六星教育 WEB 前端面试圣经

一个页面输入 URL 到页面加载显示完成，中间发生了什么？

浏览器和向服务器发起一个 HTTP 请求

要进行 DNS 解析

先找浏览器缓存 chrome://net-internals/#dns

再找操作系统缓存

读取本地 HOST 文件

发起网络 DNS 请求调用

得到最后的服务器 IP

客户端启动一个随机端口 ,经过三次握手进入到服务器的网卡。(cdm 上可以输入 netstat, 可以看到本机开启的随机端口，小于 65535 的)

浏览器就可以发送真正的请求

服务器接收到 HTTP 请求，解析路径和参数，经后台的处理完成响应。

浏览器可以收到服务器端的响应，并开始渲染页面。DOM 树+CSS 树=Render Tree

添加用户与界面的交互，绑定一些事件，执行一些动态的行为.

还可刷新一些局部内容。

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 面 title 要有所不

同； description 把 面内容高度概括， 度合适，不可过分堆砌关键词，不同面

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

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 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

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 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 面底部
- 异步加载不是太重要的资源
- 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

六星教育 WEB 前端面试圣经

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

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

网页渲染优化

禁止使用 iframe （阻塞父文档 onload 事件）

- iframe 会阻塞主 面的 Onload 事件
- 搜索引擎的检索程序无法解读这种 面，不利于 SEO
- iframe 和主 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 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

面头部的<style></style> <script></script> 会阻塞 面；（因为 Renderer 进程中 JS 线程和渲染线程是互斥的）

面中空的 href 和 src 会阻塞 面其他资源的加载 (阻塞下载进程)

网 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

你做的页面在哪些流览器测试过？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)

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 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

为什么利用多个域名来存储网站资源会更有效?

DN 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 面响应速度

防止不必要的安全问题

六星教育 WEB 前端面试圣经

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　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

一个　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　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　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。

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

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,dl,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

简单聊下 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。

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 面刷新，重新发请求,等请求完， 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， 面不刷新，新内容也会出现，用户看到新内容

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可好，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

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
```

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

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

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”

六星教育 WEB 前端面试圣经

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

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

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

对原生Javascript 了解程度

六星教育 WEB 前端面试圣经

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

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

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 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 度。读取属性（数组 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面上 的选择器相关元素，速度会大大降低

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 面底部

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 时，直接从服务器加载，跳过强缓存和协商缓存；

当 f5 刷新网 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

六星教育 WEB 前端面试圣经

谈一谈　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

谈一谈 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。

谈一谈　内存泄漏

定义：程序中己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 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

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

cookie 和 localSrorage、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	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	每次都会携带在	不参与	不参与	不参与

特性	cookie	localStorage	sessionStorage	indexDB
务端通信	header 中，对于请求性能影响			

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
value	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
http-only	不能通过 JS 访问 Cookie ，减少 XSS 攻击
secure	只能在协议为 HTTPS 的请求中携带
same-site	规定浏览器不能在跨域请求中携带 Cookie ，减少 CSRF 攻击

怎么判断 面是否加载完成

Load 事件触发代表 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

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 面，可以在开发者工具中的 Application 看到 Service Worker 已经启动了

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 ，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>
```

```
12.
13.         <script>
14.         setTimeout(() => {
15.         // 引起回流
16.         document.querySelector('.test').style.top = '100px'
17.         }, 1000)
18.         </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 点的属性值放在一个循环里当成循环里的变量

```
1.          for(let i = 0; i < 1000; i++) {
2.          // 获取 offsetTop 会导致回流，因为需要去获取正确的值
3.          console.log(document.querySelector('.test').style.offsetTop)
4.          }
```

  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

浏览器的渲染机制

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 ，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

如何前端页面的安全

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 上，其他使用者在观看网 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

六星教育 WEB 前端面试圣经

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, "&quto;");
    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>')
```

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&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

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 网站中加入一个图片，图片的地址就是评论接口

```
<img src="http://www.domain.com/xxx?comment='attack'"/>
```

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 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

谈一谈 MVVM 这种框架

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 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 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 Angluar 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 仁智者 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 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。

```
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
```

```
21.             switch (method) {
22.                 case 'push':
23.                 case 'unshift':
24.                     inserted = args
25.                     break
26.                 case 'splice':
27.                     inserted = args.slice(2)
28.                     break
29.             }
30.             if (inserted) ob.observeArray(inserted)
31.             // 触发更新
32.             ob.dep.notify()
33.             return result
34.         })
35.         })
```

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
```

谈一谈框架的路由原理

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 面，并且无须刷新。目前单 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 面。

History 模式是 HTML5 新推出的功能，比之 Hash URL 更加美观

谈一谈框架的 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.ke
    }
```

```
    // 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]
                el.setAttribute(key, value)
            }
        }
        if (key) {
            el.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)
        }
        el.appendChild(child)
    })
}
return el
}
}
```

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 = {}
```

```
6.          // 一开始的索引为 0
7.          dfs(oldDomTree, newDomTree, 0, pathchs)
8.          return pathchs
9.          }
10.         function dfs(oldNode, newNode, index, patches) {
11.         // 用于保存子树的更改
12.         let curPatches = []
13.         // 需要判断三种情况
14.         // 1.没有新的节点，那么什么都不用做
15.         // 2.新的节点的 tagName 和 `key` 和旧的不同，就替换
16.         // 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
17.         if (!newNode) {
18.
19.         } else if (newNode.tag === oldNode.tag && newNode.key ===
oldNode.key) {
20.             // 判断属性是否变更
21.             let props = diffProps(oldNode.props, newNode.props)
22.             if (props.length) curPatches.push({ type:
StateEnums.ChangeProps, props
23.                 // 遍历子树
```

```
24.                 diffChildren(oldNode.children,  newNode.children,  index,
patches)
25.             } else {
26.                 // 节点不同，需要替换
27.                 curPatches.push({  type:  StateEnums.Replace,  node:
newNode })
28.             }
29.             if (curPatches.length) {
30.                 if (patches[index]) {
31.                     patches[index] = patches[index].concat(curPatches)
32.                 } else {
33.                     patches[index] = curPatches
34.                 }
35.             }
36.         }
37.     }
```

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，且让我一一为你道来。 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
-
-

```
1.
2.              function diffChildren(oldChild, newChild, index, patches) {
3.              let { changes, list } = listDiff(oldChild, newChild, index, patches)
4.              if (changes.length) {
5.                  if (patches[index]) {
6.                      patches[index] = patches[index].concat(changes)
7.                  } else {
8.                      patches[index] = changes
9.                  }
10.             }
11.             // 记录上一个遍历过的节点
12.             let last = null
13.             oldChild &&
14.             oldChild.forEach((item, i) => {
15.                 let child = item && item.children
16.                 if (child) {
17.                     index = last && last.children ? index + last.children.length +
1 : index
```

```
            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
```

```
6.              if (!childNodes) index += 1
7.              if (changes && changes.length && patchs[index]) {
8.                  changeDom(node, changes)
9.              }
10.             let last = null
11.             if (childNodes && childNodes.length) {
12.                 childNodes.forEach((item, i) => {
13.                     index = last && last.children ? index + last.children.length +
1 : index +
14.                     patch(item, patchs)
15.                     last = item
16.                 })
17.             }
18.             }
19.             function changeDom(node, changes, noChild) {
20.             changes &&
21.             changes.forEach(change => {
22.                 let { type } = change
23.                 switch (type) {
24.                     case StateEnums.ChangeProps:
25.                         let { props } = change
26.                         props.forEach(item => {
```

```
27.                     if (item.value) {
28.                         node.setAttribute(item.prop, item.value)
29.                     } else {
30.                         node.removeAttribute(item.prop)
31.                     }
32.                 })
33.                 break
34.             case StateEnums.Remove:
35.                 node.childNodes[change.index].remove()
36.                 break
37.             case StateEnums.Insert:
38.                 let dom
39.                 if (isString(change.node)) {
40.                     dom = document.createTextNode(change.node)
41.                 } else if (change.node instanceof Element) {
42.                     dom = change.node.create()
43.                 }
44.                 node.insertBefore(dom,
node.childNodes[change.index])
45.                 break
46.             case StateEnums.Replace:
```

```
47.                 node.parentNode.replaceChild(change.node.create(),
node)
48.                 break
49.             case StateEnums.Move:
50.                 let fromNode = node.childNodes[change.from]
51.                 let toNode = node.childNodes[change.to]
52.                 let cloneFromNode = fromNode.cloneNode(true)
53.                 let cloenToNode = toNode.cloneNode(true)
54.                 node.replaceChild(cloneFromNode, toNode)
55.                 node.replaceChild(cloenToNode, fromNode)
56.                 break
57.             default:
58.                 break
59.         }
60.     })
61.     }
```

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 pathchs = diff(test1, test2)
console.log(pathchs)
setTimeout(() => {
console.log('开始更新')
patch(root, pathchs)
console.log('结束更新')
}, 1000)
```

浏览器渲染原理

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 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。

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 ）

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 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

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 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 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，这样肯定会造成 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 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 ，但是渲染的 DOM 元素永远只有那么 几个，当我们滚动 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会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 Eventloop 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 ，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

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>
```

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)
}
```

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，避免节点层级过多

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

从 V8 中看 JS 性能优化

1 测试性能工具

Chrome 已经提供了一个大而全的性能测试工具 Audits

点我们点击 Audits 后，可以看到如下的界面

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 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

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

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，他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 ，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 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 test1() {
// 会被重复解析
function test2() {}
}
function test(x) {
    return x + x
}
test(3)
test(4)
test(1)
test(2)
```

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 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

WEB 前端有哪些性能优化

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：

1. 减少像素点
2. 减少每个像素点能够显示的颜色

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

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">
```

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
)
```

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)
        }
    }
```

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 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.poetries.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。

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.poetries.top">
```

预渲染虽然可以提高 面的加载速度，但是要确保该 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。

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。

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等。

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 ，平白消耗流量

六星教育 WEB 前端面试圣经