeMount` 钩子函数，开始创建 `VDOM`，最后执行 `mounted` 钩子，并将 `VDOM` 渲染为真实 `DOM` 并且渲染数据。组件中如果有子组件的话，会递归挂载子组件，只有当所有子组件全部挂载完毕，才会执行根组件的挂载钩子。
- 接下来是数据更新时会调用的钩子函数 `beforeUpdate` 和 `updated`，这两个钩子函数没什么好说的，就是分别在数据更新前和更新后会调用。
- 另外还有 `keep-alive` 独有的生命周期，分别为 `activated` 和 `deactivated`。用 `keep-alive` 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 `deactivated` 钩子函数，命中缓存渲染后会执行 `activated` 钩子函数。
- 最后就是销毁组件的钩子函数 `beforeDestroy` 和 `destroyed`。前者适合移除事件、定时器等，否则可能会引起内存泄露的问题。然后进行一系列的销毁操作，如果有子组件的话，也会递归销毁子组件，所有子组件都销毁完毕后才会执行根组件的 `destroyed` 钩子函数

#28.2 组件通信

组件通信一般分为以下几种情况：

- 父子组件通信
- 兄弟组件通信
- 跨多层级组件通信

对于以上每种情况都有多种方式去实现，接下来就来学习下如何实现。

1. 父子通信

- 父组件通过 `props` 传递数据给子组件，子组件通过 `emit` 发送事件传递数据给父组件，这两种方式是最常用的父子通信实现办法。

- 这种父子通信方式也就是典型的单向数据流，父组件通过 `props` 传递数据，子组件不能直接修改 `props`，而是必须通过发送事件的方式告知父组件修改数据。
- 另外这两种方式还可以使用语法糖 `v-model` 来直接实现，因为 `v-model` 默认会解析成名为 `value` 的 `prop` 和名为 `input` 的事件。这种语法糖的方式是典型的双向绑定，常用于 `UI` 控件上，但是究其根本，还是通过事件的方法让父组件修改数据。
- 当然我们还可以通过访问 `$parent` 或者 `$children` 对象来访问组件实例中的方法和数据。
- 另外如果你使用 Vue 2.3 及以上版本的话还可以使用 `$listeners` 和 `.sync` 这两个属性。
- `$listeners` 属性会将父组件中的（不含 `.native` 修饰器的）`v-on` 事件监听器传递给子组件，子组件可以通过访问 `$listeners` 来自定义监听器。
- `.sync` 属性是个语法糖，可以很简单的实现子组件与父组件通信

```
<!--父组件中-->
<input :value.sync="value" />
<!--以上写法等同于-->
<input :value="value" @update:value="v => value = v"></comp>
<!--子组件中-->
<script>
  this.$emit('update:value', 1)
</script>
```

2. 兄弟组件通信

对于这种情况可以通过查找父组件中的子组件实现，也就是 `this.$parent.$children`，在 `$children` 中可以通过组件 `name` 查询到需要的组件实例，然后进行通信。

3. 跨多层次组件通信

对于这种情况可以使用 `vue 2.2` 新增的 `API provide / inject`，虽然文档中不推荐直接使用在业务中，但是如果用得好的话还是很有用的。

假设有父组件 `A`，然后有一个跨多层级的子组件 `B`

```
// 父组件 A
export default {
  provide: {
    data: 1
  }
}
// 子组件 B
export default {
  inject: ['data'],
  mounted() {
    // 无论跨几层都能获得父组件的 data 属性
    console.log(this.data) // => 1
  }
}
```

终极办法解决一切通信问题

只要你不怕麻烦，可以使用 `Vuex` 或者 `Event Bus` 解决上述所有的通信情况。

#28.3 extend 能做什么

这个 API 很少用到，作用是扩展组件生成一个构造器，通常会与 \$mount 一起使用。

```
// 创建组件构造器
let Component = Vue.extend({
  template: '<div>test</div>'
})
// 挂载到 #app 上
new Component().$mount('#app')
// 除了上面的方式，还可以用来扩展已有的组件
let SuperComponent = Vue.extend(Component)
new SuperComponent({
  created() {
    console.log(1)
  }
})
new SuperComponent().$mount('#app')
```

#28.4 mixin 和 mixins 区别

mixin 用于全局混入，会影响到每个组件实例，通常插件都是这样做初始化的

```
Vue.mixin({
  beforeCreate() {
    // ...逻辑
    // 这种方式会影响到每个组件的 beforeCreate 钩子函数
  }
})
```

- 虽然文档不建议我们在应用中直接使用 mixin，但是如果不用的话也是很有帮助的，比如可以全局混入封装好的 ajax 或者一些工具函数等等。
- mixin 应该是我们最常使用的扩展组件的方式了。如果多个组件中有相同的业务逻辑，就可以将这些逻辑剥离出来，通过 mixins 混入代码，比如上拉加载数据这种逻辑等等。
- 另外需要注意的是 mixins 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会有选择性的进行合并，具体可以阅读文档。

#28.5 computed 和 watch 区别

- computed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计算值变化才会返回内容。
- watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。
- 所以一般来说需要依赖别的属性来动态获得值的时候可以使用 computed，对于监听到值的变化需要做一些复杂业务逻辑的情况可以使用 watch。
- 另外 computer 和 watch 还都支持对象的写法，这种方式知道的人并不多。

```
vm.$watch('obj', {
  // 深度遍历
  deep: true,
  // 立即触发
  immediate: true,
  // 执行的函数
  handler: function(val, oldVal) {}
})
```

```

var vm = new Vue({
  data: { a: 1 },
  computed: {
    aPlus: {
      // this.aPlus 时触发
      get: function () {
        return this.a + 1
      },
      // this.aPlus = 1 时触发
      set: function (v) {
        this.a = v - 1
      }
    }
  }
})

```

#28.6 keep-alive 组件有什么作用

- 如果你需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 `keep-alive` 组件包裹需要保存的组件。
- 对于 `keep-alive` 组件来说，它拥有两个独有的生命周期钩子函数，分别为 `activated` 和 `deactivated`。用 `keep-alive` 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 `deactivated` 钩子函数，命中缓存渲染后会执行 `activated` 钩子函数。

#28.7 v-show 与 v-if 区别

- `v-show` 只是在 `display: none` 和 `display: block` 之间切换。无论初始条件是什么都会被渲染出来，后面只需要切换 CSS，DOM 还是一直保留着的。所以总的来说 `v-show` 在初始渲染时有更高的开销，但是切换开销很小，更适合于频繁切换的场景。
- `v-if` 的话就得说到 `Vue` 底层的编译了。当属性初始为 `false` 时，组件就不会被渲染，直到条件为 `true`，并且切换条件时会触发销毁/挂载组件，所以总的来说在切换时开销更高，更适合不经常切换的场景。
- 并且基于 `v-if` 的这种惰性渲染机制，可以在必要的时候才去渲染组件，减少整个页面的初始渲染开销。

#28.8 组件中 data 什么时候可以使用对象

这道题目其实更多考的是 JS 功底。

- 组件复用时所有组件实例都会共享 `data`，如果 `data` 是对象的话，就会造成一个组件修改 `data` 以后会影响到其他所有组件，所以需要将 `data` 写成函数，每次用到就调用一次函数获得新的数据。
- 当我们使用 `new Vue()` 的方式的时候，无论我们将 `data` 设置为对象还是函数都是可以的，因为 `new Vue()` 的方式是生成一个根组件，该组件不会复用，也就不存在共享 `data` 的情况了

以下是进阶部分

#28.9 响应式原理

`Vue` 内部使用了 `Object.defineProperty()` 来实现数据响应式，通过这个函数可以监听到 `set` 和 `get` 的事件

```

var data = { name: 'poetries' }
observe(data)
let name = data.name // -> get value

```

```

data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(obj).forEach(key => {
    defineReactive(obj, key, obj[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    // 可枚举
    enumerable: true,
    // 可配置
    configurable: true,
    // 自定义函数
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}

```

以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的，因为自定义的函数一开始是不会执行的。只有先执行了依赖收集，从能在属性更新的时候派发更新，所以接下来我们需要先触发依赖收集

```

<div>
  {{name}}
</div>

```

- 在解析如上模板代码时，遇到 `name` 就会进行依赖收集。
- 接下来我们先来实现一个 `Dep` 类，用于解耦属性的依赖收集和派发更新操作

```

// 通过 Dep 解耦属性的依赖和更新操作
class Dep {
  constructor() {
    this.subs = []
  }
  // 添加依赖
  addSub(sub) {
    this.subs.push(sub)
  }
  // 更新
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}

```

```
        })
    }
}

// 全局属性，通过该属性配置 watcher
Dep.target = null
```

以上的代码实现很简单，当需要依赖收集的时候调用 `addSub`，当需要派发更新的时候调用 `notify`。

接下来我们先来简单的了解下 `vue` 组件挂载时添加响应式的过程。在组件挂载时，会先对所有需要的属性调用 `Object.defineProperty()`，然后实例化 `watcher`，传入组件更新的回调。在实例化过程中，会对模板中的属性进行求值，触发依赖收集。

因为这一小节主要目的是学习响应式原理的细节，所以接下来的代码会简略的表达触发依赖收集时的操作。

```
class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
```

以上就是 `Watcher` 的简单实现，在执行构造函数的时候将 `Dep.target` 指向自身，从而使得收集到了对应的 `Watcher`，在派发更新的时候取出对应的 `Watcher` 然后执行 `update` 函数。

接下来，需要对 `defineReactive` 函数进行改造，在自定义函数中添加依赖收集和派发更新相关的代码

```
function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
      return val
    },
  })
}
```

```

set: function reactiveSetter(newVal) {
  console.log('change value')
  val = newVal
  // 执行 watcher 的 update 方法
  dp.notify()
}
})
}

```

以上所有代码实现了一个简易的数据响应式，核心思路就是手动触发一次属性的 `getter` 来实现依赖收集。

现在我们就来测试下代码的效果，只需要把所有的代码复制到浏览器中执行，就会发现页面的内容全部被替换了

```

var data = { name: 'poetries' }
observe(data)
function update(value) {
  document.querySelector('div').innerText = value
}
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'

```

#28.9.1 Object.defineProperty 的缺陷

- 以上已经分析完了 `Vue` 的响应式原理，接下来说一点 `Object.defineProperty` 中的缺陷。
- 如果通过下标方式修改数组数据或者给对象新增属性并不会触发组件的重新渲染，因为 `Object.defineProperty` 不能拦截到这些操作，更精确的来说，对于数组而言，大部分操作都是拦截不到的，只是 `Vue` 内部通过重写函数的方式解决了这个问题。
- 对于第一个问题，`Vue` 提供了一个 `API` 解决

```

export function set (target: Array<any> | Object, key: any, val: any): any {
  // 判断是否为数组且下标是否有效
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    // 调用 splice 函数触发派发更新
    // 该函数已被重写
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  // 判断 key 是否已经存在
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  // 如果对象不是响应式对象，就赋值返回
  if (!ob) {
    target[key] = val
    return val
  }
  // 进行双向绑定
  defineReactive(ob.value, key, val)
  // 手动派发更新
}
```

```
    ob.dep.notify()
    return val
}
```

对于数组而言，`Vue` 内部重写了以下函数实现派发更新

```
// 获得数组原型
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// 重写以下函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 缓存原生函数
  const original = arrayProto[method]
  // 重写函数
  def(arrayMethods, method, function mutator (...args) {
    // 先调用原生函数获得结果
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    // 调用以下几个函数时，监听新数据
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 手动派发更新
    ob.dep.notify()
    return result
  })
})
```

#28.9.2 编译过程

想必大家在使用 `Vue` 开发的过程中，基本都是使用模板的方式。那么你有过「模板是怎么在浏览器中运行的」这种疑虑嘛？

- 首先直接把模板丢到浏览器中肯定是不能运行的，模板只是为了方便开发者进行开发。`Vue` 会通过编译器将模板通过几个阶段最终编译为 `render` 函数，然后通过执行 `render` 函数生成 `Virtual DOM` 最终映射为真实 `DOM`。
- 接下来我们就来学习这个编译的过程，了解这个过程中大概发生了什么事情。**这个过程其中又分为三个阶段**，分别为：

- 将模板解析为 `AST`
- 优化 `AST`
- 将 `AST` 转换为 `render` 函数

在第一个阶段中，最主要的事情还是通过各种各样的正则表达式去匹配模板中的内容，然后将内容提取出来做各种逻辑操作，接下来会生成一个最基本的 `AST` 对象

```
{
  // 类型
  type: 1,
  // 标签
  tag,
  // 属性列表
  attrsList: attrs,
  // 属性映射
  attrsMap: makeAttrsMap(attrs),
  // 父节点
  parent,
  // 子节点
  children: []
}
```

- 然后会根据这个最基本的 `AST` 对象中的属性，进一步扩展 `AST`。
- 当然在这一阶段中，还会进行其他的一些判断逻辑。比如说对比前后开闭标签是否一致，判断根组件是否只存在一个，判断是否符合 `HTML5 Content Model` 规范等等问题。
- 接下来就是优化 `AST` 的阶段。在当前版本下，`Vue` 进行的优化内容其实还是不多的。只是对节点进行了静态内容提取，也就是将永远不会变动的节点提取了出来，实现复用 `Virtual DOM`，跳过对比算法的功能。在下一个大版本中，`Vue` 会在优化 `AST` 的阶段继续发力，实现更多的优化功能，尽可能的在编译阶段压榨更多的性能，比如说提取静态的属性等等优化行为。
- 最后一个阶段就是通过 `AST` 生成 `render` 函数了。其实这一阶段虽然分支有很多，但是最主要的目的就是遍历整个 `AST`，根据不同的条件生成不同的代码罢了。

#28.9.3 NextTick 原理分析

`nextTick` 可以让我们在下次 `DOM` 更新循环结束之后执行延迟回调，用于获得更新后的 `DOM`。

- 在 `Vue 2.4` 之前都是使用的 `microtasks`，但是 `microtasks` 的优先级过高，在某些情况下可能会出现比事件冒泡更快的情况，但如果都使用 `macrotasks` 又可能会出现渲染的性能问题。所以在新版本中，会默认使用 `microtasks`，但在特殊情况下会使用 `macrotasks`，比如 `v-on`。
- 对于实现 `macrotasks`，会先判断是否能使用 `setImmediate`，不能的话降级为 `MessageChannel`，以上都不行的话就使用 `setTimeout`

```
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (
  typeof MessageChannel !== 'undefined' &&
  (isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
  const channel = new MessageChannel()
  const port = channel.port2
  channel.port1.onmessage = flushCallbacks
}
```

```
macroTimerFunc = () => {
  port.postMessage(1)
}
} else {
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

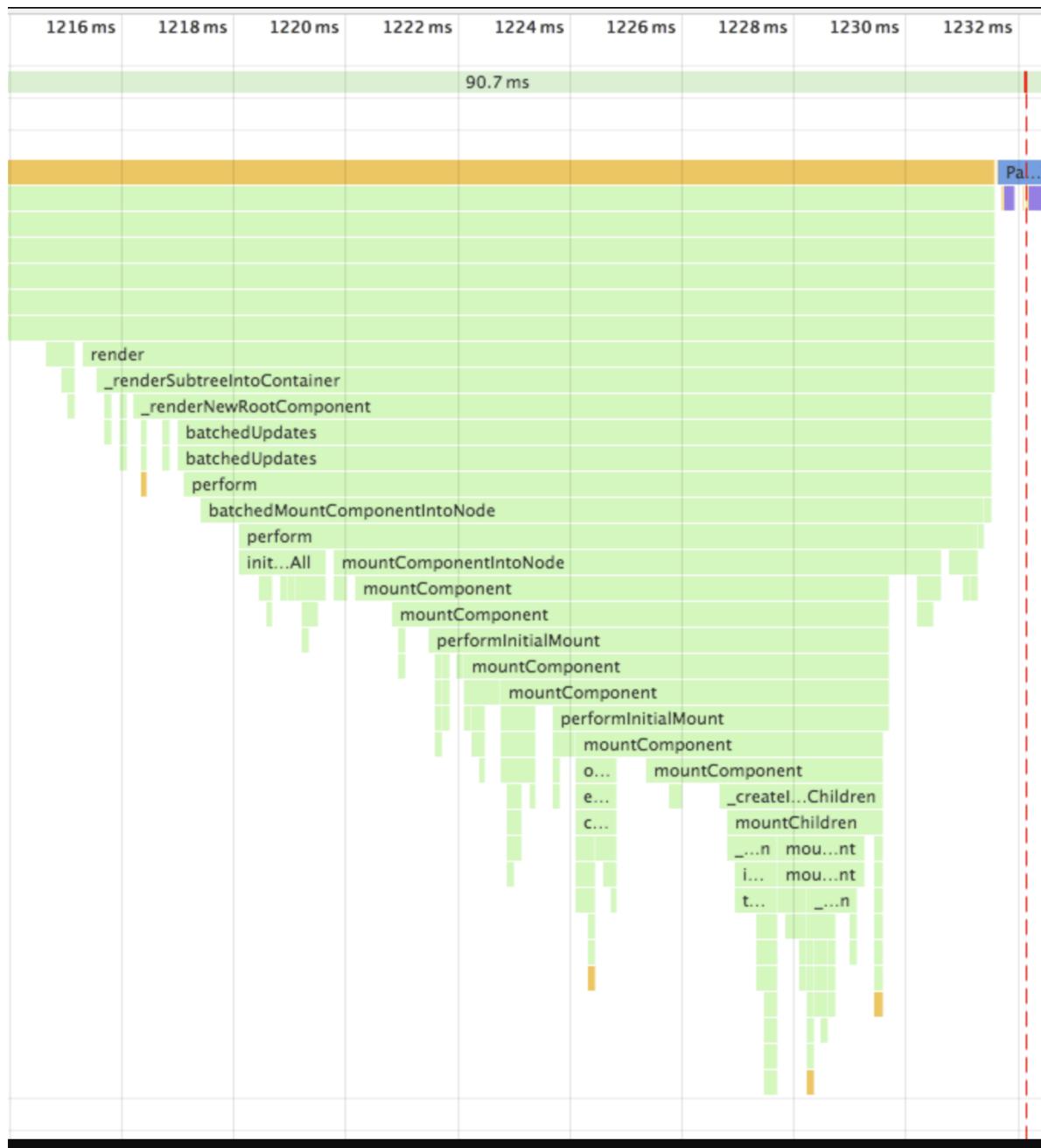
以上代码很简单，就是判断能不能使用相应的 API

#29 React常考知识点

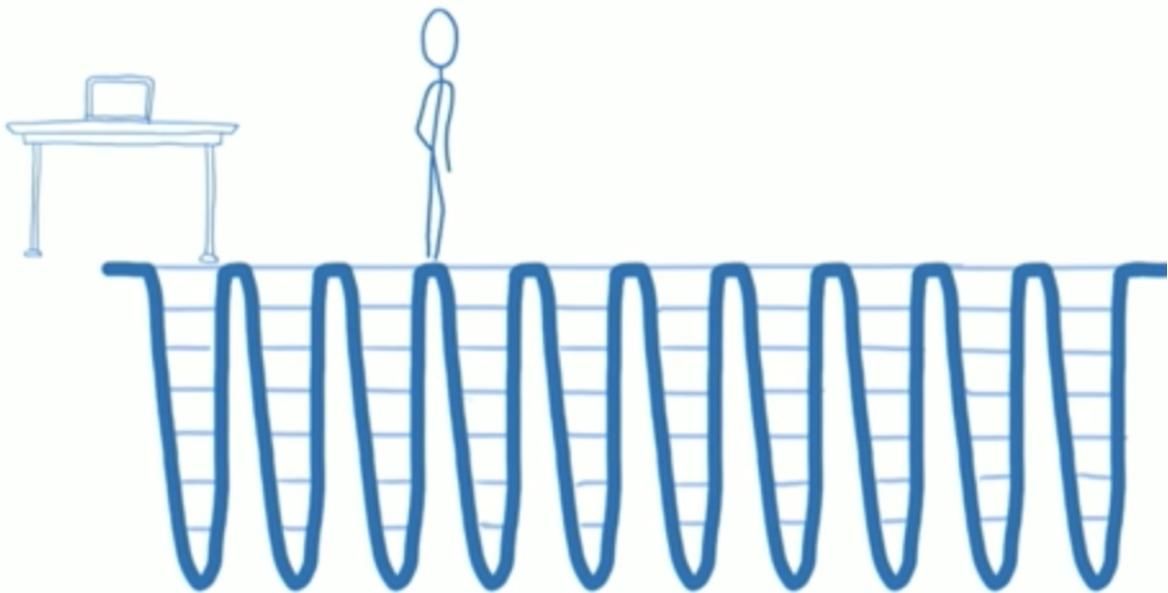
#29.1 生命周期

在 v16 版本中引入了 Fiber 机制。这个机制一定程度上的影响了部分生命周期的调用，并且也引入了新的 2 个 API 来解决问题

在之前的版本中，如果你拥有一个很复杂的复合组件，然后改动了最上层组件的 state，那么调用栈可能会很长



- 调用栈过长，再加上中间进行了复杂的操作，就可能导致长时间阻塞主线程，带来不好的用户体验。`Fiber` 就是为了解决该问题而生
- `Fiber` 本质上是一个虚拟的堆栈帧，新的调度器会按照优先级自由调度这些帧，从而将之前的同步渲染改成了异步渲染，在不影响体验的情况下分段计算更新



- 对于如何区别优先级，`React` 有自己的一套逻辑。对于动画这种实时性很高的东西，也就是 `16 ms` 必须渲染一次保证不卡顿的情况下，`React` 会每 `16 ms` (以内) 暂停一下更新，返回来继续渲染动画
- 对于异步渲染，现在渲染有两个阶段：`reconciliation` 和 `commit`。前者过程是可以打断的，后者不能暂停，会一直更新界面直到完成。

1. Reconciliation 阶段

- `componentWillMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`

2. Commit 阶段

- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`

因为 `Reconciliation` 阶段是可以被打断的，所以 `Reconciliation` 阶段会执行的生命周期函数就可能会出现调用多次的情况，从而引起 `Bug`。由此对于 `Reconciliation` 阶段调用的几个函数，除了 `shouldComponentUpdate` 以外，其他都应避免去使用，并且 `v16` 中也引入了新的 `API` 来解决这个问题。

`getDerivedStateFromProps` 用于替换 `componentWillReceiveProps`，该函数会在初始化和 `update` 时被调用

```
class ExampleComponent extends React.Component {  
  // Initialize state in constructor,  
  // Or with a property initializer.  
  state = {};  
  
  static getDerivedStateFromProps(nextProps, prevState) {  
    if (prevState.someMirroredValue !== nextProps.someValue) {  
      return {  
        derivedData: computeDerivedState(nextProps),  
        someMirroredValue: nextProps.someValue  
      };  
    }  
  }  
}
```

```
}

// Return null to indicate no change to state.
return null;
}
}
```

`getSnapshotBeforeUpdate` 用于替换 `componentwillupdate`，该函数会在 `update` 后 `DOM` 更新前被调用，用于读取最新的 `DOM` 数据

更多详情 <http://blog.poeties.top/2018/11/18/react-lifecycle>

#29.2 setState

- `setState` 在 `React` 中是经常使用的一个 `API`，但是它存在一些的问题经常会导致初学者出错，核心原因就是因为这个 `API` 是异步的。
- 首先 `setState` 的调用并不会马上引起 `state` 的改变，并且如果你一次调用了多个 `setState`，那么结果可能并不如你期待的一样。

```
handle() {
  // 初始化 `count` 为 0
  console.log(this.state.count) // -> 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // -> 0
}
```

- 第一，两次的打印都为 `0`，因为 `setState` 是个异步 `API`，只有同步代码运行完毕才会执行。`setState` 异步的原因我认为在于，`setState` 可能会导致 `DOM` 的重绘，如果调用一次就马上进行重绘，那么调用多次就会造成不必要的性能损失。设计成异步的话，就可以将多次调用放入一个队列中，在恰当的时候统一进行更新过程。

```
Object.assign(
  {},
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
)
```

当然你也可以通过以下方式来实现调用三次 `setState` 使得 `count` 为 `3`

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
}
```

如果你想在每次调用 `setState` 后获得正确的 `state`，可以通过如下代码实现

```
handle() {
    this.setState((prevState) => ({ count: prevState.count + 1 }), () => {
        console.log(this.state)
    })
}
```

| 更多详情 <http://blog.poeties.top/2018/12/20/react-setState>

#29.3 性能优化

- 在 `shouldComponentUpdate` 函数中我们可以通过返回布尔值来决定当前组件是否需要更新。这层代码逻辑可以是简单地浅比较一下当前 `state` 和之前的 `state` 是否相同，也可以是判断某个值更新了才触发组件更新。一般来说不推荐完整地对比当前 `state` 和之前的 `state` 是否相同，因为组件更新触发可能会很频繁，这样的完整对比性能开销会有点大，可能会造成得不偿失的情况。
- 当然如果真的想完整对比当前 `state` 和之前的 `state` 是否相同，并且不影响性能也是行得通的，可以通过 `immutable` 或者 `immer` 这些库来生成不可变对象。这类库对于操作大规模的数据来说会提升不错的性能，并且一旦改变数据就会生成一个新的对象，对比前后 `state` 是否一致也就方便多了，同时也很推荐阅读下 `immer` 的源码实现
- 另外如果只是单纯的浅比较一下，可以直接使用 `PureComponent`，底层就是实现了浅比较 `state`

```
class Test extends React.PureComponent {
    render() {
        return (
            <div>
                PureComponent
            </div>
        )
    }
}
```

| 这时候你可能会考虑到函数组件就不能使用这种方式了，如果你使用 `16.6.0` 之后的版本的话，可以使用 `React.memo` 来实现相同的功能

```
const Test = React.memo(() => (
    <div>
        PureComponent
    </div>
))
```

| 通过这种方式我们就可以既实现了 `shouldComponentUpdate` 的浅比较，又能够使用函数组件

#29.4 通信

1. 父子通信

- 父组件通过 `props` 传递数据给子组件，子组件通过调用父组件传来的函数传递数据给父组件，这两种方式是最常用的父子通信实现办法。
- 这种父子通信方式也就是典型的单向数据流，父组件通过 `props` 传递数据，子组件不能直接修改 `props`，而是必须通过调用父组件函数的方式告知父组件修改数据。

2. 兄弟组件通信

对于这种情况可以通过共同的父组件来管理状态和事件函数。比如说其中一个兄弟组件调用父组件传递过来的事件函数修改父组件中的状态，然后父组件将状态传递给另一个兄弟组件

3. 跨多层次组件通信

如果你使用 16.3 以上版本的话，对于这种情况可以使用 Context API

```
// 创建 Context，可以在开始就传入值
const StateContext = React.createContext()
class Parent extends React.Component {
  render () {
    return (
      // value 就是传入 Context 中的值
      <StateContext.Provider value='yck'>
        <Child />
      </StateContext.Provider>
    )
  }
}
class Child extends React.Component {
  render () {
    return (
      <ThemeContext.Consumer>
        // 取出值
        {context => (
          name is { context }
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

4. 任意组件

这种方式可以通过 Redux 或者 Event Bus 解决，另外如果你不怕麻烦的话，可以使用这种方式解决上述所有的通信情况

#29.5 HOC 是什么？相比 mixins 有什么优点？

很多人看到高阶组件 (HOC) 这个概念就被吓到了，认为这东西很难，其实这东西概念真的很简单，我们先来看一个例子。

```
function add(a, b) {
  return a + b
}
```

现在如果我想给这个 add 函数添加一个输出结果的功能，那么你可能会考虑我直接使用 console.log 不就实现了么。说的没错，但是如果我们要做的更加优雅并且容易复用和扩展，我们可以这样去做

```

function withLog (fn) {
  function wrapper(a, b) {
    const result = fn(a, b)
    console.log(result)
    return result
  }
  return wrapper
}
const withLogAdd = withLog(add)
withLogAdd(1, 2)

```

- 其实这个做法在函数式编程里称之为高阶函数，大家都知道 React 的思想中是存在函数式编程的，高阶组件和高阶函数就是同一个东西。我们实现一个函数，传入一个组件，然后在函数内部再实现一个函数去扩展传入的组件，最后返回一个新的组件，这就是高阶组件的概念，作用就是为了更好的复用代码。
- 其实 HOC 和 vue 中的 mixins 作用是一致的，并且在早期 React 也是使用 mixins 的方式。但是在使用 class 的方式创建组件以后，mixins 的方式就不能使用了，并且其实 mixins 也是存在一些问题的，比如
 1. 隐含了一些依赖，比如我在组件中写了某个 state 并且在 mixin 中使用了，这就存在了一个依赖关系。万一下次别人要移除它，就得去 mixin 中查找依赖
 2. 多个 mixin 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。
 3. 雪球效应，虽然我一个组件还是使用着同一个 mixin，但是一个 mixin 会被多个组件使用，可能会存在需求使得 mixin 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本

HOC 解决了这些问题，并且它们达成的效果也是一致的，同时也更加的政治正确（毕竟更加函数式了）

#29.6 事件机制

React 其实自己实现了一套事件机制，首先我们考虑一下以下代码：

```

const Test = ({ list, handleClick }) => ({
  list.map((item, index) => (
    <span onClick={handleClick} key={index}>{index}</span>
  ))
})

```

- 以上类似代码想必大家经常会写到，但是你是否考虑过点击事件是否绑定在了每一个标签上？事实当然不是，JSX 上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 document 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。
- 另外冒泡到 document 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 event.stopPropagation 是无效的，而应该调用 event.preventDefault

那么实现合成事件的目的是什么呢？总的来说在我看来好处有两点，分别是：

1. 合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力
2. 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专

门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

#30 监控

前端监控一般分为三种，分别为页面埋点、性能监控以及异常监控。

这一章节我们将来学习这些监控相关的内容，但是基本不会涉及到代码，只是让大家了解下前端监控该用什么方式实现。毕竟大部分公司都只是使用到了第三方的监控工具，而不是选择自己造轮子。

#30.1 页面埋点

页面埋点应该是大家最常写的监控了，一般起码会监控以下几个数据：

- `PV / UV`
- 停留时长
- 流量来源
- 用户交互

对于这几类统计，一般的实现思路大致可以分为两种，分别为手写埋点和无埋点的方式。

相信第一种方式也是大家最常用的方式，可以自主选择需要监控的数据然后在相应的地方写入代码。这种方式的灵活性很大，但是唯一的缺点就是工作量较大，每个需要监控的地方都得插入代码。

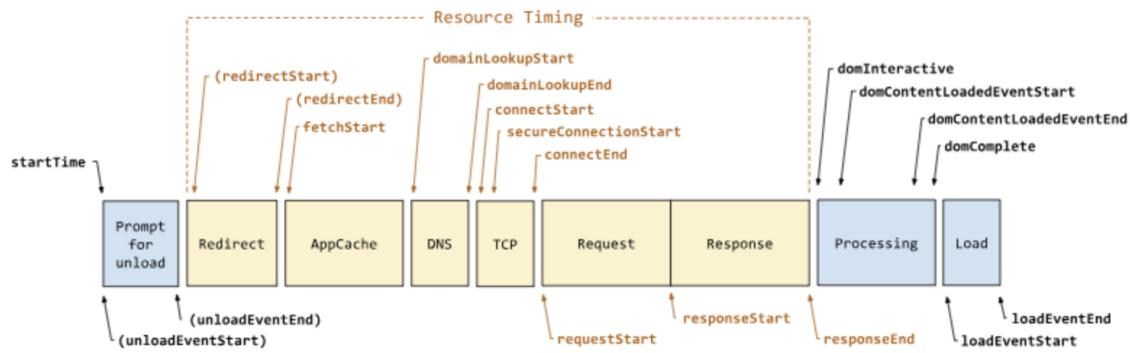
另一种无埋点的方式基本不需要开发者手写埋点了，而是统计所有的事件并且定时上报。这种方式虽然没有前一种方式繁琐了，但是因为统计的是所有事件，所以还需要后期过滤出需要的数据。

#30.2 性能监控

- 性能监控可以很好的帮助开发者了解在各种真实环境下，页面的性能情况是如何的。
- 对于性能监控来说，我们可以直接使用浏览器自带的 `Performance API` 来实现这个功能。
- 对于性能监控来说，其实我们只需要调用 `performance.getEntriesByType('navigation')` 这行代码就行了。对，你没看错，一行代码我们就可以获得页面中各种详细的性能相关信息

```
> performance.getEntriesByType('navigation')
<- ▶ [PerformanceNavigationTiming] ⓘ
  ▼ 0: PerformanceNavigationTiming
    connectEnd: 14.159999991534278
    connectStart: 14.159999991534278
    decodedBodySize: 17583
    domComplete: 3008.5649999964517
    domContentLoadedEventEnd: 1379.350000002887
    domContentLoadedEventStart: 1035.8849999902304
    domInteractive: 1035.8499999856576
    domainLookupEnd: 14.159999991534278
    domainLookupStart: 14.159999991534278
    duration: 3010.3399999788962
    encodedBodySize: 11548
    entryType: "navigation"
    fetchStart: 14.159999991534278
    initiatorType: "navigation"
    loadEventEnd: 3010.3399999788962
    loadEventStart: 3008.5850000032224
    name: "https://juejin.im/"
    nextHopProtocol: ""
    redirectCount: 0
    redirectEnd: 0
    redirectStart: 0
    requestStart: 25.439999997615814
    responseEnd: 102.87499998230487
    responseStart: 54.64499999652617
    secureConnectionStart: 0
    ▶ serverTiming: []
    startTime: 0
    transferSize: 0
    type: "navigate"
    unloadEventEnd: 0
    unloadEventStart: 0
    workerStart: 0
```

我们可以发现这行代码返回了一个数组，内部包含了相当多的信息，从数据开始在网络中传输到页面加载完成都提供了相应的数据



#30.3 异常监控

- 对于异常监控来说，以下两种监控是必不可少的，分别是代码报错以及接口异常上报。
- 对于代码运行错误，通常的办法是使用 `window.onerror` 拦截报错。该方法能拦截到大部分的详细报错信息，但是也有例外

- 对于跨域的代码运行错误会显示 `script error`。对于这种情况我们需要给 `script` 标签添加 `crossorigin` 属性
- 对于某些浏览器可能不会显示调用栈信息，这种情况可以通过 `arguments.callee.caller` 来做栈递归
 - 对于异步代码来说，可以使用 `catch` 的方式捕获错误。比如 `Promise` 可以直接使用 `catch` 函数，`async await` 可以使用 `try catch`
 - 但是要注意线上运行的代码都是压缩过的，需要在打包时生成 `sourceMap` 文件便于 `debug`
 - 对于捕获的错误需要上传给服务器，通常可以通过 `img` 标签的 `src` 发起一个请求。
 - 另外接口异常就相对来说简单了，可以列举出出错的状态码。一旦出现此类的状态码就可以立即上报出错。接口异常上报可以让开发人员迅速知道有哪些接口出现了大面积的报错，以便迅速修复问题。

#31 TCP/UDP

#31.1 UDP

网络协议是每个前端工程师都必须要掌握的知识，我们将先来学习传输层中的两个协议：`UDP` 以及 `TCP`。对于大部分工程师来说最常用的协议也就是这两个了，并且面试中经常会提问的也是关于这两个协议的区别

常考面试题：`UDP` 与 `TCP` 的区别是什么？

首先 `UDP` 协议是面向无连接的，也就是说不需要在正式传递数据之前先连接起双方。然后 `UDP` 协议只是数据报文的搬运工，不保证有序且不丢失的传递到对端，并且 `UDP` 协议也没有任何控制流量的算法，总的来说 `UDP` 相较于 `TCP` 更加的轻便

1. 面向无连接

- 首先 `UDP` 是不需要和 `TCP` 一样在发送数据前进行三次握手建立连接的，想发数据就可以开始发送了。
- 并且也只是数据报文的搬运工，不会对数据报文进行任何拆分和拼接操作。

具体来说就是：

- 在发送端，应用层将数据传递给传输层的 `UDP` 协议，`UDP` 只会给数据增加一个 `UDP` 头标识下是 `UDP` 协议，然后就传递给网络层了 在接收端，网络层将数据传递给传输层，`UDP` 只去除 `IP` 报文

头就传递给应用层，不会任何拼接操作

2. 不可靠性

- 首先不可靠性体现在无连接上，通信都不需要建立连接，想发就发，这样的情况肯定不可靠。
- 并且收到什么数据就传递什么数据，并且也不会备份数据，发送数据也不会关心对方是否已经正确接收到数据了。
- 再者网络环境时好时坏，但是 **UDP** 因为没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 **UDP** 而不是 **TCP**

3. 高效

- 虽然 **UDP** 协议不是那么的可靠，但是正是因为它不是那么的可靠，所以也就没有 **TCP** 那么复杂了，需要保证数据不丢失且有序到达。
- 因此 **UDP** 的头部开销小，只有八字节，相比 **TCP** 的至少二十字节要少得多，在传输数据报文时是很高效的。

UDP 头部包含了以下几个数据

- 两个十六位的端口号，分别为源端口（可选字段）和目标端口 整个数据报文的长度
- 整个数据报文的检验和（**IPv4** 可选 字段），该字段用于发现头部信息和数据中的错误

4. 传输方式

UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式，也就是说 **UDP** 提供了单播，多播，广播的功能。

5. 适合使用的场景

UDP 虽然对比 **TCP** 有很多缺点，但是正是因为这些缺点造就了它高效的特性，在很多实时性要求高的地方都可以看到 **UDP** 的身影。

5.1 直播

- 想必大家都看过直播吧，大家可以考虑下如果直播使用了基于 **TCP** 的协议会发生什么事情？
 - TCP** 会严格控制传输的正确性，一旦有某一个数据对端没有收到，就会停下来直到对端收到这个数据。这种问题在网络条件不错的情况下可能并不会发生什么事情，但是在网络情况差的时候就会变成画面卡住，然后再继续播放下一帧的情况。
 - 但是对于直播来说，用户肯定关注的是最新的画面，而不是因为网络条件差而丢失的老旧画面，所以 **TCP** 在这种情况下无用武之地，只会降低用户体验。

5.2 王者荣耀

- 首先对于王者荣耀来说，用户体量是相当大的，如果使用 **TCP** 连接的话，就可能会出现服务器不够用的情况，因为每台服务器可供支撑的 **TCP** 连接数量是有限制的。
- 再者，因为 **TCP** 会严格控制传输的正确性，如果因为用户网络条件不好就造成页面卡顿然后再传输旧的游戏画面是肯定不能接受的，毕竟对于这类实时性要求很高的游戏来说，最新的游戏画面才是最需要的，而不是老旧的画面，否则角色都不知道死多少次了。

#31.2 TCP

常考面试题：**UDP** 与 **TCP** 的区别是什么？

TCP 基本是和 **UDP** 反着来，建立连接断开连接都需要先需要进行握手。在传输数据的过程中，通过各种算法保证数据的可靠性，当然带来的问题就是相比 **UDP** 来说不那么的高效

1. 头部

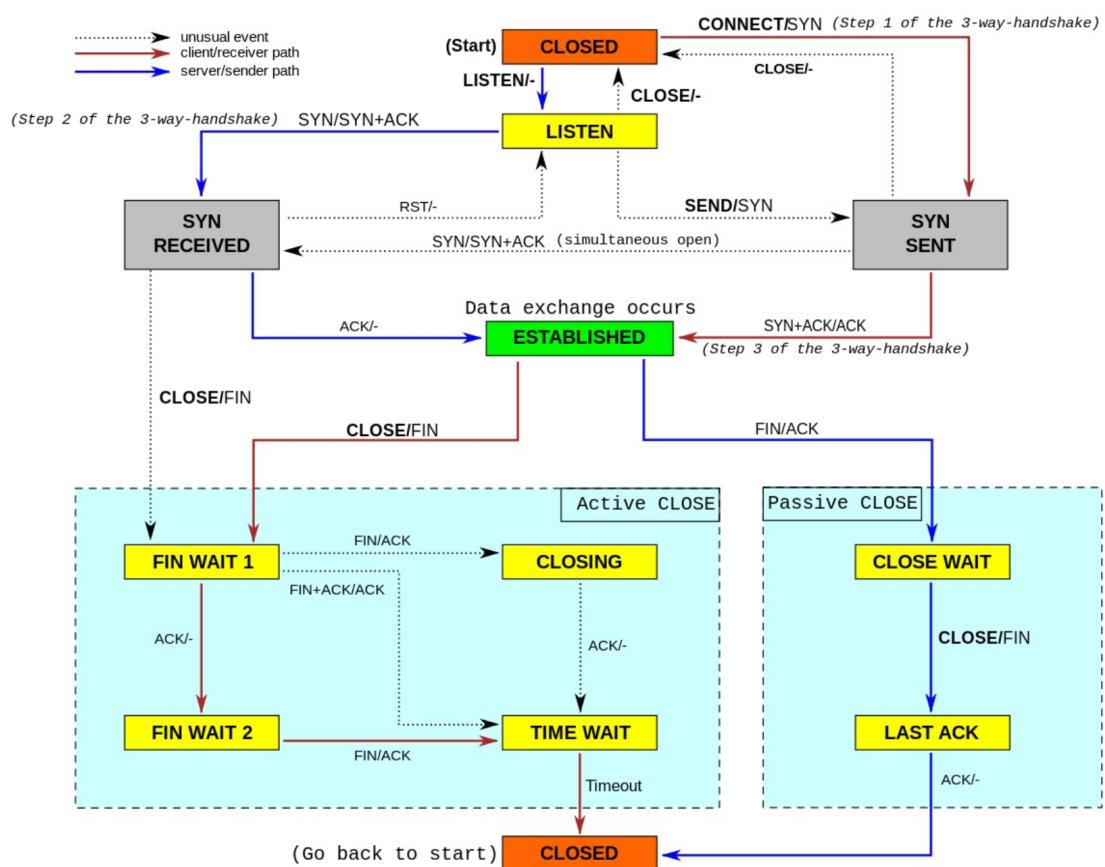
TCP 头部比 UDP 头部复杂的多

对于 TCP 头部来说，以下几个字段是很重要的

- Sequence number，这个序号保证了 TCP 传输的报文都是有序的，对端可以通过序号顺序的拼接报文
- Acknowledgement Number，这个序号表示数据接收端期望接收的下一个字节的编号是多少，同时也表示上一个序号的数据已经收到
- window size，窗口大小，表示还能接收多少字节的数据，用于流量控制
- 标识符
 - URG=1：该字段为一表示本数据报的数据部分包含紧急信息，是一个高优先级数据报文，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。
 - ACK=1：该字段为一表示确认号字段有效。此外，TCP 还规定在连接建立后传送的所有报文段都必须把 ACK 置为一。
 - PSH=1：该字段为一表示接收端应该立即将数据 push 给应用层，而不是等到缓冲区满后再提交。
 - RST=1：该字段为一表示当前 TCP 连接出现严重问题，可能需要重新建立 TCP 连接，也可以用于拒绝非法的报文段和拒绝连接请求。
 - SYN=1：当 SYN=1，ACK=0 时，表示当前报文段是一个连接请求报文。当 SYN=1，ACK=1 时，表示当前报文段是一个同意建立连接的应答报文。
 - FIN=1：该字段为一表示此报文段是一个释放连接的请求报文。

2. 状态机

TCP 的状态机是很复杂的，并且与建立断开连接时的握手息息相关，接下来就来详细描述下两种握手



在这之前需要了解一个重要的性能指标 **RTT**。该指标表示发送端发送数据到接收到对端数据所需的往返时间

2.1. 建立连接三次握手

- 首先假设主动发起请求的一端称为客户端，被动连接的一端称为服务端。不管是客户端还是服务端，**TCP** 连接建立完后都能发送和接收数据，所以 **TCP** 是一个全双工的协议。
- 起初，两端都为 **CLOSED** 状态。在通信开始前，双方都会创建 **TCB**。服务器创建完 **TCB** 后便进入 **LISTEN** 状态，此时开始等待客户端发送数据

第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 **SYN-SENT** 状态

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 **SYN-RECEIVED** 状态

第三次握手

- 当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 **ESTABLISHED** 状态，服务端收到这个应答后也进入 **ESTABLISHED** 状态，此时连接建立成功。
- PS：第三次握手中可以包含数据，通过快速打开（**TFO**）技术就可以实现这一功能。其实只要涉及到握手的协议，都可以使用类似 **TFO** 的方式，客户端和服务端存储相同的 **cookie**，下次握手时发出 **cookie** 达到减少 **RTT** 的目的。

常考面试题：为什么 **TCP** 建立连接需要三次握手，明明两次就可以建立起连接

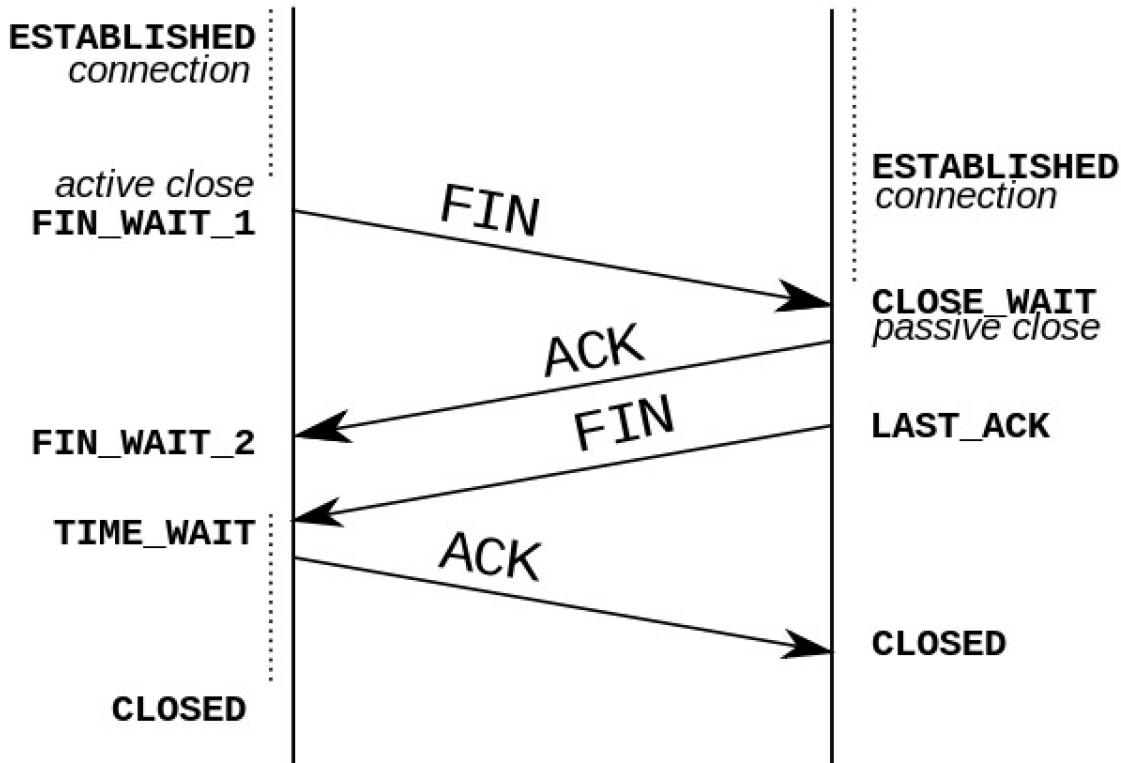
- 因为这是为了防止出现失效的连接请求报文段被服务端接收的情况，从而产生错误。
- 可以想象如下场景。客户端发送了一个连接请求 **A**，但是因为网络原因造成了超时，这时 **TCP** 会启动超时重传的机制再次发送一个连接请求 **B**。此时请求顺利到达服务端，服务端应答完就建立了请求，然后接收数据后释放了连接。

假设这时候连接请求 **A** 在两端关闭后终于抵达了服务端，那么此时服务端会认为客户端又需要建立 **TCP** 连接，从而应答了该请求并进入 **ESTABLISHED** 状态。但是客户端其实是 **CLOSED** 的状态，那么就会导致服务端一直等待，造成资源的浪费。

PS：在建立连接中，任意一端掉线，**TCP** 都会重发 **SYN** 包，一般会重试五次，在建立连接中可能会遇到 **SYN Flood** 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求

2.2. 断开链接四次握手

Initiator Receiver



TCP`是全双工的，在断开连接时两端都需要发送`FIN`和`ACK`

第一次握手

若客户端 A 认为数据发送完成，则它需要向服务端 B 发送连接释放请求。

第二次握手

B`收到连接释放请求后，会告诉应用层要释放`TCP`链接。然后会发送`ACK`包，并进入`CLOSE_WAIT`状态，此时表明`A`到`B`的连接已经释放，不再接收`A`发的数据了。但是因为`TCP`连接是双向的，所以`B`仍旧可以发送数据给`A`

3. ARQ 协议

ARQ 协议也就是超时重传机制。通过确认和超时机制保证了数据的正确送达，ARQ 协议包含停止等待 ARQ 和连续 ARQ 两种协议。

停止等待 ARQ

正常传输过程

只要 A 向 B 发送一段报文，都要停止发送并启动一个定时器，等待对端回应，在定时器时间内接收到对端应答就取消定时器并发送下一段报文。

报文丢失或出错

- 在报文传输的过程中可能会出现丢包。这时候超过定时器设定的时间就会再次发送丢失的数据直到对端响应，所以需要每次都备份发送的数据。
- 即使报文正常的传输到对端，也可能出现在传输过程中报文出错的问题。这时候对端会抛弃该报文并等待 A 端重传。
- PS：一般定时器设定的时间都会大于一个 RTT 的平均时间。

第三次握手

- B 如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 LAST-ACK 状态。
- PS：通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。

第四次握手

A 收到释放请求后，向 B 发送确认应答，此时 A 进入 TIME-WAIT 状态。该状态会持续 2MSL（最大段生存期，指报文在网络中生存的时间，超时会被抛弃）时间，若该时间段内没有 B 的重发请求的话，就进入 CLOSED 状态。当 B 收到确认应答后，也便进入 CLOSED 状态。

- 为什么 A 要进入 TIME-WAIT 状态，等待 2MSL 时间后才进入 CLOSED 状态？
- 为了保证 B 能收到 A 的确认应答。若 A 发完确认应答后直接进入 CLOSED 状态，如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭。

ACK 超时或丢失

- 对端传输的应答也可能出现丢失或超时的情况。那么超过定时器时间 A 端照样会重传报文。这时候 B 端收到相同序号的报文会丢弃该报文并重传应答，直到 A 端发送下一个序号的报文。
- 在超时的情况下也可能出现应答很迟到达，这时 A 端会判断该序号是否已经接收过，如果接收过只需要丢弃应答即可。
- 从上面的描述中大家肯定可以发现这肯定不是一个高效的方式。假设在良好的网络环境中，每次发送数据都需要等待片刻肯定是不能接受的。那么既然我们不能接受这个不那么高效的协议，就来继续学习相对高效的协议吧。

连续 ARQ

在连续 ARQ 中，发送端拥有一个发送窗口，可以在没有收到应答的情况下持续发送窗口内的数据，这样相比停止等待 ARQ 协议来说减少了等待时间，提高了效率。

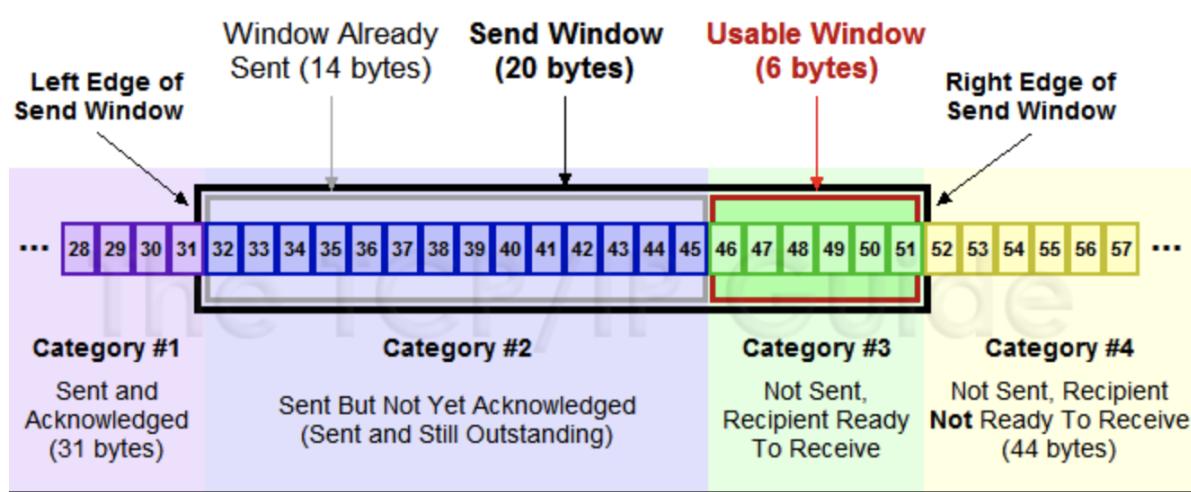
累计确认

连续 ARQ 中，接收端会持续不断收到报文。如果和停止等待 ARQ 中接收一个报文就发送一个应答一样，就太浪费资源了。通过累计确认，可以在收到多个报文以后统一回复一个应答报文。报文中的 ACK 标志位可以用来告诉发送端这个序号之前的数据已经全部接收到了，下次请发送这个序号后的数据。

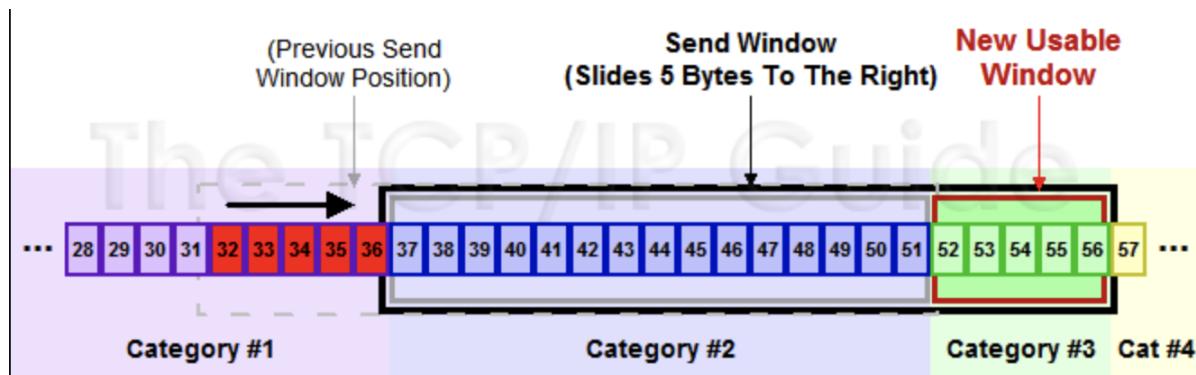
但是累计确认也有一个弊端。在连续接收报文时，可能会遇到接收到序号 5 的报文后，并未接收到序号 6 的报文，然而序号 7 以后的报文已经接收。遇到这种情况时，ACK 只能回复 6，这样就会造成发送端重复发送数据的情况。

4. 滑动窗口

- 上面小节中讲到了发送窗口。在 TCP 中，两端其实都维护着窗口：分别为发送端窗口和接收端窗口。
- 发送端窗口包含已发送但未收到应答的数据和可以发送但是未发送的数据。



- 发送端窗口是由接收窗口剩余大小决定的。接收方会把当前接收窗口的剩余大小写入应答报文，发送端收到应答后根据该值和当前网络拥塞情况设置发送窗口的大小，所以发送窗口的大小是不断变化的。
- 当发送端接收到应答报文后，会随之将窗口进行滑动



滑动窗口是一个很重要的概念，它帮助 TCP 实现了流量控制的功能。接收方通过报文告知发送方还可以发送多少数据，从而保证接收方能够来得及接收数据，防止出现接收方带宽已满，但是发送方还一直发送数据的情况

Zero 窗口

在发送报文的过程中，可能会遇到对端出现零窗口的情况。在该情况下，发送端会停止发送数据，并启动 `persistent timer`。该定时器会定时发送请求给对端，让对端告知窗口大小。在重试次数超过一定次数后，可能会中断 `TCP` 链接

5. 拥塞处理

- 拥塞处理和流量控制不同，后者是作用于接收方，保证接收方来得及接受数据。而前者是作用于网络，防止过多的数据拥塞网络，避免出现网络负载过大的情况。
- 拥塞处理包括了四个算法，分别为：慢开始，拥塞避免，快速重传，快速恢复

慢开始算法

慢开始算法，顾名思义，就是在传输开始时将发送窗口慢慢指数级扩大，从而避免一开始就传输大量数据导致网络拥塞。想必大家都下载过资源，每当我们开始下载的时候都会发现下载速度是慢慢提升的，而不是一蹴而就直接拉满带宽

慢开始算法步骤具体如下

- 连接初始设置拥塞窗口（Congestion Window）为 `1 MSS`（一个分段的最大数据量）
- 每过一个 `RTT` 就将窗口大小乘二
- 指数级增长肯定不能没有限制的，所以有一个阈值限制，当窗口大小大于阈值时就会启动拥塞避免算法。

拥塞避免算法

- 拥塞避免算法相比简单点，每过一个 RTT 窗口大小只加一，这样能够避免指数级增长导致网络拥塞，慢慢将大小调整到最佳值。
- 在传输过程中可能定时器超时的情况，这时候 TCP 会认为网络拥塞了，会马上进行以下步骤：
 1. 将阈值设为当前拥塞窗口的一半
 2. 将拥塞窗口设为 1 MSS
 3. 启动拥塞避免算法

快速重传

快速重传一般和快恢复一起出现。一旦接收端收到的报文出现失序的情况，接收端只会回复最后一个顺序正确的报文序号。如果发送端收到三个重复的 ACK，无需等待定时器超时而是直接启动快速重传算法。具体算法分为两种：

TCP Tahoe 实现如下

- 将阈值设为当前拥塞窗口的一半
- 将拥塞窗口设为 1 MSS
- 重新开始慢开始算法
- TCP Reno 实现如下

拥塞窗口减半

- 将阈值设为当前拥塞窗口
- 进入快恢复阶段（重发对端需要的包，一旦收到一个新的 ACK 答复就退出该阶段），这种方式在丢失多个包的情况下就不那么好了
- 使用拥塞避免算法

TCP New Reno 改进后的快恢复

- TCP New Reno 算法改进了之前 TCP Reno 算法的缺陷。在之前，快恢复中只要收到一个新的 ACK 包，就会退出快恢复。
- 在 TCP New Reno 中，TCP 发送方先记下三个重复 ACK 的分段的最大序号。

假如我有一个分段数据是 1 ~ 10 这十个序号的报文，其中丢失了序号为 3 和 7 的报文，那么该分段的最大序号就是 10。发送端只会收到 ACK 序号为 3 的应答。这时候重发序号为 3 的报文，接收方顺利接收的话就会发送 ACK 序号为 7 的应答。这时候 TCP 知道对端是有多个包未收到，会继续发送序号为 7 的报文，接收方顺利接收并会发送 ACK 序号为 11 的应答，这时发送端认为这个分段接收端已经顺利接收，接下来会退出快恢复阶段。

#32 HTTP/TLS

#32.1 HTTP 请求中的内容

HTTP 请求由三部分构成，分别为：

- 请求行
 - 首部
 - 实体
- 请求行大概长这样 GET /images/logo.gif HTTP/1.1，基本由请求方法、URL、协议版本组成，这其中值得一提的就是请求方法了。
 - 请求方法分为很多种，最常用的也就是 Get 和 Post 了。虽然请求方法有很多，但是更多的是传达一个语义，而不是说 Post 能做的事情 Get 就不能做了。如果你愿意，都使用

`Get` 请求或者 `Post` 请求都是可以的

常考面试题：Post 和 Get 的区别？

- 首先先引入副作用和幂等的概念。
 - 副作用指对服务器上的资源做改变，搜索是无副作用的，注册是副作用的。
 - 幂等指发送 M 和 N 次请求（两者不相同且都大于 1），服务器上资源的状态一致，比如注册 10 个和 11 个帐号是不幂等的，对文章进行更改 10 次和 11 次是幂等的。因为前者是多了一个账号（资源），后者只是更新同一个资源。
 - 在规范的应用场景上说，`Get` 多用于无副作用，幂等的场景，例如搜索关键字。`Post` 多用于副作用，不幂等的场景，例如注册。
- `Get` 请求能缓存，`Post` 不能
 - `Post` 相对 `Get` 安全一点点，因为 `Get` 请求都包含在 `URL` 里（当然你想写到 `body` 里也是可以的），且会被浏览器保存历史纪录。`Post` 不会，但是在抓包的情况下都是一样的。
 - `URL` 有长度限制，会影响 `Get` 请求，但是这个长度限制是浏览器规定的，不是 `RFC` 规定的
 - `Post` 支持更多的编码类型且不对数据类型限制

1. 首部

首部分为请求首部和响应首部，并且部分首部两种通用，接下来我们就来学习一部分的常用首部。

1.1 通用首部

| 通用字段 | 作用 |
|--------------------------------|---|
| <code>Cache-Control</code> | 控制缓存的行为 |
| <code>Connection</code> | 浏览器想要优先使用的连接类型，比如 <code>keep-alive</code> |
| <code>Date</code> | 创建报文时间 |
| <code>Pragma</code> | 报文指令 |
| <code>Via</code> | 代理服务器相关信息 |
| <code>Transfer-Encoding</code> | 传输编码方式 |
| <code>Upgrade</code> | 要求客户端升级协议 |
| <code>Warning</code> | 在内容中可能存在错误 |

1.2 请求首部

| 请求首部 | 作用 |
|---------------------|----------------------|
| Accept | 能正确接收的媒体类型 |
| Accept-Charset | 能正确接收的字符集 |
| Accept-Encoding | 能正确接收的编码格式列表 |
| Accept-Language | 能正确接收的语言列表 |
| Expect | 期待服务端的指定行为 |
| From | 请求方邮箱地址 |
| Host | 服务器的域名 |
| If-Match | 两端资源标记比较 |
| If-Modified-Since | 本地资源未修改返回 304 (比较时间) |
| If-None-Match | 本地资源未修改返回 304 (比较标记) |
| User-Agent | 客户端信息 |
| Max-Forwards | 限制可被代理及网关转发的次数 |
| Proxy-Authorization | 向代理服务器发送验证信息 |
| Range | 请求某个内容的一部分 |
| Referer | 表示浏览器所访问的前一个页面 |
| TE | 传输编码方式 |

1.3 响应首部

| 响应首部 | 作用 |
|--------------------|---------------|
| Accept-Ranges | 是否支持某些种类的范围 |
| Age | 资源在代理缓存中存在的时间 |
| ETag | 资源标识 |
| Location | 客户端重定向到某个 URL |
| Proxy-Authenticate | 向代理服务器发送验证信息 |
| Server | 服务器名字 |
| WWW-Authenticate | 获取资源需要的验证信息 |

1.4 实体首部

| 实体首部 | 作用 |
|------------------|------------------------|
| Allow | 资源的正确请求方式 |
| Content-Encoding | 内容的编码格式 |
| Content-Language | 内容使用的语言 |
| Content-Length | request body 长度 |
| Content-Location | 返回数据的备用地址 |
| Content-MD5 | Base64 加密格式的内容 MD5 检验值 |
| Content-Range | 内容的位置范围 |
| Content-Type | 内容的媒体类型 |
| Expires | 内容的过期时间 |
| Last_modified | 内容的最后修改时间 |

2. 常见状态码

状态码表示了响应的一个状态，可以让我们清晰的了解到这一次请求是成功还是失败，如果失败的话，是什么原因导致的，当然状态码也是用于传达语义的。如果胡乱使用状态码，那么它存在的意义就没有了

2XX 成功

- 200 OK，表示从客户端发来的请求在服务器端被正确处理
- 204 No content，表示请求成功，但响应报文不含实体的主体部分
- 205 Reset Content，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容
- 206 Partial Content，进行范围请求

3XX 重定向

- 301 moved permanently，永久性重定向，表示资源已被分配了新的 URL
- 302 found，临时性重定向，表示资源临时被分配了新的 URL
- 303 see other，表示资源存在着另一个 URL，应使用 GET 方法获取资源
- 304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况
- 307 temporary redirect，临时重定向，和 302 含义类似，但是期望客户端保持请求方法不变向新的地址发出请求

4XX 客户端错误

- 400 bad request，请求报文存在语法错误
- 401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
- 403 forbidden，表示对请求资源的访问被服务器拒绝
- 404 not found，表示在服务器上没有找到请求的资源

5XX 服务器错误

- 500 internal sever error，表示服务器端在执行请求时发生了错误
- 501 Not Implemented，表示服务器不支持当前请求所需要的某个功能
- 503 service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求

#32.2 TLS

- HTTPS 还是通过了 HTTP 来传输信息，但是信息通过 TLS 协议进行了加密。
- TLS 协议位于传输层之上，应用层之下。首次进行 TLS 协议传输需要两个 RTT，接下来可以通过 Session Resumption 减少到一个 RTT。
- 在 TLS 中使用了两种加密技术，分别为：对称加密和非对称加密。

对称加密

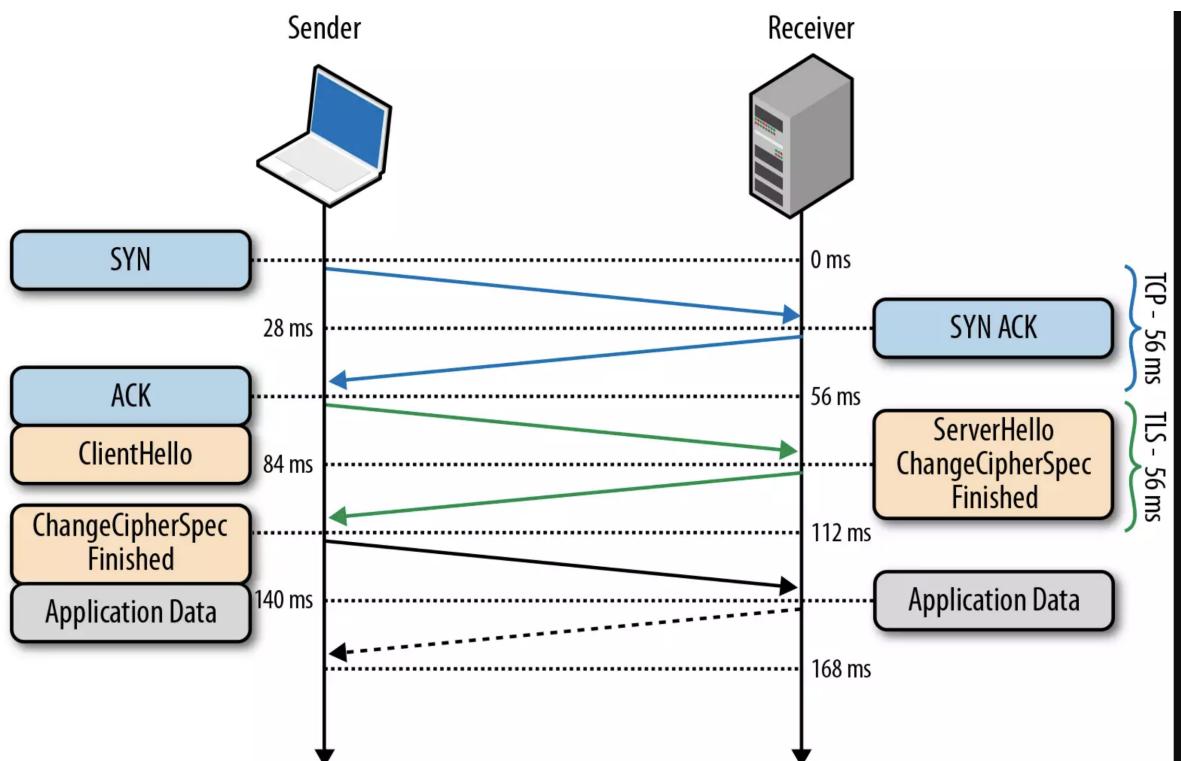
- 对称加密就是两边拥有相同的密钥，两边都知道如何将密文加密解密。
- 这种加密方式固然很好，但是问题就在于如何让双方知道密钥。因为传输数据都是走的网络，如果将密钥通过网络的方式传递的话，一旦密钥被截获就没有加密的意义的。

非对称加密

- 有公钥私钥之分，公钥所有人都可以知道，可以将数据用公钥加密，但是将数据解密必须使用私钥解密，私钥只有分发公钥的一方才知道。
- 这种加密方式就可以完美解决对称加密存在的问题。假设现在两端需要使用对称加密，那么在这之前，可以先使用非对称加密交换密钥。

简单流程如下：首先服务端将公钥公布出去，那么客户端也就知道公钥了。接下来客户端创建一个密钥，然后通过公钥加密并发送给服务端，服务端接收到密文以后通过私钥解密出正确的密钥，这时候两端就都知道密钥是什么了。

TLS 握手过程如下图：



- 客户端发送一个随机值以及需要的协议和加密方式。
- 服务端收到客户端的随机值，自己也产生一个随机值，并根据客户端需求的协议和加密方式来使用对应的方式，并且发送自己的证书（如果需要验证客户端证书需要说明）
- 客户端收到服务端的证书并验证是否有效，验证通过会再生成一个随机值，通过服务端证书的公钥去加密这个随机值并发送给服务端，如果服务端需要验证客户端证书的话会附带证书
- 服务端收到加密过的随机值并使用私钥解密获得第三个随机值，这时候两端都拥有了三个随机值，可以通过这三个随机值按照之前约定的加密方式生成密钥，接下来的通信就可以通过该密钥来加密解密
- 通过以上步骤可知，在 TLS 握手阶段，两端使用非对称加密的方式来通信，但是因为非对称加密损耗的性能比对称加密大，所以在正式传输数据时，两端使用对称加密的方式通信。

PS：以上说明的都是 `TLS 1.2` 协议的握手情况，在 1.3 协议中，首次建立连接只需要一个 RTT，后面恢复连接不需要 RTT 了

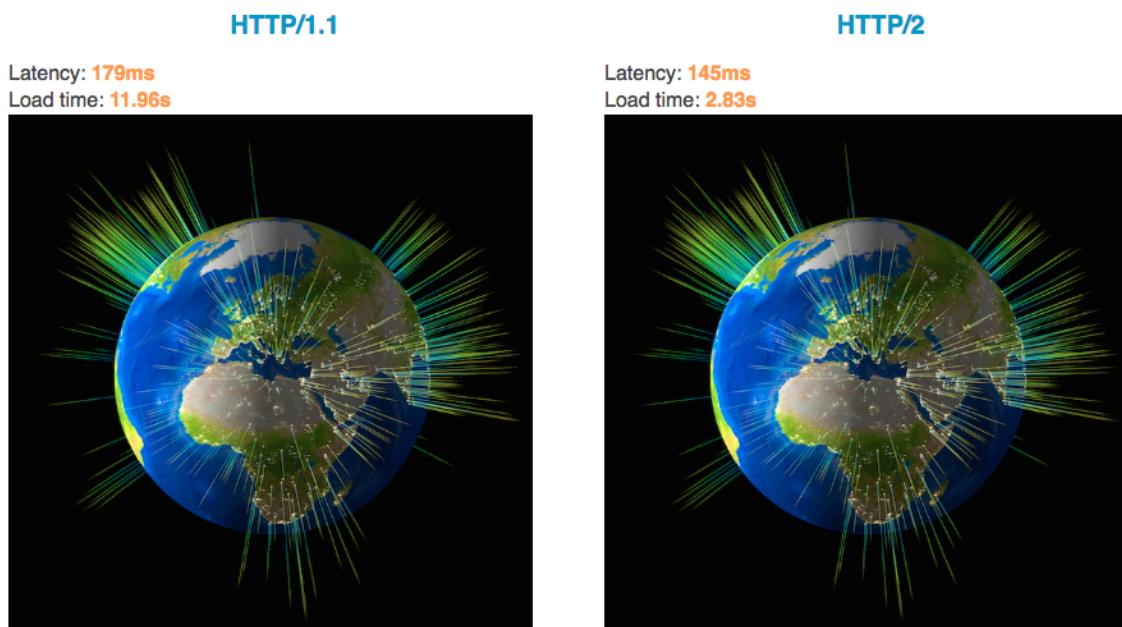
#33 HTTP2.0

- `HTTP/2` 很好的解决了当下最常用的 `HTTP/1` 所存在的一些性能问题，只需要升级到该协议就可以减少很多之前需要做的性能优化工作，当然兼容问题以及如何优雅降级应该是国内还不普遍使用的原因之一。
- 虽然 `HTTP/2` 已经解决了很多问题，但是并不代表它已经是完美的了，`HTTP/3` 就是为了解决 `HTTP/2` 所存在的一些问题而被推出来的。

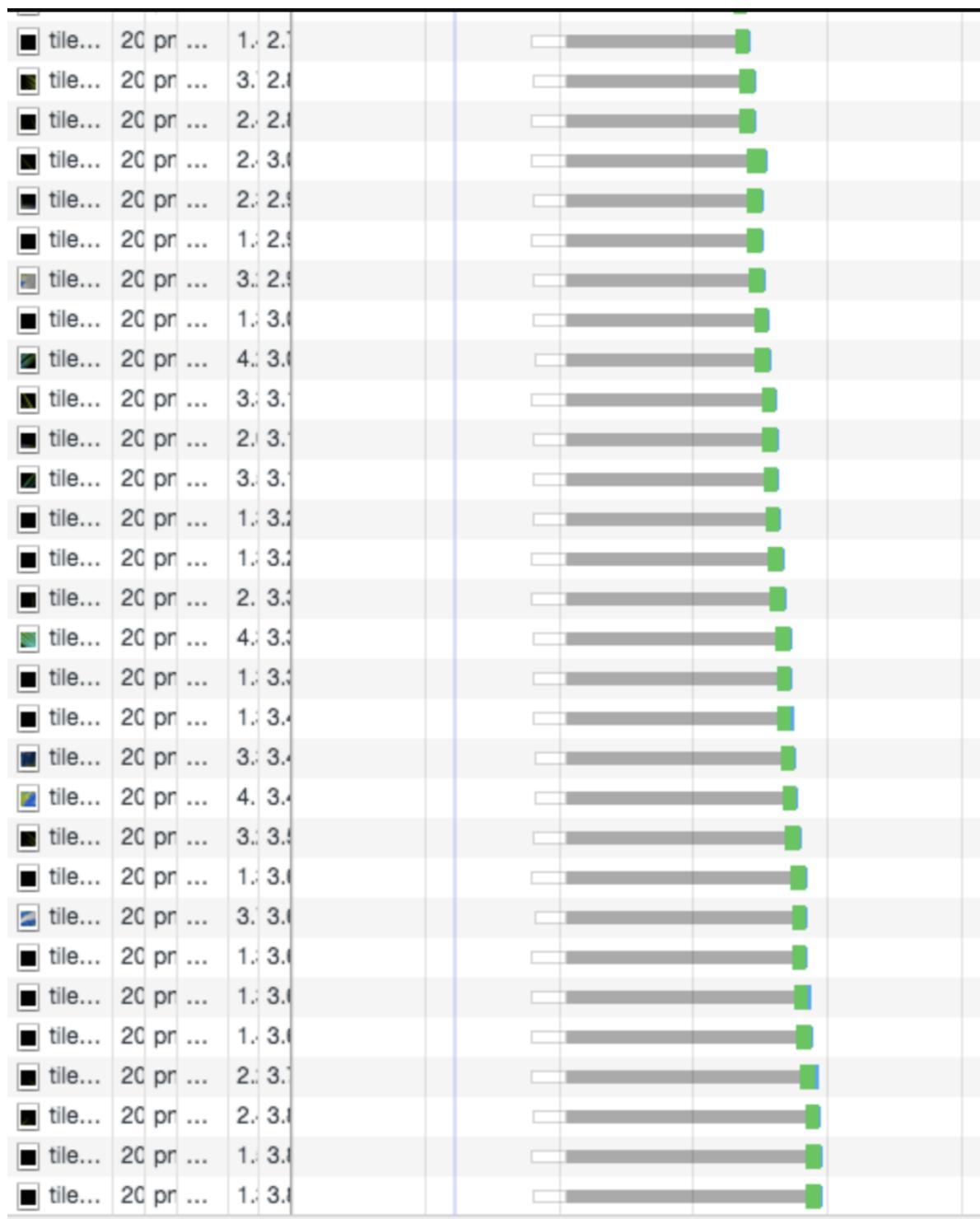
#33.1 HTTP/2

- `HTTP/2` 相比于 `HTTP/1`，可以说是大幅度提高了网页的性能。
- 在 `HTTP/1` 中，为了性能考虑，我们会引入雪碧图、将小图内联、使用多个域名等等的方式。这一切都是因为浏览器限制了同一个域名下的请求数量（Chrome 下一般是限制六个连接），当页面中需要请求很多资源的时候，队头阻塞（Head of line blocking）会导致在达到最大请求数量时，剩余的资源需要等待其他资源请求完成后才能发起请求。
- 在 `HTTP/2` 中引入了多路复用的技术，这个技术可以只通过一个 `TCP` 连接就可以传输所有的请求数据。多路复用很好的解决了浏览器限制同一个域名下的请求数量的问题，同时也接更容易实现全速传输，毕竟新开一个 `TCP` 连接都需要慢慢提升传输速度。

大家可以通过 [该链接](#) 感受下 `HTTP/2` 比 `HTTP/1` 到底快了多少



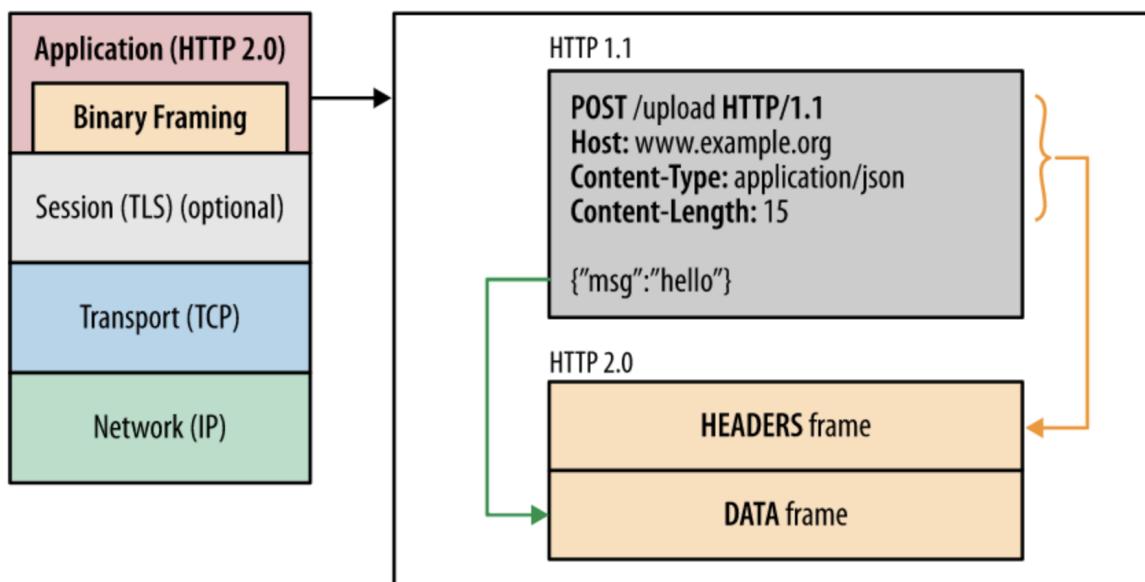
在 `HTTP/1` 中，因为队头阻塞的原因，你会发现发送请求是长这样的



在 `HTTP/2` 中，因为可以复用同一个 `TCP` 连接，你会发现发送请求是长这样的

#33.2 二进制传输

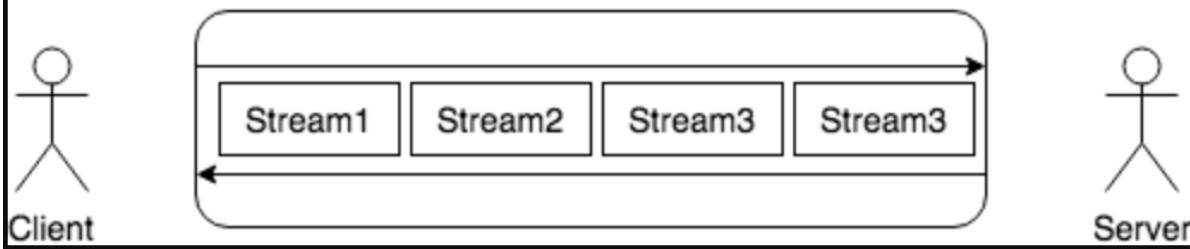
`HTTP/2` 中所有加强性能的核心点在此。在之前的 `HTTP` 版本中，我们是通过文本的方式传输数据。在 `HTTP/2` 中引入了新的编码机制，所有传输的数据都会被分割，并采用二进制格式编码。



#33.3 多路复用

- 在 `HTTP/2` 中，有两个非常重要的概念，分别是帧（frame）和流（stream）。
 - 帧代表着最小的数据单位，每个帧会标识出该帧属于哪个流，流也就是多个帧组成的数据流。
 - 多路复用，就是在一个 `TCP` 连接中可以存在多条流。换句话说，也就是可以发送多个请求，对端可以通过帧中的标识知道属于哪个请求。通过这个技术，可以避免 `HTTP` 旧版本中的队头阻塞问题，极大的提高传输性能。

HTTP 2.0 Connect



#33.4 Header 压缩

- 在 `HTTP/1` 中，我们使用文本的形式传输 `header`，在 `header` 携带 cookie 的情况下，可能每次都需要重复传输几百到几千的字节。
- 在 `HTTP / 2` 中，使用了 `HPACK` 压缩格式对传输的 `header` 进行编码，减少了 `header` 的大小。并在两端维护了索引表，用于记录出现过的 `header`，后面在传输过程中就可以传输已经记录过的 `header` 的键名，对端收到数据后就可以通过键名找到对应的值。

#33.5 服务端 Push

- 在 `HTTP/2` 中，服务端可以在客户端某个请求后，主动推送其他资源。
- 可以想象以下情况，某些资源客户端是一定会请求的，这时就可以采取服务端 `push` 的技术，提前给客户端推送必要的资源，这样就可以相对减少一点延迟时间。当然在浏览器兼容的情况下你也可以使用 `prefetch`

#33.6 HTTP/3

- 虽然 `HTTP/2` 解决了很多之前旧版本的问题，但是它还是存在一个巨大的问题，虽然这个问题并不是它本身造成的，而是底层支撑的 `TCP` 协议的问题。
- 因为 `HTTP/2` 使用了多路复用，一般来说同一域名下只需要使用一个 `TCP` 连接。当这个连接中出现了丢包的情况，那就会导致 `HTTP/2` 的表现情况反倒不如 `HTTP/1` 了。
- 因为在出现丢包的情况下，整个 `TCP` 都要开始等待重传，也就导致了后面的所有数据都被阻塞了。但是对于 `HTTP/1` 来说，可以开启多个 `TCP` 连接，出现这种情况反到只会影响其中一个连接，剩余的 `TCP` 连接还可以正常传输数据。
- 那么可能就会有人考虑到去修改 `TCP` 协议，其实这已经是一件不可能完成的任务了。因为 `TCP` 存在的时间实在太长，已经充斥在各种设备中，并且这个协议是由操作系统实现的，更新起来不大现实。
- 基于这个原因，Google 就另起炉灶搞了一个基于 `UDP` 协议的 `QUIC` 协议，并且使用在了 `HTTP/3` 上，当然 `HTTP/3` 之前名为 `HTTP-over-QUIC`，从这个名字中我们也可以发现，`HTTP/3` 最大的改造就是使用了 `QUIC`，接下来我们就来学习关于这个协议的内容。

QUIC

之前我们学习过 `UDP` 协议的内容，知道这个协议虽然效率很高，但是并不是那么的可靠。`QUIC` 虽然基于 `UDP`，但是在原本的基础上新增了很多功能，比如多路复用、0-RTT、使用 `TLS1.3` 加密、流量控制、有序交付、重传等等功能。这里我们就挑选几个重要的功能学习下这个协议的内容。

多路复用

虽然 `HTTP/2` 支持了多路复用，但是 `TCP` 协议终究是没有这个功能的。`QUIC` 原生就实现了这个功能，并且传输的单个数据流可以保证有序交付且不会影响其他的数据流，这样的技术就解决了之前 `TCP` 存在的问题。

- 并且 QUIC 在移动端的表现也会比 TCP 好。因为 TCP 是基于 IP 和端口去识别连接的，这种方式在多变的移动端网络环境下是很脆弱的。但是 QUIC 是通过 ID ** 的方式去识别一个连接，不管你网络环境如何变化，只要 ID 不变，就能迅速重连上。

0-RTT

通过使用类似 TCP 快速打开的技术，缓存当前会话的上下文，在下次恢复会话的时候，只需要将之前的缓存传递给服务端验证通过就可以进行传输了。

纠错机制

- 假如说这次我要发送三个包，那么协议会算出这三个包的异或值并单独发出一个校验包，也就是总共发出了四个包。
- 当出现其中的非校验包丢包的情况时，可以通过另外三个包计算出丢失的数据包的内容。
- 当然这种技术只能使用在丢失一个包的情况下，如果出现丢失多个包就不能使用纠错机制了，只能使用重传的方式了

#34 设计模式

设计模式总的来说是一个抽象的概念，前人通过无数次的实践总结出的一套写代码的方式，通过这种方式写的代码可以让别人更加容易阅读、维护以及复用。

#34.1 工厂模式

工厂模式分为好几种，这里就不一一讲解了，以下是一个简单工厂模式的例子

```
class Man {
  constructor(name) {
    this.name = name
  }
  alertName() {
    alert(this.name)
  }
}

class Factory {
  static create(name) {
    return new Man(name)
  }
}

Factory.create('yck').alertName()
```

- 当然工厂模式并不仅仅是用来 new 出实例。
- 可以想象一个场景。假设有一份很复杂的代码需要用户去调用，但是用户并不关心这些复杂的代码，只需要你提供给我一个接口去调用，用户只负责传递需要的参数，至于这些参数怎么使用，内部有什么逻辑是不关心的，只需要你最后返回我一个实例。这个构造过程就是工厂。
- 工厂起到的作用就是隐藏了创建实例的复杂度，只需要提供一个接口，简单清晰。
- 在 vue 源码中，你也可以看到工厂模式的使用，比如创建异步组件

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
```

```

    children: ?Array<VNode>,
    tag?: string
  ): VNode | Array<VNode> | void {
    // 逻辑处理...

    const vnode = new VNode(
      `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
      data, undefined, undefined, undefined, context,
      { Ctor, propsData, listeners, tag, children },
      asyncFactory
    )
  }

  return vnode
}

```

在上述代码中，我们可以看到我们只需要调用 `createComponent` 传入参数就能创建一个组件实例，但是创建这个实例是很复杂的一个过程，工厂帮助我们隐藏了这个复杂的过程，只需要一句代码调用就能实现功能

#34.2 单例模式

- 单例模式很常用，比如全局缓存、全局状态管理等等这些只需要一个对象，就可以使用单例模式。
- 单例模式的核心就是保证全局只有一个对象可以访问。因为 JS 是一门无类的语言，所以别的语言实现单例的方式并不能套入 JS 中，我们只需要用一个变量确保实例只创建一次就行，以下是如何实现单例模式的例子

```

class Singleton {
  constructor() {}
}

Singleton.getInstance = (function() {
  let instance
  return function() {
    if (!instance) {
      instance = new Singleton()
    }
    return instance
  }
})()

```

```

let s1 = Singleton.getInstance()
let s2 = Singleton.getInstance()
console.log(s1 === s2) // true

```

在 `vuex` 源码中，你也可以看到单例模式的使用，虽然它的实现方式不大一样，通过一个外部变量来控制只安装一次 `vuex`

```

let vue // bind on install

export function install (_vue) {
  if (vue && _vue === vue) {
    // 如果发现 vue 有值，就不重新创建实例了
    return
  }
  vue = _vue
  applyMixin(vue)
}

```

#34.3 适配器模式

- 适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。
- 以下是如何实现适配器模式的例子

```

class Plug {
  getName() {
    return '港版插头'
  }
}

class Target {
  constructor() {
    this.plug = new Plug()
  }
  getName() {
    return this.plug.getName() + ' 适配器转二脚插头'
  }
}

let target = new Target()
target.getName() // 港版插头 适配器转二脚插头

```

在 `vue` 中，我们其实经常使用到适配器模式。比如父组件传递给子组件一个时间戳属性，组件内部需要将时间戳转为正常的日期显示，一般会使用 `computed` 来做转换这件事情，这个过程就使用到了适配器模式

#34.4 装饰模式

- 装饰模式不需要改变已有的接口，作用是给对象添加功能。就像我们经常需要给手机戴个保护套防摔一样，不改变手机自身，给手机添加了保护套提供防摔功能。
- 以下是如何实现装饰模式的例子，使用了 ES7 中的装饰器语法

```

function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Test {
  @readonly
  name = 'yck'
}

let t = new Test()

t.yck = '111' // 不可修改

```

在 `React` 中，装饰模式其实随处可见

```

import { connect } from 'react-redux'
class MyComponent extends React.Component {
  // ...
}
export default connect(mapStateToProps)(MyComponent)

```

#34.5 代理模式

- 代理是为了控制对对象的访问，不让外部直接访问到对象。在现实生活中，也有很多代理的场景。比如你需要买一件国外的产品，这时候你可以通过代购来购买产品。
- 在实际代码中其实代理的场景很多，也就不举框架中的例子了，比如事件代理就用到了代理模式

```

<ul id="u1">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
<script>
  let ul = document.querySelector('#u1')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>

```

因为存在太多的 `li`，不可能每个都去绑定事件。这时候可以通过给父节点绑定一个事件，让父节点作为代理去拿到真实点击的节点。

#34.6 发布-订阅模式

- 发布-订阅模式也叫做观察者模式。通过一对一或者一对多的依赖关系，当对象发生改变时，订阅方都会收到通知。在现实生活中，也有很多类似场景，比如我在购物网站上购买一个产品，但是发现该产品目前处于缺货状态，这时候我可以点击有货通知的按钮，让网站在产品有货的时候通过短信通知我。
- 在实际代码中其实发布-订阅模式也很常见，比如我们点击一个按钮触发了点击事件就是使用了该模式

```
<ul id="u1"></ul>
<script>
  let u1 = document.querySelector('#u1')
  u1.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```

在 `Vue` 中，如何实现响应式也是使用了该模式。对于需要实现响应式的对象来说，在 `get` 的时候会进行依赖收集，当改变了对象的属性时，就会触发派发更新。

#34.7 外观模式

- 外观模式提供了一个接口，隐藏了内部的逻辑，更加方便外部调用。
- 举个例子来说，我们现在需要实现一个兼容多种浏览器的添加事件方法

```
function addEvent(elm, evType, fn, useCapture) {
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture)
    return true
  } else if (elm.attachEvent) {
    var r = elm.attachEvent("on" + evType, fn)
    return r
  } else {
    elm["on" + evType] = fn
  }
}
```

对于不同的浏览器，添加事件的方式可能会存在兼容问题。如果每次都需要去这样写一遍的话肯定是不能接受的，所以我们将这些判断逻辑统一封装在一个接口中，外部需要添加事件只需要调用 `addEvent` 即可。

#35 常见数据结构

#35.1 时间复杂度

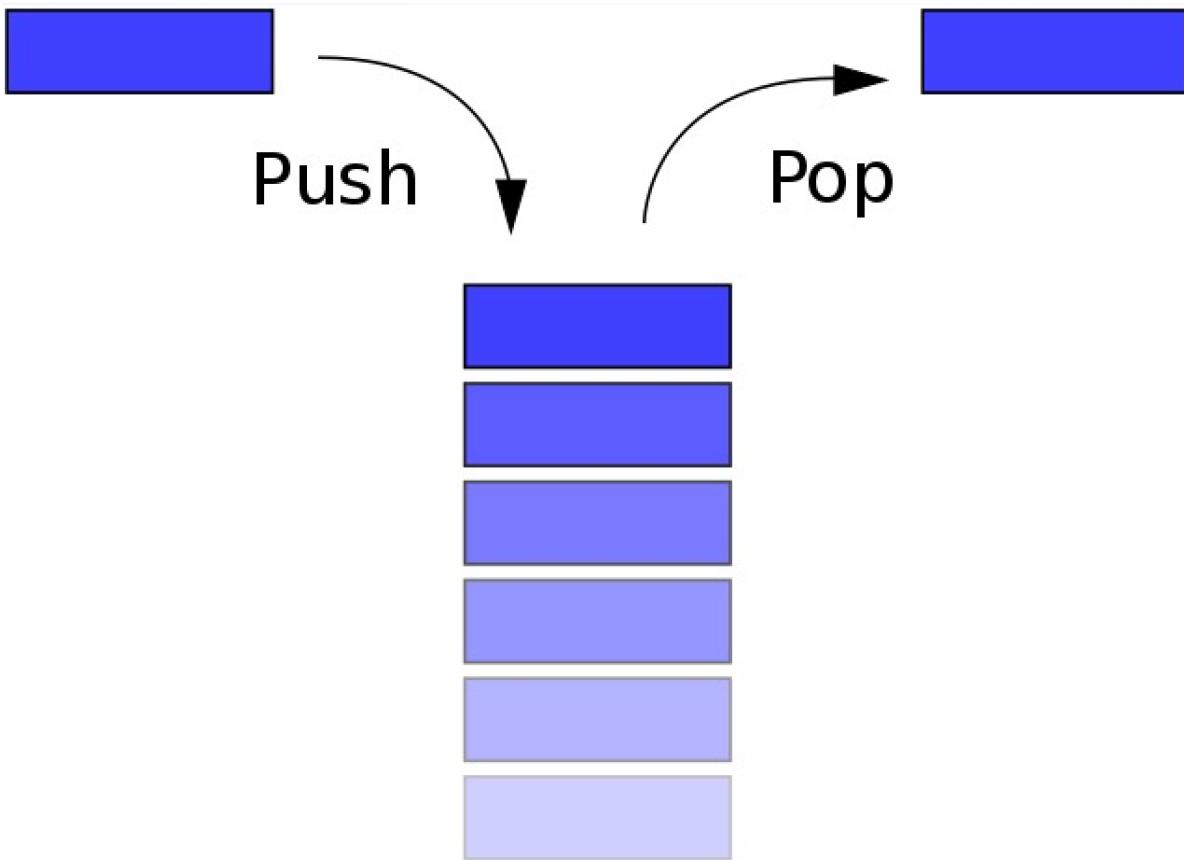
在进入正题之前，我们先来了解下什么是时间复杂度。

- 通常使用最差的时间复杂度来衡量一个算法的好坏。
- 常数时间 $O(1)$ 代表这个操作和数据量没关系，是一个固定时间的操作，比如说四则运算。
- 对于一个算法来说，可能会计算出操作次数为 $aN + 1$ ， N 代表数据量。那么该算法的时间复杂度就是 $O(N)$ 。因为我们在计算时间复杂度的时候，数据量通常是非常大的，这时候低阶项和常数项可以忽略不计。
- 当然可能会出现两个算法都是 $O(N)$ 的时间复杂度，那么对比两个算法的好坏就要通过对比低阶项和常数项了

#35.2 栈

概念

- 栈是一个线性结构，在计算机中是一个相当常见的数据结构。
- 栈的特点是只能在某一端添加或删除数据，遵循先进后出的原则



实现

每种数据结构都可以用很多种方式来实现，其实可以把栈看成是数组的一个子集，所以这里使用数组来实现

```
class Stack {
  constructor() {
    this.stack = []
  }
  push(item) {
    this.stack.push(item)
  }
  pop() {
    this.stack.pop()
  }
  peek() {
    return this.stack[this.getCount() - 1]
  }
  getCount() {
    return this.stack.length
  }
  isEmpty() {
    return this.getCount() === 0
  }
}
```

#35.3 应用

选取了 LeetCode 上序号为 [20 的题](#) 题意是匹配括号，可以通过栈的特性来完成这道题目

```
var isValid = function (s) {
  let map = {
```

```

        ')': -1,
        ')': 1,
        '[': -2,
        ']': 2,
        '{': -3,
        '}': 3
    }
    let stack = []
    for (let i = 0; i < s.length; i++) {
        if (map[s[i]] < 0) {
            stack.push(s[i])
        } else {
            let last = stack.pop()
            if (map[last] + map[s[i]] != 0) return false
        }
    }
    if (stack.length > 0) return false
    return true
};

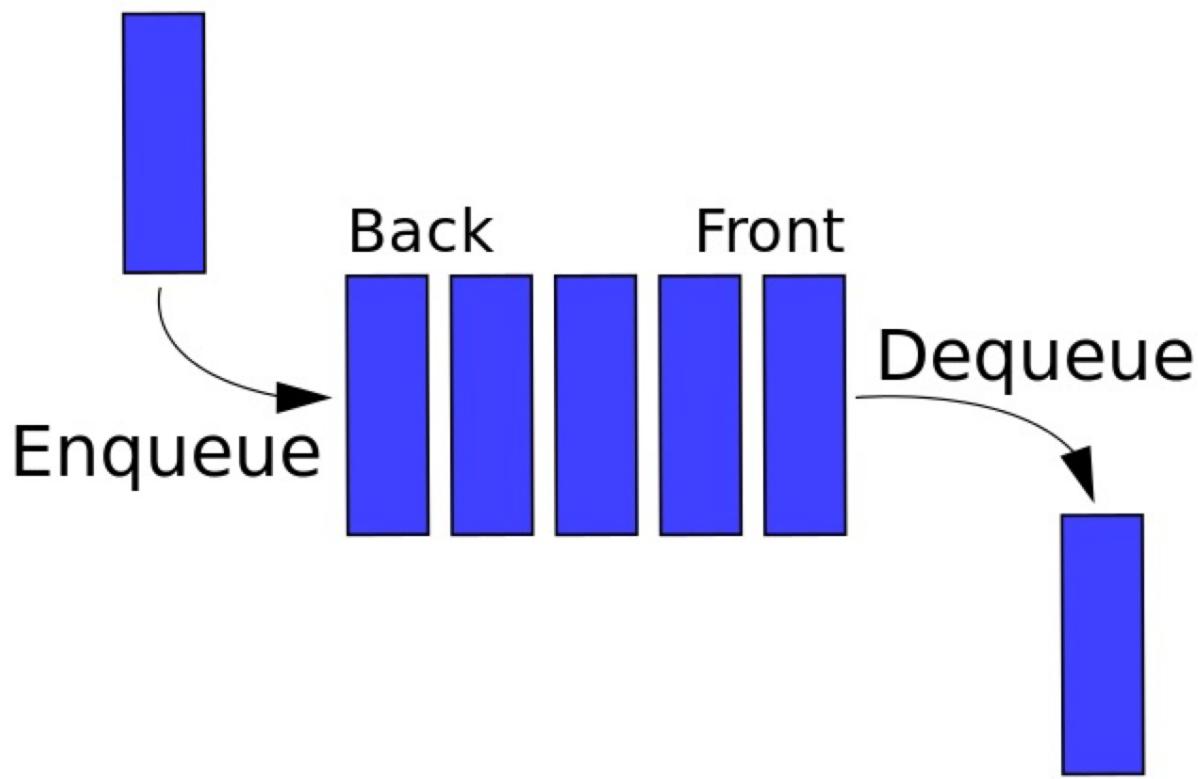
```

其实在 `Vue` 中关于模板解析的代码，就有应用到匹配尖括号的内容

#35.4 队列

概念

队列是一个线性结构，特点是在某一端添加数据，在另一端删除数据，遵循先进先出的原则



实现

这里会讲解两种实现队列的方式，分别是单链队列和循环队列。

单链队列

```
class Queue {
```

```

constructor() {
    this.queue = []
}
enqueue(item) {
    this.queue.push(item)
}
dequeue() {
    return this.queue.shift()
}
getHeader() {
    return this.queue[0]
}
getLength() {
    return this.queue.length
}
isEmpty() {
    return this.getLength() === 0
}
}

```

因为单链队列在出队操作的时候需要 $O(n)$ 的时间复杂度，所以引入了循环队列。循环队列的出队操作平均是 $O(1)$ 的时间复杂度。

循环队列

```

class SqQueue {
constructor(length) {
    this.queue = new Array(length + 1)
    // 队头
    this.first = 0
    // 队尾
    this.last = 0
    // 当前队列大小
    this.size = 0
}
enqueue(item) {
    // 判断队尾 + 1 是否为队头
    // 如果是就代表需要扩容数组
    // % this.queue.length 是为了防止数组越界
    if (this.first === (this.last + 1) % this.queue.length) {
        this.resize(this.getLength() * 2 + 1)
    }
    this.queue[this.last] = item
    this.size++
    this.last = (this.last + 1) % this.queue.length
}
dequeue() {
    if (this.isEmpty()) {
        throw Error('Queue is empty')
    }
    let r = this.queue[this.first]
    this.queue[this.first] = null
    this.first = (this.first + 1) % this.queue.length
    this.size--
    // 判断当前队列大小是否过小
    // 为了保证不浪费空间，在队列空间等于总长度四分之一时
    // 且不为 2 时缩小总长度为当前的一半
}

```

```

        if (this.size === this.getLength() / 4 && this.getLength() / 2 !== 0) {
            this.resize(this.getLength() / 2)
        }
        return r
    }
    getHeader() {
        if (this.isEmpty()) {
            throw Error('Queue is empty')
        }
        return this.queue[this.first]
    }
    getLength() {
        return this.queue.length - 1
    }
    isEmpty() {
        return this.first === this.last
    }
    resize(length) {
        let q = new Array(length)
        for (let i = 0; i < length; i++) {
            q[i] = this.queue[(i + this.first) % this.queue.length]
        }
        this.queue = q
        this.first = 0
        this.last = this.size
    }
}

```

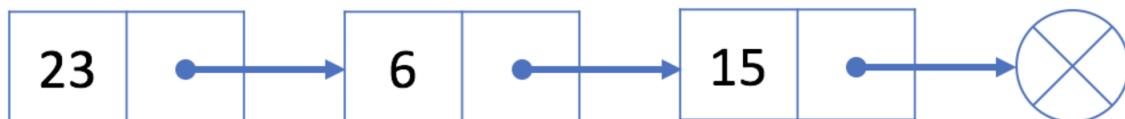
#35.5 链表

概念

链表是一个线性结构，同时也是一个天然的递归结构。链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。

概念

链表是一个线性结构，同时也是一个天然的递归结构。链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。



实现

单向链表

```

class Node {
    constructor(v, next) {
        this.value = v
        this.next = next
    }
}

```

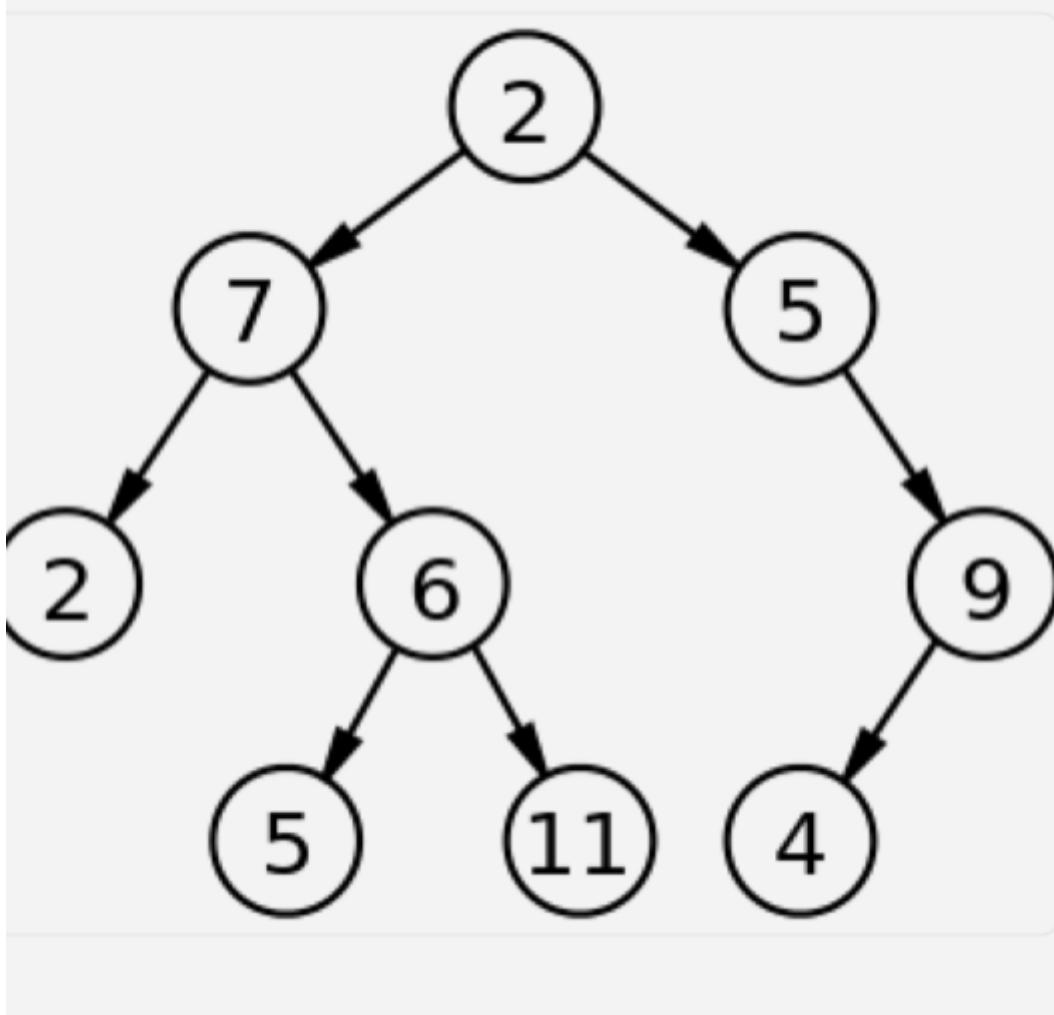
```
class LinkList {
  constructor() {
    // 链表长度
    this.size = 0
    // 虚拟头部
    this.dummyNode = new Node(null, null)
  }
  find(header, index, currentIndex) {
    if (index === currentIndex) return header
    return this.find(header.next, index, currentIndex + 1)
  }
  addNode(v, index) {
    this.checkIndex(index)
    // 当往链表末尾插入时, prev.next 为空
    // 其他情况时, 因为要插入节点, 所以插入的节点
    // 的 next 应该是 prev.next
    // 然后设置 prev.next 为插入的节点
    let prev = this.find(this.dummyNode, index, 0)
    prev.next = new Node(v, prev.next)
    this.size++
    return prev.next
  }
  insertNode(v, index) {
    return this.addNode(v, index)
  }
  addToFirst(v) {
    return this.addNode(v, 0)
  }
  addToLast(v) {
    return this.addNode(v, this.size)
  }
  removeNode(index, isLast) {
    this.checkIndex(index)
    index = isLast ? index - 1 : index
    let prev = this.find(this.dummyNode, index, 0)
    let node = prev.next
    prev.next = node.next
    node.next = null
    this.size--
    return node
  }
  removeFirstNode() {
    return this.removeNode(0)
  }
  removeLastNode() {
    return this.removeNode(this.size, true)
  }
  checkIndex(index) {
    if (index < 0 || index > this.size) throw Error('Index error')
  }
  getNode(index) {
    this.checkIndex(index)
    if (this.isEmpty()) return
    return this.find(this.dummyNode, index, 0).next
  }
  isEmpty() {
    return this.size === 0
  }
}
```

```
getsize() {  
    return this.size  
}  
}
```

#35.6 树

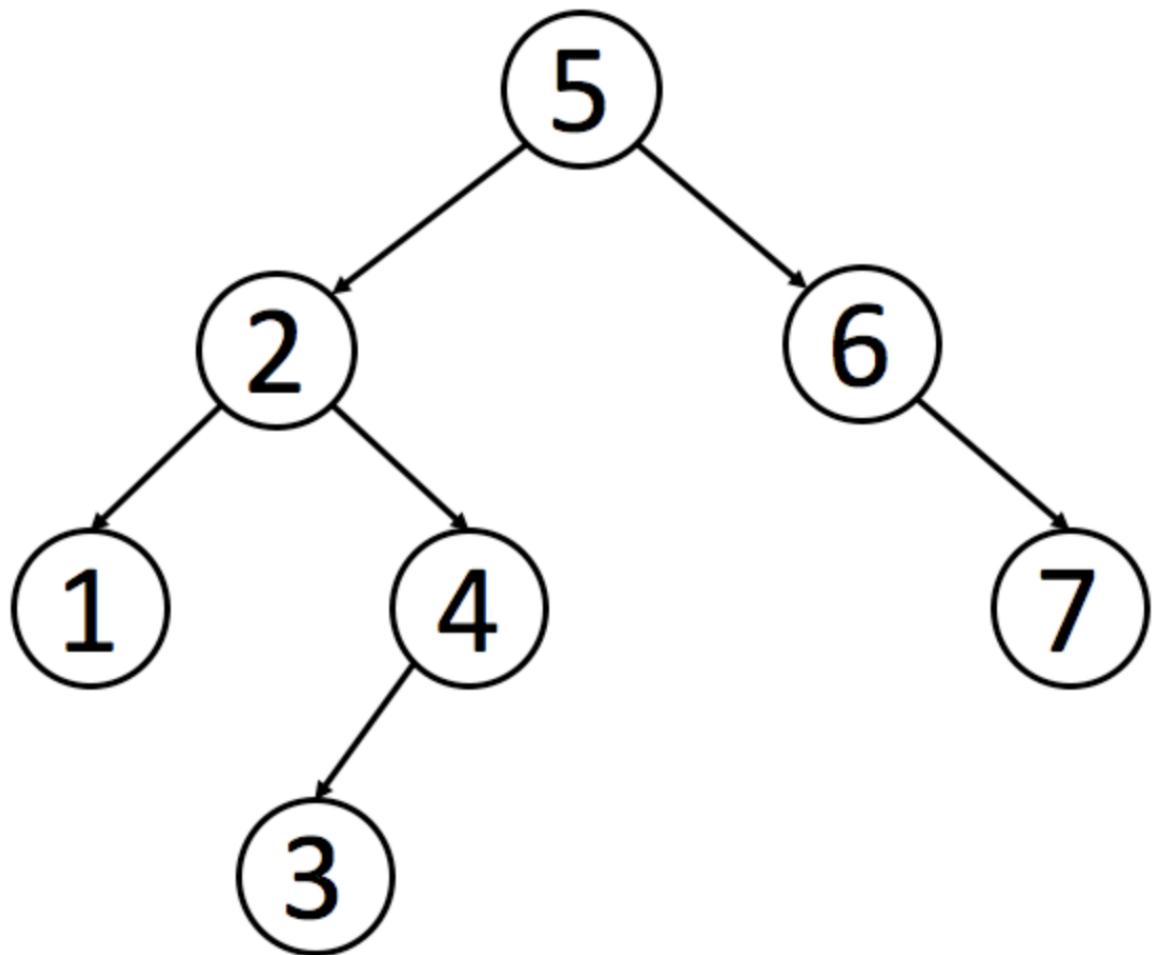
二叉树

- 树拥有很多种结构，二叉树是树中最常用的结构，同时也是一个天然的递归结构。
- 二叉树拥有一个根节点，每个节点至多拥有两个子节点，分别为：左节点和右节点。树的最底部节点称之为叶节点，当一颗树的叶节点数量为满时，该树可以称之为满二叉树。



二分搜索树

- 二分搜索树也是二叉树，拥有二叉树的特性。但是区别在于二分搜索树每个节点的值都比他的左子树的值大，比右子树的值小。
- 这种存储方式很适合于数据搜索。如下图所示，当需要查找 6 的时候，因为需要查找的值比根节点的值大，所以只需要在根节点的右子树上寻找，大大提高了搜索效率。



实现

```
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

class BST {
  constructor() {
    this.root = null
    this.size = 0
  }
  getSize() {
    return this.size
  }
  isEmpty() {
    return this.size === 0
  }
  addNode(v) {
    this.root = this._addChild(this.root, v)
  }
  // 添加节点时，需要比较添加的节点值和当前
  // 节点值的大小
  _addChild(node, v) {
    if (!node) {
      this.size++
      return new Node(v)
    }
  }
}
```

```

    if (node.value > v) {
        node.left = this._addChild(node.left, v)
    } else if (node.value < v) {
        node.right = this._addChild(node.right, v)
    }
    return node
}
}

```

- 以上是最基本的二分搜索树实现，接下来实现树的遍历。
- 对于树的遍历来说，有三种遍历方法，分别是先序遍历、中序遍历、后序遍历。三种遍历的区别在于何时访问节点。在遍历树的过程中，每个节点都会遍历三次，分别是遍历到自己，遍历左子树和遍历右子树。如果需要实现先序遍历，那么只需要第一次遍历到节点时进行操作即可。

```

// 先序遍历可用于打印树的结构
// 先序遍历先访问根节点，然后访问左节点，最后访问右节点。
preTraversal() {
    this._pre(this.root)
}

_pre(node) {
    if (node) {
        console.log(node.value)
        this._pre(node.left)
        this._pre(node.right)
    }
}

// 中序遍历可用于排序
// 对于 BST 来说，中序遍历可以实现一次遍历就
// 得到有序的值
// 中序遍历表示先访问左节点，然后访问根节点，最后访问右节点。
midTraversal() {
    this._mid(this.root)
}

_mid(node) {
    if (node) {
        this._mid(node.left)
        console.log(node.value)
        this._mid(node.right)
    }
}

// 后序遍历可用于先操作子节点
// 再操作父节点的场景
// 后序遍历表示先访问左节点，然后访问右节点，最后访问根节点。
backTraversal() {
    this._back(this.root)
}

_back(node) {
    if (node) {
        this._back(node.left)
        this._back(node.right)
        console.log(node.value)
    }
}

```

以上的这几种遍历都可以称之为深度遍历，对应的还有种遍历叫做广度遍历，也就是一层层地遍历树。对于广度遍历来说，我们需要利用之前讲过的队列结构来完成。

```

breadthTraversal() {
    if (!this.root) return null
    let q = new Queue()
    // 将根节点入队
    q.enqueue(this.root)
    // 循环判断队列是否为空，为空
    // 代表树遍历完毕
    while (!q.isEmpty()) {
        // 将队首出队，判断是否有左右子树
        // 有的话，就先左后右入队
        let n = q.dequeue()
        console.log(n.value)
        if (n.left) q.enqueue(n.left)
        if (n.right) q.enqueue(n.right)
    }
}

```

接下来先介绍如何在树中寻找最小值或最大数。因为二分搜索树的特性，所以最小值一定在根节点的最左边，最大值相反

```

getMin() {
    return this._getMin(this.root).value
}
_getMin(node) {
    if (!node.left) return node
    return this._getMin(node.left)
}
getMax() {
    return this._getMax(this.root).value
}
_getMax(node) {
    if (!node.right) return node
    return this._getMax(node.right)
}

```

向上取整和向下取整，这两个操作是相反的，所以代码也是类似的，这里只介绍如何向下取整。既然是向下取整，那么根据二分搜索树的特性，值一定在根节点的左侧。只需要一直遍历左子树直到当前节点的值不再大于等于需要的值，然后判断节点是否还拥有右子树。如果说有的话，继续上面的递归判断。

```

floor(v) {
    let node = this._floor(this.root, v)
    return node ? node.value : null
}
_floor(node, v) {
    if (!node) return null
    if (node.value === v) return v
    // 如果当前节点值还比需要的值大，就继续递归
    if (node.value > v) {
        return this._floor(node.left, v)
    }
    // 判断当前节点是否拥有右子树
    let right = this._floor(node.right, v)
    if (right) return right
    return node
}

```

```
}
```

排名，这是用于获取给定值的排名或者排名第几的节点的值，这两个操作也是相反的，所以这个只介绍如何获取排名第几的节点的值。对于这个操作而言，我们需要略微的改造点代码，让每个节点拥有一个 size 属性。该属性表示该节点下有多少子节点（包含自身）

```
class Node {
    constructor(value) {
        this.value = value
        this.left = null
        this.right = null
        // 修改代码
        this.size = 1
    }
}
// 新增代码
_getSize(node) {
    return node ? node.size : 0
}
addChild(node, v) {
    if (!node) {
        return new Node(v)
    }
    if (node.value > v) {
        // 修改代码
        node.size++
        node.left = this._addChild(node.left, v)
    } else if (node.value < v) {
        // 修改代码
        node.size++
        node.right = this._addChild(node.right, v)
    }
    return node
}
select(k) {
    let node = this._select(this.root, k)
    return node ? node.value : null
}
_select(node, k) {
    if (!node) return null
    // 先获取左子树下有几个节点
    let size = node.left ? node.left.size : 0
    // 判断 size 是否大于 k
    // 如果大于 k，代表所需要的节点在左节点
    if (size > k) return this._select(node.left, k)
    // 如果小于 k，代表所需要的节点在右节点
    // 注意这里需要重新计算 k，减去根节点除了右子树的节点数量
    if (size < k) return this._select(node.right, k - size - 1)
    return node
}
```

接下来讲解的是二分搜索树中最难实现的部分：删除节点。因为对于删除节点来说，会存在以下几种情况

- 需要删除的节点没有子树
- 需要删除的节点只有一条子树

- 需要删除的节点有左右两条树

对于前两种情况很好解决，但是第三种情况就有难度了，所以先来实现相对简单的操作：删除最小节点，对于删除最小节点来说，是不存在第三种情况的，删除最大节点操作是和删除最小节点相反的，所以这里也就不再赘述。

```
deleteMin() {
  this.root = this._deleteMin(this.root)
  console.log(this.root)
}

_deleteMin(node) {
  // 一直递归左子树
  // 如果左子树为空，就判断节点是否拥有右子树
  // 有右子树的话就把需要删除的节点替换为右子树
  if ((node != null) & !node.left) return node.right
  node.left = this._deleteMin(node.left)
  // 最后需要重新维护下节点的 `size`
  node.size = this._getSize(node.left) + this._getSize(node.right) + 1
  return node
}
```

- 最后讲解的就是如何删除任意节点了。对于这个操作，T.Hibbard 在 1962 年提出了解决这个难题的办法，也就是如何解决第三种情况。
- 当遇到这种情况时，需要取出当前节点的后继节点（也就是当前节点右子树的最小节点）来替换需要删除的节点。然后将需要删除节点的左子树赋值给后继结点，右子树删除后继结点后赋值给他。
- 你如果对于这个解决办法有疑问的话，可以这样考虑。因为二分搜索树的特性，父节点一定比所有左子节点大，比所有右子节点小。那么当需要删除父节点时，势必需要拿出一个比父节点大的节点来替换父节点。这个节点肯定不存在于左子树，必然存在于右子树。然后又需要保持父节点都是比右子节点小的，那么就可以取出右子树中最小的那个节点来替换父节点。

```
delete(v) {
  this.root = this._delete(this.root, v)
}

_delete(node, v) {
  if (!node) return null
  // 寻找的节点比当前节点小，去左子树找
  if (node.value < v) {
    node.right = this._delete(node.right, v)
  } else if (node.value > v) {
    // 寻找的节点比当前节点大，去右子树找
    node.left = this._delete(node.left, v)
  } else {
    // 进入这个条件说明已经找到节点
    // 先判断节点是否拥有左右子树中的一个
    // 是的话，将子树返回出去，这里和 `deleteMin` 的操作一样
    if (!node.left) return node.right
    if (!node.right) return node.left
    // 进入这里，代表节点拥有左右子树
    // 先取出当前节点的后继结点，也就是取当前节点右子树的最小值
    let min = this._getMin(node.right)
    // 取出最小值后，删除最小值
    // 然后把删除节点后的子树赋值给最小值节点
    min.right = this._deleteMin(node.right)
    // 左子树不动
    min.left = node.left
    node = min
  }
}
```

```

    }
    // 维护 size
    node.size = this._getSize(node.left) + this._getSize(node.right) + 1
    return node
}

```

#35.7 AVL 树

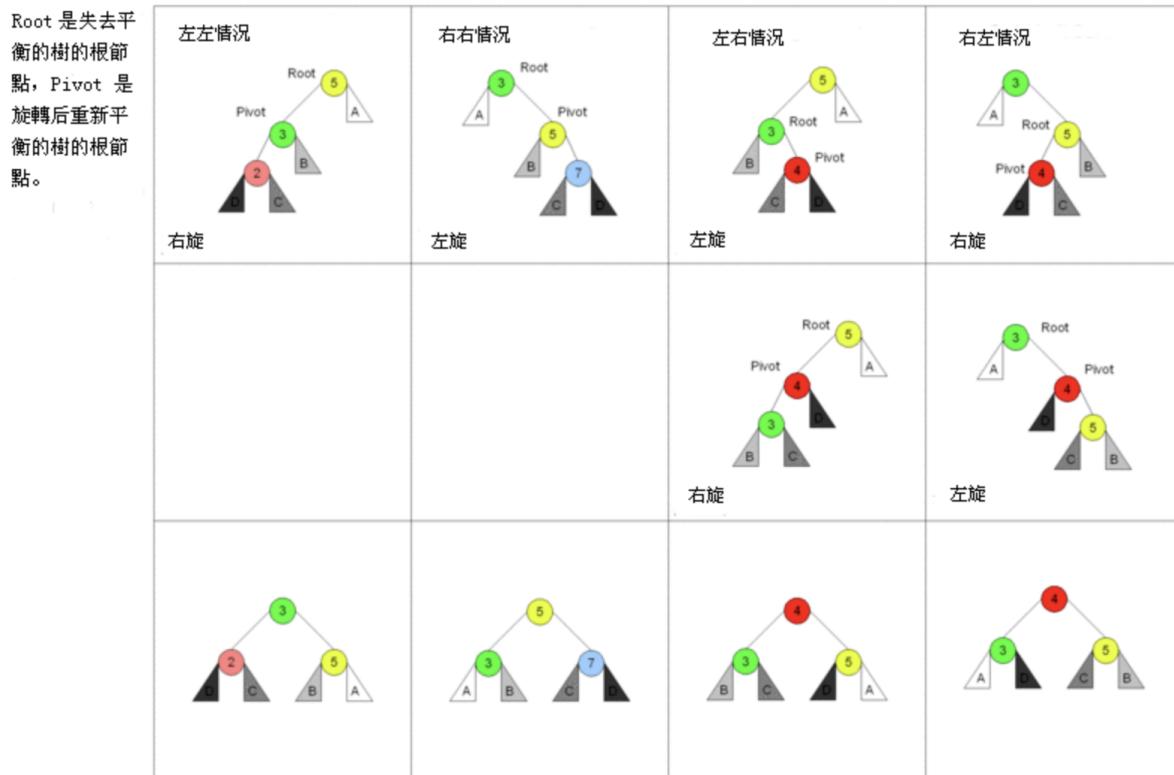
概念

二分搜索树实际在业务中是受到限制的，因为并不是严格的 $O(\log N)$ ，在极端情况下会退化成链表，比如加入一组升序的数字就会造成这种情况。

AVL 树改进了二分搜索树，在 AVL 树中任意节点的左右子树的高度差都不大于 1，这样保证了时间复杂度是严格的 $O(\log N)$ 。基于此，对 AVL 树增加或删除节点时可能需要旋转树来达到高度的平衡。

实现

- 因为 AVL 树是改进了二分搜索树，所以部分代码是于二分搜索树重复的，对于重复内容不作再次解析。
- 对于 AVL 树来说，添加节点会有四种情况



- 对于左左情况来说，新增加的节点位于节点 2 的左侧，这时树已经不平衡，需要旋转。因为搜索树的特性，节点比左节点大，比右节点小，所以旋转以后也要实现这个特性。
- 旋转之前：`new < 2 < C < 3 < B < 5 < A`，右旋之后节点 3 为根节点，这时候需要将节点 3 的右节点加到节点 5 的左边，最后还需要更新节点的高度。
- 对于右右情况来说，相反于左左情况，所以不再赘述。
- 对于左右情况来说，新增加的节点位于节点 4 的右侧。对于这种情况，需要通过两次旋转来达到目的。
- 首先对节点的左节点左旋，这时树满足左左的情况，再对节点进行一次右旋就可以达到目的。

```

class Node {

```

```

constructor(value) {
    this.value = value
    this.left = null
    this.right = null
    this.height = 1
}
}

class AVL {
    constructor() {
        this.root = null
    }
    addNode(v) {
        this.root = this._addChild(this.root, v)
    }
    _addChild(node, v) {
        if (!node) {
            return new Node(v)
        }
        if (node.value > v) {
            node.left = this._addChild(node.left, v)
        } else if (node.value < v) {
            node.right = this._addChild(node.right, v)
        } else {
            node.value = v
        }
        node.height =
            1 + Math.max(this._getHeight(node.left), this._getHeight(node.right))
        let factor = this._getBalanceFactor(node)
        // 当需要右旋时，根节点的左树一定比右树高度高
        if (factor > 1 && this._getBalanceFactor(node.left) >= 0) {
            return this._rightRotate(node)
        }
        // 当需要左旋时，根节点的左树一定比右树高度矮
        if (factor < -1 && this._getBalanceFactor(node.right) <= 0) {
            return this._leftRotate(node)
        }
        // 左右情况
        // 节点的左树比右树高，且节点的左树的右树比节点的左树的左树高
        if (factor > 1 && this._getBalanceFactor(node.left) < 0) {
            node.left = this._leftRotate(node.left)
            return this._rightRotate(node)
        }
        // 右左情况
        // 节点的左树比右树矮，且节点的右树的右树比节点的右树的左树矮
        if (factor < -1 && this._getBalanceFactor(node.right) > 0) {
            node.right = this._rightRotate(node.right)
            return this._leftRotate(node)
        }
        return node
    }
    _getHeight(node) {
        if (!node) return 0
        return node.height
    }
    _getBalanceFactor(node) {
        return this._getHeight(node.left) - this._getHeight(node.right)
    }
}

```

```

    }

    // 节点右旋
    //      5          2
    //      / \        / \
    //      2   6  ==>  1   5
    //      / \        /   / \
    //      1   3      new  3   6
    //      /
    //      new

    _rightRotate(node) {
        // 旋转后新根节点
        let newRoot = node.left
        // 需要移动的节点
        let moveNode = newRoot.right
        // 节点 2 的右节点改为节点 5
        newRoot.right = node
        // 节点 5 左节点改为节点 3
        node.left = moveNode
        // 更新树的高度
        node.height =
            1 + Math.max(this._getHeight(node.left), this._getHeight(node.right))
        newRoot.height =
            1 +
            Math.max(this._getHeight(newRoot.left), this._getHeight(newRoot.right))

        return newRoot
    }

    // 节点左旋
    //      4          6
    //      / \        / \
    //      2   6  ==>  4   7
    //      / \        /   \ \
    //      5   7      2     5   new
    //                  \
    //                  new

    _leftRotate(node) {
        // 旋转后新根节点
        let newRoot = node.right
        // 需要移动的节点
        let moveNode = newRoot.left
        // 节点 6 的左节点改为节点 4
        newRoot.left = node
        // 节点 4 右节点改为节点 5
        node.right = moveNode
        // 更新树的高度
        node.height =
            1 + Math.max(this._getHeight(node.left), this._getHeight(node.right))
        newRoot.height =
            1 +
            Math.max(this._getHeight(newRoot.left), this._getHeight(newRoot.right))

        return newRoot
    }
}

```

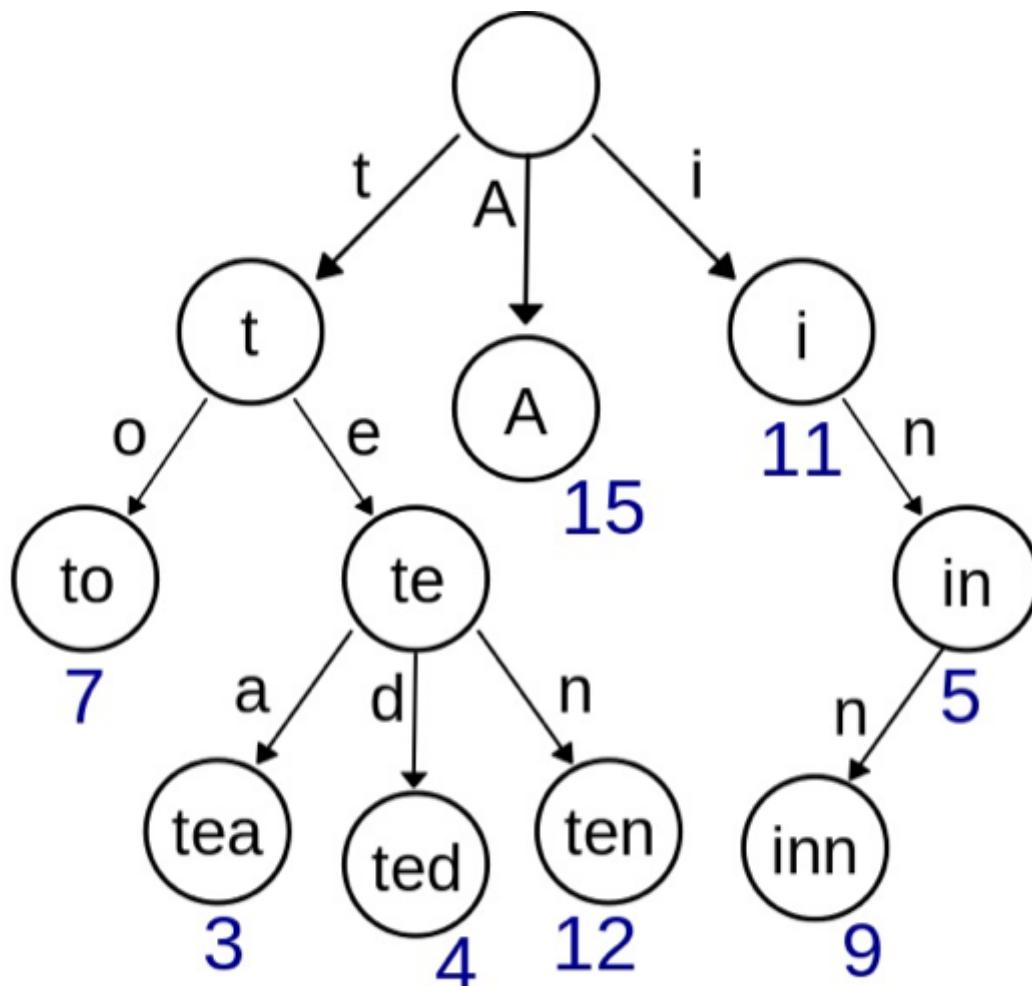
#35.8 Trie

概念

- 在计算机科学，trie，又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是以字符串。

简单点来说，这个结构的作用大多是为了方便搜索字符串，该树有以下几个特点

- 根节点代表空字符串，每个节点都有 N（假如搜索英文字母，就有 26 条）条链接，每条链接代表一个字符
- 节点不存储字符，只有路径才存储，这点和其他的树结构不同
- 从根节点开始到任意一个节点，将沿途经过的字符连接起来就是该节点对应的字符串



实现

总得来说 Trie 的实现相比别的树结构来说简单的很多，实现就以搜索英文字符为例。

```
class TrieNode {  
    constructor() {  
        // 代表每个字符经过节点的次数  
        this.path = 0  
        // 代表到该节点的字符串有几个  
        this.end = 0  
        // 链接  
        this.next = new Array(26).fill(null)  
    }  
}
```

```
class Trie {
    constructor() {
        // 根节点，代表空字符
        this.root = new TrieNode()
    }
    // 插入字符串
    insert(str) {
        if (!str) return
        let node = this.root
        for (let i = 0; i < str.length; i++) {
            // 获得字符先对应的索引
            let index = str[i].charCodeAt() - 'a'.charCodeAt()
            // 如果索引对应没有值，就创建
            if (!node.next[index]) {
                node.next[index] = new TrieNode()
            }
            node.path += 1
            node = node.next[index]
        }
        node.end += 1
    }
    // 搜索字符串出现的次数
    search(str) {
        if (!str) return
        let node = this.root
        for (let i = 0; i < str.length; i++) {
            let index = str[i].charCodeAt() - 'a'.charCodeAt()
            // 如果索引对应没有值，代表没有需要搜索的字符串
            if (!node.next[index]) {
                return 0
            }
            node = node.next[index]
        }
        return node.end
    }
    // 删除字符串
    delete(str) {
        if (!this.search(str)) return
        let node = this.root
        for (let i = 0; i < str.length; i++) {
            let index = str[i].charCodeAt() - 'a'.charCodeAt()
            // 如果索引对应的节点的 Path 为 0，代表经过该节点的字符串
            // 已经一个，直接删除即可
            if (--node.next[index].path == 0) {
                node.next[index] = null
                return
            }
            node = node.next[index]
        }
        node.end -= 1
    }
}
```

#35.9 并查集

概念

- 并查集是一种特殊的树结构，用于处理一些不交集的合并及查询问题。该结构中每个节点都有一个父节点，如果只有当前一个节点，那么该节点的父节点指向自己。

这个结构中有两个重要的操作，分别是：

- `Find`：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- `Union`：将两个子集合并成同一个集合。



`MakeSet` creates 8 singletons.



After some operations of `Union`, some sets are grouped together.

实现

```
class DisjointSet {  
    // 初始化样本  
    constructor(count) {  
        // 初始化时，每个节点的父节点都是自己  
        this.parent = new Array(count)  
        // 用于记录树的深度，优化搜索复杂度  
        this.rank = new Array(count)  
        for (let i = 0; i < count; i++) {  
            this.parent[i] = i  
            this.rank[i] = 1  
        }  
    }  
    find(p) {  
        // 寻找当前节点的父节点是否为自己，不是的话表示还没找到  
        // 开始进行路径压缩优化  
        // 假设当前节点父节点为 A  
        // 将当前节点挂载到 A 节点的父节点上，达到压缩深度的目的  
        while (p != this.parent[p]) {  
            this.parent[p] = this.parent[this.parent[p]]  
            p = this.parent[p]  
        }  
        return p  
    }  
    isConnected(p, q) {  
        return this.find(p) === this.find(q)  
    }  
    // 合并
```

```

union(p, q) {
    // 找到两个数字的父节点
    let i = this.find(p)
    let j = this.find(q)
    if (i === j) return
    // 判断两棵树的深度，深度小的加到深度大的树下面
    // 如果两棵树深度相等，那就无所谓怎么加
    if (this.rank[i] < this.rank[j]) {
        this.parent[i] = j
    } else if (this.rank[i] > this.rank[j]) {
        this.parent[j] = i
    } else {
        this.parent[i] = j
        this.rank[j] += 1
    }
}
}

```

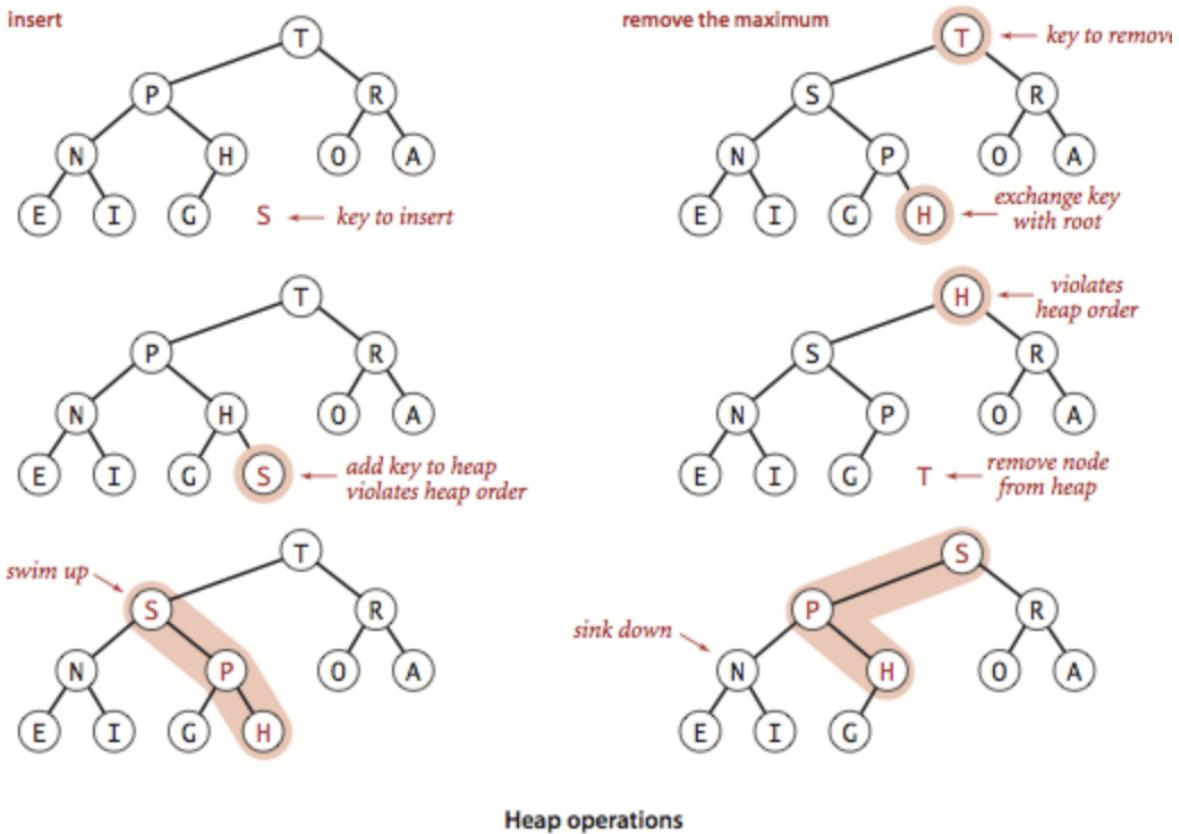
#35.10 堆

概念

- 堆通常是一个可以被看做一棵树的数组对象。
- 堆的实现通过构造二叉堆，实为二叉树的一种。这种数据结构具有以下性质。
- 任意节点小于（或大于）它的所有子节点
 - 堆总是一棵完全树。即除了最底层，其他层的节点都被元素填满，且最底层从左到右填入。
- 将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。
 - 优先队列也完全可以用堆来实现，操作是一模一样的。

实现大根堆

- 堆的每个节点的左边子节点索引是 $i * 2 + 1$ ，右边是 $i * 2 + 2$ ，父节点是 $(i - 1) / 2$ 。
- 堆有两个核心的操作，分别是 `shiftUp` 和 `shiftDown`。前者用于添加元素，后者用于删除根节点。
- `shiftUp` 的核心思路是一路将节点与父节点对比大小，如果比父节点大，就和父节点交换位置。
- `shiftDown` 的核心思路是先将根节点和末尾交换位置，然后移除末尾元素。接下来循环判断父节点和两个子节点的大小，如果子节点大，就把最大的子节点和父节点交换。



Heap operations

```

class MaxHeap {
    constructor() {
        this.heap = []
    }
    size() {
        return this.heap.length
    }
    empty() {
        return this.size() == 0
    }
    add(item) {
        this.heap.push(item)
        this._shiftUp(this.size() - 1)
    }
    removeMax() {
        this._shiftDown(0)
    }
    getParentIndex(k) {
        return parseInt((k - 1) / 2)
    }
    getLeftIndex(k) {
        return k * 2 + 1
    }
    _shiftUp(k) {
        // 如果当前节点比父节点大，就交换
        while (this.heap[k] > this.heap[this.getParentIndex(k)]) {
            this._swap(k, this.getParentIndex(k))
            // 将索引变成父节点
            k = this.getParentIndex(k)
        }
    }
    _shiftDown(k) {
    }
}

```

```

// 交换首位并删除末尾
this._swap(k, this.size() - 1)
this.heap.splice(this.size() - 1, 1)
// 判断节点是否有左孩子，因为二叉堆的特性，有右必有左
while (this.getLeftIndex(k) < this.size()) {
    let j = this.getLeftIndex(k)
    // 判断是否有右孩子，并且右孩子是否大于左孩子
    if (j + 1 < this.size() && this.heap[j + 1] > this.heap[j]) j++
    // 判断父节点是否已经比子节点都大
    if (this.heap[k] >= this.heap[j]) break
    this._swap(k, j)
    k = j
}
}

_swap(left, right) {
let rightValue = this.heap[right]
this.heap[right] = this.heap[left]
this.heap[left] = rightValue
}
}

```

#36 常考算法题解析

对于大部分公司的面试来说，排序的内容已经足以应付了，由此为了更好的符合大众需求，排序的内容是最多的。当然如果你还想冲击更好的公司，那么整一个章节的内容都是需要掌握的。对于字节跳动这类十分看重算法的公司来说，这一章节是远远不够的，剑指Offer应该是你更好的选择

这一章节的内容信息量会很大，不适合在非电脑环境下阅读，请各位打开代码编辑器，一行行的敲代码，单纯阅读是学习不了算法的

另外学习算法的时候，有一个可视化界面会相对减少点学习的难度，具体可以阅读 [algorithm-visualizer](#) 这个仓库

#36.1 位运算

- 在进入正题之前，我们先来学习一下位运算的内容。因为位运算在算法中很有用，速度可以比四则运算快很多。
- 在学习位运算之前应该知道十进制如何转二进制，二进制如何转十进制。这里说明下简单的计算方式
 - 十进制 33 可以看成是 $32 + 1$ ，并且 33 应该是六位二进制的（因为 33 近似 32，而 32 是 2 的五次方，所以是六位），那么十进制 33 就是 100001，只要是 2 的次方，那么就是 1 否则都为 0
 - 那么二进制 100001 同理，首位是 2^5 ，末位是 2^0 ，相加得出 33

1. 左移 <<

```
10 << 1 // -> 20
```

左移就是将二进制全部往左移动，10 在二进制中表示为 1010，左移一位后变成 10100，转换为十进制也就是 20，所以基本可以把左移看成以下公式 $a * (2 ^ b)$

2. 算数右移 >>

```
10 >> 1 // -> 5
```

算数右移就是将二进制全部往右移动并去除多余的右边, 10 在二进制中表示为 1010, 右移一位后变成 101, 转换为十进制也就是 5, 所以基本可以把右移看成以下公式 int v = a / (2 ^ b)

右移很好用, 比如可以用在二分算法中取中间值

```
13 >> 1 // -> 6
```

3. 按位操作

3.1 按位与

每一位都为 1, 结果才为 1

```
8 & 7 // -> 0  
// 1000 & 0111 -> 0000 -> 0
```

3.2 按位或

其中一位为 1, 结果就是 1

```
8 | 7 // -> 15  
// 1000 | 0111 -> 1111 -> 15
```

3.3 按位异或

每一位都不同, 结果才为 1

```
8 ^ 7 // -> 15  
8 ^ 8 // -> 0  
// 1000 ^ 0111 -> 1111 -> 15  
// 1000 ^ 1000 -> 0000 -> 0
```

- 从以上代码中可以发现按位异或就是不进位加法
- 面试题: 两个数不使用四则运算得出和

这道题中可以按位异或, 因为按位异或就是不进位加法, $8 \wedge 8 = 0$ 如果进位了, 就是 16 了, 所以我们只需要将两个数进行异或操作, 然后进位。那么也就是说两个二进制都是 1 的位置, 左边应该有一个进位 1, 所以可以得出以下公式 $a + b = (a \wedge b) + ((a \wedge b) \ll 1)$, 然后通过迭代的方式模拟加法

```
function sum(a, b) {  
    if (a == 0) return b  
    if (b == 0) return a  
    let newA = a ^ b  
    let newB = (a & b) << 1  
    return sum(newA, newB)  
}
```

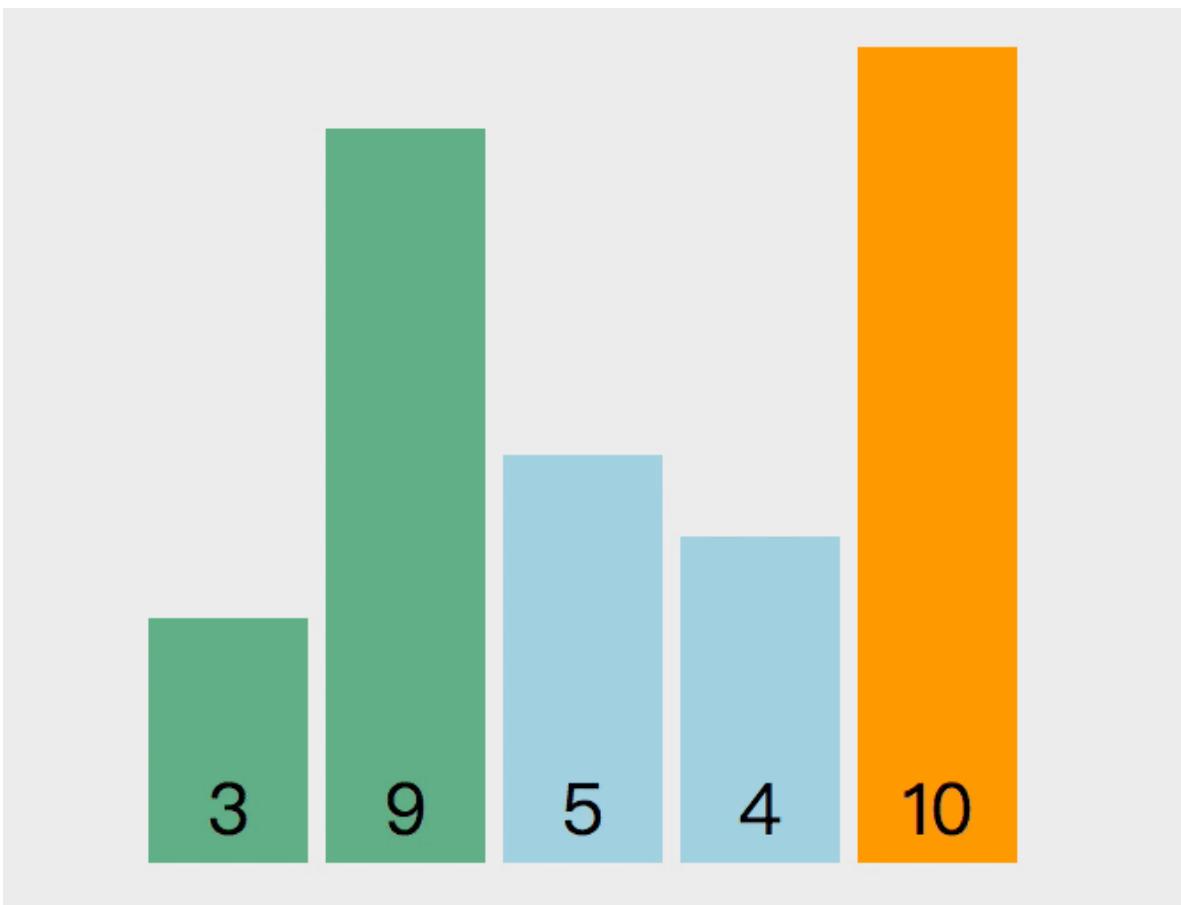
#36.2 排序

以下两个函数是排序中会用到的通用函数，就不一一写了

```
function checkArray(array) {  
    if (!array) return  
}  
function swap(array, left, right) {  
    let rightValue = array[right]  
    array[right] = array[left]  
    array[left] = rightValue  
}
```

#36.2.1 冒泡排序

冒泡排序的原理如下，从第一个元素开始，把当前元素和下一个索引元素进行比较。如果当前元素大，那么就交换位置，重复操作直到比较到最后一个元素，那么此时最后一个元素就是该数组中最大的数。下一轮重复以上操作，但是此时最后一个元素已经是最大数了，所以不需要再比较最后一个元素，只需要比较到 `length - 1` 的位置。



以下是实现该算法的代码

```

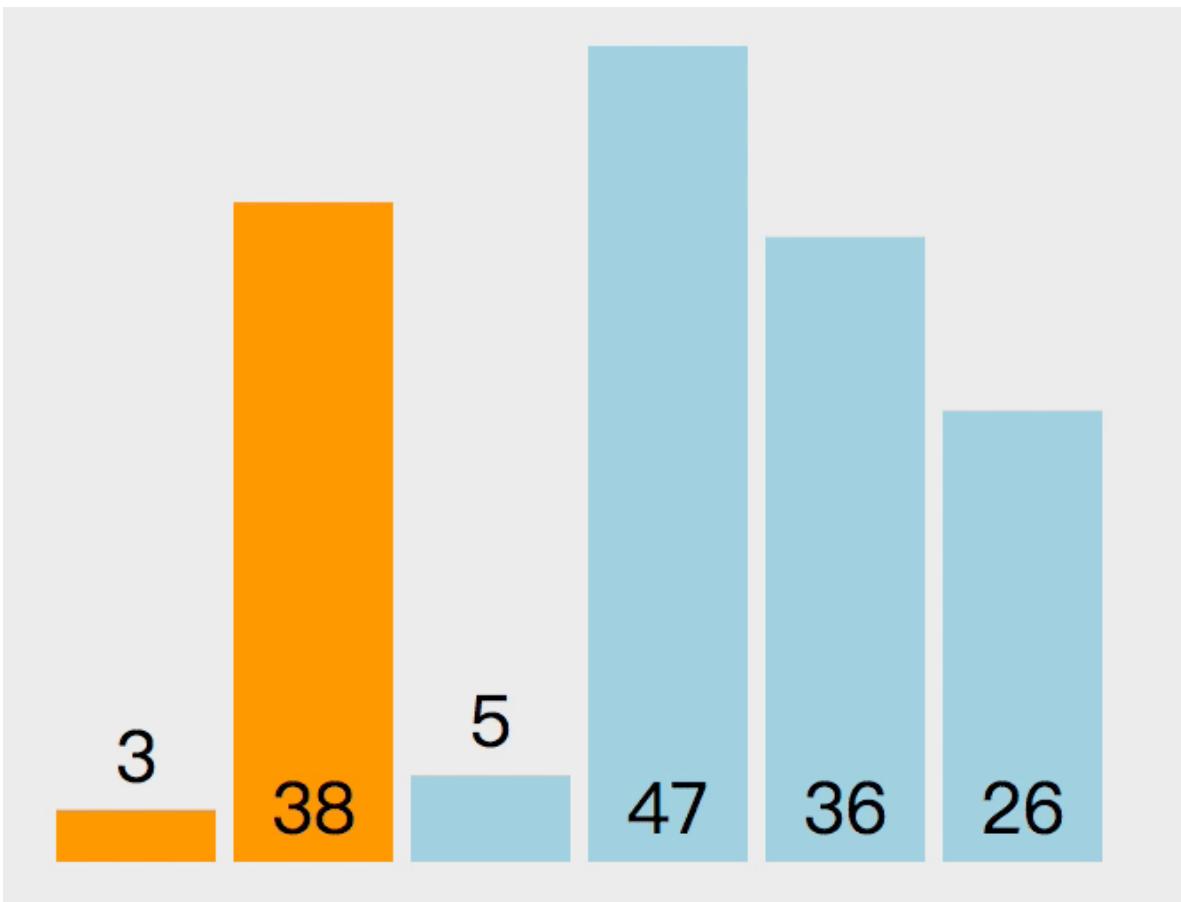
function bubble(array) {
    checkArray(array);
    for (let i = array.length - 1; i > 0; i--) {
        // 从 0 到 `length - 1` 遍历
        for (let j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) swap(array, j, j + 1)
        }
    }
    return array;
}

```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + \dots + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

#36.2.2 插入排序

插入排序的原理如下。第一个元素默认是已排序元素，取出下一个元素和当前元素比较，如果当前元素大就交换位置。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始，向前对比，重复之前的操作



以下是实现该算法的代码

```

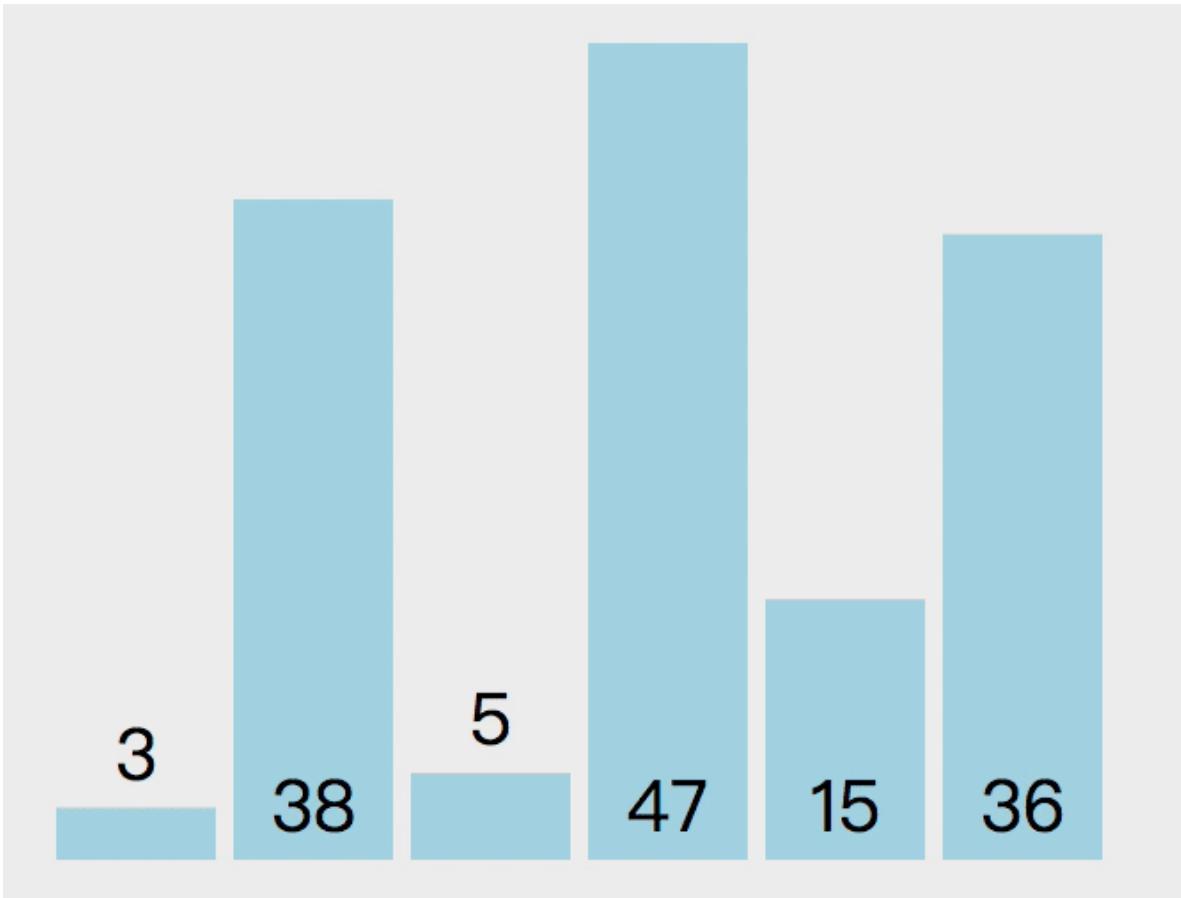
function insertion(array) {
    checkArray(array);
    for (let i = 1; i < array.length; i++) {
        for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)
            swap(array, j, j + 1);
    }
    return array;
}

```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + \dots + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

#36.2.3 选择排序

选择排序的原理如下。遍历数组，设置最小值的索引为 0，如果取出的值比当前最小值小，就替换最小值索引，遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后，第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作



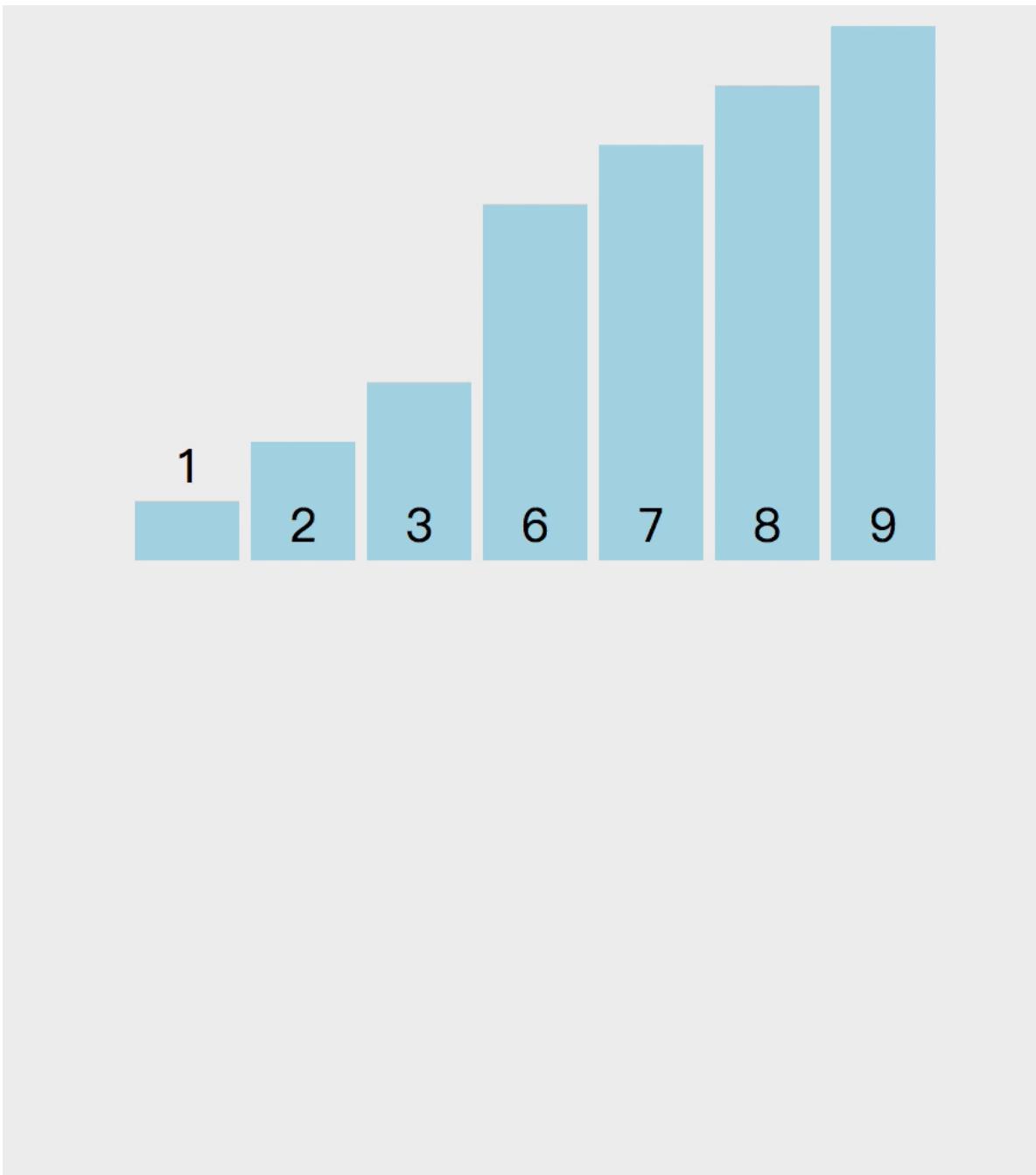
以下是实现该算法的代码

```
function selection(array) {
    checkArray(array);
    for (let i = 0; i < array.length - 1; i++) {
        let minIndex = i;
        for (let j = i + 1; j < array.length; j++) {
            minIndex = array[j] < array[minIndex] ? j : minIndex;
        }
        swap(array, i, minIndex);
    }
    return array;
}
```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + \dots + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

#36.2.4 归并排序

归并排序的原理如下。递归的将数组两两分开直到最多包含两个元素，然后将数组排序合并，最终合并为排序好的数组。假设我有一组数组 [3, 1, 2, 8, 9, 7, 6]，中间数索引是 3，先排序数组 [3, 1, 2, 8]。在这个左边数组上，继续拆分直到变成数组包含两个元素（如果数组长度是奇数的话，会有一个拆分数组只包含一个元素）。然后排序数组 [3, 1] 和 [2, 8]，然后再排序数组 [1, 3, 2, 8]，这样左边数组就排序完成，然后按照以上思路排序右边数组，最后将数组 [1, 2, 3, 8] 和 [6, 7, 9] 排序



以下是实现该算法的代码

```
function sort(array) {
    checkArray(array);
    mergeSort(array, 0, array.length - 1);
    return array;
}

function mergeSort(array, left, right) {
    // 左右索引相同说明已经只有一个数
```

```

if (left === right) return;
// 等同于 `left + (right - left) / 2`
// 相比 `(left + right) / 2` 来说更加安全，不会溢出
// 使用位运算是因为位运算比四则运算快
let mid = parseInt(left + ((right - left) >> 1));
mergeSort(array, left, mid);
mergeSort(array, mid + 1, right);

let help = [];
let i = 0;
let p1 = left;
let p2 = mid + 1;
while (p1 <= mid && p2 <= right) {
    help[i++] = array[p1] < array[p2] ? array[p1++] : array[p2++];
}
while (p1 <= mid) {
    help[i++] = array[p1++];
}
while (p2 <= right) {
    help[i++] = array[p2++];
}
for (let i = 0; i < help.length; i++) {
    array[left + i] = help[i];
}
return array;
}

```

以上算法使用了递归的思想。递归的本质就是压栈，每递归执行一次函数，就将该函数的信息（比如参数，内部的变量，执行到的行数）压栈，直到遇到终止条件，然后出栈并继续执行函数。对于以上递归函数的调用轨迹如下

```

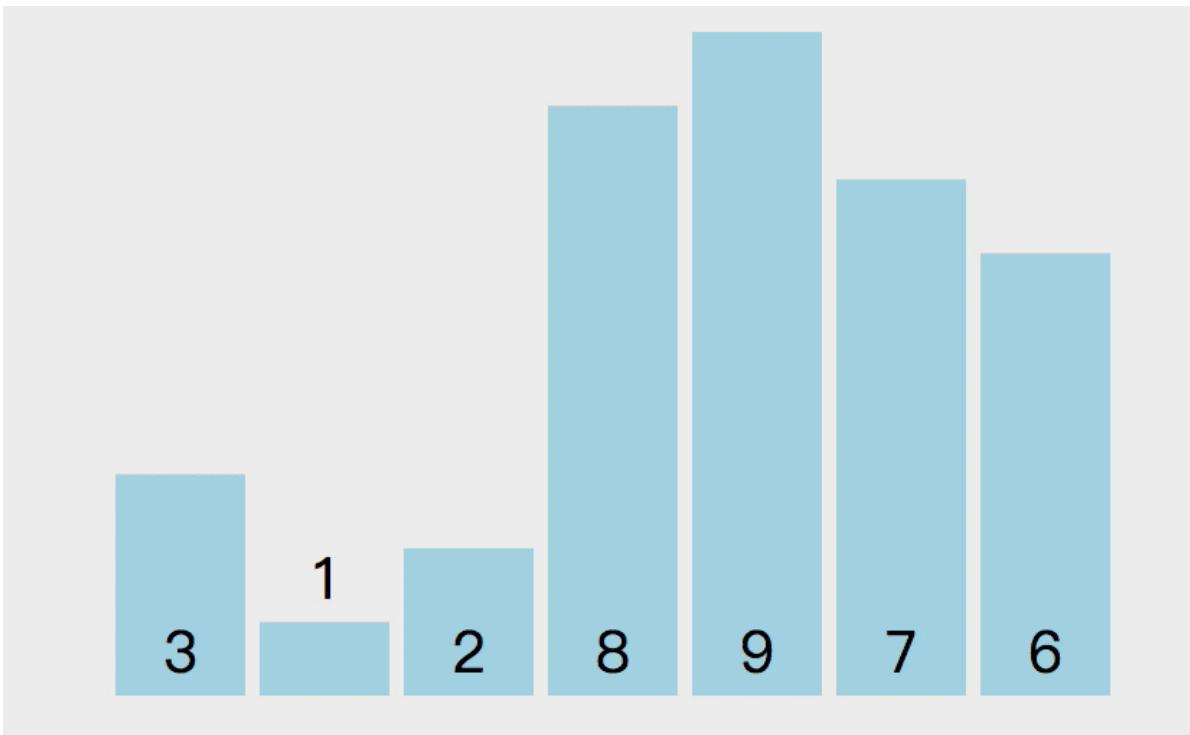
mergeSort(data, 0, 6) // mid = 3
mergeSort(data, 0, 3) // mid = 1
mergeSort(data, 0, 1) // mid = 0
    mergeSort(data, 0, 0) // 遇到终止，回退到上一步
mergeSort(data, 1, 1) // 遇到终止，回退到上一步
// 排序 p1 = 0, p2 = mid + 1 = 1
// 回退到 `mergeSort(data, 0, 3)` 执行下一个递归
mergeSort(2, 3) // mid = 2
mergeSort(3, 3) // 遇到终止，回退到上一步
// 排序 p1 = 2, p2 = mid + 1 = 3
// 回退到 `mergeSort(data, 0, 3)` 执行合并逻辑
// 排序 p1 = 0, p2 = mid + 1 = 2
// 执行完毕回退
// 左边数组排序完毕，右边也是如上轨迹

```

该算法的操作次数是可以这样计算：递归了两次，每次数据量是数组的一半，并且最后把整个数组迭代了一次，所以得出表达式 $2T(N/2) + T(N)$ (T 代表时间， N 代表数据量)。根据该表达式可以套用该公式 得出时间复杂度为 $O(N * \log N)$

#36.2.5 快排

快排的原理如下。随机选取一个数组中的值作为基准值，从左至右取值与基准值对比大小。比基准值小的放数组左边，大的放右边，对比完成后将基准值和第一个比基准值大的值交换位置。然后将数组以基准值的位置分为两部分，继续递归以上操作



以下是实现该算法的代码

```
function sort(array) {
    checkArray(array);
    quickSort(array, 0, array.length - 1);
    return array;
}

function quickSort(array, left, right) {
    if (left < right) {
        swap(array, , right)
        // 随机取值，然后和末尾交换，这样做比固定取一个位置的复杂度略低
        let indexs = part(array, parseInt(Math.random() * (right - left + 1)) + left, right);
        quickSort(array, left, indexs[0]);
        quickSort(array, indexs[1] + 1, right);
    }
}

function part(array, left, right) {
    let less = left - 1;
    let more = right;
    while (left < more) {
        if (array[left] < array[right]) {
            // 当前值比基准值小，`less` 和 `left` 都加一
            ++less;
            ++left;
        } else if (array[left] > array[right]) {
            // 当前值比基准值大，将当前值和右边的值交换
            // 并且不改变 `left`，因为当前换过来的值还没有判断过大小
            swap(array, --more, left);
        } else {
            // 和基准值相同，只移动下标
            left++;
        }
    }
    // 将基准值和比基准值大的第一个值交换位置
}
```

```

    // 这样数组就变成 `[[比基准值小, 基准值, 比基准值大]]` 
    swap(array, right, more);
    return [less, more];
}

```

该算法的复杂度和归并排序是相同的，但是额外空间复杂度比归并排序少，只需 $O(\log N)$ ，并且相比归并排序来说，所需的常数时间也更少。

面试题

Sort Colors: 该题目来自 LeetCode，题目需要我们将 $[2, 0, 2, 1, 1, 0]$ 排序成 $[0, 0, 1, 1, 2, 2]$ ，这个问题就可以使用三路快排的思想。

以下是代码实现

```

var sortColors = function(nums) {
    let left = -1;
    let right = nums.length;
    let i = 0;
    // 下标如果遇到 right, 说明已经排序完成
    while (i < right) {
        if (nums[i] == 0) {
            swap(nums, i++, ++left);
        } else if (nums[i] == 1) {
            i++;
        } else {
            swap(nums, i, --right);
        }
    }
};

```

Kth Largest Element in an Array: 该题目来自 LeetCode，题目需要找出数组中第 K 大的元素，这问题也可以使用快排的思路。并且因为是找出第 K 大元素，所以在分离数组的过程中，可以找出需要的元素在哪边，然后只需要排序相应的一边数组就好。

以下是代码实现

```

var findKthLargest = function(nums, k) {
    let l = 0
    let r = nums.length - 1
    // 得出第 k 大元素的索引位置
    k = nums.length - k
    while (l < r) {
        // 分离数组后获得比基准树大的第一个元素索引
        let index = part(nums, l, r)
        // 判断该索引和 k 的大小
        if (index < k) {
            l = index + 1
        } else if (index > k) {
            r = index - 1
        } else {
            break
        }
    }
    return nums[k];
}

function part(array, left, right) {

```

```

let less = left - 1;
let more = right;
while (left < more) {
    if (array[left] < array[right]) {
        ++less;
        ++left;
    } else if (array[left] > array[right]) {
        swap(array, --more, left);
    } else {
        left++;
    }
}
swap(array, right, more);
return more;
}

```

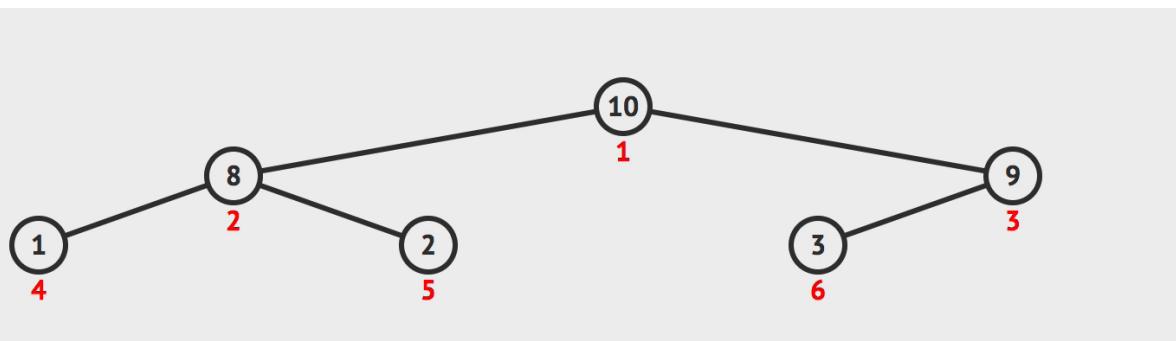
#36.2.6 堆排序

堆排序利用了二叉堆的特性来做，二叉堆通常用数组表示，并且二叉堆是一颗完全二叉树（所有叶节点（最底层的节点）都是从左往右顺序排序，并且其他层的节点都是满的）。二叉堆又分为大根堆与小根堆

- 大根堆是某个节点的所有子节点的值都比他小
- 小根堆是某个节点的所有子节点的值都比他大

堆排序的原理就是组成一个大根堆或者小根堆。以小根堆为例，某个节点的左边子节点索引是 $i * 2 + 1$ ，右边是 $i * 2 + 2$ ，父节点是 $(i - 1) / 2$

1. 首先遍历数组，判断该节点的父节点是否比他小，如果小就交换位置并继续判断，直到他的父节点比他大
2. 重新以上操作 1，直到数组首位是最大值
3. 然后将首位和末尾交换位置并将数组长度减一，表示数组末尾已是最大值，不需要再比较大小
4. 对比左右节点哪个大，然后记住大的节点的索引并且和父节点对比大小，如果子节点大就交换位置
5. 重复以上操作 3 - 4 直到整个数组都是大根堆。



以下是实现该算法的代码

```

function heap(array) {
    checkArray(array);
    // 将最大值交换到首位
    for (let i = 0; i < array.length; i++) {
        heapInsert(array, i);
    }
    let size = array.length;
    // 交换首位和末尾
    swap(array, 0, --size);
    while (size > 0) {

```

```

        heapify(array, 0, size);
        swap(array, 0, --size);
    }
    return array;
}

function heapInsert(array, index) {
    // 如果当前节点比父节点大，就交换
    while (array[index] > array[parseInt((index - 1) / 2)]) {
        swap(array, index, parseInt((index - 1) / 2));
        // 将索引变成父节点
        index = parseInt((index - 1) / 2);
    }
}

function heapify(array, index, size) {
    let left = index * 2 + 1;
    while (left < size) {
        // 判断左右节点大小
        let largest =
            left + 1 < size && array[left] < array[left + 1] ? left + 1 : left;
        // 判断子节点和父节点大小
        largest = array[index] < array[largest] ? largest : index;
        if (largest === index) break;
        swap(array, index, largest);
        index = largest;
        left = index * 2 + 1;
    }
}

```

- 以上代码实现了小根堆，如果需要实现大根堆，只需要把节点对比反一下就好。
- 该算法的复杂度是 $O(n \log n)$

#36.3 链表

反转单向链表

该题目来自 LeetCode，题目需要将一个单向链表反转。思路很简单，使用三个变量分别表示当前节点和当前节点的前后节点，虽然这题很简单，但是却是一道面试常考题

以下是实现该算法的代码

```

var reverseList = function(head) {
    // 判断下变量边界问题
    if (!head || !head.next) return head
    // 初始设置为空，因为第一个节点反转后就是尾部，尾部节点指向 null
    let pre = null
    let current = head
    let next
    // 判断当前节点是否为空
    // 不为空就先获取当前节点的下一节点
    // 然后把当前节点的 next 设为上一个节点
    // 然后把 current 设为下一个节点，pre 设为当前节点
    while(current) {
        next = current.next
        current.next = pre
        pre = current
        current = next
    }
}

```

```
    }
    return pre
};
```

#36.4 树

二叉树的先序，中序，后序遍历

- 先序遍历表示先访问根节点，然后访问左节点，最后访问右节点。
- 中序遍历表示先访问左节点，然后访问根节点，最后访问右节点。
- 后序遍历表示先访问左节点，然后访问右节点，最后访问根节点。

递归实现

递归实现相当简单，代码如下

```
function TreeNode(val) {
  this.val = val;
  this.left = this.right = null;
}
var traversal = function(root) {
  if (root) {
    // 先序
    console.log(root);
    traversal(root.left);
    // 中序
    // console.log(root);
    traversal(root.right);
    // 后序
    // console.log(root);
  }
};
```

对于递归的实现来说，只需要理解每个节点都会被访问三次就明白为什么这样实现了。

非递归实现

非递归实现使用了栈的结构，通过栈的先进后出模拟递归实现。

以下是先序遍历代码实现

```
function pre(root) {
  if (root) {
    let stack = [];
    // 先将根节点 push
    stack.push(root);
    // 判断栈中是否为空
    while (stack.length > 0) {
      // 弹出栈顶元素
      root = stack.pop();
      console.log(root);
      // 因为先序遍历是先左后右，栈是先进后出结构
      // 所以先 push 右边再 push 左边
      if (root.right) {
        stack.push(root.right);
      }
      if (root.left) {
        stack.push(root.left);
      }
    }
  }
}
```

```
        }
    }
}
}
```

以下是中序遍历代码实现

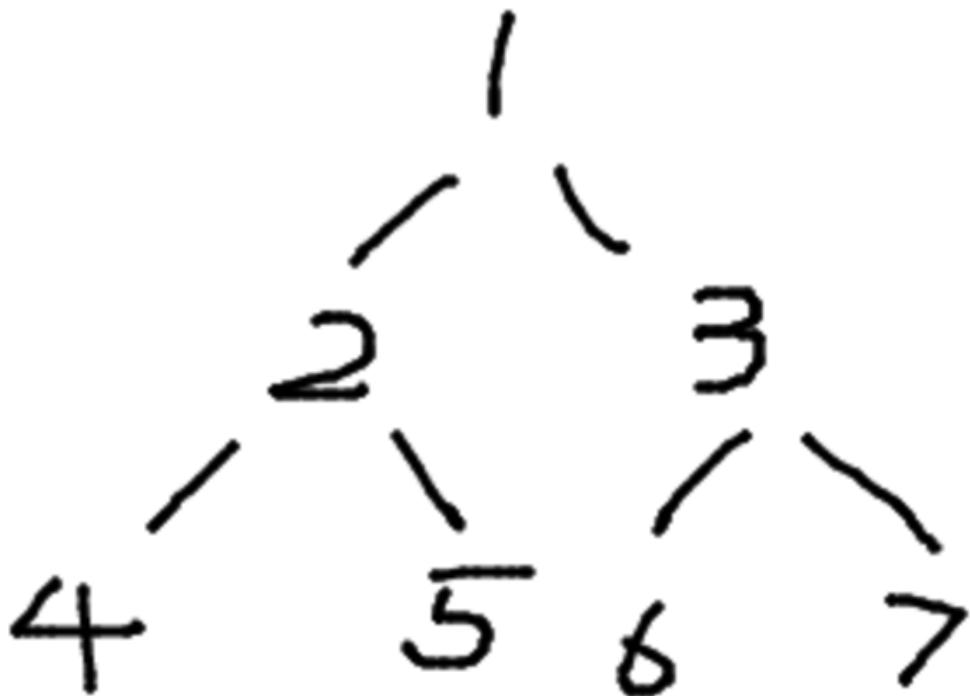
```
function mid(root) {
    if (root) {
        let stack = [];
        // 中序遍历是先左再根最后右
        // 所以首先应该先把最左边节点遍历到底依次 push 进栈
        // 当左边没有节点时，就打印栈顶元素，然后寻找右节点
        // 对于最左边的叶节点来说，可以把它看成是两个 null 节点的父节点
        // 左边打印不出东西就把父节点拿出来打印，然后再看右节点
        while (stack.length > 0 || root) {
            if (root) {
                stack.push(root);
                root = root.left;
            } else {
                root = stack.pop();
                console.log(root);
                root = root.right;
            }
        }
    }
}
```

以下是后序遍历代码实现，该代码使用了两个栈来实现遍历，相比一个栈的遍历来说要容易理解很多

```
function pos(root) {
    if (root) {
        let stack1 = [];
        let stack2 = [];
        // 后序遍历是先左再右最后根
        // 所以对于一个栈来说，应该先 push 根节点
        // 然后 push 右节点，最后 push 左节点
        stack1.push(root);
        while (stack1.length > 0) {
            root = stack1.pop();
            stack2.push(root);
            if (root.left) {
                stack1.push(root.left);
            }
            if (root.right) {
                stack1.push(root.right);
            }
        }
        while (stack2.length > 0) {
            console.log(stack2.pop());
        }
    }
}
```

中序遍历的前驱后继节点

实现这个算法的前提是节点有一个 `parent` 的指针指向父节点，根节点指向 `null`。



如图所示，该树的中序遍历结果是 `4, 2, 5, 1, 6, 3, 7`

前驱节点

对于节点 2 来说，他的前驱节点就是 4，按照中序遍历原则，可以得出以下结论

1. 如果选取的节点的左节点不为空，就找该左节点最右的节点。对于节点 1 来说，他有左节点 2，那么节点 2 的最右节点就是 5
2. 如果左节点为空，且目标节点是父节点的右节点，那么前驱节点为父节点。对于节点 5 来说，没有左节点，且是节点 2 的右节点，所以节点 2 是前驱节点
3. 如果左节点为空，且目标节点是父节点的左节点，向上寻找到第一个是父节点的右节点的节点。对于节点 6 来说，没有左节点，且是节点 3 的左节点，所以向上寻找到节点 1，发现节点 3 是节点 1 的右节点，所以节点 1 是节点 6 的前驱节点

以下是算法实现

```
function predecessor(node) {  
    if (!node) return  
    // 结论 1  
    if (node.left) {  
        return getRight(node.left)  
    } else {  
        let parent = node.parent  
        // 结论 2 3 的判断  
        while (parent && parent.right === node) {  
            node = parent  
            parent = node.parent  
        }  
        return parent  
    }  
}  
function getRight(node) {
```

```

if (!node) return
node = node.right
while(node) node = node.right
return node
}

```

后继节点

- 对于节点 2 来说，他的后继节点就是 5，按照中序遍历原则，可以得出以下结论
 - 如果有右节点，就找到该右节点的最左节点。对于节点 1 来说，他有右节点 3，那么节点 3 的最左节点就是 6
 - 如果没有右节点，就向上遍历直到找到一个节点是父节点的左节点。对于节点 5 来说，没有右节点，就向上寻找到节点 2，该节点是父节点 1 的左节点，所以节点 1 是后继节点

以下是算法实现

```

function successor(node) {
  if (!node) return
  // 结论 1
  if (node.right) {
    return getLeft(node.right)
  } else {
    // 结论 2
    let parent = node.parent
    // 判断 parent 为空
    while(parent && parent.left === node) {
      node = parent
      parent = node.parent
    }
    return parent
  }
}

function getLeft(node) {
  if (!node) return
  node = node.left
  while(node) node = node.left
  return node
}

```

树的深度

树的最大深度：该题目来自 Leetcode，题目需要求出一颗二叉树的最大深度

以下是算法实现

```

var maxDepth = function(root) {
  if (!root) return 0
  return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1
};

```

对于该递归函数可以这样理解：一旦没有找到节点就会返回 0，每弹出一次递归函数就会加一，树有三层就会得到3。

#36.5 动态规划

- 动态规划背后的基本思想非常简单。就是将一个问题拆分为子问题，一般来说这些子问题都是非常相似的，那么我们可以通过只解决一次每个子问题来达到减少计算量的目的。
- 一旦得出每个子问题的解，就存储该结果以便下次使用。

斐波那契数列

斐波那契数列就是从 0 和 1 开始，后面的数都是前两个数之和

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89....
```

那么显然易见，我们可以通过递归的方式来完成求解斐波那契数列

```
function fib(n) {  
    if (n < 2 && n >= 0) return n  
    return fib(n - 1) + fib(n - 2)  
}  
fib(10)
```

以上代码已经可以完美的解决问题。但是以上解法却存在很严重的性能问题，当 n 越大的时候，需要的时间是指数增长的，这时候就可以通过动态规划来解决这个问题。

动态规划的本质其实就是两点

- 自底向上分解子问题
- 通过变量存储已经计算过的解

根据上面两点，我们的斐波那契数列的动态规划思路也就出来了

- 斐波那契数列从 0 和 1 开始，那么这就是这个子问题的最底层
- 通过数组来存储每一位所对应的斐波那契数列的值

```
function fib(n) {  
    let array = new Array(n + 1).fill(null)  
    array[0] = 0  
    array[1] = 1  
    for (let i = 2; i <= n; i++) {  
        array[i] = array[i - 1] + array[i - 2]  
    }  
    return array[n]  
}  
fib(10)
```

0 - 1背包问题

该问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。每个问题只能放入至多一次。

假设我们有以下物品

| 物品 ID / 重量 | 价值 |
|------------|----|
| 1 | 3 |
| 2 | 7 |
| 3 | 12 |

- 对于一个总容量为 5 的背包来说，我们可以放入重量 2 和 3 的物品来达到背包内的物品总价值最高。
- 对于这个问题来说，子问题就两个，分别是放物品和不放物品，可以通过以下表格来理解子问题

| 物品 ID / 剩余容量 | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|----|----|----|
| 1 | 0 | 3 | 3 | 3 | 3 | 3 |
| 2 | 0 | 3 | 7 | 10 | 10 | 10 |
| 3 | 0 | 3 | 7 | 12 | 15 | 19 |

直接来分析能放三种物品的情况，也就是最后一行

- 当容量少于 3 时，只取上一行对应的数据，因为当前容量不能容纳物品 3
- 当容量为 3 时，考虑两种情况，分别为放入物品 3 和不放物品 3
 - 不放物品 3 的情况下，总价值为 10
 - 放入物品 3 的情况下，总价值为 12，所以应该放入物品 3
- 当容量为 4 时，考虑两种情况，分别为放入物品 3 和不放物品 3
 - 不放物品 3 的情况下，总价值为 10
 - 放入物品 3 的情况下，和放入物品 1 的价值相加，得出总价值为 15，所以应该放入物品 3
- 当容量为 5 时，考虑两种情况，分别为放入物品 3 和不放物品 3
 - 不放物品 3 的情况下，总价值为 10
 - 放入物品 3 的情况下，和放入物品 2 的价值相加，得出总价值为 19，所以应该放入物品 3

以下代码对照上表更容易理解

```
/**
 * @param {*} w 物品重量
 * @param {*} v 物品价值
 * @param {*} C 总容量
 * @returns
 */
function knapsack(w, v, C) {
  let length = w.length
  if (length === 0) return 0

  // 对照表格，生成的二维数组，第一维代表物品，第二维代表背包剩余容量
  // 第二维中的元素代表背包物品总价值
  let array = new Array(length).fill(new Array(C + 1).fill(null))

  // 完成底部子问题的解
  for (let i = 0; i <= C; i++) {
    // 对照表格第一行，array[0] 代表物品 1
    // i 代表剩余总容量
    // 当剩余总容量大于物品 1 的重量时，记录下背包物品总价值，否则价值为 0
    array[0][i] = i >= w[0] ? v[0] : 0
  }
}
```

```

}

// 自底向上开始解决子问题，从物品 2 开始
for (let i = 1; i < length; i++) {
  for (let j = 0; j <= C; j++) {
    // 这里求解子问题，分别为不放当前物品和放当前物品
    // 先求不放当前物品的背包总价值，这里的值也就是对应表格中上一行对应的值
    array[i][j] = array[i - 1][j]
    // 判断当前剩余容量是否可以放入当前物品
    if (j >= w[i]) {
      // 可以放入的话，就比大小
      // 放入当前物品和不放入当前物品，哪个背包总价值大
      array[i][j] = Math.max(array[i][j], v[i] + array[i - 1][j - w[i]])
    }
  }
}
return array[length - 1][C]
}

```

最长递增子序列

最长递增子序列意思是在一组数字中，找出最长一串递增的数字，比如

0, 3, 4, 17, 2, 8, 6, 10

对于以上这串数字来说，最长递增子序列就是 0, 3, 4, 8, 10，可以通过以下表格更清晰的理解

| 数字 | 0 | 3 | 4 | 17 | 2 | 8 | 6 | 10 |
|----|---|---|---|----|---|-----|---|----|
| 长度 | 1 | 2 | 3 | 4 | 2 | 4 4 | 5 | |

通过以上表格可以很清晰的发现一个规律，找出刚好比当前数字小的数，并且在小的数组成的长度基础上加一。

这个问题的动态思路解法很简单，直接上代码

```

function lis(n) {
  if (n.length === 0) return 0
  // 创建一个和参数相同大小的数组，并填充值为 1
  let array = new Array(n.length).fill(1)
  // 从索引 1 开始遍历，因为数组已经所有都填充为 1 了
  for (let i = 1; i < n.length; i++) {
    // 从索引 0 遍历到 i
    // 判断索引 i 上的值是否大于之前的值
    for (let j = 0; j < i; j++) {
      if (n[i] > n[j]) {
        array[i] = Math.max(array[i], 1 + array[j])
      }
    }
  }
  let res = 1
  for (let i = 0; i < array.length; i++) {
    res = Math.max(res, array[i])
  }
  return res
}

```

面试指南

#一、准备：简历编写和面试前准备

一般来说，跳槽找工作要经历投递简历、准备面试、面试和谈 offer 四个阶段。其中面试题目会因你的等级和职位而异，从入门级到专家级，广度和深度都会有所增加。不过，不管什么级别和职位，面试题目一般都可以分类为理论知识、算法、项目细节、技术视野、开放性题、工作案例等内容。接下来重点来说下简历和知识点梳理的方法。

#准备一份合适的简历

首先，什么样子的简历更加容易拿到面试邀请？笔者作为一名在 BAT 中待过两家的面试官，见过各种各样的简历，先说一下一些比较不受欢迎的简历：

- 招聘网站上的简历：有些简历是 HR 直接从某招聘网站直接下载下来的，格式统一，而且对于自己的技能还有自己打分，这类简历有可能是候选人根本就没自己精心准备简历，而是网站根据他填写的内容自动生成的，遇到这样的简历笔者一定会让 HR 或者候选人更新一份简历给我。
- 太花俏的简历：有人觉得简历花俏一点会让人眼前一亮，但是公司招聘的是前端不是视觉设计，所以如果找的不是视觉设计工作，还是工工整整的简历会比较受欢迎，而且太花俏的简历有可能让人第一感觉是华而不实，并不是关注候选人的技能。造假或者描述太出格的简历：看到你简历的人可能是不懂技术的 HR，也可能是专业领域的大牛，如果数据造假或者夸大其实，这样很容易就让人给卡掉。

那么，怎样的简历才是好的简历呢？

一份合适的技术型简历最重要的三部分是：

- 个人掌握的技能，是否有岗位需要用到的技能，及其技能掌握的熟练程度：熟悉、了解还是精通
- **项目经历**，项目经历是否对现在岗位有用或者有重叠，是否能够驾驭大型项目
- **实习经历**，对于没有经验的应届生来说，实习经历是很重要的部分，是否有大公司或者具体项目的实习经历是筛选简历的重要参考

技术型简历一般不要太花俏，关键要语言表达通顺清楚，让语言准确和容易理解，在 HR 筛选简历的时候，可以瞬间抓住他的眼球。另外如果有一些特殊奖项，也可以在简历中突出出来，比如：季度之星、最佳个人之类的奖项，应届生会有优秀毕业生、全额奖学金等。

#推荐使用 PDF 版本的简历

一般来说简历会有 Word、Markdown 等版本，这里笔者推荐使用 PDF 版本的简历，主要原因如下：

- 内容丰富，布局调整方便
- 字体等格式有保障，你不知道收到你简历的人用的是什么环境，PDF 版本不会因为不同操作系统等原因而受限
- 便于携带和传播，始终存一份简历在手机或者邮箱内，随时发送
- 不容易被涂改
- 一般 Windows 系统的 Word、Mac 系统的 Pages 都支持导出 PDF 格式的文件，原稿可以保存到云端或者 iCloud，方便以后修改。

虽然我们是 Web 前端工程师，笔者还是不推荐使用 HTML 格式的简历，HTML 版本的简历容易受浏览器等环境因素影响，而且接收方不一定是技术人员，你做的炫酷的效果也不一定会被看到。

#简历最好要有针对性地来写

简历是「敲门砖」，笔者建议根据你想要找的公司、岗位和职位描述来有针对性地写简历。尤其是个人技能和项目（实习）经验部分，要根据岗位要求来写，这样才能增加受邀面试的机会。

举个例子：好友给你推荐了百度地图部门的一个高级 Web 前端工程师工作，并且把职位描述（JD）发给你了，里面有要求哪些技能，用到哪些技术，还有加分项。那么你写简历就应该思考自己有没有这些技能。如果没有 JD，那么至少你应该知道：地图部门肯定做一些跟地图相关的工作，如果恰巧你之前研究过地图定位，了解 `HTML5 Geolocation` 定位接口，那么你可以在简历里提一下。

- 很多时候我们并不知道简历会被谁看到，也不知道简历会被朋友/猎头投递到什么公司或者职位，那么这样的简历应该是一种「通用简历」。所谓通用简历，应该是与你找的职位和期望的级别相匹配的简历，比如想找大概 T4 水平的 Web 前端工作，那么你就应该在简历体现出来自己的技能能够达到 T4 的水平。不要拿着一两年前的简历去找工作，前端这两年发展速度很快，只靠一两年前简历上面「精通、熟悉」的库和框架，可能已经找不到工作了。

所以，写简历也是个技术活，而且是一个辛苦活！不要用千篇一律的模板！

#简历是面试时「点菜」用的菜单

简历除了是「敲门砖」之外，还是供面试官提问用的「菜单」。面试官会从你简历上面写的技能、项目进行提问。所以简历是候选人「反客为主」的重要工具，这也是笔者一直提到的：不要造假或者描述太出格，而应该实事求是地写简历。简历中的技能和项目都要做好知识点梳理，尽量多地梳理出面试官可能问到的问题，并且想出怎么回答应对，千万不要在简历上自己给自己挖坑。

- 案例：记得有一个候选人，写的工作时间段有问题，简历上写在 2015 年 3 月到 2017 年 4 月在 A 公司工作，但是面试自我介绍的时候说自己在 A 公司工作了一年，这就有可能让面试官认为个人工作经历存在造假可能。不要小看细节！
- 另外简历中不要出现错误的单词拼写，注意单词的大小写，比如jQuery之类

#拿到面试邀请之后做的准备工作

当有公司邀请你去面试的时候，应该针对性地做一些功课。

了解部门和团队

了解部门做的事情，团队用的技术栈，前文提到这部分信息一般从 JD 当中就可以看到，如果 JD 并没有这些信息，那么可以根据面试的部门搜索下，总会找到一些零星的信息，如果实在没有任何信息，就准备岗位需要的通用技术。

了解面试官

通过邀请电话或者面试邀请邮件，可以找到面试官信息。通过这些信息查找面试官技术博客、GitHub 等，了解面试官最近关注的技术和擅长的技术，因为面试官往往会在面试的过程中问自己擅长的技术

#面试中出现的常规问题

对于面试中出现的常规问题要做好准备，比如：介绍下自己，为什么跳槽，面试最后一般会问有什么要问的。

介绍自己

介绍自己时，切忌从自己大学实习一直到最新公司全部毫无侧重地介绍，这些在简历当中都有，最好的方式是在介绍中铺垫自己的技术特长、做的项目，引导面试官问自己准备好的问题。

为什么跳槽

- 这个问题一定要慎重和认真思考，诚实回答。一般这个问题是想评估你入职能够待多长时间，是否能够融入团队。
- 每个人跳槽前肯定想了很多原因，最终才走出这一步，不管现在工作怎样，切忌抱怨，不要吐槽，更不要说和现在领导不和睦之类的话。多从自身发展找原因，可以表达寻找自己心目中的好的技术团队氛围和平台机会，比如：个人遇见了天花板，希望找个更好的发展机会。

#利用脑图来梳理知识点

- 对于统一校招类的面试，要重点梳理前端的所有知识点，校招面试一般是为了做人才储备，所以看的是候选人的可塑性和学习能力；对于社招类面试，则看重的是业务能力和JD 匹配程度，所以要针对性地整理前端知识点，针对性的内容包括：项目用到的技术细节、个人技能部分需要加强或提升的常考知识点。
- 所以，不仅仅简历要针对性地来写，知识点也要根据自己的经历、准备的简历、公司和职位描述来针对性地梳理。每个读者的技术能力和工作经历不同，因而知识点梳理大纲也不同，本小册重点介绍如何梳理自己的面试知识点，并且对一些常考的题目进行解答，起到知识点巩固和讲解的作用。
- 基础知识来自于自己平时的储备，一般对着一本系统的书籍或者自己平时的笔记过一遍即可，但是提到自己做到的项目是没有固定的复习套路的，而且围绕项目可以衍生出来各种问题，都需要了解，项目讲清楚对于候选人也特别重要。基础是固定的，任何人经过一段时间都可以学完的，但是项目经历是实打实的经验。
- 对于项目的复习和准备，笔者建议是列思维导图（脑图），针对自己重点需要讲的项目，列出用到的技术点（知识点），介绍背景、项目上线后的收益以及后续优化点。这是第一层，第二层就是针对技术点（知识点）做各种发散的问题。

#二、一面 1：基础知识点与高频考题解析

`JavaScript` 是 `ECMAScript` 规范的一种实现，本小节重点梳理下 `ECMAScript` 中的常考知识点，然后就一些容易出现的题目进行解析。

#知识点梳理

- 变量类型
 - JS 的数据类型分类和判断
 - 值类型和引用类型
- 原型与原型链（继承）
 - 原型和原型链定义
 - 继承写法
- 作用域和闭包
 - 执行上下文
 - `this`
 - 闭包是什么
- 异步
 - 同步 vs 异步
 - 异步和单线程
 - 前端异步的场景

新标准的考查

- 箭头函数
- Module
- Class
- Set 和 Map
- Promise

#变量类型

JavaScript 是一种弱类型脚本语言，所谓弱类型指的是定义变量时，不需要什么类型，在程序运行过程中会自动判断类型。

ECMAScript 中定义了 6 种原始类型：

- Boolean
- String
- Number
- Null
- Undefined
- Symbol (ES6 新定义)

注意：原始类型不包含 Object。

题目：类型判断用到哪些方法？

#typeof

typeof xxx 得到的值有以下几种类型：`undefined` `boolean` `number` `string` `object` `function`、`symbol`，比较简单，不再一一演示了。这里需要注意的有三点：

- `typeof null` 结果是 `object`，实际这是 `typeof` 的一个 bug，null 是原始值，非引用类型
- `typeof [1, 2]` 结果是 `object`，结果中没有 `array` 这一项，引用类型除了 `function` 其他的全部都是 `object`
- `typeof Symbol()` 用 `typeof` 获取 `symbol` 类型的值得到的是 `symbol`，这是 ES6 新增的知识点

#instanceof

用于实例和构造函数的对应。例如判断一个变量是否是数组，使用 `typeof` 无法判断，但可以使用 `[1, 2] instanceof Array` 来判断。因为，`[1, 2]` 是数组，它的构造函数就是 `Array`。同理：

```
function Foo(name) {  
    this.name = name  
}  
var foo = new Foo('bar')  
console.log(foo instanceof Foo) // true
```

题目：值类型和引用类型的区别

#值类型 vs 引用类型

除了原始类型，ES 还有引用类型，上文提到的 `typeof` 识别出来的类型中，只有 `object` 和 `function` 是引用类型，其他都是值类型。

根据 JavaScript 中的变量类型传递方式，又分为**值类型**和**引用类型**，值类型变量包括 Boolean、String、Number、Undefined、Null，引用类型包括了 Object 类的所有，如 Date、Array、Function 等。在参数传递方式上，值类型是按值传递，引用类型是按共享传递。

下面通过一个小题目，来看下两者的主要区别，以及实际开发中需要注意的地方。

```
// 值类型
var a = 10
var b = a
b = 20
console.log(a) // 10
console.log(b) // 20
```

上述代码中，`a` `b` 都是值类型，两者分别修改赋值，相互之间没有任何影响。再看引用类型的例子：

```
// 引用类型
var a = {x: 10, y: 20}
var b = a
b.x = 100
b.y = 200
console.log(a) // {x: 100, y: 200}
console.log(b) // {x: 100, y: 200}
```

上述代码中，`a` `b` 都是引用类型。在执行了`b = a` 之后，修改`b` 的属性值，`a` 的也跟着变化。因为`a` 和 `b` 都是引用类型，指向了同一个内存地址，即两者引用的是同一个值，因此`b` 修改属性时，`a` 的值随之改动。

再借助题目进一步讲解一下。

说出下面代码的执行结果，并分析其原因。

```
function foo(a){
  a = a * 10;
}
function bar(b){
  b.value = 'new';
}
var a = 1;
var b = {value: 'old'};
foo(a);
bar(b);
console.log(a); // 1
console.log(b); // value: new
```

通过代码执行，会发现：

- `a` 的值没有发生改变
- 而 `b` 的值发生了改变

这就是因为 `Number` 类型的 `a` 是按值传递的，而 `Object` 类型的 `b` 是按共享传递的。

JS 中这种设计的原因是：按值传递的类型，复制一份存入栈内存，这类类型一般不占用太多内存，而且按值传递保证了其访问速度。按共享传递的类型，是复制其引用，而不是整个复制其值（C 语言中的指针），保证过大的对象等不会因为不停复制内容而造成内存的浪费。

引用类型经常会在代码中按照下面的写法使用，或者说容易不知不觉中造成错误！

```
var obj = {
  a: 1,
  b: [1, 2, 3]
}
var a = obj.a
var b = obj.b
a = 2
b.push(4)
console.log(obj, a, b)
```

虽然 `obj` 本身是个引用类型的变量（对象），但是内部的 `a` 和 `b` 一个是值类型一个是引用类型，`a` 的赋值不会改变 `obj.a`，但是 `b` 的操作却会反映到 `obj` 对象上。

#原型和原型链

JavaScript 是基于原型的语言，原型理解起来非常简单，但却特别重要，下面还是通过题目来理解下 JavaScript 的原型概念。

题目：如何理解 JavaScript 的原型

对于这个问题，可以从下面这几个要点来理解和回答，**下面几条必须记住并且理解**

- 所有的引用类型（数组、对象、函数），都具有对象特性，即可自由扩展属性（`null` 除外）
- 所有的引用类型（数组、对象、函数），都有一个 `__proto__` 属性，属性值是一个普通的对象
- 所有的函数，都有一个 `prototype` 属性，属性值也是一个普通的对象
- 所有的引用类型（数组、对象、函数），`__proto__` 属性值指向它的构造函数的 `prototype` 属性值

通过代码解释一下，大家可自行运行以下代码，看结果。

```
// 要点一：自由扩展属性
var obj = {}; obj.a = 100;
var arr = []; arr.a = 100;
function fn () {}
fn.a = 100;

// 要点二：__proto__
console.log(obj.__proto__);
console.log(arr.__proto__);
console.log(fn.__proto__);

// 要点三：函数有 prototype
console.log(fn.prototype)

// 要点四：引用类型的 __proto__ 属性值指向它的构造函数的 prototype 属性值
console.log(obj.__proto__ === Object.prototype)
```

1. 原型

先写一个简单的代码示例。

```
// 构造函数
function Foo(name, age) {
  this.name = name
```

```
}

Foo.prototype.alertName = function () {
    alert(this.name)
}

// 创建示例
var f = new Foo('zhangsan')
f.printName = function () {
    console.log(this.name)
}

// 测试
f.printName()
f.alertName()
```

执行 `printName` 时很好理解，但是执行 `alertName` 时发生了什么？这里再记住一个重点 **当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__`（即它的构造函数的 `prototype`）中寻找**，因此 `f.alertName` 就会找到 `Foo.prototype.alertName`。

那么如何判断这个属性是不是对象本身的属性呢？使用 `hasOwnProperty`，常用的地方是遍历一个对象的时候。

```
var item
for (item in f) {
    // 高级浏览器已经在 for in 中屏蔽了来自原型的属性，但是这里建议大家还是加上这个判断，保证
    // 程序的健壮性
    if (f.hasOwnProperty(item)) {
        console.log(item)
    }
}
```

题目：如何理解 JS 的原型链

2. 原型链

还是接着上面的示例，如果执行 `f.toString()` 时，又发生了什么？

```
// 省略 N 行

// 测试
f.printName()
f.alertName()
f.toString()
```

因为 `f` 本身没有 `toString()`，并且 `f.__proto__`（即 `Foo.prototype`）中也没有 `toString`。这个问题还是得拿出刚才那句话——**当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__`（即它的构造函数的 `prototype`）中寻找**。

如果在 `f.__proto__` 中没有找到 `toString`，那么就继续去 `f.__proto__.__proto__` 中寻找，因为 `f.__proto__` 就是一个普通的对象而已嘛！

- `f.__proto__` 即 `Foo.prototype`，没有找到 `toString`，继续往上找
- `f.__proto__.__proto__` 即 `Foo.prototype.__proto__`。`Foo.prototype` 就是一个普通的对象，因此 `Foo.prototype.__proto__` 就是 `Object.prototype`，在这里可以找到 `toString`
- 因此 `f.toString` 最终对应到了 `Object.prototype.toString`

```
这样一直往上找，你会发现是一个链式的结构，所以叫做“原型链”。如果一直找到最上层都没有找到，那么就宣告失败，返回 undefined。最上层是什么——Object.prototype.__proto__  
==== null
```

#原型链中的 this

所有从原型或更高级原型中得到、执行的方法，其中的 `this` 在执行时，就指向了当前这个触发事件执行的对象。因此 `printName` 和 `alertName` 中的 `this` 都是 `f`。

#作用域和闭包

作用域和闭包是前端面试中，最可能考查的知识点。例如下面的题目：

题目：现在有个 HTML 片段，要求编写代码，点击编号为几的链接就 `alert` 弹出其编号

```
<ul>
    <li>编号1，点击我请弹出1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
```

一般不知道这个题目用闭包的话，会写出下面的代码：

```
var list = document.getElementsByTagName('li');
for (var i = 0; i < list.length; i++) {
    list[i].addEventListener('click', function(){
        alert(i + 1)
    }, true)
}
```

实际上执行才会发现始终弹出的是 6，这时候就应该通过闭包来解决：

```
var list = document.getElementsByTagName('li');
for (var i = 0; i < list.length; i++) {
    list[i].addEventListener('click', function(i){
        return function(){
            alert(i + 1)
        }
    }(i), true)
}
```

要理解闭包，就需要我们从「执行上下文」开始讲起。

#执行上下文

先讲一个关于 **变量提升** 的知识点，面试中可能会遇见下面的问题，很多候选人都回答错误：

题目：说出下面执行的结果（这里笔者直接注释输出了）

```

console.log(a) // undefined
var a = 100

fn('zhangsan') // 'zhangsan' 20
function fn(name) {
    age = 20
    console.log(name, age)
    var age
}

console.log(b); // 这里报错
// Uncaught ReferenceError: b is not defined
b = 100;

```

- 在一段JS脚本（即一个`<script>`标签中）执行之前，要先解析代码（所以说JS是解释执行的脚本语言），解析的时候会先创建一个**全局执行上下文**环境，先把代码中即将执行的（内部函数的不算，因为你不知道函数何时执行）变量、函数声明都拿出来。变量先暂时赋值为`undefined`，函数则先声明好可使用。这一步做完了，然后再开始正式执行程序。再次强调，这是在代码执行之前才开始的工作。
- 我们来看下上面的面试小题目，为什么`a`是`undefined`，而`b`却报错了，实际JS在代码执行之前，要「全文解析」，发现`var a`，知道有个`a`的变量，存入了执行上下文，而`b`没有找到`var`关键字，这时候没有在执行上下文提前「占位」，所以代码执行的时候，提前报到的`a`是有记录的，只不过值暂时还没有赋值，即为`undefined`，而`b`在执行上下文没有找到，自然会报错（没有找到`b`的引用）。
- 另外，一个函数在执行之前，也会创建一个**函数执行上下文**环境，跟**全局上下文**差不多，不过**函数执行上下文**中会多出`this` `arguments`和函数的参数。参数和`arguments`好理解，这里的`this`咱们需要专门讲解。

总结一下：

- 范围：一段`<script>`、js文件或者一个函数
- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明，`this`, `arguments`

#this

先搞明白一个很重要的概念——`this`的值是在执行的时候才能确认，**定义的时候不能确认！**为什么呢——因为`this`是执行上下文环境的一部分，而执行上下文需要在代码执行之前确定，而不是定义的时候。看如下例子

```

var a = {
    name: 'A',
    fn: function () {
        console.log(this.name)
    }
}
a.fn() // this === a
a.fn.call({name: 'B'}) // this === {name: 'B'}
var fn1 = a.fn
fn1() // this === window

```

`this`执行会有不同，主要集中在这几个场景中

- 作为构造函数执行，构造函数中

- 作为对象属性执行，上述代码中 `a.fn()`
- 作为普通函数执行，上述代码中 `fn1()`
- 用于 `call` `apply` `bind`，上述代码中 `a.fn.call({name: 'B'})`

下面再来讲解下什么是作用域和作用域链，作用域链和作用域也是常考的题目。

题目：如何理解 JS 的作用域和作用域链

#作用域

ES6 之前 JS 没有块级作用域。例如

```
if (true) {
  var name = 'zhangsan'
}
console.log(name)
```

从上面的例子可以体会到作用域的概念，作用域就是一个独立的地盘，让变量不会外泄、暴露出去。上面的 `name` 就被暴露出去了，因此，**JS 没有块级作用域，只有全局作用域和函数作用域**。

```
var a = 100
function fn() {
  var a = 200
  console.log('fn', a)
}
console.log('global', a)
fn()
```

全局作用域就是最外层的作用域，如果我们写了很多行 JS 代码，变量定义都没有用函数包括，那么它们就全部都在全局作用域中。这样的坏处就是很容易撞车、冲突。

```
// 张三写的代码中
var data = {a: 100}

// 李四写的代码中
var data = {x: true}
```

这就是为何 jQuery、Zepto 等库的源码，所有的代码都会放在 `(function(){...})()` 中。因为放在里面的所有变量，都不会被外泄和暴露，不会污染到外面，不会对其他的库或者 JS 脚本造成影响。这是函数作用域的一个体现。

附：ES6 中开始加入了块级作用域，使用 `let` 定义变量即可，如下：

```
if (true) {
  let name = 'zhangsan'
}
console.log(name) // 报错，因为let定义的name是在if这个块级作用域
```

#作用域链

先认识一下什么叫做 **自由变量**。如下代码中，`console.log(a)` 要得到 `a` 变量，但是在当前的作用域中没有定义 `a`（可对比一下 `b`）。当前作用域没有定义的变量，这成为 **自由变量**。自由变量如何得到——向父级作用域寻找。

```
var a = 100
function fn() {
  var b = 200
  console.log(a)
  console.log(b)
}
fn()
```

如果父级也没呢？再一层一层向上寻找，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是**作用域链**。

```
var a = 100
function F1() {
  var b = 200
  function F2() {
    var c = 300
    console.log(a) // 自由变量，顺作用域链向父作用域找
    console.log(b) // 自由变量，顺作用域链向父作用域找
    console.log(c) // 本作用域的变量
  }
  F2()
}
F1()
```

#闭包

讲完这些内容，我们再来看一个例子，通过例子来理解闭包。

```
function F1() {
  var a = 100
  return function () {
    console.log(a)
  }
}
var f1 = F1()
var a = 200
f1()
```

自由变量将从作用域链中去寻找，但是**依据的是函数定义时的作用域链，而不是函数执行时**，以上这个例子就是闭包。闭包主要有两个应用场景：

- **函数作为返回值**，上面的例子就是
- **函数作为参数传递**，看以下例子

```
function f1() {
    var a = 100
    return function () {
        console.log(a)
    }
}
function f2(f1) {
    var a = 200
    console.log(f1())
}
var f1 = f1()
f2(f1)
```

至此，对应着「作用域和闭包」这部分一开始的点击弹出 `alert` 的代码再看闭包，就很好理解了。

#异步

异步和同步也是面试中常考的内容，下面笔者来讲解下同步和异步的区别。

1. 同步 vs 异步

先看下面的 demo，根据程序阅读起来表达的意思，应该是先打印 `100`，1秒钟之后打印 `200`，最后打印 `300`。但是实际运行根本不是那么回事。

```
console.log(100)
setTimeout(function () {
    console.log(200)
}, 1000)
console.log(300)
```

再对比以下程序。先打印 `100`，再弹出 `200`（等待用户确认），最后打印 `300`。这个运行效果就符合预期要求。

```
console.log(100)
alert(200) // 1秒钟之后点击确认
console.log(300)
```

这俩到底有何区别？—— 第一个示例中间的步骤根本没有阻塞接下来程序的运行，而第二个示例却阻塞了后面程序的运行。前面这种表现就叫做 **异步**（后面这个叫做 **同步**），即**不会阻塞后面程序的运行**。

2. 异步和单线程

JS 需要异步的根本原因是 **JS 是单线程运行的**，即在同一时间只能做一件事，不能“一心二用”。

一个 Ajax 请求由于网络比较慢，请求需要 5 秒钟。如果是同步，这 5 秒钟页面就卡死在这里啥也干不了了。异步的话，就好很多了，5 秒等待就等待了，其他事情不耽误做，至于那 5 秒钟等待是网速太慢，不是因为 JS 的原因。

讲到单线程，我们再来看个真题：

题目：讲解下面代码的执行过程和结果

```
var a = true;
setTimeout(function(){
    a = false;
}, 100)
while(a){
    console.log('while执行了')
}
```

这是一个很有迷惑性的题目，不少候选人认为`100ms`之后，由于`a`变成了`false`，所以`while`就中止了，实际不是这样，因为JS是单线程的，所以进入`while`循环之后，没有「时间」（线程）去跑定时器了，所以这个代码跑起来是个死循环！

3. 前端异步的场景

- 定时 `setTimeout` `setInterval`
- 网络请求，如 `Ajax` `` 加载

Ajax 代码示例

```
console.log('start')
$.get('./data1.json', function (data1) {
    console.log(data1)
})
console.log('end')
```

`img` 代码示例（常用于打点统计）

```
console.log('start')
var img = document.createElement('img')
// 或者 img = new Image()
img.onload = function () {
    console.log('loaded')
    img.onload = null
}
img.src = '/xxx.png'
console.log('end')
```

#ES6/7 新标准的考查

题目：ES6 箭头函数中的 `this` 和普通函数中的有什么不同

1. 箭头函数

箭头函数是 ES6 中新的函数定义形式，`function name(arg1, arg2) {...}` 可以使用`(arg1, arg2) => {...}` 来定义。示例如下：

```
// JS 普通函数
var arr = [1, 2, 3]
arr.map(function (item) {
  console.log(index)
  return item + 1
})

// ES6 箭头函数
const arr = [1, 2, 3]
arr.map((item, index) => {
  console.log(index)
  return item + 1
})
```

箭头函数存在的意义，第一写起来更加简洁，第二可以解决 ES6 之前函数执行中 `this` 是全局变量的问题，看如下代码

```
function fn() {
  console.log('real', this) // {a: 100}，该作用域下的 this 的真实的值
  var arr = [1, 2, 3]
  // 普通 JS
  arr.map(function (item) {
    console.log('js', this) // window。普通函数，这里打印出来的是全局变量，令人费解
    return item + 1
  })
  // 箭头函数
  arr.map(item => {
    console.log('es6', this) // {a: 100}。箭头函数，这里打印的就是父作用域的 this
    return item + 1
  })
}
fn.call({a: 100})
```

题目：ES6 模块化如何使用？

2. Module

ES6 中模块化语法更加简洁，直接看示例。

如果只是输出一个唯一的对象，使用 `export default` 即可，代码如下

```
// 创建 util1.js 文件，内容如
export default {
  a: 100
}

// 创建 index.js 文件，内容如
import obj from './util1.js'
console.log(obj)
```

如果想要输出许多个对象，就不能用 `default` 了，且 `import` 时候要加 `{...}`，代码如下

```
// 创建 util2.js 文件，内容如
export function fn1() {
    alert('fn1')
}

export function fn2() {
    alert('fn2')
}

// 创建 index.js 文件，内容如
import { fn1, fn2 } from './util2.js'
fn1()
fn2()
```

题目：ES6 class 和普通构造函数的区别

3. class

`class` 其实一直是 JS 的关键字（保留字），但是一直没有正式使用，直到 ES6。ES6 的 `class` 就是取代之前构造函数初始化对象的形式，从语法上更加符合面向对象的写法。例如：

JS 构造函数的写法

```
function MathHandle(x, y) {
    this.x = x;
    this.y = y;
}

MathHandle.prototype.add = function () {
    return this.x + this.y;
};

var m = new MathHandle(1, 2);
console.log(m.add())
```

用 ES6 class 的写法

```
class MathHandle {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    add() {
        return this.x + this.y;
    }
}
const m = new MathHandle(1, 2);
console.log(m.add())
```

注意以下几点，全都是关于 `class` 语法的：

- `class` 是一种新的语法形式，是 `class Name { ... }` 这种形式，和函数的写法完全不一样
- 两者对比，构造函数函数体的内容要放在 `class` 中的 `constructor` 函数中，`constructor` 即构造器，初始化实例时默认执行
- `class` 中函数的写法是 `add() { ... }` 这种形式，并没有 `function` 关键字

使用 class 来实现继承就更加简单了，至少比构造函数实现继承简单很多。看下面例子

JS 构造函数实现继承

```
// 动物
function Animal() {
    this.eat = function () {
        console.log('animal eat')
    }
}
// 狗
function Dog() {
    this.bark = function () {
        console.log('dog bark')
    }
}
Dog.prototype = new Animal()
// 哈士奇
var hashiqi = new Dog()
```

ES6 class 实现继承

```
class Animal {
    constructor(name) {
        this.name = name
    }
    eat() {
        console.log(` ${this.name} eat`)
    }
}

class Dog extends Animal {
    constructor(name) {
        super(name)
        this.name = name
    }
    say() {
        console.log(` ${this.name} say`)
    }
}
const dog = new Dog('哈士奇')
dog.say()
dog.eat()
```

注意以下两点：

- 使用 extends 即可实现继承，更加符合经典面向对象语言的写法，如 Java
- 子类的 constructor 一定要执行 super()，以调用父类的 constructor

题目：ES6 中新增的数据类型有哪些？

#Set 和 Map

Set 和 Map 都是 ES6 中新增的数据结构，是对当前 JS 数组和对象这两种重要数据结构的扩展。由于是新增的数据结构，目前尚未被大规模使用，但是作为前端程序员，提前了解是必须做到的。先总结一下两者最关键的地方：

- Set 类似于数组，但数组可以允许元素重复，Set 不允许元素重复
- Map 类似于对象，但普通对象的 key 必须是字符串或者数字，而 Map 的 key 可以是任何数据类型

Set

`Set` 实例不允许元素有重复，可以通过以下示例证明。可以通过一个数组初始化一个 Set 实例，或者通过 `add` 添加元素，元素不能重复，重复的会被忽略。

```
// 例1
const set = new Set([1, 2, 3, 4, 4]);
console.log(set) // Set(4) {1, 2, 3, 4}

// 例2
const set = new Set();
[2, 3, 5, 4, 5, 8, 8].forEach(item => set.add(item));
for (let item of set) {
  console.log(item);
}
// 2 3 5 4 8
```

Set 实例的属性和方法有

- `size`：获取元素数量。
- `add(value)`：添加元素，返回 Set 实例本身。
- `delete(value)`：删除元素，返回一个布尔值，表示删除是否成功。
- `has(value)`：返回一个布尔值，表示该值是否是 Set 实例的元素。
- `clear()`：清除所有元素，没有返回值。

```
const s = new Set();
s.add(1).add(2).add(2); // 添加元素

s.size // 2

s.has(1) // true
s.has(2) // true
s.has(3) // false

s.delete(2);
s.has(2) // false

s.clear();
console.log(s); // Set(0) {}
```

Set 实例的遍历，可使用如下方法

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。不过由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以 `keys()` 和 `values()` 返回结果一致。
- `entries()`：返回键值对的遍历器。
- `forEach()`：使用回调函数遍历每个成员。

```
let set = new Set(['aaa', 'bbb', 'ccc']);

for (let item of set.keys()) {
    console.log(item);
}

// aaa
// bbb
// ccc

for (let item of set.values()) {
    console.log(item);
}

// aaa
// bbb
// ccc

for (let item of set.entries()) {
    console.log(item);
}

// ["aaa", "aaa"]
// ["bbb", "bbb"]
// ["ccc", "ccc"]

set.forEach((value, key) => console.log(key + ' : ' + value))
// aaa : aaa
// bbb : bbb
// ccc : ccc
```

Map

`Map` 的用法和普通对象基本一致，先看一下它能用非字符串或者数字作为 key 的特性。

```
const map = new Map();
const obj = {p: 'Hello World'};

map.set(obj, 'OK')
map.get(obj) // "OK"

map.has(obj) // true
map.delete(obj) // true
map.has(obj) // false
```

需要使用 `new Map()` 初始化一个实例，下面代码中 `set` `get` `has` `delete` 顾名即可思义（下文也会演示）。其中，`map.set(obj, 'OK')` 就是用对象作为的 key（不光可以是对象，任何数据类型都可以），并且后面通过 `map.get(obj)` 正确获取了。

Map 实例的属性和方法如下：

- `size`：获取成员的数量
- `set`：设置成员 key 和 value
- `get`：获取成员属性值
- `has`：判断成员是否存在
- `delete`：删除成员
- `clear`：清空所有

```
const map = new Map();
map.set('aaa', 100);
map.set('bbb', 200);

map.size // 2

map.get('aaa') // 100

map.has('aaa') // true

map.delete('aaa')
map.has('aaa') // false

map.clear()
```

Map 实例的遍历方法有：

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。
- `entries()`：返回所有成员的遍历器。
- `forEach()`：遍历 Map 的所有成员。

```
const map = new Map();
map.set('aaa', 100);
map.set('bbb', 200);

for (let key of map.keys()) {
  console.log(key);
}
// "aaa"
// "bbb"

for (let value of map.values()) {
  console.log(value);
}
// 100
// 200

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
// aaa 100
// bbb 200

// 或者
for (let [key, value] of map.entries()) {
  console.log(key, value);
}
// aaa 100
// bbb 200
```

#Promise

`Promise` 是 CommonJS 提出来的这一种规范，有多个版本，在 ES6 当中已经纳入规范，原生支持 `Promise` 对象，非 ES6 环境可以用类似 Bluebird、Q 这类库来支持。

`Promise` 可以将回调变成链式调用写法，流程更加清晰，代码更加优雅。

简单归纳下 `Promise`：三个状态、两个过程、一个方法，快速记忆方法：3-2-1

三个状态：`pending`、`fulfilled`、`rejected`

两个过程：

- `pending` → `fulfilled` (`resolve`)
- `pending` → `rejected` (`reject`)

一个方法：`then`

当然还有其他概念，如 `catch`、`Promise.all/race`，这里就不展开了。

关于 ES6/7 的考查内容还有很多，本小节就不逐一介绍了，如果想继续深入学习，可以在线看《[ES6入门](#)》。

#三、一面 2：JS-Web-API 知识点与高频考题解析

除 ES 基础之外，Web 前端经常会用到一些跟浏览器相关的 API，接下来我们一起梳理一下。

#知识点梳理

- BOM 操作
- DOM 操作
- 事件绑定
- Ajax
- 存储

#BOM

`BOM`（浏览器对象模型）是浏览器本身的一些信息的设置和获取，例如获取浏览器的宽度、高度，设置让浏览器跳转到哪个地址。

- `navigator`
- `screen`
- `location`
- `history`

这些对象就是一堆非常简单粗暴的 API，没任何技术含量，讲起来一点意思都没有，大家去 MDN 或者 w3school 这种网站一查就都明白了。面试的时候，面试官基本不会出太多这方面的题目，因为只要基础知识过关了，这些 API 即便你记不住，上网一查也都知道了。下面列举一下常用功能的代码示例

获取浏览器特性（即俗称的 UA）然后识别客户端，例如判断是不是 Chrome 浏览器

```
var ua = navigator.userAgent
var isChrome = ua.indexOf('Chrome')
console.log(isChrome)
```

获取屏幕的宽度和高度

```
console.log(screen.width)
console.log(screen.height)
```

获取网址、协议、`path`、参数、`hash` 等

```
// 例如当前网址是 https://juejin.im/timeline/frontend?a=10&b=10#some
console.log(location.href) // https://juejin.im/timeline/frontend?
a=10&b=10#some
console.log(location.protocol) // https:
console.log(location.pathname) // /timeline/frontend
console.log(location.search) // ?a=10&b=10
console.log(location.hash) // #some
```

另外，还有调用浏览器的前进、后退功能等

```
history.back()
history.forward()
```

#DOM

题目：`DOM` 和 `HTML` 区别和联系

1. 什么是 DOM

讲 `DOM` 先从 `HTML` 讲起，讲 `HTML` 先从 `XML` 讲起。`XML` 是一种可扩展的标记语言，所谓可扩展就是它可以描述任何结构化的数据，它是一棵树！

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
  <other>
    <a></a>
    <b></b>
  </other>
</note>
```

`HTML` 是一个有既定标签标准的 `XML` 格式，标签的名字、层级关系和属性，都被标准化（否则浏览器无法解析）。同样，它也是一棵树。

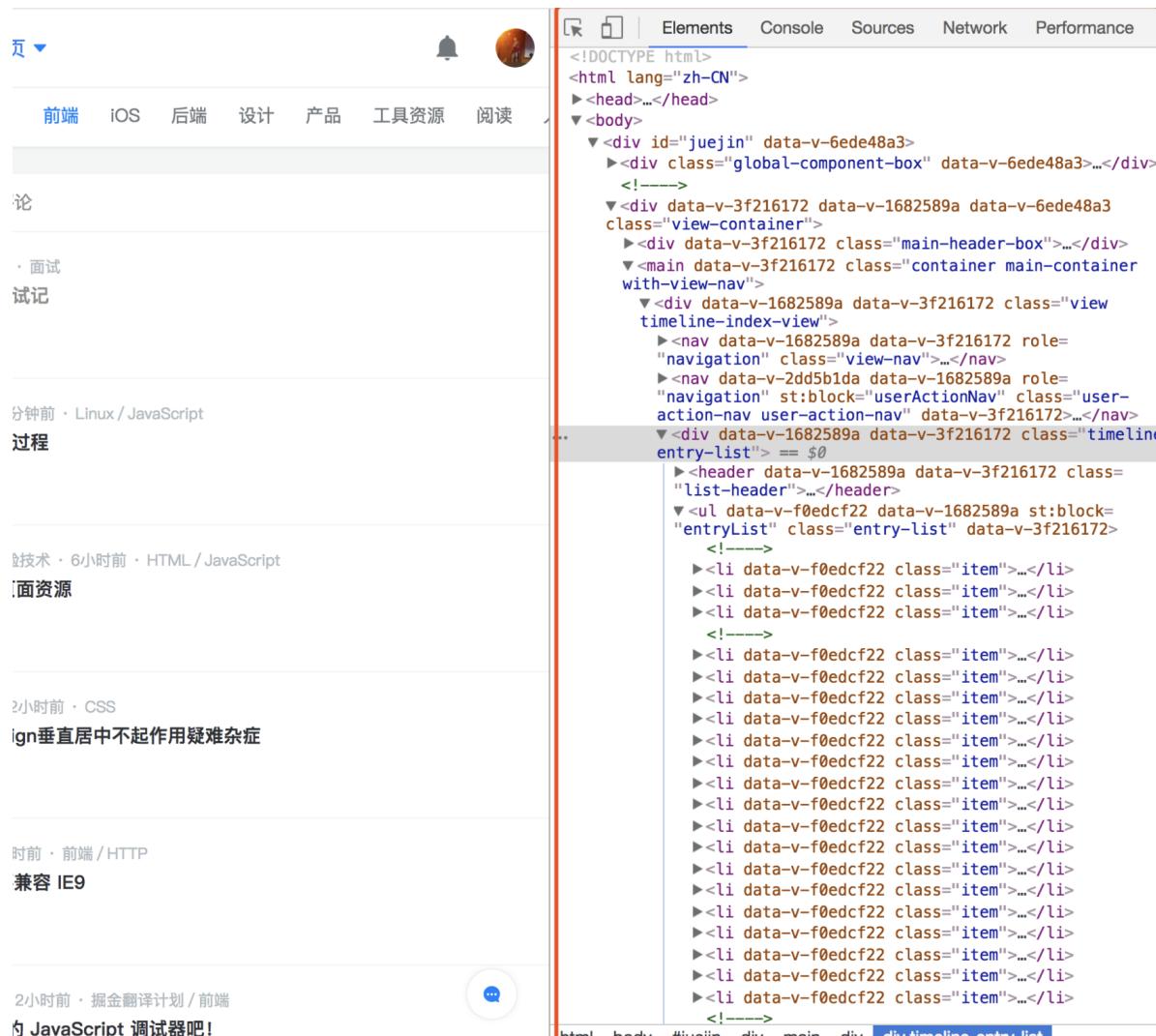
```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <div>
        <p>this is p</p>
    </div>
</body>
</html>

```

我们开发完的 `HTML` 代码会保存到一个文档中（一般以 `.html` 或者 `.htm` 结尾），文档放在服务器上，浏览器请求服务器，这个文档被返回。因此，最终浏览器拿到的是一个文档而已，文档的内容就是 `HTML` 格式的代码。

- 但是浏览器要把这个文档中的 `HTML` 按照标准渲染成一个页面，此时浏览器就需要将这堆代码处理成自己能理解的东西，也得处理成 `JS` 能理解的东西，因为还得允许 `JS` 修改页面内容呢。
- 基于以上需求，浏览器就需要把 `HTML` 变成 `DOM`，`HTML` 是一棵树，`DOM` 也是一棵树。对 `DOM` 的理解，可以暂时先抛开浏览器的内部因素，先从 `JS` 着手，即可以认为 `DOM` 就是 `JS` 能识别的 `HTML` 结构，一个普通的 `JS` 对象或者数组。



2. 获取 DOM 节点

最常用的 `DOM API` 就是获取节点，其中常用的获取方法如下面代码示例：

```
// 通过 id 获取
var div1 = document.getElementById('div1') // 元素

// 通过 tagname 获取
var divList = document.getElementsByTagName('div') // 集合
console.log(divList.length)
console.log(divList[0])

// 通过 class 获取
var containerList = document.getElementsByClassName('container') // 集合

// 通过 CSS 选择器获取
var pList = document.querySelectorAll('p') // 集合
```

题目：`property` 和 `attribute` 的区别是什么？

3. property

`DOM` 节点就是一个 JS 对象，它符合之前讲述的对象的特征——可扩展属性，因为 `DOM` 节点本质上也是一个 JS 对象。因此，如下代码所示，`p` 可以有 `style` 属性，有 `className` `nodeName` `nodeType` 属性。注意，**这些都是 JS 范畴的属性，符合 JS 语法规则的。**

```
var pList = document.querySelectorAll('p')
var p = pList[0]
console.log(p.style.width) // 获取样式
p.style.width = '100px' // 修改样式
console.log(p.className) // 获取 class
p.className = 'p1' // 修改 class

// 获取 nodeName 和 nodeType
console.log(p.nodeName)
console.log(p.nodeType)
```

4. attribute

- `property` 的获取和修改，是直接改变 JS 对象，而 `attribute` 是直接改变 HTML 的属性，两种有很大的区别。- `attribute` 就是对 HTML 属性的 `get` 和 `set`，和 `DOM` 节点的 JS 范畴的 `property` 没有关系。

```
var pList = document.querySelectorAll('p')
var p = pList[0]
p.getAttribute('data-name')
p.setAttribute('data-name', 'juejin')
p.getAttribute('style')
p.setAttribute('style', 'font-size:30px;')
```

而且，`get` 和 `set` `attribute` 时，还会触发 `DOM` 的查询或者重绘、重排，频繁操作会影响页面性能。

题目：DOM 操作的基本 API 有哪些？

5. DOM 树操作

新增节点

```
var div1 = document.getElementById('div1')

// 添加新节点
var p1 = document.createElement('p')
p1.innerHTML = 'this is p1'
div1.appendChild(p1) // 添加新创建的元素

// 移动已有节点。注意，这里是“移动”，并不是拷贝
var p2 = document.getElementById('p2')
div1.appendChild(p2)
```

获取父元素

```
var div1 = document.getElementById('div1')
var parent = div1.parentElement
```

获取子元素

```
var div1 = document.getElementById('div1')
var child = div1.childNodes
```

删除节点

```
var div1 = document.getElementById('div1')
var child = div1.childNodes
div1.removeChild(child[0])
```

还有其他操作的API，例如获取前一个节点、获取后一个节点等，但是面试过程中经常考到的就是上面几个。

#事件

1. 事件绑定

普通的事件绑定写法如下：

```
var btn = document.getElementById('btn1')
btn.addEventListener('click', function (event) {
    // event.preventDefault() // 阻止默认行为
    // event.stopPropagation() // 阻止冒泡
    console.log('clicked')
})
```

为了编写简单的事件绑定，可以编写通用的事件绑定函数。这里虽然比较简单，但是会随着后文的讲解，来继续完善和丰富这个函数。

```
// 通用的事件绑定函数
function bindEvent(elem, type, fn) {
    elem.addEventListener(type, fn)
}

var a = document.getElementById('link1')
// 写起来更加简单了
bindEvent(a, 'click', function(e) {
    e.preventDefault() // 阻止默认行为
    alert('clicked')
})
})
```

最后，如果面试被问到 IE 低版本兼容性问题，我劝你果断放弃这份工作机会。现在互联网流量都在 App 上，IE 占比越来越少，再去为 IE 浪费青春不值得，要尽量去做 App 相关的工作。

题目：什么是事件冒泡？

2. 事件冒泡

```
<body>
<div id="div1">
    <p id="p1">激活</p>
    <p id="p2">取消</p>
    <p id="p3">取消</p>
    <p id="p4">取消</p>
</div>
<div id="div2">
    <p id="p5">取消</p>
    <p id="p6">取消</p>
</div>
</body>
```

对于以上 HTML 代码结构，要求点击 p1 时候进入激活状态，点击其他任何 <p> 都取消激活状态，如何实现？代码如下，注意看注释：

```
var body = document.body
bindEvent(body, 'click', function (e) {
    // 所有 p 的点击都会冒泡到 body 上，因为 DOM 结构中 body 是 p 的上级节点，事件会沿着
    // DOM 树向上冒泡
    alert('取消')
})

var p1 = document.getElementById('p1')
bindEvent(p1, 'click', function (e) {
    e.stopPropagation() // 阻止冒泡
    alert('激活')
})
```

如果我们在 p1 div1 body 中都绑定了事件，它是会根据 DOM 的结构来冒泡，从下到上挨个执行的。但是我们使用 e.stopPropagation() 就可以阻止冒泡

题目：如何使用事件代理？有何好处？

3. 事件代理

我们设定一种场景，如下代码，一个 <div> 中包含了若干个 <a>，而且还能继续增加。那如何快捷方便地为所有 <a> 绑定事件呢？

```
<div id="div1">
  <a href="#">a1</a>
  <a href="#">a2</a>
  <a href="#">a3</a>
  <a href="#">a4</a>
</div>
<button>点击增加一个 a 标签</button>
```

这里就会用到事件代理。我们要监听[的事件，但要把具体的事件绑定到

上，然后看事件的触发点是不是。](#)

```
var div1 = document.getElementById('div1')
div1.addEventListener('click', function (e) {
  // e.target 可以监听到触发点击事件的元素是哪一个
  var target = e.target
  if (e.nodeName === 'A') {
    // 点击的是 <a> 元素
    alert(target.innerHTML)
  }
})
```

我们现在完善一下之前写的通用事件绑定函数，加上事件代理。

```
function bindEvent(elem, type, selector, fn) {
  // 这样处理，可接收两种调用方式 bindEvent(div1, 'click', 'a', function () {...})
  和 bindEvent(div1, 'click', function () {...}) 这两种
  if (fn == null) {
    fn = selector
    selector = null
  }

  // 绑定事件
  elem.addEventListener(type, function (e) {
    var target
    if (selector) {
      // 有 selector 说明需要做事件代理
      // 获取触发时间的元素，即 e.target
      target = e.target
      // 看是否符合 selector 这个条件
      if (target.matches(selector)) {
        fn.call(target, e)
      }
    } else {
      // 无 selector，说明不需要事件代理
      fn(e)
    }
  })
}
```

然后这样使用，简单很多。

```
// 使用代理, bindEvent 多一个 'a' 参数
var div1 = document.getElementById('div1')
bindEvent(div1, 'click', 'a', function (e) {
    console.log(this.innerHTML)
})

// 不使用代理
var a = document.getElementById('a1')
bindEvent(div1, 'click', function (e) {
    console.log(a.innerHTML)
})
```

最后，使用代理的优点如下：

- 使代码简洁
- 减少浏览器的内存占用

#Ajax

1. XMLHttpRequest

题目：手写 XMLHttpRequest 不借助任何库

这是很多奇葩的、个性的面试官经常用的手段。这种考查方式存在很多争议，但是你不能完全说它是错误的，毕竟也是考查对最基础知识的掌握情况。

```
var xhr = new XMLHttpRequest()
xhr.onreadystatechange = function () {
    // 这里的函数异步执行，可参考之前 JS 基础中的异步模块
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            alert(xhr.responseText)
        }
    }
}
xhr.open("GET", "/api", false)
xhr.send(null)
```

当然，使用 jQuery、Zepto 或 Fetch 等库来写就更加简单了，这里不再赘述。

2. 状态码说明

上述代码中，有两处状态码需要说明。`xhr.readyState` 是浏览器判断请求过程中各个阶段的，`xhr.status` 是 HTTP 协议中规定的不同结果的返回状态说明。

`xhr.readyState` 的状态码说明：

- 0 代理被创建，但尚未调用 `open()` 方法。
- 1 `open()` 方法已经被调用。
- 2 `send()` 方法已经被调用，并且头部和状态已经可获得。
- 3 下载中，`responseText` 属性已经包含部分数据。
- 4 下载操作已完成

题目：HTTP 协议中，`response` 的状态码，常见的有哪些？

`xhr.status` 即 HTTP 状态码，有 2xx 3xx 4xx 5xx 这几种，比较常用的有以下几种：

- 200 正常
- 3xx
 - 301 永久重定向。如 `http://xxx.com` 这个 GET 请求（最后没有 /），就会被 301 到 `http://xxx.com/`（最后是 /）
 - 302 临时重定向。临时的，不是永久的
 - 304 资源找到但是不符合请求条件，不会返回任何主体。如发送 GET 请求时，`head` 中有 `If-Modified-Since: xxx`（要求返回更新时间是 `xxx` 时间之后的资源），如果此时服务器端资源未更新，则会返回 304，即不符合要求
- 404 找不到资源
- 5xx 服务器端出错了

看完要明白，为何上述代码中要同时满足 `xhr.readyState == 4` 和 `xhr.status == 200`。

3. Fetch API

目前已经有一个获取 HTTP 请求更加方便的 API：`Fetch`，通过 `Fetch` 提供的 `fetch()` 这个全局函数方法可以很简单地发起异步请求，并且支持 `Promise` 的回调。但是 Fetch API 是比较新的 API，具体使用的时候还需要查查 [caniuse](#)，看下其浏览器兼容情况。

看一个简单的例子：

```
fetch('some/api/data.json', {
  method: 'POST', // 请求类型 GET、POST
  headers: {}, // 请求的头信息，形式为 Headers 对象或 ByteString
  body: {}, // 请求发送的数据 blob、BufferSource、FormData、URLSearchParams（get 或 head 方法中不能包含 body）
  mode: '', // 请求的模式，是否跨域等，如 cors、no-cors 或 same-origin
  credentials: '', // cookie 的跨域策略，如 omit、same-origin 或 include
  cache: '', // 请求的 cache 模式：default、no-store、reload、no-cache、force-cache 或 only-if-cached
}).then(function(response) { ... });
```

`Fetch` 支持 `headers` 定义，通过 `headers` 自定义可以方便地实现多种请求方法（PUT、GET、POST 等）、请求头（包括跨域）和 `cache` 策略等；除此之外还支持 `response`（返回数据）多种类型，比如支持二进制文件、字符串和 `formData` 等。

#跨域

题目：如何实现跨域？

浏览器中有 **同源策略**，即一个域下的页面中，无法通过 Ajax 获取到其他域的接口。例如有一个接口 `http://m.test.com/course/ajaxcourserecom?cid=459`，你自己的一个页面 `http://www.yourname.com/page1.html` 中的 Ajax 无法获取这个接口。这正是命中了“同源策略”。如果浏览器哪些地方忽略了同源策略，那就是浏览器的安全漏洞，需要紧急修复。

url 哪些地方不同算作跨域？

- 协议
- 域名
- 端口

但是 HTML 中几个标签能逃避过同源策略——`<script src="xxx">`、``、`<link href="xxxx">`，这三个标签的 `src/href` 可以加载其他域的资源，不受同源策略限制。

因此，这使得这三个标签可以做一些特殊的事情

- `` 可以做打点统计，因为统计方并不一定是同域的，在讲解 JS 基础知识异步的时候有过代码示例。除了能跨域之外，`` 几乎没有浏览器兼容问题，它是一个非常古老的标签。
- `<script>` 和 `<link>` 可以使用 CDN，CDN 基本都是其他域的链接。
- 另外 `<script>` 还可以实现 JSONP，能获取其他域接口的信息，接下来马上讲解。

但是请注意，所有的跨域请求方式，最终都需要信息提供方来做出相应的支持和改动，也就是要经过信息提供方的同意才行，否则接收方是无法得到它们的信息的，浏览器是不允许的。

1. 解决跨域 - JSONP

首先，有一个概念你要明白，例如访问 `http://coding.m.test.com/classindex.html` 的时候，服务器端就一定有一个 `classindex.html` 文件吗？——不一定，服务器可以拿到这个请求，动态生成一个文件，然后返回。同理，`<script src="http://coding.m.test.com/api.js">` 也不一定加载一个服务器端的静态文件，服务器也可以动态生成文件并返回。OK，接下来正式开始。

例如我们的网站和掘金网，肯定不是一个域。我们需要掘金网提供一个接口，供我们来获取。首先，我们在自己的页面这样定义

```
<script>
window.callback = function (data) {
    // 这是我们跨域得到信息
    console.log(data)
}
</script>
```

然后网给我提供了一个 `http://coding.m.test.com/api.js`，内容如下（之前说过，服务器可动态生成内容）

```
callback({x:100, y:200})
```

最后我们在页面中加入 `<script src="http://coding.m.test.com/api.js"></script>`，那么这个js加载之后，就会执行内容，我们就得到内容了。

2. 解决跨域 - 服务器端设置 http header

这是需要在服务器端设置的，作为前端工程师我们不用详细掌握，但是要知道有这么个解决方案。而且，现在推崇的跨域解决方案是这一种，比 JSONP 简单许多。

```
response.setHeader("Access-Control-Allow-Origin", "http://m.juejin.com/"); // 第二个参数填写允许跨域的域名称，不建议直接写 "*"
response.setHeader("Access-Control-Allow-Headers", "X-Requested-With");
response.setHeader("Access-Control-Allow-Methods",
"PUT, POST, GET, DELETE, OPTIONS");

// 接收跨域的cookie
response.setHeader("Access-Control-Allow-Credentials", "true");
```

#存储

题目：`cookie` 和 `localStorage` 有何区别？

1. cookie

`cookie` 本身不是用来做服务器端存储的（计算机领域有很多这种“狗拿耗子”的例子，例如 CSS 中的 `float`），它是设计用来在服务器和客户端进行信息传递的，因此我们的每个 HTTP 请求都带着 `cookie`。但是 `cookie` 也具备浏览器端存储的能力（例如记住用户名和密码），因此就被开发者用上了。

使用起来也非常简单，`document.cookie =` 即可。

但是 `cookie` 有它致命的缺点：

- 存储量太小，只有 `4KB`
- 所有 HTTP 请求都带着，会影响获取资源的效率
- `API` 简单，需要封装才能用

2. localStorage 和 sessionStorage

后来，HTML5 标准就带来了 `sessionStorage` 和 `localStorage`，先拿 `localStorage` 来说，它是专门为了浏览器端缓存而设计的。其优点有：

- 存储量增大到 `5MB`
- 不会带到 HTTP 请求中
- `API` 适用于数据存储 `localStorage.setItem(key, value)` `localStorage.getItem(key)`

`sessionStorage` 的区别就在于它是根据 session 过去时间而实现，而 `localStorage` 会永久有效，应用场景不同。例如，一些需要及时失效的重要信息放在 `sessionStorage` 中，一些不重要但是不经常设置的信息，放在 `localStorage` 中。

另外告诉大家一个小技巧，针对 `localStorage.setItem`，使用时尽量加入到 `try-catch` 中，某些浏览器是禁用这个 `API` 的，要注意。

#小结

本小节总结了 W3C 标准中 `Web-API` 部分，面试中常考的知识点，这些也是日常开发中最常用的 API 和知识。

#四、一面 3：CSS-HTML 知识点与高频考题解析

`CSS` 和 `HTML` 是网页开发中布局相关的组成部分，涉及的内容比较多和杂乱，本小节重点介绍下常考的知识点。

#知识点梳理

- 选择器的权重和优先级
- 盒模型
 - 盒子大小计算
 - `margin` 的重叠计算
- 浮动

`float`

- 浮动布局概念
- 清理浮动
- 定位

position

- 文档流概念
- 定位分类
- `fixed` 定位特点
- 绝对定位计算方式
- `flex` 布局
- 如何实现居中对齐?
- 理解语义化
- CSS3 动画
- 重绘和回流

#选择器的权重和优先级

CSS 选择器有很多，不同的选择器的权重和优先级不一样，对于一个元素，如果存在多个选择器，那么就需要根据权重来计算其优先级。

权重分为四级，分别是：

1. 代表内联样式，如 `style="xxx"`，权值为 1000；
2. 代表 ID 选择器，如 `#content`，权值为 100；
3. 代表类、伪类和属性选择器，如 `.content`、`:hover`、`[attribute]`，权值为 10；
4. 代表元素选择器和伪元素选择器，如 `div`、`p`，权值为 1。

需要注意的是：通用选择器（`*`）、子选择器（`>`）和相邻同胞选择器（`+`）并不在这四个等级中，所以他们的权值都为 0。权重值大的选择器其优先级也高，相同权重的优先级又遵循后定义覆盖前面定义的情况。

#盒模型

1. 什么是“盒子”

初学 CSS 的朋友，一开始学 CSS 基础知识的时候一定学过 `padding` `border` 和 `margin`，即内边距、边框和外边距。它们三者就构成了一个“盒子”。就像我们收到的快递，本来买了一部小小的手机，收到的却是那么大一个盒子。因为手机白色的包装盒和手机机器之间有间隔层（内边距），手机白色盒子有厚度，虽然很薄（边框），盒子和快递箱子之间还有一层泡沫板（外边距）。这就是一个典型的盒子。

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》。文章言简意赅的介绍了浏览器的工作过程，web前端

如上图，真正的内容就是这些文字，文字外围有 10px 的内边距，5px 的边框，10px 的外边距。看到盒子了吧？

题目：盒子模型的宽度如何计算

2. 固定宽度的盒子

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px;">  
    文章言简意赅的介绍了浏览器的工作过程，web前端  
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》。文章言简意赅的介绍了浏览器的工作过程，web前端

300px

如上图，得到网页效果之后，我们可以用截图工具来量一下文字内容的宽度。发现，文字内容的宽度刚好是 300px，也就是我们设置的宽度。

因此，在盒子模型中，我们设置的宽度都是内容宽度，不是整个盒子的宽度。而整个盒子的宽度是：(内容宽度 + border 宽度 + padding 宽度 + margin 宽度) 之和。这样我们改四个中的其中一个，都会导致盒子宽度的改变。这对我们来说不友好。

没关系，这个东西不友好早就有人发现了，而且已经解决，下文再说。

3. 充满父容器的盒子

默认情况下，`div` 是 `display:block`，宽度会充满整个父容器。如下图：

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px;">  
    之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》，  
    文章言简意赅的介绍了浏览器的工作过程，web前端  
    之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》，  
    文章言简意赅的介绍了浏览器的工作过程，web前端  
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》。文章言简意赅的介绍了浏览器的工作过程，web前端 之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》。文章言简意赅的介绍了浏览器的工作过程，web前端

- 但是别忘记，这个 div 是个盒子模型，它的整个宽度包括（内容宽度 + border 宽度 + padding 宽度 + margin 宽度），整个的宽度充满父容器。
- 问题就在这里。如果父容器宽度不变，我们手动增大 margin、border 或 padding 其中一项的宽度值，都会导致内容宽度的减少。极端情况下，如果内容的宽度压缩到不能再压缩了（例如一个字的宽度），那么浏览器会强迫增加父容器的宽度。这可不是我们想要看到的。

4. 包裹内容的盒子

这种情况下比较简单，内容的宽度按照内容计算，盒子的宽度将在内容宽度的基础上再增加（padding 宽度 + border 宽度 + margin 宽度）之和。

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px;">  
    之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》  
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》

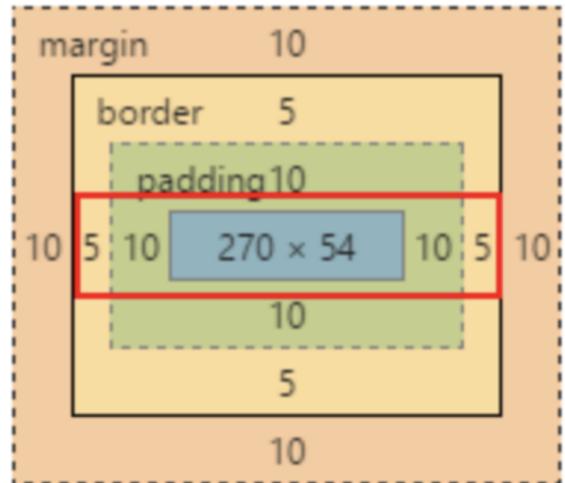
5. box-sizing:border-box

前面提到，为盒子模型设置宽度，结果只是设置了内容的宽度，这个不合理。如何解决这一问题？答案就是为盒子指定样式：box-sizing:border-box。

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px; box-sizing:border-box;">  
    之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》  
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》

```
element.style {  
    padding: 10px;  
    border: 5px solid blue;  
    margin: 10px;  
    width: 300px;  
    box-sizing: border-box;  
}
```



上图中，为 `div` 设置了 `box-sizing: border-box` 之后，300px 的宽度是内容 + `padding` + 边框的宽度（不包括 `margin`），这样就比较符合我们的实际要求了。建议大家在为系统写 CSS 时候，第一个样式是：

```
* {  
    box-sizing: border-box;  
}
```

大名鼎鼎的 Bootstrap 也把 `box-sizing: border-box` 加入到它的 `*` 选择器中，我们为什么不这样做呢？

6. 纵向 margin 重叠

这里提到 `margin`，就不得不提一下 `margin` 的这一特性——纵向重叠。如 `<p>` 的纵向 `margin` 是 16px，那么两个 `<p>` 之间纵向的距离是多少？——按常理来说应该是 $16 + 16 = 32\text{px}$ ，但是答案仍然是 16px。因为纵向的 `margin` 是会重叠的，如果两者不一样大的话，大的会把小的“吃掉”。

#浮动 float

`float` 用于网页布局比较多，使用起来也比较简单，这里总结了一些比较重要、需要注意的知识点，供大家参考。

1. 误解和误用

`float` 被设计出来的初衷是用于文字环绕效果，即一个图片一段文字，图片 `float: left` 之后，文字会环绕图片。

```
<div>  
      
    一段文字一段文字一段文字一段文字一段文字一段文字一段文字  
</div>
```

但是，后来大家发现结合 `float + div` 可以实现之前通过 `table` 实现的网页布局，因此就被“误用”于网页布局了。

题目：为何 `float` 会导致父元素塌陷？

2. 破坏性

```
<div style='border:1px solid blue; padding:3px;'>  
      
</div>
```



```
<div style='border:1px solid blue; padding:3px;'>  
      
</div>
```



`float` 的**破坏性**——`float` 破坏了父标签的原本结构，使得父标签出现了坍塌现象。导致这一现象的最根本原因在于：**被设置了 `float` 的元素会脱离文档流**。其根本原因在于 `float` 的设计初衷是解决文字环绕图片的问题。大家要记住 `float` 的这个影响。

3. 包裹性

包裹性也是 `float` 的一个非常重要的特性，大家用 `float` 时一定要熟知这一特性。咱们还是先从一个例子看起：

没有 `float`

`float:left`

如上图，普通的 `div` 如果没有设置宽度，它会撑满整个屏幕，在之前的盒子模型那一节也讲到过。而如果给 `div` 增加 `float:left` 之后，它突然变得紧凑了，宽度发生了变化，把内容中的三个字包裹了——这就是包裹性。为 `div` 设置了 `float` 之后，其宽度会自动调整为包裹住内容宽度，而不是撑满整个父容器。

- 注意，此时 `div` 虽然体现了包裹性，但是它的 `display` 样式是没有变化的，还是 `display: block`。
- `float` 为什么要具有包裹性？其实答案还是得从 `float` 的设计初衷来寻找，`float` 是被设计用于实现文字环绕效果的。文字环绕图片比较好理解，但是如果想要让文字环绕一个 `div` 呢？此时 `div` 不被“包裹”起来的话，就无法实现环绕效果了。

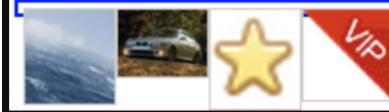
4. 清空格

`float` 还有一个大家可能不是很熟悉的特性——清空格。按照惯例，咱还是先举例子说明。

```
<div style="border: 2px solid blue; padding:3px;">  
      
      
      
      
</div>
```



加上 `float:left` 之后：



- 上面第一张图中，正常的 img 中间是会有空格的，因为多个 img 标签会有换行，而浏览器识别换行为空格，这也是很正常的。第二张图中，为 img 增加了 `float:left` 的样式，这就使得 img 之间没有了空格，4 个 img 紧紧挨着。
- 如果大家之前没注意，现在想想之前写过的程序，是不是有这个特性。为什么 float 适合用于网页排版（俗称“砌砖头”）？就是因为 float 排版出来的网页严丝合缝，中间连个苍蝇都飞不进去。
- “清空格”这一特性的根本原因是 float 会导致节点脱离文档流结构。它都不属于文档流结构了，那么它身边的什么换行、空格就都和它没了关系，它就尽量往一边靠拢，能靠多近就靠多近，这就是清空格的本质。

题目：手写 clearfix

5. `clearfix`

清除浮动的影响，一般使用的样式如下，统称 `clearfix` 代码。所有 float 元素的父容器，一般情况下都应该加 `clearfix` 这个 class。

```
.clearfix:after {  
    content: '';  
    display: table;  
    clear: both;  
}  
.clearfix {  
    *zoom: 1; /* 兼容 IE 低版本 */  
}  


  
      
</div>


```

6. 小结

float 的设计初衷是解决文字环绕图片的问题，后来误打误撞用于做布局，因此有许多不合适或者需要注意的地方，上文基本都讲到了需要的知识点。如果是刚开始接触 float 的同学，学完上面的基础知识之后，还应该做一些练习实战一下——经典的“圣杯布局”和“双飞翼布局”。这里就不再展开讲了，网上资料非常多，例如[浅谈面试中常考的两种经典布局——圣杯与双飞翼](#)（此文的最后两张图清晰地展示了这两种布局）。

#定位 position

`position` 用于网页元素的定位，可设置 `static/relative/absolute/fixed` 这些值，其中 `static` 是默认值，不用介绍。

题目：relative 和 absolute 有何区别？

1. `relative`

相对定位 `relative` 可以用一个例子很轻松地演示出来。例如我们写 4 个 `<p>`，出来的样子大家不用看也能知道。

```
<p>第一段文字</p>
<p>第二段文字</p>
<p>第三段文字</p>
<p>第四段文字</p>
```

第一段文字
第二段文字
第三段文字
第四段文字

然后我们在第三个 `<p>` 上面，加上 `position: relative` 并且设置 `left` 和 `top` 值，看这个 `<p>` 有什么变化。

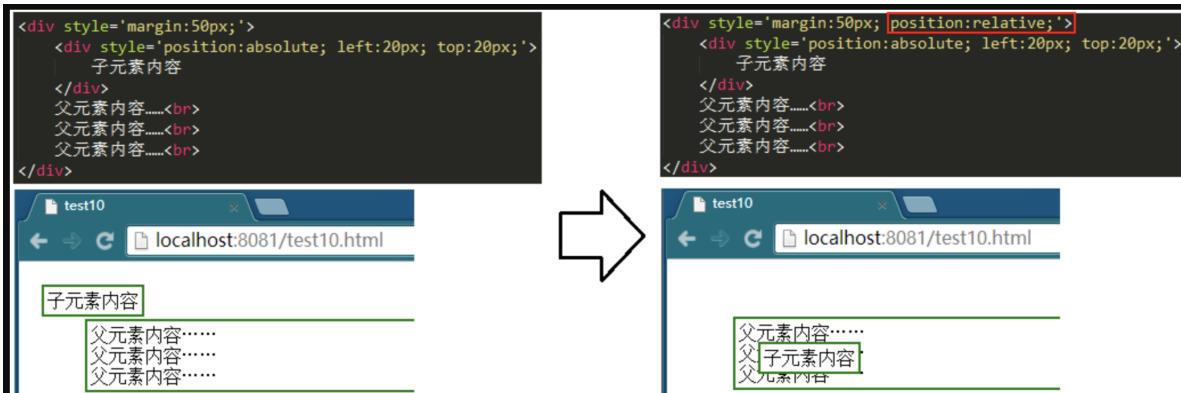
```
<p>第一段文字</p>
<p>第二段文字</p>
<p style="position: relative; top: 10px; left: 10px">第三段文字</p>
<p>第四段文字</p>
```

第一段文字
第二段文字
第三段文字
第四段文字

上图中，大家应该要识别出两个信息（相信大部分人会忽略第二个信息）

- 第三个 `<p>` 发生了位置变化，分别向右向下移动了 `10px`；
- 其他的三个 `<p>` 位置没有发生变化，这一点也很重要。

可见，**relative 会导致自身位置的相对变化，而不会影响其他元素的位置、大小**。这是 `relative` 的要点之一。还有第二个要点，就是 `relative` 产生一个新的定位上下文。下文有关于定位上下文的详细介绍，这里可以先通过一个例子来展示一下区别：



注意看这两图的区别，下文将有解释。

2. absolute

还是先写一个基本的 demo。

```
<p>第一段文字</p>
<p>第二段文字</p>
<p style="background: yellow">第三段文字</p>
<p>第四段文字</p>
```

第一段文字

第二段文字

第三段文字

第四段文字

然后，我们把第三个 `<p>` 改为 `position: absolute;`，看看会发生什么变化。

第一段文字

第二段文字

第四段文字

第三段文字

从上面的结果中，我们能看出几点信息：

- `absolute` 元素脱离了文档结构。和 `relative` 不同，其他三个元素的位置重新排列了。只要元素会脱离文档结构，它就会产生破坏性，导致父元素坍塌。（此时你应该能立刻想起来，`float` 元素也会脱离文档结构。）
- `absolute` 元素具有“包裹性”。之前 `<p>` 的宽度是撑满整个屏幕的，而此时 `<p>` 的宽度刚好是内容的宽度。
- `absolute` 元素具有“跟随性”。虽然 `absolute` 元素脱离了文档结构，但是它的位置并没有发生变化，还是老老实实地呆在它原本的位置，因为我们此时没有设置 `top`、`left` 的值。
- `absolute` 元素会悬浮在页面上方，会遮挡住下方的页面内容。

最后，通过给 `absolute` 元素设置 `top`、`left` 值，可自定义其内容，这个都是平时比较常用的了。这里需要注意的是，设置了 `top`、`left` 值时，元素是相对于最近的定位上下文来定位的，而不是相对于浏览器定位。

3. fixed

其实 `fixed` 和 `absolute` 是一样的，唯一的区别在于：`absolute` 元素是根据最近的定位上下文确定位置，而 `fixed` 根据 `window`（或者 `iframe`）确定位置。

题目：`relative`、`absolute` 和 `fixed` 分别依据谁来定位？

4. 定位上下文

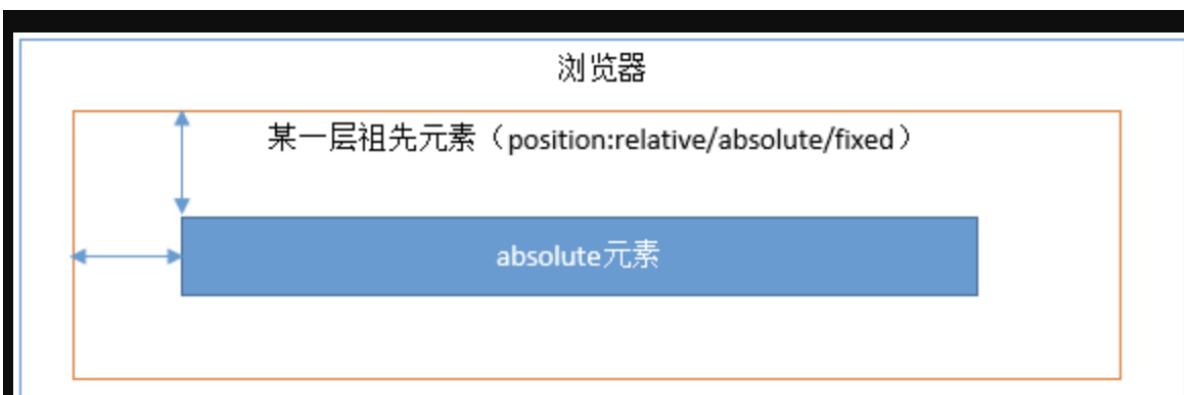
`relative` 元素的定位永远是相对于元素自身位置的，和其他元素没关系，也不会影响其他元素。



`fixed` 元素的定位是相对于 `window` (或者 `iframe`) 边界的，和其他元素没有关系。但是它具有破坏性，会导致其他元素位置的变化。



`absolute` 的定位相对于前两者要复杂许多。如果为 `absolute` 设置了 `top`、`left`，浏览器会根据什么去确定它的纵向和横向的偏移量呢？答案是浏览器会递归查找该元素的所有父元素，如果找到一个设置了 `position:relative/absolute/fixed` 的元素，就以该元素为基准定位，如果没有找到，就以浏览器边界定位。如下两个图所示：



#flex布局

布局的传统解决方案基于盒子模型，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（下文会专门讲解）就不容易实现。在目前主流的移动端页面中，使用 flex 布局能更好地完成需求，因此 flex 布局的知识是必须要掌握的。

1. 基本使用

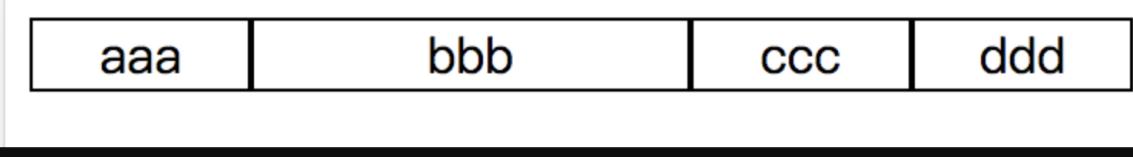
任何一个容器都可以使用 flex 布局，代码也很简单。

```

<style type="text/css">
    .container {
        display: flex;
    }
    .item {
        border: 1px solid #000;
        flex: 1;
    }
</style>

<div class="container">
    <div class="item">aaa</div>
    <div class="item" style="flex: 2">bbb</div>
    <div class="item">ccc</div>
    <div class="item">ddd</div>
</div>

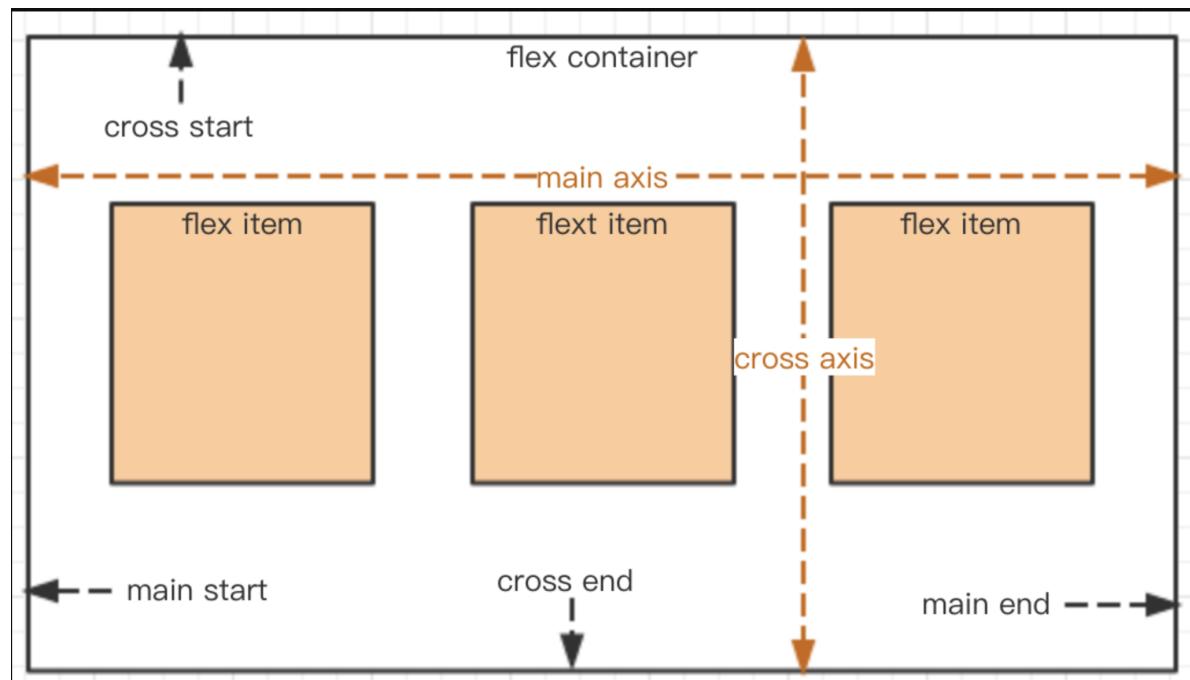
```



注意，第三个 `<div>` 的 `flex: 2`，其他的 `<div>` 的 `flex: 1`，这样第二个 `<div>` 的宽度就是其他的 `<div>` 的两倍。

2. 设计原理

设置了 `display: flex` 的元素，我们称为“容器”（flex container），其所有的子节点我们称为“成员”（flex item）。容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis）。主轴的开始位置（与边框的交叉点）叫做 main start，结束位置叫做 main end；交叉轴的开始位置叫做 cross start，结束位置叫做 cross end。项目默认沿主轴排列。单个项目占据的主轴空间叫做 main size，占据的交叉轴空间叫做 cross size。



将以上文字和图片结合起来，再详细看一遍，这样就能理解 flex 的设计原理，才能更好地实际使用。

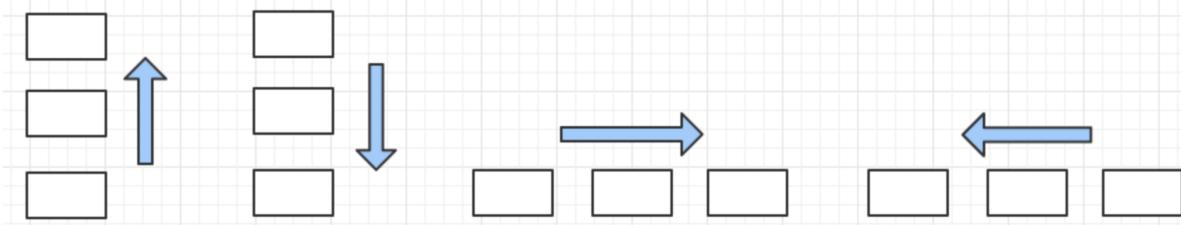
3. 设置主轴的方向

`flex-direction` 可决定主轴的方向，有四个可选值：

- `row` (默认值)：主轴为水平方向，起点在左端。
- `row-reverse`：主轴为水平方向，起点在右端。
- `column`：主轴为垂直方向，起点在上沿。
- `column-reverse`：主轴为垂直方向，起点在下沿。

```
.box {  
  flex-direction: column-reverse | column | row | row-reverse;  
}
```

以上代码设置的主轴方向，将依次对应下图：



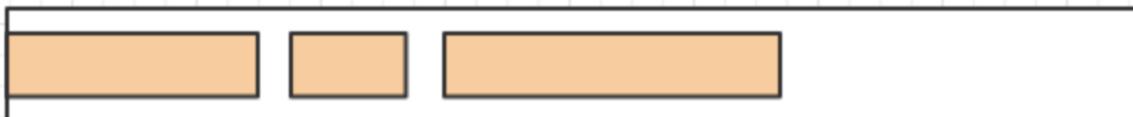
4. 设置主轴的对齐方式

`justify-content` 属性定义了项目在主轴上的对齐方式，值如下：

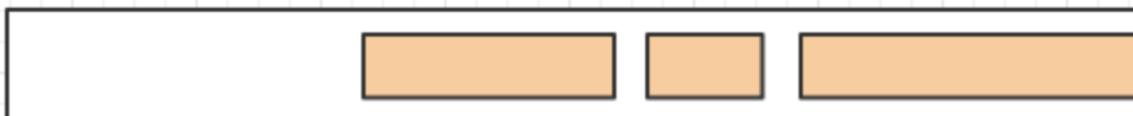
- `flex-start` (默认值)：向主轴开始方向对齐。
- `flex-end`：向主轴结束方向对齐。
- `center`：居中。
- `space-between`：两端对齐，项目之间的间隔都相等。
- `space-around`：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。

```
.box {  
  justify-content: flex-start | flex-end | center | space-between | space-around;  
}
```

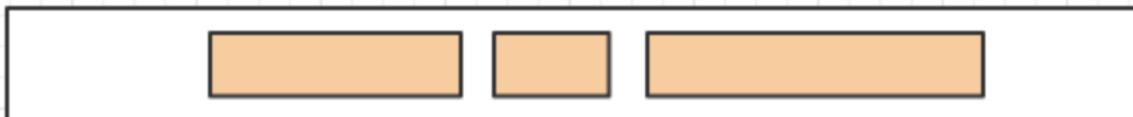
flex-start



flex-end



center



space-between



space-around

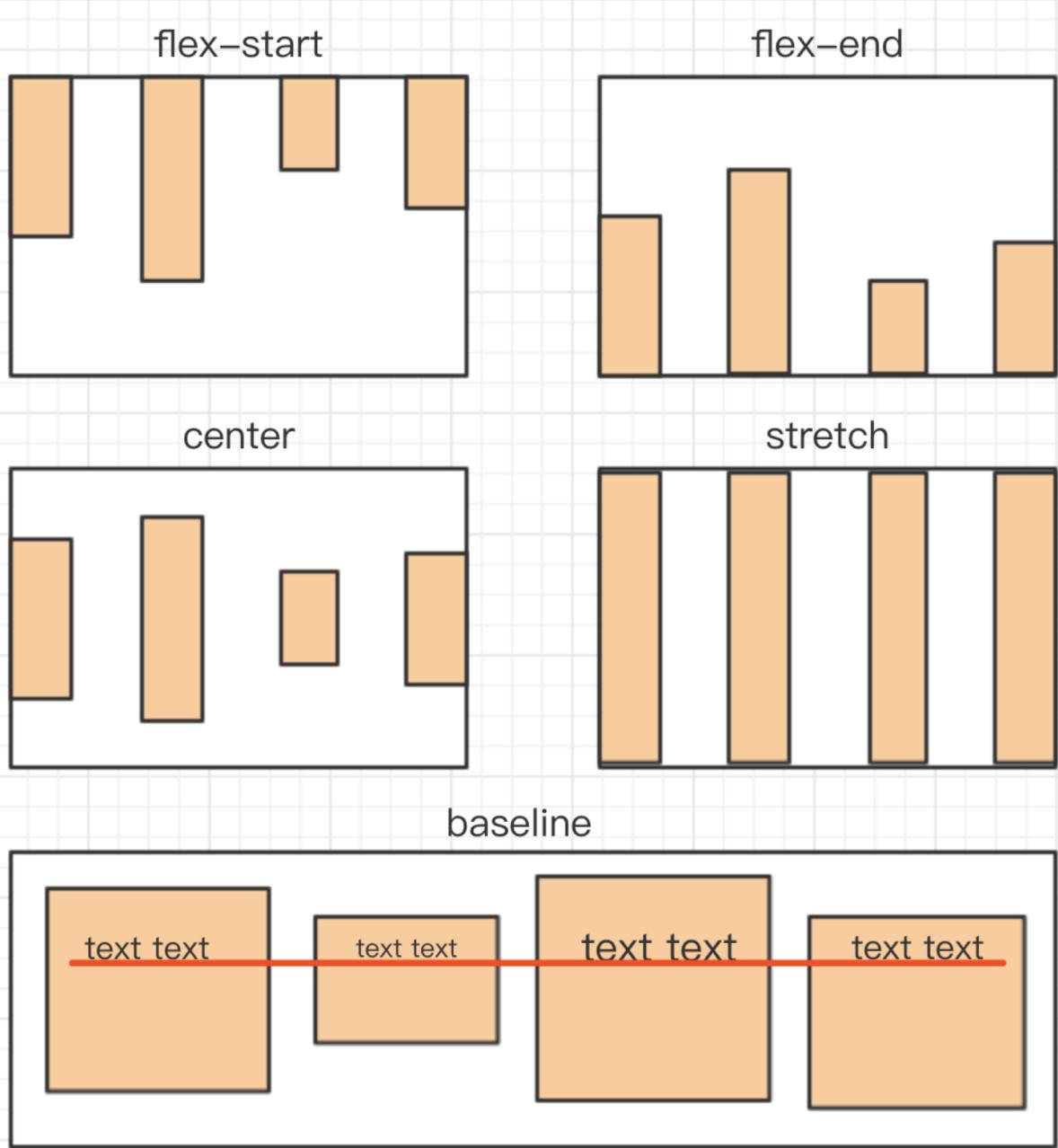


5. 交叉轴的对齐方式

`align-items` 属性定义项目在交叉轴上如何对齐，值如下：

- `flex-start`：交叉轴的起点对齐。
- `flex-end`：交叉轴的终点对齐。
- `center`：交叉轴的中点对齐。
- `baseline`：项目的第一行文字的基线对齐。
- `stretch`（默认值）：如果项目未设置高度或设为 `auto`，将占满整个容器的高度。

```
.box {  
    align-items: flex-start | flex-end | center | baseline | stretch;  
}
```



#如何实现居中对齐?

题目：如何实现水平居中？

1. 水平居中

`inline` 元素用 `text-align: center;` 即可，如下：

```
.container {
  text-align: center;
}
```

`block` 元素可使用 `margin: auto;`，PC 时代的很多网站都这么搞。

```
.container {  
    text-align: center;  
}  
.item {  
    width: 1000px;  
    margin: auto;  
}
```

绝对定位元素可结合 `left` 和 `margin` 实现，但是必须知道宽度。

```
.container {  
    position: relative;  
    width: 500px;  
}  
.item {  
    width: 300px;  
    height: 100px;  
    position: absolute;  
    left: 50%;  
    margin: -150px;  
}
```

题目：如何实现垂直居中？

2. 垂直居中

`inline` 元素可设置 `line-height` 的值等于 `height` 值，如单行文字垂直居中：

```
.container {  
    height: 50px;  
    line-height: 50px;  
}
```

绝对定位元素，可结合 `left` 和 `margin` 实现，但是必须知道尺寸。

- 优点：兼容性好
- 缺点：需要提前知道尺寸

```
.container {  
    position: relative;  
    height: 200px;  
}  
.item {  
    width: 80px;  
    height: 40px;  
    position: absolute;  
    left: 50%;  
    top: 50%;  
    margin-top: -20px;  
    margin-left: -40px;  
}
```

绝对定位可结合 `transform` 实现居中。

- 优点：不需要提前知道尺寸
- 缺点：兼容性不好

```
.container {  
    position: relative;  
    height: 200px;  
}  
.item {  
    width: 80px;  
    height: 40px;  
    position: absolute;  
    left: 50%;  
    top: 50%;  
    transform: translate(-50%, -50%);  
    background: blue;  
}
```

绝对定位结合 `margin: auto`，不需要提前知道尺寸，兼容性好。

```
.container {  
    position: relative;  
    height: 300px;  
}  
.item {  
    width: 100px;  
    height: 50px;  
    position: absolute;  
    left: 0;  
    top: 0;  
    right: 0;  
    bottom: 0;  
    margin: auto;  
}
```

其他的解决方案还有，不过没必要掌握太多，能说出上文的这几个解决方案即可。

#理解语义化

题目：如何理解 HTML 语义化？

所谓“语义”就是为了更易读懂，这要分两部分：

- 让人（写程序、读程序）更易读懂
- 让机器（浏览器、搜索引擎）更易读懂

1. 让人更易读懂

- 对于人来说，代码可读性、语义化就是一个非常广泛的概念了，例如定义 JS 变量的时候使用更易读懂的名称，定义 CSS class 的时候也一样，例如 `length`、`list` 等，而不是使用 `a`、`b` 这种谁都看不懂的名称。
- 不过我们平常考查的“语义化”并不会考查这么广义、这么泛的问题，而是考查 HTML 的语义化，是为了更好地让机器读懂 HTML。

2. 让机器更易读懂

- HTML 符合 XML 标准，但又和 XML 不一样——HTML 不允许像 XML 那样自定义标签名称，HTML 有自己规定的标签名称。问题就在这里——HTML 为何要自己规定那么多标签名称呢，例如 `p`、`div`、`h1`、`u1` 等——就是为了语义化。其实，如果你精通 CSS 的话，你完全可以全部用 `<div>` 标签来实现所有的网页效果，其他的 `p`、`h1`、`u1` 等标签可以一个都不用。但是我们不推荐这么做，这样做就失去了 HTML 语义化的意义。

- 拿搜索引擎来说，爬虫下载到我们网页的 HTML 代码，它如何更好地去理解网页的内容呢？——就是根据 HTML 既定的标签。`h1` 标签就代表是标题；`p` 里面的就是段落详细内容，权重肯定没有标题高；`ul` 里面就是列表；`strong` 就是加粗的强调的内容 如果我们不按照 HTML 语义化来写，全部都用 `<div>` 标签，那搜索引擎将很难理解我们网页的内容。
- 为了加强 HTML 语义化，HTML5 标准中又增加了 `header` `section` `article` 等标签。因此，书写 HTML 时，语义化是非常重要的，否则 W3C 也没必要辛辛苦苦制定出这些标准来。

#CSS3 动画

CSS3 可以实现动画，代替原来的 Flash 和 JavaScript 方案。

首先，使用 `@keyframes` 定义一个动画，名称为 `testAnimation`，如下代码，通过百分比来设置不同的 CSS 样式，规定动画的变化。所有的动画变化都可以这么定义出来。

```
@keyframes testAnimation
{
    0%   {background: red; left:0; top:0;}
    25%  {background: yellow; left:200px; top:0;}
    50%  {background: blue; left:200px; top:200px;}
    75%  {background: green; left:0; top:200px;}
    100% {background: red; left:0; top:0;}
}
```

后，针对一个 CSS 选择器来设置动画，例如针对 `div` 元素设置动画，如下：

```
div {
    width: 100px;
    height: 50px;
    position: absolute;

    animation-name: myfirst;
    animation-duration: 5s;
}
```

`animation-name` 对应到动画名称，`animation-duration` 是动画时长，还有其他属性：

- `animation-timing-function`：规定动画的速度曲线。默认是 `ease`
- `animation-delay`：规定动画何时开始。默认是 0
- `animation-iteration-count`：规定动画被播放的次数。默认是 1
- `animation-direction`：规定动画是否在下一周期逆向地播放。默认是 `normal`
- `animation-play-state`：规定动画是否正在运行或暂停。默认是 `running`
- `animation-fill-mode`：规定动画执行之前和之后如何给动画的目标应用，默认是 `none`，保留在最后一帧可以用 `forwards`

题目：CSS 的 `transition` 和 `animation` 有何区别？

首先 `transition` 和 `animation` 都可以做动效，从语义上来理解，`transition` 是过渡，由一个状态过渡到另一个状态，比如高度 `100px` 过渡到 `200px`；而 `animation` 是动画，即更专业做动效的，`animation` 有帧的概念，可以设置关键帧 `keyframe`，一个动画可以由多个关键帧多个状态过渡组成，另外 `animation` 也包含上面提到的多个属性。

#重绘和回流

重绘和回流是面试题经常考的题目，也是性能优化当中应该注意的点，下面笔者简单介绍下。

- **重绘**: 指的是当页面中的元素不脱离文档流，而简单地进行样式的变化，比如修改颜色、背景等，浏览器重新绘制样式
- **回流**: 指的是处于文档流中 DOM 的尺寸大小、位置或者某些属性发生变化时，导致浏览器重新渲染部分或全部文档的情况

相比之下，**回流要比重绘消耗性能开支更大**。另外，一些属性的读取也会引起回流，比如读取某个 DOM 的高度和宽度，或者使用 `getComputedStyle` 方法。在写代码的时候要避免回流和重绘。比如在笔试中可能会遇见下面的题目：

题目：找出下面代码的优化点，并且优化它

```
var data = ['string1', 'string2', 'string3'];
for(var i = 0; i < data.length; i++){
    var dom = document.getElementById('list');
    dom.innerHTML += '<li>' + data[i] + '</li>';
}
```

上面的代码在循环中每次都获取 `dom`，然后对其内部的 HTML 进行累加 `li`，每次都会操作 DOM 结构，可以改成使用 `documentFragment` 或者先遍历组成 HTML 的字符串，最后操作一次 `innerHTML`。

#小结

本小节总结了 CSS 和 HTML 常考的知识点，包括 CSS 中比较重要的定位、布局的知识，也介绍了一些 CSS3 的知识点概念和题目，以及 HTML 的语义化。

#五、一面 4：从容应对算法题目

由冯·诺依曼机组成我们知道：数据存储和运算是计算机工作的主要内容。`程序=数据结构+算法`，所以计算机类工程师必须掌握一定的数据结构和算法知识。

#知识点梳理

- 常见的数据结构
 - 栈、队列、链表
 - 集合、字典、散列集
- 常见算法
 - 递归
 - 排序
 - 枚举
- 算法复杂度分析
- 算法思维
 - 分治
 - 贪心
 - 动态规划
- 高级数据结构
 - 树、图

- 深度优先和广度优先搜索

本小节会带领大家快速过一遍数据结构和算法，重点讲解一些常考、前端会用到的算法和数据结构。

#数据结构

数据结构决定了数据存储的空间和时间效率问题，数据的写入和提取速度要求也决定了应该选择怎样的数据结构。

根据对场景需求的不同，我们设计不同的数据结构，比如：

- 读得多的数据结构，应该想办法提高数据的读取效率，比如 IP 数据库，只需要写一次，剩下的都是读取；
- 读写都多的数据结构，要兼顾两者的需求平衡，比如 LRU Cache 算法。
- 算法是数据加工处理的方式，一定的算法会提升数据的处理效率。比如有序数组的二分查找，要比普通的顺序查找快很多，尤其是在处理大量数据的时候。
- 数据结构和算法是程序开发的通用技能，所以在任何面试中都可能会遇见。随着近几年 AI、大数据、小游戏越来越火，Web 前端职位难免会跟数据结构和算法打交道，面试中也会出现越来越多的算法题目。学习数据结构和算法也能够帮助我们打开思路，突破技能瓶颈。

#前端常遇見的数据结构問題

現在我來梳理下前端常遇見的数据结构：

- 简单数据结构（必须理解掌握）
 - 有序数据结构：栈、队列、链表，有序数据结构省空间（存储空间小）
 - 无序数据结构：集合、字典、散列表，无序数据结构省时间（读取时间快）
- 复杂数据结构
 - 树、堆
 - 图

对于简单数据结构，在 ES 中对应的是数组（`Array`）和对象（`Object`）。可以想一下，数组的存储是有序的，对象的存储是无序的，但是我要在对象中根据 `key` 找到一个值是立即返回的，数组则需要查找的过程。

这里我通过一个真实面试题目来说明介绍下数据结构设计。

题目：使用 ECMAScript (JS) 代码实现一个事件类`Event`，包含下面功能：绑定事件、解绑事件和派发事件。

- 在稍微复杂点的页面中，比如组件化开发的页面，同一个页面由两三个人来开发，为了保证组件的独立性和降低组件间耦合度，我们往往使用「订阅发布模式」，即组件间通信使用事件监听和派发的方式，而不是直接相互调用组件方法，这就是题目要求写的 `Event` 类。
- 这个题目的核心是一个事件类型对应回调函数的数据设计。为了实现绑定事件，我们需要一个 `_cache` 对象来记录绑定了哪些事件。而事件发生的时候，我们需要从 `_cache` 中读取出来事件回调，依次执行它们。一般页面中事件派发（读）要比事件绑定（写）多。所以我们设计的数据结构应该尽量地能够在事件发生时，更加快速地找到对应事件的回调函数们，然后执行。

经过这样一番考虑，我简单写了下代码实现：

```
class Event {  
    constructor() {  
        // 存储事件的数据结构  
        // 为了查找迅速，使用了对象（字典）  
        this._cache = {};  
    }  
}
```

```

}
// 绑定
on(type, callback) {
    // 为了按类查找方便和节省空间,
    // 将同一类型事件放到一个数组中
    // 这里的数组是队列, 遵循先进先出
    // 即先绑定的事件先触发
    let fns = (this._cache[type] = this._cache[type] || []);
    if (fns.indexOf(callback) === -1) {
        fns.push(callback);
    }
    return this;
}
// 触发
trigger(type, data) {
    let fns = this._cache[type];
    if (Array.isArray(fns)) {
        fns.forEach((fn) => {
            fn(data);
        });
    }
    return this;
}
// 解绑
off(type, callback) {
    let fns = this._cache[type];
    if (Array.isArray(fns)) {
        if (callback) {
            let index = fns.indexOf(callback);
            if (index !== -1) {
                fns.splice(index, 1);
            }
        } else {
            //全部清空
            fns.length = 0;
        }
    }
    return this;
}
}
// 测试用例
const event = new Event();
event.on('test', (a) => {
    console.log(a);
});
event.trigger('test', 'hello world');

event.off('test');
event.trigger('test', 'hello world');

```

类似于树、堆、图这些高级数据结构，前端一般也不会考查太多，但是它们的查找方法却常考，后面介绍。高级数据应该平时多积累，好好理解，比如理解了堆是什么样的数据结构，在面试中遇见的「查找最大的 K 个数」这类算法问题，就会迎刃而解。

#算法的效率是通过算法复杂度来衡量的

算法的好坏可以通过算法复杂度来衡量，算法复杂度包括时间复杂度和空间复杂度两个。时间复杂度由于好估算、好评估等特点，是面试中考查的重点。空间复杂度在面试中考查得不多。

常见的时间复杂度有：

- 常数阶 $O(1)$
- 对数阶 $O(\log N)$
- 线性阶 $O(n)$
- 线性对数阶 $O(n \log N)$
- 平方阶 $O(n^2)$
- 立方阶 $O(n^3)$
- k 次方阶 $O(n^k)$
- 指数阶 $O(2^n)$

随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

一般做算法复杂度分析的时候，遵循下面的技巧：

- 看看有几重循环，一般来说一重就是 $O(n)$ ，两重就是 $O(n^2)$ ，以此类推
- 如果有二分，则为 $O(\log N)$
- 保留最高项，去除常数项

题目：分析下面代码的算法复杂度（为了方便，我已经在注释中加了代码分析）

```
let i = 0; // 语句执行一次
while (i < n) { // 语句执行 n 次
    console.log(`current i is ${i}`); // 语句执行 n 次
    i++; // 语句执行 n 次
}
```

根据注释可以得到，算法复杂度为 $1 + n + n + n = 1 + 3n$ ，去除常数项，为 $O(n)$ 。

```
let number = 1; // 语句执行一次
while (number < n) { // 语句执行 logN 次
    number *= 2; // 语句执行 logN 次
}
```

上面代码 `while` 的跳出判断条件是 `number < n`，而循环体内 `number` 增长速度是 (2^n) ，所以循环代码实际执行 $\log n$ 次，复杂度为： $1 + 2 * \log n = O(\log n)$

```
for (let i = 0; i < n; i++) { // 语句执行 n 次
    for (let j = 0; j < n; j++) { // 语句执行 n^2 次
        console.log('I am here!'); // 语句执行 n^2 次
    }
}
```

上面代码是两个 `for` 循环嵌套，很容易得出复杂度为： $O(n^2)$

#人人都要掌握的基础算法

枚举和递归是最最简单的算法，也是复杂算法的基础，人人都应该掌握！枚举相对比较简单，我们重点说下递归。

递归由下面两部分组成：

- 递归主体，就是要循环解决问题的代码
- 递归的跳出条件，递归不能一直递归下去，需要完成一定条件后跳出

关于递归有个经典的面试题目是：

实现 JS 对象的深拷贝

什么是深拷贝？

「深拷贝」就是在拷贝数据的时候，将数据的所有**引用结构**都拷贝一份。简单的说就是，在内存中存在两个数据结构完全相同又相互独立的数据，将引用型类型进行复制，而不是只复制其引用关系。

分析下怎么做「深拷贝」：

- 首先假设深拷贝这个方法已经完成，为 deepClone
- 要拷贝一个数据，我们肯定要去遍历它的属性，如果这个对象的属性仍是对象，继续使用这个方法，如此往复

```
function deepClone(o1, o2) {  
    for (let k in o2) {  
        if (typeof o2[k] === 'object') {  
            o1[k] = {};  
            deepClone(o1[k], o2[k]);  
        } else {  
            o1[k] = o2[k];  
        }  
    }  
}  
// 测试用例  
let obj = {  
    a: 1,  
    b: [1, 2, 3],  
    c: {}  
};  
let emptyObj = Object.create(null);  
deepClone(emptyObj, obj);  
console.log(emptyObj.a === obj.a);  
console.log(emptyObj.b === obj.b);
```

- 递归容易造成爆栈，尾部调用可以解决递归的这个问题，Chrome 的 V8 引擎做了尾部调用优化，我们在写代码的时候也要注意尾部调用写法。递归的爆栈问题可以通过将递归改写成枚举的方式来解决，就是通过 for 或者 while 来代替递归。
- 我们在使用递归的时候，要注意做优化，比如下面的题目。

题目：求斐波那契数列（兔子数列） 1,1,2,3,5,8,13,21,34,55,89...中的第 n 项

下面的代码中 count 记录递归的次数，我们看下两种差异性的代码中的 count 的值：

```
let count = 0;  
function fn(n) {
```

```

let cache = {};
function _fn(n) {
    if (cache[n]) {
        return cache[n];
    }
    count++;
    if (n == 1 || n == 2) {
        return 1;
    }
    let prev = _fn(n - 1);
    cache[n - 1] = prev;
    let next = _fn(n - 2);
    cache[n - 2] = next;
    return prev + next;
}
return _fn(n);
}

let count2 = 0;
function fn2(n) {
    count2++;
    if (n == 1 || n == 2) {
        return 1;
    }
    return fn2(n - 1) + fn2(n - 2);
}

console.log(fn(20), count); // 6765 20
console.log(fn2(20), count2); // 6765 13529

```

#快排和二分查找

- 前端中面试排序和查找的可能性比较小，因为 JS 引擎已经把这些常用操作优化得很好了，可能项目中你费劲写的一个排序方法，都不如 `Array.sort` 速度快且代码少。因此，掌握快排和二分查找就可以了。
- 快排和二分查找都基于一种叫做「分治」的算法思想，通过对数据进行分类处理，不断降低数量级，实现 $O(\log N)$ (对数级别，比 $O(n)$ 这种线性复杂度更低的一种，快排核心是二分法的 $O(\log N)$ ，实际复杂度为 $O(N \cdot \log N)$) 的复杂度。

1. 快速排序

快排大概的流程是：

- 随机选择数组中的一个数 A，以这个数为基准
- 其他数字跟这个数进行比较，比这个数小的放在其左边，大的放到其右边
- 经过一次循环之后，A 左边为小于 A 的，右边为大于 A 的
- 这时候将左边和右边的数再递归上面的过程

具体代码如下：

```

// 划分操作函数
function partition(array, left, right) {
    // 用index取中间值而非splice
    const pivot = array[Math.floor((right + left) / 2)]
    let i = left
    let j = right

```

```

        while (i <= j) {
            while (compare(array[i], pivot) === -1) {
                i++
            }
            while (compare(array[j], pivot) === 1) {
                j--
            }
            if (i <= j) {
                swap(array, i, j)
                i++
                j--
            }
        }
        return i
    }

// 比较函数
function compare(a, b) {
    if (a === b) {
        return 0
    }
    return a < b ? -1 : 1
}

function quick(array, left, right) {
    let index
    if (array.length > 1) {
        index = partition(array, left, right)
        if (left < index - 1) {
            quick(array, left, index - 1)
        }
        if (index < right) {
            quick(array, index, right)
        }
    }
    return array
}
function quickSort(array) {
    return quick(array, 0, array.length - 1)
}

// 原地交换函数，而非用临时数组
function swap(array, a, b) {
    ;[array[a], array[b]] = [array[b], array[a]]
}
const Arr = [85, 24, 63, 45, 17, 31, 96, 50];
console.log(quickSort(Arr));
// 本版本来自: https://juejin.im/post/5af4902a6fb9a07abf728c40#heading-12

```

2. 二分查找

二分查找法主要是解决「在一堆有序的数中找出指定的数」这类问题，不管这些数是一维数组还是多维数组，只要有序，就可以用二分查找来优化。

二分查找是一种「分治」思想的算法，大概流程如下：

- 数组中排在中间的数字 A，与要找的数字比较大小

- 因为数组是有序的，所以： a) A 较大则说明要查找的数字应该从前半部分查找 b) A 较小则说明应该从查找数字的后半部分查找
- 这样不断查找缩小数量级（扔掉一半数据），直到找完数组为止

题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```
function Find(target, array) {
    let i = 0;
    let j = array[i].length - 1;
    while (i < array.length && j >= 0) {
        if (array[i][j] < target) {
            i++;
        } else if (array[i][j] > target) {
            j--;
        } else {
            return true;
        }
    }
    return false;
}

// 测试用例
console.log(Find(10, [
    [1, 2, 3, 4],
    [5, 9, 10, 11],
    [13, 20, 21, 23]
]))
);
```

另外笔者在面试中遇见过下面的问题：

题目：现在我有一个 1~1000 区间中的正整数，需要你猜下这个数字是几，你只能问一个问题：大了还是小了？问需要猜几次才能猜对？

- 拿到这个题目，笔者想到的就是电视上面有个「猜价格」的购物节目，在规定时间内猜对价格就可以把实物抱回家。所以问题就是让面试官不停地回答我猜的数字比这个数字大了还是小了。这就是二分查找！
- 猜几次呢？其实这个问题就是个二分查找的算法时间复杂度问题，二分查找的时间复杂度是 $O(\log N)$ ，所以求 $\log 1000$ 的解就是猜的次数。我们知道 $2^{10}=1024$ ，所以可以快速估算出： $\log 1000$ 约等于 10，最多问 10 次就能得到这个数！

#面试遇见不会的算法问题怎么办

面试的时候，在遇见算法题目的时候，应该揣摩面试官的意图，听好关键词，比如：有序的数列做查找、要求算法复杂度是 $O(\log N)$ 这类一般就是用二分的思想。

一般来说算法题目的解题思路分以下四步：

- 先降低数量级，拿可以计算出来的情况（数据）来构思解题步骤
- 根据解题步骤编写程序，优先将特殊情况做好判断处理，比如一个大数组的问题，如果数组为两个数长度的情况
- 检验程序正确性
- 是否可以优化（由浅到深），有能力的话可以故意预留优化点，这样可以体现个人技术能力

#正则匹配解题

很多算法题目利用 ES 语法的特性来回答更加简单，比如正则匹配就是常用的一种方式。笔者简单通过几个真题来汇总下正则的知识点。

题目：字符串中第一个出现一次的字符

- 请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符「go」时，第一个只出现一次的字符是「g」。当从该字符流中读出前六个字符「google」时，第一个只出现一次的字符是「l」。
- 这个如果用纯算法来解答需要遍历字符串，统计每个字符出现的次数，然后按照字符串的顺序来找出第一次出现一次的字符，整个过程比较繁琐，如果用正则就简单多了。

```
function find(str){  
    for (var i = 0; i < str.length; i++) {  
        let char = str[i]  
        let reg = new RegExp(char, 'g');  
        let l = str.match(reg).length  
        if(l === 1){  
            return char  
        }  
    }  
}
```

当然，使用 `indexOf/lastIndexOf` 也是一个取巧的方式。再来看一个千分位问题。

题目：将 `1234567` 变成 `1,234,567`，即千分位标注

- 这个题目可以用算法直接来解，如果候选人使用正则来回答，这样主动展现了自己其他方面的优势，即使不是算法解答出来的，面试官一般也不会太难为他。这道题目可以利用正则的「零宽断言」(`(?=exp)`)，意思是它断言自身出现的位置的后面能匹配表达式 `exp`。数字千分位的特点是，第一个逗号后面数字的个数是3的倍数，正则：`/(\d{3})+$`；第一个逗号前最多可以有1~3个数字，正则：`/\d{1,3}/`。加起来就是 `/\d{1,3}(\d{3})+$`，分隔符要从前往后加。

对于零宽断言的详细介绍可以阅读[「零宽断言」这篇文章](#)。

```
function exchange(num) {  
    num += '';  
    if (num.length <= 3) {  
        return num;  
    }  
  
    num = num.replace(/\d{1,3}(?=(\d{3})+$)/g, (v) => {  
        console.log(v)  
        return v + ',';  
    });  
    return num;  
}  
  
console.log(exchange(1234567));
```

当然上面讲到的多数是算法题目取巧的方式，下面这个题目是纯正则考查，笔者在面试的过程中碰见过，这里顺便提一下。

题目，请写出下面的代码执行结果

```
var str = 'google';
var reg = /o/g;
console.log(reg.test(str))
console.log(reg.test(str))
console.log(reg.test(str))
```

代码执行后，会发现，最后一个不是为 `true`，而是 `false`，这是因为 `reg` 这个正则有个 `g`，即 `global` 全局的属性，这种情况下 `lastIndex` 就发挥作用了，可以看下面的代码执行结果就明白了。

```
console.log(reg.test(str), reg.lastIndex)
console.log(reg.test(str), reg.lastIndex)
console.log(reg.test(str), reg.lastIndex)
```

实际开发中也会犯这样的错误，比如为了减少变量每次都重新定义，会把用到的变量提前定义好，这样在使用的时候容易掉进坑里，比如下面代码：

```
(function(){
    const reg = /o/g;
    function isHasO(str){
        // reg.lastIndex = 0; 这样就可以避免这种情况
        return reg.test(str)
    }
    var str = 'google';
    console.log(isHasO(str))
    console.log(isHasO(str))
    console.log(isHasO(str))
})()
```

#小结

本小节介绍了数据结构和算法的关系，作为普通的前端也应该学习数据结构和算法知识，并且顺带介绍了下正则匹配。具体来说，本小节梳理了以下几部分数据结构和算法知识点：

1. 经常用到的数据结构有哪些，它们的特点有哪些
2. 递归和枚举是最基础的算法，必须牢牢掌握
3. 排序里面理解并掌握快速排序算法，其他排序算法可以根据个人实际情况大概了解
4. 有序查找用二分查找
5. 遇见不会的算法问题，先缩小数量级，然后分析推导

当然算法部分还有很多知识，比如动态规划这些算法思想，还有图和树常用到的广度优先搜索和深度优先搜索。这些知识在前端面试和项目中遇见过不多，感兴趣的读者可以在梳理知识点的时候根据个人情况自行决定是否复习。

#六、一面 5：浏览器相关知识点与高频考题解析

`web` 前端工程师写的页面要跑在浏览器里面，所以面试中也会出现很多跟浏览器相关的面试题目。

#知识点梳理

- 浏览器加载页面和渲染过程
- 性能优化
- Web 安全

本小节会从浏览器的加载过程开始讲解，然后介绍如何进行性能优化，最后介绍下 Web 开发中常见的安全问题和预防。

#加载页面和渲染过程

可将加载过程和渲染过程分开说。回答问题的时候，关键要抓住核心的要点，把要点说全面，稍加解析即可，简明扼要不拖沓。

题目：浏览器从加载页面到渲染页面的过程

1. 加载过程

要点如下：

- 浏览器根据 DNS 服务器得到域名的 IP 地址
- 向这个 IP 的机器发送 HTTP 请求
- 服务器收到、处理并返回 HTTP 请求
- 浏览器得到返回内容

例如在浏览器输入 `https://test.com/timeline`，然后经过 DNS 解析，`juejin.im` 对应的 IP 是 `36.248.217.149`（不同时间、地点对应的 IP 可能会不同）。然后浏览器向该 IP 发送 HTTP 请求。

`server` 端接收到 `HTTP` 请求，然后经过计算（向不同的用户推送不同的内容），返回 `HTTP` 请求，返回的内容如下：

```
x Headers Preview Response Cookies Timing
1 <!DOCTYPE html><html lang=zh-CN data-vue-meta-server-rendered><head><meta c
2   <div id="jjis" style="display:none;" data-state="N4Igbglgpg7iBcoDOUCG
3     <script type=text/javascript src=https://gold-cdn.xitu.io/v3/static/j
4   </div>
```

其实就是一堆 HTML 格式的字符串，因为只有 HTML 格式浏览器才能正确解析，这是 W3C 标准的要求。接下来就是浏览器的渲染过程。

2. 渲染过程

要点如下：

- 根据 HTML 结构生成 `DOM` 树
- 根据 CSS 生成 `CSSOM`
- 将 `DOM` 和 `CSSOM` 整合形成 `RenderTree`
- 根据 `RenderTree` 开始渲染和展示
- 遇到 `<script>` 时，会执行并阻塞渲染
- 上文中，浏览器已经拿到了 `server` 端返回的 HTML 内容，开始解析并渲染。最初拿到的内容就是一堆字符串，必须先结构化成计算机擅长处理的基本数据结构，因此要把 HTML 字符串转化成 `DOM` 树——树是最基本的数据结构之一。
- 解析过程中，如果遇到 `<link href="...">` 和 `<script src="...">` 这种外链加载 CSS 和 JS 的标签，浏览器会异步下载，下载过程和上文中下载 HTML 的流程一样。只不过，这里下载下来的字

字符串是 CSS 或者 JS 格式的。

- 浏览器将 CSS 生成 CSSOM，再将 DOM 和 CSSOM 整合成 RenderTree，然后针对 RenderTree 即可进行渲染了。大家可以想一下，有 DOM 结构、有样式，此时就能满足渲染的条件了。另外，这里也可以解释一个问题 —— **为何要将 CSS 放在 HTML 头部？** —— 这样会让浏览器尽早拿到 CSS 尽早生成 CSSOM，然后在解析 HTML 之后可一次性生成最终的 RenderTree，渲染一次即可。如果 CSS 放在 HTML 底部，会出现渲染卡顿的情况，影响性能和体验。
- 最后，渲染过程中，如果遇到 `<script>` 就停止渲染，执行 JS 代码。因为浏览器渲染和 JS 执行共用一个线程，而且这里必须是单线程操作，多线程会产生渲染 DOM 冲突。待 `<script>` 内容执行完之后，浏览器继续渲染。最后再思考一个问题 —— **为何要将 JS 放在 HTML 底部？** —— JS 放在底部可以保证让浏览器优先渲染完现有的 HTML 内容，让用户先看到内容，体验好。另外，JS 执行如果涉及 DOM 操作，得等待 DOM 解析完成才行，JS 放在底部执行时，HTML 肯定都解析成了 DOM 结构。JS 如果放在 HTML 顶部，JS 执行的时候 HTML 还没来得及转换为 DOM 结构，可能会报错。

关于浏览器整个流程，百度的多益大神有更加详细的文章，推荐阅读下：《[从输入 URL 到页面加载完成的过程中都发生了什么事情？](#)》。

#性能优化

性能优化的题目也是面试常考的，这类题目有很大的扩展性，能够扩展出来很多小细节，而且对个人的技术视野和业务能力有很大的挑战。这部分笔者会重点讲下常用的性能优化方案。

题目：总结前端性能优化的解决方案

1. 优化原则和方向

性能优化的原则是以**更好的用户体验为标准**，具体就是实现下面的目标：

1. 多使用内存、缓存或者其他方法
2. 减少 CPU 和 GPU 计算，更快展现

优化的方向有两个：

- **减少页面体积，提升网络加载**
- **优化页面渲染**

2. 减少页面体积，提升网络加载

- 静态资源的压缩合并 (JS 代码压缩合并、CSS 代码压缩合并、雪碧图)
- 静态资源缓存 (资源名称加 MD5 截)
- 使用 `CDN` 让资源加载更快

3. 优化页面渲染

- CSS 放前面，JS 放后面
- 懒加载 (图片懒加载、下拉加载更多)
- 减少 `DOM` 查询，对 `DOM` 查询做缓存
- 减少 `DOM` 操作，多个操作尽量合并在一起执行 (`DocumentFragment`)
- 事件节流
- 尽早执行操作 (`DOMContentLoaded`)
- 使用 `SSR` 后端渲染，数据直接输出到 HTML 中，减少浏览器使用 JS 模板渲染页面 HTML 的时间

#详细解释

1. 静态资源的压缩合并

如果不合并，每个都会走一遍之前介绍的请求过程

```
<script src="a.js"></script>
<script src="b.js"></script>
<script src="c.js"></script>
```

如果合并了，就只走一遍请求过程

```
<script src="abc.js"></script>
```

2. 静态资源缓存

通过链接名称控制缓存

```
<script src="abc_1.js"></script>
```

只有内容改变的时候，链接名称才会改变

```
<script src="abc_2.js"></script>
```

这个名称不用手动改，可通过前端构建工具根据文件内容，为文件名称添加 MD5 后缀。

3. 使用 CDN 让资源加载更快

CDN 会提供专业的加载优化方案，静态资源要尽量放在 CDN 上。例如：

```
<script src="https://cdn.bootcss.com/zepto/1.0rc1/zepto.min.js"></script>
```

4. 使用 SSR 后端渲染

可一次性输出 HTML 内容，不用在页面渲染完成之后，再通过 Ajax 加载数据、再渲染。例如使用 smarty、Vue SSR 等。

5. CSS 放前面，JS 放后面

上文讲述浏览器渲染过程时已经提过，不再赘述。

6. 懒加载

一开始先给为 `src` 赋值成一个通用的预览图，下拉时候再动态赋值成正式的图片。如下，`preview.png` 是预览图片，比较小，加载很快，而且很多图片都共用这个 `preview.png`，加载一次即可。待页面下拉，图片显示出来时，再去替换 `src` 为 `data-realsrc` 的值。

```

```

另外，这里为何要用 `data-` 开头的属性值？——所有 HTML 中自定义的属性，都应该用 `data-` 开头，因为 `data-` 开头的属性浏览器渲染的时候会忽略掉，提高渲染性能。

7. DOM 查询做缓存

两段代码做一下对比：

```
var pList = document.getElementsByTagName('p') // 只查询一个 DOM，缓存在 pList 中了
var i
for (i = 0; i < pList.length; i++) {
}
var i
for (i = 0; i < document.getElementsByTagName('p').length; i++) { // 每次循环，都会查询 DOM，耗费性能
}
```

总结：DOM 操作，无论查询还是修改，都是非常耗费性能的，应尽量减少。

8. 合并 DOM 插入

DOM 操作是非常耗费性能的，因此插入多个标签时，先插入 Fragment 然后再统一插入 DOM。

```
var listNode = document.getElementById('list')
// 要插入 10 个 li 标签
var frag = document.createDocumentFragment();
var x, li;
for(x = 0; x < 10; x++) {
    li = document.createElement("li");
    li.innerHTML = "List item " + x;
    frag.appendChild(li); // 先放在 frag 中，最后一次性插入到 DOM 结构中。
}
listNode.appendChild(frag);
```

10. 事件节流

例如要在文字改变时触发一个 change 事件，通过 keyup 来监听。使用节流。

```
var textarea = document.getElementById('text')
var timeoutId
textarea.addEventListener('keyup', function () {
    if (timeoutId) {
        clearTimeout(timeoutId)
    }
    timeoutId = setTimeout(function () {
        // 触发 change 事件
    }, 100)
})
```

11. 尽早执行操作

```
window.addEventListener('load', function () {
    // 页面的全部资源加载完才会执行，包括图片、视频等
})
document.addEventListener('DOMContentLoaded', function () {
    // DOM 渲染完即可执行，此时图片、视频还可能没有加载完
})
```

12. 性能优化怎么做

上面提到的都是性能优化的单个点，性能优化项目具体实施起来，应该按照下面步骤推进：

1. 建立性能数据收集平台，摸底当前性能数据，通过性能打点，将上述整个页面打开过程消耗时间记录下来
2. 分析耗时较长时间段原因，寻找优化点，确定优化目标
3. 开始优化
4. 通过数据收集平台记录优化效果
5. 不断调整优化点和预期目标，循环2~4步骤

性能优化是个长期的事情，不是一蹴而就的，应该本着先摸底、再分析、后优化的原则逐步来做。

#Web 安全

题目：前端常见的安全问题有哪些？

- Web 前端的安全问题，能回答出下文的两个问题，这个题目就能基本过关了。开始之前，先说一个最简单的攻击方式——SQL 注入。
- 上学的时候就知道有一个「SQL注入」的攻击方式。例如做一个系统的登录界面，输入用户名和密码，提交之后，后端直接拿到数据就拼接 SQL 语句去查询数据库。如果在输入时进行了恶意的 SQL 拼装，那么最后生成的 SQL 就会有问题。但是现在稍微大型一点的系统，都不会这么做，从提交登录信息到最后拿到授权，要经过层层的验证。因此，SQL 注入都只出现在比较低端小型的系统上。

1. XSS (Cross Site Scripting, 跨站脚本攻击)

- 这是前端最常见的攻击方式，很多大型网站（如 Facebook）都被 XSS 攻击过。
- 举一个例子，我在一个博客网站正常发表一篇文章，输入汉字、英文和图片，完全没有问题。但是如果我写的是恶意的 JS 脚本，例如获取到 `document.cookie` 然后传输到自己的服务器上，那我这篇博客的每一次浏览都会执行这个脚本，都会把访客 cookie 中的信息偷偷传递到我的服务器上来。

其实原理上就是黑客通过某种方式（发布文章、发布评论等）将一段特定的 JS 代码隐蔽地输入进去。然后别人再看这篇文章或者评论时，之前注入的这段 JS 代码就执行了。**JS 代码一旦执行，那就可就不受控制了，因为它跟网页原有的 JS 有同样的权限**，例如可以获取 server 端数据、可以获取 cookie 等。于是，攻击就这样发生了。

XSS的危害

XSS 的危害相当大，如果页面可以随意执行别人不安全的 JS 代码，轻则会让页面错乱、功能缺失，重则会造成用户的信息泄露。

比如早些年社交网站经常爆出 XSS 蠕虫，通过发布的文章内插入 JS，用户访问了感染不安全 JS 注入的文章，会自动重新发布新的文章，这样的文章会通过推荐系统进入到每个用户的的文章列表面前，很快就会造成大规模的感染。

还有利用获取 cookie 的方式，将 cookie 传入入侵者的服务器上，入侵者就可以模拟 cookie 登录网站，对用户的信息进行篡改。

XSS的预防

那么如何预防 XSS 攻击呢？——最根本的方式，就是对用户输入的内容进行验证和替换，需要替换的字符有：

```
& 替换为: &amp;
< 替换为: &lt;
> 替换为: &gt;
" 替换为: &quot;
' 替换为: &#x27;
/ 替换为: &#x2f;
```

替换了这些字符之后，黑客输入的攻击代码就会失效，XSS 攻击将不会轻易发生。

除此之外，还可以通过对 cookie 进行较强的控制，比如对敏感的 cookie 增加 `http-only` 限制，让 JS 获取不到 cookie 的内容。

2. CSRF (Cross-site request forgery, 跨站请求伪造)

CSRF 是借用了当前操作者的权限来偷偷地完成某个操作，而不是拿到用户的信息。

- 例如，一个支付类网站，给他人转账的接口是 `http://buy.com/pay?touid=999&money=100`，而这个接口在使用时没有任何密码或者 token 的验证，只要打开访问就直接给他人转账。一个用户已经登录了 `http://buy.com`，在选择商品时，突然收到一封邮件，而这封邮件正文有这么一行代码 ``，他访问了邮件之后，其实就已经完成了购买。
- CSRF 的发生其实是借助了一个 cookie 的特性。我们知道，登录了 `http://buy.com` 之后，cookie 就会有登录过的标记了，此时请求 `http://buy.com/pay?touid=999&money=100` 是会带着 cookie 的，因此 server 端就知道已经登录了。而如果在 `http://buy.com` 去请求其他域名的 API 例如 `http://abc.com/api` 时，是不会带 cookie 的，这是浏览器的同源策略的限制。但是——**此时在其他域名的页面中，请求 `http://buy.com/pay?touid=999&money=100`，会带着 `buy.com` 的 cookie，这是发生 CSRF 攻击的理论基础。**

预防 CSRF 就是加入各个层级的权限验证，例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者指纹才行。除此之外，敏感的接口使用 `POST` 请求而不是 `GET` 也是很重要的。

#七、一面 6：开发环境相关知识点与高频考题解析

#知识点梳理

- IDE
- Git
- Linux 基础命令
- 前端构建工具
- 调试方法

本小节会重点介绍 Git 的基本用法、代码部署和开发中常用的 Linux 命令，然后以 webpack 为例介绍下前端构建工具，最后介绍怎么抓包解决线上问题。这些都是日常开发和面试中常用到的知识。

#IDE

题目：你平时都使用什么 IDE 编程？有何提高效率的方法？

- 前端最常用的 IDE 有 [Webstorm](#)、[Sublime](#)、[Atom](#) 和 [VSCode](#)，我们可以分别去它们的官网看一下。
- Webstorm 是最强大的编辑器，因为它拥有各种强大的插件和功能，但是我没有用过，因为它收费。不是我舍不得花钱，而是因为我觉得免费的 Sublime 已经够我用了。跟面试官聊到 Webstorm 的时候，没用过沒事儿，但一定要知道它：第一，强大；第二，收费。
- Sublime 是我日常用的编辑器，第一它免费，第二它轻量、高效，第三它插件非常多。用 Sublime 一定要安装各种插件配合使用，可以去网上搜一下“sublime”常用插件的安装以及用法，还有它的各种快捷键，并且亲自使用它。这里就不一一演示了，网上的教程也很傻瓜式。
- Atom 是 GitHub 出品的编辑器，跟 Sublime 差不多，免费并且插件丰富，而且跟 Sublime 相比风格上还有些小清新。但是我用过几次就不用了，因此它打开的时候会比较慢，卡一下才打开。当然总体来说也是很好用的，只是个人习惯问题。

- VSCode 是微软出品的轻量级（相对于 Visual Studio 来说）编辑器，微软做 IDE 那是出了名的好，出了名的大而全，因此 VSCode 也有上述 Sublime 和 Atom 的各种优点，但是我也是因为个人习惯问题（本人不愿意尝试没有新意的新东西），用过几次就不用了。

总结一下：

- 如果你要走大牛、大咖、逼格的路线，就用 Webstorm
- 如果你走普通、屌丝、低调路线，就用 Sublime
- 如果你走小清新、个性路线，就用 VSCode 或者 Atom
- 如果你面试，最好有一个用的熟悉，其他都会一点

最后注意：千万不要说你使用 Dreamweaver 或者 notepad++ 写前端代码，会被鄙视的。如果你不做 .NET 也不要用 Visual Studio，不做 Java 也不要用 Eclipse。

#Git

- 你此前做过的项目一定要用过 Git，而且必须是命令行，如果没用过，你自己也得恶补一下。对 Git 的基本应用比较熟悉的同学，可以跳过这一部分了。macOS 自带 Git，Windows 需要安装 Git 客户端，去 [Git 官网](#) 下载即可。
- 国内比较好的 Git 服务商有 coding.net，国外有大名鼎鼎的 GitHub，但是有时会有网络问题，因此建议大家注册一个 coding.net 账号然后创建项目，来练练手。

题目：常用的 Git 命令有哪些？如何使用 Git 多人协作开发？

1. 常用的 Git 命令

首先，通过 `git clone <项目远程地址>` 下载下来最新的代码，例如 `git clone git@git.coding.net:username/project-name.git`，默认会下载 master 分支。

- 然后修改代码，修改过程中可以通过 `git status` 看到自己的修改情况，通过 `git diff <文件名>` 可查阅单个文件的差异。
- 最后，将修改的内容提交到远程服务器，做如下操作

```
git add .
git commit -m "xxx"
git push origin master
```

如果别人也提交了代码，你想同步别人提交的内容，执行 `git pull origin master` 即可。

2. 如何多人协作开发

- 多人协作开发，就不能使用 master 分支了，而是要每个开发者单独拉一个分支，使用 `git checkout -b <branchname>`，运行 `git branch` 可以看到本地所有的分支名称。
- 自己的分支，如果想同步 master 分支的内容，可运行 `git merge master`。切换分支可使用 `git checkout <branchname>`。
- 在自己的分支上修改了内容，可以将自己的分支提交到远程服务器

```
git add .
git commit -m "xxx"
git push origin <branchname>
```

最后，待代码测试没问题，再将自己分支的内容合并到 master 分支，然后提交到远程服务器。

```
git checkout master
git merge <branchname>
git push origin master
```

#关于 SVN

关于 SVN 笔者的态度和针对 IE 低版本浏览器的态度一样，你只需要查询资料简单了解一下。面试的时候可能会问到，但你只要熟悉了 Git 的操作，面试官不会因为你不熟悉 SVN 而难为你。前提是你要知道一点 SVN 的基本命令，自己上网一查就行。

不过 SVN 和 Git 的区别你得了解。SVN 是每一步操作都离不开服务器，创建分支、提交代码都需要连接服务器。而 Git 就不一样了，你可以在本地创建分支、提交代码，最后再一起 push 到服务器上。因此，Git 拥有 SVN 的所有功能，但是却比 SVN 强大得多。（Git 是 Linux 的创始人 Linus 发明的东西，因此也倍得推崇。）

#Linux 基础命令

- 目前互联网公司的线上服务器都使用 Linux 系统，测试环境为了保证和线上一致，肯定也是使用 Linux 系统，而且都是命令行的，没有桌面，不能用鼠标操作。因此，掌握基础的 Linux 命令是非常必要的。下面总结一些最常用的 Linux 命令，建议大家在真实的 Linux 系统下亲自试一下。
- 关于如何得到 Linux 系统，有两种选择：第一，在自己电脑的虚拟机中安装一个 Linux 系统，例如 Ubuntu/CentOS 等，下载这些都不用花钱；第二，花钱去阿里云等云服务商租一个最便宜的 Linux 虚拟机。推荐第二种。一般正式入职之后，公司都会给你分配开发机或者测试机，给你账号和密码，你自己可以远程登录。

题目：常见 linux 命令有哪些？

1. 登录

入职之后，一般会有现有的用户名和密码给你，你拿来之后直接登录就行。运行 `ssh name@server` 然后输入密码即可登录。

2. 目录操作

- 创建目录 `mkdir <目录名称>`
- 删除目录 `rm <目录名称>`
- 定位目录 `cd <目录名称>`
- 查看目录文件 `ls ll`
- 修改目录名 `mv <目录名称> <新目录名称>`
- 拷贝目录 `cp <目录名称> <新目录名称>`

3. 文件操作

- 创建文件 `touch <文件名称> vi <文件名称>`
- 删除文件 `rm <文件名称>`
- 修改文件名 `mv <文件名称> <新文件名称>`
- 拷贝文件 `cp <文件名称> <新文件名称>`

4. 文件内容操作

- 查看文件 `cat <文件名称> head <文件名称> tail <文件名称>`
- 编辑文件内容 `vi <文件名称>`
- 查找文件内容 `grep '关键字' <文件名称>`

#前端构建工具

构建工具是前端工程化中不可缺少的一环，非常重要，而在面试中却有其特殊性——**面试官会通过询问构建工具的作用、目的来询问你对构建工具的了解，只要这些你都知道，不会再追问细节。**因为，在实际工作中，真正能让你编写构建工具配置文件的机会非常少，一个项目就配置一次，后面就很少改动了。而且，如果是大众使用的框架（如 React、Vue 等），还会直接有现成的脚手架工具，一键创建开发环境，不用手动配置。

题目：前端为何要使用构建工具？它解决了什么问题？

1. 何为构建工具

“构建”也可理解为“编译”，就是将开发环境的代码转换成运行环境代码的过程。**开发环境的代码是为了更好地阅读，而运行环境的代码是为了更快地执行，两者目的不一样，因此代码形式也不一样。**例如，开发环境写的 JS 代码，要通过混淆压缩之后才能放在线上运行，因为这样代码体积更小，而且对代码执行不会有任何影响。总结一下需要构建工具处理的几种情况：

- **处理模块化**：CSS 和 JS 的模块化语法，目前都无法被浏览器兼容。因此，开发环境可以使用既定的模块化语法，但是需要构建工具将模块化语法编译为浏览器可识别形式。例如，使用 webpack、Rollup 等处理 JS 模块化。
- **编译语法**：编写 CSS 时使用 Less、Sass，编写 JS 时使用 ES6、TypeScript 等。这些标准目前也都无法被浏览器兼容，因此需要构建工具编译，例如使用 Babel 编译 ES6 语法。
- **代码压缩**：将 CSS、JS 代码混淆压缩，为了让代码体积更小，加载更快。

2. 构建工具介绍

- 最早普及使用的构建工具是 [Grunt](#)，不久又被 [Gulp](#) 给追赶上。Gulp 因其简单的配置以及高效的性能而被大家所接受，也是笔者个人比较推荐的构建工具之一。如果你做一些简单的 JS 开发，可以考虑使用。
- 如果你的项目比较复杂，而且是多人开发，那么你就需要掌握目前构建工具届的神器——webpack。不过神器也有一个缺点，就是学习成本比较高，需要拿出专门的时间来专心学习，而不是三言两语就能讲完的。我们下面就演示一下 webpack 最简单的使用，全面的学习还得靠大家去认真查阅相关文档，或者参考专门讲解 webpack 的教程。

3. webpack 演示

- 接下来我们演示一下 webpack 处理模块化和混淆压缩代码这两个基本功能。
- 首先，你需要安装 Node.js，没有安装的可以去 [Node.js 官网](#) 下载并安装。安装完成后运行如下命令来验证是否安装成功。

```
node -v  
npm -v
```

- 然后，新建一个目录，进入该目录，运行 `npm init`，按照提示输入名称、版本、描述等信息。完成之后，该目录下出现了一个 `package.json` 文件，是一个 JSON 文件。
- 接下来，安装 webpack，运行 `npm i --save-dev webpack`，网络原因需要耐心等待几分钟。
- 接下来，编写源代码，在该目录下创建 `src` 文件夹，并在其中创建 `app.js` 和 `dt.js` 两个文件，文件内容分别是：

```
// dt.js 内容  
module.exports = {  
    getDateNow: function () {  
        return Date.now()  
    }  
  
// app.js 内容  
var dt = require('./dt.js')  
alert(dt.getDateNow())
```

然后，再返回上一层目录，新建 `index.html` 文件（该文件和 `src` 属于同一层级），内容是

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>test</title>
</head>
<body>
    <div>test</div>

    <script src='./dist/bundle.js'></script>
</body>
</html>
```

然后，编写 webpack 配置文件，新建 `webpack.config.js`，内容是

```
const path = require('path');
const webpack = require('webpack');
module.exports = {
    context: path.resolve(__dirname, './src'),
    entry: {
        app: './app.js',
    },
    output: {
        path: path.resolve(__dirname, './dist'),
        filename: 'bundle.js',
    },
    plugins: [
        new webpack.optimize.UglifyJsPlugin({
            compress: {
                //suppresses warnings, usually from module minification
                warnings: false
            }
        }),
    ]
};
```

总结一下，目前项目的文件目录是：

```
src
  +- app.js
  +- dt.js
index.html
package.json
webpack.config.js
```

接下来，打开 `package.json`，然后修改其中 `scripts` 的内容为：

```
"scripts": {
    "start": "webpack"
}
```

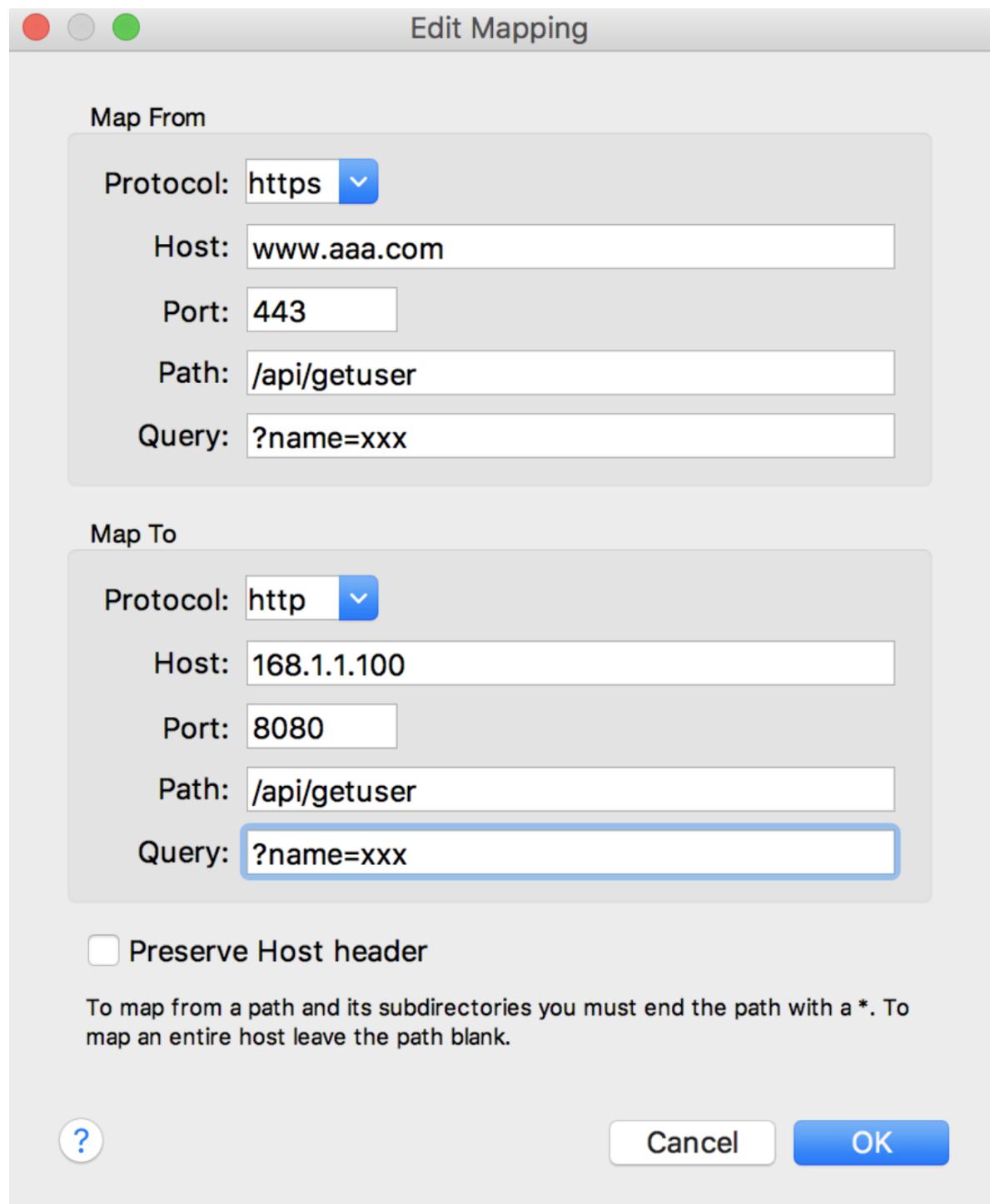
在命令行中运行 `npm start`，即可看到编译的结果，最后在浏览器中打开 `index.html`，即可弹出 `Date.now()` 的值。

#调试方法

调试方法这块被考查最多的就是如何进行抓包。

题目：如何抓取数据？如何使用工具来配置代理？

- PC 端的网页，我们可以通过 Chrome、Firefox 等浏览器自带的开发者工具来查看网页的所有网络请求，以帮助排查 bug。这种监听、查看网络请求的操作称为**抓包**。
- 针对移动端的抓包工具，Mac 系统下推荐使用 Charles 这个工具，首先[下载](#)并安装，打开。Windows 系统推荐使用[Fiddler](#)，下载安装打开。两者使用基本一致，下面以 Charles 为例介绍。
- 接下来，将安装好 Charles 的电脑和要抓包的手机，连接到同一个网络（一般为公司统一提供的内网，由专业网络工程师搭建），保证 IP 段相同。然后，将手机设置网络代理（每种不同手机如何设置网络代理，网上都有傻瓜式教程），代理的 IP 为电脑的 IP，代理的端口为 8888。然后，Charles 可能会有一个弹框提示是否允许连接代理，这里选择“允许”即可。这样，使用手机端访问的网页或者联网的请求，Charles 就能监听到了。
- 在开发过程中，经常用到抓包工具来做代理，将线上的地址代理到测试环境，Charles 和 Fiddler 都可实现这个功能。以 Charles 为例，点击菜单栏中 Tools 菜单，然后二级菜单中点击 Map Remote，会弹出配置框。首先，选中 Enable Map Remote 复选框，然后点击 Add 按钮，添加一个代理项。例如，如果要将线上的 `https://www.aaa.com/api/getuser?name=xxx` 这个地址代理到测试地址 `http://168.1.1.100:8080/api/getuser?name=xxx`，配置如下图



#小结

本小节总结了前端开发环境常考查的知识，这些知识也是前端程序员必须掌握的，否则会影响开发效率。

#八、二面 1：如何回答常见的软技能问题

面试是个技术活，不仅仅是技术，各种软技能的面试技巧也是非常重要的，尤其是程序员一般对于自己的软技能不是很看重，其实软技能才是决定你职场能够走多远的关键。

#程序员应该具备的软技能

程序员除了业务技能外，应该具有下面的软技能：

1. 韧性：抗压能力，在一定项目压力下能够迎难而上，比如勇于主动承担和解决技术难题
2. 责任心：对于自己做过的项目，能够出现 bug 之类主动解决
3. 持续学习能力：IT 行业是个需要不断充电的行业，尤其 Web 前端这些年一直在巨变，所以持续学习能力很重要
4. 团队合作能力：做项目不能个人英雄主义，应该融入团队，跟团队一起打仗
5. 交流沟通能力：经常会遇见沟通需求和交互设计的工作，应该乐于沟通分享

另外在《软技能：代码之外的生存指南》这本书里提到了下面一些软技能：

1. 职业
2. 自我营销
3. 学习能力
4. 提升工作效率
5. 理财
6. 健身
7. 积极的人生观

#常见的软技能问题和提升

回答软技能类的问题，应该注意在回答过程中体现自己具备的软技能。下面列举几个常见的软技能类的问题。

1. 回想下你遇见过最难打交道的同事，你是如何跟他沟通的

- 一般来说，工作中总会遇见一两个自己不喜欢的人，这种情况应该尽量避免冲突，从自己做起慢慢让对方感觉到自己的合作精神。
- 所以，遇见难打交道的同事，不要急于上报领导，应该自己主动多做一些事情，比如规划好工作安排，让他选择自己做的事情，有了结论记得发邮件确认下来，这样你们的领导和其他成员都会了解到工作的安排，在鞭笞对方的同时，也做到了职责明确。在项目当中，多主动检查项目进展，提前发现逾期的问题。
- 重点是突出：自己主动沟通解决问题的意识，而不是遇见问题就找领导。

2. 当你被分配一个几乎不可能完成的任务时，你会怎么做

这种情况下，一般通过下面方式来解决：

1. 自己先查找资料，寻找解决方案，评估自己需要怎样的资源来完成，需要多长时间
2. 能不能借助周围同事来解决问题
3. 拿着分析结果跟上级反馈，寻求帮助或者资源

突出的软技能：分析和解决问题，沟通寻求帮助。

3. 业余时间都做什么？除了写码之外还有什么爱好

这类问题也是面试官的高频问题，「一个人的业余时间决定了他的未来」，如果回答周末都在追剧打游戏之类的，未免显得太不上进。

一般来说，推荐下面的回答：

周末一般会有三种状态：

- 和朋友一起去做运动，也会聚会聊天，探讨下新技术之类的；
- 也会看一些书籍充充电，比如我最近看的 xx，有什么的想法；
- 有时候会闷在家用最近比较火的技术做个小项目或者实现个小功能之类的。

这样的回答，既能表现自己阳光善于社交沟通的一面，又能表现自己的上进心。

#小结

本小节介绍了程序员除了业务技术能力之外应该日常修炼的软技能，在面试中，软技能会被以各种形式问起，候选人应该先了解有哪些软技能可以修炼，才能在回答软技能问题的时候，尽量提到自己具备的软技能。

#九、二面 2：如何介绍项目及应对项目细节追问

一个标准的面试流程中，肯定会在一面二面中问到你具体做过的项目，然后追问项目的细节。这类问题往往通过下面形式来提问：

1. 发现你简历的一个项目，直接让你介绍下这个项目
2. 让你回忆下你做过的项目中，最值得分享（最大型/最困难/最能体现技术能力/最难忘）的
3. 如果让你设计 xx 系统/项目，你会怎么着手干

这类跟项目相关的综合性问题，既能体现候选人的技术水平、业务水平和架构能力，也能够辨别候选人是不是真的做过项目，还能够发现候选人的一些软技能。

下面分享下，遇见这类问题应该怎样回答。

#怎样介绍自己做过的一个项目

按照第 1 小节说的，简历当中的项目，你要精挑细选，既要体现技术难度，又要想好细节。具体要介绍一个项目（包括梳理一个项目），可以按照下面几个阶段来做。

1. 介绍项目背景

这个项目为什么做，当初大的环境背景是什么？还是为了解决一个什么问题而设立的项目？背景是很重要的，如果不了解背景，一上来就听一个结论性的项目，面试官可能对于项目的技术选型、技术难度会有理解偏差，甚至怀疑是否真的有过这样的项目。

比如一上来就说：我们的项目采用了「backbone」来做框架，然后。。。而「backbone」已经是三四年以前比较新鲜的技术，现在会有更好的选择方案，如果不介绍项目的时间背景，面试官肯定一脸懵逼。

2. 承担角色

项目涉及的人员角色有哪些，自己在其中扮演的角色是什么？

》这里候选往往人会自己给自己挖坑，比如把自己在项目中起到的作用夸大等。一般来说，面试官细节追问的时候，如果候选人能够把细节或者技术方案等讲明白、讲清楚，不管他是真的做过还是跟别人做过，或者自己认真思考过，都能体现候选人的技术水平和技术视野。前提还是在你能够兜得住的可控范围之内做适当的「美化」。

3. 最终的结果和收益

》项目介绍过程中，应该介绍项目最终的结果和收益，比如项目最后经过多久的开发上线了，上线后的数据是怎样的，是否达到预期，还是带来了新的问题，遇见了问题自己后续又是怎样补救的。

4. 有始有终：项目总结和反思

有总结和反思，才会有进步。项目做完了往往有一些心得和体会，这时候应该跟面试官说出来。在梳理项目的总结和反思时，可以按照下面的列表来梳理：

- 收获有哪些？
- 是否有做得不足的地方，怎么改进？

- 是否具有可迁移性？

比如，之前详细介绍了某个项目，这个项目当时看来没有什么问题，但是现在有更好的解决方案了，候选人就应该在这里提出来：现在看来，这个项目还有 xx 的问题，我可以通过 xx 的方式来解决。

再比如：做这个项目的时候，你做得比较出彩的地方，可以迁移到其他项目中直接使用，小到代码片段，大到解决方案，总会有你值得总结和梳理的地方。

介绍完项目总结这部分，也可以引导面试官往自己擅长的领域思考。比如上面提到项目中的问题，可以往你擅长的方面引导，即使面试官没有问到，你也介绍到了。

按照上面的四段体介绍项目，会让面试官感觉候选人有清晰的思路，对整个项目也有理解和想法，还能够总结反思项目的收益和问题，可谓「一箭三雕」。

#没有做过大型项目怎么办

- 对于刚刚找工作的应届生，或者面试官让你进行一个大型项目的设计，候选人可能没有类似的经验。这时候不要用「我不会、没做过」一句话就带过。
- 如果是实在没有项目可以说，那么可以提自己日常做的练手项目，或者看到一个解决方案的文章/书，提到的某个项目，抒发下自己的想法。

如果是对于面试官提出来需要你设计的项目/系统，可以按照下面几步思考：

1. 有没有遇见过类似的项目
2. 有没有读过类似解决方案的文章
3. 项目能不能拆解，拆解过程中能不能发现自己做过的项目可以用
4. 项目解决的问题是什么，这类问题有没有更好的解决方案

总之，切记不要一句「不知道、没做过」就放弃，每一次提问都是自己表现的机会。

#项目细节和技术点的追问

介绍项目的过程中，面试官可能会追问技术细节，所以我们在准备面试的时候，应该尽量把技术细节梳理清楚，技术细节包括：

1. 技术选型方案：当时做技术选型所面临的状况
2. 技术解决方案：最终确定某种技术方案的原因，比如：选择用 Vue 而没有用 React 是为什么？
3. 项目数据和收益
4. 项目中最难的地方
5. 遇见的坑：如使用某种框架遇见哪些坑

一般来说，做技术选型的时候需要考虑下面几个因素：

1. 时代：现在比较火的技术是什么，为什么火起来，解决了什么问题，能否用到我的项目中？
2. 团队：个人或者团队对某种技术的熟悉程度是怎样的，学习成本又是怎样的？
3. 业务需求：需求是怎样的，能否套用现在的成熟解决方案/库来快速解决？
4. 维护成本：一个解决方案的是否再能够 cover 住的范围之内？

在项目中遇見的数据和收益应该做好跟踪，保证数据的真实性和可信性。另外，遇見的坑可能是面试官问得比较多的，尤其现在比较火的一些技术（Vue、React、webpack），一般团队都在使用，所以一定要提前准备下。

#小结

- 本小节介绍了面试中关于项目类问题的回答方法，介绍项目要使用四段体的方式，从背景、承担角色、收益效果和总结反思四个部分来介绍项目。
- 准备这个面试环节的时候，利用笔者一直提倡的「思维导图」法，好好回顾和梳理自己的项目。

#十、HR 面：谈钱不伤感情

当你顺利通过面试，最后 HR 面试主要有两大环节：

1. 了解候选人是否在岗位、团队、公司文化等方面能够跟要求匹配，并且能够长期服务
2. 谈薪资

#匹配度考查

- 很多情况下 HR 并不懂技术，但是也会问你项目上的问题，这时候其实是考查候选人对自己所做项目和技术的掌握能力。「检验一个人是否掌握一个专业知识，看他能不能把专业知识通俗易懂地对一个外行讲明白」。在面对 HR 询问项目或者技术点的细节时，你应该尽量通俗易懂地将知识点讲明白。怎样做到通俗易懂？笔者建议多作类比，跟生活中常见的或者大家都明白的知识作对比。举个例子，讲解「减少 Cookie 对页面打开速度优化有效果」的时候，笔者会这样来介绍：

你应该知道平时上传文件（比如头像）要比下载文件慢，这是因为网络上行带宽要比下行带宽窄，HTTP 请求的时候其实是双向的，先上传本地的信息，包括要访问的网址、本地携带的一些 Cookie 之类的数据，这些数据因为是上行（从用户手中发到服务器，给服务器上的代码使用），本来上行带宽小，所以对速度的影响更大。因此在 HTTP 上行请求中，减少 Cookie 的大小等方式可以有效提高打开速度，尤其是在带宽不大的网络环境中，比如手机的 2G 弱网络环境。

- 如果他还是不太清楚，那么带宽、上行、下行这些概念都可以类比迅雷下载这个场景，一解释应该就明白了。
- HR 面试还会通过一些问题，判断你与公司文化是否契合，比如阿里的 HR 有政委体系，会严格考查候选人是否符合公司企业文化。针对这类问题应该在回答过程中体现出自己阳光正能量的一面，不要抱怨前公司，抱怨前领导，多从自身找原因和不足，谦虚谨慎。

#谈薪资

谈 offer 的环节并不轻松，包括笔者在内往往在这个阶段「折了兵」。不会谈 offer 往往会遇见这样的情况：

1. 给你多少就要多少，不会议价
2. 谈一次被打击一次，最后越来越没有底气

尤其是很多不专业的 HR 以压低工资待遇为自己的首要目标，把候选人打击得不行不行的。一个不够满意的 offer 往往导致入职后出现抱怨，本小节重点讲下如何谈到自己中意的 offer。

#准确定位和自我估值

- 在准备跳槽时，每个人肯定会对自己的一个预估，做好足够的心理准备。下面谈下怎么对自己的薪酬做个评估。一般来说跳槽的薪水是根据现在薪酬的基础上浮 15~30%，具体看个人面试的情况。对于应届毕业生，大公司基本都有标准薪水，同期的应届生差别不会特别大。
- 除了上面的方法，还应该按照公司的技术职级进行估值。每个公司都有对应的技术职级，不同的技术职级薪酬范围是固定的，如果是小公司，则可以参考大公司的职级范围来确定薪资范围。
- 根据职级薪资范围和自己现在薪酬基础上浮后的薪酬，做个比较，取其较高的结果。
- 当然如果面试结果很好，你可以适当地提高下薪酬预期。除了这种情况，应该针对不同的性质来对 offer 先做好不同的估值。这里的预期估值只是心理预期，也就是自己的「底牌」。

所谓不同性质的 offer 指的是：

1. 是否是自己真心喜欢的工作岗位：如果是自己真心喜欢的工作岗位，比如对于个人成长有利，或者希望进入某个公司部门，从事某个专业方向的工作，而你自己对于薪酬又不是特别在意，这时候可以适当调低薪酬预期，以拿到这个工作机会为主。

2. 是否只是做 backup 的岗位：面试可能不止面试一家，对于不是特别喜欢的公司部门，那么可以把这个 offer 做为 backup，后面遇见喜欢的公司可以以此基础来谈薪水。
- 这时候分两种情况：如果面试结果不是很好，这种情况应该优先拿到 offer，所以可以适当降低期望薪酬；如果面试结果很好，这种情况应该多要一些薪酬，增加的薪酬可以让你加入这家公司也心里很舒服。
 - 对于自己真正的目标职位，面试之前应该先找 backup 岗位练练手，一是为了找出面试的感觉，二是为了拿到几个 offer 做好 backup。

关于如何客观评估自己的身价，有篇知乎的帖子比较专业，有时间可以读一下：[如何在跳槽前客观地评估自己的身价？](#)

#跟 HR 沟通的技巧

跟 HR 沟通的时候，不要夸大现在的薪酬，HR 知道的信息往往会超出你的认知，尤其大公司还会有背景调查，所以不要撒谎，实事求是。

跟 HR 沟通的技巧有以下几点：

1. 不要急于出价

- 不要急于亮出自己的底牌，一旦你说出一个薪酬范围，自己就不能增加薪酬了，还给了对方砍价的空间。而且一个不合理的价格反而会让对方直接放弃。所以不要着急出价，先让对方出价。
- 同时，对于公司级别也是，不要一开始就奔着某个目标去面试，这样会加大面试的难度，比如：

目标是拿到阿里 P7 的职位，不要说不给 P7 我就不去面试之类的，这样的要求会让对方一开始就拿 P7 的标准来面试，可能会找 P8+ 的面试官来面试你，这样会大大提升面试难度。

2. 要有底气足够自信

要有底气，自信，自己按照上面的估值盘算好了想要的薪酬，那么应该有底气地说出来，并且给出具体的原因，比如：

- 我已经对贵公司的薪酬范围和级别有了大概的了解，我现在的水平大概范围是多少
- 现在公司很快就有调薪机会，自己已经很久没有调薪，年前跳槽会损失年终奖等情况
- 现在我已经有某个公司多少 K 的 offer

如果 HR 表示你想要的薪酬不能满足，这时候你应该给出自己评估的依据，是根据行业职级标准还是自己现有薪酬范围，这样做到有理有据。

3. 谈好 offer 就要尽快落实

对于已经谈拢的薪酬待遇，一定要 HR 以发邮件 offer 的形式来确认。

#十一、其他：面试注意事项

除了前面小节中提到的一些面试中应该注意的问题，本小节再整理一些面试中应该注意的事项。

#注意社交礼仪

虽然说 IT 行业不怎么注重工作环境，上下级也没有繁文缛节，但是在面试中还是应该注意一些社交礼仪的。像进门敲门、出门关门、站着迎人这类基本礼仪还是要做的。

#舒适但不随意的着装

首先着装方面，不要太随意，也不要太正式，太正式的衣服可能会使人紧张，所以建议穿自己平时喜欢的衣服，关键是干净整洁。

#约个双方都舒服的面试时间

如果 HR 打电话预约面试时间，记得一定要约个双方都舒服的时间，宁愿请假也要安排好面试时间。

有个case：前几天有个朋友说为了给公司招人，晚上住公司附近酒店，原因是候选人为了不耽误现在公司的工作，想在 10 点之前按时上班，预约的面试时间是早上 8 点。这样对于面试官来说增加了负担，心里肯定不会特别舒服，可能会影响候选人的面试结果。

面试时间很重要，**提前十分钟到面试地点**，熟悉下环境，做个登记之类的，留下个守时的好印象。如果因为堵车之类的原因不能按时到达，则要在约定时间之前电话通知对方。

#面试后的提问环节

面试是一个双向选择的事情，所以面试后一般会有提问环节。在提问环节，候选人最好不要什么都不问，更不要只问薪水待遇、是否加班之类的问题。

其实这个时候可以反问面试官了解团队情况、团队做的业务、本职位具体做的工作、工作的规划，甚至一些数据（可能有些问题不会直面回答）。

还可以问一些关于公司培训机会和晋升机会之类的问题。如果是一些高端职位，则可以问一下：自己的 leader 想把这个职位安排给什么样的人，希望多久的时间内可以达到怎样的水平。

#面试禁忌

- 不要对老东家有太多埋怨和负面评价
- 不要有太多负面情绪，多表现自己阳光的一面
- 不要夸大其词，尤其是数据方面
- 不要贬低任何人，包括自己之前的同事，比如有人喜欢说自己周围同事多么的差劲，来突出自己的优秀
- 不要过多争辩。你是来展现自己胜任能力的，不是来证明面试官很蠢的

#最好自己带电脑

有些面试会让候选人直接笔试，或者直接去小黑板上面画图写代码，这种笔试的时候会非常痛苦，我经常见单词拼写错误的候选人，这种情况最好是自己带着电脑，直接在自己熟悉的 IDE 上面编写。

这里应该注意，自己带电脑可能也有弊端。如果你对自己的开发环境和电脑足够熟悉，操作起来能够得心应手，那么可以带着；如果你本身电脑操作就慢，比如 Linux 命令不熟悉，打开命令行忘记了命令，这种情况下会被减分。带与不带自己根据自己情况权衡。

#面试后的总结和思考

- 面试完了多总结自己哪里做得不好，哪里做得好，都记录下来，后续扬长避短
- 通过面试肯定亲身体会到了公司团队文化、面试官体现出来的技术能力、专业性以及职位将来所做的事情，跟自己预期是否有差距，多个 offer 的话多做对比

每次面试应该都有所收获，毕竟花费了时间和精力。即使面上上也可以知道自己哪方面做得不好，继续加强。

#十二、总结

#准备阶段

简历准备：

1. 简历要求尽量平实，不要太花俏
2. 格式推荐 PDF
3. 内容包含：个人技能、项目经验和实习经验
4. 简历应该针对性来写
5. 简历提到的项目、技能都要仔细回想细节，挖掘可能出现的面试题

拿到面邀之后准备：

1. 开场问题：自我介绍、离职原因等
2. 了解面试官、了解公司和部门做的事情
3. 知识梳理推荐使用思维导图

#技术面部分

集中梳理了 ECMAScript 基础、JS-Web-API、CSS 和 HTML、算法、浏览器和开发环境六大部分内容，并且就一些高频考题进行讲解。

#非技术面试部分

主要从软技能和项目介绍两个部分来梳理。在软技能方面，介绍了工程师从业人员应该具有的软技能，并且通过几个面试真题介绍了怎么灵活应对面试官；在项目介绍小节，推荐按照项目背景、承担角色、项目收益和项目总结反思四步来介绍，并且继续推荐使用思维导图方式来梳理项目的细节。

#HR 面

在最后，介绍了 HR 面试应该注意的问题，重点分享了作为一个 Web 前端工程师怎么对自己进行估值，然后跟 HR 进行沟通，拿到自己可以接受的 offer。

最后还介绍了一些面试注意事项，在面试整个流程中，太多主观因素，细节虽小也可能决定候选人面试的结果。

#补充说明

本着通用性和面试门槛考虑的设计，对于一些前端进阶和框架类的问题没有进行梳理，没有涉及的内容主要有：

1. `Node.js` 部分
2. 类库：`Zepto`、`jQuery`、`React`、`Vue` 和 `Angular` 等
3. 移动开发

下面简单展开下上面的内容。

#Node.js部分

Node.js 涉及的知识点比较多，而且比较偏后端和工具性，如果用 Node.js 来做 Server 服务，需要补充大量的后端知识和运维知识，这里帮助梳理下知识点：

- `Node`

开发环境

- `npm` 操作

- `package.json`
- Node 基础 API 考查
 - `file system`
 - `Event`
 - 网络
 - `child process`
- Node

重点和难点

- 事件和异步理解
- `Stream` 相关概念
- `Buffer` 相关概念
- `domain`
- `vm`
- `cluster`
- 异常调优
- Server

相关

- 库
 - `Koa`
 - `Express`
- 数据库
 - `MongoDB`
 - `MySQL`
 - `Redis`
- 运维部署
 - `Nginx`
 - 进程守候
 - 日志

Node 的出现让前端可以做的事情更多，除了做一些 Server 的工作以外，Node 在日常开发中可以做一些工具来提升效率，比如常见的前端构建工具目前都是 Node 来编写的，而我们在研发中，一些类似 Mock、本地 server、代码实时刷新之类的功能，都可以使用 Node 来自己实现。

#前端框架（库）

jQuery 和 Zepto 分别是应用在 PC 和移动上面的库，大大降低了前端开发人员的门槛，很多前端工程师都是从写 jQuery 代码开始的。jQuery 和 Zepto 这两个库对外的 API 都是相同的。在面试的时候可能会问到一些具体代码的实现，比如下面两个问题：

题目：谈谈 jQuery 的 `delegate` 和 `bind` 有什么区别；`window.onload` 和 `$(document).ready` 有什么区别

这实际上都是 JS-Web-API 部分基础知识的实际应用：

- `delegate` 是事件代理（委托），`bind` 是直接绑定事件
- `onload` 是浏览器部分的全部加载完成，包括页面的图片之类资源；`ready` 则是 `DOMContentLoaded` 事件，比 `onload` 提前一些

下面再说下比较火的 Angular、React 和 Vue。

为什么会出现 Angular、React 和 Vue 这种库？

- 理解为什么会出现一种新技术，以及新技术解决了什么问题，才能够更好地选择和运用新技术，不至于落入「喜新厌旧」的怪圈。
- 首先在互联网用户界面和交互越来越复杂的阶段，这些 `MV*` 库是极大提升了开发效率，比如在数据流为主的后台系统，每天打交道最多的就是数据的增删改查，这时候如果使用这些库，可以将注意力转移到数据本身来，而不再是页面交互，从而极大地提升开发效率和沟通成本。
- React 还有个很好的想法是 React Native，只需要写一套代码就可以实现 Web、安卓、iOS 三端相同的效果，但是在实际使用和开发中会有比较大的坑。而且就像 Node 一样，前端用 Node 写 Server 可能需要用到的后端知识要比前端知识多，想要写好 React Native，客户端的知识也是必不可少的。React Native 和 Node 都是拓展了 Web 前端工程师可以走的路，既可以向后又可以向前，所谓「全栈」。

Angular、React 和 Vue 各自的特点

- AngularJS 有着诸多特性，最为核心的是 MVVM、模块化、自动化双向数据绑定、语义化标签、依赖注入等
- React 是一个为数据提供渲染为 HTML 视图的开源 JavaScript 库，最大特点是引入 Virtual DOM，极大提升数据修改后 DOM 树的更新速度，而且也有 React Native 来做客户端开发
- Vue.js 作为后起前端框架，借鉴了 Angular、React 等现代前端框架/库的诸多特点，并取得了相当不错的成绩。

一定要用这些库吗？

- 目前这些库的确解决了实际开发中很多问题，但是这种「三足鼎立」的状况不是最终态，会是阶段性产物。从长远来说，好的想法和点子终究会体现在语言本身特性上来，即通过这些库的想法来推动标准的改进，比如 jQuery 的很多选择器 API，最终都被 CSS3 和 HTML5 接纳和实现，也就有了后来的 Zepto。
- 另外，以展现交互为主的项目 **不太推荐** 使用这类库，本身库的性能和体积就对页面造成极大的负担，比如笔者使用 Vue 做纯展现为主的项目，性能要比页面直出 HTML 慢。纯展现页面指的是那些以展现为主、用户交互少的页面，如文章列表页、文章详情页等。
- 如果是数据交互较多的页面，例如后台系统这类对性能要求不多而数据交互较多的页面，**推荐使用**。
- 另外，不管是什库和框架，我们最终应该学习的是编程思维，比如分层、性能优化等，考虑视图层、组件化和工程效率问题。相信随着 ES 标准发展、摩尔定律（硬件）和浏览器的演进，目前这些问题和状况都会得到改善。
- 关于三者的学习资料就不补充了，因为实在是太火了，随便搜索一下就会找到。

#移动开发

这里说的移动开发指的是做的项目是面向移动端的，比如 `HTML5` 页面、小程序等。做移动开发用的也是前面几个小节梳理的基础知识，唯一不同的是工程师面向的浏览器是移动端的浏览器或者固定的 `WebView`，所以会跟普通的 `PC` 开发有所不同。除了最基础的 `JSBridge` 概念之外，这里笔者重点列出以下几点：

- 移动端更加注重性能和体验，因为移动端设备和网络都比 `PC` 的差一些
- 交互跟 `PC` 不同，比如 `touch` 事件
- 浏览器和固定的 `WebView` 带来了更多兼容性的问题，如微信 `WebView`、安卓浏览器和 `iOS` 浏览器
- 调试技巧更多，在 `Chrome` 内开发完页面，放到真机需要再调试一遍，或者需要真机配合才能实现页面的完整功能

#一、网络

#1 UDP

1.1 面向报文

UDP 是一个面向报文（报文可以理解为一段段的数据）的协议。意思就是 UDP 只是报文的搬运工，不会对报文进行任何拆分和拼接操作

具体来说

- 在发送端，应用层将数据传递给传输层的 UDP 协议，UDP 只会给数据增加一个 UDP 头标识下是 UDP 协议，然后就传递给网络层了
- 在接收端，网络层将数据传递给传输层，UDP 只去除 IP 报文头就传递给应用层，不会任何拼接操作

1.2 不可靠性

- UDP 是无连接的，也就是说通信不需要建立和断开连接。
- UDP 也是不可靠的。协议收到什么数据就传递什么数据，并且也不会备份数据，对方能不能收到是不关心的
- UDP 没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP

1.3 高效

- 因为 UDP 没有 TCP 那么复杂，需要保证数据不丢失且有序到达。所以 UDP 的头部开销小，只有八字节，相比 TCP 的至少二十字节要少得多，在传输数据报文时是很高效的

头部包含了以下几个数据

- 两个十六位的端口号，分别为源端口（可选字段）和目标端口 整个数据报文的长度
- 整个数据报文的检验和（IPv4 可选字段），该字段用于发现头部信息和数据中的错误

1.4 传输方式

UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式，也就是说 UDP 提供了单播，多播，广播的功能

#2 TCP

2.1 头部

TCP 头部比 UDP 头部复杂的多

对于 TCP 头部来说，以下几个字段是很重要的

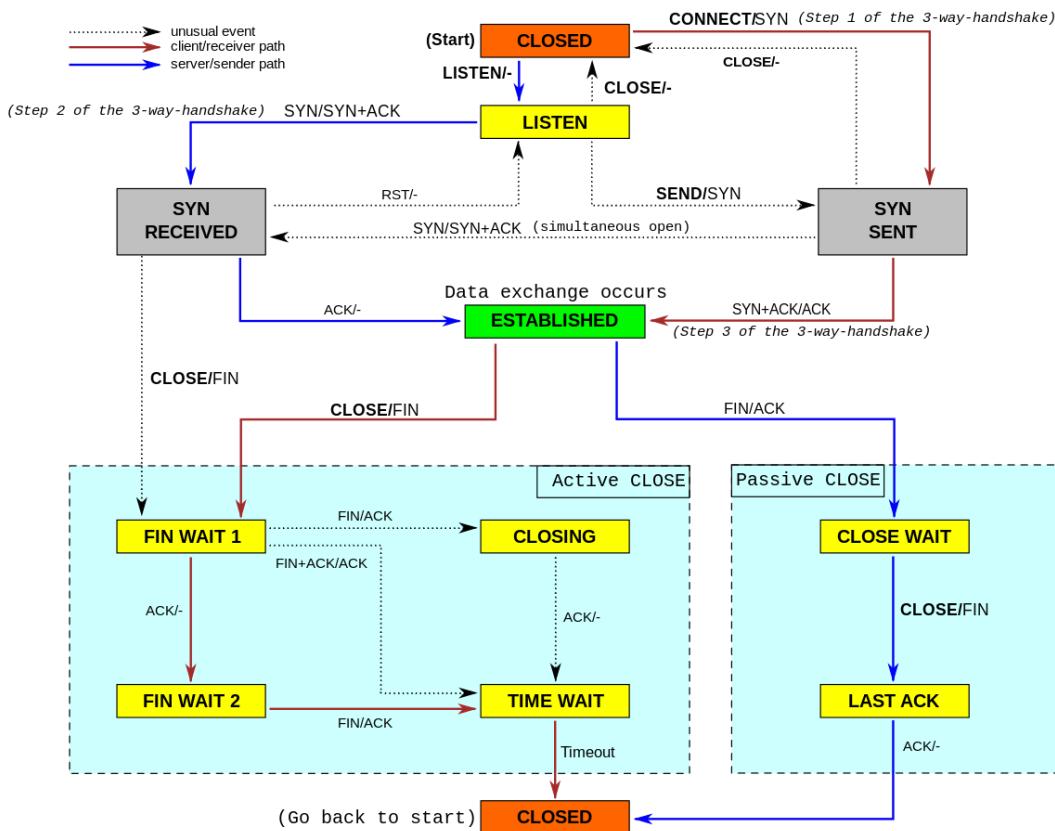
- Sequence number，这个序号保证了 TCP 传输的报文都是有序的，对端可以通过序号顺序的拼接报文
- Acknowledgement Number，这个序号表示数据接收端期望接收的下一个字节的编号是多少，同时也表示上一个序号的数据已经收到
- window size，窗口大小，表示还能接收多少字节的数据，用于流量控制

标识符

- `URG=1`：该字段为一表示本数据报的数据部分包含紧急信息，是一个高优先级数据报文，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。
- `ACK=1`：该字段为一表示确认号字段有效。此外，`TCP` 还规定在连接建立后传送的所有报文段都必须把 `ACK` 置为一 `PSH=1`：该字段为一表示接收端应该立即将数据 push 给应用层，而不是等到缓冲区满后再提交。
- `RST=1`：该字段为一表示当前 `TCP` 连接出现严重问题，可能需要重新建立 `TCP` 连接，也可以用于拒绝非法的报文段和拒绝连接请求。
- `SYN=1`：当 `SYN=1`，`ACK=0` 时，表示当前报文段是一个连接请求报文。当 `SYN=1`，`ACK=1` 时，表示当前报文段是一个同意建立连接的应答报文。
- `FIN=1`：该字段为一表示此报文段是一个释放连接的请求报文

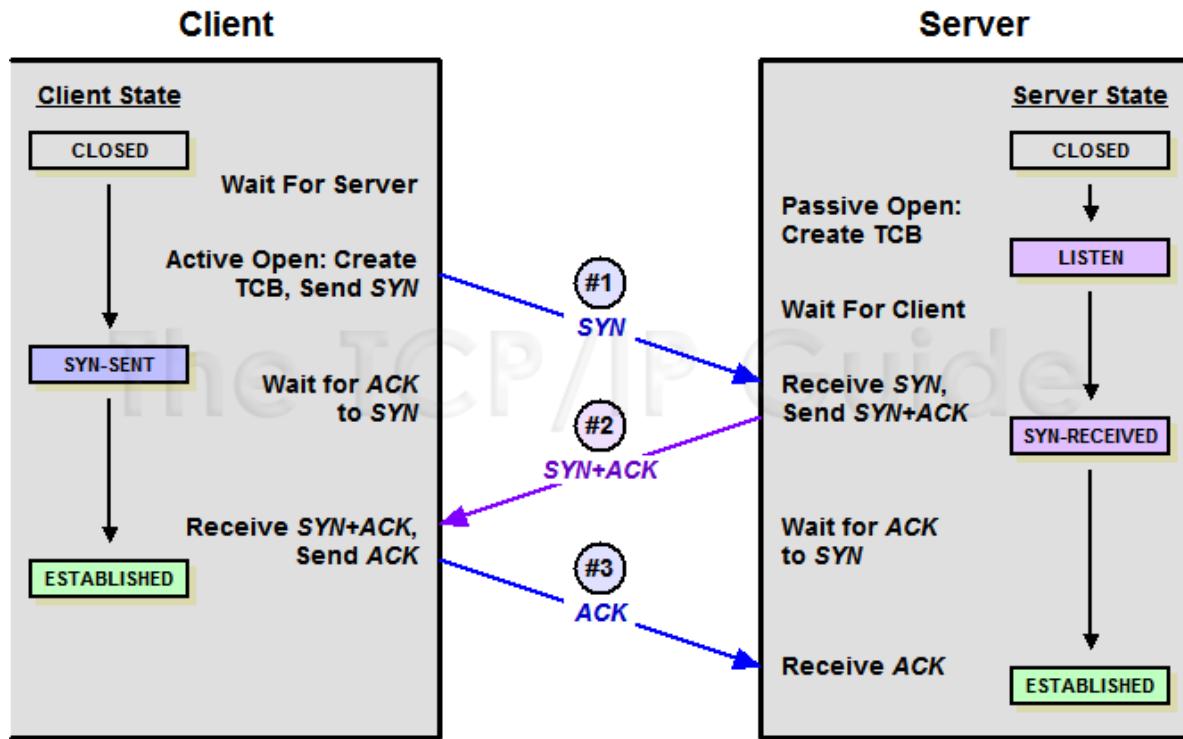
2.2 状态机

`HTTP` 是无连接的，所以作为下层的 `TCP` 协议也是无连接的，虽然看似 `TCP` 将两端连接了起来，但是其实只是两端共同维护了一个状态



- `TCP` 的状态机是很复杂的，并且与建立断开连接时的握手息息相关，接下来就来详细描述下两种握手。
- 在这之前需要了解一个重要的性能指标 RTT。该指标表示发送端发送数据到接收到对端数据所需的往返时间

建立连接三次握手



- 在 `TCP` 协议中，主动发起请求的一端为客户端，被动连接的一端称为服务端。不管是客户端还是服务端，`TCP` 连接建立完后都能发送和接收数据，所以 `TCP` 也是一个全双工的协议。
- 起初，两端都为 `CLOSED` 状态。在通信开始前，双方都会创建 `TCB`。服务器创建完 `TCB` 后遍进入 `LISTEN` 状态，此时开始等待客户端发送数据

第一次握手

客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 `SYN-SENT` 状态， x 表示客户端的数据通信初始序号。

第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 `SYN-RECEIVED` 状态。

第三次握手

当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 `ESTABLISHED` 状态，服务端收到这个应答后也进入 `ESTABLISHED` 状态，此时连接建立成功。

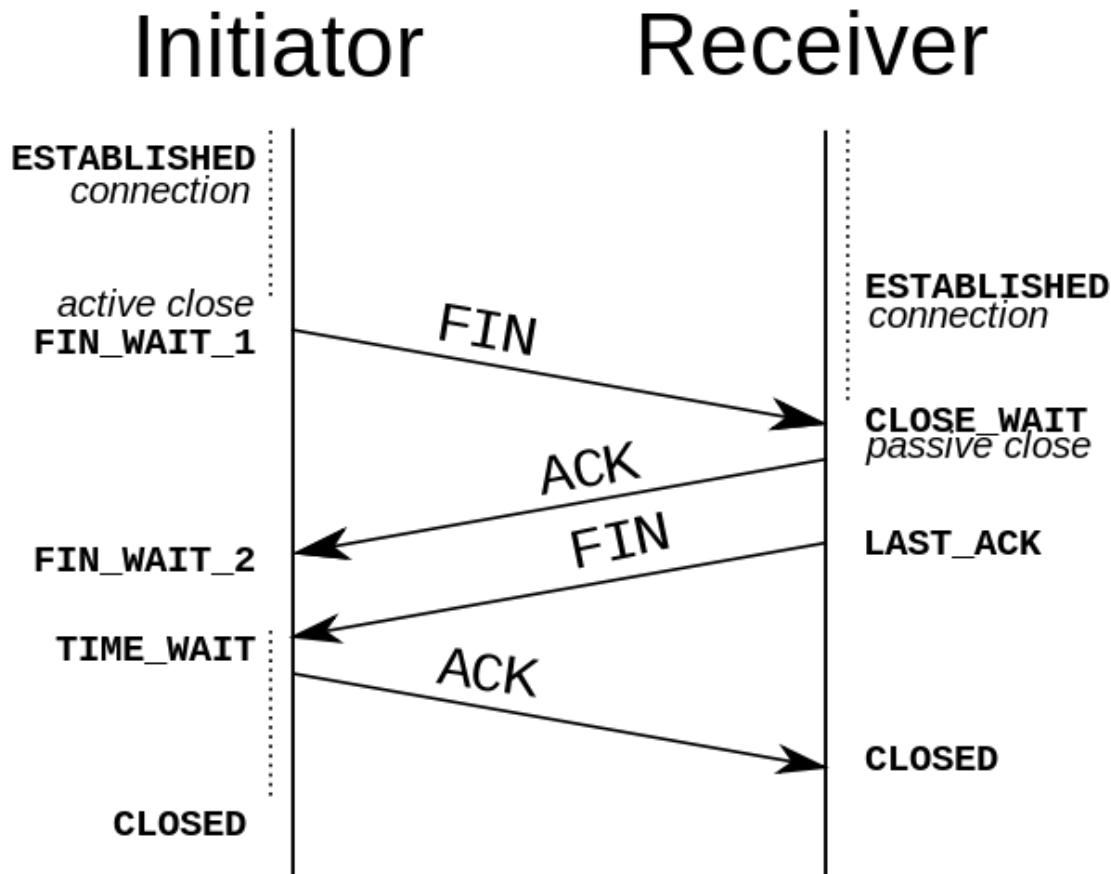
- PS：第三次握手可以包含数据，通过 `TCP` 快速打开 (`TFO`) 技术。其实只要涉及到握手的协议，都可以使用类似 `TFO` 的方式，客户端和服务端存储相同 `cookie`，下次握手时发出 `cookie` 达到减少 `RTT` 的目的

你是否有疑惑明明两次握手就可以建立起连接，为什么还需要第三次应答？

- 因为这是为了防止失效的连接请求报文段被服务端接收，从而产生错误

可以想象如下场景。客户端发送了一个连接请求 A，但是因为网络原因造成了超时，这时 `TCP` 会启动超时重传的机制再次发送一个连接请求 B。此时请求顺利到达服务端，服务端应答完就建立了请求。如果连接请求 A 在两端关闭后终于抵达了服务端，那么这时服务端会认为客户端又需要建立 `TCP` 连接，从而应答了该请求并进入 `ESTABLISHED` 状态。此时客户端其实是 `CLOSED` 状态，那么就会导致服务端一直等待，造成资源的浪费

PS：在建立连接中，任意一端掉线，`TCP` 都会重发 `SYN` 包，一般会重试五次，在建立连接中可能会遇到 `SYN FLOOD` 攻击。遇到这种情况你可以选择调低重试次数或者干脆在不能处理的情况下拒绝请求



TCP 是全双工的，在断开连接时两端都需要发送 **FIN** 和 **ACK**。

第一次握手

若客户端 A 认为数据发送完成，则它需要向服务端 B 发送连接释放请求。

第二次握手

B 收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入 **CLOSE_WAIT** 状态，表示 A 到 B 的连接已经释放，不接收 A 发的数据了。但是因为 TCP 连接时双向的，所以 B 仍旧可以发送数据给 A。

第三次握手

B 如果此时还有没发完的数据会继续发送，完毕后会向 A 发送连接释放请求，然后 B 便进入 **LAST-ACK** 状态。

PS：通过延迟确认的技术（通常有时间限制，否则对方会误认为需要重传），可以将第二次和第三次握手合并，延迟 ACK 包的发送。

第四次握手

- A 收到释放请求后，向 B 发送确认应答，此时 A 进入 **TIME-WAIT** 状态。该状态会持续 2MSL（最大段生存期，指报文在网络中生存的时间，超时会被抛弃）时间，若该时间段内没有 B 的重发请求的话，就进入 **CLOSED** 状态。当 B 收到确认应答后，也便进入 **CLOSED** 状态。

为什么 A 要进入 **TIME-WAIT** 状态，等待 2MSL 时间后才进入 **CLOSED** 状态？

- 为了保证 B 能收到 A 的确认应答。若 A 发完确认应答后直接进入 **CLOSED** 状态，如果确认应答因为网络问题一直没有到达，那么会造成 B 不能正常关闭

#3 HTTP

HTTP 协议是个无状态协议，不会保存状态

3.1 Post 和 Get 的区别

- Get 请求能缓存，Post 不能
- Post 相对 Get 安全一点点，因为 Get 请求都包含在 URL 里，且会被浏览器保存历史纪录，Post 不会，但是在抓包的情况下都是一样的。
- Post 可以通过 request body 来传输比 Get 更多的数据，Get 没有这个技术
- URL 有长度限制，会影响 Get 请求，但是这个长度限制是浏览器规定的，不是 RFC 规定的
- Post 支持更多的编码类型且不对数据类型限制

3.2 常见状态码

2XX 成功

- 200 OK，表示从客户端发来的请求在服务器端被正确处理
- 204 No content，表示请求成功，但响应报文不含实体的主体部分
- 205 Reset Content，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容
- 206 Partial Content，进行范围请求

3XX 重定向

- 301 moved permanently，永久性重定向，表示资源已被分配了新的 URL
- 302 found，临时性重定向，表示资源临时被分配了新的 URL
- 303 see other，表示资源存在着另一个 URL，应使用 GET 方法访问资源
- 304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况
- 307 temporary redirect，临时重定向，和302含义类似，但是期望客户端保持请求方法不变向新的地址发出请求

4XX 客户端错误

- 400 bad request，请求报文存在语法错误
- 401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
- 403 forbidden，表示对请求资源的访问被服务器拒绝
- 404 not found，表示在服务器上没有找到请求的资源

5XX 服务器错误

- 500 internal server error，表示服务器端在执行请求时发生了错误
- 501 Not Implemented，表示服务器不支持当前请求所需要的某个功能
- 503 service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求

3.3 HTTP 首部

| 通用字段 | 作用 |
|--------------------------------|---|
| <code>Cache-Control</code> | 控制缓存的行为 |
| <code>Connection</code> | 浏览器想要优先使用的连接类型，比如 <code>keep-alive</code> |
| <code>Date</code> | 创建报文时间 |
| <code>Pragma</code> | 报文指令 |
| <code>via</code> | 代理服务器相关信息 |
| <code>Transfer-Encoding</code> | 传输编码方式 |
| <code>Upgrade</code> | 要求客户端升级协议 |
| <code>Warning</code> | 在内容中可能存在错误 |

| 请求字段 | 作用 |
|----------------------------------|----------------------|
| <code>Accept</code> | 能正确接收的媒体类型 |
| <code>Accept-Charset</code> | 能正确接收的字符集 |
| <code>Accept-Encoding</code> | 能正确接收的编码格式列表 |
| <code>Accept-Language</code> | 能正确接收的语言列表 |
| <code>Expect</code> | 期待服务端的指定行为 |
| <code>From</code> | 请求方邮箱地址 |
| <code>Host</code> | 服务器的域名 |
| <code>If-Match</code> | 两端资源标记比较 |
| <code>If-Modified-Since</code> | 本地资源未修改返回 304 (比较时间) |
| <code>If-None-Match</code> | 本地资源未修改返回 304 (比较标记) |
| <code>User-Agent</code> | 客户端信息 |
| <code>Max-Forwards</code> | 限制可被代理及网关转发的次数 |
| <code>Proxy-Authorization</code> | 向代理服务器发送验证信息 |
| <code>Range</code> | 请求某个内容的一部分 |
| <code>Referer</code> | 表示浏览器所访问的前一个页面 |
| <code>TE</code> | 传输编码方式 |

| 响应字段 | 作用 |
|--------------------|---------------|
| Accept-Ranges | 是否支持某些种类的范围 |
| Age | 资源在代理缓存中存在的时间 |
| ETag | 资源标识 |
| Location | 客户端重定向到某个 URL |
| Proxy-Authenticate | 向代理服务器发送验证信息 |
| Server | 服务器名字 |
| WWW-Authenticate | 获取资源需要的验证信息 |

| 实体字段 | 作用 |
|------------------|------------------------|
| Allow | 资源的正确请求方式 |
| Content-Encoding | 内容的编码格式 |
| Content-Language | 内容使用的语言 |
| Content-Length | request body 长度 |
| Content-Location | 返回数据的备用地址 |
| Content-MD5 | Base64 加密格式的内容 MD5 检验值 |
| Content-Range | 内容的位置范围 |
| Content-Type | 内容的媒体类型 |
| Expires | 内容的过期时间 |
| Last_modified | 内容的最后修改时间 |

#4 DNS

DNS 的作用就是通过域名查询到具体的 IP。

- 因为 IP 存在数字和英文的组合 (IPv6)，很不利于人类记忆，所以就出现了域名。你可以把域名看成是某个 IP 的别名，DNS 就是去查询这个别名的真正名称是什么

在 TCP 握手之前就已经进行了 DNS 查询，这个查询是操作系统自己做的。当你在浏览器中想访问 www.google.com 时，会进行一下操作

- 操作系统会首先在本地缓存中查询
- 没有的话会去系统配置的 DNS 服务器中查询
- 如果这时候还没得话，会直接去 DNS 根服务器查询，这一步查询会找出负责 com 这个一级域名的服务器
- 然后去该服务器查询 google 这个二级域名
- 接下来三级域名的查询其实是我们配置的，你可以给 www 这个域名配置一个 IP，然后还可以给别的三级域名配置一个 IP

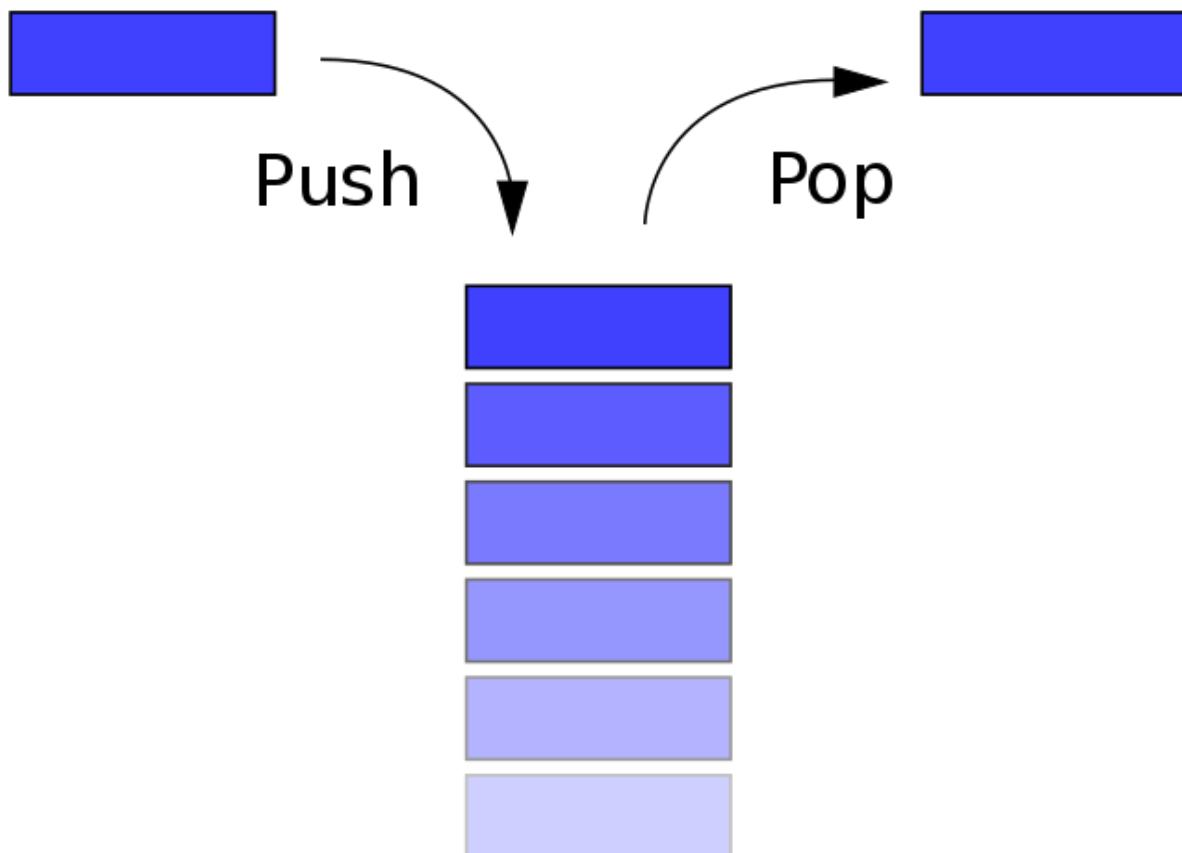
以上介绍的是 DNS 迭代查询，还有种是递归查询，区别就是前者是由客户端去做请求，后者是由系统配置的 DNS 服务器做请求，得到结果后将数据返回给客户端。

#二、数据结构

#2.1 栈

概念

- 栈是一个线性结构，在计算机中是一个相当常见的数据结构。
- 栈的特点是只能在某一端添加或删除数据，遵循先进后出的原则



实现

每种数据结构都可以用很多种方式来实现，其实可以把栈看成是数组的一个子集，所以这里使用数组来实现

```
class Stack {  
    constructor() {  
        this.stack = []  
    }  
    push(item) {  
        this.stack.push(item)  
    }  
    pop() {  
        this.stack.pop()  
    }  
    peek() {  
        return this.stack[this.getCount() - 1]  
    }  
    getCount() {  
        return this.stack.length  
    }  
    isEmpty() {  
        return this.stack.length === 0  
    }  
}
```

```
        return this.getCount() === 0
    }
}
```

应用

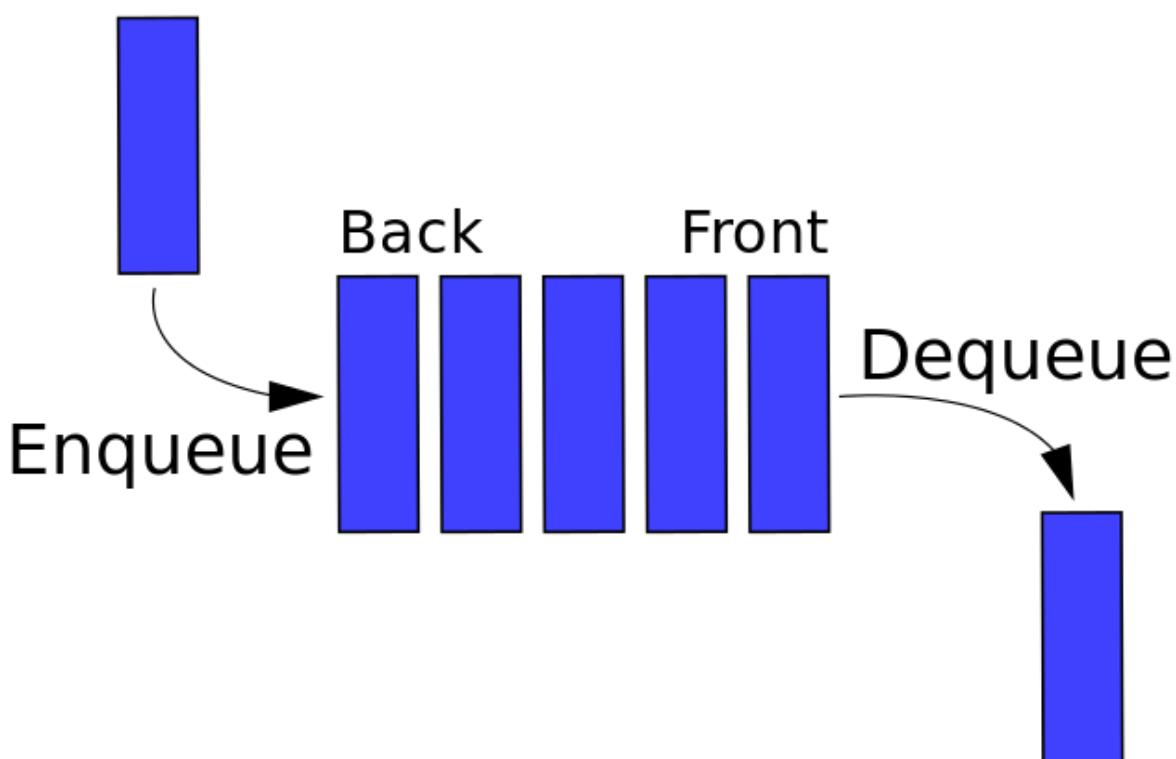
匹配括号，可以通过栈的特性来完成

```
var isValid = function (s) {
    let map = {
        '(': -1,
        ')': 1,
        '[': -2,
        ']': 2,
        '{': -3,
        '}': 3
    }
    let stack = []
    for (let i = 0; i < s.length; i++) {
        if (map[s[i]] < 0) {
            stack.push(s[i])
        } else {
            let last = stack.pop()
            if (map[last] + map[s[i]] != 0) return false
        }
    }
    if (stack.length > 0) return false
    return true
};
```

#2.2 队列

概念

队列一个线性结构，特点是在某端添加数据，在另一端删除数据，遵循先进先出的原则



实现

这里会讲解两种实现队列的方式，分别是单链队列和循环队列

- **单链队列**

```
class Queue {  
    constructor() {  
        this.queue = []  
    }  
    enqueue(item) {  
        this.queue.push(item)  
    }  
    dequeue() {  
        return this.queue.shift()  
    }  
    getHeader() {  
        return this.queue[0]  
    }  
    getLength() {  
        return this.queue.length  
    }  
    isEmpty() {  
        return this.getLength() === 0  
    }  
}
```

因为单链队列在出队操作的时候需要 $O(n)$ 的时间复杂度，所以引入了循环队列。循环队列的出队操作平均是 $O(1)$ 的时间复杂度

- **循环队列**

```
class SqQueue {  
    constructor(length) {  
        this.queue = new Array(length + 1)  
        // 队头  
        this.first = 0  
        // 队尾  
        this.last = 0  
        // 当前队列大小  
        this.size = 0  
    }  
    enqueue(item) {  
        // 判断队尾 + 1 是否为队头  
        // 如果是就代表需要扩容数组  
        // % this.queue.length 是为了防止数组越界  
        if (this.first === (this.last + 1) % this.queue.length) {  
            this.resize(this.getLength() * 2 + 1)  
        }  
        this.queue[this.last] = item  
        this.size++  
        this.last = (this.last + 1) % this.queue.length  
    }  
    dequeue() {  
        if (this.isEmpty()) {  
            throw Error('Queue is empty')  
        }  
    }  
}
```

```

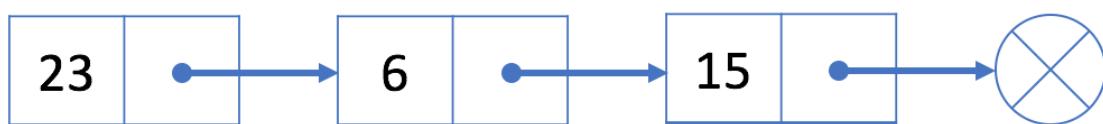
        let r = this.queue[this.first]
        this.queue[this.first] = null
        this.first = (this.first + 1) % this.queue.length
        this.size--
        // 判断当前队列大小是否过小
        // 为了保证不浪费空间，在队列空间等于总长度四分之一时
        // 且不为 2 时缩小总长度为当前的一半
        if (this.size === this.getLength() / 4 && this.getLength() / 2 !== 0) {
            this.resize(this.getLength() / 2)
        }
        return r
    }
    getHeader() {
        if (this.isEmpty()) {
            throw Error('Queue is empty')
        }
        return this.queue[this.first]
    }
    getLength() {
        return this.queue.length - 1
    }
    isEmpty() {
        return this.first === this.last
    }
    resize(length) {
        let q = new Array(length)
        for (let i = 0; i < length; i++) {
            q[i] = this.queue[(i + this.first) % this.queue.length]
        }
        this.queue = q
        this.first = 0
        this.last = this.size
    }
}

```

#2.3 链表

概念

链表是一个线性结构，同时也是一个天然的递归结构。链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大



实现

- 单向链表

```

class Node {
    constructor(v, next) {
        this.value = v
        this.next = next
    }
}

```

```
}

}

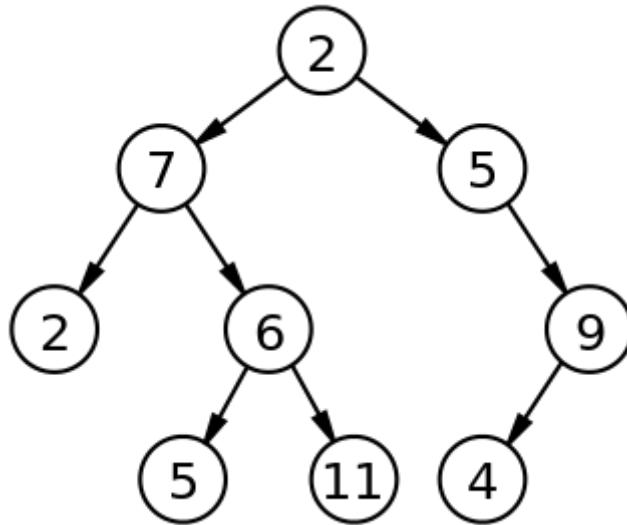
class LinkList {
    constructor() {
        // 链表长度
        this.size = 0
        // 虚拟头部
        this.dummyNode = new Node(null, null)
    }
    find(header, index, currentIndex) {
        if (index === currentIndex) return header
        return this.find(header.next, index, currentIndex + 1)
    }
    addNode(v, index) {
        this.checkIndex(index)
        // 当往链表末尾插入时, prev.next 为空
        // 其他情况时, 因为要插入节点, 所以插入的节点
        // 的 next 应该是 prev.next
        // 然后设置 prev.next 为插入的节点
        let prev = this.find(this.dummyNode, index, 0)
        prev.next = new Node(v, prev.next)
        this.size++
        return prev.next
    }
    insertNode(v, index) {
        return this.addNode(v, index)
    }
    addToFirst(v) {
        return this.addNode(v, 0)
    }
    addToLast(v) {
        return this.addNode(v, this.size)
    }
    removeNode(index, isLast) {
        this.checkIndex(index)
        index = isLast ? index - 1 : index
        let prev = this.find(this.dummyNode, index, 0)
        let node = prev.next
        prev.next = node.next
        node.next = null
        this.size--
        return node
    }
    removeFirstNode() {
        return this.removeNode(0)
    }
    removeLastNode() {
        return this.removeNode(this.size, true)
    }
    checkIndex(index) {
        if (index < 0 || index > this.size) throw Error('Index error')
    }
    getNode(index) {
        this.checkIndex(index)
        if (this.isEmpty()) return
        return this.find(this.dummyNode, index, 0).next
    }
    isEmpty() {
```

```
    return this.size === 0
}
getSize() {
    return this.size
}
}
```

#2.4 树

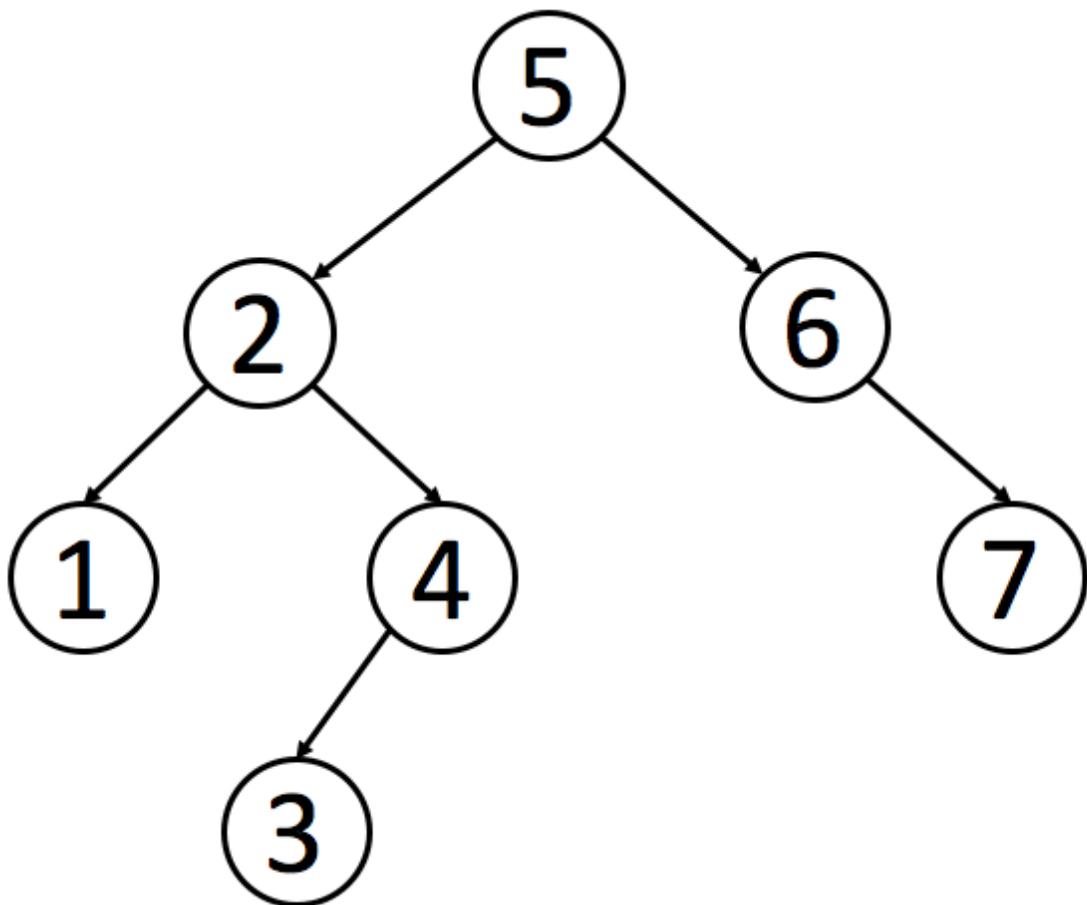
二叉树

- 树拥有很多种结构，二叉树是树中最常用的结构，同时也是一个天然的递归结构。
- 二叉树拥有一个根节点，每个节点至多拥有两个子节点，分别为：左节点和右节点。树的最底部节点称之为叶节点，当一颗树的叶数量数量为满时，该树可以称之为满二叉树



二分搜索树

- 二分搜索树也是二叉树，拥有二叉树的特性。但是区别在于二分搜索树每个节点的值都比他的左子树的值大，比右子树的值小
- 这种存储方式很适合于数据搜索。如下图所示，当需要查找 6 的时候，因为需要查找的值比根节点的值大，所以只需要在根节点的右子树上寻找，大大提高了搜索效率



- 实现

```

class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}

class BST {
  constructor() {
    this.root = null
    this.size = 0
  }

  getSize() {
    return this.size
  }

  isEmpty() {
    return this.size === 0
  }

  addNode(v) {
    this.root = this._addChild(this.root, v)
  }

  // 添加节点时，需要比较添加的节点值和当前
  // 节点值的大小
  _addChild(node, v) {
    if (!node) {
      this.size++
      return new Node(v)
    }
  }
}
  
```

```

        if (node.value > v) {
            node.left = this._addChild(node.left, v)
        } else if (node.value < v) {
            node.right = this._addChild(node.right, v)
        }
        return node
    }
}

```

- 以上是最基本的二分搜索树实现，接下来实现树的遍历。

对于树的遍历来说，有三种遍历方法，分别是先序遍历、中序遍历、后序遍历。三种遍历的区别在于何时访问节点。在遍历树的过程中，每个节点都会遍历三次，分别是遍历到自己，遍历左子树和遍历右子树。如果需要实现先序遍历，那么只需要第一次遍历到节点时进行操作即可

```

// 先序遍历可用于打印树的结构
// 先序遍历先访问根节点，然后访问左节点，最后访问右节点。
preTraversal() {
    this._pre(this.root)
}

_pre(node) {
    if (node) {
        console.log(node.value)
        this._pre(node.left)
        this._pre(node.right)
    }
}

// 中序遍历可用于排序
// 对于 BST 来说，中序遍历可以实现一次遍历就
// 得到有序的值
// 中序遍历表示先访问左节点，然后访问根节点，最后访问右节点。
midTraversal() {
    this._mid(this.root)
}

_mid(node) {
    if (node) {
        this._mid(node.left)
        console.log(node.value)
        this._mid(node.right)
    }
}

// 后序遍历可用于先操作子节点
// 再操作父节点的场景
// 后序遍历表示先访问左节点，然后访问右节点，最后访问根节点。
backTraversal() {
    this._back(this.root)
}

_back(node) {
    if (node) {
        this._back(node.left)
        this._back(node.right)
        console.log(node.value)
    }
}

```

以上的这几种遍历都可以称之为深度遍历，对应的还有种遍历叫做广度遍历，也就是一层层地遍历树。对于广度遍历来说，我们需要利用之前讲过的队列结构来完成

```

breadthTraversal() {
    if (!this.root) return null
    let q = new Queue()
    // 将根节点入队
    q.enqueue(this.root)
    // 循环判断队列是否为空，为空
    // 代表树遍历完毕
    while (!q.isEmpty()) {
        // 将队首出队，判断是否有左右子树
        // 有的话，就先左后右入队
        let n = q.dequeue()
        console.log(n.value)
        if (n.left) q.enqueue(n.left)
        if (n.right) q.enqueue(n.right)
    }
}

```

接下来先介绍如何在树中寻找最小值或最大数。因为二分搜索树的特性，所以最小值一定在根节点的最左边，最大值相反

```

getMin() {
    return this._getMin(this.root).value
}
_getMin(node) {
    if (!node.left) return node
    return this._getMin(node.left)
}
getMax() {
    return this._getMax(this.root).value
}
_getMax(node) {
    if (!node.right) return node
    return this._getMax(node.right)
}

```

向上取整和向下取整，这两个操作是相反的，所以代码也是类似的，这里只介绍如何向下取整。既然是向下取整，那么根据二分搜索树的特性，值一定在根节点的左侧。只需要一直遍历左子树直到当前节点的值不再大于等于需要的值，然后判断节点是否还拥有右子树。如果说有的话，继续上面的递归判断

```

floor(v) {
    let node = this._floor(this.root, v)
    return node ? node.value : null
}
_floor(node, v) {
    if (!node) return null
    if (node.value === v) return v
    // 如果当前节点值还比需要的值大，就继续递归
    if (node.value > v) {
        return this._floor(node.left, v)
    }
    // 判断当前节点是否拥有右子树
    let right = this._floor(node.right, v)
    if (right) return right
    return node
}

```

```
}
```

排名，这是用于获取给定值的排名或者排名第几的节点的值，这两个操作也是相反的，所以这个只介绍如何获取排名第几的节点的值。对于这个操作而言，我们需要略微的改造点代码，让每个节点拥有一个 size 属性。该属性表示该节点下有多少子节点（包含自身）

```
class Node {
    constructor(value) {
        this.value = value
        this.left = null
        this.right = null
        // 修改代码
        this.size = 1
    }
}
// 新增代码
_getSize(node) {
    return node ? node.size : 0
}
addChild(node, v) {
    if (!node) {
        return new Node(v)
    }
    if (node.value > v) {
        // 修改代码
        node.size++
        node.left = this._addChild(node.left, v)
    } else if (node.value < v) {
        // 修改代码
        node.size++
        node.right = this._addChild(node.right, v)
    }
    return node
}
select(k) {
    let node = this._select(this.root, k)
    return node ? node.value : null
}
_select(node, k) {
    if (!node) return null
    // 先获取左子树下有几个节点
    let size = node.left ? node.left.size : 0
    // 判断 size 是否大于 k
    // 如果大于 k，代表所需要的节点在左节点
    if (size > k) return this._select(node.left, k)
    // 如果小于 k，代表所需要的节点在右节点
    // 注意这里需要重新计算 k，减去根节点除了右子树的节点数量
    if (size < k) return this._select(node.right, k - size - 1)
    return node
}
```

接下来讲解的是二分搜索树中最难实现的部分：删除节点。因为对于删除节点来说，会存在以下几种情况

- 需要删除的节点没有子树
- 需要删除的节点只有一条子树

- 需要删除的节点有左右两条树
- 对于前两种情况很好解决，但是第三种情况就有难度了，所以先来实现相对简单的操作：删除最小节点，对于删除最小节点来说，是不存在第三种情况的，删除最大节点操作是和删除最小节点相反的，所以这里也就不再赘述

```
deleteMin() {
    this.root = this._deleteMin(this.root)
    console.log(this.root)
}

_deleteMin(node) {
    // 一直递归左子树
    // 如果左子树为空，就判断节点是否拥有右子树
    // 有右子树的话就把需要删除的节点替换为右子树
    if ((node != null) & !node.left) return node.right
    node.left = this._deleteMin(node.left)
    // 最后需要重新维护下节点的 `size`
    node.size = this._getSize(node.left) + this._getSize(node.right) + 1
    return node
}
```

- 最后讲解的就是如何删除任意节点了。对于这个操作，T.Hibbard 在 1962 年提出了解决这个难题的办法，也就是如何解决第三种情况。
- 当遇到这种情况时，需要取出当前节点的后继节点（也就是当前节点右子树的最小节点）来替换需要删除的节点。然后将需要删除节点的左子树赋值给后继结点，右子树删除后继结点后赋值给他。
- 你如果对于这个解决办法有疑问的话，可以这样考虑。因为二分搜索树的特性，父节点一定比所有左子节点大，比所有右子节点小。那么当需要删除父节点时，势必需要拿出一个比父节点大的节点来替换父节点。这个节点肯定不存在于左子树，必然存在于右子树。然后又需要保持父节点都是比右子节点小的，那么就可以取出右子树中最小的那个节点来替换父节点

```
delete(v) {
    this.root = this._delete(this.root, v)
}

_delete(node, v) {
    if (!node) return null
    // 寻找的节点比当前节点小，去左子树找
    if (node.value < v) {
        node.right = this._delete(node.right, v)
    } else if (node.value > v) {
        // 寻找的节点比当前节点大，去右子树找
        node.left = this._delete(node.left, v)
    } else {
        // 进入这个条件说明已经找到节点
        // 先判断节点是否拥有左右子树中的一个
        // 是的话，将子树返回出去，这里和 `deleteMin` 的操作一样
        if (!node.left) return node.right
        if (!node.right) return node.left
        // 进入这里，代表节点拥有左右子树
        // 先取出当前节点的后继结点，也就是取当前节点右子树的最小值
        let min = this._getMin(node.right)
        // 取出最小值后，删除最小值
        // 然后把删除节点后的子树赋值给最小值节点
        min.right = this._deleteMin(node.right)
        // 左子树不动
        min.left = node.left
        node = min
    }
}
```

```

    }
    // 维护 size
    node.size = this._getSize(node.left) + this._getSize(node.right) + 1
    return node
}

```

#2.5 堆

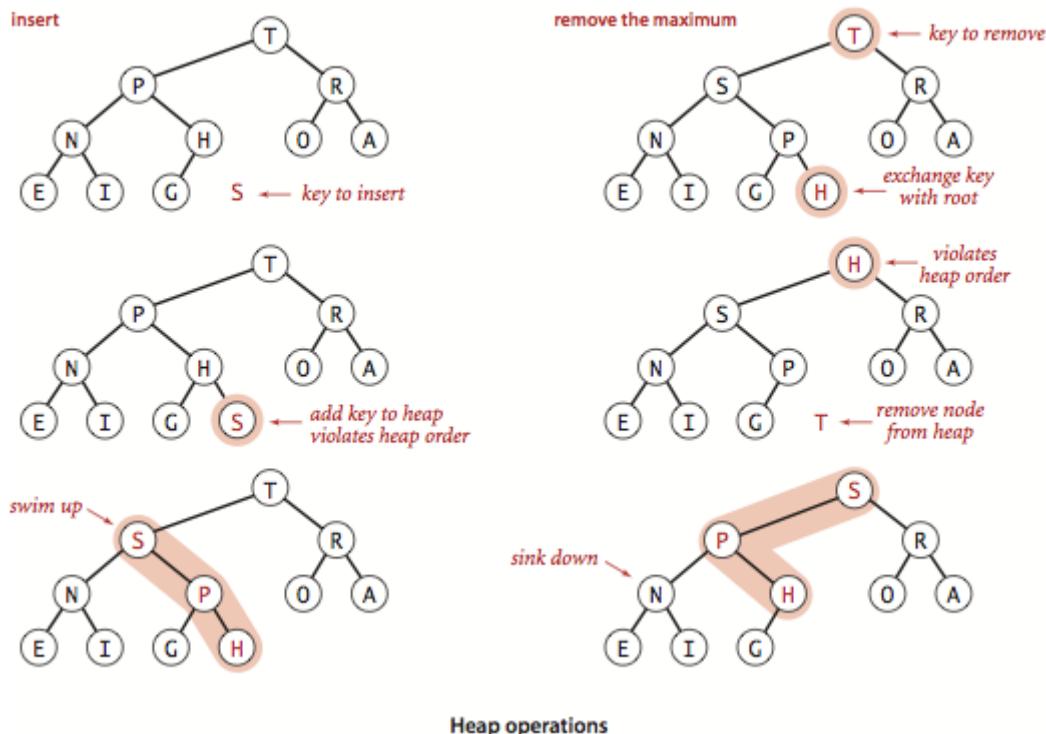
概念

- 堆通常是一个可以被看做一棵树的数组对象。
- 堆的实现通过构造二叉堆，实为二叉树的一种。这种数据结构具有以下性质。
- 任意节点小于（或大于）它的所有子节点 堆总是一棵完全树。即除了最底层，其他层的节点都被元素填满，且最底层从左到右填入。
- 将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。
- 优先队列也完全可以用堆来实现，操作是一模一样的。

实现大根堆

堆的每个节点的左边子节点索引是 $i * 2 + 1$ ，右边是 $i * 2 + 2$ ，父节点是 $(i - 1) / 2$ 。

- 堆有两个核心的操作，分别是 `shiftup` 和 `shiftDown`。前者用于添加元素，后者用于删除根节点。
- `shiftup` 的核心思路是一路将节点与父节点对比大小，如果比父节点大，就和父节点交换位置。
- `shiftDown` 的核心思路是先将根节点和末尾交换位置，然后移除末尾元素。接下来循环判断父节点和两个子节点的大小，如果子节点大，就把最大的子节点和父节点交换



```

class MaxHeap {
  constructor() {
    this.heap = []
  }
  size() {
    return this.heap.length
  }
}

```

```

empty() {
    return this.size() == 0
}
add(item) {
    this.heap.push(item)
    this._shiftUp(this.size() - 1)
}
removeMax() {
    this._shiftDown(0)
}
getParentIndex(k) {
    return parseInt((k - 1) / 2)
}
getLeftIndex(k) {
    return k * 2 + 1
}
_shiftUp(k) {
    // 如果当前节点比父节点大，就交换
    while (this.heap[k] > this.heap[this.getParentIndex(k)]) {
        this._swap(k, this.getParentIndex(k))
        // 将索引变成父节点
        k = this.getParentIndex(k)
    }
}
_shiftDown(k) {
    // 交换首位并删除末尾
    this._swap(k, this.size() - 1)
    this.heap.splice(this.size() - 1, 1)
    // 判断节点是否有左孩子，因为二叉堆的特性，有右必有左
    while (this.getLeftIndex(k) < this.size()) {
        let j = this.getLeftIndex(k)
        // 判断是否有右孩子，并且右孩子是否大于左孩子
        if (j + 1 < this.size() && this.heap[j + 1] > this.heap[j]) j++
        // 判断父节点是否已经比子节点都大
        if (this.heap[k] >= this.heap[j]) break
        this._swap(k, j)
        k = j
    }
}
_swap(left, right) {
    let rightValue = this.heap[right]
    this.heap[right] = this.heap[left]
    this.heap[left] = rightValue
}
}

```

#三、算法

#3.1 时间复杂度

- 通常使用最差的时间复杂度来衡量一个算法的好坏。
- 常数时间 $O(1)$ 代表这个操作和数据量没关系，是一个固定时间的操作，比如说四则运算。
- 对于一个算法来说，可能会计算出如下操作次数 $aN + 1$ ， N 代表数据量。那么该算法的时间复杂度就是 $O(N)$ 。因为我们在计算时间复杂度的时候，数据量通常是非常大的，这时候低阶项和常数

项可以忽略不计。

- 当然可能会出现两个算法都是 $O(N)$ 的时间复杂度，那么对比两个算法的好坏就要通过对比低阶项和常数项了

#3.2 位运算

- 位运算在算法中很有用，速度可以比四则运算快很多。
- 在学习位运算之前应该知道十进制如何转二进制，二进制如何转十进制。这里说明下简单的计算方式
- 十进制 33 可以看成是 $32 + 1$ ，并且 33 应该是六位二进制的（因为 33 近似 32，而 32 是 2 的五次方，所以是六位），那么十进制 33 就是 100001，只要是 2 的次方，那么就是 1 否则都为 0 那么二进制 100001 同理，首位是 2^5 ，末位是 2^0 ，相加得出 33

左移 <<

```
10 << 1 // -> 20
```

左移就是将二进制全部往左移动，10 在二进制中表示为 1010，左移一位后变成 10100，转换为十进制也就是 20，所以基本可以把左移看成以下公式 $a * (2^b)$

算数右移 >>

```
10 >> 1 // -> 5
```

- 算数右移就是将二进制全部往右移动并去除多余的右边，10 在二进制中表示为 1010，右移一位后变成 101，转换为十进制也就是 5，所以基本可以把右移看成以下公式 $\text{int } v = a / (2^b)$
- 右移很好用，比如可以用在二分算法中取中间值

```
13 >> 1 // -> 6
```

按位操作

• 按位与

每一位都为 1，结果才为 1

```
8 & 7 // -> 0  
// 1000 & 0111 -> 0000 -> 0
```

• 按位或

其中一位为 1，结果就是 1

```
8 | 7 // -> 15  
// 1000 | 0111 -> 1111 -> 15
```

• 按位异或

每一位都不同，结果才为 1

```
8 ^ 7 // -> 15
8 ^ 8 // -> 0
// 1000 ^ 0111 -> 1111 -> 15
// 1000 ^ 1000 -> 0000 -> 0
```

面试题：两个数不使用四则运算得出和

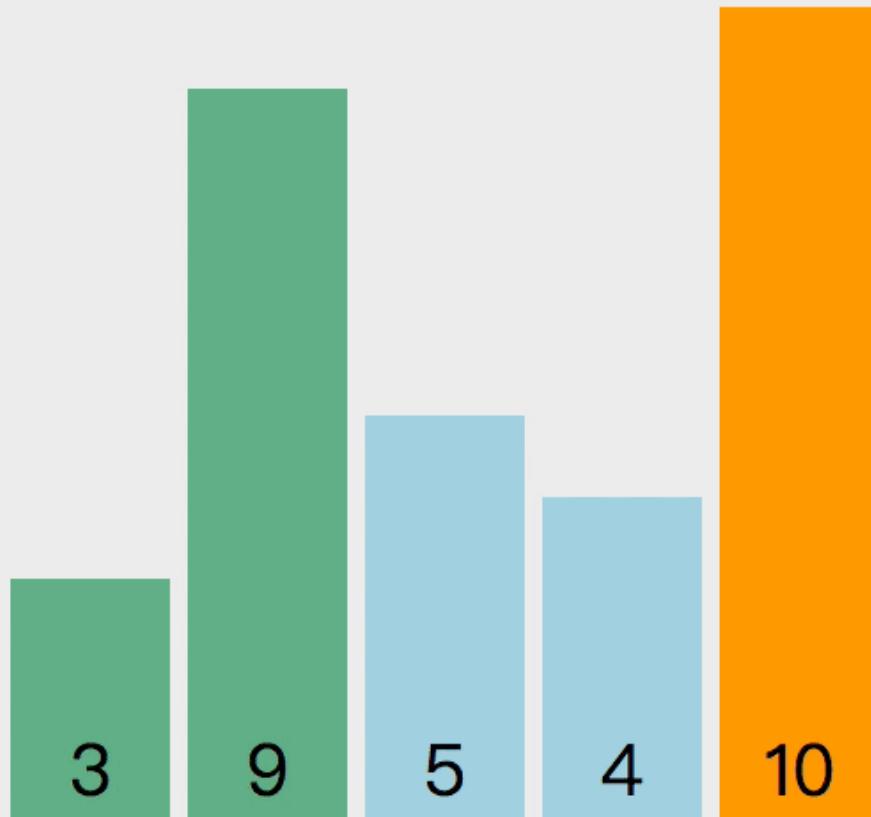
这道题中可以按位异或，因为按位异或就是不进位加法， $8 \wedge 8 = 0$ 如果进位了，就是 16 了，所以我们只需要将两个数进行异或操作，然后进位。那么也就是说两个二进制都是 1 的位置，左边应该有一个进位 1，所以可以得出以下公式 $a + b = (a \wedge b) + ((a \& b) \ll 1)$ ，然后通过迭代的方式模拟加法

```
function sum(a, b) {
    if (a == 0) return b
    if (b == 0) return a
    let newA = a ^ b
    let newB = (a & b) << 1
    return sum(newA, newB)
}
```

#3.3 排序

冒泡排序

冒泡排序的原理如下，从第一个元素开始，把当前元素和下一个索引元素进行比较。如果当前元素大，那么就交换位置，重复操作直到比较到最后一个元素，那么此时最后一个元素就是该数组中最大的数。下一轮重复以上操作，但是此时最后一个元素已经是最大数了，所以不需要再比较最后一个元素，只需要比较到 $\text{length} - 1$ 的位置



以下是实现该算法的代码

```

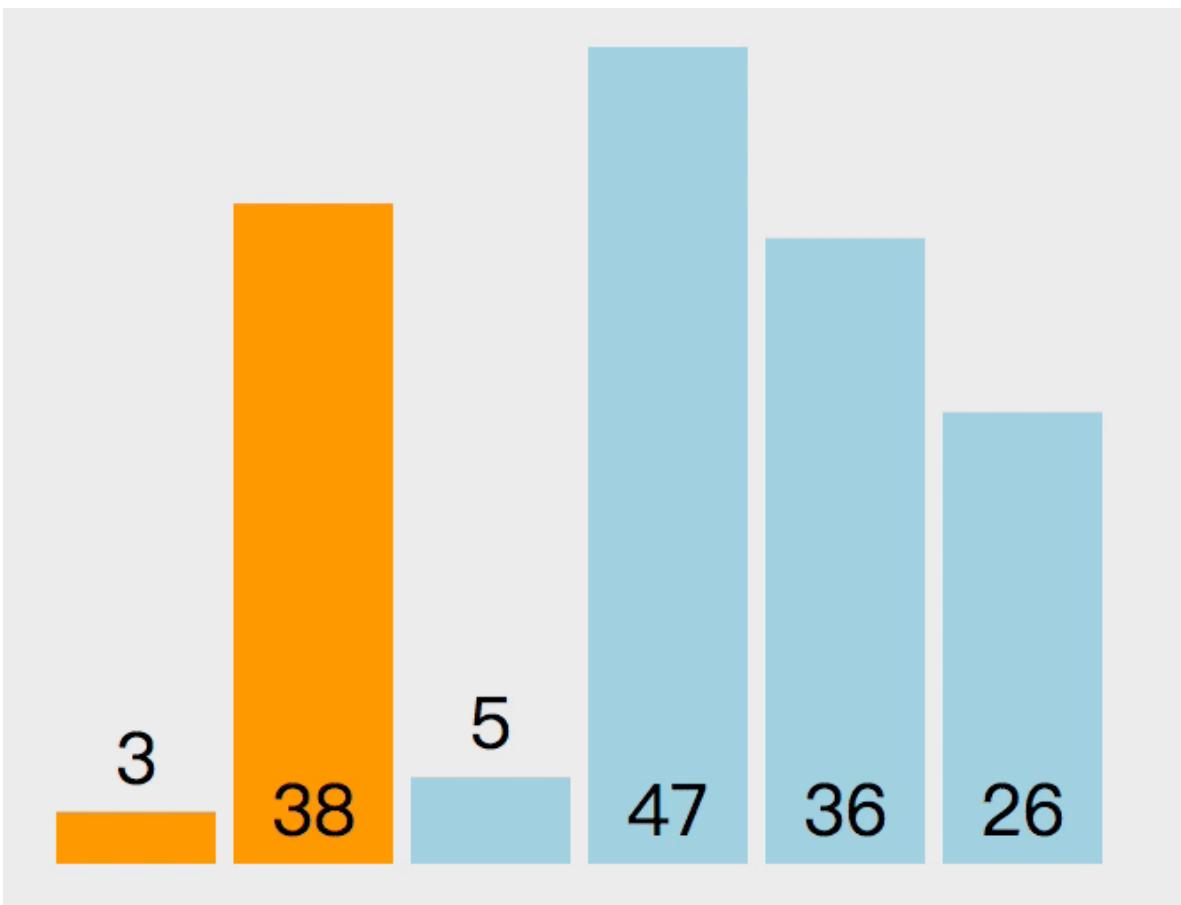
function bubble(array) {
    checkArray(array);
    for (let i = array.length - 1; i > 0; i--) {
        // 从 0 到 `length - 1` 遍历
        for (let j = 0; j < i; j++) {
            if (array[j] > array[j + 1]) swap(array, j, j + 1)
        }
    }
    return array;
}

```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + \dots + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

插入排序

插入排序的原理如下。第一个元素默认是已排序元素，取出下一个元素和当前元素比较，如果当前元素大就交换位置。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始，向前对比，重复之前的操作



以下是实现该算法的代码

```

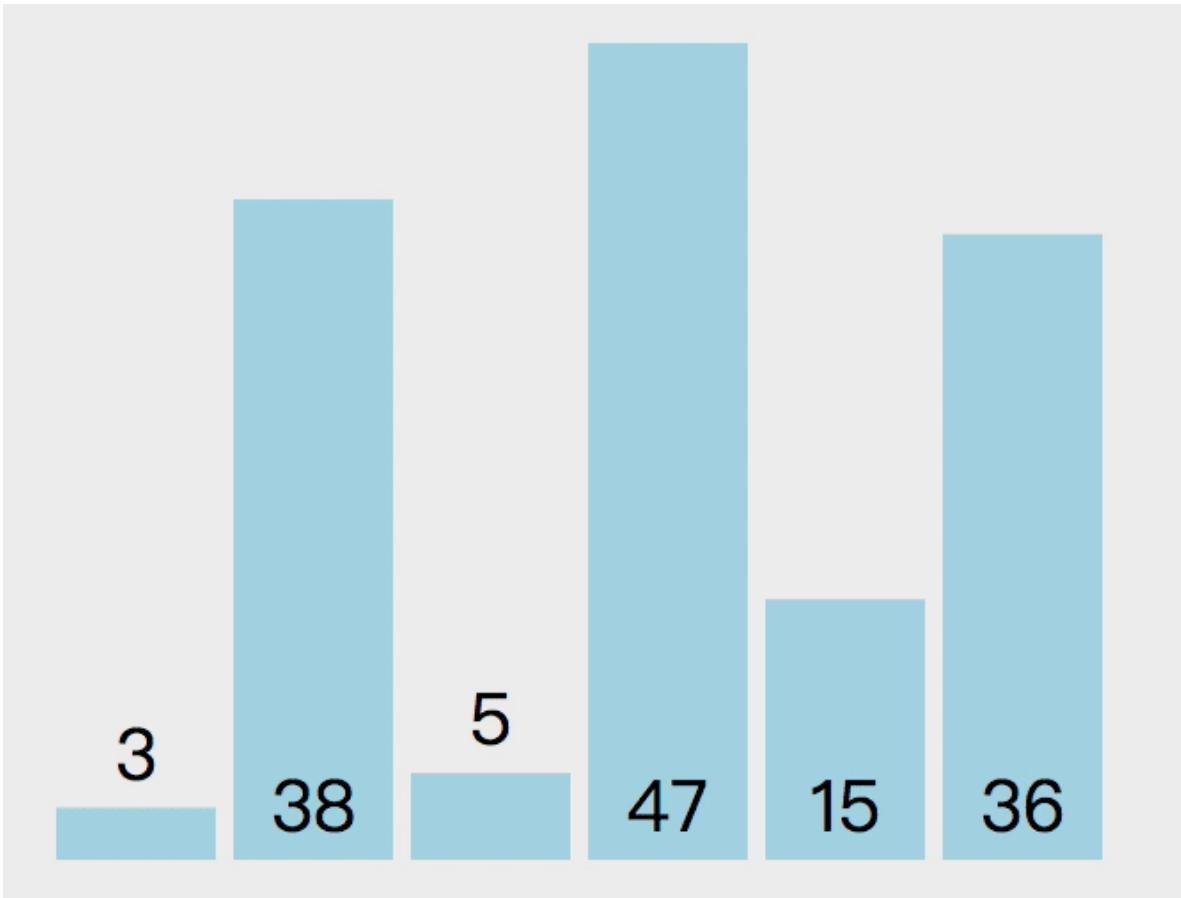
function insertion(array) {
    checkArray(array);
    for (let i = 1; i < array.length; i++) {
        for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)
            swap(array, j, j + 1);
    }
    return array;
}

```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + \dots + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

选择排序

选择排序的原理如下。遍历数组，设置最小值的索引为 0，如果取出的值比当前最小值小，就替换最小值索引，遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后，第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作



以下是实现该算法的代码

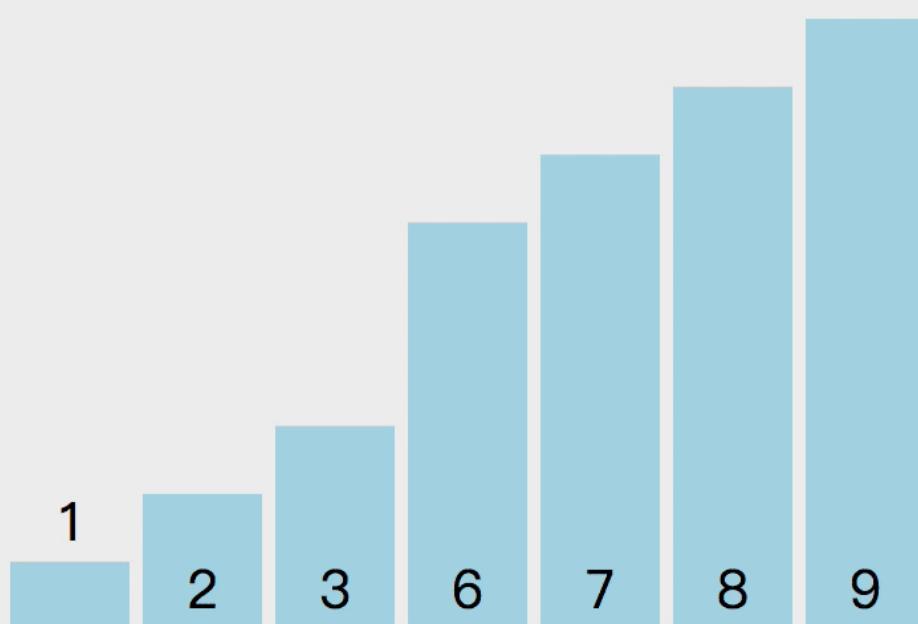
```
function selection(array) {
  checkArray(array);
  for (let i = 0; i < array.length - 1; i++) {
    let minIndex = i;
    for (let j = i + 1; j < array.length; j++) {
      minIndex = array[j] < array[minIndex] ? j : minIndex;
    }
    swap(array, i, minIndex);
  }
  return array;
}
```

该算法的操作次数是一个等差数列 $n + (n - 1) + (n - 2) + \dots + 1$ ，去掉常数项以后得出时间复杂度是 $O(n * n)$

归并排序

归并排序的原理如下。递归的将数组两两分开直到最多包含两个元素，然后将数组排序合并，最终合并为排序好的数组。假设我有一组数组 [3, 1, 2, 8, 9, 7, 6]，中间数索引是 3，先排序数组 [3, 1, 2, 8]。在这个左边数组上，继续拆分直到变成数组包含两个元素（如果数组长度是奇数的话，会有一个拆分数组只包含一个元素）。然后排序数组 [3, 1] 和 [2, 8]，然后再排

序数组 `[1, 3, 2, 8]`，这样左边数组就排序完成，然后按照以上思路排序右边数组，最后将数组 `[1, 2, 3, 8]` 和 `[6, 7, 9]` 排序



以下是实现该算法的代码

```
function sort(array) {
  checkArray(array);
  mergeSort(array, 0, array.length - 1);
  return array;
}

function mergeSort(array, left, right) {
  // 左右索引相同说明已经只有一个数
  if (left === right) return;
  // 等同于 `left + (right - left) / 2`
  // 相比 `(left + right) / 2` 来说更加安全，不会溢出
  // 使用位运算是因为位运算比四则运算快
  let mid = parseInt(left + ((right - left) >> 1));
  mergeSort(array, left, mid);
  mergeSort(array, mid + 1, right);
```

```

let help = [];
let i = 0;
let p1 = left;
let p2 = mid + 1;
while (p1 <= mid && p2 <= right) {
    help[i++] = array[p1] < array[p2] ? array[p1++] : array[p2++];
}
while (p1 <= mid) {
    help[i++] = array[p1++];
}
while (p2 <= right) {
    help[i++] = array[p2++];
}
for (let i = 0; i < help.length; i++) {
    array[left + i] = help[i];
}
return array;
}

```

以上算法使用了递归的思想。递归的本质就是压栈，每递归执行一次函数，就将该函数的信息（比如参数，内部的变量，执行到的行数）压栈，直到遇到终止条件，然后出栈并继续执行函数。对于以上递归函数的调用轨迹如下

```

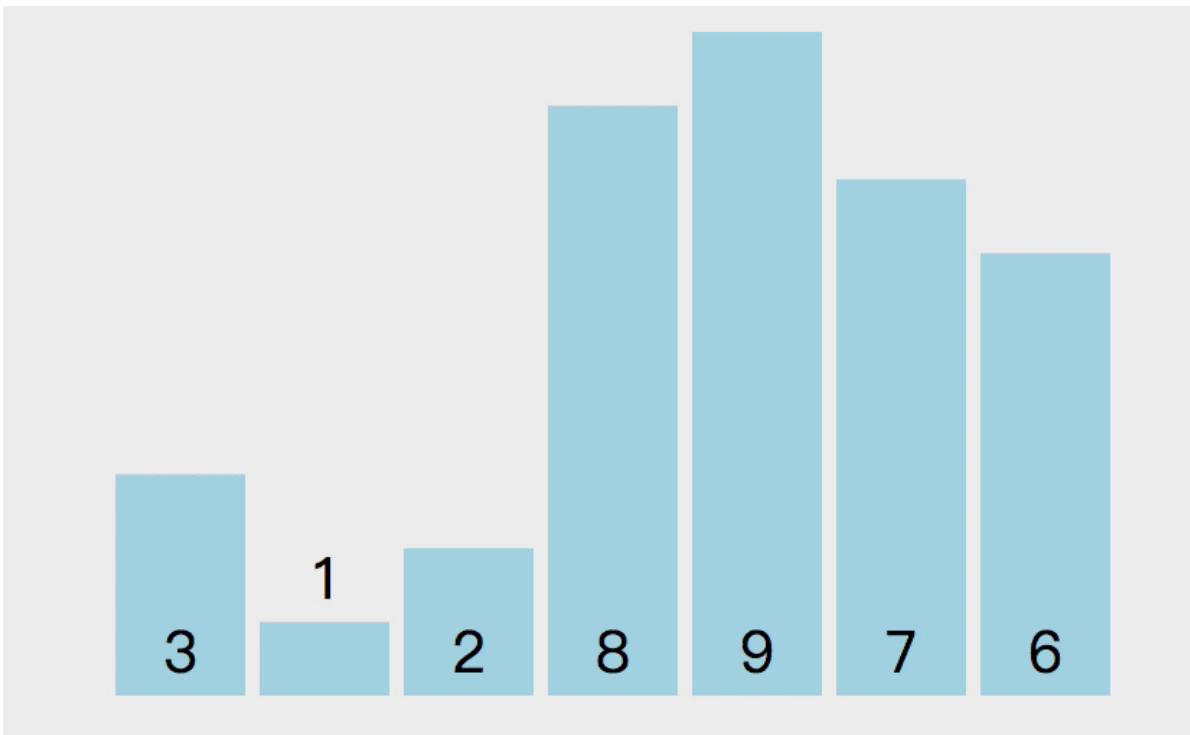
mergeSort(data, 0, 6) // mid = 3
mergeSort(data, 0, 3) // mid = 1
mergeSort(data, 0, 1) // mid = 0
    mergeSort(data, 0, 0) // 遇到终止，回退到上一步
mergeSort(data, 1, 1) // 遇到终止，回退到上一步
// 排序 p1 = 0, p2 = mid + 1 = 1
// 回退到 `mergeSort(data, 0, 3)` 执行下一个递归
mergeSort(2, 3) // mid = 2
mergeSort(3, 3) // 遇到终止，回退到上一步
// 排序 p1 = 2, p2 = mid + 1 = 3
// 回退到 `mergeSort(data, 0, 3)` 执行合并逻辑
// 排序 p1 = 0, p2 = mid + 1 = 2
// 执行完毕回退
// 左边数组排序完毕，右边也是如上轨迹

```

该算法的操作次数是可以这样计算：递归了两次，每次数据量是数组的一半，并且最后把整个数组迭代了一次，所以得出表达式 $2T(N / 2) + T(N)$ (T 代表时间， N 代表数据量)。根据该表达式可以套用该公式 得出时间复杂度为 $O(N * \log N)$

快排

快排的原理如下。随机选取一个数组中的值作为基准值，从左至右取值与基准值对比大小。比基准值小的放数组左边，大的放右边，对比完成后将基准值和第一个比基准值大的值交换位置。然后将数组以基准值的位置分为两部分，继续递归以上操作。



以下是实现该算法的代码

```
function sort(array) {
    checkArray(array);
    quickSort(array, 0, array.length - 1);
    return array;
}

function quickSort(array, left, right) {
    if (left < right) {
        swap(array, , right)
        // 随机取值，然后和末尾交换，这样做比固定取一个位置的复杂度略低
        let indexs = part(array, parseInt(Math.random() * (right - left + 1)) + left, right);
        quickSort(array, left, indexs[0]);
        quickSort(array, indexs[1] + 1, right);
    }
}

function part(array, left, right) {
    let less = left - 1;
    let more = right;
    while (left < more) {
        if (array[left] < array[right]) {
            // 当前值比基准值小，`less` 和 `left` 都加一
            ++less;
            ++left;
        } else if (array[left] > array[right]) {
            // 当前值比基准值大，将当前值和右边的值交换
            // 并且不改变 `left`，因为当前换过来的值还没有判断过大小
            swap(array, --more, left);
        } else {
            // 和基准值相同，只移动下标
            left++;
        }
    }
    // 将基准值和比基准值大的第一个值交换位置
}
```

```

    // 这样数组就变成 `[[比基准值小, 基准值, 比基准值大]]` 
    swap(array, right, more);
    return [less, more];
}

```

该算法的复杂度和归并排序是相同的，但是额外空间复杂度比归并排序少，只需 $O(\log N)$ ，并且相比归并排序来说，所需的常数时间也更少

面试题

Sort Colors: 该题目来自 LeetCode，题目需要我们将 `[2,0,2,1,1,0]` 排序成 `[0,0,1,1,2,2]`，这个问题就可以使用三路快排的思想

```

var sortColors = function(nums) {
  let left = -1;
  let right = nums.length;
  let i = 0;
  // 下标如果遇到 right, 说明已经排序完成
  while (i < right) {
    if (nums[i] === 0) {
      swap(nums, i++, ++left);
    } else if (nums[i] === 1) {
      i++;
    } else {
      swap(nums, i, --right);
    }
  }
};

```

#3.4 链表

反转单向链表

该题目来自 LeetCode，题目需要将一个单向链表反转。思路很简单，使用三个变量分别表示当前节点和当前节点的前后节点，虽然这题很简单，但是却是一道面试常考题

```

var reverseList = function(head) {
  // 判断下变量边界问题
  if (!head || !head.next) return head
  // 初始设置为空，因为第一个节点反转后就是尾部，尾部节点指向 null
  let pre = null
  let current = head
  let next
  // 判断当前节点是否为空
  // 不为空就先获取当前节点的下一节点
  // 然后把当前节点的 next 设为上一个节点
  // 然后把 current 设为下一个节点，pre 设为当前节点
  while(current) {
    next = current.next
    current.next = pre
    pre = current
    current = next
  }
  return pre
};

```

#3.5 树

二叉树的先序，中序，后序遍历

- 先序遍历表示先访问根节点，然后访问左节点，最后访问右节点。
- 中序遍历表示先访问左节点，然后访问根节点，最后访问右节点。
- 后序遍历表示先访问左节点，然后访问右节点，最后访问根节点

递归实现

递归实现相当简单，代码如下

```
function TreeNode(val) {  
    this.val = val;  
    this.left = this.right = null;  
}  
var traversal = function(root) {  
    if (root) {  
        // 先序  
        console.log(root);  
        traversal(root.left);  
        // 中序  
        // console.log(root);  
        traversal(root.right);  
        // 后序  
        // console.log(root);  
    }  
};
```

对于递归的实现来说，只需要理解每个节点都会被访问三次就明白为什么这样实现了

非递归实现

非递归实现使用了栈的结构，通过栈的先进后出模拟递归实现。

以下是先序遍历代码实现

```
function pre(root) {  
    if (root) {  
        let stack = [];  
        // 先将根节点 push  
        stack.push(root);  
        // 判断栈中是否为空  
        while (stack.length > 0) {  
            // 弹出栈顶元素  
            root = stack.pop();  
            console.log(root);  
            // 因为先序遍历是先左后右，栈是先进后出结构  
            // 所以先 push 右边再 push 左边  
            if (root.right) {  
                stack.push(root.right);  
            }  
            if (root.left) {  
                stack.push(root.left);  
            }  
        }  
    }  
}
```

以下是中序遍历代码实现

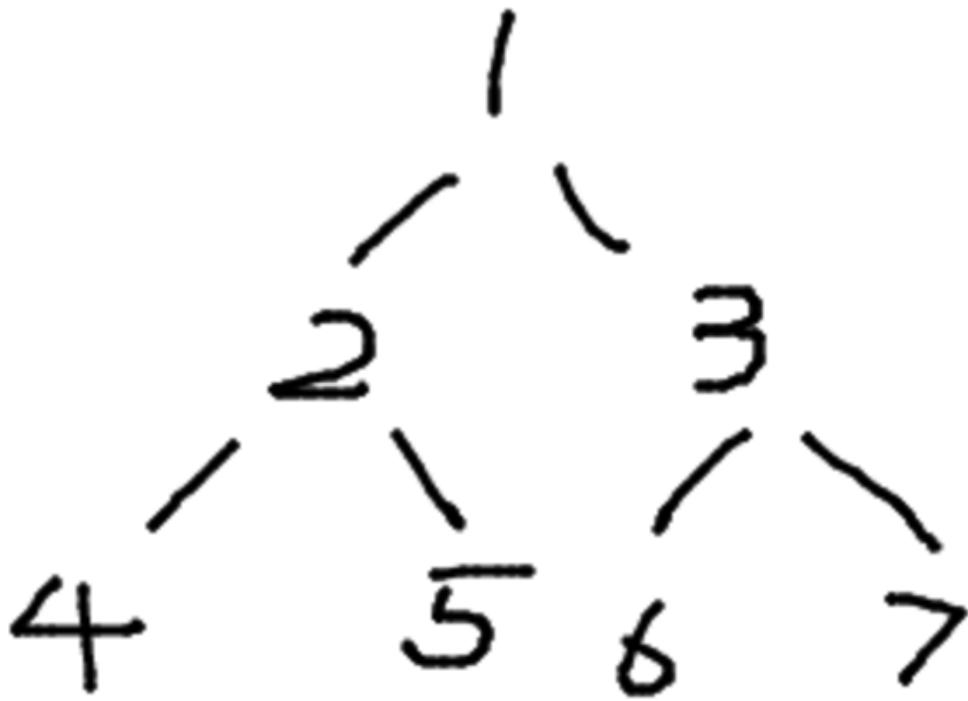
```
function mid(root) {  
    if (root) {  
        let stack = [];  
        // 中序遍历是先左再根最后右  
        // 所以首先应该先把最左边节点遍历到底依次 push 进栈  
        // 当左边没有节点时，就打印栈顶元素，然后寻找右节点  
        // 对于最左边的叶节点来说，可以把它看成是两个 null 节点的父节点  
        // 左边打印不出东西就把父节点拿出来打印，然后再看右节点  
        while (stack.length > 0 || root) {  
            if (root) {  
                stack.push(root);  
                root = root.left;  
            } else {  
                root = stack.pop();  
                console.log(root);  
                root = root.right;  
            }  
        }  
    }  
}
```

以下是后序遍历代码实现，该代码使用了两个栈来实现遍历，相比一个栈的遍历来说要容易理解很多

```
function pos(root) {  
    if (root) {  
        let stack1 = [];  
        let stack2 = [];  
        // 后序遍历是先左再右最后根  
        // 所以对于一个栈来说，应该先 push 根节点  
        // 然后 push 右节点，最后 push 左节点  
        stack1.push(root);  
        while (stack1.length > 0) {  
            root = stack1.pop();  
            stack2.push(root);  
            if (root.left) {  
                stack1.push(root.left);  
            }  
            if (root.right) {  
                stack1.push(root.right);  
            }  
        }  
        while (stack2.length > 0) {  
            console.log(stack2.pop());  
        }  
    }  
}
```

中序遍历的前驱后继节点

实现这个算法的前提是节点有一个 `parent` 的指针指向父节点，根节点指向 `null`



如图所示，该树的中序遍历结果是 [4, 2, 5, 1, 6, 3, 7]

前驱节点

对于节点 2 来说，他的前驱节点就是 4，按照中序遍历原则，可以得出以下结论

- 如果选取的节点的左节点不为空，就找该左节点最右的节点。对于节点 1 来说，他有左节点 2，那么节点 2 的最右节点就是 5
- 如果左节点为空，且目标节点是父节点的右节点，那么前驱节点为父节点。对于节点 5 来说，没有左节点，且是节点 2 的右节点，所以节点 2 是前驱节点
- 如果左节点为空，且目标节点是父节点的左节点，向上寻找到第一个是父节点的右节点的节点。对于节点 6 来说，没有左节点，且是节点 3 的左节点，所以向上寻找到节点 1，发现节点 3 是节点 1 的右节点，所以节点 1 是节点 6 的前驱节点

以下是算法实现

```

function predecessor(node) {
    if (!node) return
    // 结论 1
    if (node.left) {
        return getRight(node.left)
    } else {
        let parent = node.parent
        // 结论 2 3 的判断
        while(parent && parent.right === node) {
            node = parent
            parent = node.parent
        }
        return parent
    }
}
function getRight(node) {
    if (!node) return
    node = node.right
}

```

```
    while(node) node = node.right
    return node
}
```

后继节点

对于节点 2 来说，他的后继节点就是 5，按照中序遍历原则，可以得出以下结论

- 如果有右节点，就找到该右节点的最左节点。对于节点 1 来说，他有右节点 3，那么节点 3 的最左节点就是 6
- 如果没有右节点，就向上遍历直到找到一个节点是父节点的左节点。对于节点 5 来说，没有右节点，就向上寻找节点 2，该节点是父节点 1 的左节点，所以节点 1 是后继节点 以下是算法实现

```
function successor(node) {
  if (!node) return
  // 结论 1
  if (node.right) {
    return getLeft(node.right)
  } else {
    // 结论 2
    let parent = node.parent
    // 判断 parent 为空
    while(parent && parent.left === node) {
      node = parent
      parent = node.parent
    }
    return parent
  }
}

function getLeft(node) {
  if (!node) return
  node = node.left
  while(node) node = node.left
  return node
}
```

树的深度

树的最大深度：该题目来自 Leetcode，题目需要求出一颗二叉树的最大深度

以下是算法实现

```
var maxDepth = function(root) {
  if (!root) return 0
  return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1
};
```

对于该递归函数可以这样理解：一旦没有找到节点就会返回 0，每弹出一次递归函数就会加一，树有三层就会得到3

精简

#一、CSS相关

#1.1 左边定宽，右边自适应方案：float + margin, float + calc

```
/* 方案1 */
.left {
  width: 120px;
  float: left;
}
.right {
  margin-left: 120px;
}

/* 方案2 */
.left {
  width: 120px;
  float: left;
}
.right {
  width: calc(100% - 120px);
  float: left;
}
```

#1.2 左右两边定宽，中间自适应：float, float + calc, 圣杯布局 (设置BFC, margin负值法) , flex

```
.wrap {
  width: 100%;
  height: 200px;
}
.wrap > div {
  height: 100%;
}

/* 方案1 */
.left {
  width: 120px;
  float: left;
}
.right {
  float: right;
  width: 120px;
}
.center {
  margin: 0 120px;
}

/* 方案2 */
.left {
  width: 120px;
  float: left;
}
.right {
  float: right;
  width: 120px;
}
.center {
  width: calc(100% - 240px);
```

```
margin-left: 120px;
}
/* 方案3 */
.wrap {
  display: flex;
}
.left {
  width: 120px;
}
.right {
  width: 120px;
}
.center {
  flex: 1;
}
```

#1.3 左右居中

- 行内元素: `text-align: center`
- 定宽块状元素: 左右 `margin` 值为 `auto`
- 不定宽块状元素: `table` 布局, `position + transform`

```
/* 方案1 */
.wrap {
  text-align: center
}
.center {
  display: inline;
  /* or */
  /* display: inline-block; */
}
/* 方案2 */
.center {
  width: 100px;
  margin: 0 auto;
}
/* 方案2 */
.wrap {
  position: relative;
}
.center {
  position: absolute;
  left: 50%;
  transform: translateX(-50%);
}
```

#1.4 上下垂直居中

- 定高: `margin`, `position + margin(负值)`
- 不定高: `position + transform`, `flex`, `IFC + vertical-align:middle`

```
/* 定高方案1 */
.center {
  height: 100px;
  margin: 50px 0;
```

```

}
/* 定高方案2 */
.center {
  height: 100px;
  position: absolute;
  top: 50%;
  margin-top: -25px;
}
/* 不定高方案1 */
.center {
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
}
/* 不定高方案2 */
.wrap {
  display: flex;
  align-items: center;
}
.center {
  width: 100%;
}
/* 不定高方案3 */
/* 设置 inline-block 则会在外层产生 IFC，高度设为 100% 撑开 wrap 的高度 */
.wrap::before {
  content: '';
  height: 100%;
  display: inline-block;
  vertical-align: middle;
}
.wrap {
  text-align: center;
}
.center {
  display: inline-block;
  vertical-align: middle;
}

```

#1.5 盒模型：content（元素内容） + padding（内边距） + border（边框） + margin（外边距）

延伸：`box-sizing`

- `content-box`：默认值，总宽度 = `margin + border + padding + width`
- `border-box`：盒子宽度包含 `padding` 和 `border`，总宽度 = `margin + width`
- `inherit`：从父元素继承 `box-sizing` 属性

#1.6 BFC、IFC、GFC、FFC：FC（Formatting Contexts），格式化上下文

BFC：块级格式化上下文，容器里面的子元素不会在布局上影响到外面的元素。反之也是如此(按照这个理念来想，只要脱离文档流，肯定就能产生 `BFC`)。产生 `BFC` 方式如下

- `float` 的值不为 `none`。
- `overflow` 的值不为 `visible`。
- `position` 的值不为 `relative` 和 `static`。

- `display` 的值为 `table-cell`, `table-caption`, `inline-block` 中的任何一个

用处? 常见的多栏布局, 结合块级别元素浮动, 里面的元素则是在一个相对隔离的环境里运行

`IFC`: 内联格式化上下文, `IFC` 的 `line box` (线框) 高度由其包含行内元素中最高的实际高度计算而来 (不受到竖直方向的 `padding/margin` 影响)。

`IFC` 中的 `line box` 一般左右都贴紧整个 `IFC`, 但是会因为 `float` 元素而扰乱。`float` 元素会位于 `IFC` 与 `line box` 之间, 使得 `line box` 宽度缩短。同个 `ifc` 下的多个 `line box` 高度会不同。`IFC` 中时不可能有块级元素的, 当插入块级元素时 (如 `p` 中插入 `div`) 会产生两个匿名块与 `div` 分隔开, 即产生两个 `IFC`, 每个 `IFC` 对外表现为块级元素, 与 `div` 垂直排列。

用处?

- 水平居中: 当一个块要在环境中水平居中时, 设置其为 `inline-block` 则会在外层产生 `IFC`, 通过 `text-align` 则可以使其水平居中。
- 垂直居中: 创建一个 `IFC`, 用其中一个元素撑开父元素的高度, 然后设置其 `vertical-align: middle`, 其他行内元素则可以在此父元素下垂直居中
 - `GFC`: 网格布局格式化上下文 (`display: grid`)
 - `FFC`: 自适应格式化上下文 (`display: flex`)

#二、JS 基础 (ES5)

#2.1 原型

这里可以谈很多, 只要围绕 `[[prototype]]` 谈, 都没啥问题

#2.2 闭包

牵扯作用域, 可以两者联系起来一起谈

#2.3 作用域

词法作用域, 动态作用域

#2.4 this

不同情况的调用, `this` 指向分别如何。顺带可以提一下 es6 中箭头函数没有 `this`, `arguments`, `super` 等, 这些只依赖包含箭头函数最接近的函数

#2.5 call, apply, bind 三者用法和区别

参数、绑定规则 (显示绑定和强绑定), 运行效率 (最终都会转换成一个一个的参数去运行)、运行情况 (`call`, `apply` 立即执行, `bind` 是 `return` 出一个 `this` “固定”的函数, 这也是为什么 `bind` 是强绑定的一个原因)

注: “固定”这个词的含义, 它指的固定是指只要传进去了 `context`, 则 `bind` 中 `return` 出来的函数 `this` 便一直指向 `context`, 除非 `context` 是个变量

#2.6 变量声明提升

`js` 代码在运行前都会进行 `AST` 解析，函数申明默认会提到当前作用域最前面，变量申明也会进行提升。但赋值不会得到提升。关于 `AST` 解析，这里也可以说是形成词法作用域的主要原因

#三、JS 基础 (ES6)

#3.1 let, const

`let` 产生块级作用域（通常配合 `for` 循环或者 `{}` 进行使用产生块级作用域），`const` 申明的变量是常量（内存地址不变）

#3.2 Promise

这里你谈 `promise` 的时候，除了将他解决的痛点以及常用的 `API` 之外，最好进行拓展把 `eventloop` 带进来好好讲一下，`microtask`(微任务)、`macrotask`(任务) 的执行顺序，如果看过 `promise` 源码，最好可以谈一谈原生 `Promise` 是如何实现的。`Promise` 的关键点在于 `callback` 的两个参数，一个是 `resovle`，一个是 `reject`。还有就是 `Promise` 的链式调用 (`Promise.then()`)，每一个 `then` 都是一个责任人）

#3.3 Generator

遍历器对象生成函数，最大的特点是可以交出函数的执行权

- `function` 关键字与函数名之间有一个星号；
- 函数体内部使用 `yield` 表达式，定义不同的内部状态；
- `next` 指针移向下一个状态

这里你可以说说 `Generator` 的异步编程，以及它的语法糖 `async` 和 `awiat`，传统的异步编程。`ES6` 之前，异步编程大致如下

- 回调函数
- 事件监听
- 发布/订阅

传统异步编程方案之一：协程，多个线程互相协作，完成异步任务。

#3.4 async、await

`Generator` 函数的语法糖。有更好的语义、更好的适用性、返回值是 `Promise`。

- `async => *`
- `await => yield`

```
// 基本用法

async function timeout (ms) {
  await new Promise((resolve) => {
    setTimeout(resolve, ms)
  })
}

async function asyncConsole (value, ms) {
  await timeout(ms)
  console.log(value)
}

asyncConsole('hello async and await', 1000)
```

注：最好把2, 3, 4 连到一起讲

#3.5 AMD, CMD, CommonJs, ES6 Module：解决原始无模块化的痛点

- **AMD**: `requirejs` 在推广过程中对模块定义的规范化产出，提前执行，推崇依赖前置
- **CMD**: `seajs` 在推广过程中对模块定义的规范化产出，延迟执行，推崇依赖就近
- **CommonJs**: 模块输出的是一个值的 `copy`，运行时加载，加载的是一个对象 (`module.exports` 属性)，该对象只有在脚本运行完才会生成
- **ES6 Module**: 模块输出的是一个值的引用，编译时输出接口，`ES6` 模块不是对象，它对外接口只是一种静态定义，在代码静态解析阶段就会生成。

#四、框架相关

#4.1 数据双向绑定原理：常见数据绑定的方案

- `Object.defineProperty (vue)`：劫持数据的 `getter` 和 `setter`
- 脏值检测 (`angularjs`)：通过特定事件进行轮循 发布/订阅模式：通过消息发布并将消息进行订阅

#4.2 VDOM：三个 part

- 虚拟节点类，将真实 `DOM` 节点用 `js` 对象的形式进行展示，并提供 `render` 方法，将虚拟节点渲染成真实 `DOM`
- 节点 `diff` 比较：对虚拟节点进行 `js` 层面的计算，并将不同的操作都记录到 `patch` 对象
- `re-render`：解析 `patch` 对象，进行 `re-render`

补充1： VDOM 的必要性？

- **创建真实DOM的代价高**：真实的 `DOM` 节点 `node` 实现的属性很多，而 `vnode` 仅仅实现一些必要的属性，相比起来，创建一个 `vnode` 的成本比较低。
- **触发多次浏览器重绘及回流**：使用 `vnode`，相当于加了一个缓冲，让一次数据变动所带来的所有 `node` 变化，先在 `vnode` 中进行修改，然后 `diff` 之后对所有产生差异的节点集中一次对 `DOM tree` 进行修改，以减少浏览器的重绘及回流。

补充2： vue 为什么采用 vdom？

引入 `virtual DOM` 在性能方面的考量仅仅是一方面。

- 性能受场景的影响是非常大的，不同的场景可能造成不同实现方案之间成倍的性能差距，所以依赖细粒度绑定及 `virtual DOM` 哪个的性能更好还真不是一个容易下定论的问题。
- `Vue` 之所以引入了 `virtual DOM`，更重要的原因是为了解耦 `HTML` 依赖，这带来两个非常重要的好处是：
 - 不再依赖 `HTML` 解析器进行模版解析，可以进行更多的 `AOT` 工作提高运行时效率：通过模版 `AOT` 编译，`Vue` 的运行时体积可以进一步压缩，运行时效率可以进一步提升；
 - 可以渲染到 `DOM` 以外的平台，实现 `SSR`、同构渲染这些高级特性，`Weex` 等框架应用的就是这一特性。

综上，`virtual DOM` 在性能上的收益并不是最主要的，更重要的是它使得 `Vue` 具备了现代框架应有的高级特性。

#4.3 Vue 和 React 区别

- 相同点：都支持 `SSR`，都有 `Vdom`，组件化开发，实现 `WebComponents` 规范，数据驱动等
- 不同点：`Vue` 是双向数据流（当然为了实现单数据流方便管理组件状态，`vuex` 便出现了），`React` 是单向数据流。`Vue` 的 `Vdom` 是追踪每个组件的依赖关系，不会渲染整个组件树，`React` 每当应该状态被改变时，全部子组件都会 `re-render`

#4.4 为什么用 Vue

简洁、轻快、舒服

#五、网络基础类

#5.1 跨域

很多种方法，但万变不离其宗，都是为了搞定同源策略。重用的有 `jsonp`、`iframe`、`CORS`、`img`、`HTML5 postMessage` 等等。其中用到 `html` 标签进行跨域的原理就是 `html` 不受同源策略影响。但只是接受 `Get` 的请求方式，这个得清楚。

延伸1：img iframe script 来发送跨域请求有什么优缺点？

1. iframe

- 优点：跨域完毕之后 `DOM` 操作和互相之间的 `JavaScript` 调用都是没有问题的
- 缺点：1. 若结果要以 `URL` 参数传递，这就意味着在结果数据量很大的时候需要分割传递，巨烦。2. 还有一个是 `iframe` 本身带来的，母页面和 `iframe` 本身的交互本身就有安全性限制。

2. script

- 优点：可以直接返回 `json` 格式的数据，方便处理
- 缺点：只接受 `GET` 请求方式

3. 图片ping

- 优点：可以访问任何 `url`，一般用来进行点击追踪，做页面分析常用的方法
- 缺点：不能访问响应文本，只能监听是否响应

延伸2：配合 `webpack` 进行反向代理？

`webpack` 在 `devServer` 选项里面提供了一个 `proxy` 的参数供开发人员进行反向代理

```
'/api': {  
  target: 'http://www.example.com', // your target host  
  changeOrigin: true, // needed for virtual hosted sites  
  pathRewrite: {  
    '^/api': '' // rewrite path  
  }  
},
```

然后再配合 `http-proxy-middleware` 插件对 `api` 请求地址进行代理

```
const express = require('express');  
const proxy = require('http-proxy-middleware');  
// proxy api requests  
const exampleProxy = proxy(options); // 这里的 options 就是 webpack 里面的 proxy 选项对应的每个选项  
  
// mount `exampleProxy` in web server  
const app = express();  
app.use('/api', exampleProxy);  
app.listen(3000);
```

然后再用 `nginx` 把允许跨域的源地址添加到报头里面即可

说到 `nginx`，可以再谈谈 `CORS` 配置，大致如下

```
location / {  
  if ($request_method = 'OPTIONS') {  
    add_header 'Access-Control-Allow-Origin' '*';  
    add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';  
    add_header 'Access-Control-Allow-Credentials' 'true';  
    add_header 'Access-Control-Allow-Headers' 'DNT, X-Mx-ReqToken, Keep-Alive,  
User-Agent, X-Requested-With, If-Modified-Since, Cache-Control, Content-Type';  
    add_header 'Access-Control-Max-Age' 86400;  
    add_header 'Content-Type' 'text/plain charset=UTF-8';  
    add_header 'Content-Length' 0;  
    return 200;  
  }  
}
```

#5.2 http 无状态无连接

- `http` 协议对于事务处理没有记忆能力
- 对同一个 `url` 请求没有上下文关系
- 每次的请求都是独立的，它的执行情况和结果与前面的请求和之后的请求是无直接关系的，它不会受前面的请求应答情况直接影响，也不会直接影响后面的请求应答情况
- 服务器中没有保存客户端的状态，客户端必须每次带上自己的状态去请求服务器
- 人生若只如初见，请求过的资源下一次会继续进行请求

http协议无状态中的 状态 到底指的是什么？！

- 【状态】的含义就是：客户端和服务器在某次会话中产生的数据
- 那么对应的【无状态】就意味着：这些数据不会被保留
- 通过增加 `cookie` 和 `session` 机制，现在的网络请求其实是有状态的
- 在没有状态的 `http` 协议下，服务器也一定会保留你每次网络请求对数据的修改，但这跟保留每次访问的数据是不一样的，保留的只是会话产生的结果，而没有保留会话

#5.3 http-cache：就是 http 缓存

1. 首先得明确 http 缓存的好处

- 减少了冗余的数据传输，减少网费
- 减少服务器端的压力
- web 缓存能够减少延迟与网络阻塞，进而减少显示某个资源所用的时间
- 加快客户端加载网页的速度

2. 常见 http 缓存的类型

- 私有缓存（一般为本地浏览器缓存）
- 代理缓存

3. 然后谈谈本地缓存

本地缓存是指浏览器请求资源时命中了浏览器本地的缓存资源，浏览器并不会发送真正的请求给服务器了。它的执行过程是

- 第一次浏览器发送请求给服务器时，此时浏览器还没有本地缓存副本，服务器返回资源给浏览器，响应码是 200 OK，浏览器收到资源后，把资源和对应的响应头一起缓存下来
- 第二次浏览器准备发送请求给服务器时候，浏览器会先检查上一次服务端返回的响应头信息中的 Cache-Control，它的值是一个相对值，单位为秒，表示资源在客户端缓存的最大有效期，过期时间为第一次请求的时间减去 Cache-Control 的值，过期时间跟当前的请求时间比较，如果本地缓存资源没过期，那么命中缓存，不再请求服务器
- 如果没有命中，浏览器就会把请求发送给服务器，进入缓存协商阶段。

与本地缓存相关的头有：Cache-Control、Expires，Cache-Control 有多个可选值代表不同的意义，而 Expires 就是一个日期格式的绝对值。

3.1 Cache-Control

Cache-Control 是 HTTP 缓存策略中最重要的头，它是 HTTP/1.1 中出现的，它由如下几个值

- no-cache：不使用本地缓存。需要使用缓存协商，先与服务器确认返回的响应是否被更改，如果之前的响应中存在 ETag，那么请求的时候会与服务端验证，如果资源未被更改，则可以避免重新下载
- no-store：直接禁止浏览器缓存数据，每次用户请求该资源，都会向服务器发送一个请求，每次都会下载完整的资源
- public：可以被所有的用户缓存，包括终端用户和 CDN 等中间代理服务器。
- private：只能被终端用户的浏览器缓存，不允许 CDN 等中继缓存服务器对其缓存。
- max-age：从当前请求开始，允许获取的响应被重用的最长时间（秒）。

例如：

Cache-Control: public, max-age=1000

表示资源可以被所有用户以及代理服务器缓存，最长时间为1000秒。

3.2 Expires

Expires 是 HTTP/1.0 出现的头信息，同样也是用于决定本地缓存策略的头，它是一个绝对时间，时间格式是如 Mon, 10 Jun 2015 21:31:12 GMT，只要发送请求时间是在 Expires 之前，那么本地缓存始终有效，否则就会去服务器发送请求获取新的资源。如果同时出现 Cache-Control：max-age 和 Expires，那么 max-age 优先级更高。他们可以这样组合使用

```
Cache-Control: public  
Expires: wed, Jan 10 2018 00:27:04 GMT
```

3.3 所谓的缓存协商

当第一次请求时服务器返回的响应头中存在以下情况时

- 没有 `Cache-Control` 和 `Expires`
- `Cache-Control` 和 `Expires` 过期了
- `Cache-Control` 的属性设置为 `no-cache` 时

那么浏览器第二次请求时就会与服务器进行协商，询问浏览器中的缓存资源是不是旧版本，需不需要更新，此时，服务器就会做出判断，如果缓存和服务端资源的最新版本是一致的，那么就无需再次下载该资源，服务端直接返回 `304 Not Modified` 状态码，如果服务器发现浏览器中的缓存已经是旧版本了，那么服务器就会把最新资源的完整内容返回给浏览器，状态码就是 `200 ok`，那么服务端是根据什么来判断浏览器的缓存是不是最新的呢？其实是根据 `HTTP` 的另外两组头信息，分别是：`Last-Modified/If-Modified-since` 与 `ETag/If-None-Match`。

Last-Modified 与 If-Modified-Since

- 浏览器第一次请求资源时，服务器会把资源的最新修改时间 `Last-Modified:Thu, 29 Dec 2011 18:23:55 GMT` 放在响应头中返回给浏览器
- 第二次请求时，浏览器就会把上一次服务器返回的修改时间放在请求头 `If-Modified-since:Thu, 29 Dec 2011 18:23:55` 发送给服务器，服务器就会拿这个时间跟服务器上的资源的最新修改时间进行对比

如果两者相等或者大于服务器上的最新修改时间，那么表示浏览器的缓存是有效的，此时缓存会命中，服务器就不再返回内容给浏览器了，同时 `Last-Modified` 头也不会返回，因为资源没被修改，返回了也没什么意义。如果没命中缓存则最新修改的资源连同 `Last-Modified` 头一起返回

```
# 第一次请求返回的响应头  
Cache-Control:max-age=3600  
Expires: Fri, Jan 12 2018 00:27:04 GMT  
Last-Modified: Wed, Jan 10 2018 00:27:04 GMT  
  
# 第二次请求的请求头信息  
If-Modified-Since: wed, Jan 10 2018 00:27:04 GMT
```

这组头信息是基于资源的修改时间来判断资源有没有更新，另一种方式就是根据资源的内容来判断，就是接下来要讨论的 `ETag` 与 `If-None-Match`

ETag与If-None-Match

`ETag/If-None-Match` 与 `Last-Modified/If-Modified-since` 的流程其实是类似的，唯一的区别是它基于资源的内容的摘要信息（比如 `MD5 hash`）来判断

浏览器发送第二次请求时，会把第一次的响应头信息 `ETag` 的值放在 `If-None-Match` 的请求头中发送到服务器，与最新的资源的摘要信息对比，如果相等，取浏览器缓存，否则内容有更新，最新的资源连同最新的摘要信息返回。用 `ETag` 的好处是如果因为某种原因到时资源的修改时间没改变，那么用 `ETag` 就能区分资源是不是有被更新。

```
# 第一次请求返回的响应头:
```

```
Cache-Control: public, max-age=31536000  
ETag: "15f0fff99ed5aae4edffdd6496d7131f"  
# 第二次请求的请求头信息:  
  
If-None-Match: "15f0fff99ed5aae4edffdd6496d7131f"
```

#5.4 cookie 和 session

- `session`: 是一个抽象概念，开发者为了实现中断和继续等操作，将 `user agent` 和 `server` 之间一对一的交互，抽象为“会话”，进而衍生出“会话状态”，也就是 `session` 的概念
- `cookie`: 它是一个世纪存在的东西，`http` 协议中定义在 `header` 中的字段，可以认为是 `session` 的一种后端无状态实现

现在我们常说的 `session`，是为了绕开 `cookie` 的各种限制，通常借助 `cookie` 本身和后端存储实现的，一种更高级的会话状态实现

```
session` 的常见实现要借助`cookie`来发送 `sessionID
```

#5.5 安全问题，如 XSS 和 CSRF

- `XSS`: 跨站脚本攻击，是一种网站应用程序的安全漏洞攻击，是代码注入的一种。常见方式是将恶意代码注入合法代码里隐藏起来，再诱发恶意代码，从而进行各种各样的非法活动
- 防范：记住一点“所有用户输入都是不可信的”，所以得做输入过滤和转义
- `CSRF`: 跨站请求伪造，也称 `XSRF`，是一种挟制用户在当前已登录的 web 应用程序上执行非本意的操作的攻击方法。与 `xss` 相比，`xss` 利用的是用户对指定网站的信任，`CSRF` 利用的是网站对用户网页浏览器的信任。
- 防范：用户操作验证（验证码），额外验证机制（`token` 使用）等

前端性能优化篇

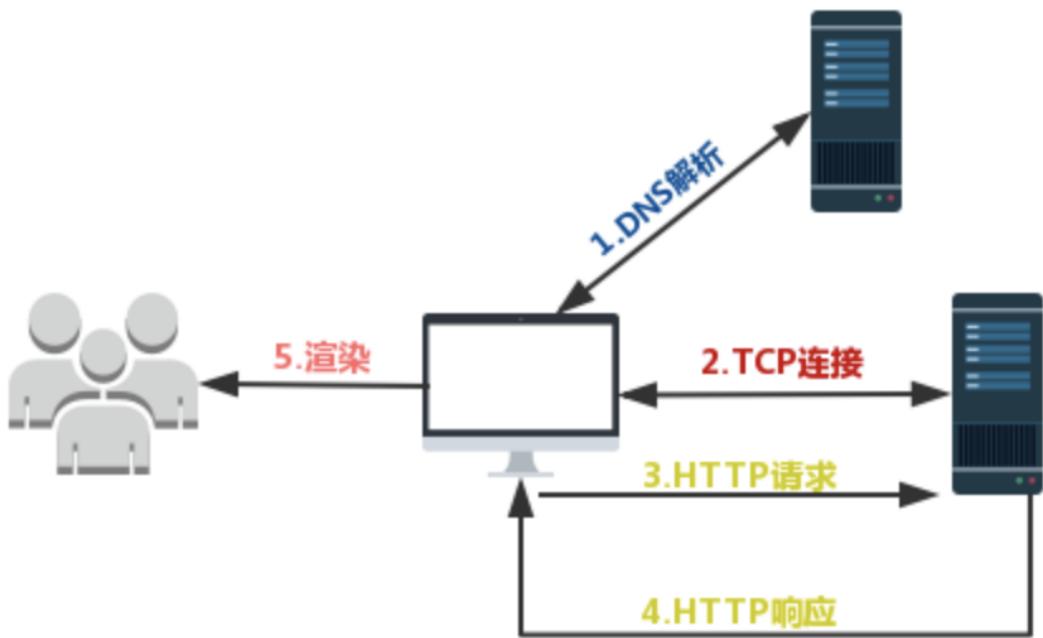
#一、前言

#知识体系：从一道面试题说起

在展开性能优化的话题之前，我想先抛出一个老生常谈的面试问题：

从输入 URL 到页面加载完成，发生了什么？

- 这个问题非常重要，因为我们后续的内容都将围绕这个问题的答案为骨架展开。我希望正在阅读这本小册的各位可以在心里琢磨一下这个问题——无须你调动太多计算机的专业知识，只需要你用最快的速度在脑海中架构起这个抽象的过程——我们接下来所有的工作，就是围绕这个过程来做文章。
- 我们现在站在性能优化的角度，一起简单地复习一遍这个经典的过程：首先我们需要通过 DNS（域名解析系统）将 URL 解析为对应的 IP 地址，然后与这个 IP 地址确定的那台服务器建立起 TCP 网络连接，随后我们向服务端抛出我们的 HTTP 请求，服务端处理完我们的请求之后，把目标数据放在 HTTP 响应里返回给客户端，拿到响应数据的浏览器就可以开始走一个渲染的流程。渲染完毕，页面便呈现给了用户，并时刻等待响应用户的操作（如下图所示）。



我们将这个过程切分为如下的过程片段：

1. DNS 解析
2. TCP 连接
3. HTTP 请求抛出
4. 服务端处理请求, HTTP 响应返回
5. 浏览器拿到响应数据, 解析响应内容, 把解析的结果展示给用户

大家谨记，我们任何一个用户端的产品，都需要把这 5 个过程滴水不漏地考虑到自己的性能优化方案内、反复权衡，从而打磨出用户满意的速度。

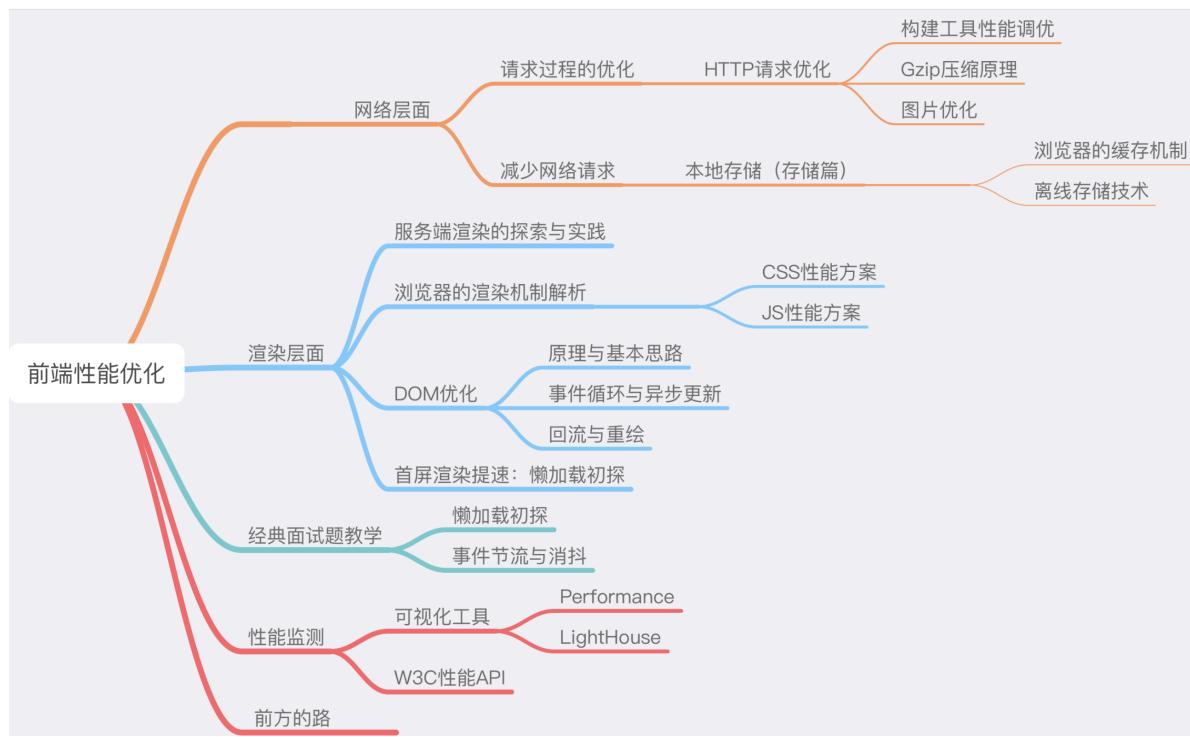
#从原理到实践：各个击破

- 我们接下来要做的事情，就是针对这五个过程进行分解，各个提问，各个击破。

具体来说，DNS 解析花时间，能不能尽量减少解析次数或者把解析前置？能——浏览器 DNS 缓存和 DNS prefetch。TCP 每次的三次握手都急死人，有没有解决方案？有——长连接、预连接、接入 SPDY 协议。如果说这两个过程的优化往往需要我们和团队的服务端工程师协作完成，前端单方面可以做的努力有限，那么 HTTP 请求呢？——在减少请求次数和减小请求体积方面，我们应该是专家！再者，服务器越远，一次请求就越慢，那部署时就把静态资源放在离我们更近的 CDN 上是不是就能更快一些？

以上提到的都是网络层面的性能优化。再往下走就是浏览器端的性能优化——这部分涉及资源加载优化、服务端渲染、浏览器缓存机制的利用、DOM 树的构建、网页排版和渲染过程、回流与重绘的考量、DOM 操作的合理规避等等——这正是前端工程师可以真正一展拳脚的地方。学习这些知识，不仅可以帮助我们从根本上提升页面性能，更能够大大加深个人对浏览器底层原理、运行机制的理解，一举两得！

我们整个的知识图谱，用思维导图展示如下：



#二、网络篇 1: webpack 性能调优与 Gzip 原理

从现在开始，我们进入网络层面的性能优化世界。

大家可以从第一节的示意图中看出，我们从输入 URL 到显示页面这个过程中，涉及到网络层面的，有三个主要过程：

- DNS 解析
- TCP 连接
- HTTP 请求/响应

对于 DNS 解析和 TCP 连接两个步骤，我们前端可以做的努力非常有限。相比之下，HTTP 连接这一层面的优化才是我们网络优化的核心。因此我们开门见山，抓主要矛盾，直接从 HTTP 开始讲起。

HTTP 优化有两个大的方向：

- 减少请求次数
- 减少单次请求所花费的时间

这两个优化点直直地指向了我们日常开发中非常常见的操作——资源的压缩与合并。没错，这就是我们每天用构建工具在做的事情。而时下最主流的构建工具无疑是 webpack，所以我们这节的主要任务就是围绕业界霸主 webpack 来做文章。

#webpack 的性能瓶颈

相信每个用过 webpack 的同学都对“打包”和“压缩”这样的事情烂熟于心。这些老生常谈的特性，我更推荐大家去阅读文档。而关于 webpack 的详细操作，则推荐大家读读这本 [关于 webpack 的掘金小册](#)，这里我们把注意力放在 webpack 的性能优化上。

webpack 的优化瓶颈，主要是两个方面：

- webpack 的构建过程太花时间
- webpack 打包的结果体积太大

#webpack 优化方案

1. 构建过程提速策略

1.1 不要让 loader 做太多事情——以 babel-loader 为例

babel-loader 无疑是强大的，但它也是慢的。

最常见的优化方式是，用 `include` 或 `exclude` 来帮我们避免不必要的转译，比如 webpack 官方在介绍 `babel-loader` 时给出的示例：

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      use: [
        {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      ]
    }
  ]
}
```

这段代码帮我们规避了对庞大的 `node__modules` 文件夹或者 `bower__components` 文件夹的处理。但通过限定文件范围带来的性能提升是有限的。除此之外，如果我们选择开启缓存将转译结果缓存至文件系统，则至少可以将 babel-loader 的工作效率提升两倍。要做到这点，我们只需要为 `loader` 增加相应的参数设定：

```
loader: 'babel-loader?cacheDirectory=true'
```

- 以上都是在讨论针对 `loader` 的配置，但我们的优化范围不止是 `loader` 们。

举个🌰，尽管我们可以在 `loader` 配置时通过写入 `exclude` 去避免 `babel-loader` 对不必要的文件的处理，但是考虑到这个规则仅作用于这个 `loader`，像一些类似 `uglifyjsPlugin` 的 `webpack` 插件在工作时依然会被这些庞大的第三方库拖累，`webpack` 构建速度依然会因此大打折扣。所以针对这些庞大的第三方库，我们还需要做一些额外的努力。

1.2 不要放过第三方库

第三方库以 `node__modules` 为代表，它们庞大得可怕，却又不可或缺。

- 处理第三方库的姿势有很多，其中，`Externals` 不够聪明，一些情况下会引发重复打包的问题；而 `CommonsChunkPlugin` 每次构建时都会重新构建一次 `vendor`；出于对效率的考虑，我们这里为大家推荐 `DllPlugin`。
- `DllPlugin` 是基于 `windows` 动态链接库（`dll`）的思想被创作出来的。这个插件会把第三方库单独打包到一个文件中，这个文件就是一个单纯的依赖库。**这个依赖库不会跟着你的业务代码一起被重新打包，只有当依赖自身发生版本变化时才会重新打包。**

用 `DllPlugin` 处理文件，要分两步走：

- 基于 `dll` 专属的配置文件，打包 `dll` 库
- 基于 `webpack.config.js` 文件，打包业务代码

以一个基于 React 的简单项目为例，我们的 `dll` 的配置文件可以编写如下：

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

module.exports = {
  entry: {
    // 依赖的库数组
    vendor: [
      'prop-types',
      'babel-polyfill',
      'react',
      'react-dom',
      'react-router-dom',
    ]
  },
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].js',
    library: '[name]_[hash]',
  },
  plugins: [
    new webpack.DllPlugin({
      // DllPlugin的name属性需要和library保持一致
      name: '[name]_[hash]',
      path: path.join(__dirname, 'dist', '[name]-manifest.json'),
      // context需要和webpack.config.js保持一致
      context: __dirname,
    }),
  ],
}
}
```

编写完成之后，运行这个配置文件，我们的 `dist` 文件夹里会出现这样两个文件：

```
vendor-manifest.json
vendor.js
```

`vendor.js` 不必解释，是我们第三方库打包的结果。这个多出来的 `vendor-manifest.json`，则用于描述每个第三方库对应的具体路径，我这里截取一部分给大家看下：

```
{
  "name": "vendor_397f9e25e49947b8675d",
  "content": {
    "./node_modules/core-js/modules/_export.js": {
      "id": 0,
      "buildMeta": {
        "providedExports": true
      }
    },
    "./node_modules/prop-types/index.js": {
      "id": 1,
      "buildMeta": {
        "providedExports": true
      }
    },
    ...
  }
}
```

随后，我们只需在 `webpack.config.js` 里针对 `dll` 稍作配置：

```
const path = require('path');
const webpack = require('webpack')
module.exports = {
  mode: 'production',
  // 编译入口
  entry: {
    main: './src/index.js'
  },
  // 目标文件
  output: {
    path: path.join(__dirname, 'dist/'),
    filename: '[name].js'
  },
  // dll相关配置
  plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      // manifest就是我们第一步中打包出来的json文件
      manifest: require('./dist/vendor-manifest.json'),
    })
  ]
}
```

一次基于 `dll` 的 `webpack` 构建过程优化，便大功告成了！

1.3 Happypack——将 loader 由单进程转为多进程

大家知道，`webpack` 是单线程的，就算此刻存在多个任务，你也只能排队一个接一个地等待处理。这是 `webpack` 的缺点，好在我们的 `CPU` 是多核的，`Happypack` 会充分释放 CPU 在多核并发方面的优势，帮我们把任务分解给多个子进程去并发执行，大大提升打包效率。

- `HappyPack` 的使用方法也非常简单，只需要我们把对 `loader` 的配置转移到 `HappyPack` 中去就好，我们可以手动告诉 `HappyPack` 我们需要多少个并发的进程：

```
const HappyPack = require('happypack')
// 手动创建进程池
const happyThreadPool = HappyPack.ThreadPool({ size: os.cpus().length })

module.exports = {
  module: {
    rules: [
      ...
      {
        test: /\.js$/,
        // 问号后面的查询参数指定了处理这类文件的HappyPack实例的名字
        loader: 'happypack/loader?id=happyBabel',
        ...
      },
    ],
  },
  plugins: [
    ...
    new HappyPack({
      // 这个HappyPack的“名字”就叫做happyBabel，和楼上的查询参数遥相呼应
    })
  ]
}
```

```

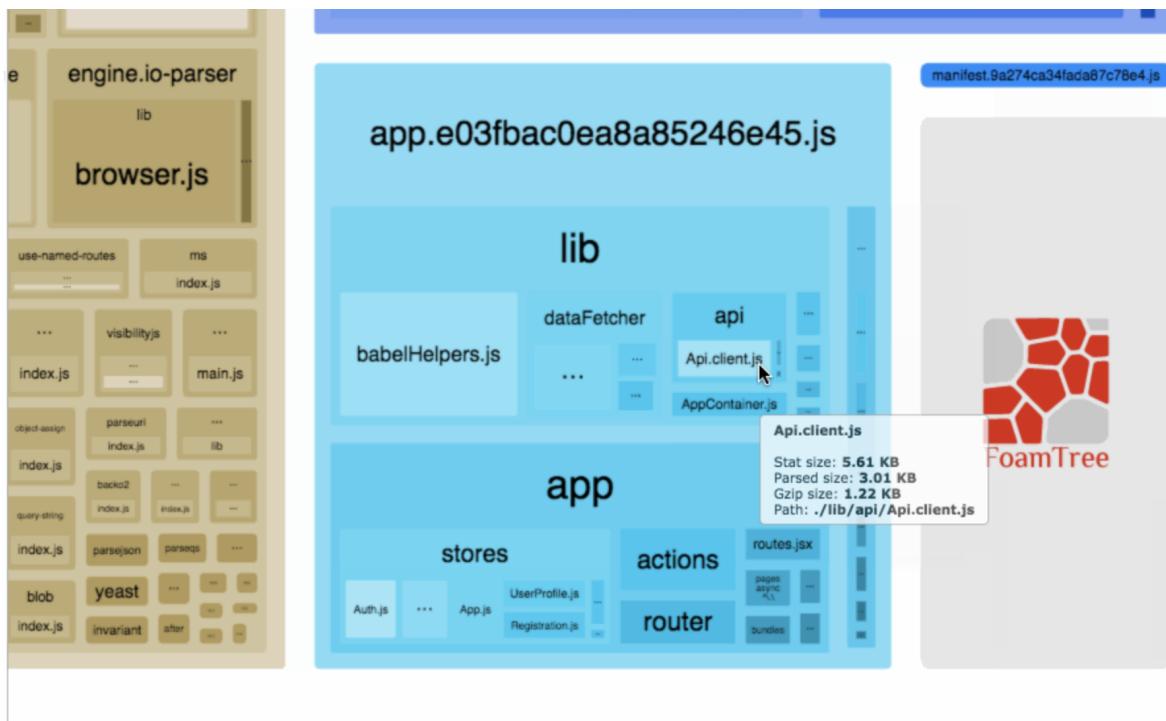
        id: 'happyBabel',
        // 指定进程池
        threadPool: happyThreadPool,
        loaders: ['babel-loader?cacheDirectory']
    })
],
}

```

#构建结果体积压缩

1. 文件结构可视化，找出导致体积过大的原因

这里为大家介绍一个非常好用的包组成可视化工具——[webpack-bundle-analyzer](#)，配置方法和普通的 plugin 无异，它会以矩形树图的形式将包内各个模块的大小和依赖关系呈现出来，格局如官方所提供这张图所示：



在使用时，我们只需要将其以插件的形式引入：

```

const BundleAnalyzerPlugin = require('webpack-bundle-
analyzer').BundleAnalyzerPlugin;

module.exports = {
  plugins: [
    new BundleAnalyzerPlugin()
  ]
}

```

2. 拆分资源

这点仍然围绕 `DllPlugin` 展开，可参考上文。

2.1 删除冗余代码

- 一个比较典型的应用，就是 `Tree-Shaking`。
- 从 `webpack2` 开始，`webpack` 原生支持了 `ES6` 的模块系统，并基于此推出了 `Tree-shaking`。
`webpack` 官方是这样介绍它的：

Tree shaking is a term commonly used in the JavaScript context for dead-code elimination, or more precisely, live-code import. It relies on ES2015 module import/export for the static structure of its module system.

- 意思是基于 `import/export` 语法, `Tree-Shaking` 可以在编译的过程中获悉哪些模块并没有真正被使用, 这些没用的代码, 在最后打包的时候会被去除。
- 举个🌰, 假设我的主干文件 (入口文件) 是这么写的:

```
import { page1, page2 } from './pages'

// show是事先定义好的函数, 大家理解它的功能是展示页面即可
show(page1)
```

`pages` 文件里, 我虽然导出了两个页面:

```
export const page1 = xxx

export const page2 = xxx
```

但因为 `page2` 事实上并没有被用到 (这个没有被用到的情况在静态分析的过程中是可以被感知出来的), 所以打包的结果里会把这部分:

```
export const page2 = xxx;
```

- 直接删掉, 这就是 `Tree-Shaking` 帮我们做的事情。
- 相信大家不难看出, `Tree-Shaking` 的针对性很强, 它更适合用来处理模块级别的冗余代码。至于 **粒度更细** 的冗余代码的去除, 往往会被整合进 JS 或 CSS 的压缩或分离过程中。
- 这里我们以当下接受度较高的 `uglifyjsPlugin` 为例, 看一下如何在压缩过程中对碎片化的冗余代码 (如 `console` 语句、注释等) 进行自动化删除:

```
const UglifyJsPlugin = require('uglifyjs-webpack-plugin');
module.exports = {
  plugins: [
    new UglifyJsPlugin({
      // 允许并发
      parallel: true,
      // 开启缓存
      cache: true,
      compress: {
        // 删除所有的console语句
        drop_console: true,
        // 把使用多次的静态值自动定义为变量
        reduce_vars: true,
      },
      output: {
        // 不保留注释
        comment: false,
        // 使输出的代码尽可能紧凑
        beautify: false
      }
    })
  ]
}
```

- 有心的同学们会注意到，这段手动引入 `uglifyjsPlugin` 的代码其实是 `webpack3` 的用法，`webpack4` 现在已经默认使用 `uglifyjs-webpack-plugin` 对代码做压缩了——在 `webpack4` 中，我们是通过配置 `optimization.minimize` 与 `optimization.minimizer` 来自定义压缩相关的操作的。
- 这里也引出了我们学习性能优化的一个核心的理念——用什么工具，怎么用，并不是我们这本小册的重点，因为所有的工具都存在用法迭代的问题。但现在大家知道了在打包的过程中做一些如上文所述的“手脚”可以实现打包结果的最优化，那下次大家再去执行打包操作，会不会对这个操作更加留心，从而自己去寻找彼时操作的具体实现方案呢？我最希望大家掌握的技能就是，先在脑海中留下“这个xx操作是对的，是有用的”，在日后的实践中，可以基于这个认知去寻找把正确的操作落地的具体方案。

2.2 按需加载

大家想象这样一个场景。我现在用 React 构建一个单页应用，用 React-Router 来控制路由，十个路由对应了十个页面，这十个页面都不简单。如果我把这整个项目打一个包，用户打开我的网站时，会发生什么？有很大机率会卡死，对不对？更好的做法肯定是先给用户展示主页，其它页面等请求到了再加载。当然这个情况也比较极端，但却能很好地引出按需加载的思想：

- 一次不加载完所有的文件内容，只加载此刻需要用到的那部分（会提前做拆分）
- 当需要更多内容时，再对用到的内容进行即时加载

好，既然说到这十个 `Router` 了，我们就拿其中一个开刀，假设我这个 Router 对应的组件叫做 `BugComponent`，来看看我们如何利用 `webpack` 做到该组件的按需加载。

当我们不需要按需加载的时候，我们的代码是这样的：

```
import BugComponent from '../pages/BugComponent'
...
<Route path="/bug" component={BugComponent}>
```

- 为了开启按需加载，我们要稍作改动。
- 首先 `webpack` 的配置文件要走起来：

```
output: {
  path: path.join(__dirname, '/../dist'),
  filename: 'app.js',
  publicPath: defaultSettings.publicPath,
  // 指定 chunkFilename
  chunkFilename: '[name].[chunkhash:5].chunk.js',
},
```

路由处的代码也要做一下配合：

```
const getComponent => (location, cb) {
  require.ensure([], (require) => {
    cb(null, require('../pages/BugComponent').default)
  }, 'bug')
},
...
<Route path="/bug" getComponent={getComponent}>
```

对，核心就是这个方法：

```
require.ensure(dependencies, callback, chunkName)
```

- 这是一个异步的方法，webpack 在打包时，`BugComponent` 会被单独打成一个文件，只有在我们跳转 bug 这个路由的时候，这个异步方法的回调才会生效，才会真正地去获取 `BugComponent` 的内容。这就是按需加载。
- 按需加载的粒度，还可以继续细化，细化到更小的组件、细化到某个功能点，都是 ok 的。

等等，这和说好的不一样啊？不是说 `Code-Splitting` 才是 React-Router 的按需加载实践吗？

- 没错，在 React-Router4 中，我们确实是用 `Code-Splitting` 替换掉了楼上这个操作。而且如果有使用过 React-Router4 实现过路由级别的按需加载的同学，可能会对 `React-Router4` 里用到的一个叫“Bundle-Loader”的东西印象深刻。我想很多同学读到按需加载这里，心里的预期或许都是时下大热的 `Code-Splitting`，而非我呈现出来的这段看似“陈旧”的代码。
- 但是，如果大家稍微留个心眼，去看一下 `Bundle Loader` 并不长的源代码的话，你会发现它竟然还是使用 `require.ensure` 来实现的——这也是我要把 `require.ensure` 单独拎出来的重要原因。所谓按需加载，根本上就是在正确的时机去触发相应的回调。理解了这个 `require.ensure` 的玩法，大家甚至可以结合业务自己去修改一个按需加载模块来用。

这也应了我之前跟大家强调那段话，工具永远在迭代，唯有掌握核心思想，才可以真正做到举一反三——唯“心”不破！

#Gzip 压缩原理

前面说了不少 webpack 的故事，目的还是帮大家更好地实现压缩和合并。说到压缩，可不只是构建工具的专利。我们日常开发中，其实还有一个便宜又好用的压缩操作：开启 `Gzip`。

具体的做法非常简单，只需要你在你的 `request headers` 中加上这么一句：

```
accept-encoding:gzip
```

相信很多同学对 `Gzip` 也是了解到这里。之所以为大家开这个彩蛋性的小节，绝不是出于炫技要来给大家展示一下 `Gzip` 的压缩算法，而是想和大家聊一个和我们前端关系更密切的话题：`HTTP` 压缩。

`HTTP` 压缩是一种内置到网页服务器和网页客户端中以改进传输速度和带宽利用率的方式。在使用 `HTTP` 压缩的情况下，`HTTP` 数据在从服务器发送前就已压缩：兼容的浏览器将在下载所需的格式前宣告支持何种方法给服务器；不支持压缩方法的浏览器将下载未经压缩的数据。最常见的压缩方案包括 `Gzip` 和 `Deflate`。

以上是摘自百科的解释，事实上，大家可以这么理解：

HTTP 压缩就是以缩小体积为目的，对 HTTP 内容进行重新编码的过程

- `Gzip` 的内核就是 `Deflate`，目前我们压缩文件用得最多的就是 `Gzip`。可以说，`Gzip` 就是 `HTTP` 压缩的经典例题。

1. 该不该用 Gzip

- 如果你的项目不是极端迷你的超小型文件，我都建议你试试 `Gzip`。
- 有的同学或许存在这样的疑问：压缩 `Gzip`，服务端要花时间；解压 `Gzip`，浏览器要花时间。中间节省出来的传输时间，真的那么可观吗？

答案是肯定的。如果你手上的项目是 `1k`、`2k` 的小文件，那确实有点高射炮打蚊子的意思，不值当。但更多的时候，我们处理的都是具备一定规模的项目文件。实践证明，这种情况下压缩和解压带来的时间开销相对于传输过程中节省下的时间开销来说，可以说是微不足道的。

2. Gzip 是万能的吗

- 首先要承认 `Gzip` 是高效的，压缩后通常能帮我们减少响应 70% 左右的大小。

- 但它并非万能。`Gzip` 并不保证针对每一个文件的压缩都会使其变小。

`Gzip` 压缩背后的原理，是在一个文本文件中找出一些重复出现的字符串、临时替换它们，从而使整个文件变小。根据这个原理，文件中代码的重复率越高，那么压缩的效率就越高，使用 `Gzip` 的收益也就越大。反之亦然。

3. webpack 的 Gzip 和服务端的 Gzip

- 一般来说，`Gzip` 压缩是服务器的活儿：服务器了解到我们这边有一个 `Gzip` 压缩的需求，它会启动自己的 `CPU` 去为我们完成这个任务。而压缩文件这个过程本身是需要耗费时间的，大家可以理解为我们以服务器压缩的时间开销和 `CPU` 开销（以及浏览器解析压缩文件的开销）为代价，省下了一些传输过程中的时间开销。
- 既然存在着这样的交换，那么就要求我们学会权衡。服务器的 `CPU` 性能不是无限的，如果存在大量的压缩需求，服务器也扛不住的。服务器一旦因此慢下来了，用户还是要等。Webpack 中 `Gzip` 压缩操作的存在，事实上就是为了在构建过程中去做一部分服务器的工作，为服务器分压。
- 因此，这两个地方的 `Gzip` 压缩，谁也不能替代谁。它们必须和平共处，好好合作。作为开发者，我们也应该结合业务压力的实际强度情况，去做好这其中的权衡。

#三、网络篇 2：图片优化——质量与性能的博弈

《高性能网站建设指南》的作者 Steve Souders 曾在 2013 年的一篇 [博客](#) 中提到：

我的大部分性能优化工作都集中在 `Javascript` 和 `css` 上，从早期的 Move Scripts to the Bottom 和 Put Stylesheets at the Top 规则。为了强调这些规则的重要性，我甚至说过，“JS 和 CSS 是页面上最重要的部分”。

几个月后，我意识到这是错误的。图片才是页面上最重要的部分。

我关注 JS 和 CSS 的重点也是如何能够更快地下载图片。图片是用户可以直观看到的。他们并不会关注 JS 和 CSS。确实，JS 和 CSS 会影响图片内容的展示，尤其是会影响图片的展示方式（比如图片轮播，CSS 背景图和媒体查询）。但是我认为 JS 和 CSS 只是展示图片的方式。在页面加载的过程中，应当先让图片和文字先展示，而不是试图保证 JS 和 CSS 更快地下载完成。

这段话可谓字字珠玑。此外，雅虎军规和 Google 官方的最佳实践也都将图片优化列为前端性能优化必不可少的环节——图片优化的优先级可见一斑。

就图片这块来说，与其说我们是在做“优化”，不如说我们是在做“权衡”。因为我们要做的事情，就是去压缩图片的体积（或者一开始就选取体积较小的图片格式）。但这个优化操作，是以牺牲一部分成像质量为代价的。因此我们的主要任务，是尽可能地去寻求一个质量与性能之间的平衡点。

#2018 年，图片依然很大

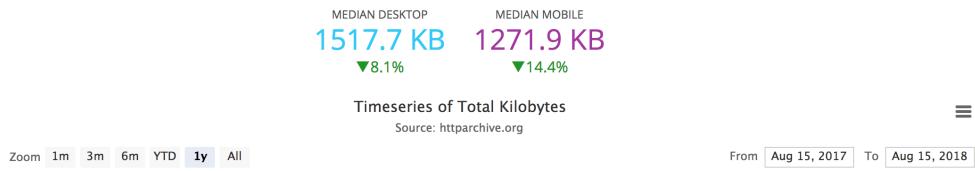
这里先给大家介绍 [HTTP-Archive](#) 这个网站，它会定期抓取 Web 上的站点，并记录资源的加载情况、Web API 的使用情况等页面的详细信息，并会对这些数据进行处理和分析以确定趋势。通过它我们可以实时地看到世界范围内的 Web 资源的统计结果。

截止到 2018 年 8 月，过去一年总的 web 资源的平均请求体积是这样的：

Total Kilobytes

The sum of transfer size kilobytes of all resources requested by the page.

See also: [Page Weight](#)

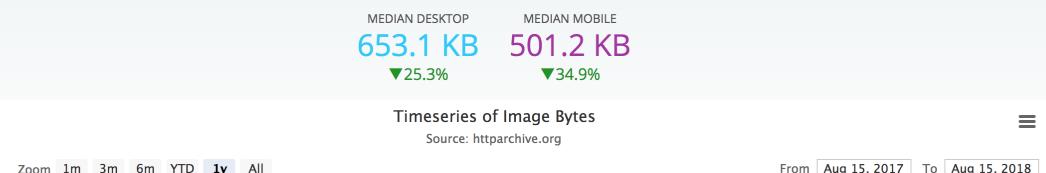


而具体到图片这一类的资源，平均请求体积是这样的：

Image Bytes

The sum of transfer size kilobytes of all external images requested by the page. An external image is identified as a resource with the `png`, `gif`, `jpg`, `jpeg`, `webp`, `ico`, or `svg` file extensions or a MIME type containing `image`.

See also: [Page Weight](#)



当然，随着我们工程师在性能方面所做的努力越来越有成效，平均来说，不管是资源总量还是图片体积，都在往越来越轻量的方向演化。这是一种值得肯定的进步。

但同时我们不得不承认，如图所示的这个图片体积，依然是太大了。图片在所有资源中所占的比重，也足够“触目惊心”了。为了改变这个现状，我们必须把图片优化提上日程。

#不同业务场景下的图片方案选型

时下应用较为广泛的 Web 图片格式有 `JPEG/JPG`、`PNG`、`WebP`、`Base64`、`SVG` 等，这些格式都是很有故事的，值得我们好好研究一把。此外，老生常谈的雪碧图（CSS Sprites）至今也仍在一线的前端应用中发光发热，我们也会有所提及。

不谈业务场景的选型都是耍流氓。下面我们就结合具体的业务场景，一起来解开图片选型的神秘面纱！

1. 前置知识：二进制位数与色彩的关系

- 在计算机中，像素用二进制数来表示。不同的图片格式中像素与二进制位数之间的对应关系是不同的。一个像素对应的二进制位数越多，它可以表示的颜色种类就越多，成像效果也就越细腻，文件体积相应也会越大。
- 一个二进制位表示两种颜色（0|1 对应黑|白），如果一种图片格式对应的二进制位数有 n 个，那么它就可以呈现 2^n 种颜色。

2. JPEG/JPG

关键字：有损压缩、体积小、加载快、不支持透明

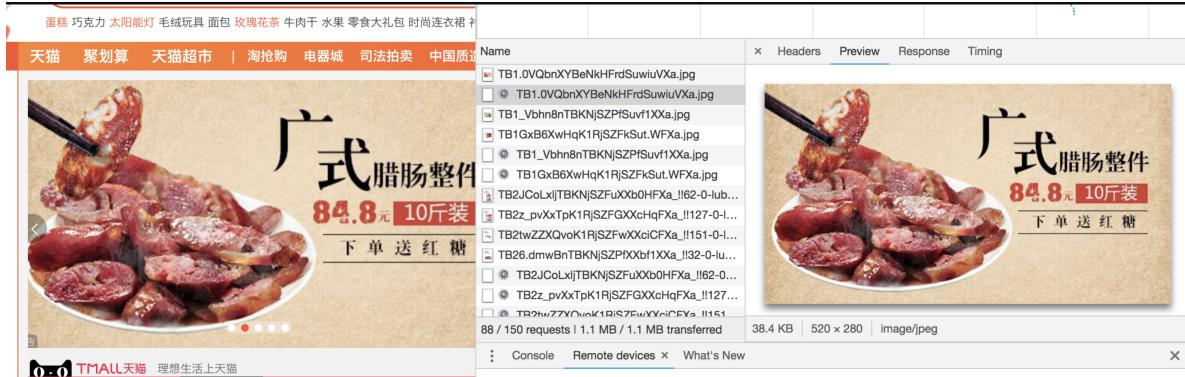
2.1 JPG 的优点

`JPG` 最大的特点是有损压缩。这种高效的压缩算法使它成为了一种非常轻巧的图片格式。另一方面，即使被称为“有损”压缩，`JPG`的压缩方式仍然是一种高质量的压缩方式：当我们把图片体积压缩至原有体积的 50% 以下时，`JPG` 仍然可以保持住 60% 的品质。此外，`JPG` 格式以 24 位存储单个图，可以呈现多达 1600 万种颜色，足以应对大多数场景下对色彩的要求，这一点决定了它压缩前后的质量损耗不容易被我们人类的肉眼所察觉——前提是用对了业务场景。

2.2 使用场景

- `JPG` 适用于呈现色彩丰富的图片，在我们日常开发中，`JPG` 图片经常作为大的背景图、轮播图或 `Banner` 图出现。

- 两大电商网站对大图的处理，是 JPG 图片应用场景的最佳写照：
- 打开淘宝首页，我们可以发现页面中最醒目、最庞大的图片，一定是以 .jpg 为后缀的：



京东首页也不例外：



使用 JPG 呈现大图，既可以保住图片的质量，又不会带来令人头疼的图片体积，是当下比较推崇的一种方案。

2.3 JPG 的缺陷

有损压缩在上文所展示的轮播图上确实很难露出马脚，但当它处理**矢量图形**和**Logo**等线条感较强、颜色对比强烈的图像时，人为压缩导致的图片模糊会相当明显。

此外，JPEG 图像**不支持透明度处理**，透明图片需要召唤 PNG 来呈现。

3. PNG-8 与 PNG-24

关键字：**无损压缩、质量高、体积大、支持透明**

3.1 PNG 的优点

- **PNG**（可移植网络图形格式）是一种无损压缩的高保真的图片格式。8 和 24，这里都是二进制数的位数。按照我们前置知识里提到的对应关系，8 位的 PNG 最多支持 256 种颜色，而 24 位的可以呈现约 1600 万种颜色。
- **PNG** 图片具有比 JPG 更强的色彩表现力，对线条的处理更加细腻，对透明度有良好的支持。它弥补了上文我们提到的 JPG 的局限性，唯一的 BUG 就是**体积太大**。

3.2 PNG-8 与 PNG-24 的选择题

- 什么时候用 **PNG-8**，什么时候用 **PNG-24**，这是一个问题。
- 理论上来说，当你追求最佳的显示效果、并且不在意文件体积大小时，是推荐使用 **PNG-24** 的。
- 但实践当中，为了规避体积的问题，我们一般不用PNG去处理较复杂的图像。当我们遇到适合 PNG 的场景时，也会优先选择更为小巧的 PNG-8。

- 如何确定一张图片是该用 PNG-8 还是 PNG-24 去呈现呢？好的做法是把图片先按照这两种格式分别输出，看 PNG-8 输出的结果是否会带来肉眼可见的质量损耗，并且确认这种损耗是否在我们（尤其是你的 UI 设计师）可接受的范围内，基于对比的结果去做判断。

3.3 应用场景

- 前面我们提到，复杂的、色彩层次丰富的图片，用 PNG 来处理的话，成本会比较高，我们一般会交给 JPG 去存储。
- 考虑到 PNG 在处理线条和颜色对比度方面的优势，我们主要用它来呈现小的 Logo、颜色简单且对比强烈的图片或背景等。
- 此时我们再次把目光转向性能方面堪称业界楷模的淘宝首页，我们会发现它页面上的 Logo，无论大小，还真的都是 PNG 格式：

主 Logo：

The screenshot shows the Taobao homepage with the main logo. To the right, the browser's developer tools Network tab is open, showing the file structure for the logo. The main file listed is "TB1UDHOcwoQMeJjy0FoXXcShVXa-286-118.png". Below it are several other PNG files, all with names starting with "TB1". The preview pane shows the large red "淘宝网" logo and the "Taobao.com" text.

较小的 Logo：

The screenshot shows a smaller version of the Taobao logo. The browser's developer tools Network tab is open, showing the file structure for this smaller logo. The main file listed is "TB1Z_HcQFXXXc7apXXXXXXXXX-26-71.png". Below it are other PNG files. The preview pane shows a smaller version of the red logo.

颜色简单、对比度较强的透明小图也在 PNG 格式下有着良好的表现：

The screenshot shows a small graphic element from the Taobao homepage, featuring sunglasses and text. The browser's developer tools Network tab is open, showing the file structure for this graphic. The main file listed is "TB1PKzHvJcnBKNjSZR0XXcFqFXa-204-82.png". Below it are other PNG files. The preview pane shows the graphic element.

4. SVG

关键字：文本文件、体积小、不失真、兼容性好

SVG（可缩放矢量图形）是一种基于 XML 语法的图像格式。它和本文提及的其它图片种类有着本质的不同：SVG 对图像的处理不是基于像素点，而是基于对图像的形状描述。

4.1 SVG 的特性

和性能关系最密切的一点就是：SVG 与 PNG 和 JPG 相比，文件体积更小，可压缩性更强。

当然，作为矢量图，它最显著的优势还是在于**图片可无限放大而不失真**这一点上。这使得 SVG 即使是被放到视网膜屏幕上，也可以一如既往地展现出较好的成像品质——1 张 SVG 足以适配 n 种分辨率。

此外，**SVG 是文本文件**。我们既可以像写代码一样定义 SVG，把它写在 HTML 里、成为 DOM 的一部分，也可以把对图形的描述写入以 .svg 为后缀的独立文件（SVG 文件在使用上与普通图片文件无异）。这使得 SVG 文件可以被非常多的工具读取和修改，具有较强的**灵活性**。

SVG 的局限性主要有两个方面，一方面是它的渲染成本比较高，这点对性能来说是很不利的。另一方面，SVG 存在着其它图片格式所没有的学习成本（它是可编程的）。

4.2 SVG 的使用方式与应用场景

SVG 是文本文件，我们既可以像写代码一样定义 SVG，把它写在 HTML 里、成为 DOM 的一部分，也可以把对图形的描述写入以 .svg 为后缀的独立文件（SVG 文件在使用上与普通图片文件无异）。

- 将 SVG 写入 HTML：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
    <svg xmlns="http://www.w3.org/2000/svg" width="200" height="200">
        <circle cx="50" cy="50" r="50" />
    </svg>
</body>
</html>
```

- 将 SVG 写入独立文件后引入 HTML：

```

```

在实际开发中，我们更多用到的是后者。很多情况下设计师会给我们 SVG 文件，就算没有设计师，我们还有非常好用的[在线矢量图形库](#)。对于矢量图，我们无须深究过多，只需要对其核心特性有所掌握、日后在应用时做到有迹可循即可。

5. Base64

关键字：**文本文件、依赖编码、小图标解决方案**

Base64 并非一种图片格式，而是一种编码方式。Base64 和雪碧图一样，是作为小图标解决方案而存在的。在了解 Base64 之前，我们先来了解一下雪碧图。

5.1 前置知识：最经典的小图标解决方案——雪碧图（CSS Sprites）

雪碧图、CSS 精灵、CSS Sprites、图像精灵，说的都是这个东西——一种将小图标和背景图像合并在一张图片上，然后利用 CSS 的背景定位来显示其中的每一部分的技术。

MDN 对雪碧图的解释已经非常到位：

图像精灵（sprite，意为精灵），被运用于众多使用大量小图标的网页应用之上。它可取图像的一部分来使用，使得使用一个图像文件替代多个小文件成为可能。相较于一个小图标一个图像文件，单独一张图片所需的 HTTP 请求更少，对内存和带宽更加友好。

我们几乎可以在每一个有小图标出现的网站里找到雪碧图的影子（下图截取自京东首页）：

| Name | x Headers Preview Response Timing |
|-------------------------|--|
| sprite.head@2x.png | |
| mobile_qrcode@2x.png | |
| sprite.service.png | |
| sprite-photo-search.png | |
| sprite.user.png |  |
| blank.png | |
| sprite.seckill@2x.png | |
| 58004d6dN2927f0f7.png | |
| toolbars.png | |

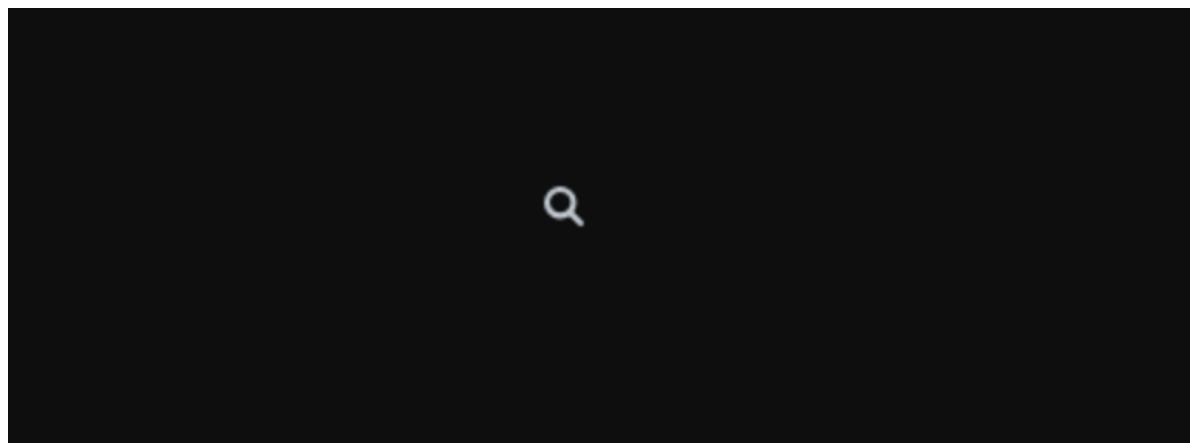
和雪碧图一样，Base64 图片的出现，也是为了减少加载网页图片时对服务器的请求次数，从而提升网页性能。**Base64 是作为雪碧图的补充而存在的。**

5.2 理解 Base64

通过我们上文的演示，大家不难看出，每次加载图片，都是需要单独向服务器请求这个图片对应的资源的——这也就意味着一次 HTTP 请求的开销。

Base64 是一种用于传输 8Bit 字节码的编码方式，通过对图片进行 Base64 编码，我们可以直接将编码结果写入 HTML 或者写入 CSS，从而减少 HTTP 请求的次数。

我们来一起看一个实例，现在我有这么一个小小的放大镜 Logo：



它对应的链接如下：

```
https://user-gold-cdn.xitu.io/2018/9/15/165db7e94699824b?w=22&h=22&f=png&s=3680
```

按照一贯的思路，我们加载图片需要把图片链接写入 img 标签：

```

```

- 浏览器就会针对我们的图片链接去发起一个资源请求。
- 但是如果我们对这个图片进行 Base64 编码，我们会得到一个这样的字符串：

data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAABYAAAACAYAADEtGw7AAAMJG1DQ1BJQ0MgUHJVzm1szQAASImV1wdUU8kagOewJCQktEAEpitEBCnSpdfQpQo2QhJIKDEKBHU7uqjgw1ARwYquitjwAshiw14wwd4fifKgo62LBhsqbFNDV89477z9n7v3yz9/mcydMwoAehxbJMpFNQDIExaI48MCmeNT05ikr4AEClAKRgEamyMRBcTFRQEoQ+9/yrubAJG9r9nLfp3c/19fk8uTcABA4iBnCCWCPMiHAMddOCJxAQCEXqg3m1YggkyEWQjtMUwQsrmMsxTsIEMMBuFjbRLjgyCnA6BCzbPFWQCoYfJiFnKyob+1pZAdhVyBEHIZZF8On82F/BnyqlY8qZDVRsFBz3znJ+sfPjOGfbLzwCosqeuuksEcisixPeP/n17/Lxm50qEYZrBr+eLweFnNsnnLmRopYyrk88KmmFjIWpcvc7hyexk/4uvdk5t2HzisIDhngAEASuwygyMhG0A2FebGRCh1vpmcUBzKOPdoocATagYi3LFU+OV/tHpPElIwhCzxJYmpSAu5SgnLnRj6PNeSzqyifmKLIE20rfCTHQfadf+F+SxkCptHlexA+KgbIRS+NlocP/HAOZ4tB4hQ1mnicZqgvz4gtYMuqo4rd1+ehCnlzATwxX+MEKeZLxUUN5cnnB1Yq6SGKeMEmZP1YuKgiMV47djsqnu9pjzbzcmjneFHKrpDBhaGxfAvxsinpxICqIS1TkhtnsyPiFHFXwxFgkAWYAIpbB1gKsgGgtbeh174s9ETcthADLIAD9grNUMjuuQ9QvhMAEXgL0g8IBkeFyjv5YFCqP8yrFU87UGmvLdQPjIHPiGCBjBLvwt1Y8SDkdLbo+hRvBTDa7MNRC2wd9POqb6kI4YQgwmmhNDiTa4Pu6Le+NR80kPmzPugXs05fxNnvCE0E54RLhb6CDCmSIoFv+QORNEgw6YY6iyuoqvq8MtovdXPBd3gf6hb5yb6wn7fAyMFID7wdiuUPT9rtLhir/NpdIX2ZGMkkeQ/cnWP2Ugm53v61fq1wzvxjv5ZQzPVtCw1Y9egr6bPy58R/5ois3GdmLnsJPYBawZawBM7DjwiF3Gjsp4eG081q+NoWjx8txyoB/BT/HYyipiWzM41jn20H5w9oEC3vQC2ccSNFU0QyzI4hcwA+BuzwoyhByHUUxnRye4i8r2fsxw8oYh39MRxsVvuvwTAHiwQmXwnx0b7kfHngBAf/dNZ/YaLvsVABxt40jFhQodLnsQAAWow9FDxjBvcsavuQm3IA38AchIAlegkSQCiBdOefDdSoG08AsMB+UgDKwAqwBVWAT2Ap2gj3gAGgAzeAkOAsugtzwa9yDa6UbvAB94B0YQBCEhNAQoqKHGCMwiB3ijHggvkgIEoXEI61OpKFCBEPMgtzgjqh5ugvsgwpRx5HjiAnkqtIO3IH6UR6knfijxRDqag2aoahoqNRDzQAjUQT0UloFpqPFqEL0WvoJvqd7kbr0ZpojFQG2oG+QPsxgk1idMWes8c8sCasFkvDMjExNgcrxSqwGmwv1gT/6wtYB9aLfcSJ0B1n4vZwvYbjSTghz8fn4EvxKnwnXo+fxq/hnXgf/pVAIxgQ7AheBBzhPCGLMI1QQqggbCccJpyB30434R2RSQQQryju8NtLjWYTzxKxejcQ9xFPENuJXCR+Eomkr7Ij+zBiswxsAamEtI60m3ScdjXUTfqqoqpirOkseqqSpijUKvapunmlckz1qspT1QgyBtmC7EWOJXPJM8jLydviteqr5G7yAEWTYkxxoSRSSinZKwUVzQz1PuUN6qqqqaqnqrjvAWq81qrVfern1ftvp1I1aLaUooE61s6jLqdUoJ6h3qGxqNZknzp6XRCmjLaLw0U7ShTA9qdDUHZYav22uwrVavdpvtzfqZHUL9QD1yePF6hxqB9WvqPdqkDUSNYI02BpzNKo1jmjc0ujxpGs6acqz5mk1dyleuhzmrZjy1irRIurtvBrq9Yprs46rjejb9E59AX0bfQz9G5toraVNks7w7tMe492q3afjpboGJ1knek61TpHDtOYGMOSWWLkMpYzDjBumj6NMBwRMII3YsmIVsoujnhivo1Lxx5enw6q7T/eG7ic9p16ixo7esr0Gvqf6uL6t/jj9afob9c/o947UHuK9kjoydosBkxcNUANbg3idmqzbds4b9bsaGYYzigzXGz4y7DViGPkbzRutNjpm1GNMN/Y1FhivNj5u/jypwwxg5jirmaezfsYgJuEmUpMtjQ0mA6Zwpkmmxab7TB+YUCw8zDLNpu1mPWZG5tHm88yrzo/a0G28LDgw6y10Gfx3tLkmsVykWWd5TMrXSuWVZFndv9a5q1n3w+dY31druijYdNjs0Gmzzb1Nbvlm9bbXvFDrVzsxPYbbBrH0UY5T1KOKpm1c17qn2Afaf9nx2na8MhyqHYochh5wjz0wmjv44+n/qro6tjruM2x3towk4RTsVOTU6vnw2doc7Vztddac6hLnNdG11ejbEBwxuzccxtv7prt0si1xbXL27ubmK3vW497ubu6e7r3w95ahvEesz1009j8Az0n0vz7PnRy82rwouA19/e9t453ru8n421Gssbu21s14+pD9tni0+HL9M33xezb4efir/br8bvkb+ZP9d/u//TAJuA7IDdAS8DHQPFgYcd3wd5bc00ohGMBYcf1wa3hmiFjIVuhTwMNQ3NCq0L7QtzdZsd1Kceb4zvjL8fsuQxWHVsvoi3CnMr5yOpEymRFZFPoqyjRJHNUwj0RHRq6Lvx1jECGmaYkEsk3Zv7IM4q7j8ud/Gecffjase9yTeKX5w/LkEeskuhF0j7xIDE5cn3kuyTpImtssrj09Mrk1+nxKcup7SMX70+NnjL6XqpwpSG9NIac1p29P6J4RMWD0he6LrxJKJNydztzo+6cjk/cm5k490UZ/CnnIwnZCekr4r/TM7113d7s9gazP60MEcdZyxnD9ua5PTwfxjnvaazPznnmsyyfrFvZPxw/fgw/vxAkqBK8yg7P3pT9Pic2Z0foYG5k7r481bz0vcNCLwg08PRUo6nTp7al7Eq1oo58r/w1+x3iSPF2CSKZJGks0Iah7MtSa+kv0s5C38Lqwg/Tkqcdnk45xtj98gzbGUtmPC0KLfpTj6tM7N11sms+bM6ZwfM3jIHmZMxp2Wu2dyFc7vhc3b0z8yp2f+n8wOxeXFbxekLGhaalhw3sKuX8j+qStRKxGx3FrkvwjTYnyxYHhrEpc165z8LewxixzLKso+7yUs/Tir06/Vv46uCxzwetyt+UbVxBXCFFcx0m3cme5zn1Redeq6Fx1q5mrS1e/XTN1zyWkmrbw11LwStd2VEZVnq4zx7di3ecqftWN6sDqfesn1i9z/34Dd8Pvjf4b924y3FS26dnwebbw8K21NdY11RsJw4t3Ppkw/K2c795/Fa7xx972Fyv04Q70nbG7zxd615bu8tg1/I6te5a17N74u62Pcf7Gvfa792yj7GvbD/YL93//Pf0328eiDzQctDj4N5DFofWH6YfLq1H6mfu9zXwGzoauXvbj0Qcawnybjr8h8Mf05pNmquP6hxdfoxyb0GxweNFx/tPiE70nsw62duypeXeqfGnRp8ed7r1tOsZ82dDz546F3Du+Hmf880Xvc4cuehxseGS26x6y66xD//p+ufhvrwf+ivuvxrbPNua2se2H7vqd/xkteBrZ6+zr1+6EXOj/wbSzdu3Jt7quM29/ex07p1XdwvvDtybd59wv/SBxo0KhWPa/518699Hw4drzuDoy8/Snh0r4vT9eKx5Phn7ovPaE8qnh0/rx3m/Ky5j7sn7fmE590vRC8Gekv+0vxr/Uvr14f+9v/7ct/4vu5X41eDr5e+0Xuz4+2Yty39cf0P3+w9G3hf+kHvw86PHh/PfUr59Hrg2mf58ovn1+avkz+vt+YNzgoYovz8qMABhuamQnA6x0A0FLh2aENAmoExd1Mlojipikn8j9YcX+TixsAO/wBSJ0hQBQ8o2yEZQIyFb51R/BEf4C6uAw3pUgyXZwVvqjwxkL4MDj4xhAAuhMAX8SDgwMbBge/bIPj3gHgRL7itigT2R10s40M2rpfgh/134RUCT2MnhaNAAAB901EQVQ4Ee1Tv0tbURQ+5yVqFVhs4pBiosAp1mAxDq05sfoKrH072QXN6HdnMTVbyolshH8D+xLg8UkhjY/tj1ERiQi1Cpkfbmn3w08e0td183Nu5x7z/m+737vnHeJHTz9d4CDLhARK1esfschwwf6TSQnRLwnSq2mp20nQTw3bxS2D349177bAijuAt0oJnfEtjikj392c6zotsfhfJfdfue+jn1ewzwe6HL6Q0yjqHyE6zALr+ek9b12rvfsc2wXKwskvAZQb1bxysYL1nu7UJ1H2BKiq+bfafs1p12jd4bHHPLcdwumQi4bBuiP+Gov3vwaqEMQqz6EER9fHjwyASMGVdU6KeB2F8jjh9cw2+ss5Hg0jodUTXRNF1EMYvzPyjBva0YCLzpc0E2pBBT

mokgmjcz5hz17RJEz/vV2oLDcajR6XvHdYT0qTdzQPFd7s9D/7/gotYhdqn/Chy3ovQrfMVMUwh3HpE5
1rLaGqw+FMNh97aa80SiAb1C9R1EN/AYejoEpGgXpARyEbzkY4i/NYkHCmux/f3GgBP618Ejivp40n
D8/c3k2Mm3Uu2pUVIVkBET3vVIpV/FYhea4660wi7IFPP140jTcfKojaBNB6mp8Wkvzjc8b7HTPvkyeh
YKh5NwxGbiP52wD7x76CB/EiWtaCMHwyUAAAAASUVORK5CYII=

字符串比较长，我们可以直接用这个字符串替换掉上文中的链接地址。你会发现浏览器原来是可以理解这个字符串的，它自动就将这个字符串解码为了一个图片，而不需再去发送 `HTTP` 请求。

5.3 Base64 的应用场景

既然 `Base64` 这么棒，我们何不把大图也换成 `Base64` 呢？

- 这是因为，`Base64` 编码后，图片大小会膨胀为原文件的 $4/3$ （这是由 `Base64` 的编码原理决定的）。如果我们把大图也编码到 `HTML` 或 `CSS` 文件中，后者的体积会明显增加，即便我们减少了 `HTTP` 请求，也无法弥补这庞大的体积带来的性能开销，得不偿失。
- 在传输非常小的图片的时候，`Base64` 带来的文件体积膨胀、以及浏览器解析 `Base64` 的时间开销，与它节省掉的 `HTTP` 请求开销相比，可以忽略不计，这时候才能真正体现出它在性能方面的优势。

因此，`Base64` 并非万全之策，我们往往在一张图片满足以下条件时会对它应用 `Base64` 编码：

- 图片的实际尺寸很小（大家可以观察一下使用的页面的 `Base64` 图，几乎没有超过 `2kb` 的）
- 图片无法以雪碧图的形式与其它小图结合（合成雪碧图仍是主要的减少 `HTTP` 请求的途径，`Base64` 是雪碧图的补充）
- 图片的更新频率非常低（不需我们重复编码和修改文件内容，维护成本较低）

5.4 Base64 编码工具推荐

这里最推荐的是利用 `webpack` 来进行 `Base64` 的编码——`webpack` 的 [url-loader](#) 非常聪明，它除了具备基本的 `Base64` 转码能力，还可以结合文件大小，帮我们判断图片是否有必要进行 `Base64` 编码。

除此之外，市面上免费的 `Base64` 编解码工具种类是非常多样化的，有很多网站都提供在线编解码的服务，大家选取自己认为顺手的工具就好。

6. WebP

关键字：年轻的全能型选手

`WebP` 是今天在座各类图片格式中最年轻的一位，它于 2010 年被提出，是 Google 专为 Web 开发的一种旨在加快图片加载速度的图片格式，它支持有损压缩和无损压缩。

6.1 WebP 的优点

`WebP` 像 `JPEG` 一样对细节丰富的图片信手拈来，像 `PNG` 一样支持透明，像 `GIF` 一样可以显示动态图片——它集多种图片文件格式的优点于一身。

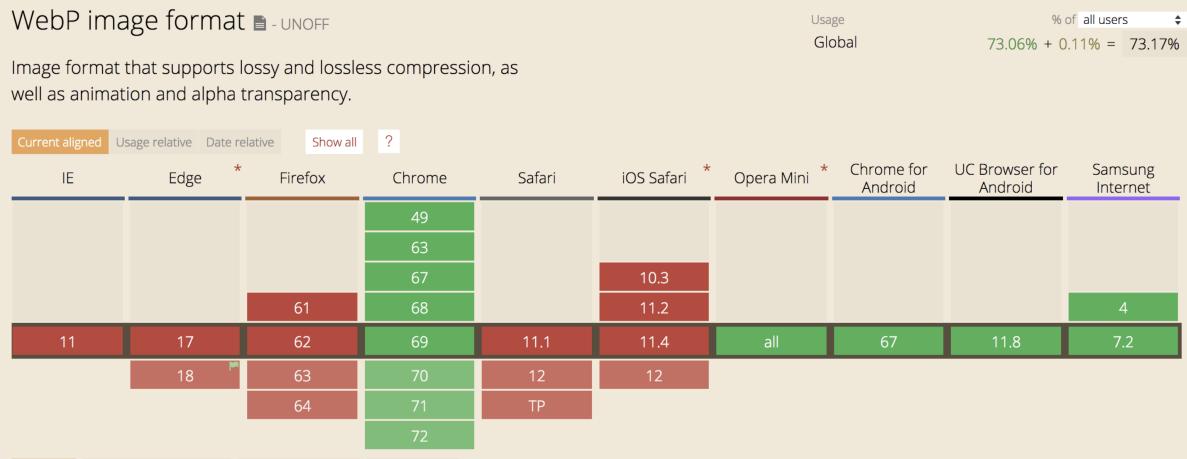
`WebP` 的官方介绍对这一点有着更权威的阐述：

与 `PNG` 相比，`WebP` 无损图像的尺寸缩小了 26%。在等效的 SSIM 质量指数下，`WebP` 有损图像比同类 `JPEG` 图像小 25-34%。无损 `WebP` 支持透明度（也称为 alpha 通道），仅需 22% 的额外字节。对于有损 `RGB` 压缩可接受的情况，有损 `WebP` 也支持透明度，与 `PNG` 相比，通常提供 3 倍的文件大小。

我们开篇提到，图片优化是质量与性能的博弈，从这个角度看，`WebP` 无疑是真正的赢家。

6.2 WebP 的局限性

`WebP` 纵有千般好，但它毕竟太年轻。我们知道，任何新生事物，都逃不开兼容性的大坑。现在是 2018 年 9 月，`WebP` 的支持情况是这样的：



坦白地说，虽然没有特别惨（毕竟还有亲爹 Chrome 在撑腰），但也足够让人望而却步了。

此外，WebP 还会增加服务器的负担——和编码 JPG 文件相比，编码同样质量的 WebP 文件会占用更多的计算资源。

6.3 WebP 的应用场景

- 现在限制我们使用 WebP 的最大问题不是“这个图片是否适合用 WebP 呈现”的问题，而是“浏览器是否允许 WebP”的问题，即我们上文谈到的兼容性问题。具体来说，一旦我们选择了 WebP，就要考虑在 Safari 等浏览器下它无法显示的问题，也就是说我们需要准备 PlanB，准备降级方案。
- 目前真正把 WebP 格式落地到网页中的网站并不是很多，这其中淘宝首页对 WebP 兼容性问题的处理方式就非常有趣。我们可以打开 Chrome 的开发者工具搜索其源码里的 WebP 关键字：

我们会发现检索结果还是挺多的（单就图示的加载结果来看，足足有 200 多条），下面大家注意一下这些 WebP 图片的链接地址（以其中一个为例）：

```

```

.webp 前面，还跟了一个 .jpg 后缀！

我们现在先大胆地猜测，这个图片应该至少存在 jpg 和 webp 两种格式，程序会根据浏览器的型号、以及该型号是否支持 WebP 这些信息来决定当前浏览器显示的是 .webp 后缀还是 .jpg 后缀。带着这个预判，我们打开并不支持 WebP 格式的 Safari 来进入同样的页面，再次搜索 WebP 关键字：

Safari 提示我们找不到，这也是情理之中。我们定位到刚刚示例的 WebP 图片所在的元素，查看一下它在 Safari 里的图片链接：

```

```

- 我们看到同样的一张图片，在 Safari 中的后缀从 `.webp` 变成了 `.jpg`！看来果然如此——站点确实是先进行了兼容性的预判，在浏览器环境支持 `WebP` 的情况下，优先使用 WebP 格式，否则就把图片降级为 JPG 格式（本质是对图片的链接地址作简单的字符串切割）。
- 此外，还有另一个维护性更强、更加灵活的方案——把判断工作交给后端，由服务器根据 `HTTP` 请求头部的 `Accept` 字段来决定返回什么格式的图片。当 `Accept` 字段包含 `image/webp` 时，就返回 `WebP` 格式的图片，否则返回原图。这种做法的好处是，当浏览器对 WebP 格式图片的兼容支持发生改变时，我们也不用再去更新自己的兼容判定代码，只需要服务端像往常一样对 `Accept` 字段进行检查即可。

由此也可以看出，我们 `WebP` 格式的局限性确实比较明显，如果决定使用 `WebP`，兼容性处理是必不可少的。

#小结

- 不知道大家有没有注意到这一点：在图片这一节，我用到的许多案例图示，都是源于一线的电商网站。
- 为什么这么做？因为图片是电商平台的重要资源，甚至有人说“做电商就是做图片”。淘宝和京东，都是流量巨大、技术成熟的站点，它们在性能优化方面起步早、成效好，很多方面说是教科书般的案例也不为过。
- 这也是非常重要的一个学习方法。在开篇我提到，性能优化不那么好学，有很大原因是因为这块的知识不成体系、难以切入，同时技术方案又迭代得飞快。当我们不知道怎么切入的时候，或者说当我们面对一个具体的问题无从下手的时候，除了翻阅手中的书本（很可能是已经过时的）和网络上收藏的文章（也许没那么权威），现在是不是又多了“打开那些优秀的网站看一看”这条路可以走了呢？
- 好了，至此，我们终于结束了图片优化的征程。下面，我们以存储篇为过渡，进入 JS 和 CSS 的世界！

#四、存储篇 1：浏览器缓存机制介绍与缓存策略剖析

缓存可以减少网络 `IO` 消耗，提高访问速度。浏览器缓存是一种操作简单、效果显著的前端性能优化手段。对于这个操作的必要性，Chrome 官方给出的解释似乎更有说服力一些：

通过网络获取内容既速度缓慢又开销巨大。较大的响应需要在客户端与服务器之间进行多次往返通信，这会延迟浏览器获得和处理内容的时间，还会增加访问者的流量费用。因此，缓存并重复利用之前获取的资源的能力成为性能优化的一个关键方面。

- 很多时候，大家倾向于将浏览器缓存简单地理解为“`HTTP 缓存`”。但事实上，浏览器缓存机制有四个方面，它们按照获取资源时请求的优先级依次排列如下：

1. `Memory Cache`
2. `Service Worker Cache`
3. `HTTP Cache`
4. `Push Cache`

大家对 `HTTP Cache`（即 `Cache-Control`、`expires` 等字段控制的缓存）应该比较熟悉，如果对其它几种缓存可能还没什么概念，我们可以先来看一张线上网站的 `Network` 面板截图：

| Name | Status | Type | Initiator | Size | Time | Waterfall |
|--|--------|-----------|--|----------------------|---------|-----------|
| ??s/8.6.5/plugin/aplus_client.js.aplus_cplugin/0.4...s,s/8.6.5/plugin/apl... | 200 | fetch | workbox-core.prod.js:1 | (from disk cache) | 7 ms | |
| TB1Z_HcQFXXXXc7epXXXXXXXXXX-26-71.png | 200 | png | ??kissy/k/6.2.4/event-custom-min.js... | (from ServiceWorker) | 0 ms | |
| data:image/webp;base... | 200 | webp | ??kissy/k/6.2.4/index.js.kg/glob... | (from memory cache) | 0 ms | |
| ?name=tbs&cna=3rEEzXNfmwCASp4S0AvTcwJ&n=25E6%2...0266... | 200 | script | ??kissy/k/6.2.4/seed-min.js.kg/glob... | 781 B | 114 ms | |
| recommend2.htm?appId=20140506002%2C20140506001%2C2...201608... | 200 | script | ??kissy/k/6.2.4/seed-min.js.kg/glob... | 5.0 KB | 314 ms | |
| data:image/png;base... | 200 | png | ??kissy/k/6.2.4/event-custom-min.js... | (from memory cache) | 0 ms | |
| mget.htm?callback=jsonpXctr106&ice_sid=1947787&tc...c=&count=&env... | 200 | script | ??kissy/k/6.2.4/seed-min.js.kg/glob... | 638 B | 72 ms | |
| jtracker.3?kissyVersion=6.2.4&screen=1440x900&ua=..._0.3%2FSW_REG... | 200 | gif | ??kissy/k/6.2.4/seed-min.js.kg/glob... | 158 B | 49 ms | |
| tbinindex.2016201863.1?gmkey=&gokey=frame%3D1%26_hng...J&spm-cn... | 200 | gif | VM96219:6 | 135 B | 51 ms | |
| g!f?logtype=1&tlt=%E6%B7%98%E5%AE%9D%E7%BD%91%...525... | 200 | gif | VM96219:6 | 266 B | 55 ms | |
| TB1w!pgbCzqK1RjS2FpxakSXa-160-280.jpg_400x400q90.jpg_webp | 200 | webp | ??kissy/k/6.2.4/event-custom-min.js... | (from ServiceWorker) | 0 ms | |
| 3rjEEzXNfmwCASp4S0AvTcwJ | 101 | websocket | VM96219:5 | 0 B | Pending | |
| TB1O1qOXSzqK1RjS2FHXb3CpXa-346-200.png_350x1000q90.jpg | 200 | png | ??kissy/k/6.2.4/event-custom-min.js... | (from ServiceWorker) | 0 ms | |
| TB2BaAVkQUmBKNIS2FOXab2Xa_!!9-0-lubanu.ioo_350x1000q90.ioo | 200 | ioeo | ??kissy/k/6.2.4/event-custom-min.js... | (from ServiceWorker) | 10 ms | |

128 requests | 332 KB transferred | Finish: 25.95 s | DOMContentLoaded: 563 ms | Load: 1.09 s

我们给 `size` 这一栏一个特写：

Size

(from disk cache)

(from ServiceWorker)

(from memory cache)

781 B

5.0 KB

(from memory cache)

638 B

158 B

135 B

266 B

(from ServiceWorker)

0 B

(from ServiceWorker)

(from ServiceWorker)

大家注意一下非数字——即形如“`(from xxx)`”这样的描述——对应的资源，这些资源就是我们通过缓存获取到的。其中，“`from memory cache`”对标到 `Memory Cache` 类型，“`from ServiceWorker`”对标到 `Service Worker Cache` 类型。至于 `Push Cache`，这个比较特殊，是 `HTTP2` 的新特性。

本节将会针对这四个方面各个击破。考虑到 HTTP 缓存是最主要、最具有代表性的缓存策略，也是每一位前端工程师都应该深刻理解掌握的性能优化知识点，我们下面优先针对 HTTP 缓存机制进行剖析。

#HTTP 缓存机制探秘

HTTP 缓存是我们日常开发中最为熟悉的一种缓存机制。它又分为**强缓存**和**协商缓存**。优先级较高的是强缓存，在命中强缓存失败的情况下，才会走协商缓存。

1. 强缓存的特征

强缓存是利用 `http` 头中的 `Expires` 和 `Cache-Control` 两个字段来控制的。强缓存中，当请求再次发出时，浏览器会根据其中的 `expires` 和 `cache-control` 判断目标资源是否“命中”强缓存，若命中则直接从缓存中获取资源，**不会再与服务端发生通信**。

命中强缓存的情况下，返回的 `HTTP` 状态码为 `200`（如下图）。

Request Method: GET
Status Code: ● 200 (from disk cache)

2. 强缓存的实现：从 `expires` 到 `cache-control`

- 实现强缓存，过去我们一直用 `expires`。
- 当服务器返回响应时，在 `Response Headers` 中将过期时间写入 `expires` 字段。像这样：

▼ Response Headers

`access-control-allow-origin: *`
`age: 734080`
`cache-control: max-age=31536000`
`content-length: 42866`
`content-type: image/jpeg`
`date: Tue, 11 Sep 2018 16:12:18 GMT`
`eagleid: 7ccbe04f15374164184436563e`
`expires: Wed, 11 Sep 2019 16:12:18 GMT`

我们给 `expires` 一个特写：

```
expires: wed, 11 Sep 2019 16:12:18 GMT
```

- 可以看到，`expires` 是一个时间戳，接下来如果我们试图再次向服务器请求资源，浏览器就会先对比本地时间和 `expires` 的时间戳，如果本地时间小于 `expires` 设定的过期时间，那么就直接去缓存中取这个资源。
- 从这样的描述中大家也不难猜测，`expires` 是有问题的，它最大的问题在于对“本地时间”的依赖。如果服务端和客户端的时间设置可能不同，或者我直接手动去把客户端的时间改掉，那么 `expires` 将无法达到我们的预期。
- 考虑到 `expires` 的局限性，`HTTP1.1` 新增了 `Cache-Control` 字段来完成 `expires` 的任务。`expires` 能做的事情，`Cache-Control` 都能做；`expires` 完成不了的事情，`Cache-Control` 也能

做。因此，`Cache-Control` 可以视作是 `expires` 的完全替代方案。在当下的前端实践里，我们继续使用 `expires` 的唯一目的就是向下兼容。

现在我们给 `Cache-Control` 字段一个特写：

```
cache-control: max-age=31536000
```

如大家所见，在 `Cache-Control` 中，我们通过 `max-age` 来控制资源的有效期。`max-age` 不是一个时间戳，而是一个时间长度。在本例中，`max-age` 是 31536000 秒，它意味着该资源在 31536000 秒以内都是有效的，完美地规避了时间戳带来的潜在问题。

Cache-Control 相对于 expires 更加准确，它的优先级也更高。当 Cache-Control 与 expires 同时出现时，我们以 Cache-Control 为准。

3. Cache-Control 应用分析

`Cache-Control` 的神通，可不止于这一个小小的 `max-age`。如下的用法也非常常见：

```
cache-control: max-age=3600, s-maxage=31536000
```

s-maxage 优先级高于 max-age，两者同时出现时，优先考虑 s-maxage。如果 s-maxage 未过期，则向代理服务器请求其缓存内容。

这个 `s-maxage` 不像 `max-age` 一样为大家所熟知。的确，在项目不是特别大的场景下，`max-age` 足够用了。但在依赖各种代理的大型架构中，我们不得不考虑代理服务器的缓存问题。`s-maxage` 就是用于表示 `cache` 服务器上（比如 `cache CDN`）的缓存的有效时间的，并只对 `public` 缓存有效。

- 此处应注意这样一个细节：`s-maxage` 仅在代理服务器中生效，客户端中我们只考虑 `max-age`
- 那么什么是 `public` 缓存呢？说到这里，`Cache-Control` 中有一些适合放在一起理解的知识点，我们集中梳理一下：

3.1 public 与 private

- `public` 与 `private` 是针对资源是否能够被代理服务缓存而存在的一组对立概念。
- 如果我们为资源设置了 `public`，那么它既可以被浏览器缓存，也可以被代理服务器缓存；如果我们设置了 `private`，则该资源只能被浏览器缓存。`private` 为默认值。但多数情况下，`public` 并不需要我们手动设置，比如有很多线上网站的 `cache-control` 是这样的：

▼ Response Headers

```
access-control-allow-origin: *
age: 21040
cache-control: max-age=3600, s-maxage=31536000
content-length: 13678
content-type: image/jpeg
```

设置了 `s-maxage`，没设置 `public`，那么 `CDN` 还可以缓存这个资源吗？答案是肯定的。因为明确的缓存信息（例如“`max-age`”）已表示响应是可以缓存的。

3.2 no-store 与 no-cache

- `no-cache` 绕开了浏览器：我们为资源设置了 `no-cache` 后，每一次发起请求都不会再去询问浏览器的缓存情况，而是直接向服务端去确认该资源是否过期（即走我们下文即将讲解的协商缓存的路线）。
- `no-store` 比较绝情，顾名思义就是不使用任何缓存策略。在 `no-cache` 的基础上，它连服务端的缓存确认也绕开了，只允许你直接向服务端发送请求、并下载完整的响应。

4. 协商缓存：浏览器与服务器合作之下的缓存策略

- 协商缓存依赖于服务端与浏览器之间的通信。

协商缓存机制下，浏览器需要向服务器去询问缓存的相关信息，进而判断是重新发起请求、下载完整的响应，还是从本地获取缓存的资源。

如果服务端提示缓存资源未改动（`Not Modified`），资源会被重定向到浏览器缓存，这种情况下网络请求对应的状态码是 `304`（如下图）。

Request Method: GET
Status Code: ● 304 Not Modified

5. 协商缓存的实现：从 `Last-Modified` 到 `Etag`

`Last-Modified` 是一个时间戳，如果我们启用了协商缓存，它会在首次请求时随着 `Response Headers` 返回：

```
Last-Modified: Fri, 27 Oct 2017 06:35:57 GMT
```

随后我们每次请求时，会带上一个叫 `If-Modified-Since` 的时间戳字段，它的值正是上一次 `response` 返回给它的 `Last-modified` 值：

```
If-Modified-Since: Fri, 27 Oct 2017 06:35:57 GMT
```

服务器接收到这个时间戳后，会比对该时间戳和资源在服务器上的最后修改时间是否一致，从而判断资源是否发生了变化。如果发生了变化，就会返回一个完整的响应内容，并在 `Response Headers` 中添加新的 `Last-Modified` 值；否则，返回如上图的 `304` 响应，`Response Headers` 不会再添加 `Last-Modified` 字段。

使用 `Last-Modified` 存在一些弊端，这其中最常见的就是这样两个场景：

- 我们编辑了文件，但文件的内容没有改变。服务端并不清楚我们是否真正改变了文件，它仍然通过最后编辑时间进行判断。因此这个资源在再次被请求时，会被当做新资源，进而引发一次完整的响应——不该重新请求的时候，也会重新请求。
- 当我们修改文件的速度过快时（比如花了 `100ms` 完成了改动），由于 `If-Modified-Since` 只能检查到以秒为最小计量单位的时间差，所以它是感知不到这个改动的——该重新请求的时候，反而没有重新请求了。

这两个场景其实指向了同一个 `bug`——服务器并没有正确感知文件的变化。为了解决这样的问题，`Etag` 作为 `Last-Modified` 的补充出现了。

- `Etag` 是由服务器为每个资源生成的唯一的**标识字符串**，这个标识字符串是基于文件内容编码的，只要文件内容不同，它们对应的 `Etag` 就是不同的，反之亦然。因此 `Etag` 能够精准地感知文件的变化。
- `Etag` 和 `Last-Modified` 类似，当首次请求时，我们在响应头里获取到一个最初的标识符字符串，举个🌰，它可以是这样的：

```
ETag: W/"2a3b-1602480f459"
```

那么下一次请求时，请求头里就会带上一个值相同的、名为 `If-None-Match` 的字符串供服务端比对了：

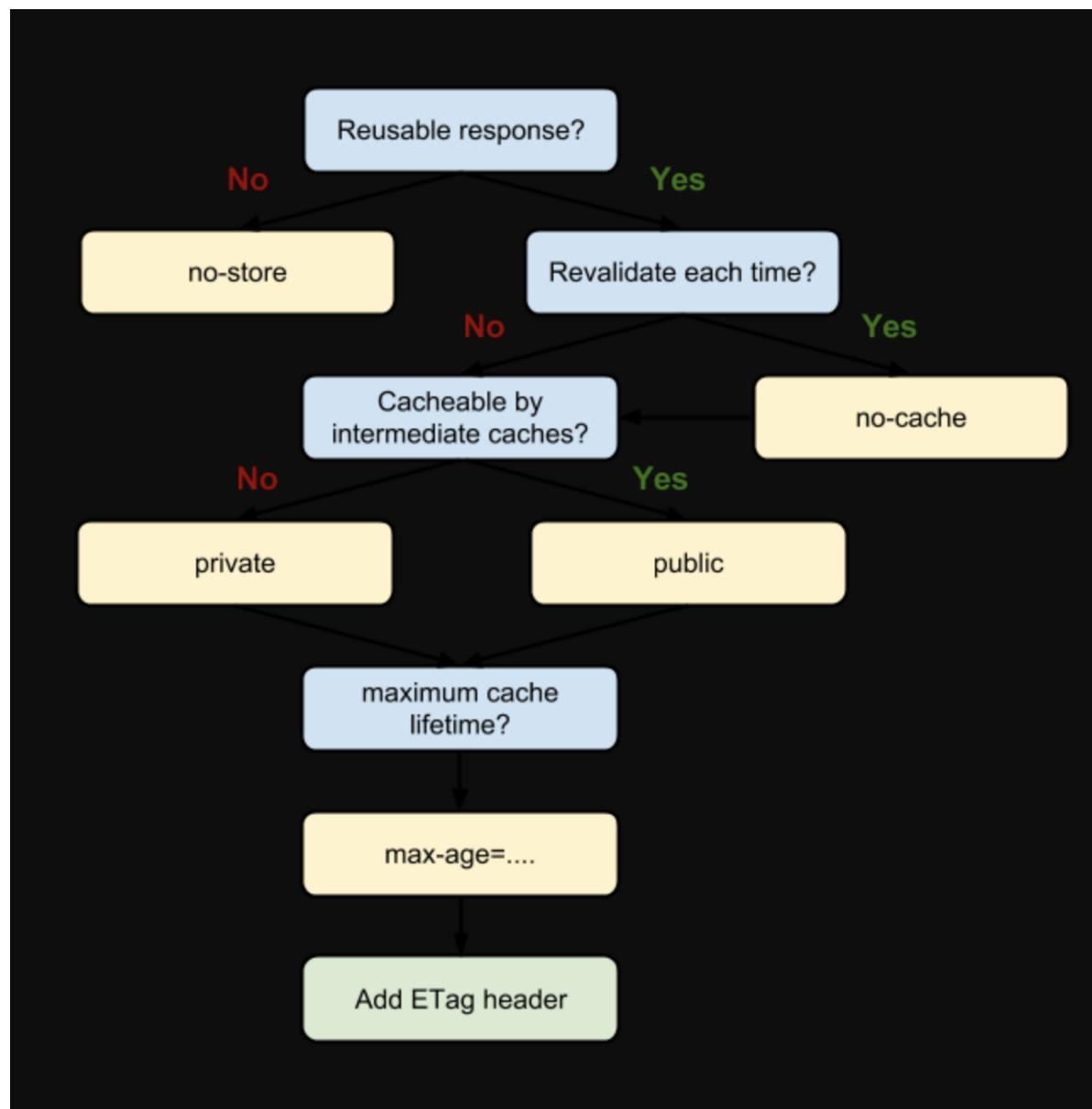
```
If-None-Match: W/"2a3b-1602480f459"
```

`Etag` 的生成过程需要服务器额外付出开销，会影响服务端的性能，这是它的弊端。因此启用 `Etag` 需要我们审时度势。正如我们刚刚所提到的——`Etag` 并不能替代 `Last-Modified`，它只能作为 `Last-Modified` 的补充和强化存在。`Etag` 在感知文件变化上比 `Last-Modified` 更加准确，优先级也更高。当 `Etag` 和 `Last-Modified` 同时存在时，以 `Etag` 为准。

#HTTP 缓存决策指南

行文至此，当代 HTTP 缓存技术用到的知识点，我们已经从头到尾挖掘了一遍了。那么在面对一个具体的缓存需求时，我们到底该怎么决策呢？

走到决策建议这一步，我本来想给大家重新画一个流程图。但是画来画去终究不如 Chrome 官方给出的这张清晰、权威：



我们现在一起解读一下这张流程图：

- 当我们的资源内容不可复用时，直接为 `Cache-Control` 设置 `no-store`，拒绝一切形式的缓存；否则考虑是否每次都需要向服务器进行缓存有效确认，如果需要，那么设 `Cache-Control` 的值为 `no-cache`；否则考虑该资源是否可以被代理服务器缓存，根据其结果决定是设置为 `private` 还是 `public`；然后考虑该资源的过期时间，设置对应的 `max-age` 和 `s-maxage` 值；最后，配置协商缓存需要用到的 `Etag`、`Last-Modified` 等参数。
- 个人非常推崇这张流程图给出的决策建议，也强烈推荐大家在理解以上知识点的基础上，将这张图保存下来、在日常开发中用用看，它的可行度非常高。

OK，走到这里，本节最大的一座山已经被大家翻过去了。接下来的内容会相对比较轻松，大家放松心情，我们继续前行！

#MemoryCache

- `MemoryCache`，是指存在内存中的缓存。从优先级上来说，它是浏览器最先尝试去命中的一种缓存。从效率上来说，它是响应速度最快的一种缓存。
- 内存缓存是快的，也是“短命”的。它和渲染进程“生死相依”，当进程结束后，也就是 tab 关闭以后，内存里的数据也将不复存在。

那么哪些文件会被放入内存呢？

- 事实上，这个划分规则，一直以来是没有定论的。不过想想也可以理解，内存是有限的，很多时候需要先考虑即时呈现的内存余量，再根据具体的情况决定分配给内存和磁盘的资源量的比重——资源存放的位置具有一定的随机性。
- 虽然划分规则没有定论，但根据日常开发中观察的结果，包括我们开篇给大家展示的 `Network` 截图，我们至少可以总结出这样的规律：资源存不存内存，浏览器秉承的是“节约原则”。我们发现，`Base64` 格式的图片，几乎永远可以被塞进 `memory cache`，这可以视作浏览器为节省渲染开销的“自保行为”；此外，体积不大的 `JS`、`CSS` 文件，也有较大地被写入内存的几率——相比之下，较大的 `JS`、`CSS` 文件就没有这个待遇了，内存资源是有限的，它们往往被直接甩进磁盘。

#Service Worker Cache

`Service worker` 是一种独立于主线程之外的 Javascript 线程。它脱离于浏览器窗体，因此无法直接访问 DOM。这样独立的个性使得 `service worker` 的“个人行为”无法干扰页面的性能，这个“幕后工作者”可以帮助我们实现离线缓存、消息推送和网络代理等功能。我们借助 `Service worker` 实现的离线缓存就称为 `Service worker Cache`。

- `Service worker` 的生命周期包括 `install`、`active`、`working` 三个阶段。一旦 `Service worker` 被 `install`，它将始终存在，只会在 `active` 与 `working` 之间切换，除非我们主动终止它。这是它可以用来实现离线存储的重要先决条件。
- 下面我们就通过实战的方式，一起见识一下 `service worker` 如何为我们实现离线缓存（注意看注释）：我们首先在入口文件中插入这样一段 JS 代码，用以判断和引入 `Service worker`：

```
window.navigator.serviceWorker.register('/test.js').then(
  function () {
    console.log('注册成功')
  }).catch(err => {
    console.error("注册失败")
  })
}
```

在 `test.js` 中，我们进行缓存的处理。假设我们需要缓存的文件分别是 `test.html`、`test.css` 和 `test.js`：

```
// Service worker会监听 install事件，我们在其对应的回调里可以实现初始化的逻辑
self.addEventListener('install', event => {
```

```

event.waitUntil(
  // 考虑到缓存也需要更新, open内传入的参数为缓存的版本号
  caches.open('test-v1').then(cache => {
    return cache.addAll([
      // 此处传入指定的需缓存的文件名
      '/test.html',
      '/test.css',
      '/test.js'
    ])
  })
)
);

// Service Worker会监听所有的网络请求, 网络请求的产生触发的是fetch事件, 我们可以在其对应的监听函数中实现对请求的拦截, 进而判断是否有对应到该请求的缓存, 实现从Service Worker中取到缓存的目的
self.addEventListener('fetch', event => {
  event.respondWith(
    // 尝试匹配该请求对应的缓存值
    caches.match(event.request).then(res => {
      // 如果匹配到了, 调用Server Worker缓存
      if (res) {
        return res;
      }
      // 如果没匹配到, 向服务端发起这个资源请求
      return fetch(event.request).then(response => {
        if (!response || response.status !== 200) {
          return response;
        }
        // 请求成功的话, 将请求缓存起来。
        caches.open('test-v1').then(function(cache) {
          cache.put(event.request, response);
        });
        return response.clone();
      });
    })
  );
});

```

PS: 大家注意 `Server Worker` 对协议是有要求的, 必须以 `https` 协议为前提。

#Push Cache

预告: 本小节定位为基础科普向, 对 Push Cache 有深入挖掘兴趣的同学, 强烈推荐拓展阅读 Chrome 工程师 Jake Archibald 的这篇 [HTTP/2 push is tougher than I thought](#)。

- `Push Cache` 是指 `HTTP2` 在 `server push` 阶段存在的缓存。这块的知识比较新, 应用也还处于萌芽阶段, 我找了好几个网站也没找到一个合适的案例来给大家做具体的介绍。但应用范围有限不代表不重要——`HTTP2` 是趋势、是未来。在它还未被推而广之的此时此刻, 我仍希望大家能对 `Push Cache` 的关键特性有所了解:
- `Push Cache` 是缓存的最后一道防线。浏览器只有在 `Memory Cache`、`HTTP Cache` 和 `Service worker Cache` 均未命中的情况下才会去询问 `Push Cache`。
- `Push Cache` 是一种存在于会话阶段的缓存, 当 `session` 终止时, 缓存也随之释放。
- 不同的页面只要共享了同一个 `HTTP2` 连接, 那么它们就可以共享同一个 `Push Cache`。

更多的特性和应用, 期待大家可以在日后的开发过程中去挖掘和实践。

#小结

小建议！很多人在学习缓存这块知识的时候可能多少会有这样的感觉：对浏览器缓存，只能描述个大致，却说不上深层原理；好不容易记住了每个字段怎么用，过几天又给忘了。这是因为缓存部分的知识，具有“细碎、迭代快”的特点。对于这样的知识，我们应该尝试先划分出层次和重点，归纳出完整的体系，然后针对每个知识点去各个击破。

- 终于结束了对缓存世界的探索，不知道大家有没有一种意犹未尽的感觉。开篇我们谈过，缓存非常重要，它几乎是我们性能优化的首选方案。
- 但页面的数据存储方案除了缓存，还有本地存储。在下一节中，我们就将围绕本地存储展开探索。

#五、存储篇 2：本地存储——从 Cookie 到 Web Storage、IndexDB

随着移动网络的发展与演化，我们手机上现在除了有原生 App，还能跑“WebApp”——它即开即用，用完即走。一个优秀的 WebApp 甚至可以拥有和原生 App 媲美的功能和体验。

我认为，`WebApp` 就是我们前端性能优化的产物，是我们前端工程师对体验不懈追求的结果，是 Web 网页在性能上向 `Native` 应用的一次“宣战”。

`WebApp` 优异的性能表现，要归功于浏览器存储技术的广泛应用——这其中除了我们上节提到的缓存，本地存储技术也功不可没。

#故事的开始：从 Cookie 说起

- `Cookie` 的本职工作并非本地存储，而是“维持状态”。
- 在 Web 开发的早期，人们亟需解决的一个问题就是状态管理的问题：HTTP 协议是一个无状态协议，服务器接收客户端的请求，返回一个响应，故事到此就结束了，服务器并没有记录下关于客户端的任何信息。那么下次请求的时候，如何让服务器知道“我是我”呢？
- 在这样的背景下，`Cookie` 应运而生。
- `Cookie` 说白了就是一个存储在浏览器里的一个小小的文本文件，它附着在 `HTTP` 请求上，在浏览器和服务器之间“飞来飞去”。它可以携带用户信息，当服务器检查 `Cookie` 的时候，便可以获取到客户端的状态。

关于 `Cookie` 的详细内容，我们可以在 Chrome 的 Application 面板中查看到：

The screenshot shows the Chrome DevTools Application tab with the 'Cookies' section selected. It displays a table of stored cookies for the domain https://www.google.com.hk. The columns in the table are: Name, Value, Dom..., Path, Expir..., Size, HTTP, Secure, and Sam.. The table contains several entries, including 1P_JAR, APISID, CGIC, DV, HSID, ICIBA_HUAYI_COOKIE, NID, SAPISID, SID, SIDCC, and SSID. Most cookies have a value starting with ".goo..." and a path of "/". Expiry dates range from 2018 to 2020. The 'HTTP' column shows a checkmark for most entries, while 'Secure' and 'Sam.' are mostly empty or have a checkmark in the last few rows.

如大家所见，`Cookie` 以键值对的形式存在。

#Cookie的性能劣势

1. Cookie 不够大

大家知道，Cookie 是有体积上限的，它最大只能有 4KB。当 Cookie 超过 4KB 时，它将面临被裁切的命运。这样看来，Cookie 只能用来存取少量的信息。

2. 过量的 Cookie 会带来巨大的性能浪费

Cookie 是紧跟域名的。我们通过响应头里的 set-cookie 指定要存储的 Cookie 值。默认情况下，domain 被设置为设置 Cookie 页面的主机名，我们也可以手动设置 domain 的值：

```
Set-Cookie: name=xiuyan; domain=xiuyan.me
```

同一个域名下的所有请求，都会携带 Cookie。大家试想，如果我们此刻仅仅是请求一张图片或者一个 CSS 文件，我们也要携带一个 Cookie 跑来跑去（关键是 Cookie 里存储的信息我现在并不需要），这是一件多么劳民伤财的事情。Cookie 虽然小，请求却可以有很多，随着请求的叠加，这样的不必要的 Cookie 带来的开销将是无法想象的。

随着前端应用复杂度的提高，cookie 也渐渐演化为了一个“存储多面手”——它不仅仅被用于维持状态，还被塞入了一些乱七八糟的其它信息，被迫承担起了本地存储的“重任”。在没有更好的本地存储解决方案的年代里，Cookie 小小的身体里承载了 4KB 内存所不能承受的压力。

为了弥补 cookie 的局限性，让“专业的人做专业的事情”，web Storage 出现了。

#向前一步：Web Storage

Web Storage 是 HTML5 专门为浏览器存储而提供的数据存储机制。它又分为 Local Storage 与 Session Storage。这两组概念非常相近，我们不妨先理解它们之间的区别，再对它们的共性进行研究。

1. Local Storage 与 Session Storage 的区别

两者区别的在于生命周期与作用域的不同。

- 生命周期：Local Storage 是持久化的本地存储，存储在其中的数据是永远不会过期的，使其消失的唯一办法是手动删除；而 Session Storage 是临时性的本地存储，它是会话级别的存储，当会话结束（页面被关闭）时，存储内容也随之被释放。
- 作用域：Local Storage、Session Storage 和 Cookie 都遵循同源策略。但 Session Storage 特别的一点在于，即便是相同域名下的两个页面，只要它们不在同一个浏览器窗口中打开，那么它们的 Session Storage 内容便无法共享。

1. Web Storage 的特性

- 存储容量大：Web Storage 根据浏览器的不同，存储容量可以达到 5-10M 之间。
- 仅位于浏览器端，不与服务端发生通信。

2. Web Storage 核心 API 使用示例

Web Storage 保存的数据内容和 Cookie 一样，是文本内容，以键值对的形式存在。LocalStorage 与 Session Storage 在 API 方面无异，这里我们以 localStorage 为例：

- 存储数据：setItem()

```
localStorage.setItem('user_name', 'xiuyan')
```

- 读取数据：getItem()

```
localStorage.getItem('user_name')
```

- 删除某一键名对应的数据: `removeItem()`

```
localStorage.removeItem('user_name')
```

- 清空数据记录: `clear()`

```
localStorage.clear()
```

#应用场景

1. Local Storage

`Local Storage` 在存储方面没有什么特别的限制, 理论上 `Cookie` 无法胜任的、可以用简单的键值对来存取的数据存储任务, 都可以交给 `Local Storage` 来做。

这里给大家举个例子, 考虑到 `Local Storage` 的特点之一是持久, 有时我们更倾向于用它来存储一些内容稳定的资源。比如图片内容丰富的电商网站会用它来存储 `Base64` 格式的图片字符串:

| Key | Value |
|--|--|
| //img.alicdn.com/taobao.com/.../160-1... | data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAK... |
| //img.alicdn.com/taobao.com/.../160-1... | data:image/webp;base64,UklGRhYqAABXRUJQVIA4W Ao AAA... |
| //img.alicdn.com/taobao.com/.../580-340... | data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAK... |
| //img.alicdn.com/taobao.com/.../200-200... | data:image/png;base64,UklGRq4FAABXRUJQVIA4KIFAABv... |
| //img.alicdn.com/taobao.com/.../244-44.png | data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAP... |
| //img.alicdn.com/taobao.com/.../256-25... | data:image/png;base64,UklGRtFAABXRUJQVIA4Wa AAA... |
| //img.alicdn.com/taobao.com/.../200-200... | data:image/png;base64,UklGRs4DAABXRUJQVIA4W Ao AAA... |
| 1 | data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAKAAAAACaCAMAAAC8EZfAAC91BMVEVxTirvUCzwTvrwWDbw... |

有的网站还会用它存储一些不经常更新的 CSS、JS 等静态资源。

2. Session Storage

`Session Storage` 更适合用来存储生命周期和它同步的会话级别的信息。这些信息只适用于当前会话, 当你开启新的会话时, 它也需要相应的更新或释放。比如微博的 `session storage` 就主要是存储你本次会话的浏览足迹:

| Key | Value |
|---------------|-------------------|
| FM_lastanchor | 0 |
| FM_lasturl | https://weibo.com |

- `lasturl` 对应的就是你上一次访问的 `URL` 地址, 这个地址是即时的。当你切换 `URL` 时, 它随之更新, 当你关闭页面时, 留着它也确实没有什么意义了, 干脆释放吧。这样的数据用 `Session Storage` 来处理再合适不过。

- 这样看来，`Web Storage` 确实也够强大了。那么 `Web Storage` 是否能 hold 住所有的存储场景呢？

答案是否定的。大家也看到了，`Web Storage` 是一个从定义到使用都非常简单的东西。它使用键值对的形式进行存储，这种模式有点类似于对象，却甚至连对象都不是——它只能存储字符串，要想得到对象，我们还需要先对字符串进行一轮解析。

说到底，`Web Storage` 是对 `Cookie` 的拓展，它只能用于存储少量的简单数据。当遇到大规模的、结构复杂的数据时，`Web Storage` 也爱莫能助了。这时候我们就要清楚我们的终极大 boss——`IndexDB`！

#终极形态：IndexDB

`IndexDB` 是一个运行在浏览器上的非关系型数据库。既然是数据库了，那就不是 `5M`、`10M` 这样小打小闹级别了。理论上来说，`IndexDB` 是没有存储上限的（一般来说不会小于 `250M`）。它不仅可以存储字符串，还可以存储二进制数据。

- `IndexDB` 从推出之日起，其优质教程就层出不穷，我们今天不再着重讲解它的详细操作。接下来，我们遵循 MDN 推荐的操作模式，通过一个基本的 IndexDB 使用流程，旨在对 `IndexDB` 形成一个感性的认知：

1. 打开/创建一个 `IndexDB` 数据库（当该数据库不存在时，`open` 方法会直接创建一个名为 `xiaoceDB` 新数据库）。

```
// 后面的回调中，我们可以通过event.target.result拿到数据库实例
let db
// 参数1为数据库名，参数2为版本号
const request = window.indexedDB.open("xiaoceDB", 1)
// 使用IndexDB失败时的监听函数
request.onerror = function(event) {
    console.log('无法使用IndexDB')
}
// 成功
request.onsuccess = function(event){
    // 此处就可以获取到db实例
    db = event.target.result
    console.log("你打开了IndexDB")
}
```

1. 创建一个 `object store` (`object store` 对应到数据库中的“表”单位)。

```
// onupgradeneeded事件会在初始化数据库/版本发生更新时被调用，我们在它的监听函数中创建object store
request.onupgradeneeded = function(event){
    let objectStore
    // 如果同名表未被创建过，则新建test表
    if (!db.objectStoreNames.contains('test')) {
        objectStore = db.createObjectStore('test', { keyPath: 'id' })
    }
}
```

1. 构建一个事务来执行一些数据库操作，像增加或提取数据等。

```
// 创建事务，指定表格名称和读写权限
const transaction = db.transaction(["test"], "readwrite")
// 拿到Object Store对象
const objectStore = transaction.objectStore("test")
// 向表格写入数据
objectStore.add({id: 1, name: 'xiuyan'})
```

1. 通过监听正确类型的事件以等待操作完成。

```
// 操作成功时的监听函数
transaction.oncomplete = function(event) {
  console.log("操作成功")
}

// 操作失败时的监听函数
transaction.onerror = function(event) {
  console.log("这里有一个Error")
}
```

IndexDB 的应用场景

通过上面的示例大家可以看出，在 IndexDB 中，我们可以创建多个数据库，一个数据库中创建多张表，一张表中存储多条数据——这足以 hold 住复杂的结构性数据。IndexDB 可以看做是 LocalStorage 的一个升级，当数据的复杂度和规模上升到了 LocalStorage 无法解决的程度，我们毫无疑问可以请出 IndexDB 来帮忙。

#小结

浏览器缓存/存储技术的出现和发展，为我们的前端应用带来了无限的转机。近年来基于缓存/存储技术的第三方库层出不穷，此外还衍生出了 PWA 这样优秀的 Web 应用模型。可以说，现代前端应用，尤其是移动端应用，之所以可以发展到在体验上叫板 Native 的地步，主要就是仰仗缓存/存储立下的汗马功劳。

#六、CDN 的缓存与回源机制解析

#CDN的缓存与回源机制解析

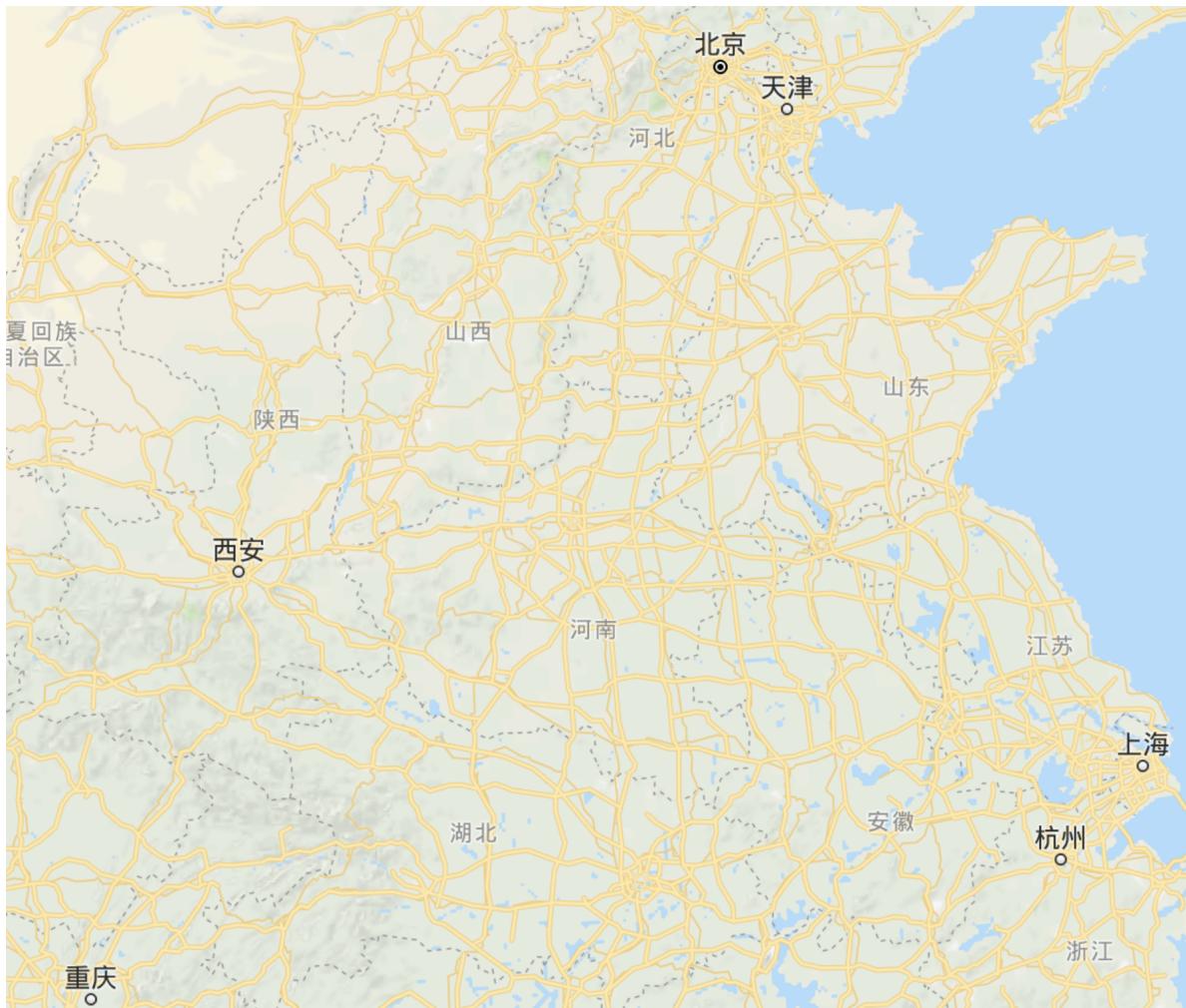
CDN (Content Delivery Network，即内容分发网络) 指的是一组分布在各个地区的服务器。这些服务器存储着数据的副本，因此服务器可以根据哪些服务器与用户距离最近，来满足数据的请求。CDN 提供快速服务，较少受高流量影响。

#为什么要用 CDN

浏览器存储的相关知识此刻离我们还不太远，大家趁热回忆一下：缓存、本地存储带来的性能提升，是不是只能在“获取到资源并把它们存起来”这件事情发生之后？也就是说，首次请求资源的时候，这些招数都是救不了我们的。要提升首次请求的响应能力，除了我们 2、3、4 节提到的方案之外，我们还需要借助 CDN 的能力。

#CDN 如何工作

借中国地图一角来给大家举一个简单的🌰：



假设我的根服务器在杭州，同时在图示的五个城市里都有自己可用的机房。

此时有一位北京的用户向我请求资源。在网络带宽小、用户访问量大的情况下，杭州的这一台服务器或许不那么给力，不能给用户非常快的响应速度。于是我灵机一动，把这批资源 copy 了一批放在北京的机房里。当用户请求资源时，就近请求北京的服务器，北京这台服务器低头一看，这个资源我存了，离得这么近，响应速度肯定噌噌的！那如果北京这台服务器没有 copy 这批资源呢？它会再向杭州的根服务器去要这个资源。在这个过程中，北京这台服务器就扮演着 CDN 的角色。

#CDN的核心功能特写

CDN 的核心点有两个，一个是缓存，一个是回源。

这两个概念都非常好理解。对标到上面描述的过程，“缓存”就是说我们把资源 copy 一份到 CDN 服务器上这个过程，“回源”就是说 CDN 发现自己没有这个资源（一般是缓存的数据过期了），转头向根服务器（或者它的上层服务器）去要这个资源的过程。

#CDN 与前端性能优化

- CDN 往往是被前端认为前端不需要了解的东西。
- 具体来说，我身边许多同学对其的了解止步于：部署界面上有一个“部署到CDN”按钮，我去点一下，资源就在 CDN 上啦！

“眼下业务开发用不到的可以暂缓了解”，这是没毛病的。但正如我小册开篇所说的，前端工程师首先是软件工程师。对整个技术架构的理解，将会反哺我们对某一具体环节的理解；知识点的适当拓展，也会对大家技术高度和技术广度的提升大有裨益。

那么，我们了解一下 CDN 是怎么帮助前端的。

CDN 往往被用来存放静态资源。上文中我们举例所提到的“根服务器”本质上是业务服务器，它的核心任务在于**生成动态页面或返回非纯静态页面**，这两种过程都是需要计算的。业务服务器仿佛一个车间，车间里运转的机器轰鸣着为我们产出所需的资源；相比之下，CDN 服务器则像一个仓库，它只充当资源的“栖息地”和“搬运工”。

所谓“静态资源”，就是像 JS、CSS、图片等**不需要业务服务器进行计算即得的资源**。而“动态资源”，顾名思义是需要**后端实时动态生成的资源**，较为常见的就是 JSP、ASP 或者依赖服务端渲染得到的 HTML 页面。

什么是“非纯静态资源”呢？它是指**需要服务器在页面之外作额外计算的 HTML 页面**。具体来说，当我打开某一网站之前，该网站需要通过权限认证等一系列手段确认我的身份、进而决定是否要把 HTML 页面呈现给我。这种情况下 HTML 确实是静态的，但它**和业务服务器的操作耦合**，我们把它丢到CDN 上显然是不合适的。

#CDN 的实际应用

静态资源本身具有访问频率高、承接流量大的特点，因此静态资源加载速度始终是前端性能的一个非常关键的指标。CDN 是静态资源提速的重要手段，在许多一线的互联网公司，“静态资源走 CDN”并不是一个建议，而是一个规定。

比如以淘宝为代表的阿里系产品，就遵循着这个“规定”。打开淘宝首页，我们可以在 Network 面板中看到，“非纯静态”的 HTML 页面，是向业务服务器请求来的：

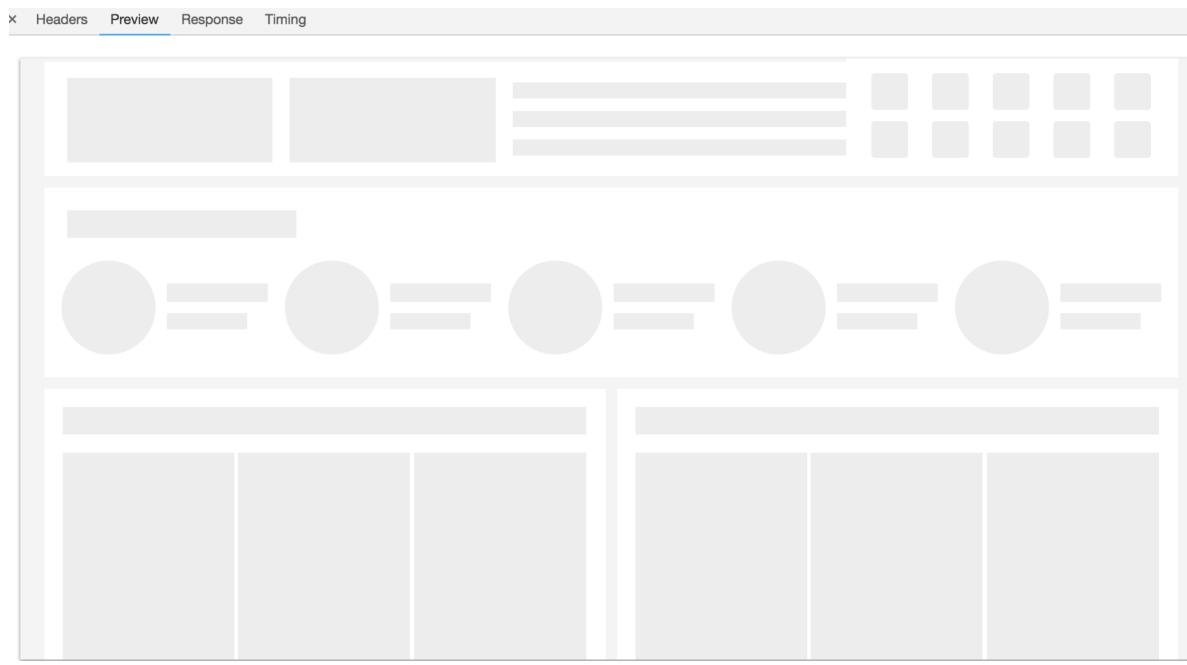
The screenshot shows the Network tab of a browser's developer tools. The table lists various resources loaded by the page. The Headers tab is currently active, displaying detailed information about the request to `https://www.taobao.com/`. The response headers include:

- Request URL: `https://www.taobao.com/`
- Request Method: GET
- Status Code: 200 OK (from ServiceWorker)
- Remote Address: 42.120.239.125:443
- Referrer Policy: no-referrer-when-downgrade

Other visible headers in the Response Headers section include:

- Age: 71
- Cache-Control: max-age=60, s-maxage=90
- Content-Encoding: gzip
- Content-MD5: 3VczKkaKV6fyPQ8Zng1njw==
- Content-Type: text/html; charset=utf-8
- Date: Tue, 25 Sep 2018 04:45:10 GMT
- EagleId: 2a78ef4615378507104604158e

我们点击 preview，可以看到业务服务器确实是返回给了我们一个尚未被静态资源加持过的简单 HTML 页面，所有的图片内容都是先以一个 div 占位：



相应地，我们随便点开一个静态资源，可以看到它都是从 CDN 服务器上请求来的。

比如说图片：

Request URL: <https://g.alicdn.com/s.gif>
Request Method: GET

再比如 JS、CSS 文件：

Request URL: https://g.alicdn.com/mm/tb-page-peel/0.0.5/index-min.js
Request Method: GET

Request URL: https://g.alicdn.com/kg/??search-suggest/6.3.1/new_suggest-min.css
Request Method: GET

#CDN 优化细节

如何让 CDN 的效用最大化？这又是需要前后端程序员一起思考的庞大命题。它涉及到 CDN 服务器本身的性能优化、CDN 节点的地址选取等。但我们今天不写高深的论文，只谈离前端最近的这部分细节：CDN 的域名选取。

大家先回头看一下我刚刚选取的淘宝首页的例子，我们注意到业务服务器的域名是这个：

www.taobao.com

而 CDN 服务器的域名是这个：

g.alicdn.com

没错，我们不一样！

再看另一方面，我们讲到 `Cookie` 的时候，为了凸显 `Local Storage` 的优越性，曾经提到过：

`Cookie` 是紧跟域名的。同一个域名下的所有请求，都会携带 `Cookie`。大家试想，如果我们此刻仅仅是请求一张图片或者一个 `CSS` 文件，我们也要携带一个 `Cookie` 跑来跑去（关键是 `Cookie` 里存储的信息我现在并不需要），这是一件多么劳民伤财的事情。`Cookie` 虽然小，请求却可以有很多，随着请求的叠加，这样的不必要的 `Cookie` 带来的开销将是无法想象的……

- 同一个域名下的请求会不分青红皂白地携带 cookie，而静态资源往往并不需要 cookie 携带什么认证信息。把静态资源和主页面置于不同的域名下，完美地避免了不必要的 cookie 的出现！
- 看起来是一个不起眼的小细节，但带来的效用却是惊人的。以电商网站静态资源的流量之庞大，如果没把这个多余的 cookie 拿下来，不仅用户体验会大打折扣，每年因性能浪费带来的经济开销也将是一个非常恐怖的数字。

#七、渲染篇 1：服务端渲染的探索与实践

服务端渲染（SSR）近两年炒得很火热，相信各位同学对这个名词多少有所耳闻。本节我们将围绕“是什么”（服务端渲染的运行机制），“为什么”（服务端渲染解决了什么性能问题），“怎么做”（服务端渲染的应用实例与使用场景）这三个点，对服务端渲染进行探索。

- 服务端渲染是一个相对的概念，它的对立面是“客户端渲染”。在运行机制解析这部分，我们会借力客户端渲染的概念，来帮大家理解服务端渲染的工作方式。基于对工作方式的了解，再去深挖它的原理与优势。
- 任何知识点都不是“一座孤岛”，服务端渲染的实践往往与当下流行的前端技术（譬如 Vue，React，Redux 等）紧密结合。本节下半场将以 React 和 Vue 下的服务端渲染实现为例，为大家呈现一个完整的 SSR 实现过程。

#服务端渲染的运行机制

相对于服务端渲染，同学们普遍对客户端渲染接受度更高一些，所以我们先从大家喜闻乐见的客户端渲染说起。

1. 客户端渲染

客户端渲染模式下，服务端会把渲染需要的静态文件发送给客户端，客户端加载过来之后，自己在浏览器里跑一遍 JS，根据 JS 的运行结果，生成相应的 DOM。这种特性使得客户端渲染的源代码总是特别简洁，往往是这个德行：

```
<!doctype html>
<html>
  <head>
    <title>我是客户端渲染的页面</title>
  </head>
  <body>
    <div id='root'></div>
    <script src='index.js'></script>
  </body>
</html>
```

根节点下到底是什么内容呢？你不知道，我不知道，只有浏览器把 index.js 跑过一遍后才知道，这就是典型的客户端渲染。

页面上呈现的内容，你在 html 源文件里里找不到——这正是它的特点。

2. 服务端渲染

服务端渲染的模式下，当用户第一次请求页面时，由服务器把需要的组件或页面渲染成 HTML 字符串，然后把它返回给客户端。客户端拿到手的，是可以直接渲染然后呈现给用户的 HTML 内容，不需要为了生成 DOM 内容自己再去跑一遍 JS 代码。

使用服务端渲染的网站，可以说是“所见即所得”，页面上呈现的内容，我们在 html 源文件里也能找到。

比如知乎就是典型的服务端渲染案例：

热门内容, 来自: 美食

打不死的小强, 但行好事, 莫问前程。

有哪些会动的食物让你感到好奇?

蛇 是一种有人爱有人怕的食
刚下锅时, 它是这样的 ↓ ↓
↓ ↓ ↓ 好了, 预警结束!
出海 青龙闹海 龙腾四海

▲ 赞同 4.6K ▼ 127 条评论 分享 收藏

| Name | Headers | Preview | Response | Cookies | Timing |
|---------------------------------|---------|---------|---|---------|--------|
| www.zhihu.com | | | 1 2 quot;蛇 是一种有人爱有人怕的食物, 做成菜以后成品如图↓活杀以后, 刚下锅时, 它是这样的↓↓↓好了, 预警结束!出海 青龙闹海 龙腾四海 | | |
| home_bottom.html | | | | | |
| home_top.html | | | | | |
| zrt_lookup.html | | | | | |
| ads?client=ca-pub-9258318038... | | | | | |
| ads?client=ca-pub-9258318038... | | | | | |
| cookie_push.html | | | | | |

zhihu.com 返回的 HTML 文件已经是可以直接进行渲染的内容了。

#服务端渲染解决了什么性能问题

- 事实上, 很多网站是出于效益的考虑才启用服务端渲染, 性能倒是在其次。
- 假设 A 网站页面中有一个关键字叫“前端性能优化”, 这个关键字是 JS 代码跑过一遍后添加到 HTML 页面中的。那么客户端渲染模式下, 我们在搜索引擎搜索这个关键字, 是找不到 A 网站的——搜索引擎只会查找现成的内容, 不会帮你跑 JS 代码。A 网站的运营方见此情形, 感到很头大: 搜索引擎搜不出来, 用户找不到我们, 谁还会用我的网站呢? 为了把“现成的内容”拿给搜索引擎看, A 网站不得不启用服务端渲染。
- 但性能在其次, 不代表性能不重要。服务端渲染解决了一个非常关键的性能问题——首屏加载速度过慢。在客户端渲染模式下, 我们除了加载 HTML, 还要等渲染所需的这部分 JS 加载完, 之后还得把这部分 JS 在浏览器上再跑一遍。这一切都是发生在用户点击了我们的链接之后的事情, 在这个过程结束之前, 用户始终见不到我们网页的庐山真面目, 也就是说用户一直在等! 相比之下, 服务端渲染模式下, 服务器给到客户端的已经是一个直接可以拿来呈现给用户的网页, 中间环节早在服务端就帮我们做掉了, 用户岂不“美滋滋”?

#服务端渲染的应用实例

下面我们先来看一下在一个 React 项目里, 服务端渲染是怎么实现的。本例中, 我们使用 Express 搭建后端服务。

项目中有一个叫做 `vdom` 的 `React` 组件, 它的内容如下。

`VDom.js:`

```
import React from 'react'

const VDom = () => {
  return <div>我是一个被渲染为真实DOM的虚拟DOM</div>
}

export default VDom
```

在服务端的入口文件中, 我引入这个组件, 对它进行渲染:

```
import express from 'express'
import React from 'react'
import { renderToString } from 'react-dom/server'
import VDom from './VDom'

// 创建一个express应用
const app = express()
// renderToString 是把虚拟DOM转化为真实DOM的关键方法
const RDom = renderToString(<VDom />)
// 编写HTML模板, 插入转化后的真实DOM内容
const Page = `
```

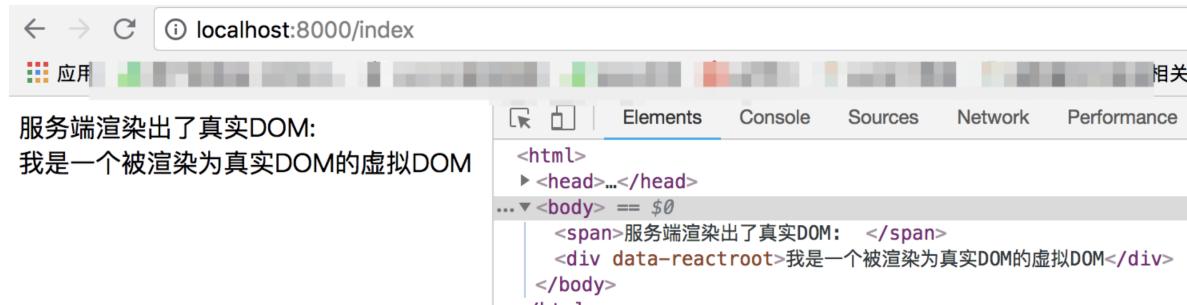
```

<head>
  <title>test</title>
</head>
<body>
  <span>服务端渲染出了真实DOM: </span>
  ${RDom}
</body>
</html>
`
```

// 配置HTML内容对应的路由
app.get('/index', function(req, res) {
 res.send(Page)
})

// 配置端口号
const server = app.listen(8000)

根据我们的路由配置，当我访问 <http://localhost:8000/index> 时，就可以呈现出服务端渲染的结果了：



我们可以看到，`vDom` 组件已经被 `renderToString` 转化为了一个内容为 `<div data-reactroot="">我是一个被渲染为真实DOM的虚拟DOM</div>` 的字符串，这个字符串被插入 HTML 代码，成为了真实 DOM 树的一部分。

那么 Vue 是如何实现服务端渲染的呢？

其实是一个套路，这里基于 [Vue SSR 指南](#) 中官方给出的例子为大家讲解 Vue 中的实现思路（思路见注释）。

该示例直接将 Vue 实例整合进了服务端的入口文件中：

```

const Vue = require('vue')
// 创建一个express应用
const server = require('express')()
// 提取出renderer实例
const renderer = require('vue-server-renderer').createRenderer()

server.get('*', (req, res) => {
  // 编写Vue实例（虚拟DOM节点）
  const app = new Vue({
    data: {
      url: req.url
    },
    // 编写模板HTML的内容
    template: `<div>访问的 URL 是: {{ url }}</div>`
  })

  // renderToString 是把Vue实例转化为真实DOM的关键方法
  renderer.renderToString(app, (err, html) => {
```

```

if (err) {
  res.status(500).end('Internal Server Error')
  return
}
// 把渲染出来的真实DOM字符串插入HTML模板中
res.end(`
  <!DOCTYPE html>
  <html lang="en">
    <head><title>Hello</title></head>
    <body>${html}</body>
  </html>
`)
})
}

server.listen(8080)

```

- 大家对比一下 React 项目中的注释内容，是不是发现这两段代码从本质上来说区别不大呢？

以上两个小 ，为大家演示了基本的服务端渲染实现流程。

实际项目比这些复杂很多，但万变不离其宗。强调的只有两点：一是这个 `renderToString()` 方法；二是把转化结果“塞”进模板里的这一步。这两个操作是服务端渲染的灵魂操作。在虚拟 DOM“横行”的当下，服务端渲染不再是早年 JSP 里简单粗暴的字符串拼接过程，它还要求这一端要具备将虚拟 `DOM` 转化为真实 `DOM` 的能力。与其说是“把 JS 在服务器上先跑一遍”，不如说是“把 `Vue`、`React` 等框架代码先在 `Node` 上跑一遍”。

#服务端渲染的应用场景

- 打眼一看，这个服务端渲染给浏览器省了这么多事儿，性能肯定是质的飞跃啊！喜闻乐见！但是大家打开自己经常访问的那些网页看一看，会发现仍然有许多网站压根儿不用服务端渲染——看来这个东西也不是万能的。
- 根据我们前面的描述，不难看出，服务端渲染本质上是**本该浏览器做的事情，分担给服务器去做**。这样当资源抵达浏览器时，它呈现的速度就快了。乍一看好像很合理：浏览器性能毕竟有限，服务器多牛逼！能者多劳，就该让服务器多干点活！
- 但仔细想想，在这个网民遍地的时代，几乎有多少个用户就有多少台浏览器。用户拥有的浏览器总量多到数不清，那么一个公司的服务器又有多少台呢？我们把这么多台浏览器的渲染压力集中起来，分散给相比之下数量并不多的服务器，服务器肯定是承受不住的。
- 这样分析下来，服务端渲染也并非万全之策。在实践中，我一般会建议大家先忘记服务端渲染这个事情——服务器稀少而宝贵，但首屏渲染体验和 SEO 的优化方案却很多——我们最好先把能用的低成本“大招”都用完。除非网页对性能要求太高了，以至于所有的招式都用完了，性能表现还是不尽人意，这时候我们就可以考虑向老板多申请几台服务器，把服务端渲染搞起来了~

#八、渲染篇 2：知己知彼——解锁浏览器背后的运行机制

平时我们几乎每天都在和浏览器打交道，在一些兼容任务比较繁重的团队里，苦逼的前端攻城师们甚至为了兼容各个浏览器而不断地去测试和调试，还要在脑子中记下各种遇到的 BUG 及解决方案。即便如此，我们好像并没有去主动地关注和了解下浏览器的工作原理。我想如果我们对此做一点了解，在项目过程中就可以有效地避免一些问题，并对页面性能做出相应的改进。

“知己知彼，百战不殆”，今天，我们就一起来揭开浏览器渲染过程的神秘面纱！

#浏览器的“心”

- 浏览器的“心”，说的就是浏览器的内核。在研究浏览器微观的运行机制之前，我们首先要对浏览器内核有一个宏观的把握。
- 开篇提到许多工程师因为业务需要，免不了需要去处理不同浏览器下代码渲染结果的差异性。这些差异性正是因为浏览器内核的不同而导致的——浏览器内核决定了浏览器解释网页语法的方式。浏览器内核可以分成两部分：渲染引擎（Layout Engine 或者 Rendering Engine）和 JS 引擎。早期渲染引擎和 JS 引擎并没有十分明确的区分，但随着 JS 引擎越来越独立，内核也成了渲染引擎的代称（**下文我们将沿用这种叫法**）。渲染引擎又包括了 HTML 解释器、CSS 解释器、布局、网络、存储、图形、音视频、图片解码器等等零部件。



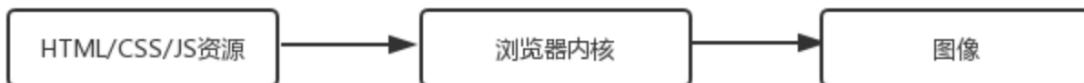
目前市面上常见的浏览器内核可以分为这四种：Trident（IE）、Gecko（火狐）、Blink（Chrome、Opera）、Webkit（Safari）。

这里面大家最耳熟能详的可能就是 Webkit 内核了。很多同学可能会听说过 Chrome 的内核就是 Webkit，殊不知 Chrome 内核早已迭代为了 Blink。但是换汤不换药，Blink 其实也是基于 Webkit 衍生而来的一个分支，因此，Webkit 内核仍然是当下浏览器世界真正的霸主。

下面我们就以 Webkit 为例，对现代浏览器的渲染过程进行一个深度的剖析。

#开启浏览器渲染“黑盒”

什么是渲染过程？简单来说，渲染引擎根据 HTML 文件描述构建相应的数学模型，调用浏览器各个零部件，从而将网页资源代码转换为图像结果，这个过程就是渲染过程（如下图）。



从这个流程来看，浏览器呈现网页这个过程，宛如一个黑盒。在这个神秘的黑盒中，有许多功能模块，内核内部的实现正是这些功能模块相互配合协同工作进行的。其中我们最需要关注的，就是 **HTML 解释器、CSS 解释器、图层布局计算模块、视图绘制模块与 JavaScript 引擎** 这几大模块：

- HTML 解释器：将 HTML 文档经过词法分析输出 DOM 树。
- CSS 解释器：解析 CSS 文档，生成样式规则。
- 图层布局计算模块：布局计算每个对象的精确位置和大小。
- 视图绘制模块：进行具体节点的图像绘制，将像素渲染到屏幕上。
- JavaScript 引擎：编译执行 Javascript 代码。

#浏览器渲染过程解析

有了对零部件的了解打底，我们就可以一起来走一遍浏览器的渲染流程了。在浏览器里，每一个页面的首次渲染都经历了如下阶段（图中箭头不代表串行，有一些操作是并行进行的，下文会说明）：



- **解析 HTML**

在这一步浏览器执行了所有的加载解析逻辑，在解析 HTML 的过程中发出了页面渲染所需的各种外部资源请求。

- **计算样式**

浏览器将识别并加载所有的 CSS 样式信息与 DOM 树合并，最终生成页面 render 树 (:after :before 这样的伪元素会在这个环节被构建到 DOM 树中)。

- **计算图层布局**

页面中所有元素的相对位置信息，大小等信息均在这一步得到计算。

- **绘制图层**

在这一步中浏览器会根据我们的 DOM 代码结果，把每一个页面图层转换为像素，并对所有的媒体文件进行解码。

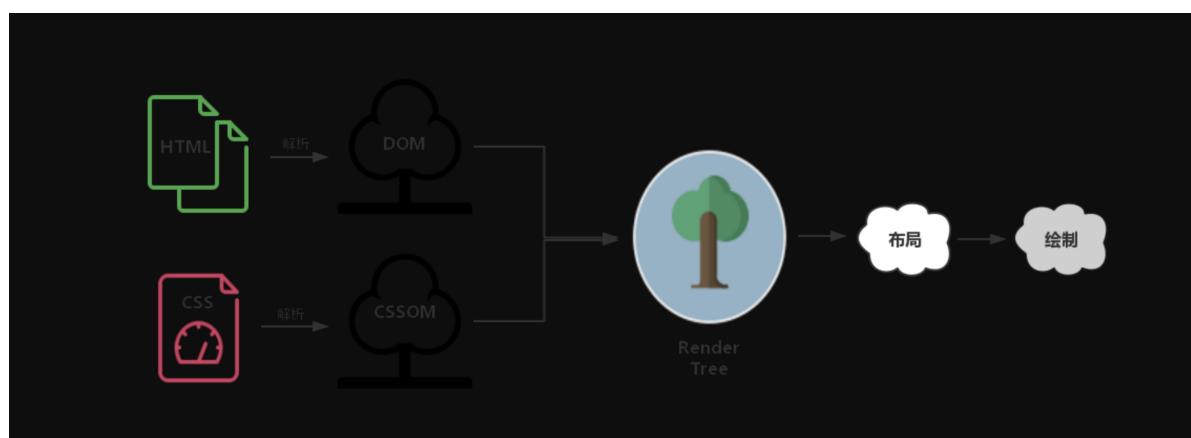
- **整合图层，得到页面**

最后一步浏览器会合并各个图层，将数据由 CPU 输出给 GPU 最终绘制在屏幕上。（复杂的视图层会给这个阶段的 GPU 计算带来一些压力，在实际应用中为了优化动画性能，我们有时会手动区分不同的图层）。

#几棵重要的“树”

上面的内容没有理解透彻？别着急，我们一起来捋一捋这个过程中的重点——树！

为了使渲染过程更明晰一些，我们需要给这些“树”们一个特写：



- **DOM 树**: 解析 HTML 以创建的是 DOM 树 (DOM tree)：渲染引擎开始解析 HTML 文档，转换树中的标签到 DOM 节点，它被称为“内容树”。
- **CSSOM 树**: 解析 CSS (包括外部 CSS 文件和样式元素) 创建的是 CSSOM 树。CSSOM 的解析过程与 DOM 的解析过程是**并行的**。
- **渲染树**: CSSOM 与 DOM 结合，之后我们得到的就是渲染树 (Render tree)。
- **布局渲染树**: 从根节点递归调用，计算每一个元素的大小、位置等，给每个节点所应该出现在屏幕上的精确坐标，我们便得到了基于渲染树的布局渲染树 (Layout of the render tree)。
- **绘制渲染树**: 遍历渲染树，每个节点将使用 UI 后端层来绘制。整个过程叫做绘制渲染树 (Painting the render tree)。

基于这些“树”，我们再梳理一番：

渲染过程说白了，首先是基于 HTML 构建一个 DOM 树，这棵 DOM 树与 CSS 解释器解析出的 CSSOM 相结合，就有了布局渲染树。最后浏览器以布局渲染树为蓝本，去计算布局并绘制图像，我们页面的初次渲染就大功告成了。

- 之后每当一个新元素加入到这个 DOM 树当中，浏览器便会通过 CSS 引擎查遍 CSS 样式表，找到符合该元素的样式规则应用到这个元素上，然后再重新去绘制它。
- 有心的同学可能已经在思考了，查表是个耗时间的活，我怎么让浏览器的查询工作又快又好地实现呢？OK，讲了这么多原理，我们终于引出了我们的第一个可转化为代码的优化点——CSS 样式表规则的优化！

#不做无用功：基于渲染流程的 CSS 优化建议

在给出 CSS 选择器方面的优化建议之前，先告诉大家一个小知识：CSS 引擎查找样式表，对每条规则都按从右到左的顺序去匹配。看如下规则：

```
#myList li {}
```

这样的写法其实很常见。大家平时习惯了从左到右阅读的文字阅读方式，会本能地以为浏览器也是从左到右匹配 CSS 选择器的，因此会推测这个选择器并不会费多少力气：#myList 是一个 id 选择器，它对应的元素只有一个，查找起来应该很快。定位到了 myList 元素，等于是缩小了范围再去查找它后代中的 li 元素，没毛病。

事实上，**CSS 选择符是从右到左进行匹配的**。我们这个看似“没毛病”的选择器，实际开销相当高：浏览器必须遍历页面上每个 li 元素，并且每次都要去确认这个 li 元素的父元素 id 是不是 myList，你说坑不坑！

说到坑，不知道大家还记得经典的通配符：

```
* {}
```

- 入门 CSS 的时候，不少同学拿通配符清除默认样式（我曾经也是通配符用户的一员）。但这个家伙很恐怖，它会匹配所有元素，所以浏览器必须去遍历每一个元素！大家低头看看自己页面里的元素个数，是不是心凉了——这得计算多少次呀！
- 这样一看，一个小小的 CSS 选择器，也有不少的门道！好的 CSS 选择器书写习惯，可以为我们带来非常可观的性能提升。根据上面的分析，我们至少可以总结出如下性能提升的方案：
- 避免使用通配符，只对需要用到的元素进行选择。
- 关注可以通过继承实现的属性，避免重复匹配重复定义。
- 少用标签选择器。如果可以，用类选择器替代，举个🌰：

错误示范：

```
#myList li{}
```

课代表：

```
.myList_li {}
```

- 不要画蛇添足，id 和 class 选择器不应该被多余的标签选择器拖后腿。举个🌰：

错误示范

```
.myList#title
```

课代表

```
#title
```

减少嵌套。后代选择器的开销是最高的，因此我们应该尽量将选择器的深度降到最低（最高不要超过三层），尽可能使用类来关联每一个标签元素。

搞定了 CSS 选择器，万里长征才刚刚开始的第一步。但现在你已经理解了浏览器的工作过程，接下来的征程对你来说并不再是什么难题~

#告别阻塞：CSS 与 JS 的加载顺序优化

说完了过程，我们来说一说特性。

HTML、CSS 和 JS，都具有**阻塞渲染**的特性。

HTML 阻塞，天经地义——没有 HTML，何来 DOM？没有 DOM，渲染和优化，都是空谈。

那么 CSS 和 JS 的阻塞又是怎么回事呢？

1. CSS 的阻塞

- 在刚刚的过程中，我们提到 DOM 和 CSSOM 合力才能构建渲染树。这一点会给性能造成严重影响：默认情况下，CSS 是阻塞的资源。浏览器在构建 CSSOM 的过程中，**不会渲染任何已处理的内容**。即便 DOM 已经解析完毕了，只要 CSSOM 不 OK，那么渲染这个事情就不 OK（这主要是为了避免没有 CSS 的 HTML 页面丑陋地“裸奔”在用户眼前）。
- 我们知道，只有当我们开始解析 HTML 后、解析到 link 标签或者 style 标签时，CSS 才登场，CSSOM 的构建才开始。很多时候，DOM 不得不等待 CSSOM。因此我们可以这样总结：

CSS 是阻塞渲染的资源。需要将它尽早、尽快地下载到客户端，以便缩短首次渲染的时间。

事实上，现在很多团队都已经做到了尽早（将 CSS 放在 head 标签里）和尽快（启用 CDN 实现静态资源加载速度的优化）。这个“把 CSS 往前放”的动作，对很多同学来说已经内化为一种编码习惯。那么现在我们还应该知道，这个“习惯”不是空穴来风，它是由 CSS 的特性决定的。

2. JS 的阻塞

- 不知道大家注意到没有，前面我们说过程的时候，花了很多笔墨去说 HTML、说 CSS。相比之下，JS 的出镜率也太低了点。这当然不是因为 JS 不重要。而是因为，在首次渲染过程中，JS 并不是一个非登场不可的角色——没有 JS，CSSOM 和 DOM 照样可以组成渲染树，页面依然会呈现——即使它死气沉沉、毫无交互。
- JS 的作用在于**修改**，它帮助我们修改网页的方方面面：内容、样式以及它如何响应用户交互。这“方方面面”的修改，本质上都是对 DOM 和 CSSDOM 进行修改。因此 JS 的执行会阻止 CSSOM，在我们不作显式声明的情况下，它也会阻塞 DOM。

我们通过一个⌚来理解一下这个机制：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>JS阻塞测试</title>
  <style>
```

```

#container {
    background-color: yellow;
    width: 100px;
    height: 100px;
}
</style>
<script>
    // 尝试获取container元素
    var container = document.getElementById("container")
    console.log('container', container)
</script>
</head>
<body>
    <div id="container"></div>
    <script>
        // 尝试获取container元素
        var container = document.getElementById("container")
        console.log('container', container)
        // 输出container元素此刻的背景色
        console.log('container bgColor',
getComputedStyle(container).backgroundColor)
    </script>
    <style>
        #container {
            background-color: blue;
        }
    </style>
</body>
</html>

```

三个 console 的结果分别为：

```

"container" null
"container" "<div id='container'></div>"
"container bgColor" "rgb(255, 255, 0)"

```

- 注：本例仅使用了内联 JS 做测试。感兴趣的同学可以把这部分 JS 当做外部文件引入看看效果——它们的表现一致。
- 第一次尝试获取 id 为 container 的 DOM 失败，这说明 JS 执行时阻塞了 DOM，后续的 DOM 无法构建；第二次才成功，这说明脚本块只能找到在它前面构建好的元素。这两者结合起来，“阻塞 DOM”得到了验证。再看第三个 console，尝试获取 CSS 样式，获取到的是在 JS 代码执行前的背景色 (yellow)，而非后续设定的新样式 (blue)，说明 CSSOM 也被阻塞了。那么在阻塞的背后，到底发生了什么呢？
- 我们前面说过，**JS 引擎是独立于渲染引擎存在的**。我们的 JS 代码在文档的何处插入，就在何处执行。当 HTML 解析器遇到一个 script 标签时，它会暂停渲染过程，将控制权交给 JS 引擎。JS 引擎对内联的 JS 代码会直接执行，对外部 JS 文件还要先获取到脚本、再进行执行。等 JS 引擎运行完毕，浏览器又会把控制权还给渲染引擎，继续 CSSOM 和 DOM 的构建。因此与其说是 JS 把 CSS 和 HTML 阻塞了，不如说是 JS 引擎抢走了渲染引擎的控制权。

- 现在理解了阻塞的表现与原理，我们开始思考一个问题。浏览器之所以让 JS 阻塞其它的活动，是因为它不知道 JS 会做什么改变，担心如果不阻止后续的操作，会造成混乱。但是我们是写 JS 的人，我们知道 JS 会做什么改变。假如我们可以确认一个 JS 文件的执行时机并不一定非要是此时此刻，我们就可以通过对它使用 `defer` 和 `async` 来避免不必要的阻塞，这里我们就引出了外部 JS 的三种加载方式。

3. JS的三种加载方式

- 正常模式：

```
<script src="index.js"></script>
```

这种情况下 JS 会阻塞浏览器，浏览器必须等待 `index.js` 加载和执行完毕才能去做其它事情。

- `async` 模式：

```
<script async src="index.js"></script>
```

`async` 模式下，JS 不会阻塞浏览器做任何其它的事情。它的加载是异步的，当它加载结束，JS 脚本会立即执行。

- `defer` 模式：

```
<script defer src="index.js"></script>
```

`defer` 模式下，JS 的加载是异步的，执行是被推迟的。等整个文档解析完成、`DOMContentLoaded` 事件即将被触发时，被标记了 `defer` 的 JS 文件才会开始依次执行。

- 从应用的角度来说，一般当我们的脚本与 DOM 元素和其它脚本之间的依赖关系不强时，我们会选用 `async`；当脚本依赖于 DOM 元素和其它脚本的执行结果时，我们会选用 `defer`。
- 通过审时度势地向 `script` 标签添加 `async/defer`，我们就可以告诉浏览器在等待脚本可用期间不阻止其它的工作，这样可以显著提升性能。

#小结

- 我们知道，当 `JS` 登场时，往往意味着对 `DOM` 的操作。DOM 操作所导致的性能开销的“昂贵”，大家可能早就有所耳闻，雅虎军规里很重要的一条就是“尽量减少 DOM 访问”。
- 那么 DOM 到底为什么慢，我们如何去规避这种慢呢？这里我们就引出了下一个章节需要重点解释的两个概念：CSS 中的回流（Reflow）与重绘（Repaint）。

九、渲染篇 3：对症下药——DOM 优化原理与基本实践

从本节开始，我们要关心的两大核心问题就是：“DOM 为什么这么慢”以及“如何使 DOM 变快”。

后者是一个比“生存还是毁灭”更加经典的问题。不仅我们为它“肝肠寸断”，许多优秀前端框架的作者大大们也曾为其绞尽脑汁。这一点可喜可贺——研究的人越多，产出优秀实践的概率就越大。因此在本章的方法论环节，我们不仅会根据 DOM 特性及渲染原理为大家讲解基本的优化思路，还会涉及到一部分生产实践。

循着这个思路，我们把 DOM 优化这块划分为三个小专题：“DOM 优化思路”、“异步更新策略”及“回流与重绘”。本节对应第一个小专题。三个小专题休戚与共、你侬我侬，在思路上相互依赖、一脉相承，因此此处严格禁止任何姿势的跳读行为。

考虑到本节内容与上一节有着密不可分的关系，因此强烈不建议没有读完上一节的同学直接跳读本节。

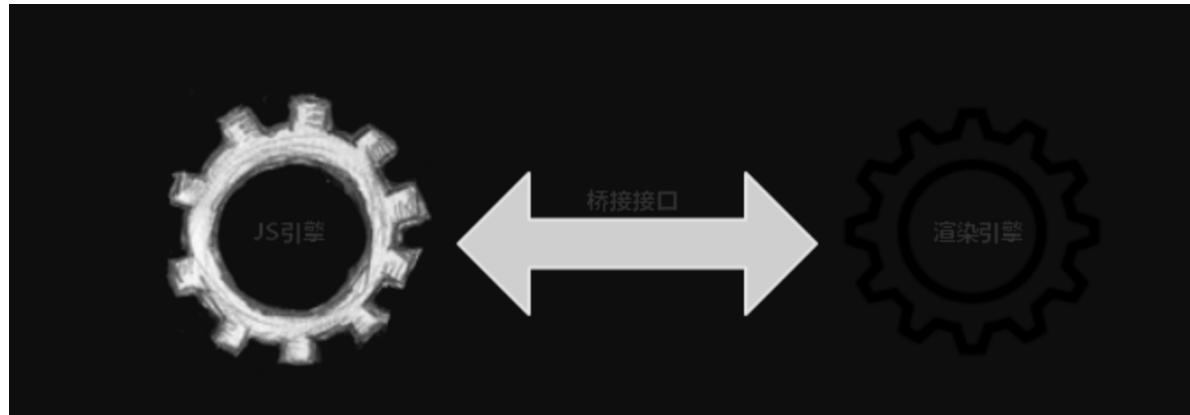
#望闻问切：DOM 为什么这么慢

1. 因为收了“过路费”

把 DOM 和 JavaScript 各自想象成一个岛屿，它们之间用收费桥梁连接。——《高性能 JavaScript》

JS 是很快的，在 JS 中修改 DOM 对象也是很快的。在 JS 的世界里，一切是简单的、迅速的。但 DOM 操作并非 JS 一个人的独舞，而是两个模块之间的协作。

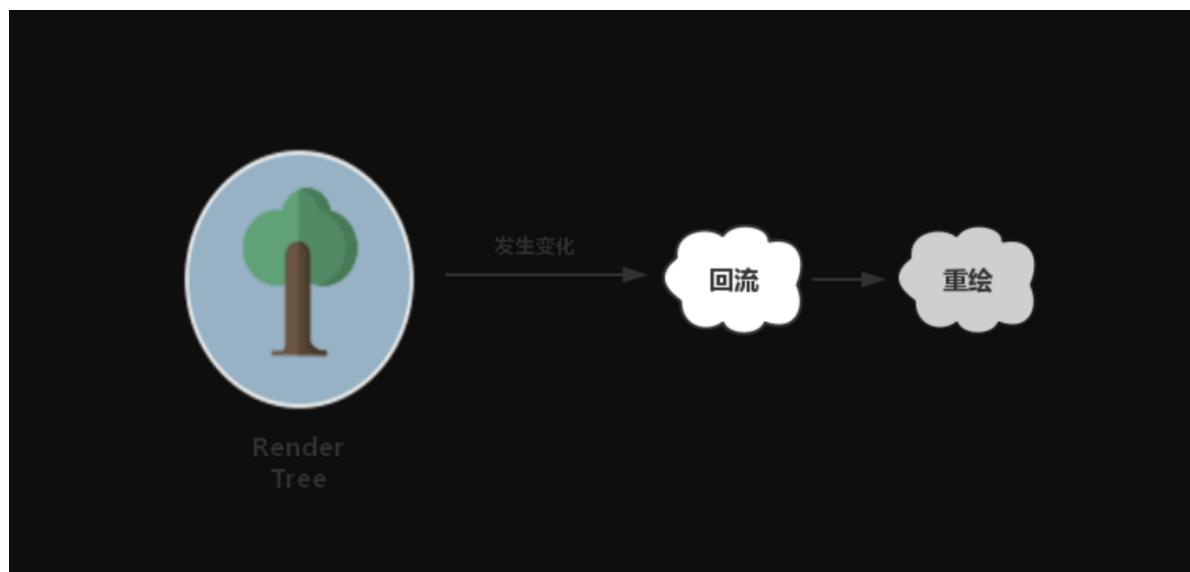
上一节我们提到，JS 引擎和渲染引擎（浏览器内核）是独立实现的。当我们用 JS 去操作 DOM 时，本质上是 JS 引擎和渲染引擎之间进行了“跨界交流”。这个“跨界交流”的实现并不简单，它依赖了桥接接口作为“桥梁”（如下图）。



过“桥”要收费——这个开销本身就是不可忽略的。我们每操作一次 DOM（不管是修改还是仅仅为了访问其值），都要过一次“桥”。过“桥”的次数一多，就会产生比较明显的性能问题。因此“减少 DOM 操作”的建议，并非空穴来风。

2. 对 DOM 的修改引发样式的更迭

- 过桥很慢，到了桥对岸，我们的更改操作带来的结果也很慢。
- 很多时候，我们对 DOM 的操作都不会局限于访问，而是为了修改它。当我们对 DOM 的修改会引发它外观（样式）上的改变时，就会触发回流或重绘。
- 这个过程本质上还是因为我们对 DOM 的修改触发了渲染树（Render Tree）的变化所导致的：



- 回流：当我们对 DOM 的修改引发了 DOM 几何尺寸的变化（比如修改元素的宽、高或隐藏元素等）时，浏览器需要重新计算元素的几何属性（其他元素的几何属性和位置也会因此受到影响），然后再将计算的结果绘制出来。这个过程就是回流（也叫重排）。
- 重绘：当我们对 DOM 的修改导致了样式的变化、却并未影响其几何属性（比如修改了颜色或背景色）时，浏览器不需重新计算元素的几何属性、直接为该元素绘制新的样式（跳过了上图所示的回流）。

流环节）。这个过程叫做重绘。

由此我们可以看出，**重绘不一定导致回流，回流一定会导致重绘**。硬要比较的话，回流比重绘做的事情更多，带来的开销也更大。但这两个说到底都是吃性能的，所以都不是什么善茬。我们在开发中，要从代码层面出发，尽可能把回流和重绘的次数最小化。

#药到病除：给你的 DOM “提提速”

知道了 DOM 慢的原因，我们就可以对症下药了。

1. 减少 DOM 操作：少交“过路费”、避免过度渲染

我们来看这样一个⌚，HTML 内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>DOM操作测试</title>
</head>
<body>
  <div id="container"></div>
</body>
</html>
```

此时我有一个假需求——我想往 `container` 元素里写 `10000` 句一样的话。如果我这么做：

```
for(var count=0;count<10000;count++){
  document.getElementById('container').innerHTML+='<span>我是一个小测试</span>'}
```

这段代码有两个明显的可优化点。

第一点，**过路费交太多了**。我们每一次循环都调用 DOM 接口重新获取了一次 `container` 元素，相当于每次循环都交了一次过路费。前后交了 10000 次过路费，但其中 9999 次过路费都可以用**缓存变量**的方式节省下来：

```
// 只获取一次container
let container = document.getElementById('container')
for(let count=0;count<10000;count++){
  container.innerHTML += '<span>我是一个小测试</span>'}
```

第二点，**不必要的 DOM 更改太多了**。我们的 10000 次循环里，修改了 10000 次 DOM 树。我们前面说过，对 DOM 的修改会引发渲染树的改变、进而去走一个（可能的）回流或重绘的过程，而这个过程的开销是很“贵”的。这么贵的操作，我们竟然重复执行了 N 多次！其实我们可以通过**就事论事**的方式节省下来不必要的渲染：

```

let container = document.getElementById('container')
let content = ''
for(let count=0;count<10000;count++){
    // 先对内容进行操作
    content += '<span>我是一个小测试</span>'
}
// 内容处理好了，最后再触发DOM的更改
container.innerHTML = content

```

所谓“就事论事”，就像大家所看到的：JS 层面的事情，JS 自己去处理，处理好了，再来找 DOM 打报告。

事实上，考虑 JS 的运行速度，比 DOM 快得多这个特性。我们减少 DOM 操作的核心思路，就是**让 JS 去给 DOM 分压**。

这个思路，在[DOM Fragment](#) 中体现得淋漓尽致。

`DocumentFragment` 接口表示一个没有父级文件的最小文档对象。它被当做一个轻量版的 `Document` 使用，用于存储已排好版的或尚未打理好格式的 XML 片段。因为 `DocumentFragment` 不是真实 DOM 树的一部分，它的变化不会引起 DOM 树的重新渲染的操作（`reflow`），且不会导致性能等问题。

- 在我们上面的例子里，字符串变量 `content` 就扮演着一个 `DOM Fragment` 的角色。其实无论字符串变量也好，`DOM Fragment` 也罢，它们本质上都作为脱离了真实 DOM 树的容器出现，用于缓存批量化的 DOM 操作。
- 前面我们直接用 `innerHTML` 去拼接目标内容，这样做固然有用，但却不够优雅。相比之下，`DOM Fragment` 可以帮助我们用更加结构化的方式去达成同样的目的，从而在维持性能的同时，保住我们代码的可拓展和可维护性。我们现在用 `DOM Fragment` 来改写上面的例子：

```

let container = document.getElementById('container')
// 创建一个DOM Fragment对象作为容器
let content = document.createDocumentFragment()
for(let count=0;count<10000;count++){
    // span此时可以通过DOM API去创建
    let oSpan = document.createElement("span")
    oSpan.innerHTML = '我是一个小测试'
    // 像操作真实DOM一样操作DOM Fragment对象
    content.appendChild(oSpan)
}
// 内容处理好了，最后再触发真实DOM的更改
container.appendChild(content)

```

- 我们运行这段代码，可以得到与前面两种写法相同的运行结果。
- 可以看出，`DOM Fragment` 对象允许我们像操作真实 DOM 一样去调用各种各样的 DOM API，我们的代码质量因此得到了保证。并且它的身份也非常纯粹：当我们试图将其 `append` 进真实 DOM 时，它会在乖乖交出自身缓存的所有后代节点后**全身而退**，完美地完成一个容器的使命，而不会出现在真实的 DOM 结构中。这种结构化、干净利落的特性，使得 `DOM Fragment` 作为经典的性能优化手段大受欢迎，这一点在 jQuery、Vue 等优秀前端框架的源码中均有体现。
- 相比 DOM 命题的博大精深，一个简单的循环 Demo 显然不能说明所有问题。不过不用着急，在本节，我只希望大家能牢记原理与宏观思路。“药到病除”到这里才刚刚开了个头，下个小节，我们将深挖事件循环机制，从而深入 JS 层面的生产实践。

#十、渲染篇 4：千方百计——Event Loop 与异步更新策略

Vue 和 React 都实现了异步更新策略。虽然实现的方式不尽相同，但都达到了减少 DOM 操作、避免过度渲染的目的。通过研究框架的运行机制，其设计思路将深化我们对 DOM 优化的理解，其实现手法将拓宽我们对 DOM 实践的认知。

本节我们将基于 Event Loop 机制，对 Vue 的异步更新策略作探讨。

#前置知识：Event Loop 中的“渲染时机”

搞懂 Event Loop，是理解 Vue 对 DOM 操作优化的第一步。

#Micro-Task 与 Macro-Task

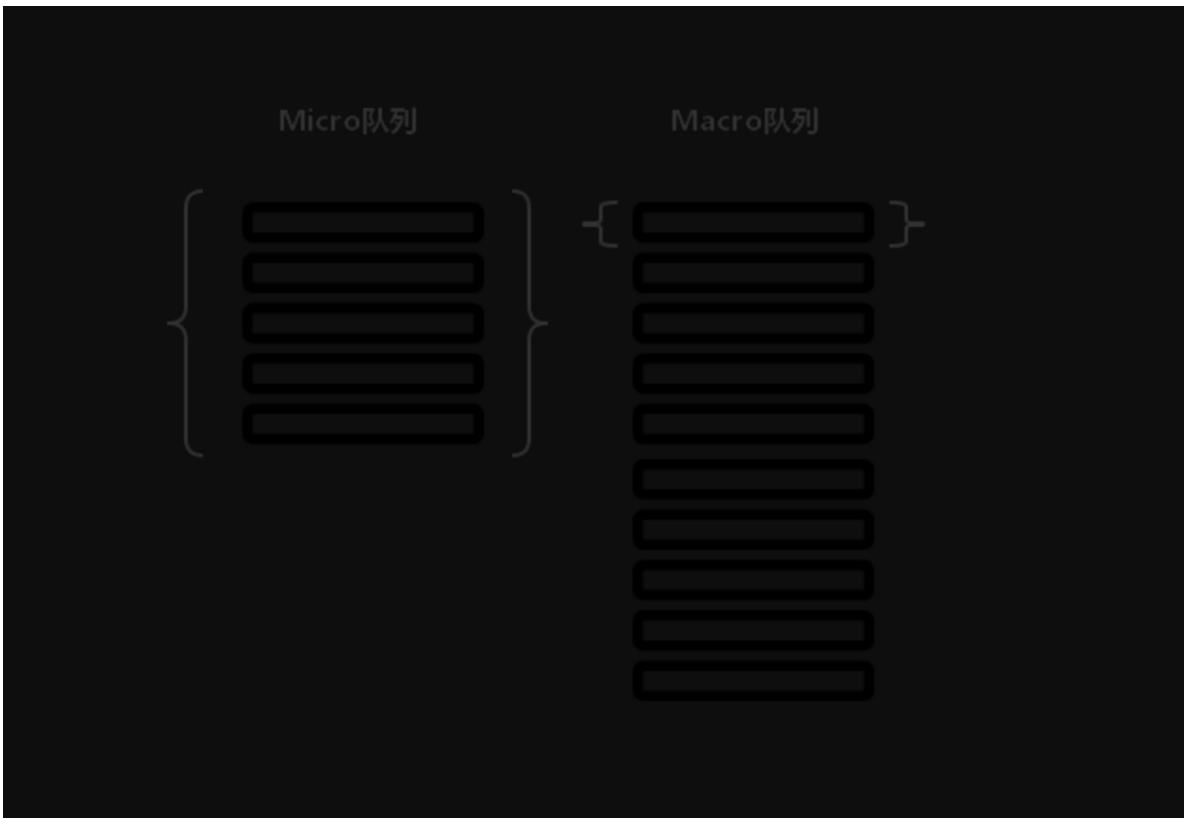
- 事件循环中的异步队列有两种：`macro`（宏任务）队列和 `micro`（微任务）队列。
- 常见的 `macro-task` 比如：`setTimeout`、`setInterval`、`setImmediate`、`script`（整体代码）、`I/O` 操作、`UI` 渲染等。
- 常见的 `micro-task` 比如：`process.nextTick`、`Promise`、`MutationObserver` 等。

#Event Loop 过程解析

基于对 micro 和 macro 的认知，我们来走一遍完整的事件循环过程。

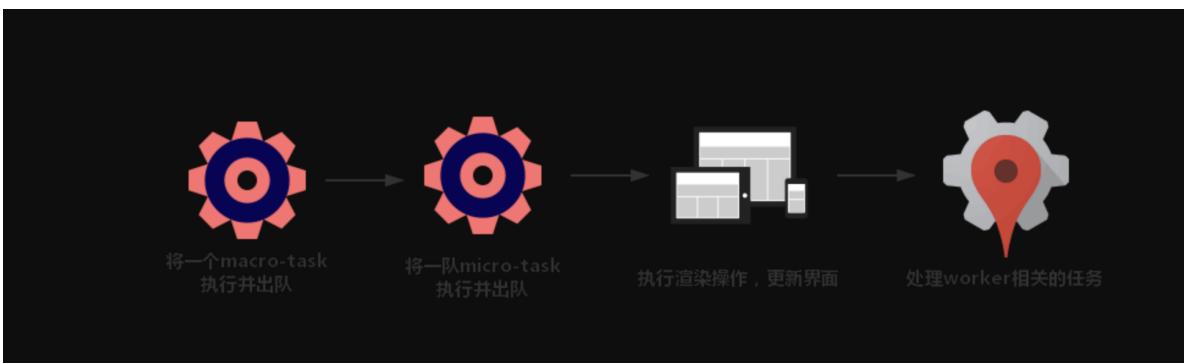
一个完整的 Event Loop 过程，可以概括为以下阶段：

- 初始状态：调用栈空。`micro` 队列空，`macro` 队列里有且只有一个 `script` 脚本（整体代码）。
- 全局上下文（`script` 标签）被推入调用栈，同步代码执行。在执行的过程中，通过对一些接口的调用，可以产生新的 `macro-task` 与 `micro-task`，它们会分别被推入各自的任务队列里。同步代码执行完了，`script` 脚本会被移出 `macro` 队列，**这个过程本质上是队列的 macro-task 的执行和出队的过程。**
- 上一步我们出队的是一个 `macro-task`，这一步我们处理的是 `micro-task`。但需要注意的是：当 `macro-task` 出队时，任务是一个一个执行的；而 `micro-task` 出队时，任务是一队一队执行的（如下图所示）。因此，我们处理 `micro` 队列这一步，会逐个执行队列中的任务并把它出队，直到队列被清空。



- 执行渲染操作，更新界面（敲黑板划重点）。
- 检查是否存在 web worker 任务，如果有，则对其进行处理。
- 上述过程循环往复，直到两个队列都清空

我们总结一下，每一次循环都是一个这样的过程：



#渲染的时机

- 大家现在思考一个这样的问题：假如我想要在异步任务里进行DOM更新，我该把它包装成 micro 还是 macro 呢？
- 我们先假设它是一个 macro 任务，比如我在 script 脚本中用 setTimeout 来处理它：

```
// task是一个用于修改DOM的回调
setTimeout(task, 0)
```

- 现在 task 被推入的 macro 队列。但因为 script 脚本本身是一个 macro 任务，所以本次执行完 script 脚本之后，下一个步骤就要去处理 micro 队列了，再往下就去执行了一次 render，对不对？
- 但本次 render 我的目标 task 其实并没有执行，想要修改的 DOM 也没有修改，因此这一次的 render 其实是一次无效的 render。

`macro` 不 ok，我们转向 `micro` 试试看。我用 `Promise` 来把 `task` 包装成是一个 `micro` 任务：

```
Promise.resolve().then(task)
```

那么我们结束了对 script 脚本的执行，是不是紧接着就去处理 micro-task 队列了？micro-task 处理完，DOM 修改好了，紧接着就可以走 render 流程了——不需要再消耗多余的一次渲染，不需要再等待一轮事件循环，直接为用户呈现最即时的更新结果。

因此，我们更新 DOM 的时间点，应该尽可能靠近渲染的时机。**当我们需要在异步任务中实现 DOM 修 改时，把它包装成 micro 任务是相对明智的选择。**

#生产实践：异步更新策略——以 Vue 为例

什么是异步更新？

当我们使用 Vue 或 React 提供的接口去更新数据时，这个更新并不会立即生效，而是会被推入到一个队列里。待到适当的时机，队列中的更新任务会被**批量触发**。这就是异步更新。

异步更新可以帮助我们避免过度渲染，是我们上节提到的“让 JS 为 DOM 分压”的典范之一。

1. 异步更新的优越性

- 异步更新的特性在于它**只看结果**，因此渲染引擎**不需要为过程买单**。
- 最典型的例子，比如有时我们会遇到这样的情况：

```
// 任务一
this.content = '第一次测试'
// 任务二
this.content = '第二次测试'
// 任务三
this.content = '第三次测试'
```

我们在三个更新任务中对同一个状态修改了三次，如果我们采取传统的同步更新策略，那么就要操作三次 DOM。但本质上需要呈现给用户的目标内容其实只是第三次的结果，也就是说只有第三次的操作是有意义的——我们白白浪费了两次计算。

但如果我们把这三个任务塞进异步更新队列里，它们会先在 JS 的层面上被**批量执行完毕**。当流程走到渲染这一步时，它仅仅需要针对有意义的计算结果操作一次 DOM——这就是异步更新的妙处。

2. Vue状态更新手法：nextTick

Vue 每次想要更新一个状态的时候，会先把它这个更新操作给包装成一个异步操作派发出去。这件事情，在源码中是由一个叫做 `nextTick` 的函数来完成的：

```
export function nextTick (cb?: Function, ctx?: Object) {
  let _resolve
  callbacks.push(() => {
    if (cb) {
      try {
        cb(ctx)
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
}
```

```

        }
        // 检查上一个异步任务队列（即名为callbacks的任务数组）是否派发和执行完毕了。pending此处相当于一个锁
        if (!pending) {
            // 若上一个异步任务队列已经执行完毕，则将pending设定为true（把锁锁上）
            pending = true
            // 是否要求一定要派发为macro任务
            if (useMacroTask) {
                macroTimerFunc()
            } else {
                // 如果不说明一定要macro 你们就全都是micro
                microTimerFunc()
            }
        }
        // $flow-disable-line
        if (!cb && typeof Promise !== 'undefined') {
            return new Promise(resolve => {
                _resolve = resolve
            })
        }
    }
}

```

- 我们看到，Vue 的异步任务默认情况下都是用 `Promise` 来包装的，也就是说它们都是 `micro-task`。这一点和我们“前置知识”中的渲染时机的分析不谋而合。
- 为了带大家熟悉一下常见的 `macro` 和 `micro` 派发方式、加深对 `Event Loop` 的理解，我们继续细化解析一下 `macroTimeFunc()` 和 `microTimeFunc()` 两个方法。

`macroTimeFunc()` 是这么实现的：

```

// macro首选setImmediate 这个兼容性最差
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
    macroTimerFunc = () => {
        setImmediate(flushCallbacks)
    }
} else if (typeof MessageChannel !== 'undefined' && (
    isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
    const channel = new MessageChannel()
    const port = channel.port2
    channel.port1.onmessage = flushCallbacks
    macroTimerFunc = () => {
        port.postMessage(1)
    }
} else {
    // 兼容性最好的派发方式是setTimeout
    macroTimerFunc = () => {
        setTimeout(flushCallbacks, 0)
    }
}

```

`microTimeFunc()` 是这么实现的：

```

// 简单粗暴 不是ios全都给我去Promise 如果不兼容promise 那么你只能将就一下变成macro了
if (typeof Promise !== 'undefined' && isNative(Promise)) {

```

```

const p = Promise.resolve()
microTimerFunc = () => {
  p.then(flushCallbacks)
  // in problematic UIWebViews, Promise.then doesn't completely break, but
  // it can get stuck in a weird state where callbacks are pushed into the
  // microtask queue but the queue isn't being flushed, until the browser
  // needs to do some other work, e.g. handle a timer. Therefore we can
  // "force" the microtask queue to be flushed by adding an empty timer.
  if (isIOS) setTimeout(noop)
}
} else {
  // 如果无法派发macro, 就退而求其次派发为macro
  microTimerFunc = macroTimerFunc
}

```

我们注意到，无论是派发 `macro` 任务还是派发 `micro` 任务，派发的任务对象都是一个叫做 `flushCallbacks` 的东西，这个东西做了什么呢？

`flushCallbacks` 源码如下：

```

function flushCallbacks () {
  pending = false
  // callbacks在nexttick中出现过 它是任务数组（队列）
  const copies = callbacks.slice(0)
  callbacks.length = 0
  // 将callbacks中的任务逐个取出执行
  for (let i = 0; i < copies.length; i++) {
    copies[i]()
  }
}

```

- 现在我们理清楚了：`vue` 中每产生一个状态更新任务，它就会被塞进一个叫 `callbacks` 的数组（此处是任务队列的实现形式）中。这个任务队列在被丢进 `micro` 或 `macro` 队列之前，会先去检查当前是否有异步更新任务正在执行（即检查 `pending` 锁）。如果确认 `pending` 锁是开着的（`false`），就把它设置为锁上（`true`），然后对当前 `callbacks` 数组的任务进行派发（丢进 `micro` 或 `macro` 队列）和执行。设置 `pending` 锁的意义在于保证状态更新任务的有序进行，避免发生混乱。
- 本小节我们从性能优化的角度出发，通过解析Vue源码，对异步更新这一高效的 DOM 优化手段有了感性的认知。同时帮助大家进一步熟悉了 `micro` 与 `macro` 在生产中的应用，加深了对 Event Loop 的理解。事实上，Vue 源码中还有许多值得称道的生产实践，其设计模式与编码细节都值得我们去细细品味。对这个话题感兴趣的同学，课后不妨移步 [Vue运行机制解析](#) 进行探索。

#小结

- 至此，我们的 DOM 优化之路才走完了一半。
- 以上我们都在讨论“如何减少 DOM 操作”的话题。这个话题比较宏观——DOM 操作也分很多种，它们带来的变化各不相同。有的操作只触发重绘，这时我们的性能损耗就小一些；有的操作会触发回流，这时我们更“肉疼”一些。那么如何理解回流与重绘，如何借助这些理解去提升页面渲染效率呢？
- 结束了 JS 的征程，我们下面就走进 CSS 的世界一窥究竟。

#十一、渲染篇 5：最后一击——回流 (Reflow) 与重绘 (Repaint)

开篇我们先对上上节介绍的回流与重绘的基础知识做个复习（跳读的同学请自觉回到上上节补齐→_→）。

回流：当我们对 DOM 的修改引发了 DOM 几何尺寸的变化（比如修改元素的宽、高或隐藏元素等）时，浏览器需要重新计算元素的几何属性（其他元素的几何属性和位置也会因此受到影响），然后再将计算的结果绘制出来。这个过程就是回流（也叫重排）。

重绘：当我们对 DOM 的修改导致了样式的变化、却并未影响其几何属性（比如修改了颜色或背景色）时，浏览器不需重新计算元素的几何属性、直接为该元素绘制新的样式（跳过了上图所示的回流环节）。这个过程叫做重绘。

由此我们可以看出，**重绘不一定导致回流，回流一定会导致重绘**。硬要比较的话，回流比重绘做的事情更多，带来的开销也更大。但这两个说到底都是吃性能的，所以都不是什么善茬。我们在开发中，要从代码层面出发，尽可能把回流和重绘的次数最小化。

#哪些实际操作会导致回流与重绘

- 要避免回流与重绘的发生，最直接的做法是避免掉可能会引发回流与重绘的 DOM 操作，就好像拆弹专家在解决一颗炸弹时，最重要的是掐灭它的导火索。
- 触发重绘的“导火索”比较好识别——只要是不触发回流，但又触发了样式改变的 DOM 操作，都会引起重绘，比如背景色、文字色、可见性(可见性这里特指形如`visibility: hidden`这样不改变元素位置和存在性的、单纯针对可见性的操作，注意与`display:none`进行区分)等。为此，我们要着重理解一下那些可能触发回流的操作。

1. 回流的“导火索”

最“贵”的操作：改变 DOM 元素的几何属性

- 这个改变几乎可以说是“牵一发动全身”——当一个DOM元素的几何属性发生变化时，所有和它相关的节点（比如父子节点、兄弟节点等）的几何属性都需要进行重新计算，它会带来巨大的计算量。
- 常见的几何属性有 `width`、`height`、`padding`、`margin`、`left`、`top`、`border` 等等。此处不再给大家一一列举。有的文章喜欢罗列属性表格，但我相信我今天列出来大家也不会看、看了也记不住（因为太多了）。我自己也不会去记这些——其实确实没必要记，一个属性是不是几何属性、会不会导致空间布局发生变化，大家写样式的时候完全可以通过代码效果看出来。多说无益，还希望大家可以多写多试，形成自己的“肌肉记忆”。
- “价格适中”的操作：改变 DOM 树的结构

这里主要指的是节点的增减、移动等操作。浏览器引擎布局的过程，顺序上可以类比于树的前序遍历——它是一个从上到下、从左到右的过程。通常在这个过程中，当前元素不会再影响其前面已经遍历过的元素。

- 最容易被忽略的操作：获取一些特定属性的值

当你要用到像这样的属性：`offsetTop`、`offsetLeft`、`offsetWidth`、`offsetHeight`、`scrollTop`、`scrollLeft`、`scrollWidth`、`scrollHeight`、`clientTop`、`clientLeft`、`clientWidth`、`clientHeight` 时，你就要注意了！

- “像这样”的属性，到底是像什么样？——这些值有一个共性，就是需要通过即时计算得到。因此浏览器为了获取这些值，也会进行回流。
- 除此之外，当我们调用了 `getComputedStyle` 方法，或者 IE 里的 `currentStyle` 时，也会触发回流。原理是一样的，都为求一个“即时性”和“准确性”。

#如何规避回流与重绘

了解了回流与重绘的“导火索”，我们就要尽量规避它们。但很多时候，我们不得不使用它们。当避无可避时，我们就要学会更聪明地使用它们。

1. 将“导火索”缓存起来，避免频繁改动

有时我们想要通过多次计算得到一个元素的布局位置，我们可能会这样做：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    #el {
      width: 100px;
      height: 100px;
      background-color: yellow;
      position: absolute;
    }
  </style>
</head>
<body>
  <div id="el"></div>
  <script>
    // 获取el元素
    const el = document.getElementById('el')
    // 这里循环判定比较简单，实际中或许会拓展出比较复杂的判定需求
    for(let i=0;i<10;i++) {
      el.style.top = el.offsetTop + 10 + "px";
      el.style.left = el.offsetLeft + 10 + "px";
    }
  </script>
</body>
</html>
```

这样做，每次循环都需要获取多次“敏感属性”，是比较糟糕的。我们可以将其以 JS 变量的形式缓存起来，待计算完毕再提交给浏览器发出重计算请求：

```
// 缓存offsetLeft与offsetTop的值
const el = document.getElementById('el')
let offLeft = el.offsetLeft, offTop = el.offsetTop

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

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

2. 避免逐条改变样式，使用类名去合并样式

比如我们可以把这段单纯的代码：

```
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

优化成一个有 `class` 加持的样子：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    .basic_style {
      width: 100px;
      height: 200px;
      border: 10px solid red;
      color: red;
    }
  </style>
</head>
<body>
  <div id="container"></div>
  <script>
    const container = document.getElementById('container')
    container.classList.add('basic_style')
  </script>
</body>
</html>
```

- 前者每次单独操作，都去触发一次渲染树更改，从而导致相应的回流与重绘过程。
- 合并之后，等于我们将所有的更改一次性发出，用一个 style 请求解决掉了。

3. 将 DOM “离线”

我们上文所说的回流和重绘，都是在“该元素位于页面上”的前提下会发生的。一旦我们给元素设置 `display: none`，将其从页面上“拿掉”，那么我们的后续操作，将无法触发回流与重绘——这个将元素“拿掉”的操作，就叫做 DOM 离线化。

仍以我们上文的代码片段为例：

```
const container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
... (省略了许多类似的后续操作)
```

离线化后就是这样：

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

有的同学会问，拿掉一个元素再把它放回去，这不也会触发一次昂贵的回流吗？这话不假，但我们把它拿下来了，后续不管我操作这个元素多少次，每一步的操作成本都会非常低。当我们只需要进行很少的 DOM 操作时，DOM 离线化的优越性确实不太明显。一旦操作频繁起来，这“拿掉”和“放回”的开销都将会是非常值得的。

#Flush 队列：浏览器并没有那么简单

- 以我们现在的知识基础，理解上面的优化操作并不难。那么现在我问大家一个问题：

```
let container = document.getElementById('container')
container.style.width = '100px'
container.style.height = '200px'
container.style.border = '10px solid red'
container.style.color = 'red'
```

这段代码里，浏览器进行了多少次的回流或重绘呢？

“`width`、`height`、`border` 是几何属性，各触发一次回流；`color` 只造成外观的变化，会触发一次重绘。”——如果你立刻这么想了，说明你是个能力不错的同学，认真阅读了前面的内容。那么我们现在立刻跑一跑这段代码，看看浏览器怎么说：



- 这里为大家截取有“Layout”和“Paint”出镜的片段（这个图是通过 Chrome 的 Performance 面板得到的，后面会教大家用这个东西）。我们看到浏览器只进行了一次回流和一次重绘——和我们想的不一样啊，为啥呢？
- 因为现代浏览器是很聪明的。浏览器自己也清楚，如果每次 DOM 操作都即时地反馈一次回流或重绘，那么性能上来说是扛不住的。于是它自己缓存了一个 flush 队列，把我们触发的回流与重绘任务都塞进去，待到队列里的任务多起来、或者达到了一定的时间间隔，或者“不得已”的时候，再将这些任务一口气出队。因此我们看到，上面就算我们进行了 4 次 DOM 更改，也只触发了一次 Layout 和一次 Paint。

大家这里尤其小心这个“不得已”的时候。前面我们在介绍回流的“导火索”的时候，提到过有一类属性很特别，它们有很强的“即时性”。当我们访问这些属性时，浏览器会为了获得此时此刻的、最准确的属性值，而提前将 flush 队列的任务出队——这就是所谓的“不得已”时刻。具体是哪些属性值，我们已经在“最容易被忽略的操作”这个小模块介绍过了，此处不再赘述。

#小结

- 整个一节读下来，可能会有同学感到疑惑：既然浏览器已经为我们做了批处理优化，为什么我们还要自己操心这么多事情呢？今天避免这个明天避免那个，多麻烦！
- 问题在于，**并不是所有的浏览器都是聪明的**。我们刚刚的性能图表，是 Chrome 的开发者工具呈现给我们的。Chrome 里行得通的东西，到了别处（比如 IE）就不一定行得通了。而我们并不知道用户会使用什么样的浏览器。如果不手动做优化，那么一个页面在不同的环境下就会呈现不同的性能

效果，这对我们、对用户都是不利的。因此，养成良好的编码习惯、从根源上解决问题，仍然是最周全的方法。

#十二、应用篇 1：优化首屏体验——Lazy-Load 初探

首先要告诉大家的是，截止到上个章节，我们需要大家绞尽脑汁去理解的“硬核”操作基本告一段落了。从本节开始，我们会一起去实现一些必知必会、同时难度不大的常用优化手段。

这部分内容不难，但很关键。尤其是近期有校招或跳槽需求的同学，还请务必对这部分内容多加留心，说不定下一次的面试题里就有它们的身影。

#Lazy-Load 初相见

`Lazy-Load`，翻译过来是“懒加载”。它是针对图片加载时机的优化：在一些图片量比较大的网站（比如电商网站首页，或者团购网站、小游戏首页等），如果我们尝试在用户打开页面的时候，就把所有的图片资源加载完毕，那么很可能会造成白屏、卡顿等现象，因为图片真的太多了，一口气处理这么多任务，浏览器做不到啊！

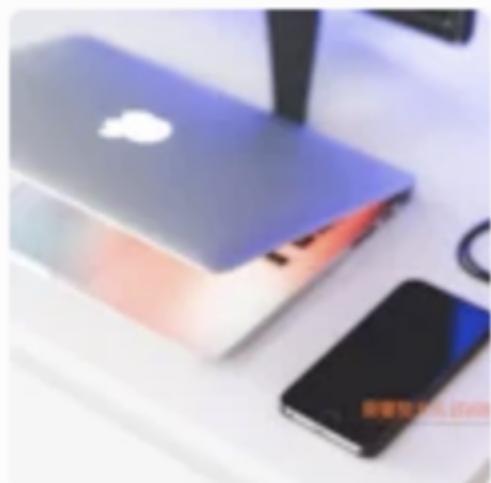
但我们再想，用户真的需要这么多图片吗？不对，用户点开页面的瞬间，呈现给他的只有屏幕的一部分（我们称之为首屏）。只要我们可以在页面打开的时候把首屏的图片资源加载出来，用户就会认为页面是没问题的。至于下面的图片，我们完全可以等用户下拉的瞬间再即时去请求、即时呈现给他。这样一来，性能的压力小了，用户的体验却没有变差——这个延迟加载的过程，就是 `Lazy-Load`。

现在我们打开掘金首页：

The screenshot shows the Zhihu homepage with several article cards. Each card includes a thumbnail image on the right side. The first card is about a developer's success story, the second is about Git command tips, the third is about naming conventions, and the fourth is about Flutter calculator development. Each card has a like count and comment count below it.

- 做完小程序项目、老板给我加了6k薪资～**
热 · 专栏 · baldwin · 5天前 · Vue.js
144 546
- 还在记git命令？快试试SourceTree**
热 · 专栏 · baldwin · 5天前 · Git
144 272
- [前端开发]--分享个人习惯的命名方式**
热 · 专栏 · 守候i · 9天前 · HTML
144 757
- Flutter最佳入门方式——写一个计算器**
热 · 专栏 · 韶(▽▽) · 4天前 · iOS
144 145

大家留意一栏文章右侧可能会出现的图片，这里咱们给个特写：





大家现在以尽可能快的速度，疯狂向下拉动页面。发现什么？是不是发现我们图示的这个图片的位置，会出现闪动——有时候我们明明已经拉到目标位置了，文字也呈现完毕了，图片却慢半拍才显示出来。这是因为，掘金首页也采用了懒加载策略。当我们的页面并未滚动至包含图片的 div 元素所在的位置时，它的样式是这样的：

```
▼ <div data-v-b2db8566 data-v-009ea7bb data-v-6b46a625 data-src="https://user-gold-cdn.xitu.io/2018/9/27/16619f449ee24252?imageView2/1/w/120/h/120/q/85/format/webp/interlace/1" class="lazy thumb thumb" style="background-image: none; background-size: cover;"> = $0
```

我们把代码提出来看一下：

```
<div data-v-b2db8566=""  
      data-v-009ea7bb=""  
      data-v-6b46a625=""  
      data-src="https://user-gold-cdn.xitu.io/2018/9/27/16619f449ee24252?  
imageview2/1/w/120/h/120/q/85/format/webp/interlace/1"  
      class="lazy thumb thumb"  
      style="background-image: none; background-size: cover;">>  
</div>
```

- 我们注意到 `style` 内联样式中，背景图片设置为了 `none`。也就是说这个 `div` 是没有内容的，它只起到一个**占位**的作用。
- 这个“占位”的概念，在这个例子里或许体现得不够直观。最直观的应该是淘宝首页的 HTML Preview 效果：

A screenshot of the Network tab in Chrome DevTools. The left sidebar shows a list of resources under the domain 'www.taobao.com', including 'search-suggest...', 'blk.html', '92ed2d80-06c...', 'creation-27084t...', and '7796533.html'. The main pane displays the preview of a Taobao page. The page has a complex layout with multiple sections and images. Some parts of the page, particularly the image placeholders, appear to be loading slowly or not fully, illustrating the concept of lazy loading where the page structure is visible before the images are loaded.

- 我们看到，这个还没来得及被图片填充完全的网页，是用大大小小的空 `div` 元素来占位的。掘金首页也是如此。

- 一旦我们通过滚动使得这个 `div` 出现在了可见范围内，那么 `div` 元素的内容就会发生变化，呈现如下的内容：

```
<div data-v-b2db8566 data-v-009ea7bb data-v-6b46a625 data-src="https://user-gold-cdn.xitu.io/2018/9/27/16619f449ee24252?imageView2/1/w/120/h/120/q/85/format/webp/interlace/1" class="lazythumb thumb loaded" style="background-image: url("https://user-gold-cdn.xitu.io/2018/9/27/16619f449ee24252?imageView2/1/w/120/h/120/q/85/format/webp/interlace/1"); background-size: cover;">
  ::before
</div>
```

我们给 `style` 一个特写：

```
style="background-image: url("https://user-gold-cdn.xitu.io/2018/9/27/16619f449ee24252?imageView2/1/w/120/h/120/q/85/format/webp/interlace/1"); background-size: cover;"
```

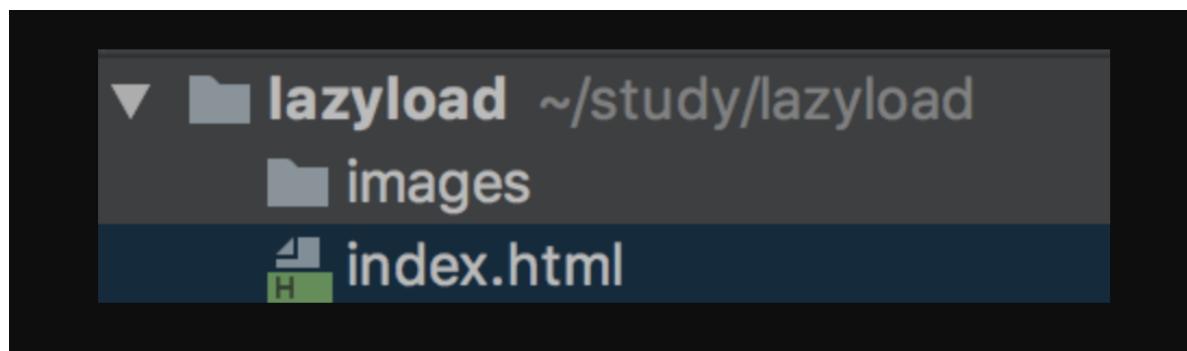
可以看出，`style` 内联样式中的背景图片属性从 `none` 变成了一个在线图片的 `URL`。也就是说，出现在可视区域的瞬间，`div` 元素的内容被即时地修改掉了——它被写入了有效的图片 `URL`，于是图片才得以呈现。这就是懒加载的实现思路。

#一起写一个 Lazy-Load 吧！

基于上面的实现思路，我们完全可以手动实现一个属于自己的 `Lazy-Load`。

此处敲黑板划重点，Lazy-Load 的思路及实现方式为大厂面试常考题，还望诸位同学引起重视

首先新建一个空项目，目录结构如下：



- 大家可以往 `images` 文件夹里塞入各种各样自己喜欢的图片。
- 我们在 `index.html` 中，为这些图片预置 `img` 标签：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Lazy-Load</title>
  <style>
    .img {
      width: 200px;
      height: 200px;
      background-color: gray;
    }
  </style>
</head>
<body>
  
</body>
</html>
```

```

.pic {
    // 必要的img样式
}
</style>
</head>
<body>
<div class="container">
    <div class="img">
        // 注意我们并没有为它引入真实的src
        
    </div>
    <div class="img">
        
    </div>
</div>
</body>
</html>

```

- 在懒加载的实现中，有两个关键的数值：一个是**当前可视区域的高度**，另一个是**元素距离可视区域顶部的高度**。
- 当前可视区域的高度**，在和现代浏览器及 IE9 以上的浏览器中，可以用 `window.innerHeight` 属性获取。在低版本 IE 的标准模式中，可以用 `document.documentElement.clientHeight` 获得，这里我们兼容两种情况：

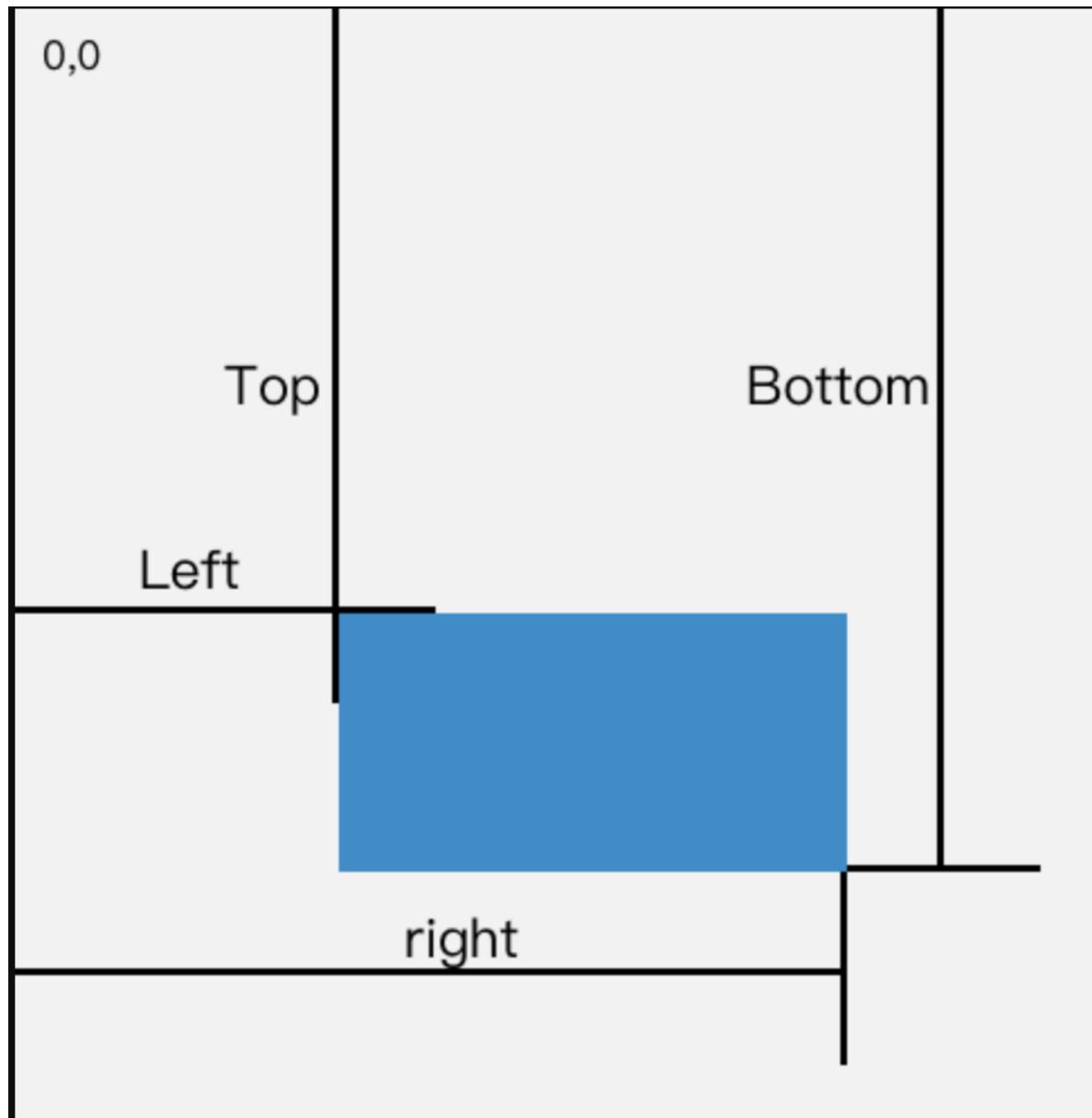
```
const viewHeight = window.innerHeight || document.documentElement.clientHeight
```

而**元素距离可视区域顶部的高度**，我们这里选用 `getBoundingClientRect()` 方法来获取返回元素的大小及其相对于视口的位置。对此 MDN 给出了非常清晰的解释：

该方法的返回值是一个 `DOMRect` 对象，这个对象是由该元素的 `getClientRects()` 方法返回的一组矩形的集合，即：是与该元素相关的 `css` 边框集合。

`DOMRect` 对象包含了一组用于描述边框的只读属性——`left`、`top`、`right` 和 `bottom`，单位为像素。除了 `width` 和 `height` 外的属性都是相对于视口的左上角位置而言的。

其中需要引起我们注意的就是 `left`、`top`、`right` 和 `bottom`，它们对应到元素上是这样的：



可以看出，`top` 属性代表了元素距离可视区域顶部的高度，正好可以为我们所用！

Lazy-Load 方法开工啦！

```
<script>
    // 获取所有的图片标签
    const imgs = document.getElementsByTagName('img')
    // 获取可视区域的高度
    const viewHeight = window.innerHeight ||
        document.documentElement.clientHeight
    // num用于统计当前显示到了哪一张图片，避免每次都从第一张图片开始检查是否露出
    let num = 0
    function lazyload(){
        for(let i=num; i<imgs.length; i++) {
            // 用可视区域高度减去元素顶部距离可视区域顶部的高度
            let distance = viewHeight - imgs[i].getBoundingClientRect().top
            // 如果可视区域高度大于等于元素顶部距离可视区域顶部的高度，说明元素露出
            if(distance >= 0 ){
                // 给元素写入真实的src，展示图片
            }
        }
    }

```

```
    imgs[i].src = imgs[i].getAttribute('data-src')
    // 前i张图片已经加载完毕，下次从第i+1张开始检查是否露出
    num = i + 1
}
}

// 监听scroll事件
window.addEventListener('scroll', lazyload, false);
</script>
```

#小结

- 本节我们实现出了一个最基本的懒加载功能。但是大家要注意一点：这个 scroll 事件，是一个危险的事件——它太容易被触发了。试想，用户在访问网页的时候，是不是可以无限次地去触发滚动？尤其是一个页面死活加载不出来的时候，疯狂调戏鼠标滚轮（或者浏览器滚动条）的用户可不在少数啊！
- 再回头看看我们上面写的代码。按照我们的逻辑，用户的每一次滚动都将触发我们的监听函数。函数执行是吃性能的，频繁地响应某个事件将造成大量不必要的页面计算。因此，我们需要针对那些有可能被频繁触发的事件作进一步地优化。这里就引出了我们下一节的两位主角——throttle 与 debounce。

#十三、应用篇 2：事件的节流 (throttle) 与防抖 (debounce)

上一节我们一起通过监听滚动事件，实现了各大网站喜闻乐见的懒加载效果。但我们提到，scroll 事件是一个非常容易被反复触发的事件。其实不止 scroll 事件，resize 事件、鼠标事件（比如 mousemove、mouseover 等）、键盘事件（keyup、keydown 等）都存在被频繁触发的风险。

- 频繁触发回调导致的大量计算会引发页面的抖动甚至卡顿。为了规避这种情况，我们需要一些手段来控制事件被触发的频率。就是在这样的背景下，throttle（事件节流）和 debounce（事件防抖）出现了。

#“节流”与“防抖”的本质

这两个东西都以闭包的形式存在。

它们通过对事件对应的回调函数进行包裹、以自由变量的形式缓存时间信息，最后用 setTimeout 来控制事件的触发频率。

#Throttle：第一个人说了算

throttle 的中心思想在于：在某段时间内，不管你触发了多少次回调，我都只认第一次，并在计时结束时给予响应。

先给大家讲个小故事：现在有一个旅客刚下了飞机，需要用车，于是打电话叫了该机场唯一的一辆机场大巴来接。司机开到机场，心想来都来了，多接几个人一起走吧，这样这趟才跑得值——我等个十分钟看看。于是司机一边打开了计时器，一边招呼后面的客人陆陆续续上车。在这十分钟内，后面下飞机的乘客都只能乘这一辆大巴，十分钟过去后，不管后面还有多少没挤上车的乘客，这班车都必须发走。

在这个故事里，“司机”就是我们的节流阀，他控制发车的时机；“乘客”就是因为我们频繁操作事件而不断涌入的回调任务，它需要接受“司机”的安排；而“计时器”，就是我们上文提到的以自由变量形式存在的时间信息，它是“司机”决定发车的依据；最后“发车”这个动作，就对应到回调函数的执行。

总结下来，所谓的“节流”，是通过在一段时间内无视后来产生的回调请求来实现的。只要一位客人叫了车，司机就会为他开启计时器，一定的时间内，后面需要乘车的客人都得排队上这一辆车，谁也无法叫到更多的车。

对应到实际的交互上是一样一样的：每当用户触发了一次 `scroll` 事件，我们就为这个触发操作开启计时器。一段时间内，后续所有的 `scroll` 事件都会被当作“一辆车的乘客”——它们无法触发新的 `scroll` 回调。直到“一段时间”到了，第一次触发的 `scroll` 事件对应的回调才会执行，而“一段时间内”触发的后续的 `scroll` 回调都会被节流阀无视掉。

理解了大致的思路，我们现在一起实现一个 `throttle`：

```
// fn是我们需要包装的事件回调，interval是时间间隔的阈值
function throttle(fn, interval) {
  // last为上一次触发回调的时间
  let last = 0

  // 将throttle处理结果当作函数返回
  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments
    // 记录本次触发回调的时间
    let now = +new Date()

    // 判断上次触发的时间和本次触发的时间差是否小于时间间隔的阈值
    if (now - last >= interval) {
      // 如果时间间隔大于我们设定的时间间隔阈值，则执行回调
      last = now;
      fn.apply(context, args);
    }
  }
}

// 用throttle来包装scroll的回调
const better_scroll = throttle(() => console.log('触发了滚动事件'), 1000)

document.addEventListener('scroll', better_scroll)
```

#Debounce：最后一个人说了算

防抖的中心思想在于：我会等你到底。在某段时间内，不管你触发了多少次回调，我都只认最后一次。

继续讲司机开车的故事。这次的司机比较有耐心。第一个乘客上车后，司机开始计时（比如说十分钟）。十分钟之内，如果又上来了一个乘客，司机会把计时器清零，重新开始等另一个十分钟（延迟了等待）。直到有这么一位乘客，从他上车开始，后续十分钟都没有新乘客上车，司机会认为确实没有人需要搭这趟车了，才会把车开走。

我们对比 `throttle` 来理解 `debounce`：在 `throttle` 的逻辑里，“第一个人说了算”，它只为第一个乘客计时，时间到了就执行回调。而 `debounce` 认为，“最后一个人说了算”，`debounce` 会为每一个新乘客设定新的定时器。

我们基于上面的理解，一起来写一个 `debounce`：

```
// fn是我们需要包装的事件回调，delay是每次推迟执行的等待时间
```

```

function debounce(fn, delay) {
  // 定时器
  let timer = null

  // 将debounce处理结果当作函数返回
  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments

    // 每次事件被触发时，都去清除之前的旧定时器
    if(timer) {
      clearTimeout(timer)
    }
    // 设立新定时器
    timer = setTimeout(function () {
      fn.apply(context, args)
    }, delay)
  }
}

// 用debounce来包装scroll的回调
const better_scroll = debounce(() => console.log('触发了滚动事件'), 1000)

document.addEventListener('scroll', better_scroll)

```

#用 Throttle 来优化 Debounce

`debounce` 的问题在于它“太有耐心了”。试想，如果用户的操作十分频繁——他每次都不等 `debounce` 设置的 `delay` 时间结束就进行下一次操作，于是每次 `debounce` 都为该用户重新生成定时器，回调函数被延迟了不计其数次。频繁的延迟会导致用户迟迟得不到响应，用户同样会产生“这个页面卡死了”的观感。

为了避免弄巧成拙，我们需要借力 `throttle` 的思想，打造一个“有底线”的 `debounce`——等你可以，但我有我的原则：`delay` 时间内，我可以为你重新生成定时器；但只要 `delay` 的时间到了，我必须要给用户一个响应。这个 `throttle` 与 `debounce` “合体”思路，已经被很多成熟的前端库应用到了它们的加强版 `throttle` 函数的实现中：

```

// fn是我们需要包装的事件回调，delay是时间间隔的阈值
function throttle(fn, delay) {
  // last为上一次触发回调的时间，timer是定时器
  let last = 0, timer = null
  // 将throttle处理结果当作函数返回

  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments
    // 记录本次触发回调的时间
    let now = +new Date()

    // 判断上次触发的时间和本次触发的时间差是否小于时间间隔的阈值
    if (now - last < delay) {
      // 如果时间间隔小于我们设定的时间间隔阈值，则为本次触发操作设立一个新的定时器
    } else {
      if(timer) {
        clearTimeout(timer)
      }
      timer = setTimeout(function () {
        fn.apply(context, args)
      }, delay)
    }
  }
}

```

```
clearTimeout(timer)
timer = setTimeout(function () {
    last = now
    fn.apply(context, args)
}, delay)
} else {
    // 如果时间间隔超出了我们设定的时间间隔阈值，那就不等了，无论如何要反馈给用户一次响应
    last = now
    fn.apply(context, args)
}
}

// 用新的throttle包装scroll的回调
const better_scroll = throttle(() => console.log('触发了滚动事件'), 1000)

document.addEventListener('scroll', better_scroll)
```

#小结

`throttle` 和 `debounce` 不仅是我们日常开发中的常用优质代码片段，更是前端面试中不可不知的高频考点。“看懂了代码”、“理解了过程”在本节都是不够的，重要的是把它写到自己的项目里去，亲自体验一把节流和防抖带来的性能提升。

#十四、性能监测篇：Performance、LightHouse 与性能 API

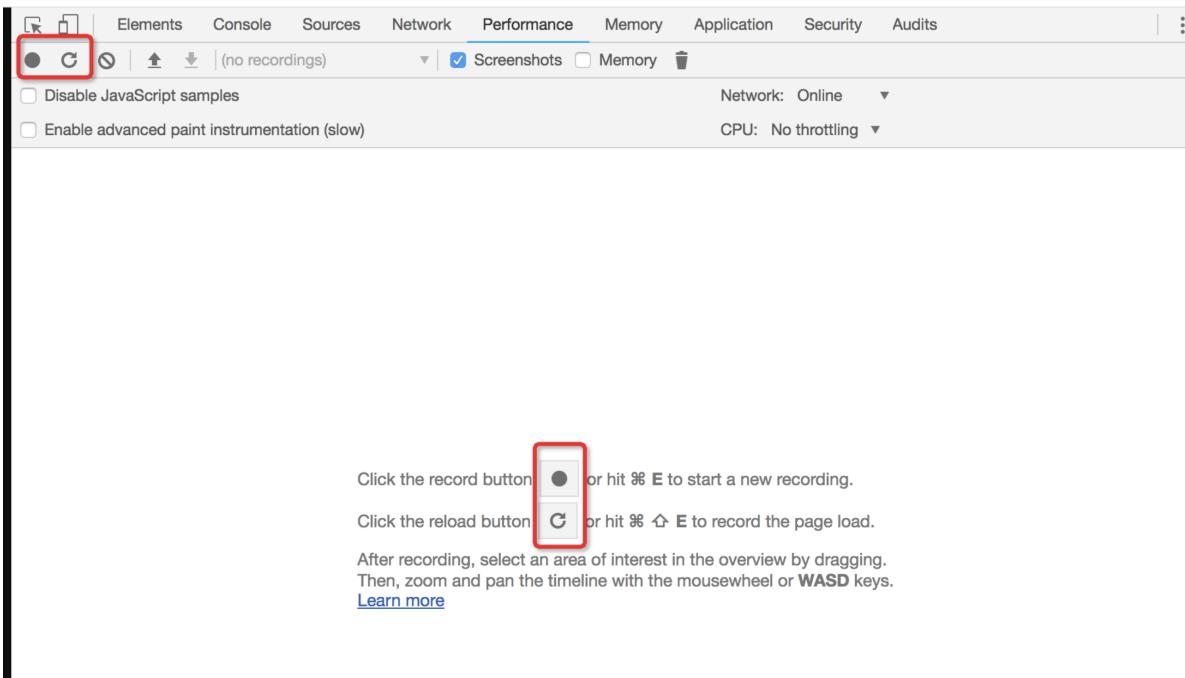
- 性能监测是前端性能优化的重要一环。监测的目的是为了确定性能瓶颈，从而有的放矢地开展具体的优化工作。
- 平时我们比较推崇的性能监测方案主要有两种：**可视化方案**、**可编程方案**。这两种方案下都有非常优秀、且触手可及的相关工具供大家选择，本节我们就一起来研究一下这些工具的用法。

#可视化监测：从 Performance 面板说起

`Performance` 是 Chrome 提供给我们的开发者工具，用于记录和分析我们的应用在运行时的所有活动。它呈现的数据具有实时性、多维度的特点，可以帮助我们很好地定位性能问题。

1. 开始记录

右键打开开发者工具，选中我们的 `Performance` 面板：



- 当我们选中图中所标示的实心圆按钮，Performance 会开始帮我们记录我们后续的交互操作；当我们选中圆箭头按钮，Performance 会将页面重新加载，计算加载过程中的性能表现。
- tips：使用 Performance 工具时，为了规避其它 Chrome 插件对页面的性能影响，我们最好在无痕模式下打开页面：

打开新的标签页(T)

⌘T

打开新的窗口(N)

⌘N

打开新的无痕窗口(I)

⇧ ⌘ N



您已进入无痕模式

现在，您便可进行私密浏览了。共用此设备的其他用户将不会看到您的活动，但您下载的内容和添加的书签仍会保存在设备上。[了解详情](#)

Chrome 不会保存以下信息：

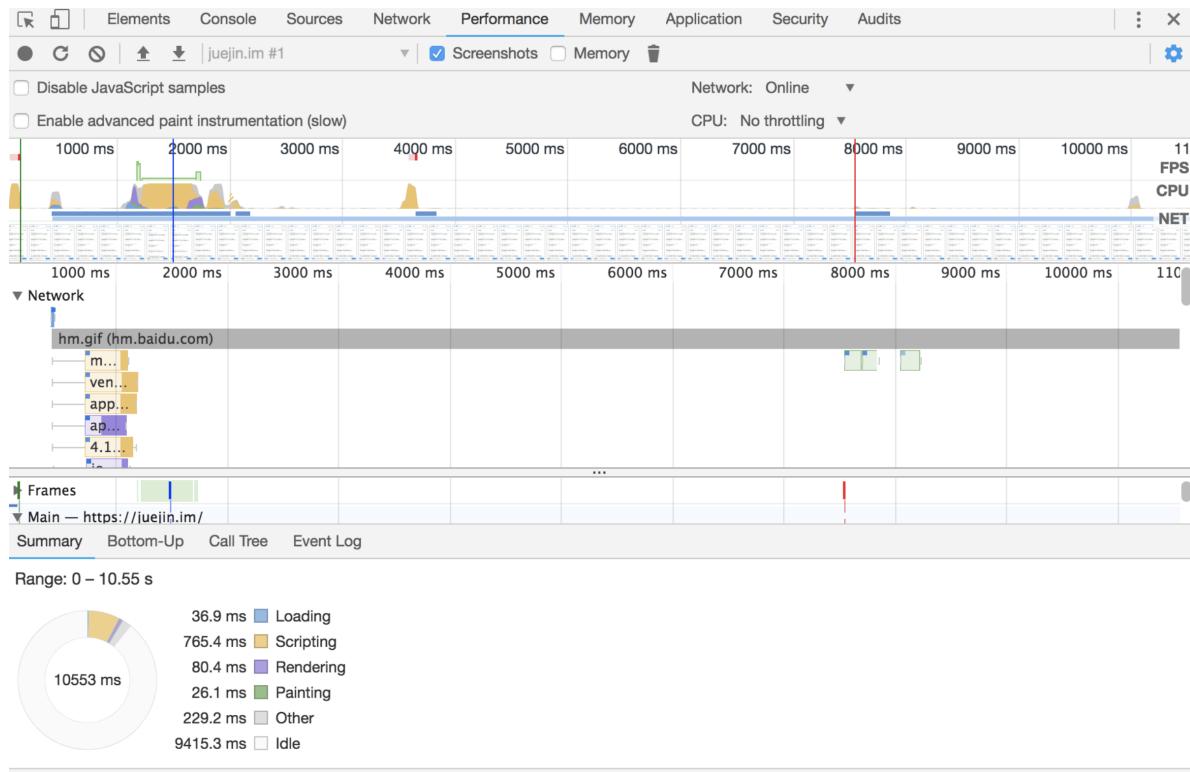
- 您的浏览记录
- Cookie 和网站数据
- 在表单中填写的信息

以下各方可能仍会看到您的活动：

- 您访问的网站
- 您的雇主或您所在的学校
- 您的互联网服务提供商

2. 简要分析

这里我打开掘金首页，选中 Performance 面板中的圆箭头，来看一下页面加载过程中的性能表现：



从上到下，依次为概述面板、详情面板。下我们先来观察一下概述面板，了解页面的基本表现：



我们看右上角的三个栏目：FPS、CPU 和 NET。

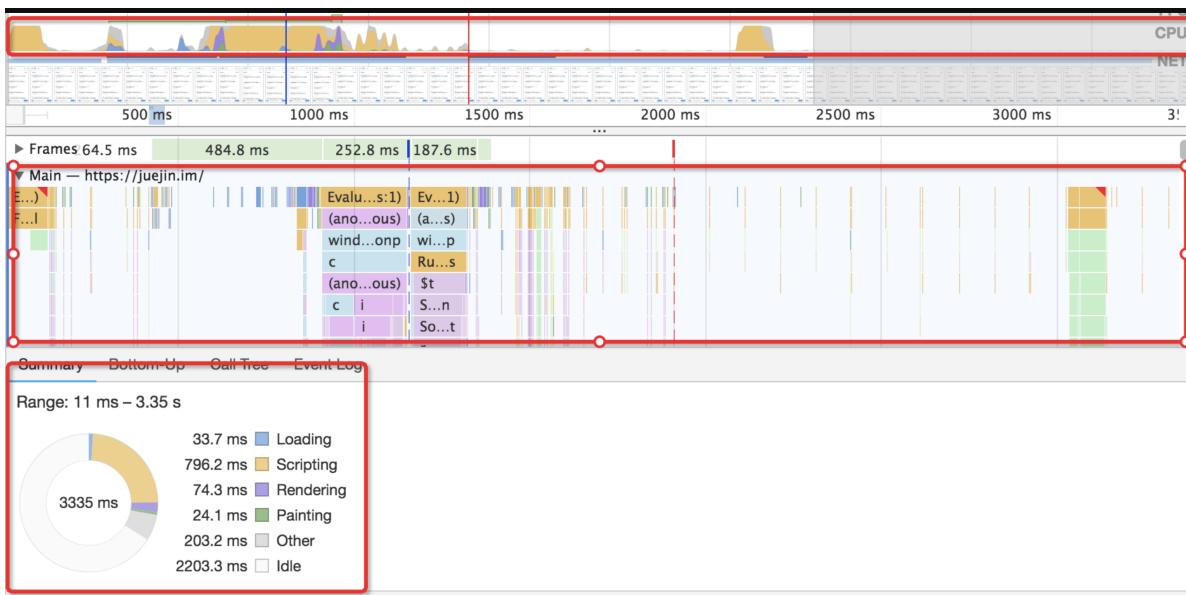
FPS：这是一个和动画性能密切相关的指标，它表示每一秒的帧数。图中绿色柱状越高表示帧率越高，体验就越流畅。若出现红色块，则代表长时间帧，很可能会出现卡顿。图中以绿色为主，偶尔出现红块，说明网页性能并不糟糕，但仍有可优化的空间。

CPU：表示CPU的使用情况，不同的颜色片段代表着消耗CPU资源的不同事件类型。这部分的图像和下文详情面板中的Summary内容有对应关系，我们可以结合这两者挖掘性能瓶颈。

NET：粗略的展示了各请求的耗时与前后顺序。这个指标一般来说帮助不大。

3. 挖掘性能瓶颈

详情面板中的内容有很多。但一般来说，我们会主要去看 Main 栏目下的火焰图和 Summary 提供给我们的饼图——这两者和概述面板中的 CPU 一栏结合，可以帮我们迅速定位性能瓶颈（如下图）。

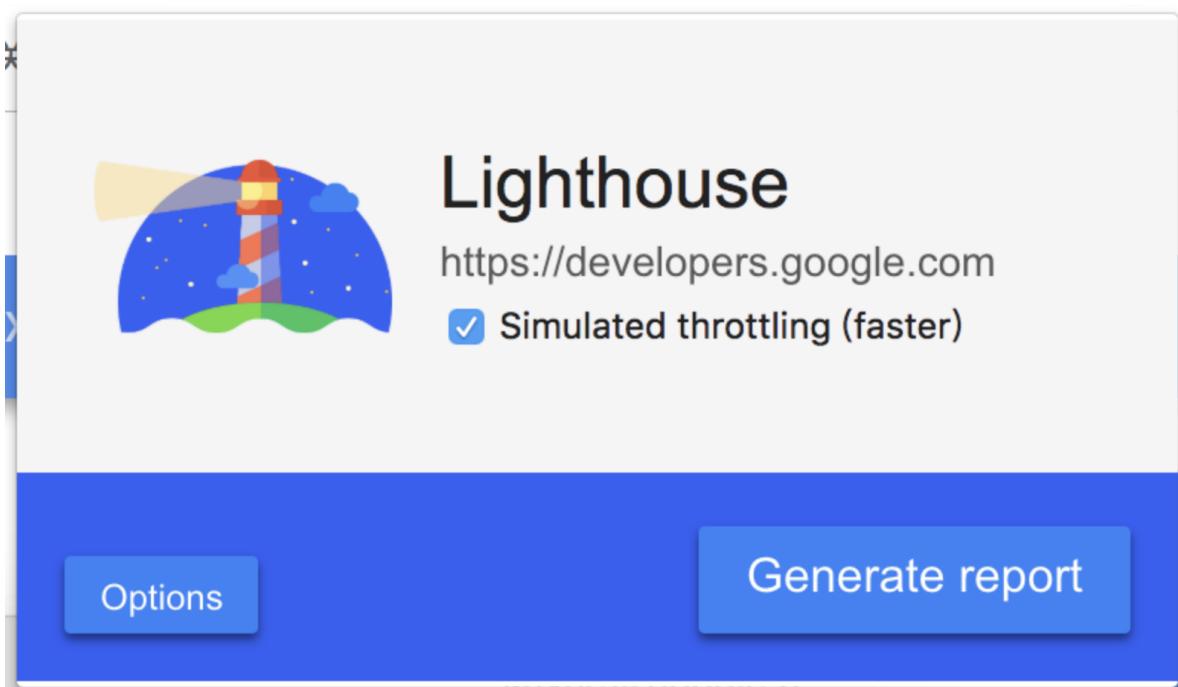


- 先看 CPU 图表和 Summary 饼图。CPU 图表中，我们可以根据颜色填充的饱满程度，确定 CPU 的忙闲，进而了解该页面的总的总任务量。而 summary 饼图则以一种直观的方式告诉了我们，哪个类型的任务最耗时（从本例来看是脚本执行过程）。这样我们在优化的时候，就可以抓到“主要矛盾”，进而有的放矢地开展后续的工作了。
- 再看 Main 提供给我们的火焰图。这个火焰图非常关键，它展示了整个运行时主进程所做的每一件事情（包括加载、脚本运行、渲染、布局、绘制等）。x 轴表示随时间的记录。每个长条就代表一个活动。更宽的条形意味着事件需要更长时间。y 轴表示调用堆栈，我们可以看到事件是相互堆叠的，上层的事件触发了下层的事件。
- CPU 图标和 Summary 图都是按照“类型”给我们提供性能信息，而 Main 火焰图则将粒度细化到了每一个函数的调用。到底是从哪个过程开始出问题、是哪个函数拖了后腿、又是哪个事件触发了这个函数，这些具体的、细致的问题都将在 Main 火焰图中得到解答。

#可视化监测：更加聪明的 LightHouse

Performance 无疑可以为我们提供很多有价值的信息，但它的展示作用大于分析作用。它要求使用者对工具本身及其所展示的信息有充分的理解，能够将晦涩的数据“翻译”成具体的性能问题。

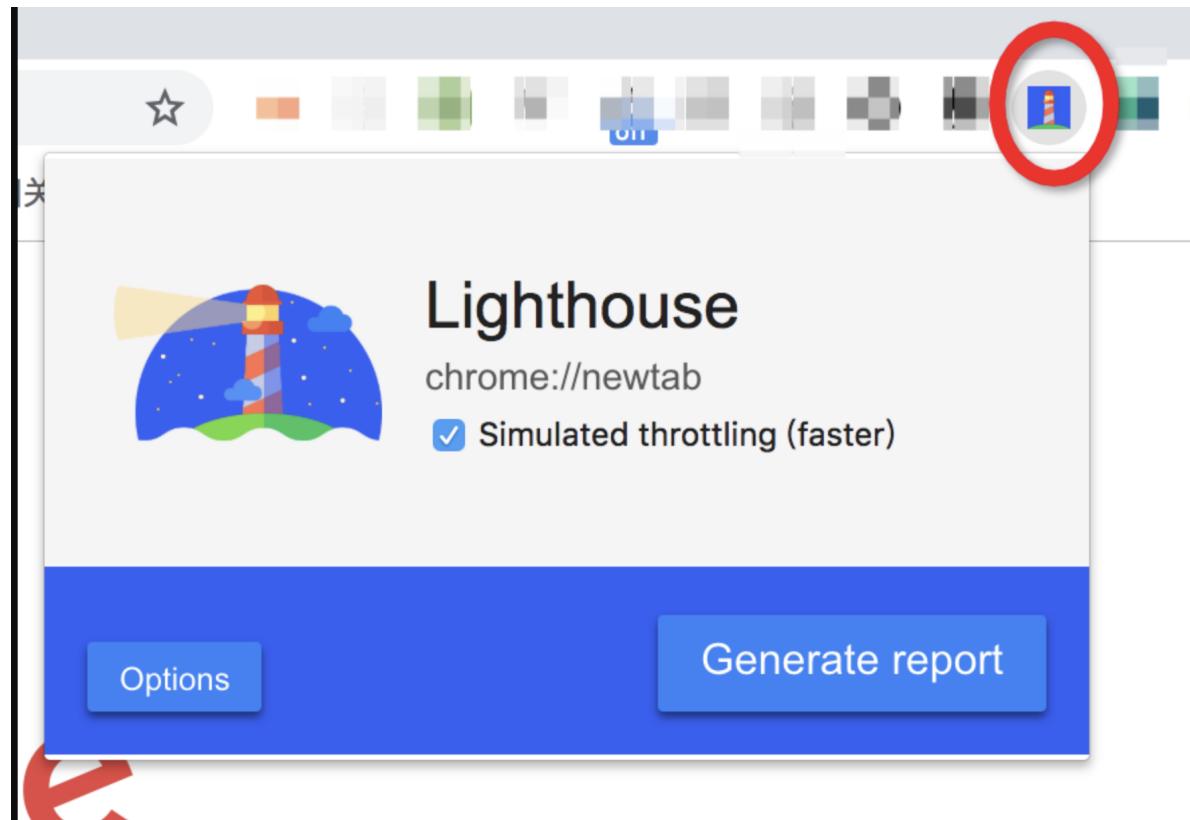
程序员们许了个愿：如果工具能帮助我们把页面的问题也分析出来就好了！上帝听到了这个愿望，于是给了我们 LightHouse：



`Lighthouse` 是一个开源的自动化工具，用于改进网络应用的质量。你可以将其作为一个 Chrome 扩展程序运行，或从命令行运行。为 `Lighthouse` 提供一个需要审查的网址，它将针对此页面运行一连串的测试，然后生成一个有关页面性能的报告。

敲黑板划重点：它生成的是一个报告！`Report`！不是干巴巴地数据，而是一个通过测试与分析呈现出来的结果（它甚至会给你你的页面跑一个分数出来）。这个东西看起来也真是太赞了，我们这就来体验一下！

首先在 `Chrome` 的应用商店里下载一个 `LightHouse`。这一步 OK 之后，我们浏览器右上角会出现一个小小的灯塔 `ICON`。打开我们需要测试的那个页面，点击这个 `ICON`，唤起如下的面板：

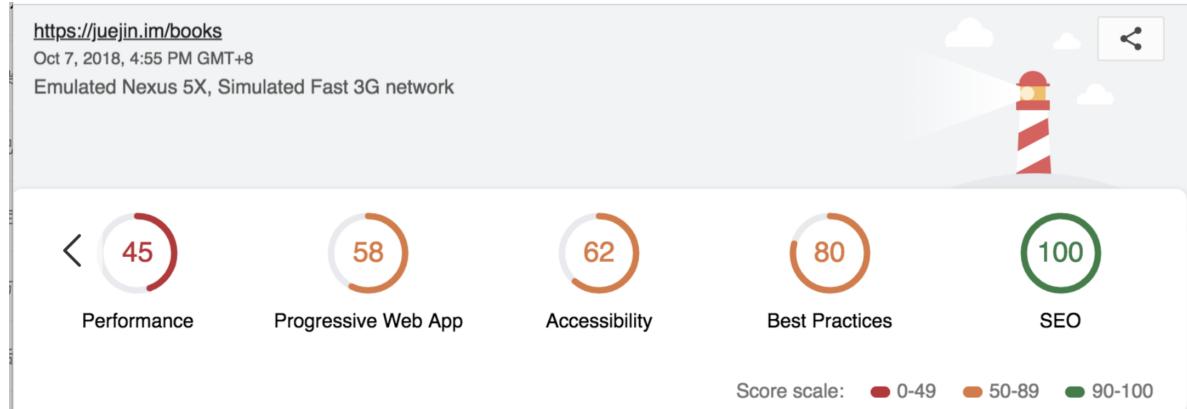


然后点击“Generate report”按钮，只需静候数秒，`Lighthouse` 就会为我们输出一个完美的性能报告。

这里我拿掘金首页“开刀”：

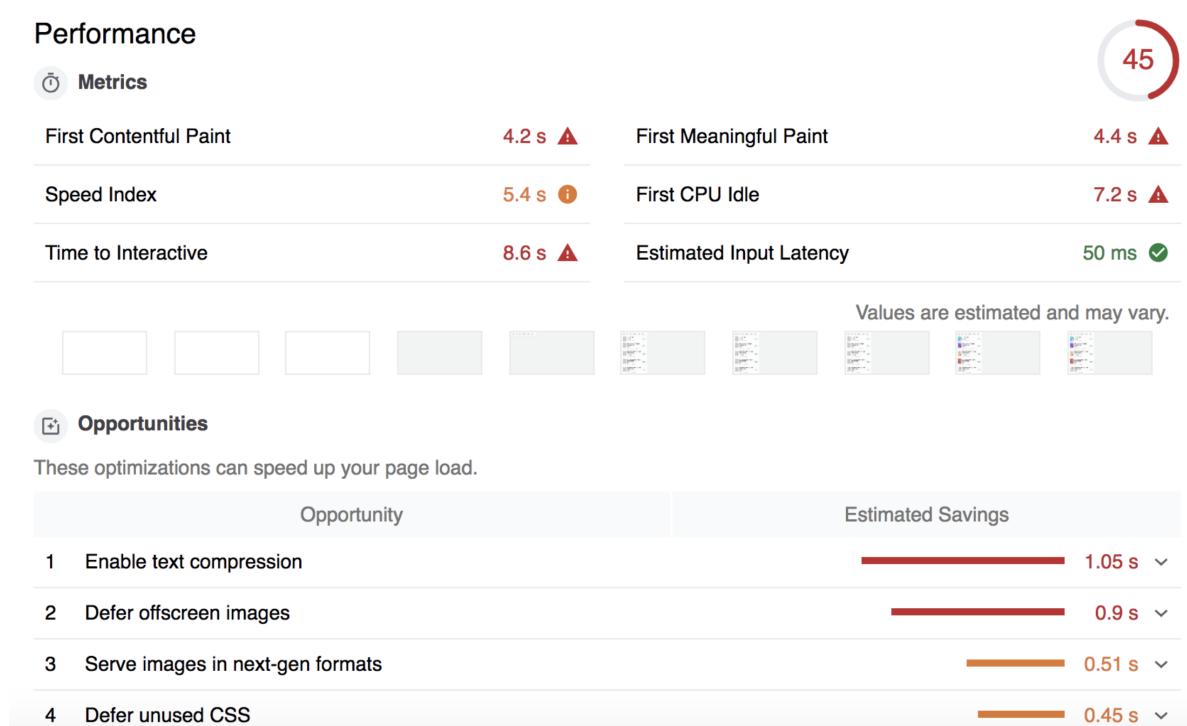
A screenshot of the Lighthouse performance report for the Zhihu homepage. The report is titled "Retrieving: CSSUsage". On the left, there is a sidebar with a list of categories: 全部, 前端, 后端, 移动开发, 区块链, 通用. Below the sidebar, there is a list of several course items with their titles, descriptions, and prices. On the right, there is a detailed breakdown of the CSS usage, showing various metrics and recommendations. The overall interface is clean and modern, with a white background and blue accents.

稍事片刻，`Report` 便输出成功了，`LightHouse` 默认会帮我们打开一个新的标签页来展示报告内容。报告内容非常丰富，首先我们看到的是整体的跑分情况：



上述分别是页面性能、PWA（渐进式 Web 应用）、可访问性（无障碍）、最佳实践、SEO 五项指标的跑分。孰强孰弱，我们一看便知。

向下拉动 `Report` 页，我们还可以看到每一个指标的细化评估：



- 在“Opportunities”中，`LightHouse` 甚至针对我们的性能问题给出了可行的建议、以及每一项优化操作预期会帮我们节省的时间。这份报告的可操作性是很强的——我们只需要对着 `LightHouse` 给出的建议，一条一条地去尝试，就可以看到自己的页面，在一秒一秒地变快。
- 除了直接下载，我们还可以通过命令行使用 `LightHouse`：

```
npm install -g lighthouse
lighthouse https://juejin.im/books
```

同样可以得到掘金的性能报告。

此外，从 `chrome 60` 开始，`DevTools` 中直接加入了基于 `LightHouse` 的 `Audits` 面板：



The screenshot shows the Chrome DevTools interface with the 'Audits' tab selected. The main area displays a summary of a new audit, including a progress bar, a preview of the audit results, and a summary table. Below the summary is a detailed list of findings, each with a preview icon, title, and severity level.



Audits

Identify and fix common problems that affect your site's performance, accessibility, and user experience. [Learn more](#)

Device

Mobile

Desktop

✓ Audits

Performance

Progressive Web App

Best practices

Accessibility

SEO

Throttling

- Simulated Fast 3G, 4x CPU Slowdown

○ Applied Fast 3G, 4x CPU Slowdown

No throttling

Clear storage

`LightHouse` 因此变得更加触手可及了，这一操作也足以证明 `Chrome` 团队对 `LightHouse` 的推崇。

#可编程的性能上报方案：W3C 性能 API

- W3C 规范为我们提供了 `Performance` 相关的接口。它允许我们获取到用户访问一个页面的每个阶段的精确时间，从而对性能进行分析。我们可以将其理解为 `Performance` 面板的进一步细化与可编程化。
 - 当下的前端世界里，数据可视化的概念已经被炒得非常热了，`Performance` 面板就是数据可视化的典范。那么为什么要把已经可视化的数据再掏出来处理一遍呢？这是因为，需要这些数据的人不止我们前端——很多情况下，后端也需要我们提供性能信息的上报。此外，`Performance` 提供的可视化结果并不一定能够满足我们实际的业务需求，只有拿到了真实的数据，我们才可以对它进行二次处理，去做一个更加深层次的可视化。

在这种需求背景下，我们就不得不祭出 Performance API 了。

1. 访问 performance 对象

`performance` 是一个全局对象。我们在控制台里输入 `window.performance`，就可一窥其全貌：

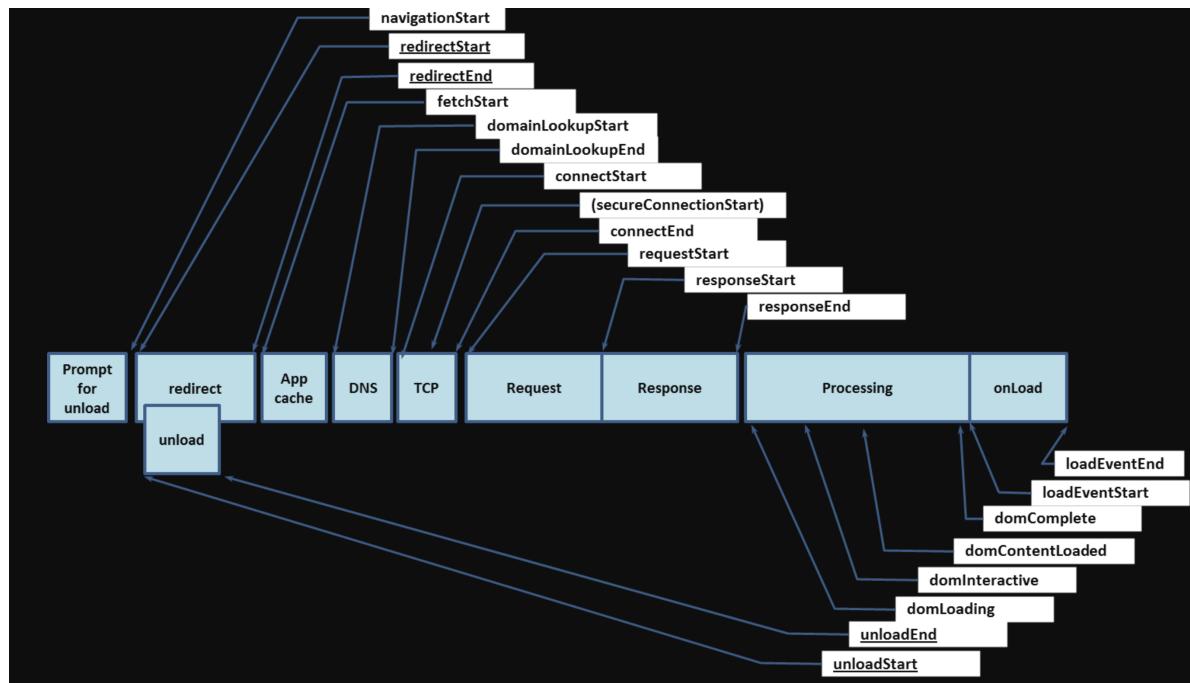
```
> window.performance
< Performance {timeOrigin: 1538894704319.074, onresourcetimingbufferfull: null, memory: MemoryInfo, navigation: PerformanceNavigation, timing: PerformanceTiming}
  > memory: MemoryInfo {totalJSHeapSize: 24417416, usedJSHeapSize: 19742944, jsHeapSizeLimit: 2217857988}
  > navigation: PerformanceNavigation {type: 0, redirectCount: 0}
    onresourcetimingbufferfull: null
    timeOrigin: 1538894704319.074
  > timing: PerformanceTiming {navigationStart: 1538894704660, unloadEventStart: 0, unloadEventEnd: 0, redirectStart: 0, redirectEnd: 0}
  > __proto__: Performance
```

2. 关键时间节点

在 `performance` 的 `timing` 属性中，我们可以查看到如下的时间戳：

```
▼ timing: PerformanceTiming
  connectEnd: 1538904461863
  connectStart: 1538904461863
  domComplete: 1538904465480
  domContentLoadedEventEnd: 1538904464780
  domContentLoadedEventStart: 1538904464730
  domInteractive: 1538904464730
  domLoading: 1538904462552
  domainLookupEnd: 1538904461863
  domainLookupStart: 1538904461863
  fetchStart: 1538904461863
  loadEventEnd: 1538904465481
  loadEventStart: 1538904465480
  navigationStart: 1538904461844
  redirectEnd: 0
  redirectStart: 0
  requestStart: 1538904461895
  responseEnd: 1538904462626
  responseStart: 1538904462545
  secureConnectionStart: 0
  unloadEventEnd: 0
  unloadEventStart: 0
```

这些时间戳与页面整个加载流程中的关键时间节点有着一一对应的关系：



通过求两个时间点之间的差值，我们可以得出某个过程花费的时间，举个🌰：

```
const timing = window.performance.timing
// DNS查询耗时
timing.domainLookupEnd - timing.domainLookupStart

// TCP连接耗时
timing.connectEnd - timing.connectStart

// 内容加载耗时
timing.responseEnd - timing.requestStart

...
```

除了这些常见的耗时情况，我们更应该去关注一些**关键性能指标**: `firstbyte`、`fpt`、`tti`、`ready` 和 `load` 时间。这些指标数据与真实的用户体验息息相关，是我们日常业务性能监测中不可或缺的一部分：

```
// firstbyte: 首包时间
timing.responseStart - timing.domainLookupStart

// fpt: First Paint Time, 首次渲染时间 / 白屏时间
timing.responseEnd - timing.fetchStart

// tti: Time to Interact, 首次可交互时间
timing.domInteractive - timing.fetchStart

// ready: HTML 加载完成时间, 即 DOM 就位的时间
timing.domContentLoaded - timing.fetchStart

// load: 页面完全加载时间
timing.loadEventStart - timing.fetchStart
```

以上这些通过 `Performance API` 获取到的时间信息都具有较高的准确度。我们可以对此进行一番格式化之后上报给服务端，也可以基于此去制作相应的统计图表，从而实现更加精准、更加个性化的性能耗时统计。

此外，通过访问 `performance` 的 `memory` 属性，我们还可以获取到内存占用相关的数据；通过对 `performance` 的其它属性方法的灵活运用，我们还可以把它耦合进业务里，实现更加多样化的性能监测需求——灵活，是可编程化方案最大的优点。

高频手写题目

面试高频手写题目

#1 实现防抖函数 (debounce)

防抖函数原理：在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时

手写简化版：

```
// func是用户传入需要防抖的函数
// wait是等待时间
```

```

const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设定过定时器了就清空上一次的定时器
  // 开始一个新的定时器，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}

```

适用场景：

按钮提交场景：防止多次提交按钮，只执行最后提交的一次 服务端验证场景：表单验证需要服务端配合，只执行一段连续的输入事件的最后一次，还有搜索联想词功能类似

#2 实现节流函数 (throttle)

节流函数原理:规定在一个单位时间内，只能触发一次函数。如果这个单位时间内触发多次函数，只有一次生效

手写简版

```

// func是用户传入需要防抖的函数
// wait是等待时间
const throttle = (func, wait = 50) => {
  // 上一次执行该函数的时间
  let lastTime = 0
  return function(...args) {
    // 当前时间
    let now = +new Date()
    // 将当前时间和上一次执行函数时间对比
    // 如果差值大于设置的等待时间就执行函数
    if (now - lastTime > wait) {
      lastTime = now
      func.apply(this, args)
    }
  }
}

setInterval(
  throttle(() => {
    console.log(1)
  }, 500),
  1
)

```

适用场景：

- 拖拽场景：固定时间内只执行一次，防止超高频次触发位置变动
- 缩放场景：监控浏览器 resize
- 动画场景：避免短时间内多次触发动画引起性能问题

#3 深克隆 (deepclone)

简单版：

```
const newObj = JSON.parse(JSON.stringify(oldObj));
```

局限性：

- 他无法实现对函数、RegExp等特殊对象的克隆
- 会抛弃对象的constructor,所有的构造函数会指向Object
- 对象有循环引用,会报错

面试够用版

```
function deepCopy(obj){  
    //判断是否是简单数据类型,  
    if(typeof obj == "object"){  
        //复杂数据类型  
        var result = obj.constructor == Array ? [] : {};  
        for(let i in obj){  
            result[i] = typeof obj[i] == "object" ? deepCopy(obj[i]) : obj[i];  
        }  
    }else {  
        //简单数据类型 直接 == 赋值  
        var result = obj;  
    }  
    return result;  
}
```

#4 实现Event(event bus)

event bus既是node中各个模块的基石，又是前端组件通信的依赖手段之一，同时涉及了订阅-发布设计模式，是非常重要的基础

简单版：

```
class EventEmmitter {  
    constructor() {  
        this._events = this._events || new Map(); // 储存事件/回调键值对  
        this._maxListeners = this._maxListeners || 10; // 设立监听上限  
    }  
  
    // 触发名为type的事件  
    EventEmmitter.prototype.emit = function(type, ...args) {  
        let handler;  
        // 从储存事件键值对的this._events中获取对应事件回调函数  
        handler = this._events.get(type);  
        if (args.length > 0) {  
            handler.apply(this, args);  
        } else {  
            handler.call(this);  
        }  
    }  
}
```

```

    }
    return true;
};

// 监听名为type的事件
EventEmitter.prototype.addListener = function(type, fn) {
    // 将type事件以及对应的fn函数放入this._events中储存
    if (!this._events.get(type)) {
        this._events.set(type, fn);
    }
};

```

面试版：

```

class EventEmitter {
    constructor() {
        this._events = this._events || new Map(); // 储存事件/回调键值对
        this._maxListeners = this._maxListeners || 10; // 设立监听上限
    }
}

// 触发名为type的事件
EventEmitter.prototype.emit = function(type, ...args) {
    let handler;
    // 从储存事件键值对的this._events中获取对应事件回调函数
    handler = this._events.get(type);
    if (args.length > 0) {
        handler.apply(this, args);
    } else {
        handler.call(this);
    }
    return true;
};

// 监听名为type的事件
EventEmitter.prototype.addListener = function(type, fn) {
    // 将type事件以及对应的fn函数放入this._events中储存
    if (!this._events.get(type)) {
        this._events.set(type, fn);
    }
};

// 触发名为type的事件
EventEmitter.prototype.emit = function(type, ...args) {
    let handler;
    handler = this._events.get(type);
    if (Array.isArray(handler)) {
        // 如果是一个数组说明有多个监听者,需要依次此触发里面的函数
        for (let i = 0; i < handler.length; i++) {
            if (args.length > 0) {
                handler[i].apply(this, args);
            } else {
                handler[i].call(this);
            }
        }
    } else {
        // 单个函数的情况我们直接触发即可
    }
};

```

```

        if (args.length > 0) {
            handler.apply(this, args);
        } else {
            handler.call(this);
        }
    }

    return true;
};

// 监听名为type的事件
EventEmitter.prototype.addListener = function(type, fn) {
    const handler = this._events.get(type); // 获取对应事件名称的函数清单
    if (!handler) {
        this._events.set(type, fn);
    } else if (handler && typeof handler === "function") {
        // 如果handler是函数说明只有一个监听者
        this._events.set(type, [handler, fn]); // 多个监听者我们需要用数组储存
    } else {
        handler.push(fn); // 已经有多个监听者,那么直接往数组里push函数即可
    }
};

EventEmitter.prototype.removeListener = function(type, fn) {
    const handler = this._events.get(type); // 获取对应事件名称的函数清单

    // 如果是函数,说明只被监听了一次
    if (handler && typeof handler === "function") {
        this._events.delete(type, fn);
    } else {
        let postion;
        // 如果handler是数组,说明被监听多次要找到对应的函数
        for (let i = 0; i < handler.length; i++) {
            if (handler[i] === fn) {
                postion = i;
            } else {
                postion = -1;
            }
        }
        // 如果找到匹配的函数,从数组中清除
        if (postion !== -1) {
            // 找到数组对应的位置,直接清除此回调
            handler.splice(postion, 1);
            // 如果清除后只有一个函数,那么取消数组,以函数形式保存
            if (handler.length === 1) {
                this._events.set(type, handler[0]);
            }
        } else {
            return this;
        }
    }
};

```

#5 实现instanceOf

核心要点：原型链的向上查找

```
function myInstanceof(left, right) {  
    let proto = Object.getPrototypeOf(left);  
    while(true) {  
        if(proto == null) return false;  
        if(proto == right.prototype) return true;  
        proto = Object.getPrototypeOf(proto);  
    }  
}
```

#6 模拟new

new操作符做了这些事：

- 创建一个全新的对象，这个对象的 __proto__ 要指向构造函数的原型对象
- 执行构造函数
- 返回值为object类型则作为new方法的返回值返回，否则返回上述全新对象

```
function myNew(fn, ...args) {  
    let instance = Object.create(fn.prototype);  
    let res = fn.apply(instance, args);  
    return typeof res === 'object' ? res: instance;  
}
```

#7 实现一个call

call做了什么：

- 将函数设为对象的属性
- 执行&删除这个函数
- 指定 this 到函数并传入给定参数执行函数
- 如果不传入参数，默认指向为 window

```
// 模拟 bar.myCall(null);  
//实现一个call方法:  
Function.prototype.myCall = function(context) {  
    //此处没有考虑context非object情况  
    context.fn = this;  
    let args = [];  
    for (let i = 1, len = arguments.length; i < len; i++) {  
        args.push(arguments[i]);  
    }  
    context.fn(...args);  
    let result = context.fn(...args);  
    delete context.fn;  
    return result;  
};
```

#8 实现apply方法

思路: 利用 `this` 的上下文特性。

```
//实现apply只要把下一行中的...args换成args即可
Function.prototype.myCall = function(context = window, ...args) {
    let func = this;
    let fn = Symbol("fn");
    context[fn] = func;

    let res = context[fn](...args); //重点代码, 利用this指向, 相当于
context.caller(...args)

    delete context[fn];
    return res;
}
```

#9 实现bind

bind 的实现对比其他两个函数略微地复杂了一点, 因为 bind 需要返回一个函数, 需要判断一些边界问题, 以下是 bind 的实现

- bind 返回了一个函数, 对于函数来说有两种方式调用, 一种是直接调用, 一种是通过 `new` 的方式, 我们先来说直接调用的方式
- 对于直接调用来说, 这里选择了 `apply` 的方式实现, 但是对于参数需要注意以下情况: 因为 bind 可以实现类似这样的代码 `f.bind(obj, 1)(2)`, 所以我们需要将两边的参数拼接起来, 于是就有了这样的实现 `args.concat(...arguments)`
- 最后来说通过 `new` 的方式, 在之前的章节中我们学习过如何判断 `this`, 对于 `new` 的情况来说, 不会被任何方式改变 `this`, 所以对于这种情况我们需要忽略传入的 `this`

```
Function.prototype.myBind = function (context) {
    if (typeof this !== 'function') {
        throw new TypeError('Error')
    }
    const _this = this
    const args = [...arguments].slice(1)
    // 返回一个函数
    return function F() {
        // 因为返回了一个函数, 我们可以 new F(), 所以需要判断
        if (this instanceof F) {
            return new _this(...args, ...arguments)
        }
        return _this.apply(context, args.concat(...arguments))
    }
}
```

#10 模拟Object.create

`Object.create()`方法创建一个新对象, 使用现有的对象来提供新创建的对象的`proto`

```
// 模拟 Object.create

function create(proto) {
    function F() {}
    F.prototype = proto;

    return new F();
}
```

#11 实现类的继承-简版

类的继承在几年前是重点内容，有n种继承方式各有优劣，es6普及后越来越不重要，那么多种写法有点『回字有四样写法』的意思，如果还想深入理解的去看红宝书即可，我们目前只实现一种最理想的继承方式。

```
function Parent(name) {
    this.parent = name
}

Parent.prototype.say = function() {
    console.log(`>${this.parent}: 你打篮球的样子像kunkun`)
}

function child(name, parent) {
    // 将父类的构造函数绑定在子类上
    Parent.call(this, parent)
    this.child = name
}

/**
1. 这一步不用child.prototype = Parent.prototype的原因是怕共享内存，修改父类原型对象就会影响子类
2. 不用child.prototype = new Parent()的原因是会调用2次父类的构造方法（另一次是call），会存在一份多余的父类实例属性
3. Object.create是创建了父类原型的副本，与父类原型完全隔离
*/
Child.prototype = Object.create(Parent.prototype);
Child.prototype.say = function() {
    console.log(`>${this.parent}好，我是练习时长两年半的${this.child}`);
}

// 注意记得把子类的构造指向子类本身
Child.prototype.constructor = Child;

var parent = new Parent('father');
parent.say() // father: 你打篮球的样子像kunkun

var child = new Child('cxk', 'father');
child.say() // father好，我是练习时长两年半的cxk
```

#12 ES5实现继承的那些事-详细

第一种方式是借助call实现继承

```

function Parent1(){
    this.name = 'parent1';
}
function Child1(){
    Parent1.call(this);
    this.type = 'child1'
}
console.log(new Child1());

```

这样写的时候子类虽然能够拿到父类的属性值，但是问题是父类中一旦存在方法那么子类无法继承。那么引出下面的方法

第二种方式借助原型链实现继承：

```

function Parent2() {
    this.name = 'parent2';
    this.play = [1, 2, 3]
}
function Child2() {
    this.type = 'child2';
}
Child2.prototype = new Parent2();

console.log(new Child2());

```

看似没有问题，父类的方法和属性都能够访问，但实际上有一个潜在的不足。举个例子：

```

var s1 = new Child2();
var s2 = new Child2();
s1.play.push(4);
console.log(s1.play, s2.play); // [1,2,3,4] [1,2,3,4]

```

明明我只改变了s1的play属性，为什么s2也跟着变了呢？很简单，因为两个实例使用的是同一个原型对象

第三种方式：将前两种组合：

```

function Parent3 () {
    this.name = 'parent3';
    this.play = [1, 2, 3];
}
function Child3(){
    Parent3.call(this);
    this.type = 'child3';
}
Child3.prototype = new Parent3();
var s3 = new Child3();
var s4 = new Child3();
s3.play.push(4);
console.log(s3.play, s4.play); // [1,2,3,4] [1,2,3]

```

之前的问题都得以解决。但是这里又徒增了一个新问题，那就是Parent3的构造函数会多执行了一次 (`Child3.prototype = new Parent3();`)。这是我们不愿看到的。那么如何解决这个问题？

第四种方式: 组合继承的优化1

```
function Parent4 () {
    this.name = 'parent4';
    this.play = [1, 2, 3];
}
function Child4() {
    Parent4.call(this);
    this.type = 'child4';
}
Child4.prototype = Parent4.prototype;
```

这里让将父类原型对象直接给到子类，父类构造函数只执行一次，而且父类属性和方法均能访问，但是我们来测试一下

```
var s3 = new Child4();
var s4 = new Child4();
console.log(s3)
```

子类实例的构造函数是Parent4，显然这是不对的，应该是Child4。

第五种方式(最推荐使用): 优化2

```
function Parent5 () {
    this.name = 'parent5';
    this.play = [1, 2, 3];
}
function Child5() {
    Parent5.call(this);
    this.type = 'child5';
}
Child5.prototype = Object.create(Parent5.prototype);
Child5.prototype.constructor = Child5;
```

这是最推荐的一种方式，接近完美的继承。

#13 实现一个JSON.stringify

```
JSON.stringify(value[, replacer [, space]]):
```

- Boolean | Number | String 类型会自动转换成对应的原始值。
- undefined、任意函数以及 symbol，会被忽略（出现在非数组对象的属性值中时），或者被转换成 null（出现在数组中时）。
- 不可枚举的属性会被忽略如果一个对象的属性值通过某种间接的方式指向该对象本身，即循环引用，属性也会被忽略
- 如果一个对象的属性值通过某种间接的方式指向该对象本身，即循环引用，属性也会被忽略

```
function jsonStringify(obj) {
    let type = typeof obj;
    if (type !== "object") {
        if (/string|undefined|function/.test(type)) {
            obj = '"' + obj + '"';
        }
    }
}
```

```

    }
    return String(obj);
} else {
    let json = []
    let arr = Array.isArray(obj)
    for (let k in obj) {
        let v = obj[k];
        let type = typeof v;
        if (/string|undefined|function/.test(type)) {
            v = '"' + v + '"';
        } else if (type === "object") {
            v = jsonStringify(v);
        }
        json.push(arr ? "" : '"' + k + '":') + string(v));
    }
    return (arr ? "[" : "{}") + string(json) + (arr ? "]" : "}")
}
jsonStringify({x : 5}) // {"x":5}
jsonStringify([1, "false", false]) // [1,"false",false]
jsonStringify({b: undefined}) // {"b":"undefined"}"

```

#14 实现一个JSON.parse

```
JSON.parse(text[, reviver])
```

用来解析JSON字符串，构造由字符串描述的JavaScript值或对象。提供可选的reviver函数用以在返回之前对所得到的对象执行变换(操作)

第一种：直接调用 eval

```

function jsonParse(opt) {
    return eval('(' + opt + ')');
}
jsonParse(jsonStringify({x : 5}))
// Object { x: 5}
jsonParse(jsonStringify([1, "false", false]))
// [1, "false", falsr]
jsonParse(jsonStringify({b: undefined}))
// Object { b: "undefined"}"

```

避免在不必要的的情况下使用 eval，eval() 是一个危险的函数，他执行的代码拥有着执行者的权利。如果你用 eval() 运行的字符串代码被恶意方（不怀好意的人）操控修改，您最终可能会在您的网页/扩展程序的权限下，在用户计算机上运行恶意代码。它会执行JS代码，有XSS漏洞。

如果你只想记这个方法，就得对参数json做校验。

```

var rx_one = /^[{}]/,:{ }\s]*$/;
var rx_two = /\\"/?:[ "\\\bfnrt"]|u[0-9a-fA-F]{4})/g;
var rx_three = /["\\\"\\n\\r]*|[true|false|null|-?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g;
var rx_four = /(?:^|:|,) (?:\s*\D+)/g;
if (
    rx_one.test(

```

```

        json
            .replace(rx_two, "@")
            .replace(rx_three, "]")
            .replace(rx_four, "")
    )
)
{
    var obj = eval("(" + json + ")");
}

```

第二种: Function

核心: Function与eval有相同的字符串参数特性

```
var func = new Function(arg1, arg2, ..., functionBody);
```

在转换JSON的实际应用中, 只需要这么做

```

var jsonStr = '{ "age": 20, "name": "jack" }'
var json = (new Function('return ' + jsonStr))();

```

`eval` 与 `Function` 都有着动态编译js代码的作用, 但是在实际的编程中并不推荐使用

#15 Promise的简单实现

```

// 使用
var promise = new Promise((resolve, reject) => {
    if (操作成功) {
        resolve(value)
    } else {
        reject(error)
    }
})
promise.then(function (value) {
    // success
}, function (value) {
    // failure
})
function myPromise(constructor) {
    let self = this;
    self.status = "pending" // 定义状态改变前的初始状态
    self.value = undefined; // 定义状态为resolved的时候的状态
    self.reason = undefined; // 定义状态为rejected的时候的状态
    function resolve(value) {
        if(self.status === "pending") {
            self.value = value;
            self.status = "resolved";
        }
    }
    function reject(reason) {
        if(self.status === "pending") {
            self.reason = reason;
            self.status = "rejected";
        }
    }
}

```

```

// 捕获构造异常
try {
    constructor(resolve,reject);
} catch(e) {
    reject(e);
}
}

// 添加 then 方法
myPromise.prototype.then = function(onFullfilled,onRejected) {
    let self = this;
    switch(self.status) {
        case "resolved":
            onFullfilled(self.value);
            break;
        case "rejected":
            onRejected(self.reason);
            break;
        default:
    }
}

var p = new myPromise(function(resolve,reject) {
    resolve(1)
});
p.then(function(x) {
    console.log(x) // 1
})

```

#16 解析 URL Params 为对象

```

let url = 'http://www.domain.com/?user=anonymous&id=123&id=456&city=%E5%8C%97%E4%BA%AC&enabled';
parseParam(url)
/* 结果
{ user: 'anonymous',
  id: [ 123, 456 ], // 重复出现的 key 要组装成数组，能被转成数字的就转成数字类型
  city: '北京', // 中文需解码
  enabled: true, // 未指定值得 key 约定为 true
}
*/
function parseParam(url) {
    const paramsStr = /.+?\?(.+)\$/ .exec(url)[1]; // 将 ? 后面的字符串取出来
    const paramsArr = paramsStr.split('&'); // 将字符串以 & 分割后存到数组中
    let paramsObj = {};
    // 将 params 存到对象中
    paramsArr.forEach(param => {
        if (/=/ .test(param)) { // 处理有 value 的参数
            let [key, val] = param.split('='); // 分割 key 和 value
            val = decodeURIComponent(val); // 解码
            val = /\d+/.test(val) ? parseFloat(val) : val; // 判断是否转为数字

            if (paramsObj.hasOwnProperty(key)) { // 如果对象有 key，则添加一个值
                paramsObj[key] = [] .concat(paramsObj[key], val);
            } else { // 如果对象没有这个 key，创建 key 并设置值
                paramsObj[key] = val;
            }
        }
    })
}

```

```

        }
    } else { // 处理没有 value 的参数
        paramsObj[param] = true;
    }
}

return paramsObj;
}

```

#17 模板引擎实现

```

let template = '我是{{name}}, 年龄{{age}}, 性别{{sex}}';
let data = {
    name: '姓名',
    age: 18
}
render(template, data); // 我是姓名, 年龄18, 性别undefined
function render(template, data) {
    const reg = /\{\{(\w+)\}\}/; // 模板字符串正则
    if (reg.test(template)) { // 判断模板里是否有模板字符串
        const name = reg.exec(template)[1]; // 查找当前模板里第一个模板字符串的字段
        template = template.replace(reg, data[name]); // 将第一个模板字符串渲染
        return render(template, data); // 递归的渲染并返回渲染后的结构
    }
    return template; // 如果模板没有模板字符串直接返回
}

```

#18 转化为驼峰命名

```

var s1 = "get-element-by-id"

// 转化为 getElementById

var f = function(s) {
    return s.replace(/-\w/g, function(x) {
        return x.slice(1).toUpperCase();
    })
}

```

#19 查找字符串中出现最多的字符和个数

例: abbccddddd -> 字符最多的是d, 出现了5次

```

let str = "abcabcbcabccccc";
let num = 0;
let char = '';

// 使其按照一定的次序排列
str = str.split('').sort().join('');

```

```
// "aaabbbbccccccc"

// 定义正则表达式
let re = /(\w)\1+/g;
str.replace(re, ($0,$1) => {
  if(num < $0.length){
    num = $0.length;
    char = $1;
  }
});
console.log(`字符最多的是${char}, 出现了${num}次`);
```

#20 字符串查找

请使用最基本的遍历来实现判断字符串 a 是否被包含在字符串 b 中，并返回第一次出现的位置（找不到返回 -1）。

```
a='34';b='1234567'; // 返回 2
a='35';b='1234567'; // 返回 -1
a='355';b='12354355'; // 返回 5
isContain(a,b);
function isContain(a, b) {
  for (let i in b) {
    if (a[0] === b[i]) {
      let tmp = true;
      for (let j in a) {
        if (a[j] !== b[~i + ~j]) {
          tmp = false;
        }
      }
      if (tmp) {
        return i;
      }
    }
  }
  return -1;
}
```

#21 实现千位分隔符

```
// 保留三位小数
parseToMoney(1234.56); // return '1,234.56'
parseToMoney(123456789); // return '123,456,789'
parseToMoney(1087654.321); // return '1,087,654.321'
function parseToMoney(num) {
  num = parseFloat(num.toFixed(3));
  let [integer, decimal] = String.prototype.split.call(num, '.');
  integer = integer.replace(/\d(?=(\d{3})+$)/g, '$&,');
  return integer + '.' + (decimal ? decimal : '');
}
```

#22 判断是否是电话号码

```
function isPhone(tel) {  
    var regex = /^1[34578]\d{9}$/;  
    return regex.test(tel);  
}
```

#23 验证是否是邮箱

```
function isEmail(email) {  
    var regex = /^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+\.(a-zA-Z0-9_-)+$/;  
    return regex.test(email);  
}
```

#24 验证是否是身份证

```
function isCardNo(number) {  
    var regex = /(^\d{15}$)|(^\d{18}$)|(^\d{17}(\d|x|x)$)/;  
    return regex.test(number);  
}
```

#25 用ES5实现数组的map方法

- 回调函数的参数有哪些，返回值如何处理
- 不修改原来的数组

```
Array.prototype.MyMap = function(fn, context){  
    var arr = Array.prototype.slice.call(this); //由于是ES5所以就不用...展开符了  
    var mappedArr = [];  
    for (var i = 0; i < arr.length; i++) {  
        mappedArr.push(fn.call(context, arr[i], i, this));  
    }  
    return mappedArr;  
}
```

#26 用ES5实现数组的reduce方法

- 初始值不传怎么处理
- 回调函数的参数有哪些，返回值如何处理。

```

Array.prototype.myReduce = function(fn, initialValue) {
  var arr = Array.prototype.slice.call(this);
  var res, startIndex;
  res = initialValue ? initialValue : arr[0];
  startIndex = initialValue ? 0 : 1;
  for(var i = startIndex; i < arr.length; i++) {
    res = fn.call(null, res, arr[i], i, this);
  }
  return res;
}

```

- 对于普通函数，绑定this指向
- 对于构造函数，要保证原函数的原型对象上的属性不能丢失

```

Function.prototype.bind = function(context, ...args) {
  let self = this; //谨记this表示调用bind的函数
  let fBound = function() {
    //this instanceof fBound为true表示构造函数的情况。如new func.bind(obj)
    return self.apply(this instanceof fBound ? this : context || window,
      args.concat(Array.prototype.slice.call(arguments)));
  }
  fBound.prototype = Object.create(this.prototype); //保证原函数的原型对象上的属性不
  //丢失
  return fBound;
}

```

大家平时说的手写bind，其实就这么简单

#27 实现单例模式

核心要点：用闭包和Proxy属性拦截

```

function proxy(func) {
  let instance;
  let handler = {
    constructor(target, args) {
      if(!instance) {
        instance = Reflect.constructor(func, args);
      }
      return instance;
    }
  }
  return new Proxy(func, handler);
}

```

#28 实现数组的flat

需求：多维数组=>一维数组

```

let ary = [1, [2, [3, [4, 5]]], 6];
let str = JSON.stringify(ary);

```

```

//第0种处理:直接的调用
arr_flat = arr.flat(Infinity);
//第一种处理
ary = str.replace(/(\[\])\g, '').split(',');
//第二种处理
str = str.replace(/(\[\])\g, '');
str = '[' + str + ']';
ary = JSON.parse(str);
//第三种处理: 递归处理
let result = [];
let fn = function(ary) {
    for(let i = 0; i < ary.length; i++) {
        let item = ary[i];
        if (Array.isArray(ary[i])){
            fn(item);
        } else {
            result.push(item);
        }
    }
}
//第四种处理: 用 reduce 实现数组的 flat 方法
function flatten(ary) {
    return ary.reduce((pre, cur) => {
        return pre.concat(Array.isArray(cur) ? flatten(cur) : cur);
    }, []);
}
let ary = [1, 2, [3, 4], [5, [6, 7]]]
console.log(flatten(ary))
//第五种处理: 扩展运算符
while (ary.some(Array.isArray)) {
    ary = [].concat(...ary);
}

```

#29 请实现一个 add 函数，满足以下功能

```

add(1);           // 1
add(1)(2);       // 3
add(1)(2)(3);   // 6
add(1)(2, 3);   // 6
add(1, 2)(3);   // 6
add(1, 2, 3);   // 6
function add() {
    let args = [].slice.call(arguments);

    let fn = function(){
        let fn_args = [].slice.call(arguments)
        return add.apply(null, args.concat(fn_args))
    }

    fn.toString = function(){
        return args.reduce((a,b)=>a+b)
    }

    return fn
}

```

#30 实现一个 sleep 函数，比如 sleep(1000) 意味着等待 1000毫秒

```
const sleep = (time) => {
  return new Promise(resolve => setTimeout(resolve, time))
}

sleep(1000).then(() => {
  // 这里写你的骚操作
})
```

#31 实现 (5).add(3).minus(2) 功能

例：5 + 3 - 2，结果为 6

```
Number.prototype.add = function(n) {
  return this.valueOf() + n;
};

Number.prototype.minus = function(n) {
  return this.valueOf() - n;
};
```

#32 给定两个数组，写一个方法来计算它们的交集

例如：给定 nums1 = [1, 2, 2, 1], nums2 = [2, 2]，返回 [2, 2]。

```
function union (arr1, arr2) {
  return arr1.filter(item => {
    return arr2.indexOf(item) > - 1;
  })
}

const a = [1, 2, 2, 1];
const b = [2, 3, 2];
console.log(union(a, b)); // [2, 2]
```

#33 实现一个JS函数柯里化

是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数且返回结果的新函数的技术

通用版

```
function curry(fn, args) {
  var length = fn.length;
  var args = args || [];
  return function() {
```

```

        newArgs = args.concat(Array.prototype.slice.call(arguments));
        if (newArgs.length < length) {
            return curry.call(this, fn, newArgs);
        } else{
            return fn.apply(this,newArgs);
        }
    }

    function multiFn(a, b, c) {
        return a * b * c;
    }

    var multi = curry(multiFn);

    multi(2)(3)(4);
    multi(2,3,4);
    multi(2)(3,4);
    multi(2,3)(4)
}

```

ES6写法

```

const curry = (fn, arr = []) => (...args) => (
    arg => arg.length === fn.length
        ? fn(...arg)
        : curry(fn, arg)
)([...arr, ...args])

let curryTest=curry((a,b,c,d)=>a+b+c+d)
curryTest(1,2,3)(4) //返回10
curryTest(1,2)(4)(3) //返回10
curryTest(1,2)(3,4) //返回10

```

#34 实现一个双向绑定

defineProperty 版本

```

// 数据
const data = {
    text: 'default'
};
const input = document.getElementById('input');
const span = document.getElementById('span');
// 数据劫持
Object.defineProperty(data, 'text', {
    // 数据变化 --> 修改视图
    set(newVal) {
        input.value = newVal;
        span.innerHTML = newVal;
    }
});
// 视图更改 --> 数据变化
input.addEventListener('keyup', function(e) {
    data.text = e.target.value;
})

```

```
});
```

proxy 版本

```
// 数据
const data = {
  text: 'default'
};
const input = document.getElementById('input');
const span = document.getElementById('span');
// 数据劫持
const handler = {
  set(target, key, value) {
    target[key] = value;
    // 数据变化 --> 修改视图
    input.value = value;
    span.innerHTML = value;
    return value;
  }
};
const proxy = new Proxy(data, handler);

// 视图更改 --> 数据变化
input.addEventListener('keyup', function(e) {
  proxy.text = e.target.value;
});
```

#35 Array.isArray 实现

```
Array.myIsArray = function(o) {
  return Object.prototype.toString.call(object(o)) === '[object Array]';
};

console.log(Array.myIsArray([])); // true
```

#36 对象数组如何去重

根据每个对象的某一个具体属性来进行去重

```
const responseList = [
  { id: 1, a: 1 },
  { id: 2, a: 2 },
  { id: 3, a: 3 },
  { id: 1, a: 4 },
];
const result = responseList.reduce((acc, cur) => {
  const ids = acc.map(item => item.id);
  return ids.includes(cur.id) ? acc : [...acc, cur];
}, []);
console.log(result); // -> [ { id: 1, a: 1}, {id: 2, a: 2}, {id: 3, a: 3} ]
```

#37 实现一个函数判断数据类型

```
function getType(obj) {
  if (obj === null) return String(obj);
  return typeof obj === 'object'
    ? Object.prototype.toString.call(obj).replace('[object ', '').replace(']', '')
    .toLowerCase()
    : typeof obj;
}

// 调用
getType(null); // -> null
getType(undefined); // -> undefined
getType({}); // -> object
getType([]); // -> array
getType(123); // -> number
getType(true); // -> boolean
getType('123'); // -> string
getType(/123/); // -> regexp
getType(new Date()); // -> date
```

#38 查找字符串中出现最多的字符和个数

```
// 例: abbcccdfffff -> 字符最多的是f, 出现了5次

let str = "abcccdfffff";
let num = 0;
let char = '';

// 使其按照一定的次序排列
str = str.split('').sort().join('');
// "aaabbbbfccccc"

// 定义正则表达式
let re = /(\w)\1+/g;
str.replace(re, ($0,$1) => {
  if(num < $0.length){
    num = $0.length;
    char = $1;
  }
});
console.log(`字符最多的是${char}, 出现了${num}次`);
```

#39 数组去重问题

首先:我知道多少种去重方式

#双层 for 循环

```
function distinct(arr) {
    for (let i=0, len=arr.length; i<len; i++) {
        for (let j=i+1; j<len; j++) {
            if (arr[i] == arr[j]) {
                arr.splice(j, 1);
                // splice 会改变数组长度，所以要将数组长度 len 和下标 j 减一
                len--;
                j--;
            }
        }
    }
    return arr;
}
```

思想: 双重 `for` 循环是比较笨拙的方法, 它实现的原理很简单: 先定义一个包含原始数组第一个元素的数组, 然后遍历原始数组, 将原始数组中的每个元素与新数组中的每个元素进行比对, 如果不重复则添加到新数组中, 最后返回新数组; 因为它的时间复杂度是 $O(n^2)$, 如果数组长度很大, 效率会很低

#Array.filter() 加 indexOf/includes

```
function distinct(a, b) {
    let arr = a.concat(b);
    return arr.filter((item, index)=> {
        //return arr.indexOf(item) === index
        return arr.includes(item)
    })
}
```

思想: 利用 `indexof` 检测元素在数组中第一次出现的位置是否和元素现在的位置相等, 如果不等则说明该元素是重复元素

#ES6 中的 Set 去重

```
function distinct(array) {
    return Array.from(new Set(array));
}
```

思想: ES6 提供了新的数据结构 `Set`, `Set` 结构的一个特性就是成员值都是唯一的, 没有重复的值。

#reduce 实现对象数组去重复

```
var resources = [
    { name: "张三", age: "18" },
    { name: "张三", age: "19" },
    { name: "张三", age: "20" },
    { name: "李四", age: "19" },
    { name: "王五", age: "20" },
    { name: "赵六", age: "21" }
]
var temp = {};
resources = resources.reduce((prev, curv) => {
```

```
// 如果临时对象中有这个名字，什么都不做
if (temp[curv.name]) {

} else {
    // 如果临时对象没有就把这个名字加进去，同时把当前的这个对象加入到prev中
    temp[curv.name] = true;
    prev.push(curv);
}
return prev
}, []));
console.log("结果", resources);
```

这种方法是利用高阶函数 `reduce` 进行去重，这里只需要注意 `initialValue` 得放一个空数组 `[]`，不然没法 `push`

设计模式

#一、基础篇

#this、new、bind、call、apply

1. this 指向的类型

刚开始学习 JavaScript 的时候，`this` 总是最能让人迷惑，下面我们一起看一下在 JavaScript 中应该如何确定 `this` 的指向。`this` 是在函数被调用时确定的，它的指向完全取决于函数调用的地方，而不是它被声明的地方（除箭头函数外）。当一个函数被调用时，会创建一个执行上下文，它包含函数在哪里被调用（调用栈）、函数的调用方式、传入的参数等信息，`this` 就是这个记录的一个属性，它会在函数执行的过程中被用到。

`this` 在函数的指向有以下几种场景：

- 作为构造函数被 `new` 调用；
- 作为对象的方法使用；
- 作为函数直接调用；
- 被 `call`、`apply`、`bind` 调用；
- 箭头函数中的 `this`；

1.1 new 绑定

函数如果作为构造函数使用 `new` 调用时，`this` 绑定的是新创建的构造函数的实例。

```
function Foo() {
    console.log(this)
}

var bar = new Foo()      // 输出：Foo 实例，this 就是 bar
```

实际上使用 `new` 调用构造函数时，会依次执行下面的操作：

- 创建一个新对象；
- 构造函数的 `prototype` 被赋值给这个新对象的 `__proto__`；
- 将新对象赋给当前的 `this`；

- 执行构造函数；
- 如果函数没有返回其他对象，那么 `new` 表达式中的函数调用会自动返回这个新对象，如果返回的不是对象将被忽略；

1.2 显式绑定

通过 `call`、`apply`、`bind` 我们可以修改函数绑定的 `this`，使其成为我们指定的对象。通过这些方法的第一个参数我们可以显式地绑定 `this`。

```
function foo(name, price) {
  this.name = name
  this.price = price
}

function Food(category, name, price) {
  foo.call(this, name, price)          // call 方式调用
  // foo.apply(this, [name, price])    // apply 方式调用
  this.category = category
}

new Food('食品', '汉堡', '5块钱')

// 浏览器中输出: {name: "汉堡", price: "5块钱", category: "食品"}
call 和 apply 的区别是 call 方法接受的是参数列表，而 apply 方法接受的是一个参数数组。

func.call(thisArg, arg1, arg2, ...)      // call 用法
func.apply(thisArg, [arg1, arg2, ...])    // apply 用法
```

而 `bind` 方法是设置 `this` 为给定的值，并返回一个新的函数，且在调用新函数时，将给定参数列表作为原函数的参数序列的前若干项。

```
func.bind(thisArg[, arg1[, arg2[, ...]]]) // bind 用法
```

举个例子：

```
var food = {
  name: '汉堡',
  price: '5块钱',
  getPrice: function(place) {
    console.log(place + this.price)
  }
}

food.getPrice('KFC ') // 浏览器中输出: "KFC 5块钱"

var getPrice1 = food.getPrice.bind({ name: '鸡腿', price: '7块钱' }, '肯打鸡 ')
getPrice1()          // 浏览器中输出: "肯打鸡 7块钱"
```

关于 `bind` 的原理，我们可以使用 `apply` 方法自己实现一个 `bind` 看一下：

```
// ES5 方式
Function.prototype.bind = Function.prototype.bind || function() {
  var self = this
  var rest1 = Array.prototype.slice.call(arguments)
  var context = rest1.shift()
  return function() {
    var rest2 = Array.prototype.slice.call(arguments)
```

```
        return self.apply(context, rest1.concat(rest2))
    }
}

// ES6 方式
Function.prototype.bind = Function.prototype.bind || function(...rest1) {
    const self = this
    const context = rest1.shift()
    return function(...rest2) {
        return self.apply(context, [...rest1, ...rest2])
    }
}
```

ES6 方式用了一些 ES6 的知识比如 `rest` 参数、数组解构

注意：如果你把 `null` 或 `undefined` 作为 `this` 的绑定对象传入 `call`、`apply`、`bind`，这些值在调用时会被忽略，实际应用的是默认绑定规则。

```
var a = 'hello'

function foo() {
    console.log(this.a)
}

foo.call(null)          // 浏览器中输出： "hello"
```

1.3 隐式绑定

函数是否在某个上下文对象中调用，如果是的话 `this` 绑定的是那个上下文对象。

```
var a = 'hello'

var obj = {
    a: 'world',
    foo: function() {
        console.log(this.a)
    }
}

obj.foo()          // 浏览器中输出： "world"
```

上面代码中，`foo` 方法是作为对象的属性调用的，那么此时 `foo` 方法执行时，`this` 指向 `obj` 对象。也就是说，此时 `this` 指向调用这个方法的对象，如果嵌套了多个对象，那么指向最后一个调用这个方法的对象：

```
var a = 'hello'

var obj = {
  a: 'world',
  b: {
    a: 'China',
    foo: function() {
      console.log(this.a)
    }
  }
}

obj.b.foo() // 浏览器中输出: "China"
```

最后一个对象是 `obj` 上的 `b`，那么此时 `foo` 方法执行时，其中的 `this` 指向的就是 `b` 对象。

1.4 默认绑定

函数独立调用，直接使用不带任何修饰的函数引用进行调用，也是上面几种绑定途径之外的方式。非严格模式下 `this` 绑定到全局对象（浏览器下是 `window`，`node` 环境是 `global`），严格模式下 `this` 绑定到 `undefined`（因为严格模式不允许 `this` 指向全局对象）。

```
var a = 'hello'

function foo() {
  var a = 'world'
  console.log(this.a)
  console.log(this)
}

foo() // 相当于执行 window.foo()

// 浏览器中输出: "hello"
// 浏览器中输出: Window 对象
```

上面代码中，变量 `a` 被声明在全局作用域，成为全局对象 `window` 的一个同名属性。函数 `foo` 被执行时，`this` 此时指向的是全局对象，因此打印出来的 `a` 是全局对象的属性。

注意有一种情况：

```
var a = 'hello'

var obj = {
  a: 'world',
  foo: function() {
    console.log(this.a)
  }
}

var bar = obj.foo

bar() // 浏览器中输出: "hello"
```

此时 `bar` 函数，也就是 `obj` 上的 `foo` 方法为什么又指向了全局对象呢，是因为 `bar` 方法此时是作为函数独立调用的，所以此时的场景属于默认绑定，而不是隐式绑定。这种情况和把方法作为回调函数的场景类似：

```
var a = 'hello'

var obj = {
  a: 'world',
  foo: function() {
    console.log(this.a)
  }
}

function func(fn) {
  fn()
}

func(obj.foo) // 浏览器中输出: "hello"
```

- 参数传递实际上也是一种隐式的赋值，只不过这里 `obj.foo` 方法是被隐式赋值给了函数 `func` 的形参 `fn`，而之前的情景是自己赋值，两种情景实际上类似。这种场景我们遇到的比较多的是 `setTimeout` 和 `setInterval`，如果回调函数不是箭头函数，那么其中的 `this` 指向的就是全局对象。
- 其实我们可以把默认绑定当作是隐式绑定的特殊情况，比如上面的 `bar()`，我们可以当作是使用 `window.bar()` 的方式调用的，此时 `bar` 中的 `this` 根据隐式绑定的情景指向的就是 `window`。

2. this 绑定的优先级

`this` 存在多个使用场景，那么多个场景同时出现的时候，`this` 到底应该如何指向呢。这里存在一个优先级的概念，`this` 根据优先级来确定指向。**优先级：new 绑定 > 显示绑定 > 隐式绑定 > 默认绑定**

所以 `this` 的判断顺序：

- `new` 绑定：函数是否在 `new` 中调用？如果是的话 `this` 绑定的是新创建的对象；
- 显示绑定：函数是否是通过 `bind`、`call`、`apply` 调用？如果是的话，`this` 绑定的是指定的对象；
- 隐式绑定：函数是否在某个上下文对象中调用？如果是的话，`this` 绑定的是那个上下文对象；
- 如果都不是的话，使用默认绑定。如果在严格模式下，就绑定到 `undefined`，否则绑定到全局对象；

3. 箭头函数中的 this

- 箭头函数是根据其声明的地方来决定 `this` 的
- 箭头函数的 `this` 绑定是无法通过 `call`、`apply`、`bind` 被修改的，且因为箭头函数没有构造函数 `constructor`，所以也不可以使用 `new` 调用，即不能作为构造函数，否则会报错。

```

var a = 'hello'

var obj = {
  a: 'world',
  foo: () => {
    console.log(this.a)
  }
}

obj.foo()          // 浏览器中输出: "hello"

```

4. 一个 this 的小练习

用一个小练习来实战一下：

```

var a = 20

var obj = {
  a: 40,
  foo: () => {
    console.log(this.a)

    function func() {
      this.a = 60
      console.log(this.a)
    }

    func.prototype.a = 50
    return func
  }
}

var bar = obj.foo()          // 浏览器中输出: 20
bar()                      // 浏览器中输出: 60
new bar()                  // 浏览器中输出: 60

```

稍微解释一下：

- `var a = 20` 这句在全局变量 `window` 上创建了个属性 `a` 并赋值为 `20`；
- 首先执行的是 `obj.foo()`，这是一个箭头函数，箭头函数不创建新的函数作用域直接沿用语句外部的作用域，因此 `obj.foo()` 执行时箭头函数中 `this` 是全局 `window`，首先打印出 `window` 上的属性 `a` 的值 `20`，箭头函数返回了一个原型上有个值为 `50` 的属性 `a` 的函数对象 `func` 给 `bar`；
- 继续执行的是 `bar()`，这里执行的是刚刚箭头函数返回的闭包 `func`，其内部的 `this` 指向 `window`，因此 `this.a` 修改了 `window.a` 的值为 `60` 并打印出来；
- 然后执行的是 `new bar()`，根据之前的表述，`new` 操作符会在 `func` 函数中创建一个继承了 `func` 原型的实例对象并用 `this` 指向它，随后 `this.a = 60` 又在实例对象上创建了一个属性 `a`，在之后的打印中已经在实例上找到了属性 `a`，因此就不继续往对象原型上查找了，所以打印出第三个 `60`；
- 如果把上面例子的箭头函数换成普通函数呢，结果会是什么样？

```

var a = 20

var obj = {
  a: 40,

```

```

foo: function() {
    console.log(this.a)

    function func() {
        this.a = 60
        console.log(this.a)
    }

    func.prototype.a = 50
    return func
}

var bar = obj.foo()          // 浏览器中输出: 40
bar()                      // 浏览器中输出: 60
new bar()                  // 浏览器中输出: 60

```

#闭包与高阶函数

1. 闭包

1.1 什么是闭包

当函数可以记住并访问所在的词法作用域时，就产生了闭包，即使函数是在当前词法作用域之外执行。

我们首先来看一个闭包的例子：

```

function foo() {
    var a = 2

    function bar() {
        console.log(a)
    }

    return bar
}

var baz = foo()

baz()           // 输出: 2

```

- `foo` 函数传递出了一个函数 `bar`，传递出来的 `bar` 被赋值给 `baz` 并调用，虽然这时 `baz` 是在 `foo` 作用域外执行的，但 `baz` 在调用的时候可以访问到前面的 `bar` 函数所在的 `foo` 的内部作用域。
- 由于 `bar` 声明在 `foo` 函数内部，`bar` 拥有涵盖 `foo` 内部作用域的闭包，使得 `foo` 的内部作用域一直存活不被回收。一般来说，函数在执行完后其整个内部作用域都会被销毁，因为 `JavaScript` 的 `GC` (Garbage Collection) 垃圾回收机制会自动回收不再使用的内存空间。但是闭包会阻止某些 `GC`，比如本例中 `foo()` 执行完，因为返回的 `bar` 函数依然持有其所在作用域的引用，所以其内部作用域不会被回收。
- 注意：如果不是必须使用闭包，那么尽量避免创建它，因为闭包在处理速度和内存消耗方面对性能具有负面影响。

1.2 利用闭包实现结果缓存（备忘模式）

备忘模式就是应用闭包的特点的一个典型应用。比如有个函数：

```
function add(a) {
    return a + 1;
}
```

- 多次运行 `add()` 时，每次得到的结果都是重新计算得到的，如果是开销很大的计算操作的话就比较消耗性能了，这里可以对已经计算过的输入做一个缓存。
- 所以这里可以利用闭包的特点来实现一个简单的缓存，在函数内部用一个对象存储输入的参数，如果下次再输入相同的参数，那就比较一下对象的属性，如果有缓存，就直接把值从这个对象里面取出来。

```
/* 备忘函数 */
function memorize(fn) {
    var cache = {}
    return function() {
        var args = Array.prototype.slice.call(arguments)
        var key = JSON.stringify(args)
        return cache[key] || (cache[key] = fn.apply(fn, args))
    }
}

/* 复杂计算函数 */
function add(a) {
    return a + 1
}

var adder = memorize(add)

adder(1)          // 输出: 2    当前: cache: { '[1]': 2 }
adder(1)          // 输出: 2    当前: cache: { '[1]': 2 }
adder(2)          // 输出: 3    当前: cache: { '[1]': 2, '[2]': 3 }
```

使用 `ES6` 的方式会更优雅一些：

```
/* 备忘函数 */
function memorize(fn) {
    const cache = {}
    return function(...args) {
        const key = JSON.stringify(args)
        return cache[key] || (cache[key] = fn.apply(fn, args))
    }
}

/* 复杂计算函数 */
function add(a) {
    return a + 1
}

const adder = memorize(add)

adder(1)          // 输出: 2    当前: cache: { '[1]': 2 }
adder(1)          // 输出: 2    当前: cache: { '[1]': 2 }
adder(2)          // 输出: 3    当前: cache: { '[1]': 2, '[2]': 3 }
```

稍微解释一下：

- 备忘函数中用 `JSON.stringify` 把传给 `adder` 函数的参数序列化成字符串，把它当做 `cache` 的索引，将 `add` 函数运行的结果当做索引的值传递给 `cache`，这样 `adder` 运行的时候如果传递的参数之前传递过，那么就返回缓存好的计算结果，不用再计算了，如果传递的参数没计算过，则计算并缓存 `fn.apply(fn, args)`，再返回计算的结果。
- 当然这里的实现如果要实际应用的话，还需要继续改进一下，比如：
- 缓存不可以永远扩张下去，这样太耗费内存资源，我们可以只缓存最新传入的 `n` 个；
- 在浏览器中使用的时候，我们可以借助浏览器的持久化手段，来进行缓存的持久化，比如 `cookie`、`localStorage` 等；
- 这里的复杂计算函数可以是过去的某个状态，比如对某个目标的操作，这样把过去的状态缓存起来，方便地进行状态回退。
- 复杂计算函数也可以是一个返回时间比较慢的异步操作，这样如果把结果缓存起来，下次就可以直接从本地获取，而不是重新进行异步请求。

注意：`cache` 不可以是 `Map`，因为 `Map` 的键是使用 `==` 比较的，因此当传入引用类型值作为键时，虽然它们看上去是相等的，但实际并不是，比如 `[1] != [1]`，所以还是被存为不同的键。

```
// ✗ 错误示范
function memorize(fn) {
  const cache = new Map()
  return function(...args) {
    return cache.get(args) || cache.set(args, fn.apply(fn, args)).get(args)
  }
}

function add(a) {
  return a + 1
}

const adder = memorize(add)

adder(1) // 2 cache: { [1] => 2 }
adder(1) // 2 cache: { [1] => 2, [1] => 2 }
adder(2) // 3 cache: { [1] => 2, [1] => 2, [2] => 3 }
```

2. 高阶函数

高阶函数就是输入参数里有函数，或者输出是函数的函数。

2.1 函数作为参数

如果你用过 `setTimeout`、`setInterval`、`ajax` 请求，那么你已经用过高阶函数了，这是我们最常看到的场景：回调函数，因为它将函数作为参数传递给另一个函数。

比如 `ajax` 请求中，我们通常使用回调函数来定义请求成功或者失败时的操作逻辑：

```
$.ajax("/request/url", function(result){
  console.log("请求成功！")
})
```

在 `Array`、`Object`、`String` 等等基本对象的原型上有很多操作方法，可以接受回调函数来方便地进行对象操作。这里举一个很常用的 `Array.prototype.filter()` 方法，这个方法返回一个新创建的数组，包含所有回调函数执行后返回 `true` 或真值的数组元素。

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

var result = words.filter(function(word) {
    return word.length > 6
}) // 输出: ["exuberant", "destruction", "present"]
```

回调函数还有一个应用就是钩子，如果你用过 Vue 或者 React 等框架，那么你应该对钩子很熟悉了，它的形式是这样的：

```
function foo(callback) {
    // ... 一些操作
    callback()
}
```

2.2 函数作为返回值

另一个经常看到的高阶函数的场景是在一个函数内部输出另一个函数，比如：

```
function foo() {
    return function bar() {}
}
```

主要是利用闭包来保持着作用域：

```
function add() {
    var num = 0
    return function(a) {
        return num = num + a
    }
}
var adder = add()

adder(1) // 输出: 1
adder(2) // 输出: 3
```

1. 柯里化

- 柯里化 (Currying)，又称部分求值 (Partial Evaluation)，是把接受多个参数的原函数变换为接受一个单一参数 (原函数的第一个参数) 的函数，并且返回一个新函数，新函数能够接受余下的参数，最后返回同原函数一样的结果。
- 核心思想是把多参数传入的函数拆成单 (或部分) 参数函数，内部再返回调用下一个单 (或部分) 参数函数，依次处理剩余的参数。

柯里化有 3 个常见作用：

- 参数复用
- 提前返回
- 延迟计算/运行
- 先来看看柯里化的通用实现：

```
// ES5 方式
function currying(fn) {
    var rest1 = Array.prototype.slice.call(arguments)
    rest1.shift()
    return function() {
```

```

        var rest2 = Array.prototype.slice.call(arguments)
        return fn.apply(null, rest1.concat(rest2))
    }
}

// ES6 方式
function currying(fn, ...rest1) {
    return function(...rest2) {
        return fn.apply(null, rest1.concat(rest2))
    }
}

```

用它将一个 `sayHello` 函数柯里化试试：

```

// 接上面
function sayHello(name, age, fruit) {
    console.log(`我叫 ${name}, 我 ${age} 岁了, 我喜欢吃 ${fruit}`))
}

var curryingShowMsg1 = currying(sayHello, '小明')
curryingShowMsg1(22, '苹果')           // 输出：我叫 小明,我 22 岁了, 我喜欢吃 苹果

var curryingShowMsg2 = currying(sayHello, '小袁', 20)
curryingShowMsg2('西瓜')             // 输出：我叫 小袁,我 20 岁了, 我喜欢吃 西瓜

```

更高阶的用法参见：JavaScript 函数式编程技巧 - 柯里化

2. 反柯里化

- 柯里化是固定部分参数，返回一个接受剩余参数的函数，也称为部分计算函数，目的是为了缩小适用范围，创建一个针对性更强的函数。核心思想是把多参数传入的函数拆成单参数（或部分）函数，内部再返回调用下一个单参数（或部分）函数，依次处理剩余的参数。
- 而反柯里化，从字面讲，意义和用法跟函数柯里化相比正好相反，扩大适用范围，创建一个应用范围更广的函数。使本来只有特定对象才适用的方法，扩展到更多的对象。

先来看看反柯里化的通用实现吧~

```

// ES5 方式
Function.prototype.unCurrying = function() {
    var self = this
    return function() {
        var rest = Array.prototype.slice.call(arguments)
        return Function.prototype.call.apply(self, rest)
    }
}

```

```

// ES6 方式
Function.prototype.unCurrying = function() {
    const self = this
    return function(...rest) {
        return Function.prototype.call.apply(self, rest)
    }
}

```

如果你觉得把函数放在 `Function` 的原型上不太好，也可以这样：

```

// ES5 方式
function unCurrying(fn) {

```

```

        return function (tar) {
            var rest = Array.prototype.slice.call(arguments)
            rest.shift()
            return fn.apply(tar, rest)
        }
    }

// ES6 方式
function unCurrying(fn) {
    return function(tar, ...argu) {
        return fn.apply(tar, argu)
    }
}

```

下面简单试用一下反柯里化通用实现，我们将 `Array` 上的 `push` 方法借出来给 `arguments` 这样的类数组增加一个元素：

```

// 接上面
var push = unCurrying(Array.prototype.push)

function execPush() {
    push(arguments, 4)
    console.log(arguments)
}

execPush(1, 2, 3)      // 输出: [1, 2, 3, 4]

```

简单说，函数柯里化就是对高阶函数的降阶处理，缩小适用范围，创建一个针对性更强的函数。

```

function(arg1, arg2)          // => function(arg1)(arg2)
function(arg1, arg2, arg3)    // => function(arg1)(arg2)(arg3)
function(arg1, arg2, arg3, arg4) // => function(arg1)(arg2)(arg3)(arg4)
function(arg1, arg2, ..., argn) // => function(arg1)(arg2)...(argn)

```

而反柯里化就是反过来，增加适用范围，让方法使用场景更大。使用反柯里化，可以把原生方法借出来，让任何对象拥有原生对象的方法。

```
obj.func(arg1, arg2)          // => func(obj, arg1, arg2)
```

可以这样理解柯里化和反柯里化的区别：

- 柯里化是在运算前传参，可以传递多个参数；
- 反柯里化是延迟传参，在运算时把原来已经固定的参数或者 `this` 上下文等当作参数延迟到未来传递。
- 更高阶的用法参见：JavaScript 函数式编程技巧 - 反柯里化

3. 偏函数

偏函数是创建一个调用另外一个部分（参数或变量已预制的函数）的函数，函数可以根据传入的参数来生成一个真正执行的函数。其本身不包括我们真正需要的逻辑代码，只是根据传入的参数返回其他的函数，返回的函数中才有真正的处理逻辑比如：

```
var isType = function(type) {
    return function(obj) {
        return Object.prototype.toString.call(obj) === `[object ${type}]`
    }
}

var isString = isType('String')
var isFunction = isType('Function')
```

这样就用偏函数快速创建了一组判断对象类型的方法~

偏函数和柯里化的区别：

- 柯里化是把一个接受 n 个参数的函数，由原本的一次性传递所有参数并执行变成了可以分多次接受参数再执行，例如：`add = (x, y, z) => x + y + z`-`curryAdd = x => y => z => x + y + z;`
- 偏函数固定了函数的某个部分，通过传入的参数或者方法返回一个新的函数来接受剩余的参数，数量可能是一个也可能是多个；
- 当一个柯里化函数只接受两次参数时，比如 `curry()`，这时的柯里化函数和偏函数概念类似，可以认为偏函数是柯里化函数的退化版

#ES6

1. let、const

一个显而易见特性是 `let` 声明的变量还可以更改，而 `const` 一般用来声明常量，声明之后就不能更改了：

```
let foo = 1;
const bar = 2;
foo = 3;
bar = 3; // 报错 TypeError
```

1.1 作用域差别

刚学 JavaScript 的时候，我们总是看到类似于「JavaScript 中没有块级作用域，只有函数作用域」的说法。举个例子：

```
var arr = [];
for (var i = 0; i < 4; i++) {
    arr[i] = function() {
        console.log(i)
    }
}
arr[2]() // 期望值：2，输出：4
```

因为 `i` 变量是 `var` 命令声明的，`var` 声明的变量的作用域是函数作用域，因此此时 `i` 变量是在全局范围内都有效，也就是说全局只有一个变量 `i`，每次循环只是修改同一个变量 `i` 的值。虽然函数的定义是在循环中进行，但是每个函数的 `i` 都指向这个全局唯一的变量 `i`。在函数执行时，`for` 循环已经结束，`i` 最终的值是 4，所以无论执行数组里的哪个函数，结果都是 `i` 最终的值 4。

ES6 引入的 `let`、`const` 声明的变量是仅在块级作用域中有效：

```
var arr = [];
for (let i = 0; i < 4; i++) {
  arr[i] = function () {
    console.log(i)
  }
}
arr[2]() // 期望值: 2, 输出: 2
```

这个代码中，变量 `i` 是 `let` 声明的，也就是说 `i` 只在本轮循环有效，所以每次循环 `i` 都是一个新的变量，最后输出的是 2。

那如果我们不使用 ES6 的 `let`、`const` 怎样去实现？可以使用函数的参数来缓存变量的值，让闭包在执行时索引到的变量为函数作用域中缓存的函数参数变量值：

```
var arr = []
for (var i = 0; i < 4; i++) {
  (function(j) {
    arr[i] = function(j) {
      console.log(j)
    }
  })(i)
}
arr[2]() // 输出: 2
```

这个做法归根结底还是使用函数作用域来变相实现块级作用域，事实上 Babel 编译器也是使用这个做法，我们来看看 Babel 编译的结果：

```
// 编译前, ES6 语法
var arr = [];
for (let i = 0; i < 4; i++) {
  arr[i] = function () {
    console.log(i)
  }
}
arr[2]() // 输出: 2

// 编译后, Babel 编译后的 ES5 语法
"use strict";
var arr = [];
var _loop = function _loop(i) {
  arr[i] = function () {
    console.log(i);
  };
}

for (var i = 0; i < 4; i++) {
  _loop(i);
}

arr[2](); // 输出: 2
```

可以看到 Babel 编译后的代码，也是使用了这个做法。

1.2 不存在变量提升

`var` 命令声明的变量会发生变量提升的现象，也就是说变量在声明之前使用，其值为 `undefined`，`function` 声明的函数也是有这样的特性。而 `let`、`const` 命令声明的变量没有变量提升，如果在声明之前使用，会直接报错。

```
// var 命令存在变量提升
console.log(tmp) // undefined
var tmp = 1
console.log(tmp) // 1

// let、const 命令不存在变量提升
console.log(boo) // 报错 ReferenceError
let boo = 2
```

1.3 暂时性死区

在一个块级作用域中对一个变量使用 `let`、`const` 声明前，该变量都是不可使用的，这被称为暂时性死区（Temporal Dead Zone, TDZ）：

```
tmp = 'asd';
if (true) {
    // 虽然在这之前定义了一个全局变量 tmp，但是块内重新定义了一个 tmp
    console.log(tmp); // 报错 ReferenceError
    let tmp;
}
```

1.4 不允许重复声明

`let`、`const` 命令是不允许重复声明同一个变量的：

```
if (true) {
    let tmp;
    let tmp; // 报错 SyntaxError
}

function func(arg) { // 因为已经有一个 arg 变量名的形参了
    let arg;
}
func() // 报错 SyntaxError
```

2. 箭头函数

2.1 基本用法

ES6 中可以使用箭头函数来定义函数。下面例子中，同名函数的定义是等价的：

```
// 基础用法
const test1 = function (参数1, 参数2, ..., 参数N) { 函数声明 }
const test1 = (参数1, 参数2, ..., 参数N) => { 函数声明 }

// 当只有一个参数时，圆括号是可选的
const test2 = (单一参数) => { 函数声明 }
const test2 = 单一参数 => { 函数声明 }

// 没有参数时，圆括号不能省略
const test3 = () => { 函数声明 }
```

```
// 当函数体只是 return 一个单一表达式时，可以省略花括号和 return 关键词
const test4 = () => return 表达式(单一)
const test4 = () => 表达式(单一)

// 函数体返回对象字面表达式时，如果省略花括号和 return 关键词，返回值需要加括号
const test5 = () => { return {foo: 'bar'} }
const test5 = () => ({foo: 'bar'}) // 输出 {foo: 'bar'}
const test6 = () => {foo: 'bar'} // 输出 undefined, 大括号被识别为代码块
```

总结：

- 参数如果只有一个，可以不加圆括号 ()；
- 没有参数时，不能省略圆括号 ()；
- 如果函数体只返回单一表达式，那么函数体可以不使用大括号 {} 和 return，直接写表达式即可；
- 在 3 的基础上，如果返回值是一个对象字面量，那么返回值需要加圆括号 ()，避免被识别为代码块。

2.2 箭头函数中的 this

箭头函数出来之前，函数在执行时才能确定 this 的指向，所以会经常出现闭包中的 this 指向不是期望值的情况。在以前的做法中，如果要给闭包指定 this，可以用 bind\call\apply，或者把 this 值分配给封闭的变量（一般是 that）。箭头函数出来之后，给我们提供了不一样的选择。

箭头函数不会创建自己的 this，只会从自己定义位置的作用域的上一层直接继承 this。

```
function Person() {
  this.age = 10;

  setInterval(() => {
    this.age++; // this 正确地指向 p 实例
  }, 1000);
}

var p = new Person(); // 1s后打印出 10
```

另外因为箭头函数没有自己的 this 指针，因此对箭头函数使用 call、apply、bind 时，只能传递函数，不能绑定 this，它们的第一个参数将被忽略：

```
this.param = 1

const func1 = () => console.log(this.param)
const func2 = function() {
  console.log(this.param)
}
func1.apply({ param: 2 }) // 输出: 1
func2.apply({ param: 2 }) // 输出: 2
```

总结一下：

- 箭头函数中的 this 就是定义时所在的对象，而不是使用时所在的对象；
- 无法作为构造函数，不可以使用 new 命令，否则会抛错；
- 箭头函数中不存在 arguments 对象，但我们可以用 Rest 参数来替代；
- 箭头函数无法使用 yield 命令，所以不能作为 Generator 函数；

- 不能通过 `bind`、`call`、`apply` 绑定 `this`，但是可以通过 `call`、`apply` 传递参数。

3. class 语法

在 `class` 语法出来之前，我们一般通过上一章介绍的一些方法，来间接实现面向对象三个要素：封装、继承、多态。ES6 给我们提供了更面向对象（更 `OO`, `Object Oriented`）的写法，我们可以通过 `class` 关键字来定义一个类。

基本用法：

```
// ES5 方式定义一个类
function Foo() { this.kind = 'foo' }

Foo.staticMethod = function() { console.log('静态方法') }

Foo.prototype.doThis = function() { console.log(`实例方法 kind:${this.kind}`) }

// ES6 方式定义一个类
class Foo {
  /* 构造函数 */
  constructor() { this.kind = 'foo' }

  /* 静态方法 */
  static staticMethod() { console.log('静态方法') }

  /* 实例方法 */
  doThis() {
    console.log(`实例方法 kind:${this.kind}`)
  }
}
```

ES6 方式实现继承：

```
// 接上
class Bar extends Foo {
  constructor() {
    super()
    this.type = 'bar'
  }

  doThat() {
    console.log(`实例方法 type:${this.type} kind:${this.kind}`)
  }
}

const bar = new Bar()
bar.doThat() // 实例方法 type:bar kind:foo
```

总结一下：

- `static` 关键字声明的是静态方法，不会被实例继承，只可以调用；
- `class` 没有变量提升，因此必须在定义之后才使用；
- `constructor` 为构造函数，子类构造函数中的 `super` 代表父类的构造函数，必须执行一次，否则新建实例时会抛错；
- `new.target` 一般用在构造函数中，返回 `new` 命令作用于的那个构造函数；
- `class` 用 `extends` 来实现继承，子类继承父类所有实例方法和属性。

4. 解构赋值

ES6 允许按照一定方式，从数组和对象中提取值。本质上这种写法属于模式匹配，只要等号两边的模式相同，左边的变量就会被赋予相对应的值。

数组解构基本用法：

```
let [a, b, c] = [1, 2, 3]           // a:1 b:2 c:3
let [a, [[b], c]] = [1, [[2], 3]]  // a:1 b:2 c:3
let [a, , b] = [1, 2, 3]           // a:1 b:3
let [a, ...b] = [1, 2, 3]           // a:1 b:[2, 3]
let [a, b, ...c] = [1]             // a:1 b:undefined c: []
let [a, b = 4] = [null, undefined] // a:null b:4
let [a, b = 4] = [1]               // a:1 b:4
let [a, b = 4] = [1, null]        // a:1 b:null
```

- 解构不成功，变量的值为 `undefined`；
- 解构可以指定默认值，如果被解构变量的对应位置没有值，即为空，或者值为 `undefined`，默认值才会生效。

对象解构基本用法：

```
let { a, b } = { a: 1, b: 2 }      // a:1 b:2
let { c } = { a: 1, b: 2 }          // c:undefined
let { c = 4 } = { a: 1, b: 2 }      // c:4
let { a: c } = { a: 1, b: 2 }      // c:1
let { a: c = 4, d: e = 5 } = { a: 1, b: 2 } // c:1 e:5
let { length } = [1, 2]            // length:2
```

- 解构不成功，变量的值为 `undefined`；
- 解构可以指定默认值，如果被解构变量严格为 `undefined` 或为空，默认值才会生效；
- 如果变量名和属性名不一致，可以赋给其它名字的变量 `{a:c}`，实际上对象解构赋值 `{a}` 是简写 `{a:a}`，对象的解构赋值是先找到同名属性，再赋给对应的变量，真正被赋值的是后者。

5. 扩展运算符

扩展运算符和 `Rest` 参数的形式一样 `...`，作用相当于 `Rest` 参数的逆运算。它将一个数组转化为逗号分割的参数序列。事实上实现了迭代器（`Iterator`）接口的对象都可以使用扩展运算符，包括 `Array`、`String`、`Set`、`Map`、`NodeList`、`arguments` 等。

数组可以使用扩展运算符：

```
console.log(...[1, 2, 3])          // 1 2 3
console.log(1, ...[2, 3, 4], 5)      // 1 2 3 4 5
[...document.querySelectorAll('div')] // [<div>, <div>, <div>]
[...[1], ...[2, 3]]                // [1, 2, 3]

const arr = [1]
arr.push(...[2, 3])                // arr:[1, 2, 3]
```

对象也可以使用扩展运算符，通常被用来合并对象：

```
{...{a: 1}, ...{a: 2, b: 3}}       // {a: 2, b: 3}
```

6. 默认参数

ES6 允许给函数的参数设置默认值，如果不传递、或者传递为 `undefined` 则会采用默认值：

```
function log(x, y = 'World') {
    console.log(x, y)
}

log('Hello')          // Hello world
log('Hello', undefined) // Hello world
log('Hello', 'China') // Hello China
log(undefined, 'China') // undefined China
log(), 'China') // 报错 SyntaxError
log('Hello', '') // Hello
log('Hello', null) // Hello null
```

注意：

- 参数不传递或者传递 `undefined` 会让参数等于默认值，但是如果参数不是最后一个，不传递参数会报错；
- 特别注意，传递 `null` 不会让函数参数等于默认值。
- 默认参数可以和解构赋值结合使用：

```
function log({x, y = 'World'} = {}) {
    console.log(x, y)
}

log({x: 'hello'})          // hello world
log({x: 'hello', y: 'China'}) // hello China
log({y: 'China'})          // undefined "China"
log({})                    // undefined "World"
log()                      // undefined "World"
```

分析一下后两种情况：

- 传递参数为 `{}` 时，因为被解构变量既不为空，也不是 `undefined`，所以不会使用解构赋值的默认参数 `{}`。虽然最终形参的赋值过程还是 `{x, y = 'World'} = {}`，但是这里等号右边的空对象是调用时传递的，而不是形参对象的默认值；
- 不传参时，即被解构变量为空，那么会使用形参的默认参数 `{}`，形参的赋值过程相当于 `{x, y = 'World'} = {}`，注意这里等号右边的空对象，是形参对象的默认值。
- 上面是给被解构变量的整体设置了一个默认值 `{}`。下面细化一下，给默认值 `{}` 中的每一项也设置默认值：

```
function log({x, y} = {x: 'yes', y: 'World'}) {
    console.log(x, y)
}

log({x: 'hello'})          // hello undefined
log({x: 'hello', y: 'China'}) // hello China
log({y: 'China'})          // undefined "China"
log({})                    // undefined undefined
log()                      // yes World
```

也分析一下后两种情况：

- 传递参数为 `{}` 时，被解构变量不为空，也不为 `undefined`，因此不使用默认参数 `{x, y: 'World'}`，形参的赋值过程相当于 `{x, y} = {}`，所以 `x` 与 `y` 都是 `undefined`；

- 不传参时，等式右边采用默认参数，形参赋值过程相当于 `{x, y} = {x: 'yes', y: 'world'}`。

7. Rest 参数

我们知道 `arguments` 是类数组，没有数组相关方法。为了使用数组上的一些方法，我们需要先用 `Array.prototype.slice.call(arguments)` 或者 `[...arguments]` 来将 `arguments` 类数组转化为数组。

ES6 允许我们通过 Rest 参数来获取函数的多余参数：

```
// 获取函数所有的参数, rest 为数组
function func1(...rest){ /* ... */}

// 获取函数第一个参数外其他的参数, rest 为数组
function func1(val, ...rest){ /* ... */}
```

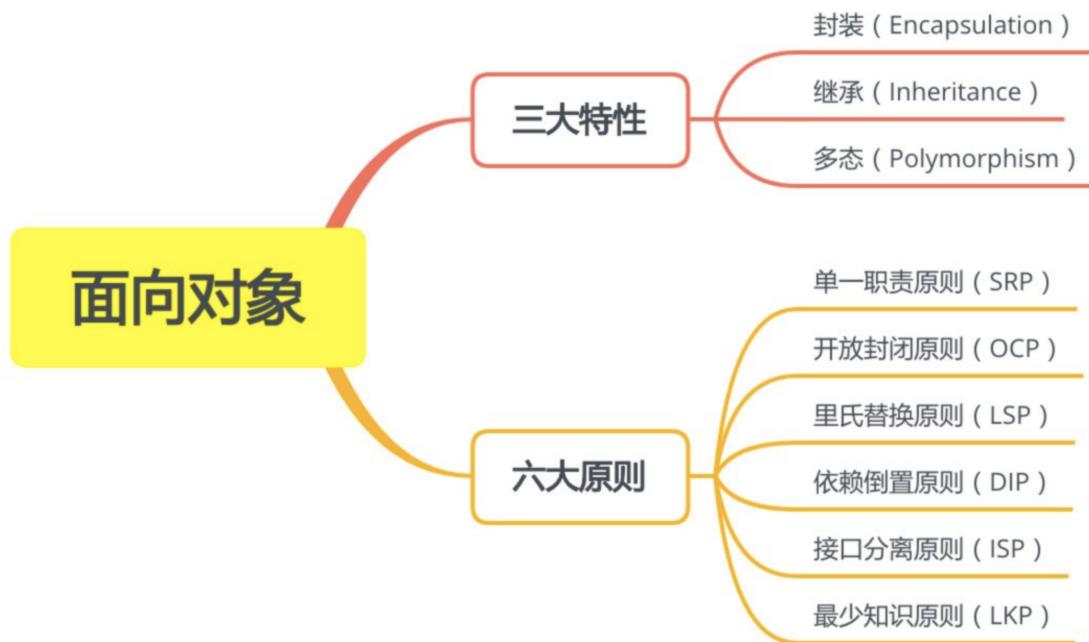
注意，`Rest` 参数只能放在最后一个，否则会报错：

```
// 报错 SyntaxError: Rest 参数必须是最后一个参数
function func1(...rest, a){ /* ... */}
```

形参名并不必须是 `rest`，也可以是其它名称，使用者可以根据自己的习惯来命名

#继承与原型链

JavaScript 是一种灵活的语言，兼容并包含面向对象风格、函数式风格等编程风格。我们知道面向对象风格有三大特性和六大原则，三大特性是封装、继承、多态，六大原则是单一职责原则 (SRP)、开放封闭原则 (OCP)、里氏替换原则 (LSP)、依赖倒置原则 (DIP)、接口分离原则 (ISP)、最少知识原则 (LKP)。



`JavaScript` 并不是强面向对象语言，因此它的灵活性决定了并不是所有面向对象的特征都适合 `JavaScript` 开发，本教程将会着重介绍三大特性中的继承，和六大原则里的单一职责原则、开放封闭原则、最少知识原则

1. 原型对象链

JavaScript 内建的继承方法被称为原型对象链，又称为原型对象继承。对于一个对象，因为它继承了它的原型对象的属性，所以它可以访问到这些属性。同理，原型对象也是一个对象，它也有自己的原型对象，因此也可以继承它的原型对象的属性。

这就是原型继承链：对象继承其原型对象，而原型对象继承它的原型对象，以此类推。

2. 对象继承

使用对象字面量形式创建对象时，会隐式指定 `Object.prototype` 为新对象的 `[[Prototype]]`。使用 `Object.create()` 方式创建对象时，可以显式指定新对象的 `[[Prototype]]`。该方法接受两个参数：第一个参数为新对象的 `[[Prototype]]`，第二个参数描述了新对象的属性，格式如在 `Object.defineProperties()` 中使用的一样。

```
// 对象字面量形式，原型被隐式地设置为 Object.prototype
var rectangle = { sizeType: '四边形' }

// Object.create() 创建，显示指定为 Object.prototype，等价于 ↑
var rectangle = Object.create(Object.prototype, {
    sizeType: {
        configurable: true,
        enumerable: true,
        value: '四边形',
        writable: true
    }
})
```

我们可以用这个方法来实现对象继承：

```
var rectangle = {
    sizeType: '四边形',
    getSize: function() {
        console.log(this.sizeType)
    }
}

var square = Object.create(rectangle, {
    sizeType: { value: '正方形' }
})

rectangle.getSize() // "四边形"
square.getSize() // "正方形"

console.log(rectangle.hasOwnProperty('getSize')) // true
console.log(rectangle.isPrototypeOf(square)) // true
console.log(square.hasOwnProperty('getSize')) // false
console.log('getSize' in square) // true

console.log(square.__proto__ === rectangle) // true
console.log(square.__proto__.__proto__ === Object.prototype) // true
```



- 对象 `square` 继承自对象 `rectangle`，也就继承了 `rectangle` 的 `sizeType` 属性和 `getSize()` 方法，又通过重写 `sizeType` 属性定义了一个自有属性，隐藏并替代了原型对象中的同名属性。所以 `rectangle.getSize()` 输出「四边形」而 `square.getSize()` 输出「正方形」。
- 在访问一个对象的时候，JavaScript 引擎会执行一个搜索过程，如果在对象实例上发现该属性，该属性值就会被使用，如果没有发现则搜索其原型对象 `[[Prototype]]`，如果仍然没有发现，则继续搜索该原型对象的原型对象 `[[Prototype]]`，直到继承链顶端，顶端通常是一个 `Object.prototype`，其 `[[prototype]]` 为 `null`。这就是原型链的查找过程。
- 可以通过 `Object.create()` 创建 `[[Prototype]]` 为 `null` 的对象：`var obj = Object.create(null)`。对象 `obj` 是一个没有原型链的对象，这意味着 `toString()` 和 `valueOf` 等存在于 `Object` 原型上的方法同样不存在于该对象上，通常我们将这样创建出来的对象为纯净对象。

3. 原型链继承

- JavaScript 中的对象继承是构造函数继承的基础，几乎所有的函数都有 `prototype` 属性（通过 `Function.prototype.bind` 方法构造出来的函数是个例外），它可以被替换和修改。
- 函数声明创建函数时，函数的 `prototype` 属性被自动设置为一个继承自 `Object.prototype` 的对象，该对象有个自有属性 `constructor`，其值就是函数本身。

```
// 构造函数
function YourConstructor() {}

// JavaScript 引擎在背后做的：
YourConstructor.prototype = Object.create(Object.prototype, {
  constructor: {
    configurable: true,
    enumerable: true,
    value: YourConstructor,
    writable: true
  }
})

console.log(YourConstructor.prototype.__proto__ === Object.prototype) // true
```

JavaScript 引擎帮你把构造函数的 `prototype` 属性设置为一个继承自 `Object.prototype` 的对象，这意味着我们创建出来的构造函数都继承自 `Object.prototype`。由于 `prototype` 可以被赋值和改写，所以通过改写它来改变原型链：

```
/* 四边形 */
function Rectangle(length, width) {
  this.length = length // 长
  this.width = width // 宽
}

/* 获取面积 */
Rectangle.prototype.getArea = function() {
  return this.length * this.width
}

/* 获取尺寸信息 */
Rectangle.prototype.getSize = function() {
  console.log(`Rectangle: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}
```

```

}

/* 正方形 */
function Square(size) {
    this.length = size
    this.width = size
}

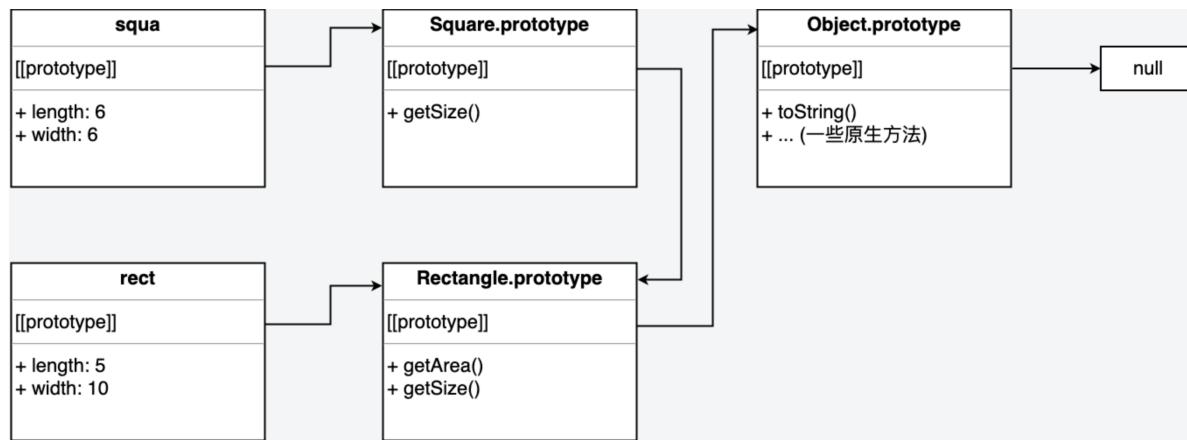
Square.prototype = new Rectangle()
Square.prototype.constructor = Square // 原本为 Rectangle, 重置回 Square 构造函数

Square.prototype.getSize = function() {
    console.log(`Square: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}

var rect = new Rectangle(5, 10)
var squa = new Square(6)

rect.getSize()          // Rectangle: 5x10, 面积: 50
squa.getSize()          // Square: 6x6, 面积: 36

```



- 为什么使用 `Square.prototype = new Rectangle()` 而不用 `Square.prototype = Rectangle.prototype` 呢。这是因为后者使得两个构造函数的 `prototype` 指向了同一个对象，当修改其中一个函数的 `prototype` 时，另一个函数也会受影响。
- 所以 `Square` 构造函数的 `prototype` 属性被改写为了 `Rectangle` 的一个实例。
- 但是仍然有问题。当一个属性只存在于构造函数的 `prototype` 上，而构造函数本身没有时，该属性会在构造函数的所有实例间共享，其中一个实例修改了该属性，其他所有实例都会受影响：

```

/* 四边形 */
function Rectangle(sizes) {
    this.sizes = sizes
}

/* 正方形 */
function Square() {}

Square.prototype = new Rectangle([1, 2])

var squa1 = new Square() // sizes: [1, 2]

squa1.sizes.push(3)     // 在 squa1 中修改了 sizes

```

```
console.log(squa1.sizes) // sizes: [1, 2, 3]

var squa2 = new Square()

console.log(squa2.sizes) // sizes: [1, 2, 3] 应该是 [1, 2], 得到的是修改后的 sizes
```

4. 构造函数窃取

构造函数窃取又称构造函数借用、经典继承。这种技术的基本思想相当简单，即在子类型构造函数的内部调用父类构造函数。

```
function getArea() {
    return this.length * this.width
}

/* 四边形 */
function Rectangle(length, width) {
    this.length = length
    this.width = width
}

/* 获取面积 */
Rectangle.prototype.getArea = getArea

/* 获取尺寸信息 */
Rectangle.prototype.getSize = function() {
    console.log(`Rectangle: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}

/* 正方形 */
function Square(size) {
    Rectangle.call(this, size, size)

    this.getArea = getArea

    this.getSize = function() {
        console.log(`Square: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
    }
}

var rect = new Rectangle(5, 10)
var squa = new Square(6)

rect.getSize()      // Rectangle: 5x10, 面积: 50
squa.getSize()     // Square: 6x6, 面积: 36
```

- 这样的实现避免了引用类型的属性被所有实例共享的问题，在父类实例创建时还可以自定义地传参，缺点是方法都是在构造函数中定义，每次创建实例都会重新赋值一遍方法，即使方法的引用是一致的。
- 这种方式通过构造函数窃取来设置属性，模仿了那些基于类的语言的类继承，所以这通常被称为伪类继承或经典继承。

5. 组合继承

组合继承又称伪经典继承，指的是将原型链和借用构造函数的技术组合发挥二者之长的一种继承模式。其背后的思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又能够保证每个实例都有它自己的属性。

```
/* 四边形 */
function Rectangle(length, width) {
    this.length = length
    this.width = width
    this.color = 'red'
}

/* 获取面积 */
Rectangle.prototype.getArea = function() {
    return this.length * this.width
}

/* 获取尺寸信息 */
Rectangle.prototype.getSize = function() {
    console.log(`Rectangle: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}

/* 正方形 */
function Square(size) {
    Rectangle.call(this, size, size) // 第一次调用 Rectangle 函数
    this.color = 'blue'
}

Square.prototype = new Rectangle() // 第二次调用 Rectangle 函数
Square.prototype.constructor = Square

Square.prototype.getSize = function() {
    console.log(`Square: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}

var rect = new Rectangle(5, 10)
var squa = new Square(6)

rect.getSize() // Rectangle: 5x10, 面积: 50
squa.getSize() // Square: 6x6, 面积: 36
```

组合继承是 JavaScript 中最常用的继承模式，但是父类构造函数被调用了两次。

6. 寄生组合式继承

```
/* 实现继承逻辑 */
function inheritPrototype(sub, sup) {
    var prototype = Object.create(sup.prototype)
    prototype.constructor = sub
    sub.prototype = prototype
}

/* 四边形 */
function Rectangle(length, width) {
```

```

        this.length = length
        this.width = width
        this.color = 'red'
    }

/* 获取面积 */
Rectangle.prototype.getArea = function() {
    return this.length * this.width
}

/* 获取尺寸信息 */
Rectangle.prototype.getSize = function() {
    console.log(`Rectangle: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}

/* 正方形 */
function Square(size) {
    Rectangle.call(this, size, size) // 第一次调用 Rectangle 函数
    this.color = 'blue'
}

// 实现继承
inheritPrototype(Square, Rectangle)

Square.prototype.getSize = function() {
    console.log(`Square: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
}

var rect = new Rectangle(5, 10)
var squa = new Square(6)

rect.getSize()      // Rectangle: 5x10, 面积: 50
squa.getSize()     // Square: 6x6, 面积: 36

```

- 这种方式的高效率体现它只调用了一次父类构造函数，并且因此避免了在 `Rectangle.prototype` 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变。因此，还能够正常使用 `instanceof` 和 `isPrototypeOf`。开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。
- 不过这种实现有些麻烦，推介使用组合继承和下面的 ES6 方式实现继承。

7. ES6 的 `extends` 方式实现继承

ES6 中引入了 `class` 关键字，`class` 之间可以通过 `extends` 关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰、方便和语义化的多。

```

/* 四边形 */
class Rectangle {
    constructor(length, width) {
        this.length = length
        this.width = width
        this.color = 'red'
    }

/* 获取面积 */
getArea() {

```

```

        return this.length * this.width
    }

    /* 获取尺寸信息 */
    getSize() {
        console.log(`Rectangle: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
    }
}

/* 正方形 */
class Square extends Rectangle {
    constructor(size) {
        super(size, size)
        this.color = 'blue'
    }

    getSize() {
        console.log(`Square: ${this.length}x${this.width}, 面积: ${this.getArea()}`)
    }
}

var rect = new Rectangle(5, 10)
var squa = new Square(6)

rect.getSize()      // Rectangle: 5x10, 面积: 50
squa.getSize()     // Square: 6x6, 面积: 36

```

然而并不是所有浏览器都支持 `class/extends` 关键词，不过我们可以引入 `Babel` 来进行转译。
`class` 语法实际上也是之前语法的语法糖，用户可以把上面的代码放到 Babel 的在线编译中看看，编译出来是什么样子

#设计原则

在前文我们介绍了面向对象三大特性之继承，本文将主要介绍面向对象六大原则中的单一职责原则（SRP）、开放封闭原则（OCP）、最少知识原则（LKP）。

设计原则是指导思想，从思想上给我们指明程序设计的正确方向，是我们在开发设计过程中应该尽力遵守的准则。而设计模式是实现手段，因此设计模式也应该遵守这些原则，或者说，设计模式就是这些设计原则的一些具体体现。要达到的目标就是高内聚低耦合，高内聚是说模块内部要高度聚合，是模块内部的关系，低耦合是说模块与模块之间的耦合度要尽量低，是模块与模块间的关系。

注意，遵守设计原则是好，但是过犹不及，在实际项目中我们不要刻板遵守，需要根据实际情况灵活运用

1. 单一职责原则 SRP

- 单一职责原则（Single Responsibility Principle, SRP）是指对一个类（方法、对象，下文统称对象）来说，应该仅有一个引起它变化的原因。也就是说，一个对象只做一件事。
- 单一职责原则可以让我们对对象的维护变得简单，如果一个对象具有多个职责的话，那么如果一个职责的逻辑需要修改，那么势必会影响到其他职责的代码。如果一个对象具有多种职责，职责之间相互耦合，对一个职责的修改会影响到其他职责的实现，这就是属于模块内低内聚高耦合的情况。负责的职责越多，耦合越强，对模块的修改就越来越危险。

优点：

- 降低单个类（方法、对象）的复杂度，提高可读性和可维护性，功能之间的界限更清晰；类（方法、对象）之间根据功能被分为更小的粒度，有助于代码的复用；
- 缺点：增加系统中类（方法、对象）的个数，实际上也增加了这些对象之间相互联系的难度，同时也引入了额外的复杂度。

2. 开放封闭原则 OCP

开放封闭原则（Open - Close Principle, OCP）是指一个模块在扩展性方面应该是开放的，而在更改性方面应该是封闭的，也就是对扩展开放，对修改封闭。

当需要增加需求的时候，则尽量通过扩展新代码的方式，而不是修改已有代码。因为修改已有代码，则会给依赖原有代码的模块带来隐患，因此修改之后需要把所有依赖原有代码的模块都测试一遍，修改一遍测试一遍，带来的成本很大，如果是上线的大型项目，那么代价和风险可能更高。

优点：

- 增加可维护性，避免因为修改给系统带来的不稳定性。

3. 最少知识原则 LKP

- 最少知识原则（Least Knowledge Principle, LKP）又称为迪米特原则（Law of Demeter, LOD），一个对象应该对其他对象有最少的了解。
- 通俗地讲，一个类应该对自己需要耦合或调用的类知道得最少，类的内部如何实现、如何复杂都与调用者或者依赖者没关系，调用者或者依赖者只需要知道他需要的方法即可，其他的我一概不关心。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。
- 通常为了减少对象之间的联系，是通过引入一个第三者来帮助进行通信，阻隔对象之间的直接通信，从而减少耦合。

优点：

- 降低类（方法、对象）之间不必要的依赖，减少耦合。

缺点：

- 类（方法、对象）之间不直接通信也会经过一个第三者来通信，那么就要权衡引入第三者带来的复杂度是否值得。

#二、创建型模式

#单例模式

- 单例模式可能是设计模式里面最简单的模式了，虽然简单，但在我们日常生活和编程中却经常接触到，本节我们一起来学习一下。
- 单例模式（Singleton Pattern）又称为单体模式，保证一个类只有一个实例，并提供一个访问它的全局访问点。也就是说，第二次使用同一个类创建新对象的时候，应该得到与第一次创建的对象完全相同的对象。

1. 你曾经遇见过的单例模式

- 当我们在电脑上玩经营类的游戏，经过一番眼花缭乱的骚操作好不容易走上正轨，夜深了我们去休息，第二天打开电脑，发现要从头玩，立马就把电脑扔窗外了，所以一般希望从前一天的进度接着打，这里就用到了存档。每次玩这游戏的时候，我们都希望拿到同一个存档接着玩，这就是属于单例模式的一个实例。
- 编程中也有很多对象我们只需要唯一一个，比如数据库连接、线程池、配置文件缓存、浏览器中的window/document等，如果创建多个实例，会带来资源耗费严重，或访问行为不一致等情况。
- 类似于数据库连接实例，我们可能频繁使用，但是创建它所需要的开销又比较大，这时只使用一个数据库连接就可以节约很多开销。一些文件的读取场景也类似，如果文件比较大，那么文件读取就

是一个比较重的操作。比如这个文件是一个配置文件，那么完全可以将读取到的文件内容缓存一份，每次来读取的时候访问缓存即可，这样也可以达到节约开销的目的。

在类似场景中，这些例子有以下特点：

- 每次访问者来访问，返回的都是同一个实例；
- 如果一开始实例没有创建，那么这个特定类需要自行创建这个实例；

2. 实例的代码实现

- 如果你是一个前端er，那么你肯定知道浏览器中的 `window` 和 `document` 全局变量，这两个对象都是单例，任何时候访问他们都是一样的对象，`window` 表示包含 `DOM` 文档的窗口，`document` 是窗口中载入的 `DOM` 文档，分别提供了各自相关的方法。
- 在 ES6 新增语法的 `Module` 模块特性，通过 `import/export` 导出模块中的变量是单例的，也就是说，如果在某个地方改变了模块内部变量的值，别的地方再引用的这个值是改变之后的。除此之外，项目中的全局状态管理模式 Vuex、Redux、MobX 等维护的全局状态，`vue-router`、`react-router` 等维护的路由实例，在单页应用的单页面中都属于单例的应用（但不属于单例模式的应用）。
- 在 JavaScript 中使用字面量方式创建一个新对象时，实际上没有其他对象与其类似，因为新对象已经是单例了：

```
{ a: 1 } === { a: 1 }      // false
```

- 那么问题来了，如何对构造函数使用 `new` 操作符创建多个对象时，仅获取同一个单例对象呢。
- 对于刚刚打经营游戏的例子，我们可以用 JavaScript 来实现一下：

```
function ManageGame() {
    if (ManageGame._schedule) {           // 判断是否已经有单例了
        return ManageGame._schedule
    }
    ManageGame._schedule = this
}

ManageGame.getInstance = function() {
    if (ManageGame._schedule) {           // 判断是否已经有单例了
        return ManageGame._schedule
    }
    return ManageGame._schedule = new ManageGame()
}

const schedule1 = new ManageGame()
const schedule2 = ManageGame.getInstance()

console.log(schedule1 === schedule2)
```

稍微解释一下，这个构造函数在内部维护（或者直接挂载自己身上）一个实例，第一次执行 `new` 的时候判断这个实例有没有创建过，创建过就直接返回，否则走创建流程。我们可以用 `ES6` 的 `class` 语法改造一下：

```
class ManageGame {
    static _schedule = null

    static getInstance() {
        if (ManageGame._schedule) {           // 判断是否已经有单例了
            return ManageGame._schedule
        }
    }
}
```

```

    }
    return ManageGame._schedule = new ManageGame()
}

constructor() {
    if (ManageGame._schedule) {           // 判断是否已经有单例了
        return ManageGame._schedule
    }
    ManageGame._schedule = this
}

const schedule1 = new ManageGame()
const schedule2 = ManageGame.getInstance()

console.log(schedule1 === schedule2)    // true

```

上面方法的缺点在于维护的实例作为静态属性直接暴露，外部可以直接修改。

3. 单例模式的通用实现

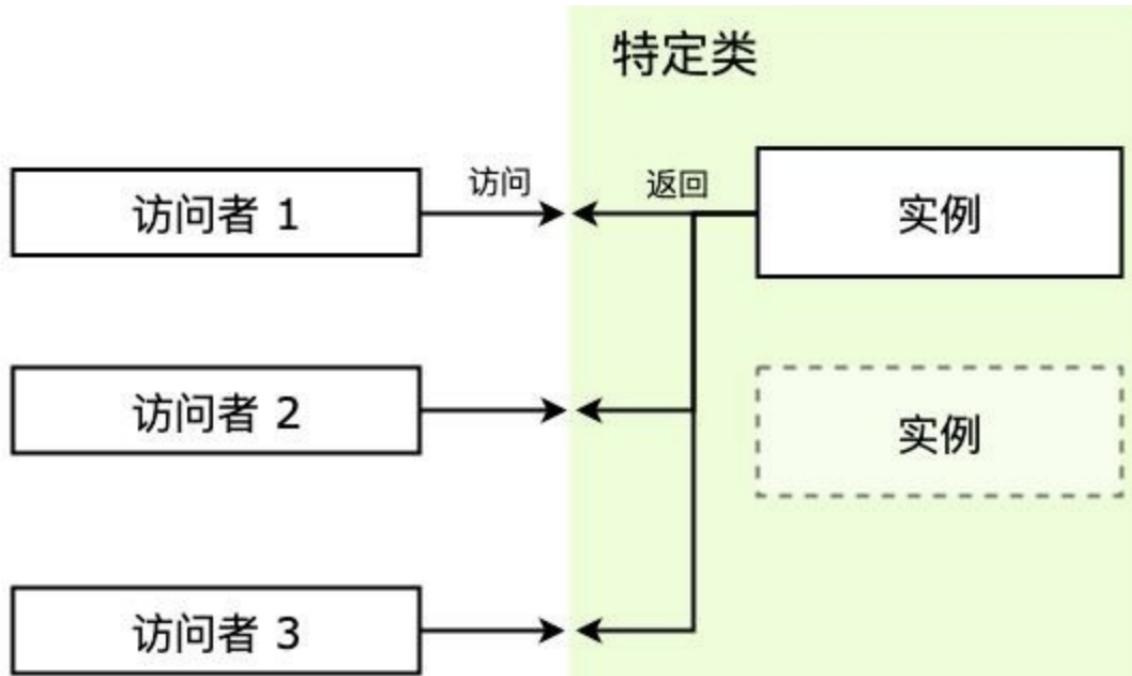
根据上面的例子提炼一下单例模式，游戏可以被认为是一个特定的类 (Singleton)，而存档是单例 (instance)，每次访问特定类的时候，都会拿到同一个实例。主要有下面几个概念：

- `singleton`：特定类，这是我们需要访问的类，访问者要拿到的是它的实例；
- `instance`：单例，是特定类的实例，特定类一般会提供 `getInstance` 方法来获取该单例；
- `getInstance`：获取单例的方法，或者直接由 `new` 操作符获取；

这里有几个实现点要关注一下：

- 访问时始终返回的是同一个实例；
- 自行实例化，无论是一开始加载的时候就创建好，还是在第一次被访问时；
- 一般还会提供一个 `getInstance` 方法用来获取它的实例；

结构大概如下图：



下面使用通用的方法来实现一下。

3.1 IIFE 方式创建单例模式

- 简单实现中，我们提到了缺点是实例会暴露，那么这里我们首先使用立即调用函数 IIFE 将不希望公开的单例实例 instance 隐藏。
- 当然也可以使用构造函数复写将闭包进行的更彻底，具体代码参看 Github 仓库，这里就不贴了。

```

const Singleton = (function() {
    let _instance = null           // 存储单例

    const Singleton = function() {
        if (_instance) return _instance // 判断是否已有单例
        _instance = this
        this.init()                  // 初始化操作
        return _instance
    }

    Singleton.prototype.init = function() {
        this.foo = 'Singleton Pattern'
    }
}

return Singleton
})()

const visitor1 = new Singleton()
const visitor2 = new Singleton()

console.log(visitor1 === visitor2) // true

```

- 这样一来，虽然仍使用一个变量 `_instance` 来保存单例，但是由于在闭包的内部，所以外部代码无法直接修改。
- 在这个基础上，我们可以继续改进，增加 `getInstance` 静态方法：

```

const Singleton = (function() {
    let _instance = null           // 存储单例

    const Singleton = function() {
        if (_instance) return _instance // 判断是否已有单例
        _instance = this
        this.init()                  // 初始化操作
        return _instance
    }

    Singleton.prototype.init = function() {
        this.foo = 'Singleton Pattern'
    }
}

Singleton.getInstance = function() {
    if (_instance) return _instance
    _instance = new Singleton()
    return _instance
}

return Singleton
})()

const visitor1 = new Singleton()
const visitor2 = new Singleton()           // 既可以 new 获取单例
const visitor3 = Singleton.getInstance() // 也可以 getInstance 获取单例

```

```
console.log(visitor1 === visitor2) // true
console.log(visitor1 === visitor3) // true
```

- 代价和上例一样是闭包开销，并且因为 IIFE 操作带来了额外的复杂度，让可读性变差。
- IIFE 内部返回的 Singleton 才是我们真正需要的单例的构造函数，外部的 Singleton 把它和一些单例模式的创建逻辑进行了一些封装。
- IIFE 方式除了直接返回一个方法/类实例之外，还可以通过模块模式的方式来进行，就不贴代码了，代码实现在 Github 仓库中，读者可以自己瞅瞅。

3.2 块级作用域方式创建单例

IIFE 方式本质还是通过函数作用域的方式来隐藏内部作用域的变量，有了 ES6 的 `let/const` 之后，可以通过 `{ }` 块级作用域的方式来隐藏内部变量：

```
let getInstance

{
    let _instance = null // 存储单例

    const Singleton = function() {
        if (_instance) return _instance // 判断是否已有单例
        _instance = this
        this.init() // 初始化操作
        return _instance
    }

    Singleton.prototype.init = function() {
        this.foo = 'Singleton Pattern'
    }
}

getInstance = function() {
    if (_instance) return _instance
    _instance = new Singleton()
    return _instance
}
}

const visitor1 = getInstance()
const visitor2 = getInstance()

console.log(visitor1 === visitor2)
```

输出: `true` 怎么样，是不是对块级作用域的理解更深了呢~

3.3 单例模式赋能

之前的例子中，单例模式的创建逻辑和原先这个类的一些功能逻辑（比如 `init` 等操作）混杂在一起，根据单一职责原则，这个例子我们还可以继续改进一下，将单例模式的创建逻辑和特定类的功能逻辑拆开，这样功能逻辑就可以和正常的类一样。

```
/* 功能类 */
class FuncClass {
    constructor(bar) {
        this.bar = bar
        this.init()
    }
}
```

```

    init() {
        this.foo = 'Singleton Pattern'
    }
}

/* 单例模式的赋能类 */
const Singleton = (function() {
    let _instance = null // 存储单例

    const Proxysingleton = function(bar) {
        if (_instance) return _instance // 判断是否已有单例
        _instance = new FuncClass(bar)
        return _instance
    }

    Proxysingleton.getInstance = function(bar) {
        if (_instance) return _instance
        _instance = new Singleton(bar)
        return _instance
    }

    return Proxysingleton
})()

const visitor1 = new Singleton('单例1')
const visitor2 = new Singleton('单例2')
const visitor3 = Singleton.getInstance()

console.log(visitor1 === visitor2) // true
console.log(visitor1 === visitor3) // true

```

- 这样的单例模式赋能类也可被称为代理类，将业务类和单例模式的逻辑解耦，把单例的创建逻辑抽象封装出来，有利于业务类的扩展和维护。代理的概念我们将在后面代理模式的章节中更加详细地探讨。
- 使用类似的概念，配合 ES6 引入的 `Proxy` 来拦截默认的 `new` 方式，我们可以写出更简化的单例模式赋能方法：

```

/* Person 类 */
class Person {
    constructor(name, age) {
        this.name = name
        this.age = age
    }
}

/* 单例模式的赋能方法 */
function Singleton(FuncClass) {
    let _instance
    return new Proxy(FuncClass, {
        construct(target, args) {
            return _instance || (_instance = Reflect.construct(FuncClass, args))
        }
    })
}

```

```
const PersonInstance = Singleton(Person)

const person1 = new PersonInstance('张小帅', 25)
const person2 = new PersonInstance('李小美', 23)

console.log(person1 === person2) // true
```

4. 惰性单例、懒汉式-饿汉式

- 有时候一个实例化过程比较耗费性能的类，但是却一直用不到，如果一开始就对这个类进行实例化就显得有些浪费，那么这时我们就可以使用惰性创建，即延迟创建该类的单例。之前的例子都属于惰性单例，实例的创建都是 `new` 的时候才进行。

惰性单例又被成为懒汉式，相对应的概念是饿汉式：

- 懒汉式单例是在使用时才实例化
- 饿汉式是当程序启动时或单例模式类一加载的时候就被创建。
- 我们可以举一个简单的例子比较一下：

```
class FuncClass {
    constructor() { this.bar = 'bar' }
}

// 饿汉式
const HungrySingleton = (function() {
    const _instance = new FuncClass()

    return function() {
        return _instance
    }
})()

// 懒汉式
const LazySingleton = (function() {
    let _instance = null

    return function() {
        return _instance || (_instance = new FuncClass())
    }
})()

const visitor1 = new HungrySingleton()
const visitor2 = new HungrySingleton()
const visitor3 = new LazySingleton()
const visitor4 = new LazySingleton()

console.log(visitor1 === visitor2) // true
console.log(visitor3 === visitor4) // true
```

可以打上 `debugger` 在控制台中看一下，饿汉式在 `HungrySingleton` 这个 IIFE 执行的时候就进入到 `FuncClass` 的实例化流程了，而懒汉式的 `LazySingleton` 中 `FuncClass` 的实例化过程是在第一次 `new` 的时候才进行的。

惰性创建在实际开发中使用很普遍，了解一下对以后的开发工作很有帮助。

5. 源码中的单例模式

以 ElementUI 为例，ElementUI 中的全屏 Loading 蒙层调用有两种形式：

```
// 1. 指令形式
Vue.use(Loading.directive)
// 2. 服务形式
Vue.prototype.$loading = service
```

- 上面的是指令形式注册，使用的方式 `<div :v-loading.fullscreen="true">...</div>`；
- 下面的是服务形式注册，使用的方式 `this.$loading({ fullscreen: true })`；

用服务方式使用全屏 Loading 是单例的，即在前一个全屏 Loading 关闭前再次调用全屏 Loading，并不会创建一个新的 Loading 实例，而是返回现有全屏 Loading 的实例。

下面我们可以看看 ElementUI 2.9.2 的源码是如何实现的，为了观看方便，省略了部分代码：

```
import Vue from 'vue'
import LoadingVue from './loading.vue'

const LoadingConstructor = Vue.extend(LoadingVue)

let fullscreenLoading

const Loading = (options = {}) => {
  if (options.fullscreen && fullscreenLoading) {
    return fullscreenLoading
  }

  let instance = new LoadingConstructor({
    el: document.createElement('div'),
    data: options
  })

  if (options.fullscreen) {
    fullscreenLoading = instance
  }
  return instance
}

export default Loading
```

- 这里的单例是 `fullscreenLoading`，是存放在闭包中的，如果用户传的 `options` 的 `fullscreen` 为 `true` 且已经创建了单例的情况下则会直接返回之前创建的单例，如果之前没有创建过，则创建单例并赋值给闭包中的 `fullscreenLoading` 后返回新创建的单例实例。
- 这是一个典型的单例模式的应用，通过复用之前创建的全屏蒙层单例，不仅减少了实例化过程，而且避免了蒙层叠加蒙层出现的底色变深的情况。

6. 单例模式的优缺点

单例模式主要解决的问题就是节约资源，保持访问一致性。

简单分析一下它的优点：

- 单例模式在创建后在内存中只存在一个实例，节约了内存开支和实例化时的性能开支，特别是需要重复使用一个创建开销比较大的类时，比起实例不断地销毁和重新实例化，单例能节约更多资源，比如数据库连接；

- 单例模式可以解决对资源的多重占用，比如写文件操作时，因为只有一个实例，可以避免对一个文件进行同时操作；
- 只使用一个实例，也可以减小垃圾回收机制 GC (Garbage Collection) 的压力，表现在浏览器中就是系统卡顿减少，操作更流畅，CPU 资源占用更少；

单例模式也是有缺点的

- 单例模式对扩展不友好，一般不容易扩展，因为单例模式一般自行实例化，没有接口；
- 与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化；

. 单例模式的使用场景

那我们应该在什么场景下使用单例模式呢：

- 当一个类的实例化过程消耗的资源过多，可以使用单例模式来避免性能浪费；
- 当项目中需要一个公共的状态，那么需要使用单例模式来保证访问一致性；

#工厂模式

工厂模式 (Factory Pattern)，根据不同的输入返回不同类的实例，一般用来创建同一类对象。
工厂方式的主要思想是将对象的创建与对象的实现分离。

1. 你曾见过的工厂模式

今天你的老同学找你来玩，你决定下个馆子（因为不会做饭），于是你来到了小区门口的饭店，跟老板说，来一份鱼香肉丝，一份宫保鸡丁。等会儿菜就烧好端到你的面前，不用管菜烧出来的过程，你只要负责吃就行了。

上面这两个例子都是工厂模式的实例，老板相当于工厂，负责生产产品，访问者通过老板就可以拿到想要的产品。

在类似场景中，这些例子有以下特点：

- 访问者只需要知道产品名，就可以从工厂获得对应实例；
- 访问者不关心实例创建过程；

2. 实例的代码实现

如果你使用过 `document.createElement` 方法创建过 DOM 元素，那么你已经使用过工厂方法了，虽然这个方法实际上很复杂，但其使用的就是工厂方法的思想：访问者只需提供标签名（如 `div`、`img`），那么这个方法就会返回对应的 DOM 元素。

我们可以使用 JavaScript 将上面饭馆例子实现一下：

```
/* 饭店方法 */
function restaurant(menu) {
    switch (menu) {
        case '鱼香肉丝':
            return new YuXiangRouSi()
        case '宫保鸡丁':
            return new GongBaoJiDin()
        default:
            throw new Error('这个菜本店没有 -。-')
    }
}

/* 鱼香肉丝类 */
function YuXiangRouSi() { this.type = '鱼香肉丝' }

YuXiangRouSi.prototype.eat = function() {
```

```

        console.log(this.type + ' 真香~')
    }

/* 宫保鸡丁类 */
function GongBaoJiDin() { this.type = '宫保鸡丁' }

GongBaoJiDin.prototype.eat = function() {
    console.log(this.type + ' 让我想起了外婆做的菜~')
}

const dish1 = restaurant('鱼香肉丝')
dish1.eat() // 输出: 鱼香肉丝 真香~
const dish2 = restaurant('红烧排骨') // 输出: Error 这个菜本店没有 -。-

```

工厂方法中这里使用 `switch-case` 语法，你也可以用 `if-else`，都可以。

下面使用 ES6 的 class 语法改写一下：

```

/* 饭店方法 */
class Restaurant {
    static getMenu(menu) {
        switch (menu) {
            case '鱼香肉丝':
                return new YuXiangRouSi()
            case '宫保鸡丁':
                return new GongBaoJiDin()
            default:
                throw new Error('这个菜本店没有 -。-')
        }
    }
}

/* 鱼香肉丝类 */
class YuXiangRousi {
    constructor() { this.type = '鱼香肉丝' }

    eat() { console.log(this.type + ' 真香~') }
}

/* 宫保鸡丁类 */
class GongBaoJiDin {
    constructor() { this.type = '宫保鸡丁' }

    eat() { console.log(this.type + ' 让我想起了外婆做的菜~') }
}

const dish1 = Restaurant.getMenu('鱼香肉丝')
dish1.eat() // 输出: 鱼香肉丝 真香~

const dish2 = Restaurant.getMenu('红烧排骨') // 输出: Error 这个菜本店没有 -。-

```

- 这样就完成了一个工厂模式，但是这个实现有一个问题：工厂方法中包含了很多与创建产品相关的过程，如果产品种类很多的话，这个工厂方法中就会罗列很多产品的创建逻辑，每次新增或删除产品种类，不仅要增加产品类，还需要对应修改在工厂方法，违反了开闭原则，也导致这个工厂方法变得臃肿、高耦合。

- 严格上这种实现在面向对象语言中叫做简单工厂模式。适用于产品种类比较少，创建逻辑不复杂的时候使用。
- 工厂模式的本意是将实际创建对象的过程推迟到子类中，一般用抽象类来作为父类，创建过程由抽象类的子类来具体实现。JavaScript 中没有抽象类，所以我们可以简单地将工厂模式看做是一个实例化对象的工厂类即可。关于抽象类的有关内容，可以参看抽象工厂模式。
- 然而作为灵活的 JavaScript，我们不必如此较真，可以把易变的参数提取出来：

```

/* 饭店方法 */
class Restaurant {
    constructor() {
        this.menuData = {}
    }

    /* 创建菜品 */
    getMenu(menu) {
        if (!this.menuData[menu])
            throw new Error('这个菜本店没有 -。-')
        const { type, message } = this.menuData[menu]
        return new Menu(type, message)
    }

    /* 增加菜品种类 */
    addMenu(menu, type, message) {
        if (this.menuData[menu]) {
            console.info('已经有这个菜了!')
            return
        }
        this.menuData[menu] = { type, message }
    }

    /* 移除菜品 */
    removeMenu(menu) {
        if (!this.menuData[menu]) return
        delete this.menuData[menu]
    }
}

/* 菜品类 */
class Menu {
    constructor(type, message) {
        this.type = type
        this.message = message
    }

    eat() { console.log(this.type + this.message) }
}

const restaurant = new Restaurant()
restaurant.addMenu('YuXiangRouSi', '鱼香肉丝', '真香~')           // 注册菜品
restaurant.addMenu('GongBaoJiDin', '宫保鸡丁', '让我想起了外婆做的菜~')

const dish1 = restaurant.getMenu('YuXiangRouSi')
dish1.eat()
// 输出：鱼香肉丝 真香~
const dish2 = restaurant.getMenu('HongSaoPaiGu')      // 输出：Error 这个菜本店没有
-。-

```

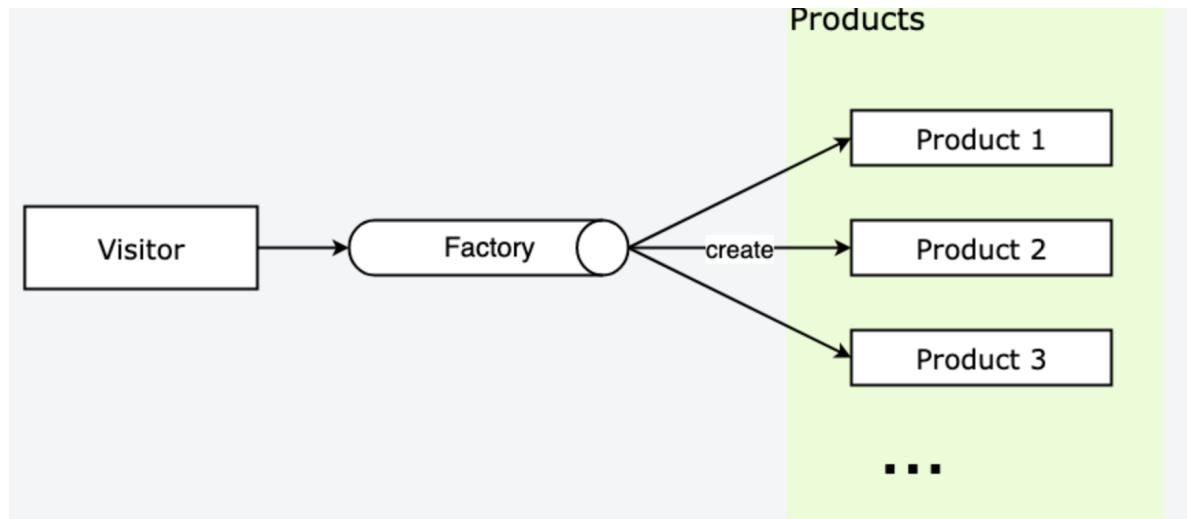
- 我们还给 Restaurant 类增加了 `addMenu/removeMenu` 私有方法，以便于扩展。
- 当然这里如果菜品参数不太一致，可以在 `addMenu` 时候注册构造函数或者类，创建的时候返回 `new` 出的对应类实例，灵活变通即可。

3. 工厂模式的通用实现

根据上面的例子我们可以提炼一下工厂模式，饭店可以被认为是工厂类（Factory），菜品是产品（Product），如果我们希望获得菜品实例，通过工厂类就可以拿到产品实例，不用关注产品实例创建流程。主要有下面几个概念：

- Factory：工厂，负责返回产品实例；
- Product：产品，访问者从工厂拿到产品实例；

结构大概如下：



下面用通用的方法实现，这里直接用 class 语法：

```

/* 工厂类 */
class Factory {
    static getInstance(type) {
        switch (type) {
            case 'Product1':
                return new Product1()
            case 'Product2':
                return new Product2()
            default:
                throw new Error('当前没有这个产品')
        }
    }
}

/* 产品类1 */
class Product1 {
    constructor() { this.type = 'Product1' }

    operate() { console.log(this.type) }
}

/* 产品类2 */
class Product2 {
    constructor() { this.type = 'Product2' }

    operate() { console.log(this.type) }
}
  
```

```
}

const prod1 = Factory.getInstance('Product1')
prod1.operate()
// 输出: Product1
const prod2 = Factory.getInstance('Product3')          // 输出: Error 当前没有这个产品
```

注意，由于 JavaScript 的灵活，简单工厂模式返回的产品对象不一定非要是类实例，也可以是字面量形式的对象，所以读者可以根据场景灵活选择返回的产品对象形式。

4. 源码中的工厂模式

4.1 Vue/React 源码中的工厂模式

和原生的 `document.createElement` 类似，Vue 和 React 这种具有虚拟 DOM 树（Virtual Dom Tree）机制的框架在生成虚拟 DOM 的时候，都提供了 `createElement` 方法用来生成 `VNode`，用来作为真实 DOM 节点的映射：

```
// Vue
createElement('h3', { class: 'main-title' }, [
  createElement('img', { class: 'avatar', attrs: { src: '../avatar.jpg' } }),
  createElement('p', { class: 'user-desc' }, '长得帅气的快，长得丑活得久')
])

// React
React.createElement('h3', { className: 'user-info' },
  React.createElement('img', { src: '../avatar.jpg', className: 'avatar' }),
  React.createElement('p', { className: 'user-desc' }, '长得帅气的快，长得丑活得久')
)
```

`createElement` 函数结构大概如下：

```
class Vnode (tag, data, children) { ... }

function createElement(tag, data, children) {
  return new Vnode(tag, data, children)
}
```

可以看到 `createElement` 函数内会进行 `VNode` 的具体创建，创建的过程是很复杂的，而框架提供的 `createElement` 工厂方法封装了复杂的创建与验证过程，对于使用者来说就很方便了。

4.2 vue-router 源码中的工厂模式

工厂模式在源码中应用频繁，以 `vue-router` 中的源码为例，代码位置：`vue-router/src/index.js`

```
// src/index.js
export default class VueRouter {
  constructor(options) {
    this.mode = mode      // 路由模式

    switch (mode) {        // 简单工厂
      case 'history':     // history 方式
        this.history = new HTML5History(this, options.base)
        break
      case 'hash':         // hash 方式
        this.hash = new HashHistory(this, options.base)
        break
    }
  }
}
```

```

        this.history = new HashHistory(this, options.base,
this.fallback)
            break
        case 'abstract':      // abstract 方式
            this.history = new AbstractHistory(this, options.base)
            break
        default:
            // ... 初始化失败报错
    }
}
}

```

稍微解释一下这里的源码。mode 是路由创建的模式，这里有三种 History、Hash、Abstract，前两种我们已经很熟悉了，History 是 H5 的路由方式，Hash 是路由中带 # 的路由方式，Abstract 代表非浏览器环境中路由方式，比如 Node、weex 等；this.history 用来保存路由实例，vue-router 中使用了工厂模式的思想来获得响应路由控制类的实例。

- 源码里没有把工厂方法的产品创建流程封装出来，而是直接将产品实例的创建流程暴露在 VueRouter 的构造函数中，在被 new 的时候创建对应产品实例，相当于 VueRouter 的构造函数就是一个工厂方法。
- 如果一个系统不是 SPA (Single Page Application, 单页应用)，而是是 MPA (Multi Page Application, 多页应用)，那么就需要创建多个 VueRouter 的实例，此时 VueRouter 的构造函数也就是工厂方法将会被多次执行，以分别获得不同实例。

5. 工厂模式的优缺点

工厂模式将对象的创建和实现分离，这带来了优点：

- 良好的封装，代码结构清晰，访问者无需知道对象的创建流程，特别是创建比较复杂的情况下；
- 扩展性优良，通过工厂方法隔离了用户和创建流程隔离，符合开放封闭原则；
- 解耦了高层逻辑和底层产品类，符合最少知识原则，不需要的就不要去交流；
- 工厂模式的缺点：带来了额外的系统复杂度，增加了抽象性；

6. 工厂模式的使用场景

那么什么时候使用工厂模式呢：

- 对象的创建比较复杂，而访问者无需知道创建的具体流程；
- 处理大量具有相同属性的小对象；

什么时候不该用工厂模式：滥用只是增加了不必要的系统复杂度，过犹不及。

7. 其他相关模式

7.1 工厂模式与抽象工厂模式

这两个方式可以组合使用，具体联系与区别在抽象工厂模式中讨论。

7.2 工厂模式与模板方法模式

这两个模式看起来比较类似，不过主要区别是：

- 工厂模式 主要关注产品实例的创建，对创建流程封闭起来；
- 模板方法模式 主要专注的是为固定的算法骨架提供某些步骤的实现；
- 这两个模式也可以组合一起来使用，比如在模板方法模式里面，使用工厂方法来创建模板方法需要的对象。

#抽象工厂模式

工厂模式（Factory Pattern），根据输入的不同返回不同类的实例，一般用来创建同一类对象。工厂方式的主要思想是将对象的创建与对象的实现分离。

- 抽象工厂（Abstract Factory）：通过对类的工厂抽象使其业务用于对产品类簇的创建，而不是负责创建某一类产品的实例。关键在于使用抽象类制定了实例的结构，调用者直接面向实例的结构编程，从实例的具体实现中解耦。
- 我们知道 JavaScript 并不是强面向对象语言，所以使用传统编译型语言比如 JAVA、C#、C++ 等实现的设计模式和 JavaScript 不太一样，比如 JavaScript 中没有原生的类和接口等（不过 ES6+ 渐渐提供类似的语法糖），我们可以用变通的方式来解决。最重要的是设计模式背后的核心思想，和它所要解决的问题。

1. 你曾见过的抽象工厂模式

还是使用上一节工厂模式中使用的饭店例子。

- 你再次来到了小区的饭店，跟老板说来一份鱼香肉丝，来一份宫保鸡丁，来一份番茄鸡蛋汤，来一份排骨汤（今天可能比较想喝汤）。无论什么样的菜，还是什么样的汤，他们都具有同样的属性，比如菜都可以吃，汤都可以喝。所以我们不论拿到什么菜，都可以吃，而不论拿到什么汤，都可以喝。对于饭店也一样，这个饭店可以做菜做汤，另一个饭店也可以，那么这两个饭店就具有同样的功能结构。
- 面的场景都是属于抽象工厂模式的例子。菜类属于抽象产品类，制定具体产品菜类所具备的属性，而饭店和之前的工厂模式一样，负责具体生产产品实例，访问者通过老板获取想拿的产品。只要我们点的是汤类，即使还没有被做出来，我们就知道是可以喝的。推广一下，饭店功能也可以被抽象（抽象饭店类），继承这个类的饭店实例都具有做菜和做汤的功能，这样也完成了抽象类对实例的结构约束。
- 在类似场景中，这些例子有特点：只要实现了抽象类的实例，都实现了抽象类制定的结构；

2. 实例的代码实现

我们知道 JavaScript 并不强面向对象，也没有提供抽象类（至少目前没有提供），但是可以模拟抽象类。用对 `new.target` 来判断 new 的类，在父类方法中 `throw new Error()`，如果子类中没有实现这个方法就会抛错，这样来模拟抽象类：

```
/* 抽象类，ES6 class 方式 */
class AbstractClass1 {
    constructor() {
        if (new.target === AbstractClass1) {
            throw new Error('抽象类不能直接实例化!')
        }
    }

    /* 抽象方法 */
    operate() { throw new Error('抽象方法不能调用!') }
}

/* 抽象类，ES5 构造函数方式 */
var AbstractClass2 = function () {
    if (new.target === AbstractClass2) {
        throw new Error('抽象类不能直接实例化!')
    }
}

/* 抽象方法，使用原型方式添加 */
AbstractClass2.prototype.operate = function () { throw new Error('抽象方法不能调用!') }
```

下面用 JavaScript 将上面介绍的饭店例子实现一下。

首先使用原型方式：

```
/* 饭店方法 */
function Restaurant() {}

Restaurant.orderDish = function(type) {
    switch (type) {
        case '鱼香肉丝':
            return new YuXiangRouSi()
        case '宫保鸡丁':
            return new GongBaoJiDing()
        case '紫菜蛋汤':
            return new ZiCaiDanTang()
        default:
            throw new Error('本店没有这个 -。-')
    }
}

/* 菜品抽象类 */
function Dish() { this.kind = '菜' }

/* 抽象方法 */
Dish.prototype.eat = function() { throw new Error('抽象方法不能调用!') }

/* 鱼香肉丝类 */
function YuXiangRouSi() { this.type = '鱼香肉丝' }

YuXiangRouSi.prototype = new Dish()

YuXiangRouSi.prototype.eat = function() {
    console.log(this.kind + ' - ' + this.type + ' 真香~')
}

/* 宫保鸡丁类 */
function GongBaoJiDing() { this.type = '宫保鸡丁' }

GongBaoJiDing.prototype = new Dish()

GongBaoJiDing.prototype.eat = function() {
    console.log(this.kind + ' - ' + this.type + ' 让我想起了外婆做的菜~')
}

const dish1 = Restaurant.orderDish('鱼香肉丝')
dish1.eat()
const dish2 = Restaurant.orderDish('红烧排骨')

// 输出: 菜 - 鱼香肉丝 真香~
// 输出: Error 本店没有这个 -。-
使用 class 语法改写一下:
```

```
/* 饭店方法 */
class Restaurant {
    static orderDish(type) {
        switch (type) {
            case '鱼香肉丝':
```

```

        return new YuXiangRousi()
    case '宫保鸡丁':
        return new GongBaoJiDin()
    default:
        throw new Error('本店没有这个 -。-')
    }
}

/*
菜品抽象类 */
class Dish {
    constructor() {
        if (new.target === Dish) {
            throw new Error('抽象类不能直接实例化!')
        }
        this.kind = '菜'
    }

    /* 抽象方法 */
    eat() { throw new Error('抽象方法不能调用!') }
}

/*
鱼香肉丝类 */
class YuXiangRousi extends Dish {
    constructor() {
        super()
        this.type = '鱼香肉丝'
    }

    eat() { console.log(this.kind + ' - ' + this.type + ' 真香~') }
}

/*
宫保鸡丁类 */
class GongBaoJiDin extends Dish {
    constructor() {
        super()
        this.type = '宫保鸡丁'
    }

    eat() { console.log(this.kind + ' - ' + this.type + ' 让我想起了外婆做的菜~') }
}

const dish0 = new Dish() // 输出: Error 抽象方法不能调用!
const dish1 = Restaurant.orderDish('鱼香肉丝')
dish1.eat()
// 输出: 菜 - 鱼香肉丝 真香~
const dish2 = Restaurant.orderDish('红烧排骨') // 输出: Error 本店没有这个 -。-

```

- 这里的 Dish 类就是抽象产品类，继承该类的子类需要实现它的方法 eat。
- 上面的实现将产品的功能结构抽象出来成为抽象产品类。事实上我们还可以更进一步，将工厂类也使用抽象类约束一下，也就是抽象工厂类，比如这个饭店可以做菜和汤，另一个饭店也可以做菜和汤，存在共同的功能结构，就可以将共同结构作为抽象类抽象出来，实现如下：

```

/*
饭店 抽象类，饭店都可以做菜和汤 */
class AbstractRestaurant {
    constructor() {

```

```
        if (new.target === AbstractRestaurant)
            throw new Error('抽象类不能直接实例化!')
        this.signborad = '饭店'
    }

    /* 抽象方法: 创建菜 */
    createDish() { throw new Error('抽象方法不能调用!') }

    /* 抽象方法: 创建汤 */
    createSoup() { throw new Error('抽象方法不能调用!') }
}

/* 具体饭店类 */
class Restaurant extends AbstractRestaurant {
    constructor() { super() }

    createDish(type) {
        switch (type) {
            case '鱼香肉丝':
                return new YuXiangRousi()
            case '宫保鸡丁':
                return new GongBaoJiDing()
            default:
                throw new Error('本店没这个菜')
        }
    }

    createSoup(type) {
        switch (type) {
            case '紫菜蛋汤':
                return new ZiCaiDanTang()
            default:
                throw new Error('本店没这个汤')
        }
    }
}

/* 菜 抽象类, 菜都有吃的功能 eat */
class AbstractDish {
    constructor() {
        if (new.target === AbstractDish) {
            throw new Error('抽象类不能直接实例化!')
        }
        this.kind = '菜'
    }

    /* 抽象方法 */
    eat() { throw new Error('抽象方法不能调用!') }
}

/* 菜 鱼香肉丝类 */
class YuXiangRousi extends AbstractDish {
    constructor() {
        super()
        this.type = '鱼香肉丝'
    }

    eat() { console.log(this.kind + ' - ' + this.type + ' 真香~') }
}
```

```

}

/* 菜 宫保鸡丁类 */
class GongBaoJiDing extends AbstractDish {
    constructor() {
        super()
        this.type = '宫保鸡丁'
    }

    eat() { console.log(this.kind + ' - ' + this.type + ' 让我想起了外婆做的菜~') }
}

/* 汤 抽象类，汤都有喝的功能 drink */
class AbstractSoup {
    constructor() {
        if (new.target === AbstractDish) {
            throw new Error('抽象类不能直接实例化!')
        }
        this.kind = '汤'
    }

    /* 抽象方法 */
    drink() { throw new Error('抽象方法不能调用!') }
}

/* 汤 紫菜蛋汤类 */
class ZiCaiDanTang extends AbstractSoup {
    constructor() {
        super()
        this.type = '紫菜蛋汤'
    }

    drink() { console.log(this.kind + ' - ' + this.type + ' 我从小喝到大~') }
}

const restaurant = new Restaurant()

const soup1 = restaurant.createSoup('紫菜蛋汤')
soup1.drink()
// 输出：汤 - 紫菜蛋汤 我从小喝到大~

const dish1 = restaurant.createDish('鱼香肉丝')
dish1.eat()
// 输出：菜 - 鱼香肉丝 真香~
const dish2 = restaurant.createDish('红烧排骨') // 输出：Error 本店没有这个 - -

```

这样如果创建新的饭店，新的饭店继承这个抽象饭店类，那么也要实现抽象饭店类，这样就都具有抽象饭店类制定的结构。

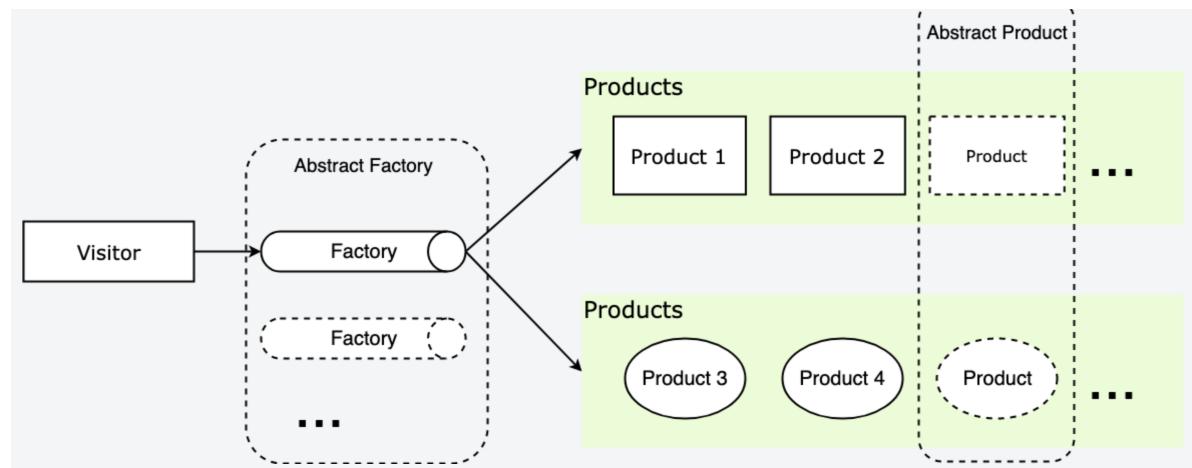
3. 抽象工厂模式的通用实现

我们提炼一下抽象工厂模式，饭店还是工厂（Factory），菜品种类是抽象类（AbstractFactory），而实现抽象类的菜品是具体的产品（Product），通过工厂拿到实现了不同抽象类的产品，这些产品可以根据实现的抽象类被区分为类簇。主要有下面几个概念：

- Factory：工厂，负责返回产品实例；
- AbstractFactory：虚拟工厂，制定工厂实例的结构；
- Product：产品，访问者从工厂中拿到的产品实例，实现抽象类；

- AbstractProduct：产品抽象类，由具体产品实现，制定产品实例的结构；

概略图如下：



下面是通用的实现，原型方式略过：

```

/* 工厂 抽象类 */
class AbstractFactory {
    constructor() {
        if (new.target === AbstractFactory)
            throw new Error('抽象类不能直接实例化!')
    }

    /* 抽象方法 */
    createProduct1() { throw new Error('抽象方法不能调用!') }
}

/* 具体饭店类 */
class Factory extends AbstractFactory {
    constructor() { super() }

    createProduct1(type) {
        switch (type) {
            case 'Product1':
                return new Product1()
            case 'Product2':
                return new Product2()
            default:
                throw new Error('当前没有这个产品 -。-')
        }
    }
}

/* 抽象产品类 */
class AbstractProduct {
    constructor() {
        if (new.target === AbstractProduct)
            throw new Error('抽象类不能直接实例化!')
        this.kind = '抽象产品类1'
    }

    /* 抽象方法 */
    operate() { throw new Error('抽象方法不能调用!') }
}
  
```

```

/* 具体产品类1 */
class Product1 extends AbstractProduct {
    constructor() {
        super()
        this.type = 'Product1'
    }

    operate() { console.log(this.kind + ' - ' + this.type) }
}

/* 具体产品类2 */
class Product2 extends AbstractProduct {
    constructor() {
        super()
        this.type = 'Product2'
    }

    operate() { console.log(this.kind + ' - ' + this.type) }
}

const factory = new Factory()

const prod1 = factory.createProduct1('Product1')
prod1.operate()
// 输出: 抽象产品类1 - Product1
const prod2 = factory.createProduct1('Product3')      // 输出: Error 当前没有这个产品
-。-

```

- 如果希望增加第二个类簇的产品，除了需要改一下对应工厂类之外，还需要增加一个抽象产品类，并在抽象产品类基础上扩展新的产品。
- 我们在实际使用的时候不一定需要每个工厂都继承抽象工厂类，比如只有一个工厂的话我们可以直接使用工厂模式，在实战中灵活使用。

4. 抽象工厂模式的优缺点

抽象模式的优点：

抽象产品类将产品的结构抽象出来，访问者不需要知道产品的具体实现，只需要面向产品的结构编程即可，从产品的具体实现中解耦；

抽象模式的缺点：

- 扩展新类簇的产品类比较困难，因为需要创建新的抽象产品类，并且还要修改工厂类，违反开闭原则；
- 带来了系统复杂度，增加了新的类，和新的继承关系；

5. 抽象工厂模式的使用场景

如果一组实例都有相同的结构，那么就可以使用抽象工厂模式。

6. 其他相关模式 6.1 抽象工厂模式与工厂模式

工厂模式和抽象工厂模式的区别：

- 工厂模式 主要关注单独的产品实例的创建；
- 抽象工厂模式 主要关注产品类簇实例的创建，如果产品类簇只有一个产品，那么这时的抽象工厂模式就退化为工厂模式了；根据场景灵活使用即可。

#建造者模式

建造者模式 (Builder Pattern) 又称生成器模式，分步构建一个复杂对象，并允许按步骤构造。同样的构建过程可以采用不同的表示，将一个复杂对象的构建层与其表示层分离。

- 在工厂模式中，创建的结果都是一个完整的个体，我们对创建的过程并不关心，只需了解创建的结果。而在建造者模式中，我们关心的是对象的创建过程，因此我们通常将创建的复杂对象的模块化，使得被创建的对象的每一个子模块都可以得到高质量的复用，当然在灵活的 JavaScript 中我们可以有更灵活的实现。

1. 你曾见过的建造者模式

- 假定我们需要建造一个车，车这个产品是由多个部件组成，车身、引擎、轮胎。汽车制造厂一般不会自己完成每个部件的制造，而是把部件的制造交给对应的汽车零部件制造商，自己只进行装配，最后生产出整车。整车的每个部件都是一个相对独立的个体，都具有自己的生产过程，多个部件经过一系列的组装共同组成了一个完整的车。
- 类似的场景还有很多，比如生产一个笔记本电脑，由主板、显示器、壳子组成，每个部件都有自己独立的行为和功能，他们共同组成了一个笔记本电脑。笔记本电脑厂从部件制造商处获得制造完成的部件，再由自己完成组装，得到笔记本电脑这个完整的产品。

在这些场景中，有以下特点：

- 整车制造厂（指挥者）无需知道零部件的生产过程，零部件的生产过程一般由零部件厂商（建造者）来完成；
- 整车制造厂（指挥者）决定以怎样的装配方式来组装零部件，以得到最终的产品；

2. 实例的代码实现

我们可以使用 JavaScript 来将上面的装配汽车的例子实现一下。

```
// 建造者，汽车部件厂家，提供具体零部件的生产
function CarBuilder({ color = 'white', weight = 0 }) {
    this.color = color
    this.weight = weight
}

// 生产部件，轮胎
CarBuilder.prototype.buildTyre = function(type) {
    switch (type) {
        case 'small':
            this.tyreType = '小号轮胎'
            this.tyreIntro = '正在使用小号轮胎'
            break
        case 'normal':
            this.tyreType = '中号轮胎'
            this.tyreIntro = '正在使用中号轮胎'
            break
        case 'big':
            this.tyreType = '大号轮胎'
            this.tyreIntro = '正在使用大号轮胎'
            break
    }
}

// 生产部件，发动机
CarBuilder.prototype.buildEngine = function(type) {
    switch (type) {
        case 'small':
```

```

        this.engineType = '小马力发动机'
        this.engineIntro = '正在使用小马力发动机'
        break
    case 'normal':
        this.engineType = '中马力发动机'
        this.engineIntro = '正在使用中马力发动机'
        break
    case 'big':
        this.engineType = '大马力发动机'
        this.engineIntro = '正在使用大马力发动机'
        break
    }
}

/* 奔驰厂家，负责最终汽车产品的装配 */
function benchiDirector(tyre, engine, param) {
    var _car = new CarBuilder(param)
    _car.buildTyre(tyre)
    _car.buildEngine(engine)
    return _car
}

// 获得产品实例
var benchi1 = benchiDirector('small', 'big', { color: 'red', weight: '1600kg' })

console.log(bENCHI1)

// 输出:
// {
//   color: "red"
//   weight: "1600kg"
//   tyre: Tyre {tyreType: "小号轮胎", tyreIntro: "正在使用小号轮胎"}
//   engine: Engine {engineType: "大马力发动机", engineIntro: "正在使用大马力发动机"}
// }

```

如果访问者希望获得另一个型号的车，比如有「空调」功能的车，那么我们只需要给 `CarBuilder` 的原型 `prototype` 上增加一个空调部件的建造方法，然后再新建一个新的奔驰厂家指挥者方法。

也可以使用 ES6 的写法改造一下：

```

// 建造者，汽车部件厂家，提供具体零部件的生产
class CarBuilder {
    constructor({ color = 'white', weight = 0 }) {
        this.color = color
        this.weight = weight
    }

    /* 生产部件，轮胎 */
    buildTyre(type) {
        const tyre = {}
        switch (type) {
            case 'small':
                tyre.tyreType = '小号轮胎'
                tyre.tyreIntro = '正在使用小号轮胎'
                break
            case 'normal':

```

```

        tyre.tyreType = '中号轮胎'
        tyre.tyreIntro = '正在使用中号轮胎'
        break
    case 'big':
        tyre.tyreType = '大号轮胎'
        tyre.tyreIntro = '正在使用大号轮胎'
        break
    }
    this.tyre = tyre
}

/* 生产部件，发动机 */
buildEngine(type) {
    const engine = {}
    switch (type) {
        case 'small':
            engine.engineType = '小马力发动机'
            engine.engineIntro = '正在使用小马力发动机'
            break
        case 'normal':
            engine.engineType = '中马力发动机'
            engine.engineIntro = '正在使用中马力发动机'
            break
        case 'big':
            engine.engineType = '大马力发动机'
            engine.engineIntro = '正在使用大马力发动机'
            break
    }
    this.engine = engine
}
}

/* 指挥者，负责最终汽车产品的装配 */
class BenChiDirector {
    constructor(tyre, engine, param) {
        const _car = new CarBuilder(param)
        _car.buildTire(tyre)
        _car.buildEngine(engine)
        return _car
    }
}

// 获得产品实例
const benchi1 = new BenChiDirector('small', 'big', { color: 'red', weight: '1600kg' })

console.log(benchi1)

// 输出:
// {
//   color: "red"
//   weight: "1600kg"
//   tyre: Tyre {tyreType: "小号轮胎", tyreIntro: "正在使用小号轮胎"}
//   engine: Engine {engineType: "大马力发动机", engineIntro: "正在使用大马力发动机"}
// }

```

作为灵活的JavaScript，我们还可以使用链模式来完成部件的装配，对链模式还不熟悉的同学可以看一下后面有一篇单独介绍链模式的文章～

```
// 建造者，汽车部件厂家
class CarBuilder {
    constructor({ color = 'white', weight = '0' }) {
        this.color = color
        this.weight = weight
    }

    /* 生产部件，轮胎 */
    buildTyre(type) {
        const tyre = {}
        switch (type) {
            case 'small':
                tyre.tyreType = '小号轮胎'
                tyre.tyreIntro = '正在使用小号轮胎'
                break
            case 'normal':
                tyre.tyreType = '中号轮胎'
                tyre.tyreIntro = '正在使用中号轮胎'
                break
            case 'big':
                tyre.tyreType = '大号轮胎'
                tyre.tyreIntro = '正在使用大号轮胎'
                break
        }
        this.tyre = tyre
        return this
    }

    /* 生产部件，发动机 */
    buildEngine(type) {
        const engine = {}
        switch (type) {
            case 'small':
                engine.engineType = '小马力发动机'
                engine.engineIntro = '正在使用小马力发动机'
                break
            case 'normal':
                engine.engineType = '中马力发动机'
                engine.engineIntro = '正在使用中马力发动机'
                break
            case 'big':
                engine.engineType = '大马力发动机'
                engine.engineIntro = '正在使用大马力发动机'
                break
        }
        this.engine = engine
        return this
    }
}

// 汽车装配，获得产品实例
const benchil = new CarBuilder({ color: 'red', weight: '1600kg' })
.builtTyre('small')
.buildEngine('big')
```

```

console.log(bench1)

// 输出:
// {
//   color: "red"
//   weight: "1600kg"
//   tyre: Tyre {tyre: "小号轮胎", tyreIntro: "正在使用小号轮胎"}
//   engine: Engine {engine: "大马力发动机", engineIntro: "正在使用大马力发动机"}
// }

```

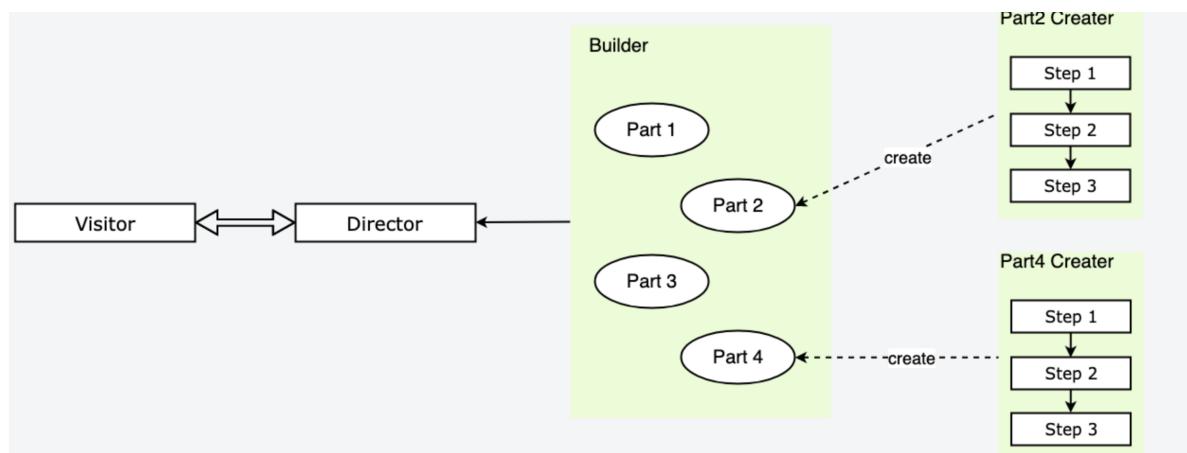
这样将最终产品的创建流程使用链模式来实现，相当于将指挥者退化，指挥的过程通过链模式让用户自己实现，这样既增加了灵活性，装配过程也一目了然。如果希望扩展产品的部件，那么在建造者上增加部件实现方法，再适当修改链模式即可。

3. 建造者模式的通用实现

我们提炼一下建造者模式，这里的生产汽车的奔驰厂家就相当于指挥者（Director），厂家负责将不同的部件组装成最后的产品（Product），而部件的生产者是部件厂家相当于建造者（Builder），我们通过指挥者就可以获得希望的复杂的产品对象，再通过访问不同指挥者获得装配方式不同的产品。主要有下面几个概念：

- Director：指挥者，调用建造者中的部件具体实现进行部件装配，相当于整车组装厂，最终返回装配完毕的产品；
- Builder：建造者，含有不同部件的生产方式给指挥者调用，是部件真正的生产者，但没有部件的装配流程；
- Product：产品，要返回给访问者的复杂对象；
- 建造者模式的主要功能是构建复杂的产品，并且是复杂的、需要分步骤构建的产品，其构建的算法是统一的，构建的过程由指挥者决定，只要配置不同的指挥者，就可以构建出不同的复杂产品来。也就是说，建造者模式将产品装配的算法和具体部件的实现分离，这样构建的算法可以扩展和复用，部件的具体实现也可以方便地扩展和复用，从而可以灵活地通过组合来构建出不同的产品对象。

概略图如下：



下面是通用的实现。

首先使用 ES6 的 class 语法：

```

// 建造者，部件生产
class ProductBuilder {
  constructor(param) {
    this.param = param
  }
}

```

```

/* 生产部件, part1 */
buildPart1() {
    // ... Part1 生产过程
    this.part1 = 'part1'

}

/* 生产部件, part2 */
buildPart2() {
    // ... Part2 生产过程
    this.part2 = 'part2'
}

}

/* 指挥者, 负责最终产品的装配 */
class Director {
    constructor(param) {
        const _product = new ProductBuilder(param)
        _product.buildPart1()
        _product.buildPart2()
        return _product
    }
}

```

// 获得产品实例
`const product = new Director('param')`

结合链模式:

```

// 建造者, 汽车部件厂家
class CarBuilder {
    constructor(param) {
        this.param = param
    }

    /* 生产部件, part1 */
    buildPart1() {
        this.part1 = 'part1'
        return this
    }

    /* 生产部件, part2 */
    buildPart2() {
        this.part2 = 'part2'
        return this
    }
}

// 汽车装配, 获得产品实例
const bench1 = new CarBuilder('param')
    .buildPart1()
    .buildPart2()

```

- 如果希望扩展实例的功能, 那么只需要在建造者类的原型上增加一个实例方法, 再返回 `this` 即可。
- 值得一提的是, 结合链模式的建造者模式中, 装配复杂对象的链式装配过程就是指挥者 `Director` 角色, 只不过在链式装配过程中不再封装在具体指挥者中, 而是由使用者自己确定装配过程。

4. 实战中的建造者模式

4.1 重构一个具有很多参数的构造函数

有时候你会遇到一个参数很多的构造函数，比如：

```
// 汽车建造者
class CarBuilder {
    constructor(engine, weight, height, color, tyre, name, type) {
        this.engine = engine
        this.weight = weight
        this.height = height
        this.color = color
        this.tyre = tyre
        this.name = name
        this.type = type
    }
}

const benchi = new CarBuilder('大马力发动机', '2ton', 'white', '大号轮胎', '奔驰',
'AMG')
```

如果构造函数的参数多于 3 个，在使用的时候就很容易弄不清哪个参数对应的是什么含义，你可以使用对象解构赋值的方式来提高可读性和使用便利性，也可以使用建造者模式的思想来进行属性赋值，这是另一个思路。代码如下：

```
// 汽车建造者
class CarBuilder {
    constructor(engine, weight, height, color, tyre, name, type) {
        this.engine = engine
        this.weight = weight
        this.height = height
        this.color = color
        this.tyre = tyre
        this.name = name
        this.type = type
    }

    setCarProperty(key, value) {
        if (Object.getOwnPropertyNames(this).includes(key)) {
            this[key] = value
            return this
        }
        throw new Error(`Key error : ${key} 不是本实例上的属性`)
    }
}

const benchi = new CarBuilder()
.setCarProperty('engine', '大马力发动机')
.setCarProperty('weight', '2ton')
.setCarProperty('height', '2000mm')
.setCarProperty('color', 'white')
.setCarProperty('tyre', '大号轮胎')
.setCarProperty('name', '奔驰')
.setCarProperty('type', 'AMG')
```

每个键都是用一个同样的方法来设置，或许你觉得不太直观，我们可以将设置每个属性的操作都单独列作为一个方法，这样可读性就更高了：

```
// 汽车建造者
class CarBuilder {
    constructor(engine, weight, height, color, tyre, name, type) {
        this.engine = engine
        this.weight = weight
        this.height = height
        this.color = color
        this.tyre = tyre
        this.name = name
        this.type = type
    }

    setPropertyFuncChain() {
        Object.getOwnPropertyNames(this)
            .forEach(key => {
                const funcName = 'set' + key.replace(/\w/g, str =>
str.toUpperCase())
                this[funcName] = value => {
                    this[key] = value
                    return this
                }
            })
        return this
    }
}

const benchi = new CarBuilder().setPropertyFuncchain()
.setEngine('大马力发动机')
.setWeight('2ton')
.setHeight('2000mm')
.setColor('white')
.setTyre('大号轮胎')
.setName('奔驰')
.setType('AMG')
```

4.2 重构 React 的书写形式

- 注意：这个方式不一定推荐，只是用来开阔视野。
- 当我们写一个 React 组件的时候，一般结构形式如下；

```
class ContainerComponent extends Component {
    componentDidMount() {
        this.props.fetchThings()
    }
    render() {
        return <PresentationalComponent {...this.props}>
    }
}

ContainerComponent.propTypes = {
    fetchThings: PropTypes.func.isRequired
}

const mapStateToProps = state => ({
```

```

    things: state.things
  })
  const mapDispatchToProps = dispatch => ({
    fetchThings: () => dispatch(fetchThings()),
    selectThing: id => dispatch(selectThing(id)),
    blowshitup: () => dispatch(blowshitup())
  })

  export default connect(
    mapStateToProps,
    mapDispatchToProps
  )(ContainerComponent)

```

通过建造者模式重构，我们可以将组件形式写成如下方式：

```

export default ComponentBuilder('ContainerComponent')
  .render(props => <PresentationalComponent {...props}>)
  .componentDidMount(props => props.fetchThings())
  .propTypes({
    fetchThings: PropTypes.func.isRequired
  })
  .mapStateToProps(state => ({
    things: state.things
  }))
  .mapDispatchToProps(dispatch => ({
    fetchThings: () => dispatch(fetchThings()),
    selectThing: id => dispatch(selectThing(id)),
    blowshitup: () => dispatch(blowshitup())
  }))
  .build()

```

5. 建造者模式的优缺点

建造者模式的优点：

- 使用建造者模式可以使产品的构建流程和产品的表现分离，也就是将产品的创建算法和产品组成的实现隔离，访问者不必知道产品部件实现的细节；
- 扩展方便，如果希望生产一个装配顺序或方式不同的新产品，那么直接新建一个指挥者即可，不用修改既有代码，符合开闭原则；
- 更好的复用性，建造者模式将产品的创建算法和产品组成的实现分离，所以产品创建的算法可以复用，产品部件的实现也可以复用，带来很大的灵活性；

建造者模式的缺点：

- 建造者模式一般适用于产品之间组成部件类似的情况，如果产品之间差异性很大、复用性不高，那么不要使用建造者模式；
- 实例的创建增加了许多额外的结构，无疑增加了许多复杂度，如果对象粒度不大，那么我们最好直接创建对象；

6. 建造者模式的适用场景

- 相同的方法，不同的执行顺序，产生不一样的产品时，可以采用建造者模式；
- 产品的组成部件类似，通过组装不同的组件获得不同产品时，可以采用建造者模式；

7. 其他相关模式

7.1 建造者模式与工厂模式

- 建造者模式和工厂模式最终都是创建一个完整的产品，但是在建造者模式中我们更关心对象创建的过程，将创建对象的方法模块化，从而更好地复用这些模块。
- 当然建造者模式与工厂模式也是可以组合使用的，比如建造者中一般会提供不同的部件实现，那么这里就可以使用工厂模式来提供具体的部件对象，再通过指挥者来进行装配。

7.2 建造者模式与模版方法模式

- 指挥者的实现可以和模版方法模式相结合。也就是说，指挥者中部件的装配过程，可以使用模版方法模式来固定装配算法，把部件实现方法分为模板方法和基本方法，进一步提取公共代码，扩展可变部分。
- 是否采用模版方法模式看具体场景，如果产品的部件装配顺序很明确，但是具体的实现是未知的、灵活的，那么你可以适当考虑是否应该将算法骨架提取出来。

#三、结构型模式

#代理模式

代理模式（Proxy Pattern）又称委托模式，它为目标对象创造了一个代理对象，以控制对目标对象的访问。

- 代理模式把代理对象插入到访问者和目标对象之间，从而为访问者对目标对象的访问引入一定的间接性。正是这种间接性，给了代理对象很多操作空间，比如在调用目标对象前和调用后进行一些预操作和后操作，从而实现新的功能或者扩展目标的功能。

1. 你曾见过的代理模式

明星总是有个助理，或者说经纪人，如果某导演来请这个明星演出，或者某个品牌来找明星做广告，需要经纪人帮明星做接洽工作。而且经纪人也起到过滤的作用，毕竟明星也不是什么电影和广告都会接。类似的场景还有很多，再比如领导和秘书... (emmm)

- 再看另一个例子。打官司是件非常麻烦的事，包括查找法律条文、起草法律文书、法庭辩论、签署法律文件、申请法院执行等等流程。此时，当事人就可聘请代理律师来完成整个打官司的所有事务。当事人只需与代理律师签订全权委托协议，那么整个打官司的过程，当事人都可以不用出现。法院的一些复杂事务都可以通过代理律师来完成，而法院需要当事人完成某些工作的时候，比如出庭，代理律师才会通知当事人，并为当事人出谋划策。

在类似的场景中，有以下特点：

- 导演/法院（访问者）对明星/当事人（目标）的访问都是通过经纪人/律师（代理）来完成；
- 经纪人/律师（代理）对访问有过滤的功能；

2. 实例的代码实现

我们使用 JavaScript 来将上面的明星例子实现一下。

```
/* 明星 */
var SuperStar = {
  name: '小鲜肉',
  playAdvertisement: function(ad) {
    console.log(ad)
  }
}

/* 经纪人 */
var ProxyAssistant = {
  name: '经纪人张某',
```

```

playAdvertisement: function(reward, ad) {
    if (reward > 1000000) { // 如果报酬超过100w
        console.log('没问题，我们小鲜鲜最喜欢拍广告了！')
        SuperStar.playAdvertisement(ad)
    } else
        console.log('没空，滚！')
}

ProxyAssistant.playAdvertisement(10000, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没空，滚

```

这里我们通过经纪人的方式来和明星取得联系，经纪人会视条件过滤一部分合作请求。

- 我们可以升级一下，比如如果明星没有档期的话，可以通过经纪人安排档期，当明星有空的时候才让明星来拍广告。这里通过 `Promise` 的方式来实现档期的安排：

```

/* 明星 */
const SuperStar = {
    name: '小鲜肉',
    playAdvertisement(ad) {
        console.log(ad)
    }
}

/* 经纪人 */
const ProxyAssistant = {
    name: '经纪人张某',
    scheduleTime() {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                console.log('小鲜鲜有空了')
                resolve()
            }, 2000) // 发现明星有空了
        })
    },
    playAdvertisement(reward, ad) {
        if (reward > 1000000) { // 如果报酬超过100w
            console.log('没问题，我们小鲜鲜最喜欢拍广告了！')
            ProxyAssistant.scheduleTime() // 安排上了
                .then(() => SuperStar.playAdvertisement(ad))
        } else
            console.log('没空，滚！')
    }
}

ProxyAssistant.playAdvertisement(10000, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没空，滚

ProxyAssistant.playAdvertisement(1000001, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没问题，我们小鲜鲜最喜欢拍广告了！
// 2秒后
// 输出： 小鲜鲜有空了
// 输出： 纯蒸酸牛奶，味道纯纯，尽享纯蒸

```

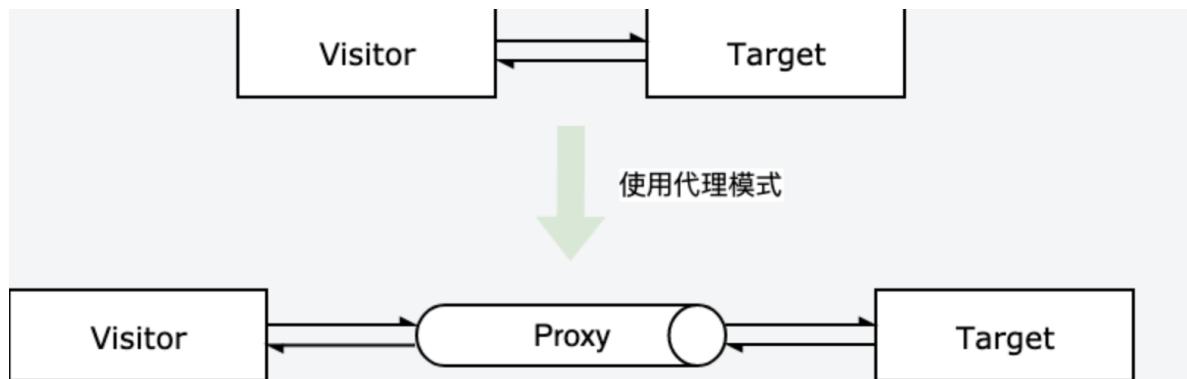
这里就简单实现了经纪人对请求的过滤，对明星档期的安排，实现了一个代理对象的基本功能。

3. 代理模式的概念

对于上面的例子，明星就相当于被代理的目标对象（`Target`），而经纪人就相当于代理对象（`Proxy`），希望找明星的人是访问者（`Visitor`），他们直接找不到明星，只能找明星的经纪人来进行行业务商洽。主要有以下几个概念：

- `Target`：目标对象，也是被代理对象，是具体业务的实际执行者；
- `Proxy`：代理对象，负责引用目标对象，以及对访问的过滤和预处理；

概略图如下：



ES6 原生提供了 `Proxy` 构造函数，这个构造函数让我们可以很方便地创建代理对象：

```
var proxy = new Proxy(target, handler);
```

参数中 `target` 是被代理对象，`handler` 用来设置代理行为。

这里使用 `Proxy` 来实现一下上面的经纪人例子：

```
/* 明星 */
const SuperStar = {
  name: '小鲜肉',
  scheduleFlag: false, // 档期标识位, false-没空(默认值), true-有空
  playAdvertisement(ad) {
    console.log(ad)
  }
}

/* 经纪人 */
const ProxyAssistant = {
  name: '经纪人张某',
  scheduleTime(ad) {
    const schedule = new Proxy(SuperStar, { // 在这里监听
      scheduleFlag 值的变化
      set(obj, prop, val) {
        if (prop !== 'scheduleFlag') return
        if (obj.scheduleFlag === false &&
          val === true) { // 小鲜肉现在有空了
          obj.scheduleFlag = true
          obj.playAdvertisement(ad) // 安排上了
        }
      }
    })
    setTimeout(() => {
      console.log('小鲜肉有空了')
    })
  }
}
```

```

        schedule.scheduleFlag = true           // 明星有空了
    }, 2000)
},
playAdvertisement(reward, ad) {
    if (reward > 1000000) {                // 如果报酬超过100w
        console.log('没问题，我们小鲜鲜最喜欢拍广告了！')
        ProxyAssistant.scheduleTime(ad)
    } else
        console.log('没空，滚！')
}
}

ProxyAssistant.playAdvertisement(10000, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没空，滚

ProxyAssistant.playAdvertisement(1000001, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没问题，我们小鲜鲜最喜欢拍广告了！
// 2秒后
// 输出： 小鲜鲜有空了
// 输出： 纯蒸酸牛奶，味道纯纯，尽享纯蒸
在 ES6 之前，一般是使用 Object.defineProperty 来完成相同的功能，我们可以使用这个 API 改造一下：

/* 明星 */
const SuperStar = {
    name: '小鲜肉',
    scheduleFlagActually: false,           // 档期标识位，false-没空（默认值）,
true-有空
    playAdvertisement(ad) {
        console.log(ad)
    }
}

/* 经纪人 */
const ProxyAssistant = {
    name: '经纪人张某',
    scheduleTime(ad) {
        Object.defineProperty(SuperStar, 'scheduleFlag', { // 在这里监听
scheduleFlag 值的变化
            get() {
                return SuperStar.scheduleFlagActually
            },
            set(val) {
                if (SuperStar.scheduleFlagActually === false &&
                    val === true) {                      // 小鲜肉现在有空了
                    SuperStar.scheduleFlagActually = true
                    SuperStar.playAdvertisement(ad)      // 安排上了
                }
            }
        })
        setTimeout(() => {
            console.log('小鲜鲜有空了')
            SuperStar.scheduleFlag = true
        }, 2000)                                // 明星有空了
    },
    playAdvertisement(reward, ad) {
        if (reward > 1000000) {                // 如果报酬超过100w
            console.log('没问题，我们小鲜鲜最喜欢拍广告了！')
            ProxyAssistant.scheduleTime(ad)
        }
    }
}

```

```

        console.log('没问题，我们小鲜鲜最喜欢拍广告了！')
        ProxyAssistant.scheduleTime(ad)
    } else
        console.log('没空，滚！')
    }

}

ProxyAssistant.playAdvertisement(10000, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没空，滚

ProxyAssistant.playAdvertisement(1000001, '纯蒸酸牛奶，味道纯纯，尽享纯蒸')
// 输出： 没问题，我们小鲜鲜最喜欢拍广告了！
// 2秒后
// 输出： 小鲜鲜有空了
// 输出： 纯蒸酸牛奶，味道纯纯，尽享纯蒸

```

4. 代理模式在实战中的应用 4.1 拦截器

上一小节使用代理模式代理对象的访问的方式，一般又被称为拦截器。

- 拦截器的思想在实战中应用非常多，比如我们在项目中经常使用 `Axios` 的实例来进行 `HTTP` 的请求，使用拦截器 `interceptor` 可以提前对 `request` 请求和 `response` 返回进行一些预处理，比如：
- `request` 请求头的设置，和 `Cookie` 信息的设置；
- 权限信息的预处理，常见的比如验权操作或者 `Token` 验证；
- 数据格式的格式化，比如对组件绑定的 `Date` 类型的数据在请求前进行一些格式约定好的序列化操作；
- 空字段的格式预处理，根据后端进行一些过滤操作；
- `response` 的一些通用报错处理，比如使用 `Message` 控件抛出错误；除了 `HTTP` 相关的拦截器之外，还有路由跳转的拦截器，可以进行一些路由跳转的预处理等操作。

4.2 前端框架的数据响应式化

- 现在的很多前端框架或者状态管理框架都使用上面介绍的 `Object.defineProperty` 和 `Proxy` 来实现数据的响应式化，比如 `Vue`、`Mobx`、`AvalonJS` 等，`Vue 2.x` 与 `AvalonJS` 使用前者，而 `Vue 3.x` 与 `Mobx 5.x` 使用后者。
- `Vue 2.x` 中通过 `Object.defineProperty` 来劫持各个属性的 `setter/getter`，在数据变动时，通过发布-订阅模式发布消息给订阅者，触发相应的监听回调，从而实现数据的响应式化，也就是数据到视图的双向绑定。

为什么 `Vue 2.x` 到 `3.x` 要从 `Object.defineProperty` 改用 `Proxy` 呢，是因为前者的一些局限性，导致的以下缺陷：

- 无法监听利用索引直接设置数组的一个项，例如：`vm.items[indexofItem] = newValue;`
- 无法监听数组的长度的修改，例如：`vm.items.length = newLength;`
- 无法监听 `ES6` 的 `Set`、`WeakSet`、`Map`、`WeakMap` 的变化；
- 无法监听 `Class` 类型的数据；
- 无法监听对象属性的新加或者删除；
- 除此之外还有性能上的差异，基于这些原因，`Vue 3.x` 改用 `Proxy` 来实现数据监听了。当然缺点就是对 `IE` 用户的不友好，兼容性敏感的场景需要做一些取舍。

4.3 缓存代理

在高阶函数的文章中，就介绍了备忘模式，备忘模式就是使用缓存代理的思想，将复杂计算的结果缓存起来，下次传参一致时直接返回之前缓存的计算结果。

4.4 保护代理和虚拟代理

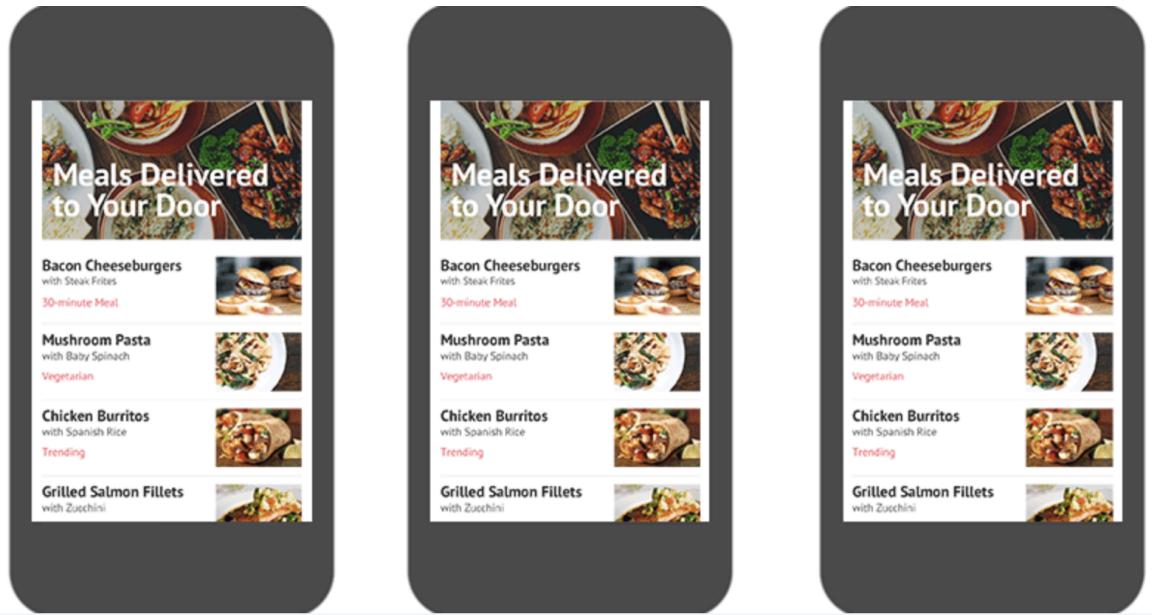
有的书籍中着重强调代理的两种形式：保护代理和虚拟代理：

- 保护代理：当一个对象可能会收到大量请求时，可以设置保护代理，通过一些条件判断对请求进行过滤；
- 虚拟代理：在程序中可能有一些代价昂贵的操作，此时可以设置虚拟代理，虚拟代理会在适合的时候才执行操作。

保护代理其实就是对访问的过滤，之前的经纪人例子就属于这种类型。

而虚拟代理是为一个开销很大的操作先占位，之后再执行，比如：

一个很大的图片加载前，一般使用菊花图、低质量图片等提前占位，优化图片加载导致白屏的情况；现在很流行的页面加载前使用骨架屏来提前占位，很多 `WebApp` 和 `NativeApp` 都采用这种方式来优化用户白屏体验



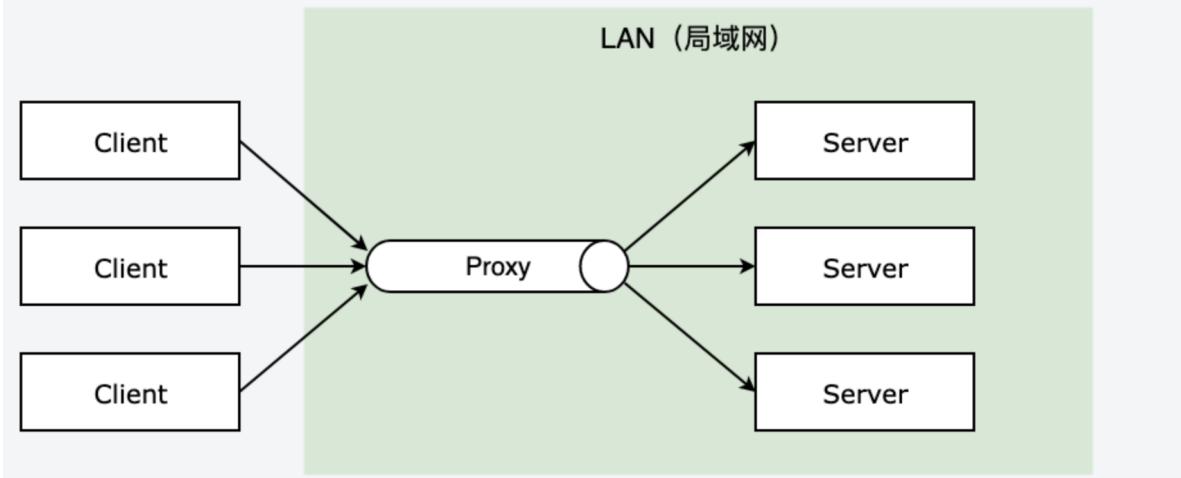
4.5 正向代理与反向代理

还有个经常用的例子是反向代理（Reverse Proxy），反向代理对应的是正向代理（Forward Proxy），他们的区别是：

- 正向代理：一般的访问流程是客户端直接向目标服务器发送请求并获取内容，使用正向代理后，客户端改为向代理服务器发送请求，并指定目标服务器（原始服务器），然后由代理服务器和原始服务器通信，转交请求并获得的内容，再返回给客户端。正向代理隐藏了真实的客户端，为客户端收发请求，使真实客户端对服务器不可见；
- 反向代理：与一般访问流程相比，使用反向代理后，直接收到请求的服务器是代理服务器，然后将请求转发给内部网络上真正进行处理的服务器，得到的结果返回给客户端。反向代理隐藏了真实的服务器，为服务器收发请求，使真实服务器对客户端不可见。

反向代理一般在处理跨域请求的时候比较常用，属于服务端开发人员的日常操作了，另外在缓存服务器、负载均衡服务器等等场景也是使用到代理模式的思想。

反向代理



5. 代理模式的优缺点

代理模式的主要优点有：

- 代理对象在访问者与目标对象之间可以起到中介和保护目标对象的作用；
- 代理对象可以扩展目标对象的功能；
- 代理模式能将访问者与目标对象分离，在一定程度上降低了系统的耦合度，如果我们希望适度扩展目标对象的一些功能，通过修改代理对象就可以了，符合开闭原则；
- 代理模式的缺点主要是增加了系统的复杂度，要斟酌当前场景是不是真的需要引入代理模式（十八线明星就别请经纪人了）

6. 其他相关模式

很多其他的模式，比如状态模式、策略模式、访问者模式其实也是使用了代理模式，包括在之前高阶函数处介绍的备忘模式，本质上也是一种缓存代理。

6.1 代理模式与适配器模式

代理模式和适配器模式都为另一个对象提供间接性的访问，他们的区别：

- 适配器模式：主要用来解决接口之间不匹配的问题，通常是为所适配的对象提供一个不同的接口；
- 代理模式：提供访问目标对象的间接访问，以及对目标对象功能的扩展，一般提供和目标对象一样的接口；

6.2 代理模式与装饰者模式

装饰者模式实现上和代理模式类似，都是在访问目标对象之前或者之后执行一些逻辑，但是目的和功能不同：

- 装饰者模式：目的是为了方便地给目标对象添加功能，也就是动态地添加功能；
- 代理模式：主要目的是控制其他访问者对目标对象的访问；

#享元模式

享元模式 (Flyweight Pattern) 运用共享技术来有效地支持大量细粒度对象的复用，以减少创建的对象的数量。

享元模式的主要思想是共享细粒度对象，也就是说如果系统中存在多个相同的对象，那么只需共享一份就可以了，不必每个都去实例化每一个对象，这样来精简内存资源，提升性能和效率。

Fly 意为苍蝇，Flyweight 指轻蝇量级，指代对象粒度很小。

1. 你曾见过的享元模式

我们去驾考的时候，如果给每个考试的人都准备一辆车，那考场就挤爆了，考点都堆不下考试车，因此驾考现场一般会有几辆车给要考试的人依次使用。如果考生人数少，就分别少准备几个自动档和手动档的驾考车，考生多的话就多准备几辆。如果考手动档的考生比较多，就多准备几辆手动档的驾考车。

我们去考四六级的时候（为什么这么多考试？），如果给每个考生都准备一个考场，怕是没那么多考场也没有这么多监考老师，因此现实中的大多数情况都是几十个考生共用一个考场。四级考试和六级考试一般同时进行，如果考生考的是四级，那么就安排四级考场，听四级的听力和试卷，六级同理。

生活中类似的场景还有很多，比如咖啡厅的咖啡口味，餐厅的菜品种类，拳击比赛的重量级等等。

在类似场景中，这些例子有以下特点：

- 目标对象具有一些共同的状态，比如驾考考生考的是自动档还是手动档，四六级考生考的是四级还是六级；
- 这些共同的状态所对应的对象，可以被共享出来；

2. 实例的代码实现

首先假设考生的 ID 为奇数则考的是手动档，为偶数则考的是自动档。如果给所有考生都 new 一个驾考车，那么这个系统中就会创建了和考生数量一致的驾考车对象：

```
var candidateNum = 10    // 考生数量
var examCarNum = 0        // 驾考车的数量

/* 驾考车构造函数 */
function ExamCar(carType) {
    examCarNum++
    this.carId = examCarNum
    this.carType = carType ? '手动档' : '自动档'
}

ExamCar.prototype.examine = function(candidateID) {
    console.log('考生- ' + candidateID + ' 在' + this.carType + '驾考车- ' +
    this.carId + ' 上考试')
}

for (var candidateID = 1; candidateID <= candidateNum; candidateID++) {
    var examCar = new ExamCar(candidateID % 2)
    examCar.examine(candidateID)
}

console.log('驾考车总数 - ' + examCarNum)
// 输出：驾考车总数 - 10
```

如果考生很多，那么系统中就会存在更多个驾考车对象实例，假如驾考车对象比较复杂，那么这些新建的驾考车实例就会占用大量内存。这时我们将同种类型的驾考车实例进行合并，手动档和自动档驾考车分别引用同一个实例，就可以节约大量内存：

```
var candidateNum = 10    // 考生数量
var examCarNum = 0        // 驾考车的数量

/* 驾考车构造函数 */
function ExamCar(carType) {
    examCarNum++
    this.carId = examCarNum
    this.carType = carType ? '手动档' : '自动档'
}
```

```

ExamCar.prototype.examine = function(candidateId) {
    console.log('考生- ' + candidateId + ' 在' + this.carType + '驾考车- ' +
    this.carId + ' 上考试')
}

var manualExamCar = new ExamCar(true)
var autoExamCar = new ExamCar(false)

for (var candidateId = 1; candidateId <= candidateNum; candidateId++) {
    var examCar = candidateId % 2 ? manualExamCar : autoExamCar
    examCar.examine(candidateId)
}

console.log('驾考车总数 - ' + examCarNum)
// 输出：驾考车总数 - 2

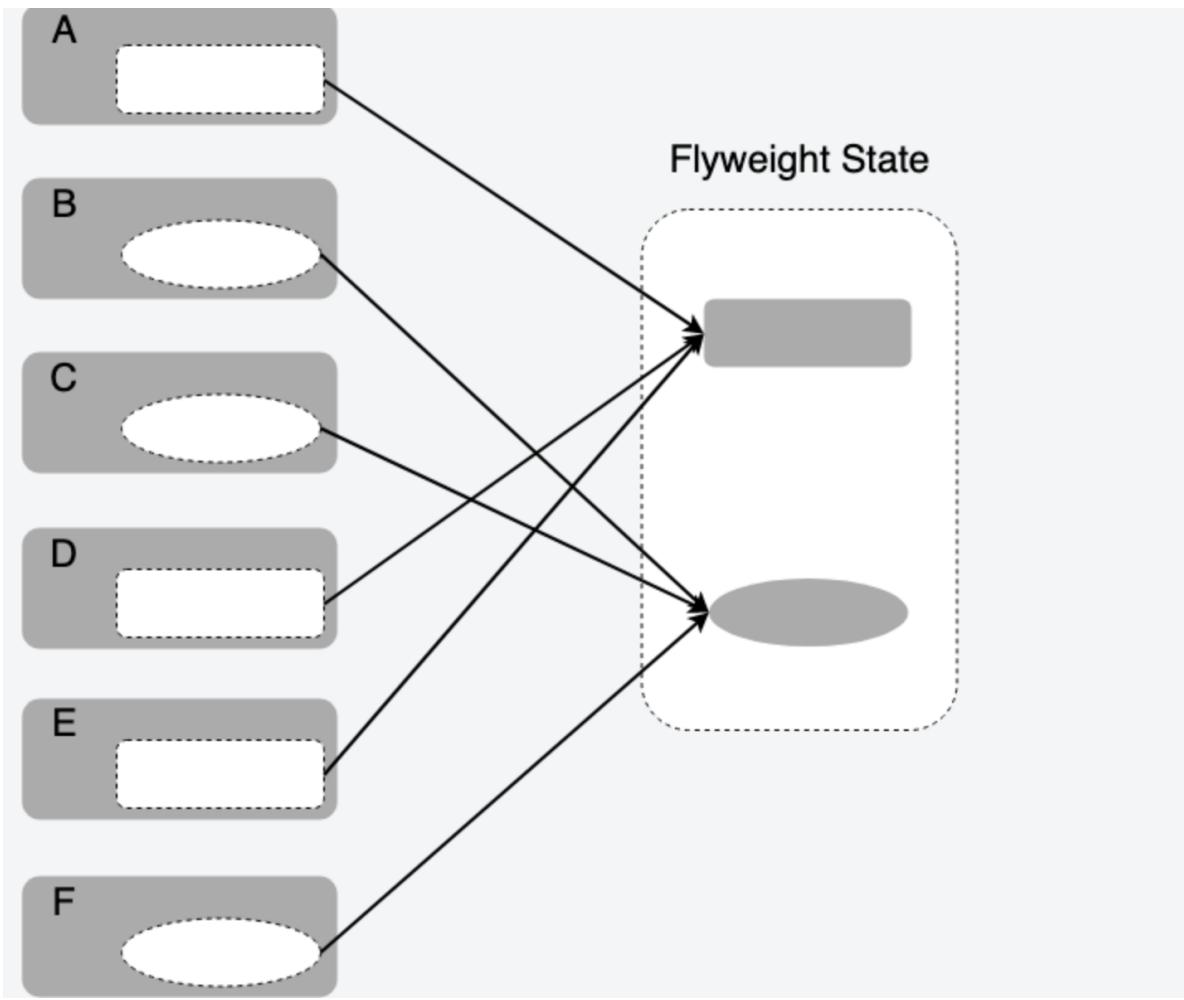
```

可以看到我们使用 2 个驾考车实例就实现了刚刚 10 个驾考车实例实现的功能。这是仅有 10 个考生的情况，如果有几百上千考生，这时我们节约的内存就比较可观了，这就是享元模式要达到的目的。

3. 享元模式改进

- 如果你阅读了之前文章关于继承部分的讲解，那么你实际上已经接触到享元模式的思想了。相比于构造函数窃取，在原型链继承和组合继承中，子类通过原型 prototype 来复用父类的方法和属性，如果子类实例每次都创建新的方法与属性，那么在子类实例很多的情况下，内存中就存在有很多重复的方法和属性，即使这些方法和属性完全一样，因此这部分内存完全可以通过复用来优化，这也是享元模式的思想。
- 传统的享元模式是将目标对象的状态区分为内部状态和外部状态，内部状态相同的对象可以被共享出来指向同一个内部状态。正如之前举的驾考和四六级考试的例子中，自动档还是手动档、四级还是六级，就属于驾考考生、四六级考生中的内部状态，对应的驾考车、四六级考场就是可以被共享的对象。而考生的年龄、姓名、籍贯等就属于外部状态，一般没有被共享出来的价值。

主要的原理可以参看下面的示意图：



- 享元模式的主要思想是细粒度对象的共享和复用，因此对之前的驾考例子，我们可以继续改进一下：
- 如果某考生正在使用一辆驾考车，那么这辆驾考车的状态就是被占用，其他考生只能选择剩下未被占用状态的驾考车；
- 如果某考生对驾考车的使用完毕，那么将驾考车开回考点，驾考车的状态改为未被占用，供给其他考生使用；
- 如果所有驾考车都被占用，那么其他考生只能等待正在使用驾考车的考生使用完毕，直到有驾考车的状态变为未被占用；
- 组织单位可以根据考生数量多准备几辆驾考车，比如手动档考生比较多，那么手动档驾考车就应该比自动档驾考车多准备几辆；
- 我们可以简单实现一下，为了方便起见，这里就直接使用 ES6 的语法。
- 首先创建 3 个手动档驾考车，然后注册 10 个考生参与考试，一开始肯定有 3 个考生同时上车，然后在某个考生考完之后其他考生接着后面考。为了实现这个过程，这里使用了 Promise，考试的考生在 0 到 2 秒后的随机时间考试完毕归还驾考车，其他考生在前面考生考完之后接着进行考试：

```

let examCarNum = 0 // 驾考车总数

/* 驾考车对象 */
class ExamCar {
  constructor(carType) {
    examCarNum++
    this.carId = examCarNum
    this.carType = carType ? '手动档' : '自动档'
    this.usingState = false // 是否正在使用
  }

  /* 在本车上考试 */
  examine(candidateId) {
    ...
  }
}

```

```

        return new Promise((resolve => {
            this.usingState = true
            console.log(`考生- ${candidateId} 开始在${this.carType}驾考车- ${this.carId} 上考试`)
            setTimeout(() => {
                this.usingState = false
                console.log(`%c考生- ${candidateId} 在${this.carType}驾考车- ${this.carId} 上考试完毕`, 'color:#f40')
                resolve() // 0~2秒后考试完毕
            }, Math.random() * 2000)
        }))
    }
}

/* 手动档汽车对象池 */
ManualExamCarPool = {
    _pool: [], // 驾考车对象池
    _candidateQueue: [], // 考生队列

    /* 注册考生 ID 列表 */
    registCandidates(candidateList) {
        candidateList.forEach(candidateId => this.registCandidate(candidateId)),
    }

    /* 注册手动档考生 */
    registCandidate(candidateId) {
        const examCar = this.getManualExamCar() // 找一个未被占用的手动档驾考车
        if (examCar) {
            examCar.examine(candidateId) // 开始考试, 考完了让队列中的下一个
            考生开始考试
            .then(() => {
                const nextCandidateId = this._candidateQueue.length &&
                this._candidateQueue.shift()
                nextCandidateId && this.registCandidate(nextCandidateId)
            })
        } else this._candidateQueue.push(candidateId),
    },
}

/* 注册手动档车 */
initManualExamCar(manualExamCarNum) {
    for (let i = 1; i <= manualExamCarNum; i++) {
        this._pool.push(new ExamCar(true))
    }
},
}

/* 获取状态为未被占用的手动档车 */
getManualExamCar() {
    return this._pool.find(car => !car.usingState)
}
}

ManualExamCarPool.initManualExamCar(3) // 一共有3个驾考车
ManualExamCarPool.registCandidates([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) // 10个考生来
考试

```

在浏览器中运行下试试：

The screenshot shows the Google Chrome DevTools Console tab. The title bar indicates the URL is DevTools - chrome-extension://lecdifefmfjnjinahaennhdilmcaeeeb/main.html. The console window displays the following JavaScript code:

```
registCandidate(candidateId) {
    const examCar = this.getManualExamCar() // 找一个未被占用的手动档驾考车
    if (examCar) {
        examCar.examine(candidateId) // 开始考试，考完了让队列中的下一个考生开始考试
        .then(() => {
            const nextCandidateId = this._candidateQueue.length && this._candidateQueue.shift()
            nextCandidateId && this.registCandidate(nextCandidateId)
        })
    } else this._candidateQueue.push(candidateId)
},
/**
 * 注册手动档车
 */
initManualExamCar(manualExamCarNum) {
    for (let i = 1; i <= manualExamCarNum; i++) {
        this._pool.push(new ExamCar(true))
    }
},
/**
 * 获取状态为未被占用的手动档车
 * @returns {*}
 */
getManualExamCar() {
    return this._pool.find(car => !car.usingState)
}
}

ManualExamCarPool.initManualExamCar(3) // 一共有3个驾考车
ManualExamCarPool.registCandidates([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) // 10个考生来考试
```

可以看到一个驾考的过程被模拟出来了，这里只简单实现了手动档，自动档驾考场景同理，就不进行实现了。上面的实现还可以进一步优化，比如考生多的时候自动新建驾考车，考生少的时候逐渐减少驾考车，但又不能无限新建驾考车对象，这些情况读者可以自行发挥～

- 如果可以将目标对象的内部状态和外部状态区分的比较明显，就可以将内部状态一致的对象很方便地共享出来，但是对 JavaScript 来说，我们并不一定要严格区分内部状态和外部状态才能进行资源共享，比如资源池模式。

4. 资源池 上

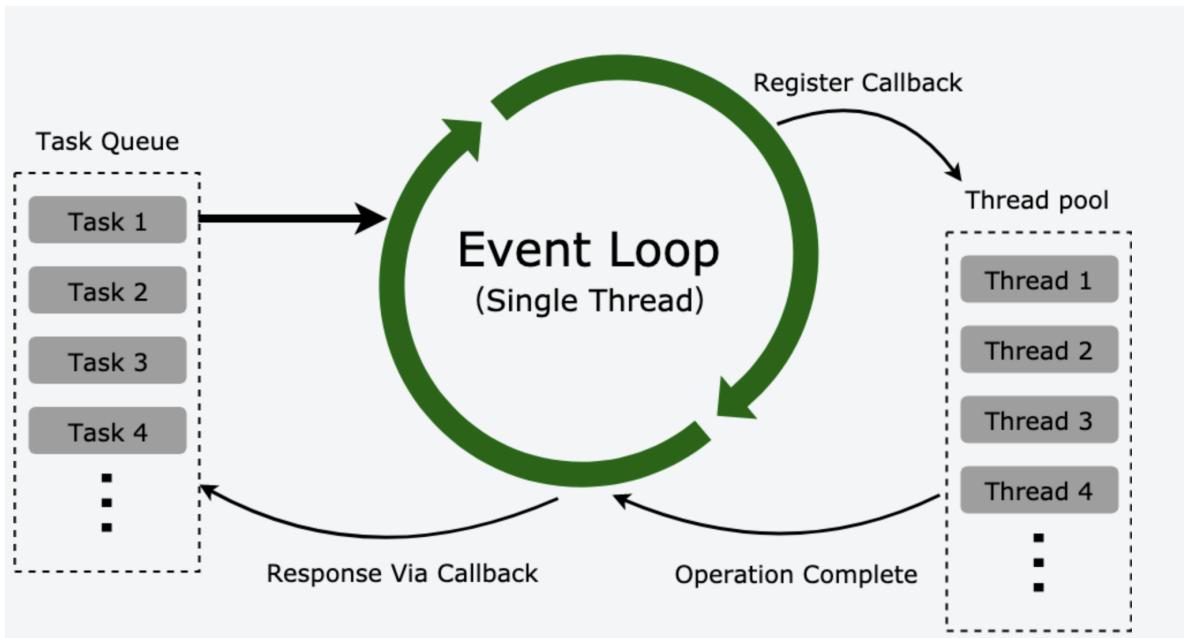
- 面这种改进的模式一般叫做资源池（Resource Pool），或者叫对象池（Object Pool），可以当作是享元模式的升级版，实现不一样，但是目的相同。资源池一般维护一个装载对象的池子，封装有获取、释放资源的方法，当需要对象的时候直接从资源池中获取，使用完毕之后释放资源等待下次被获取。
- 在上面的例子中，驾考车相当于有限资源，考生作为访问者根据资源的使用情况从资源池中获取资源，如果资源池中的资源都正在被占用，要么资源池创建新的资源，要么访问者等待占用的资源被释放。
- 资源池在后端应用相当广泛，比如缓冲池、连接池、线程池、字符常量池等场景，前端使用场景不多，但是也有使用，比如有些频繁的 DOM 创建销毁操作，就可以引入对象池来节约一些 DOM 创建损耗。

下面介绍资源池的几种主要应用。

4.1 线程池

以 Node.js 中的线程池为例，Node.js 的 Javascript 引擎是执行在单线程中的，启动的时候会新建 4 个线程放到线程池中，当遇到一些异步 I/O 操作（比如文件异步读写、DNS 查询等）或者一些 CPU 密集的操作（Crypto、zlib 模块等）的时候，会在线程池中拿出一个线程去执行。如果有需要，线程池会按需创建新的线程。

线程池在整个 Node.js 事件循环中的位置可以参照下图：



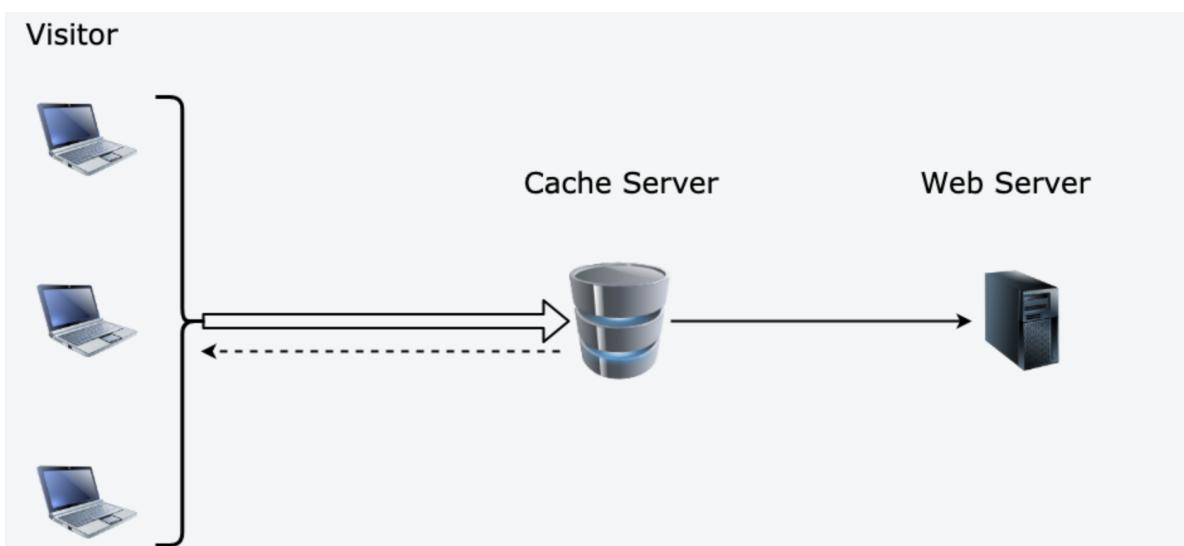
上面这个图就是 Node.js 的事件循环（Event Loop）机制，简单解读一下（扩展视野，不一定需要懂）：

- 所有任务都在主线程上执行，形成执行栈（Execution Context Stack）；
- 主线程之外维护一个任务队列（Task Queue），接到请求时将请求作为一个任务放入这个队列中，然后继续接收其他请求；
- 一旦执行栈中的任务执行完毕，主线程空闲时，主线程读取任务队列中的任务，检查队列中是否有要处理的事件，这时要分两种情况：如果是非 I/O 任务，就亲自处理，并通过回调函数返回到上层调用；如果是 I/O 任务，将传入的参数和回调函数封装成请求对象，并将这个请求对象推入线程池等待执行，主线程则读取下一个任务队列的任务，以此类推处理完任务队列中的任务；
- 线程池当线程可用时，取出请求对象执行 I/O 操作，任务完成以后归还线程，并把这个完成的事件放到任务队列的尾部，等待事件循环，当主线程再次循环到该事件时，就直接处理并返回给上层调用；

4.2 缓存

根据二八原则，80% 的请求其实访问的是 20% 的资源，我们可以将频繁访问的资源缓存起来，如果用户访问被缓存起来的资源就直接返回缓存的版本，这就是 Web 开发中经常遇到的缓存。

缓存服务器就是缓存的最常见应用之一，也是复用资源的一种常用手段。缓存服务器的示意图如下：



- 缓存服务器位于访问者与业务服务器之间，对业务服务器来说，减轻了压力，减小了负载，提高了数据查询的性能。对用户来说，提升了网页打开速度，优化了体验。

- 缓存技术用的非常多，不仅仅用在缓存服务器上，浏览器本地也有缓存，查询的 DNS 也有缓存，包括我们的电脑 CPU 上，也有缓存硬件。

4.3 连接池

我们知道对数据库进行操作需要先创建一个数据库连接对象，然后通过创建好的数据库连接来对数据库进行 CRUD（增删改查）操作。如果访问量不大，对数据库的 CRUD 操作就不多，每次访问都创建连接并在使用完销毁连接就没什么，但是如果访问量比较多，并发的要求比较高时，频繁创建和销毁连接就比较消耗资源了。

- 这时，可以不销毁连接，一直使用已创建的连接，就可以避免频繁创建销毁连接的损耗了。但是有个问题，一个连接同一时间只能做一件事，某使用者（一般是线程）正在使用时，其他使用者就不可以使用了，所以如果只创建一个不关闭的连接显然不符合要求，我们需要创建多个不关闭的连接。
- 这就是连接池的来源，创建多个数据库连接，当有调用的时候直接在创建好的连接中拿出来使用，使用完毕之后将连接放回去供其他调用者使用。
- 我们以 `Node.js` 中 `mysql` 模块的连接池应用为例，看看后端一般是如何使用数据库连接池的。
在 `Node.js` 中使用 `mysql` 创建单个连接，一般这样使用：

```
var mysql = require('mysql')

var connection = mysql.createConnection({      // 创建数据库连接
  host: 'localhost',
  user: 'root',          // 用户名
  password: '123456',    // 密码
  database: 'db',        // 指定数据库
  port: '3306'           // 端口号
})

// 连接回调，在回调中增删改查
connection.connect(...)

// 关闭连接
connection.end(...)
```

在 `Node.js` 中使用 `mysql` 模块的连接池创建连接：

```
var mysql = require('mysql')

var pool = mysql.createPool({      // 创建数据库连接池
  host: 'localhost',
  user: 'root',          // 用户名
  password: '123456',    // 密码
  database: 'db',        // 指定数据库
  port: '3306'           // 端口号
})

// 从连接池中获取一个连接，进行增删改查
pool.getConnection(function(err, connection) {
  // ... 数据库操作
  connection.release() // 将连接释放回连接池中
})

// 关闭连接池
pool.end()
```

- 一般连接池在初始化的时候，都会自动打开 n 个连接，称为连接预热。如果这 n 个连接都被使用了，再从连接池中请求新的连接时，会动态地隐式创建额外连接，即自动扩容。如果扩容后的连接池一段时间后有不少连接没有被调用，则自动缩容，适当释放空闲连接，增加连接池中连接的使用效率。在连接失效的时候，自动抛弃无效连接。在系统关闭的时候，自动释放所有连接。为了维持连接池的有效运转和避免连接池无限扩容，还会给连接池设置最大最小连接数。
- 这些都是连接池的功能，可以看到连接池一般可以根据当前使用情况自动地进行缩容和扩容，来进行连接池资源的最优化，和连接池连接的复用效率最大化。这些连接池的功能点，看着是不是和之前驾校例子的优化过程有点似曾相识呢～
- 在实际项目中，除了数据库连接池外，还有 `HTTP` 连接池。使用 `HTTP` 连接池管理长连接可以复用 `HTTP` 连接，省去创建 `TCP` 连接的 3 次握手和关闭 `TCP` 连接的 4 次挥手的步骤，降低请求响应的时间。

连接池某种程度也算是一种缓冲池，只不过这种缓冲池是专门用来管理连接的。

4.4 字符常量池

很多语言的引擎为了减少字符串对象的重复创建，会在内存中维护有一个特殊的内存，这个内存就叫字符常量池。当创建新的字符串时，引擎会对这个字符串进行检查，与字符常量池中已有的字符串进行对比，如果存在有相同内容的字符串，就直接将引用返回，否则在字符常量池中创建新的字符常量，并返回引用。

类似于 `Java`、`C#` 这些语言，都有字符常量池的机制。JavaScript 有多个引擎，以 Chrome 的 V8 引擎为例，V8 在把 JavaScript 编译成字节码过程中就引入了字符常量池这个优化手段，这就是为什么很多 JavaScript 的书籍都提到了 JavaScript 中的字符串具有不可变性，因为如果内存中的字符串可变，一个引用操作改变了字符串的值，那么其他同样的字符串也会受到影响。

可以引用《JavaScript 高级程序设计》中的话解释一下：

ECMAScript 中的字符串是不可变的，也就是说，字符串一旦创建，它们的值就不能改变。要改变某个变量保存的字符串，首先要销毁原来的字符串，然后再用另一个包含新值的字符串填充该变量。

字符常量池也是复用资源的一种手段，只不过这种手段通常用在编译器的运行过程中，通常开发（搬砖）过程用不到，了解即可。

5. 享元模式的优缺点

享元模式的优点：

- 由于减少了系统中的对象数量，提高了程序运行效率和性能，精简了内存占用，加快运行速度；
- 外部状态相对独立，不会影响到内部状态，所以享元对象能够在不同的环境被共享；

享元模式的缺点：

- 引入了共享对象，使对象结构变得复杂；
- 共享对象的创建、销毁等需要维护，带来额外的复杂度（如果需要把共享对象维护起来的话）；

6. 享元模式的适用场景

- 如果一个程序中大量使用了相同或相似对象，那么可以考虑引入享元模式；
- 如果使用了大量相同或相似对象，并造成了比较大的内存开销；
- 对象的大多数状态可以被转变为外部状态；
- 剥离出对象的外部状态后，可以使用相对较少的共享对象取代大量对象；
- 在一些程序中，如果引入享元模式对系统的性能和内存的占用影响不大时，比如目标对象不多，或者场景比较简单，则不需要引入，以免适得其反。

7. 其他相关模式

- 享元模式和单例模式、工厂模式、组合模式、策略模式、状态模式等等经常会一起使用。

7.1 享元模式和工厂模式、单例模式

- 在区分出不同种类的外部状态后，创建新对象时需要选择不同种类的共享对象，这时就可以使用工厂模式来提供共享对象，在共享对象的维护上，经常会采用单例模式来提供单实例的共享对象。

7.2 享元模式和组合模式

- 在使用工厂模式来提供共享对象时，比如某些时候共享对象中的某些状态就是对象不需要的，可以引入组合模式来提升自定义共享对象的自由度，对共享对象的组成部分进一步归类、分层，来实现更复杂的多层次对象结构，当然系统也会更难维护。

7.3 享元模式和策略模式

策略模式中的策略属于一系列功能单一、细粒度的细粒度对象，可以作为目标对象来考虑引入享元模式进行优化，但是前提是这些策略是会被频繁使用的，如果不经常使用，就没有必要了。

#适配器模式

- 适配器模式（Adapter Pattern）又称包装器模式，将一个类（对象）的接口（方法、属性）转化为用户需要的另一个接口，解决类（对象）之间接口不兼容的问题。
- 主要功能是进行转换匹配，目的是复用已有的功能，而不是来实现新的接口。也就是说，访问者需要的功能应该是已经实现好了的，不需要适配器模式来实现，适配器模式主要是负责把不兼容的接口转换成访问者期望的格式而已。

1. 你曾见过的适配器模式

- 现实生活中我们会遇到形形色色的适配器，最常见的就是转接头了，比如不同规格电源接口的转接头、iPhone 手机的 3.5 毫米耳机插口转接头、DP/miniDP/HDMI/DVI/VGA 等视频转接头、电脑、手机、ipad 的电源适配器，都是属于适配器的范畴。
- 还有一个比较典型的翻译官场景，比如老板张三去国外谈合作，带了个翻译官李四，那么李四就是作为讲不同语言的人之间交流的适配器？，老板张三的话的内容含义没有变化，翻译官将老板的话转换成国外客户希望的形式。

在类似场景中，这些例子有以下特点：

- 旧有接口格式已经不满足现在的需要；
- 通过增加适配器来更好地使用旧有接口；

2. 适配器模式的实现

我们可以实现一下电源适配器的例子，一开始我们使用的中国插头标准：

```
var chinaPlug = {
  type: '中国插头',
  chinaInPlug() {
    console.log('开始供电')
  }
}

chinaPlug.chinaInPlug()
// 输出：开始供电
```

但是我们出国旅游了，到了日本，需要增加一个日本插头到中国插头的电源适配器，来将我们原来的电源线用起来：

```
var chinaPlug = {
  type: '中国插头',
  chinaInPlug() {
```

```

        console.log('开始供电')
    }

}

var japanPlug = {
    type: '日本插头',
    japanInPlug() {
        console.log('开始供电')
    }
}

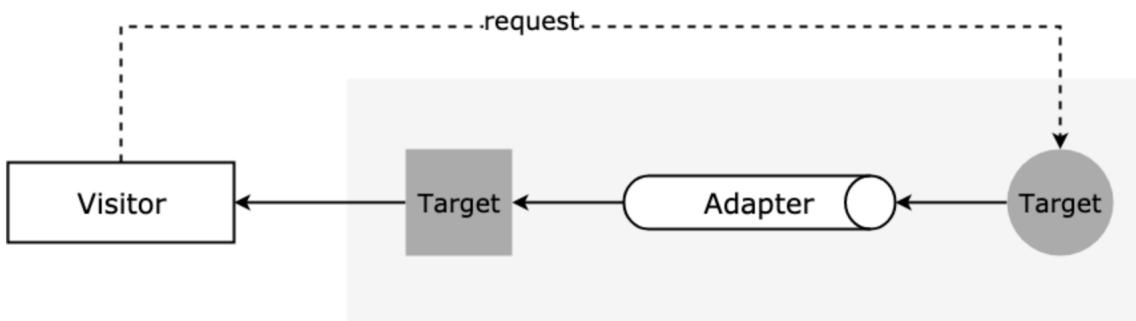
/* 日本插头电源适配器 */
function japanPlugAdapter(plug) {
    return {
        chinaInPlug() {
            return plug.japanInPlug()
        }
    }
}

japanPlugAdapter(japanPlug).chinaInPlug()
// 输出: 开始供电

```

由于适配器模式的例子太简单，如果希望看更多的实战相关应用，可以看下一个节。

适配器模式的原理大概如下图：



访问者需要目标对象的某个功能，但是这个对象的接口不是自己期望的，那么通过适配器模式对现有对象的接口进行包装，来获得自己需要的接口格式。

3. 适配器模式在实战中的应用

适配器模式在日常开发中还是比较频繁的，其实可能你已经使用了，但却不知道原来这就是适配器模式啊。？

我们可以推而广之，适配器可以将新的软件实体适配到老的接口，也可以将老的软件实体适配到新的接口，具体如何来进行适配，可以根据具体使用场景来灵活使用。

3.1 jQuery.ajax 适配 Axios

有的使用 `jQuery` 的老项目使用 `$.ajax` 来发送请求，现在的新项目一般使用 `Axios`，那么现在有个老项目的代码中全是 `$.ajax`，如果你挨个修改，那么 `bug` 可能就跟地鼠一样到处冒出来让你焦头烂额，这时可以采用适配器模式来将老的使用形式适配到新的技术栈上：

```

/* 适配器 */
function ajax2AxiosAdapter.ajaxOptions) {
    return axios({
        url: ajaxOptions.url,

```

```
        method: ajaxOptions.type,
        responseType: ajaxOptions.dataType,
        data: ajaxOptions.data
    })
    .then.ajaxOptions.success)
    .catch.ajaxOptions.error)
}

/* 经过适配器包装 */
$.ajax = function(options) {
    return ajax2AxiosAdapter(options)
}

$.ajax({
    url: '/demo-url',
    type: 'POST',
    dataType: 'json',
    data: {
        name: '张三',
        id: '2345'
    },
    success: function(data) {
        console.log('访问成功! ')
    },
    error: function(err) {
        console.err('访问失败~')
    }
})
```

可以看到老的代码表现形式依然不变，但是真正发送请求是通过新的发送方式来进行的。当然你也可以把 `Axios` 的请求适配到 `$.ajax` 上，就看你如何使用适配器了。

3.2 业务数据适配

- 在实际项目中，我们经常会遇到树形数据结构和表形数据结构的转换，比如全国省市区结构、公司组织结构、军队编制结构等等。以公司组织结构为例，在历史代码中，后端给了公司组织结构的树形数据，在以后的业务迭代中，会增加一些要求非树形结构的场景。比如增加了将组织维护起来的功能，因此就需要在新增组织的时候选择上级组织，在某个下拉菜单中选择这个新增组织的上级菜单。或者增加了将人员归属到某一级组织的需求，需要在某个下拉菜单中选择任一级组织。
 - 在这些业务场景中，都需要将树形结构平铺开，但是我们又不能直接将旧有的树形结构状态进行修改，因为在项目别的地方已经使用了老的树形结构状态，这时我们可以引入适配器来将老的数据结构进行适配：

```
/* 原来的树形结构 */
const oldTreeData = [
  {
    name: '总部',
    place: '一楼',
    children: [
      { name: '财务部', place: '二楼' },
      { name: '生产部', place: '三楼' },
      {
        name: '开发部', place: '三楼', children: [
          {
            name: '软件部', place: '四楼', children: [
              { name: '后端部', place: '五楼' },
              { name: '前端部', place: '七楼' }
            ]
          }
        ]
      }
    ]
  }
]
```

```

        { name: '技术支持部', place: '六楼' }]
    }, {
        name: '硬件部', place: '四楼', children: [
            { name: 'DSP部', place: '八楼' },
            { name: 'ARM部', place: '二楼' },
            { name: '调试部', place: '三楼' }
        ]
    }
]
]

/* 树形结构平铺 */
function treeDataAdapter(treeData, lastArrayData = []) {
    treeData.forEach(item => {
        if (item.children) {
            treeDataAdapter(item.children, lastArrayData)
        }
        const { name, place } = item
        lastArrayData.push({ name, place })
    })
    return lastArrayData
}

treeDataAdapter(oldTreeData)

// 返回平铺的组织结构

```

增加适配器后，就可以将原先状态的树形结构转化为所需的结构，而并不改动原先的数据，也不对原来使用旧数据结构的代码有所影响。

3.3 Vue 计算属性

`Vue` 中的计算属性也是一个适配器模式的实例，以官网的例子为例，我们可以一起来理解一下：

```

<template>
  <div id="example">
    <p>Original message: "{{ message }}"</p> <!-- Hello -->
    <p>Computed reversed message: "{{ reversedMessage }}"</p> <!-- olleH -->
  </div>
</template>

<script type='text/javascript'>
  export default {
    name: 'demo',
    data() {
      return {
        message: 'Hello'
      }
    },
    computed: {
      reversedMessage: function() {
        return this.message.split('').reverse().join('')
      }
    }
  }

```

```
</script>
```

旧有 `data` 中的数据不满足当前的要求，通过计算属性的规则来适配成我们需要的格式，对原有数据并没有改变，只改变了原有数据的表现形式。

4. 源码中的适配器模式

`Axios` 是比较热门的网络请求库，在浏览器中使用的时候，`Axios` 的用来发送请求的 `adapter` 本质上是封装浏览器提供的 `API XMLHttpRequest`，我们可以看看源码中是如何封装这个 `API` 的，为了方便观看，进行了一些省略：

```
module.exports = function xhrAdapter(config) {
  return new Promise(function dispatchXhrRequest(resolve, reject) {
    var requestData = config.data
    var requestHeaders = config.headers

    var request = new XMLHttpRequest()

    // 初始化一个请求
    request.open(config.method.toUpperCase(),
      buildURL(config.url, config.params, config.paramsSerializer), true)

    // 设置最大超时时间
    request.timeout = config.timeout

    // readyState 属性发生变化时的回调
    request.onreadystatechange = function handleLoad() { ... }

    // 浏览器请求退出时的回调
    request.onabort = function handleAbort() { ... }

    // 当请求报错时的回调
    request.onerror = function handleError() { ... }

    // 当请求超时调用的回调
    request.ontimeout = function handleTimeout() { ... }

    // 设置HTTP请求头的值
    if ('setRequestHeader' in request) {
      request.setRequestHeader(key, val)
    }

    // 跨域的请求是否应该使用证书
    if (config.withCredentials) {
      request.withCredentials = true
    }

    // 响应类型
    if (config.responseType) {
      request.responseType = config.responseType
    }

    // 发送请求
    request.send(requestData)
  })
}
```

可以看到这个模块主要是对请求头、请求配置和一些回调的设置，并没有对原生的 API 有改动，所以也可以在其他地方正常使用。这个适配器可以看作是对 XMLHttpRequest 的适配，是用户对 Axios 调用层到原生 XMLHttpRequest 这个 API 之间的适配层。

源码可以参见 Github 仓库： axios/lib/adapters/xhr.js

5. 适配器模式的优缺点

适配器模式的优点：

- 已有的功能如果只是接口不兼容，使用适配器适配已有功能，可以使原有逻辑得到更好的复用，有助于避免大规模改写现有代码；
- 可扩展性良好，在实现适配器功能的时候，可以调用自己开发的功能，从而方便地扩展系统的功能；
- 灵活性好，因为适配器并没有对原有对象的功能有所影响，如果不想使用适配器了，那么直接删掉即可，不会对使用原有对象的代码有影响；
- 适配器模式的缺点：会让系统变得零乱，明明调用 A，却被适配到了 B，如果系统中这样的情况很多，那么对可阅读性不太友好。如果没必要使用适配器模式的话，可以考虑重构，如果使用的话，可以考虑尽量把文档完善。

6. 适配器模式的适用场景

- 当你想用已有对象的功能，却想修改它的接口时，一般可以考虑一下是不是可以应用适配器模式。
- 如果你想要使用一个已经存在的对象，但是它的接口不满足需求，那么可以使用适配器模式，把已有的实现转换成你需要的接口；
- 如果你想创建一个可以复用的对象，而且确定需要和一些不兼容的对象一起工作，这种情况可以使用适配器模式，然后需要什么就适配什么；

7. 其他相关模式

适配器模式和代理模式、装饰者模式看起来比较类似，都是属于包装模式，也就是用一个对象来包装另一个对象的模式，他们之间的异同在代理模式中已经详细介绍了，这里再简单对比一下。

7.1 适配器模式与代理模式

- 适配器模式：提供一个不一样的接口，由于原来的接口格式不能用了，提供新的接口以满足新场景下的需求；
- 代理模式：提供一模一样的接口，由于不能直接访问目标对象，找个代理来帮忙访问，使用者可以就像访问目标对象一样来访问代理对象；

7.2 适配器模式、装饰者模式与代理模式

- 适配器模式：功能不变，只转换了原有接口访问格式；
- 装饰者模式：扩展功能，原有功能不变且可直接使用；
- 代理模式：原有功能不变，但一般是经过限制访问的；

#装饰者模式

装饰者模式（Decorator Pattern）又称装饰器模式，在不改变原对象的基础上，通过对原对象添加属性或方法来进行包装拓展，使得原有对象可以动态具有更多功能。

本质是功能动态组合，即动态地给一个对象添加额外的职责，就增加功能角度来看，使用装饰者模式比用继承更为灵活。好处是有效地把对象的核心职责和装饰功能区分开，并且通过动态增删装饰去除目标对象中重复的装饰逻辑。

1. 你曾见过的装饰者模式

- 相信大家都有过房屋装修的经历，当毛坯房建好的时候，已经可以居住了，虽然不太舒适。一般我们自己住当然不会住毛坯，因此我们还会通水电、墙壁刷漆、铺地板、家具安装、电器安装等等步

骤，让房屋渐渐具有各种各样的特性，比如墙壁刷漆和铺地板之后房屋变得更加美观，有了家具居住变得更加舒适，但这些额外的装修并没有影响房屋是用来居住的这个基本功能，这就是装饰的作用。

- 再比如现在我们经常喝的奶茶，除了奶茶之外，还可以添加珍珠、波霸、椰果、仙草、香芋等等辅料，辅料的添加对奶茶的饮用并无影响，奶茶喝起来还是奶茶的味道，只不过辅料的添加让这杯奶茶的口感变得更多样化。
- 生活中类似的场景还有很多，比如去咖啡厅喝咖啡，点了杯摩卡之后我们还可以选择添加糖、冰块、牛奶等等调味品，给咖啡添加特别的口感和风味，但这些调味品的添加并没有影响咖啡的基本性质，不会因为添加了调味品，咖啡就变成奶茶。

在类似场景中，这些例子有以下特点：

- 装饰不影响原有的功能，原有功能可以照常使用；
- 装饰可以增加多个，共同给目标对象添加额外功能；

2. 实例的代码实现

我们可以使用 JavaScript 来将装修房子的例子实现一下：

```
/* 毛坯房 - 目标对象 */
function OriginHouse() {}

OriginHouse.prototype.getDesc = function() {
    console.log('毛坯房')
}

/* 搬入家具 - 装饰者 */
function Furniture(house) {
    this.house = house
}

Furniture.prototype.getDesc = function() {
    this.house.getDesc()
    console.log('搬入家具')
}

/* 墙壁刷漆 - 装饰者 */
function Painting(house) {
    this.house = house
}

Painting.prototype.getDesc = function() {
    this.house.getDesc()
    console.log('墙壁刷漆')
}

var house = new OriginHouse()
house = new Furniture(house)
house = new Painting(house)

house.getDesc()
// 输出： 毛坯房 搬入家具 墙壁刷漆
使用 ES6 的 class 语法：

/* 毛坯房 - 目标对象 */
class OriginHouse {
    getDesc() {
```

```

        console.log('毛坯房')
    }

}

/* 搬入家具 - 装饰者 */
class Furniture {
    constructor(house) {
        this.house = house
    }

    getDesc() {
        this.house.getDesc()
        console.log('搬入家具')
    }
}

/* 墙壁刷漆 - 装饰者 */
class Painting {
    constructor(house) {
        this.house = house
    }

    getDesc() {
        this.house.getDesc()
        console.log('墙壁刷漆')
    }
}

let house = new OriginHouse()
house = new Furniture(house)
house = new Painting(house)

house.getDesc()
// 输出: 毛坯房 搬入家具 墙壁刷漆

```

是不是感觉很麻烦，装饰个功能这么复杂？我们JSer 大可不必走这一套面向对象花里胡哨的，毕竟 JavaScript 的优点就是灵活：

```

/* 毛坯房 - 目标对象 */
var originHouse = {
    getDesc() {
        console.log('毛坯房 ')
    }
}

/* 搬入家具 - 装饰者 */
function furniture() {
    console.log('搬入家具 ')
}

/* 墙壁刷漆 - 装饰者 */
function painting() {
    console.log('墙壁刷漆 ')
}

/* 添加装饰 - 搬入家具 */
originHouse.getDesc = function() {

```

```
var getDesc = originHouse.getDesc
return function() {
    getDesc()
    furniture()
}
}()

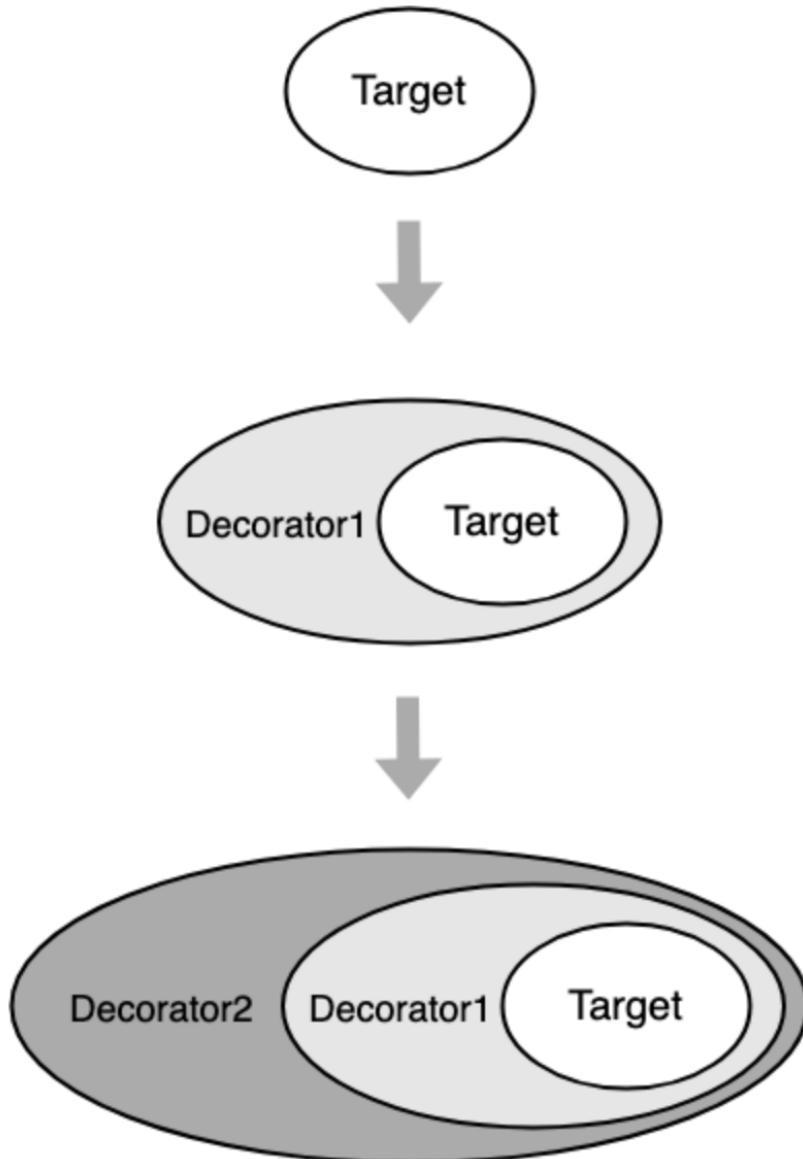
/* 添加装饰 - 墙壁刷漆 */
originHouse.getDesc = function() {
    var getDesc = originHouse.getDesc
    return function() {
        getDesc()
        painting()
    }
}
}()

originHouse.getDesc()
// 输出: 毛坯房 搬入家具 墙壁刷漆
```

简洁明了，且更符合前端日常使用的场景。

3. 装饰者模式的原理

装饰者模式的原理如下图：



可以从上图看出，在表现形式上，装饰者模式和适配器模式比较类似，都属于包装模式。在装饰者模式中，一个对象被另一个对象包装起来，形成一条包装链，并增加了原先对象的功能。

4. 实战中的装饰者模式 4.1 给浏览器事件添加新功能

之前介绍的添加装饰器函数的方式，经常被用来给原有浏览器或 DOM 绑定事件上绑定新的功能，比如在 `onload` 上增加新的事件，或在原来的事件绑定函数上增加新的功能，或者在原本的操作上增加用户行为埋点：

```

window.onload = function() {
    console.log('原先的 onload 事件')
}

/* 发送埋点信息 */
function sendUserOperation() {
    console.log('埋点：用户当前行为路径为 ...')
}

/* 将新的功能添加到 onload 事件上 */
window.onload = function() {
    var originonload = window.onload
    return function() {
        originonload && originonload()
    }
}

```

```

        sendUserOperation()
    }
}()

// 输出: 原先的 onload 事件
// 输出: 埋点: 用户当前行为路径为 ...

```

可以看到通过添加装饰函数，为 `onload` 事件回调增加新的方法，且并不影响原本的功能，我们可以把上面的方法提取出来作为一个工具方法：

```

window.onload = function() {
    console.log('原先的 onload 事件')
}

/* 发送埋点信息 */
function sendUserOperation() {
    console.log('埋点: 用户当前行为路径为 ...')
}

/* 给原生事件添加新的装饰方法 */
function originDecorateFn(originObj, originKey, fn) {
    originObj[originKey] = function() {
        var originFn = originObj[originKey]
        return function() {
            originFn && originFn()
            fn()
        }
    }
}

// 添加装饰功能
originDecorateFn(window, 'onload', sendUserOperation)

// 输出: 原先的 onload 事件
// 输出: 埋点: 用户当前行为路径为 ...

```

4.2 TypeScript 中的装饰器

- 现在的越来越多的前端项目或 `Node` 项目都在拥抱 `JavaScript` 的超集语言 `TypeScript`，如果你了解过 `C#` 中的特性 `Attribute`、`Java` 中的注解 `Annotation`、`Python` 中的装饰器 `Decorator`，那么你就不会对 `TypeScript` 中的装饰器感到陌生，下面我们简单介绍一下 `TypeScript` 中的装饰器。

`TypeScript` 中的装饰器可以被附加到类声明、方法、访问符、属性和参数上，装饰器的类型有参数装饰器、方法装饰器、访问器或参数装饰器、参数装饰器。

- `TypeScript` 中的装饰器使用 `@expression` 这种形式，`expression` 求值后为一个函数，它在运行时被调用，被装饰的声明信息会被做为参数传入。

多个装饰器应用使用在同一个声明上时：

- 由上至下依次对装饰器表达式求值；
- 求值的结果会被当成函数，由下至上依次调用；

那么使用官网的一个例子：

```
function f() {
```

```

        console.log("f(): evaluated");
        return function (target, propertyKey: string, descriptor:
PropertyDescriptor) {
            console.log("f(): called");
        }
    }

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey: string, descriptor:
PropertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}

// f(): evaluated
// g(): evaluated
// g(): called
// f(): called

```

可以看到上面的代码中，高阶函数 `f` 与 `g` 返回了另一个函数（装饰器函数），所以 `f`、`g` 这里又被称为装饰器工厂，即帮助用户传递可供装饰器使用的参数的工厂。另外注意，演算的顺序是从下到上，执行的时候是从下到上的。

再比如下面一个场景

```

class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
        return "Hello, " + this.greeting;
    }
}

for (let key in new Greeter('Jim')) {
    console.log(key);
}
// 输出: greeting greet

```

如果我们不希望 `greet` 被 `for-in` 循环遍历出来，可以通过装饰器的方式来方便地修改属性的属性描述符：

```

function enumerable(value: boolean) {
    return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
        descriptor.enumerable = value;
    }
}

```

```

    };

}

class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    @enumerable(false)
    greet() {
        return "Hello, " + this.greeting;
    }
}

for (let key in new Greeter('Jim')) {
    console.log(key);
}
// 输出: greeting

```

- 这样 `greet` 就变成不可枚举了，使用起来比较方便，对其他属性进行声明不可枚举的时候也只用在之前加一行 `@enumerable(false)` 即可，不用大费周章的 `Object.defineProperty(...)` 进行繁琐的声明了。
- `TypeScript` 的装饰器还有很多有用的用法，感兴趣的同学可以阅读一下 `TypeScript` 的 `Decorators` 官网文档相关内容。

5. 装饰者模式的优缺点

装饰者模式的优点：

- 我们经常使用继承的方式来实现功能的扩展，但这样会给系统中带来很多的子类和复杂的继承关系，装饰者模式允许用户在不引起子类数量暴增的前提下动态地修饰对象，添加功能，装饰者和被装饰者之间松耦合，可维护性好；
- 被装饰者可以使用装饰者动态地增加和撤销功能，可以在运行时选择不同的装饰器，实现不同的功能，灵活性好；
- 装饰者模式把一系列复杂的功能分散到每个装饰器当中，一般一个装饰器只实现一个功能，可以给一个对象增加多个同样的装饰器，也可以把一个装饰器用来装饰不同的对象，有利于装饰器功能的复用；
- 可以通过选择不同的装饰者的组合，创造不同行为和功能的结合体，原有对象的代码无须改变，就可以使得原有对象的功能变得更强大和更多样化，符合开闭原则；

装饰者模式的缺点：

- 使用装饰者模式时会产生很多细粒度的装饰者对象，这些装饰者对象由于接口和功能的多样化导致系统复杂度增加，功能越复杂，需要的细粒度对象越多；
- 由于更大的灵活性，也就更容易出错，特别是对于多级装饰的场景，错误定位会更加繁琐；

6. 装饰者模式的适用场景

- 如果不希望系统中增加很多子类，那么可以考虑使用装饰者模式；
- 需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，这时采用装饰者模式可以很好实现；
- 当对象的功能要求可以动态地添加，也可以动态地撤销，可以考虑使用装饰者模式；

7. 其他相关模式 7.1 装饰者模式与适配器模式

装饰者模式和适配器模式都是属于包装模式，然而他们的意图有些不一样：

- 装饰者模式：扩展功能，原有功能还可以直接使用，一般可以给目标对象多次叠加使用多个装饰者；
- 适配器模式：功能不变，但是转换了原有接口的访问格式，一般只给目标对象使用一次；

7.2 装饰者模式与组合模式

这两个模式有相似之处，都涉及到对象的递归调用，从某个角度来说，可以把装饰者模式看做是只有一个组件的组合模式。

- 装饰者模式：动态地给对象增加功能；
- 组合模式：管理组合对象和叶子对象，为它们提供一致的操作接口给客户端，方便客户端的使用；

7.3 装饰者模式与策略模式

装饰者模式和策略模式都包含有许多细粒度的功能模块，但是他们的使用思路不同：

- 装饰者模式：可以递归调用，使用多个功能模式，功能之间可以叠加组合使用；
- 策略模式：只有一层选择，选择某一个功能；

#外观模式

外观模式（Facade Pattern）又叫门面模式，定义一个将子系统的一组接口集成在一起的高层接口，以提供一个一致的外观。外观模式让外界减少与子系统内多个模块的直接交互，从而减少耦合，让外界可以更轻松地使用子系统。本质是封装交互，简化调用。

外观模式在源码中使用很多，具体可以参考后文中源码阅读部分。

1. 你曾见过的外观模式

最近这些年无人机很流行，特别是大疆的旋翼无人机。旋翼无人机的种类也很多，四旋翼、六旋翼、八旋翼、十六旋翼甚至是共轴双桨旋翼机，他们因为结构不同而各自有一套原理类似，但实现细节不同的旋翼控制方式。

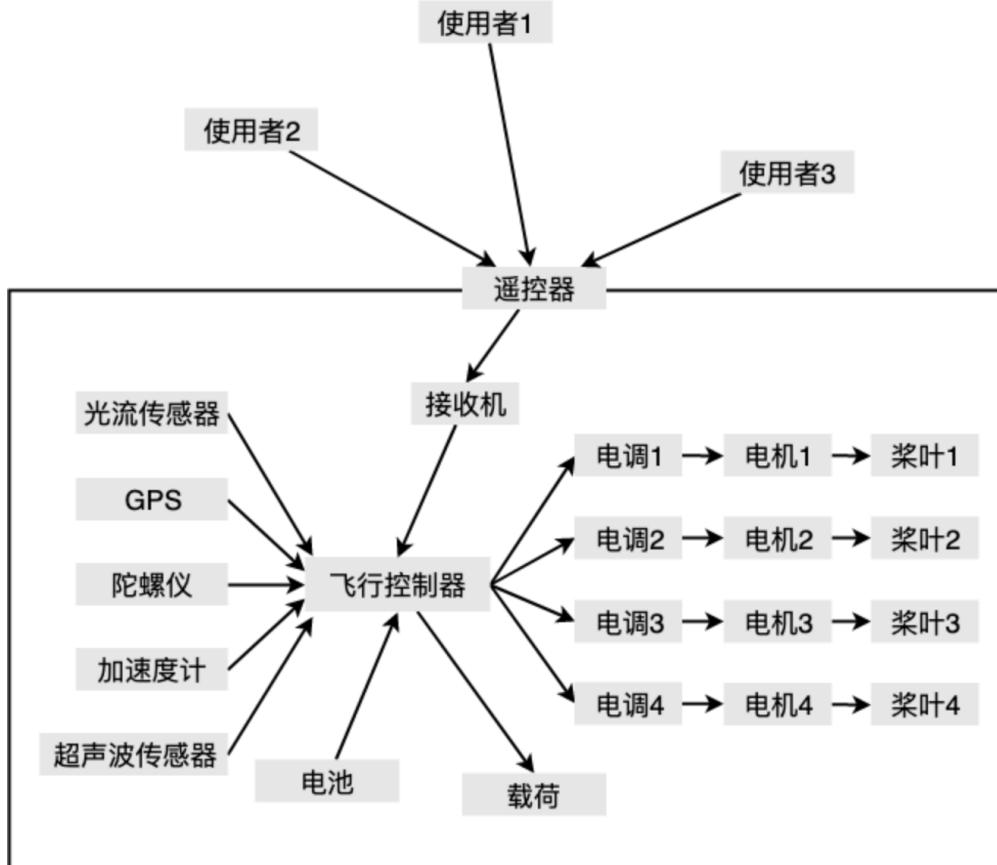
- 如果用户需要把每种旋翼的控制原理弄清楚，那么门槛就太高了，所以无人机厂商会把具体旋翼控制的细节封装起来，用户所要接触的只是手上的遥控器，无论什么类型的无人机，遥控器的控制方式都一样，前后左右上下和左转右转。
- 对于使用者来说，遥控器就相当于是无人机系统的外观，使用者只要操纵遥控器就可以达到控制无人机的目的，而具体无人机内部的飞行控制器、电调（电子调速器）、电机、数字电传、陀螺仪、加速度计等等子模块之间复杂的调用关系将被封装起来，对于使用者而言不需要了解，因此也降低了使用难度。
- 类似的例子也有不少，比如常见的空调、冰箱、洗衣机、洗碗机，内部结构都并不简单，对于我们使用者而言，理解他们内部的运行机制的门槛比较高，但是理解遥控器/控制面板上面寥寥几个按钮就相对容易的多，这就是外观模式的意义。

在类似场景中，这些例子有以下特点：

- 一个统一的外观为复杂的子系统提供一个简单的高层功能接口；
- 原本访问者直接调用子系统内部模块导致的复杂引用关系，现在可以通过只访问这个统一的外观来避免；

2. 实例的代码实现

无人机系统的模块图大概如下：



四旋翼无人机系统

可以看到无人机系统还是比较复杂的，系统内模块众多，如果用户需要对每个模块的作用都了解的话，那就太麻烦了，有了遥控器之后，使用者只要操作摇杆，发出前进、后退等等的命令，无人机系统接受到信号之后会经过算法把计算后的指令发送到电调，控制对应电机以不同转速带动桨叶，给无人机提供所需的扭矩和升力，从而实现目标运动。

关于无人机的例子，因为子模块众多，写成代码有点太啰嗦，这里只给出一个简化版本的代码：

```

var uav = {
    /* 电子调速器 */
    dianbiao1: {
        up() {
            console.log('电调1发送指令：电机1增大转速')
            uav.dianji1.up()
        },
        down() {
            console.log('电调1发送指令：电机1减小转速')
            uav.dianji1.up()
        }
    },
    dianbiao2: {
        up() {
            console.log('电调2发送指令：电机2增大转速')
            uav.dianji2.up()
        },
        down() {
            console.log('电调2发送指令：电机2减小转速')
            uav.dianji2.down()
        }
    }
}

```

```
diantiao3: {
    up() {
        console.log('电调3发送指令: 电机3增大转速')
        uav.dianji3.up()
    },
    down() {
        console.log('电调3发送指令: 电机3减小转速')
        uav.dianji3.down()
    }
},
diantiao4: {
    up() {
        console.log('电调4发送指令: 电机4增大转速')
        uav.dianji4.up()
    },
    down() {
        console.log('电调4发送指令: 电机4减小转速')
        uav.dianji4.down()
    }
},
/* 电机 */
dianji1: {
    up() { console.log('电机1增大转速') },
    down() { console.log('电机1减小转速') }
},
dianji2: {
    up() { console.log('电机2增大转速') },
    down() { console.log('电机2减小转速') }
},
dianji3: {
    up() { console.log('电机3增大转速') },
    down() { console.log('电机3减小转速') }
},
dianji4: {
    up() { console.log('电机4增大转速') },
    down() { console.log('电机4减小转速') }
},
/* 遥控器 */
controller: {
    /* 上升 */
    up() {
        uav.diantiao1.up()
        uav.diantiao2.up()
        uav.diantiao3.up()
        uav.diantiao4.up()
    },
    /* 前进 */
    forward() {
        uav.diantiao1.down()
        uav.diantiao2.down()
        uav.diantiao3.up()
        uav.diantiao4.up()
    },
    /* 下降 */
    down() {
```

```

down() {
    uav.diantiao1.down()
    uav.diantiao2.down()
    uav.diantiao3.down()
    uav.diantiao4.down()
}

/* 左转 */
left() {
    uav.diantiao1.up()
    uav.diantiao2.down()
    uav.diantiao3.up()
    uav.diantiao4.down()
}
}

/* 操纵无人机 */
uav.controller.down() // 发送下降指令
uav.controller.left() // 发送左转指令

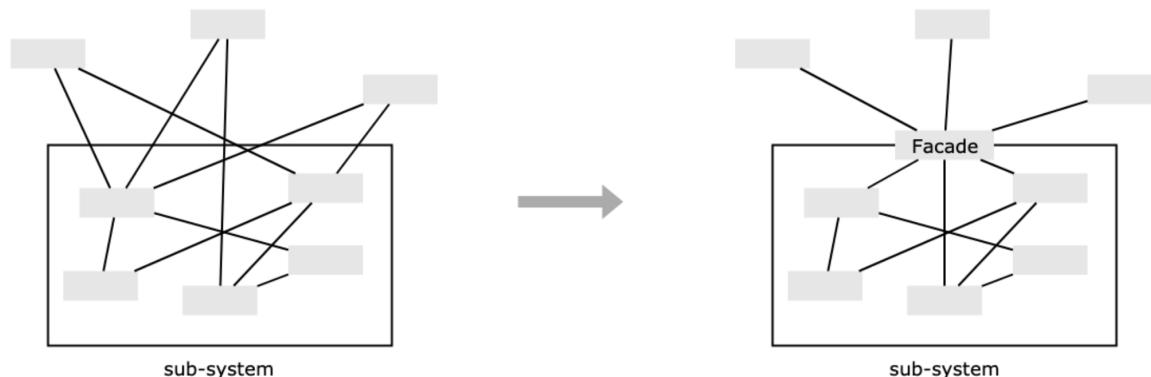
```

无人机系统是比较复杂，但是可以看到无人机的操纵却比较简单，正是因为有遥控器这个外观的存在。

3. 外观模式的原理

- 正如之前无人机的例子，虽然无人机实际操控比较复杂，但是通过对 controller 这个遥控器的使用，让使用者对无人机这个系统的控制变得简单，只需调用遥控器这个外观提供的方法即可，而这个方法里封装的一系列复杂操作，则不是我们要关注的重点。
- 从中就可以理解外观模式的意义了，遥控器作为无人机系统的功能出口，降低了使用者对复杂的无人机系统使用的难度，甚至让广场上的小朋友都能玩起来了

概略图如下：



注意：外观模式一般是作为子系统的功能出口出现，使用的时候可以在其中增加新的功能，但是不推介这样做，因为外观应该是对已有功能的包装，不应在其中掺杂新的功能。

4. 实战中的外观模式 4.1 函数参数重载

有一种情况，比如某个函数有多个参数，其中一个参数可以传递也可以不传递，你当然可以直接弄两个接口，但是使用函数参数重载的方式，可以让使用者获得更大的自由度，让两个使用上基本类似的方法获得统一的外观。

```

function domBindEvent(nodes, type, selector, fn) {
  if (fn === undefined) {
    fn = selector
    selector = null
  }
  // ... 剩下相关逻辑
}

domBindEvent(nodes, 'click', '#div1', fn)
domBindEvent(nodes, 'click', fn)

```

- 这种方式在一些工具库或者框架提供的多功能方法上经常得到使用，特别是在通用 API 的某些参数可传可不传的时候。
- 参数重载之后的函数在使用上会获得更大的自由度，而不必重新创建一个新的 API，这在 vue、React、jQuery、Lodash 等库中使用非常频繁。

4.2 抹平浏览器兼容性问题

外观模式经常被用于 JavaScript 的库中，封装一些接口用于兼容多浏览器，让我们可以间接调用我们封装的外观，从而屏蔽了浏览器差异，便于使用。

比如经常用的兼容不同浏览器的事件绑定方法：

```

function addEvent(element, type, fn) {
  if (element.addEventListener) {          // 支持 DOM2 级事件处理方法的浏览器
    element.addEventListener(type, fn, false)
  } else if (element.attachEvent) {         // 不支持 DOM2 级但支持 attachEvent
    element.attachEvent('on' + type, fn)
  } else {
    element['on' + type] = fn             // 都不支持的浏览器
  }
}

var myInput = document.getElementById('myinput')

addEvent(myInput, 'click', function() {
  console.log('绑定 click 事件')
})

```

- 下面一个小节我们可以看看 jQuery 的源码是如何进行事件绑定的。
- 除了事件绑定之外，在抹平浏览器兼容性的其他问题上我们也经常使用外观模式：

```

// 移除 DOM 上的事件
function removeEvent(element, type, fn) {
  if (element.removeEventListener) {
    element.removeEventListener(type, fn, false)
  } else if (element.detachEvent) {
    element.detachEvent('on' + type, fn)
  } else {
    element['on' + type] = null
  }
}

// 获取样式
function getStyle(obj, styleName) {
  if (window.getComputedStyle) {

```

```

        var styles = getComputedStyle(obj, null)[styleName]
    } else {
        var styles = obj.currentStyle[styleName]
    }
    return styles
}

// 阻止默认事件
var preventDefault = function(event) {
    if (event.preventDefault) {
        event.preventDefault()
    } else { // IE 下
        event.returnValue = false
    }
}

// 阻止事件冒泡
var cancelBubble = function(event) {
    if (event.stopPropagation) {
        event.stopPropagation()
    } else { // IE 下
        event.cancelBubble = true
    }
}

```

通过将处理不同浏览器兼容性问题的过程封装成一个外观，我们在使用的时候可以直接使用外观方法即可，在遇到兼容性问题的时候，这个外观方法自然帮我们解决，方便又不容易出错。

5. 源码中的外观模式 5.1 Vue 源码中的函数参数重载

`Vue` 提供的一个创建元素的方法 `createElement` 就使用了函数参数重载，使得使用者在使用这个参数的时候很灵活：

```

export function createElement(
    context,
    tag,
    data,
    children,
    normalizationType,
    alwaysNormalize
) {
    if (Array.isArray(data) || isPrimitive(data)) { // 参数的重载
        normalizationType = children
        children = data
        data = undefined
    }

    // ...
}

```

- `createElement` 方法里面对第三个参数 `data` 进行了判断，如果第三个参数的类型是 `array`、`string`、`number`、`boolean` 中的一种，那么说明是 `createElement(tag [, data], children, ...)` 这样的使用方式，用户传的第二个参数不是 `data`，而是 `children`。
- `data` 这个参数是包含模板相关属性的数据对象，如果用户没有什么要设置，那这个参数自然不传，不使用函数参数重载的情况下，需要用户手动传递 `null` 或者 `undefined` 之类，参数重载之后，用户对 `data` 这个参数可传可不传，使用自由度比较大，也很方便。

- `createElement` 方法的源码参见 Github 链接 [vue/src/core/vdom/create-element.js](#)

5.2 Lodash 源码中的函数参数重载

Lodash 的 `range` 方法的 API 为 `_.range([start=0], end, [step=1])`，这就很明显使用了参数重载，这个方法调用了一个内部函数 `createRange`：

```
function createRange(fromRight) {
  return (start, end, step) => {
    // ...

    if (end === undefined) {
      end = start
      start = 0
    }

    // ...
  }
}
```

意思就是，如果没有传第二个参数，那么就把传入的第一个参数作为 `end`，并把 `start` 置为默认值。

`createRange`e` 方法的源码参见 Github 链接 [`lodash/.internal/createRange.js](#)

5.3 jQuery 源码中的函数参数重载

函数参数重载在源码中使用比较多，jQuery 中也有大量使用，比如 `on`、`off`、`bind`、`one`、`load`、`ajaxPrefilter` 等方法，这里以 `off` 方法为例，该方法在选择元素上移除一个或多个事件的事件处理函数。源码如下：

```
off: function (types, selector, fn) {
  // ...

  if (selector === false || typeof selector === 'function') {
    // ( types [, fn] ) 的使用方式
    fn = selector
    selector = undefined
  }

  // ...
}
```

可以看到如果传入第二个参数为 `false` 或者是函数的时候，就是 `off(types [, fn])` 的使用方式。

- `off` 方法的源码参见 Github 链接 [jquery/src/event.js](#)

再比如 `load` 方法的源码：

```

jQuery.fn.load = function(url, params, callback) {
    // ...

    if (isFunction(params)) {
        callback = params
        params = undefined
    }

    // ...
}

```

- 可以看到 `jquery` 对第二个参数进行了判断，如果是函数，就是 `load(url [, callback])` 的使用方式。
- `load` 方法的源码参见 Github 链接 [jquery/src/ajax/load.js](#)

5.4 jQuery 源码中的外观模式

当我们使用 `jquery` 的 `$(document).ready(...)` 来给浏览器加载事件添加回调时，`jquery` 会使用源码中的 `bindReady` 方法：

```

bindReady: function() {
    // ...

    // Mozilla, Opera and webkit 支持
    if (document.addEventListener) {
        document.addEventListener('DOMContentLoaded', DOMContentLoaded, false)

        // A fallback to window.onload, that will always work
        window.addEventListener('load', jQuery.ready, false)

        // 如果使用了 IE 的事件绑定形式
    } else if (document.attachEvent) {
        document.attachEvent('onreadystatechange', DOMContentLoaded)

        // A fallback to window.onload, that will always work
        window.attachEvent('onload', jQuery.ready)
    }

    // ...
}

```

- 通过这个方法，`jquery` 帮我们将不同浏览器下的不同绑定形式隐藏起来，从而简化了使用。
- `bindReady` 方法的源码参见 Github 链接 [jquery/src/core.js](#)

除了屏蔽浏览器兼容性问题之外，jQuery 还有其他的一些其他外观模式的应用：

- 比如修改 `css` 的时候可以 `($('p').css('color', 'red'))`，也可以 `($('p').css('width', 100))`，对不同样式的操作被封装到同一个外观方法中，极大地方便了使用，对不同样式的特殊处理（比如设置 `width` 的时候不用加 `px`）也一同被封装了起来。
- 源码参见 Github 链接 [jquery/src/css.js](#)
- 再比如 `jquery` 的 `ajax` 的 API `$.ajax(url [, settings])`，当我们在设置以 JSONP 的形式发送请求的时候，只要传入 `dataType: 'jsonp'` 设置，`jQuery` 会进行一些额外操作帮我们启动 JSONP 流程，并不需要使用者手动添加代码，这些都被封装在 `$.ajax()` 这个外观方法中了。

源码参见 Github 链接 [jquery/src/ajax/jsonp.js](#)

5.5 Axios 源码中的外观模式

Axios 可以使用在不同环境中，那么在不同环境中发送 HTTP 请求的时候会使用不同环境中的特有模块，Axios 这里是使用外观模式来解决这个问题的：

```
function getDefaultAdapter() {
  // ...

  if (typeof process !== 'undefined' && Object.prototype.toString.call(process) === '[object process]') {
    // Nodejs 中使用 HTTP adapter
    adapter = require('./adapters/http');
  } else if (typeof XMLHttpRequest !== 'undefined') {
    // 浏览器使用 XHR adapter
    adapter = require('./adapters/xhr');
  }

  // ...
}
```

这个方法进行了一个判断，如果在 Nodejs 的环境中则使用 Nodejs 的 HTTP 模块来发送请求，在浏览器环境中则使用 XMLHttpRequest 这个浏览器 API。

getDefaultAdapter` 方法源码参见 Github 链接 `axios/lib/defaults.js`

6. 外观模式的优缺点

外观模式的优点：

- 访问者不需要再了解子系统内部模块的功能，而只需和外观交互即可，使得访问者对子系统的使用变得简单，符合最少知识原则，增强了可移植性和可读性；
- 减少了与子系统模块的直接引用，实现了访问者与子系统中模块之间的松耦合，增加了可维护性和可扩展性；
- 通过合理使用外观模式，可以帮助我们更好地划分系统访问层次，比如把需要暴露给外部的功能集中到外观中，这样既方便访问者使用，也很好地隐藏了内部的细节，提升了安全性；

外观模式的缺点：

- 不符合开闭原则，对修改关闭，对扩展开放，如果外观模块出错，那么只能通过修改的方式来解决问题，因为外观模块是子系统的唯一出口；
- 不需要或不合理的使用外观会让人迷惑，过犹不及；

7. 外观模式的适用场景

- 维护设计粗糙和难以理解的遗留系统，或者系统非常复杂的时候，可以为这些系统设置外观模块，给外界提供清晰的接口，以后新系统只需与外观交互即可；
- 你写了若干小模块，可以完成某个大功能，但日后常用的是大功能，可以使用外观来提供大功能，因为外界也不需要了解小模块的功能；
- 团队协作时，可以给各自负责的模块建立合适的外观，以简化使用，节约沟通时间；
- 如果构建多层系统，可以使用外观模式来将系统分层，让外观模块成为每层的入口，简化层间调用，松散层间耦合；

8. 其他相关模式 8.1 外观模式与中介者模式

- 外观模式：封装子使用者对子系统内模块的直接交互，方便使用者对子系统的调用；
- 中介者模式：封装子系统间各模块之间的直接交互，松散模块间的耦合；

8.2 外观模式与单例模式

有时候一个系统只需要一个外观，比如之前举的 `Axios` 的 `HTTP` 模块例子。这时我们可以将外观模式和单例模式可以一起使用，把外观实现为单例。

#组合模式

组合模式（Composite Pattern）又叫整体-部分模式，它允许你将对象组合成树形结构来表现整体-部分层次结构，让使用者可以以一致的方式处理组合对象以及部分对象

1. 你曾见过的组合模式

大家电脑里的文件夹结构相比很熟悉了，文件夹下面可以有子文件夹，也可以有文件，子文件夹下面还可以有文件夹和文件，以此类推，共同组成了一个文件树，结构如下：

```
Folder 1
├── Folder 2
|   ├── File 1.txt
|   ├── File 2.txt
|   └── File 3.txt
└── Folder 3
    ├── File 4.txt
    ├── File 5.txt
    └── File 6.txt
```

文件夹是树形结构的容器节点，容器节点可以继续包含其他容器节点，像树枝上还可以有其他树枝一样；也可以包含文件，不再增加新的层级，就像树的叶子一样处于末端，因此被称为叶节点。本文中，叶节点又称为叶对象，容器节点因为可以包含容器节点和非容器节点，又称为组合对象。

- 类似这样的结构还有公司的组织层级，比如企业下面可以有部门，部门有部门员工和科室，科室可以有科室员工和团队，团队下面又可以有团队员工和组，依次类推，共同组成完整的企业。还有生活中的容器，比如柜子可以直接放东西，也可以放盆，盆里可以放东西也可以放碗，以此类推。甚至我们的家庭结构也属于这种结构，祖父家庭有父亲家庭、伯伯家庭、叔叔家庭、姑姑家庭等，父亲家庭又有哥哥家庭、弟弟家庭，这也是很典型的整体-部分层次的结构。
- 当我们在某个文件夹下搜索某个文件的时候，通常我们希望搜索的结果包含组合对象的所有子孙对象；开家族会议的时候，开会的命令会被传达到家族中的每一个成员；领导希望我们 996 的时候，只要跟部门领导说一声，部门领导就会通知所有的员工来修福报，无论你是下属哪个组织的，都跑不掉... 😊

在类似的场景中，有以下特点：

- 结构呈整体-部分的树形关系，整体部分一般称为组合对象，组合对象下还可以有组合对象和叶对象；
- 组合对象和叶对象有一致的接口和数据结构，以保证操作一致；
- 请求从树的最顶端往下传递，如果当前处理请求的对象是叶对象，叶对象自身会对请求作出相应的处理；如果当前处理的是组合对象，则遍历其下的子节点（叶对象），将请求继续传递给这些子节点；

2. 实例的代码实现

我们可以使用 JavaScript 来将之前的文件夹例子实现一下。

在本地一个「电影」文件夹下有两个子文件夹「漫威英雄电影」和「DC英雄电影」，分别各自有一些电影文件，我们要做的就是在这个电影文件夹里找大于 2G 的电影文件，无论是在这个文件夹下还是在子文件夹下，并输出它的文件名和文件大小。

```
/* 创建文件夹 */
var createFolder = function(name) {
    return {
```

```
name: name,
_children: [],

/* 在文件夹下增加文件或文件夹 */
add(fileOrFolder) {
    this._children.push(fileOrFolder)
},

/* 扫描方法 */
scan(cb) {
    this._children.forEach(function(child) {
        child.scan(cb)
    })
}
}

/* 创建文件 */
var createFile = function(name, size) {
    return {
        name: name,
        size: size,

        /* 在文件下增加文件，应报错 */
        add() {
            throw new Error('文件下面不能再添加文件')
        },
        /* 执行扫描方法 */
        scan(cb) {
            cb(this)
        }
    }
}

var foldMovies = createFolder('电影')

// 创建子文件夹，并放入根文件夹
var foldMarvelMovies = createFolder('漫威英雄电影')
foldMovies.add(foldMarvelMovies)

var foldDCMovies = createFolder('DC英雄电影')
foldMovies.add(foldDCMovies)

// 为两个子文件夹分别添加电影
foldMarvelMovies.add(createFile('钢铁侠.mp4', 1.9))
foldMarvelMovies.add(createFile('蜘蛛侠.mp4', 2.1))
foldMarvelMovies.add(createFile('金刚狼.mp4', 2.3))
foldMarvelMovies.add(createFile('黑寡妇.mp4', 1.9))
foldMarvelMovies.add(createFile('美国队长.mp4', 1.4))

foldDCMovies.add(createFile('蝙蝠侠.mp4', 2.4))
foldDCMovies.add(createFile('超人.mp4', 1.6))

console.log('size 大于2G的文件有: ')
foldMovies.scan(function(item) {
    if (item.size > 2) {
        console.log('name:' + item.name + ' size:' + item.size + 'GB')
    }
})
```

```
        }
    })

// size 大于2G的文件有:
// name:蜘蛛侠.mp4 size:2.1GB
// name:金刚狼.mp4 size:2.3GB
// name:蝙蝠侠.mp4 size:2.4GB
```

作为灵活的 JavaScript，我们还可以使用链模式来进行改造一下，让我们添加子文件更加直观和方便。对链模式还不熟悉的同学可以看一下后面有一篇单独介绍链模式的文章～

```
/* 创建文件夹 */
const createFolder = function(name) {
    return {
        name: name,
        _children: [],

        /* 在文件夹下增加文件或文件夹 */
        add(...fileOrFolder) {
            this._children.push(...fileOrFolder)
            return this
        },

        /* 扫描方法 */
        scan(cb) {
            this._children.forEach(child => child.scan(cb))
        }
    }
}

/* 创建文件 */
const createFile = function(name, size) {
    return {
        name: name,
        size: size,

        /* 在文件下增加文件，应报错 */
        add() {
            throw new Error('文件下面不能再添加文件')
        },

        /* 执行扫描方法 */
        scan(cb) {
            cb(this)
        }
    }
}

const foldMovies = createFolder('电影')
    .add(
        createFolder('漫威英雄电影')
            .add(createFile('钢铁侠.mp4', 1.9))
            .add(createFile('蜘蛛侠.mp4', 2.1))
            .add(createFile('金刚狼.mp4', 2.3))
            .add(createFile('黑寡妇.mp4', 1.9))
            .add(createFile('美国队长.mp4', 1.4)),
        createFolder('DC英雄电影')
```

```

        .add(createFile('蝙蝠侠.mp4', 2.4))
        .add(createFile('超人.mp4', 1.6))
    )

console.log('size 大于2G的文件有: ')

foldMovies.scan(item => {
    if (item.size > 2) {
        console.log(`name:${ item.name } size:${ item.size }GB`)
    }
})

// size 大于2G的文件有:
// name:蜘蛛侠.mp4 size:2.1GB
// name:金刚狼.mp4 size:2.3GB
// name:蝙蝠侠.mp4 size:2.4GB

```

上面的代码比较 JavaScript 特色，如果我们使用传统的类呢，也是可以实现的，下面使用 ES6 的 class 语法来改写一下：

```

/* 文件夹类 */
class Folder {
    constructor(name, children) {
        this.name = name
        this.children = children
    }

    /* 在文件夹下增加文件或文件夹 */
    add(...fileOrFolder) {
        this.children.push(...fileOrFolder)
        return this
    }

    /* 扫描方法 */
    scan(cb) {
        this.children.forEach(child => child.scan(cb))
    }
}

/* 文件类 */
class File {
    constructor(name, size) {
        this.name = name
        this.size = size
    }

    /* 在文件下增加文件，应报错 */
    add(...fileOrFolder) {
        throw new Error('文件下面不能再添加文件')
    }

    /* 执行扫描方法 */
    scan(cb) {
        cb(this)
    }
}

```

```

const foldMovies = new Folder('电影', [
    new Folder('漫威英雄电影', [
        new File('钢铁侠.mp4', 1.9),
        new File('蜘蛛侠.mp4', 2.1),
        new File('金刚狼.mp4', 2.3),
        new File('黑寡妇.mp4', 1.9),
        new File('美国队长.mp4', 1.4)])
    new Folder('DC英雄电影', [
        new File('蝙蝠侠.mp4', 2.4),
        new File('超人.mp4', 1.6)])
])
console.log('size 大于2G的文件有: ')

foldMovies.scan(item => {
    if (item.size > 2) {
        console.log(`name:${ item.name } size:${ item.size }GB`)
    }
})

// size 大于2G的文件有:
// name:蜘蛛侠.mp4 size:2.1GB
// name:金刚狼.mp4 size:2.3GB
// name:蝙蝠侠.mp4 size:2.4GB

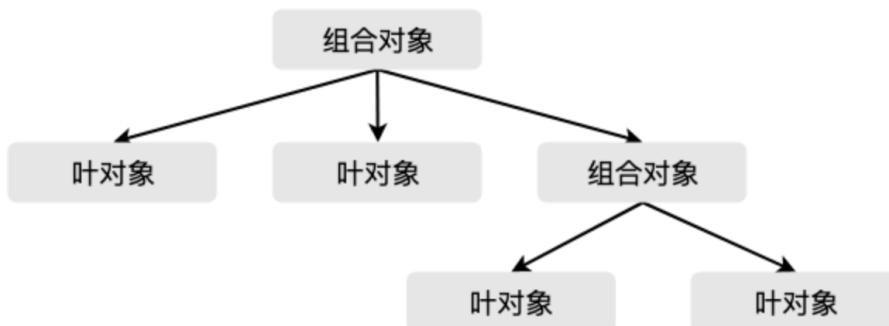
```

在传统的语言中，为了保证叶对象和组合对象的外观一致，还会让他们实现同一个抽象类或接口。

3. 组合模式的概念

- 组合模式定义的包含组合对象和叶对象的层次结构，叶对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，这样不断地组合下去。
- 在实际使用时，任何用到叶对象的地方都可以使用组合对象了。使用者可以不在意到底处理的节点是叶对象还是组合对象，也就不用写一些判断语句，让客户可以一致地使用组合结构的各节点，这就是所谓面向接口编程，从而减少耦合，便于扩展和维护。

组合模式的示意图如下：



4. 实战中的组合模式

类似于组合模式的结构其实我们经常碰到，比如浏览器的 DOM 树，从 根节点到、、