目录

[自我介绍 6](#_Toc5223975)

[项目稿子 6](#_Toc5223976)

[Node.js实现的区块链 6](#_Toc5223977)

[住房维护与维修系统 6](#_Toc5223978)

[GC－AI游戏对战平台 7](#_Toc5223979)

[操作系统 8](#_Toc5223980)

[进程线程区别 8](#_Toc5223981)

[进程间通信机制 8](#_Toc5223982)

[连续分配存储管理方式 9](#_Toc5223983)

[分区式存储管理 9](#_Toc5223984)

[覆盖和交换 10](#_Toc5223985)

[页式存储管理 10](#_Toc5223986)

[段式存储管理 13](#_Toc5223987)

[页式和段式的区别 15](#_Toc5223988)

[算法与数据结构 15](#_Toc5223989)

[排序算法 15](#_Toc5223990)

[冒泡排序 16](#_Toc5223991)

[选择排序 16](#_Toc5223992)

[插入排序 17](#_Toc5223993)

[希尔排序 17](#_Toc5223994)

[归并排序 17](#_Toc5223995)

[快速排序 17](#_Toc5223996)

[堆排序 19](#_Toc5223997)

[计数排序 20](#_Toc5223998)

[桶排序 20](#_Toc5223999)

[基数排序 20](#_Toc5224000)

[洗牌算法 21](#_Toc5224001)

[圆上选n个点在同一个半圆上的概率 21](#_Toc5224002)

[O(n)找到右边第一个比它大的数 21](#_Toc5224003)

[JavaScript 实现bind 22](#_Toc5224004)

[JavaScript 实现Promise 22](#_Toc5224005)

[JavaScript 模拟实现new 23](#_Toc5224006)

[JavaScript 实现call 与 apply 24](#_Toc5224007)

[JavaScript 深拷贝 25](#_Toc5224008)

[Git 25](#_Toc5224009)

[Webpack 26](#_Toc5224010)

[loader plugin 26](#_Toc5224011)

[tree shaking 26](#_Toc5224012)

[HMR 26](#_Toc5224013)

[如何处理依赖 26](#_Toc5224014)

[常用的loader 26](#_Toc5224015)

[bundle原理 26](#_Toc5224016)

[JavaScript 26](#_Toc5224017)

[原型与原型链，继承 26](#_Toc5224018)

[箭头函数与普通函数的区别 27](#_Toc5224019)

[Promise，async，await，generator 27](#_Toc5224020)

[Eventloop 27](#_Toc5224021)

[数组方法，map，reduce，forEach 28](#_Toc5224022)

[commonJS AMD CMD ES6用的哪一种 28](#_Toc5224023)

[this与call bind apply 28](#_Toc5224024)

[变量提升 28](#_Toc5224025)

[闭包 28](#_Toc5224026)

[绑定事件 28](#_Toc5224027)

[事件代理 29](#_Toc5224028)

[冒泡和捕获、捕获能否停止，阻止事件冒泡 29](#_Toc5224029)

[函数节流与防抖 29](#_Toc5224030)

[Node.js 30](#_Toc5224031)

[Node.js 事件循环，定时器和 process.nextTick() 30](#_Toc5224032)

[Express常用中间件 31](#_Toc5224033)

[CSS 32](#_Toc5224034)

[优先级 32](#_Toc5224035)

[预处理器 32](#_Toc5224036)

[命名规范 32](#_Toc5224037)

[水平垂直居中 33](#_Toc5224038)

[行内元素水平居中 33](#_Toc5224039)

[块级元素水平居中 33](#_Toc5224040)

[多个块级元素水平居中 33](#_Toc5224041)

[盒模型（标准盒模型与IE下的盒模型区别）,BFC 33](#_Toc5224042)

[BFC 33](#_Toc5224043)

[position 35](#_Toc5224044)

[translate 35](#_Toc5224045)

[游览器最小显示的字体大小，如何更小 35](#_Toc5224046)

[移动端兼容 35](#_Toc5224047)

[浮动的理解，清除浮动 36](#_Toc5224048)

[CSS reset 36](#_Toc5224049)

[轮播图 36](#_Toc5224050)

[Sprit 图 36](#_Toc5224051)

[Flex 36](#_Toc5224052)

[grid 37](#_Toc5224053)

[animation 37](#_Toc5224054)

[canvas SVG 37](#_Toc5224055)

[React 37](#_Toc5224056)

[Virtual DOM & Diff 37](#_Toc5224057)

[React.setState() 37](#_Toc5224058)

[无状态组件与PureComponent 38](#_Toc5224059)

[组件通信，兄弟通信 38](#_Toc5224060)

[Redux 38](#_Toc5224061)

[React生命周期 39](#_Toc5224062)

[React如何优化 39](#_Toc5224063)

[react-router 39](#_Toc5224064)

[React事件机制 40](#_Toc5224065)

[HTML 41](#_Toc5224066)

[前端路由 history 41](#_Toc5224067)

[Get与Post区别 41](#_Toc5224068)

[浏览器缓存机制（强缓存与协商缓存区别，优先级） 41](#_Toc5224069)

[状态码 301，302，304 42](#_Toc5224070)

[Cookie，sessionStorage，localStorage 44](#_Toc5224071)

[HTML5 新标签 45](#_Toc5224072)

[<script> 标签中的defer 与 async 47](#_Toc5224073)

[document.readystate中的状态 47](#_Toc5224074)

[HTTP2.0 48](#_Toc5224075)

[多路复用与长连接复用 48](#_Toc5224076)

[头部压缩和二进制格式 48](#_Toc5224077)

[服务端推送Server Push 49](#_Toc5224078)

[网络 49](#_Toc5224079)

[TCP三次握手，四次挥手 49](#_Toc5224080)

[三次握手 49](#_Toc5224081)

[四次挥手 50](#_Toc5224082)

[HTTPS过程 52](#_Toc5224083)

[其它 53](#_Toc5224084)

[MVVM缺点 53](#_Toc5224085)

[Jsonp、CORS跨域 53](#_Toc5224086)

[跨域解决办法 54](#_Toc5224087)

[浏览器线程 55](#_Toc5224088)

[eslint 56](#_Toc5224089)

[docker 56](#_Toc5224090)

[前端性能优化 56](#_Toc5224091)

[游览器渲染过程，重绘重排 56](#_Toc5224092)

[单点登录 57](#_Toc5224093)

[同父域，不同子域 57](#_Toc5224094)

[不同域父域 58](#_Toc5224095)

[负载均衡、分布式 59](#_Toc5224096)

[CSRF、XSS 60](#_Toc5224097)

# 自我介绍

您好， 很荣幸能够参加xx的面试。我是浙江大学计算机学院软件工程专业的曾辉。目前大三，学校的主修可能基本已经修完，所以我大概有6-8个月的实习时间。我平时主要关注的是前端开发领域，常用的框架是React以及相对应的组件与库，课程与课余时间也做过一些简单的实验性的项目，用于增强自己的开发能力与团队协作能力。大概就是这些。

# 项目稿子

## Node.js实现的区块链

项目主要是基于Express框架，Node.js的目的是搭建一个P2P网络。

基于Socket.io，能够双向通信，每个节点实现一个客户端与服务端，客户端启动时自动连接附近的节点的服务端，同时相应节点的客户端也与本地服务端连接，自此一个双向通信的网络建立。

挖矿时新建子进程，进行工作量证明，计算完成后广播区块

## 住房维护与维修系统

项目为小组合作项目，开发小组一共六人。

该项目是由老师牵头，联系美国一个房地产开发商，获取相应的需求后进行着手开发，作为其较为庞大的项目中的一个子系统，主要提供提醒用户定期检修住房中的设备，并提供相应的用户发起维修订单请求的功能

项目架构为，服务端，客户端，服务商界面

我与另外几名同学负责客户端的研发。

亮点与难点，小组同学除我之外，均无太多的前端开发经验。任务量大，学业压力较重，并且需要输出完善的需求分析文档，设计文档，测试文档。时间紧，我确定了技术选型为React搭配相应的生态进行开发。（考虑到，后期可能会大部分任务落在我身上，于是选择了我最熟悉的一套开发，这是我唯一能够兜住的一套技术）

我对着设计稿，拆分出了几个常用的组件，比如九宫格，列表等有几个稍微有经验的同学帮忙实现了这些较为常用的组件。只是单纯的UI组件，在一定程度上减轻了我和另两位同学的工作量。

另几位负责前端的同学，表示组件通信在学习上存在一定的难度，尤其是跨组件通信，我实现了一个工具类，EventProxy，使用发布订阅者模式进行组件通信，帮助组内同学解决了这个问题。

管理上而言，我们每周会线下聚会一次，讨论统计本周完成了哪些工作，与接下来一周需要完成的工作。我们整体上有一个项目排期，但是粒度只到什么时间完成需求分析，什么时间完成设计，什么时间完成编码，什么时间能够交付。需求，设计等排期基本上按照老师的时间来，因为需要涉及到和甲方的交流，双方需要约定时间。具体项目的推进进度，主要由我来把控，我凭借我自身的开发进度预判整个项目的进度。

目前顺利完成答辩，并且取得了整个专业的最高分，寒假的时候在协商并入其主系统相关事宜。

## GC－AI游戏对战平台

项目为社团项目，本学期计划上线

负责前端

采用React + Material-UI + Redux + React-Router进行开发

前后端分离，通过RESTful API进行交互，后端采用Go语言开发，整合了Swagger，前端mock

目前主要逻辑已经通畅

难点与亮点：

主要是进行Redux 的实验，对于Redux状态管理有了一个较为清晰的理解，对于其中间件的作用有了一个较为清晰的认知，模仿Redux官方样例，写了一个发起异步请求的中间件

页面中需要嵌入Unity编译出来的游戏页面，采用iframe的方式嵌入。

游戏需要展示对战过程，前端从后端获取相应的对战过程数据，传入Unity进行展示，一开始不知道如何处理这一过程，通过查阅相应的文档，发现Unity编译出来的游戏在UnityLoader.js中会暴露全局方法的接口，采用iframe的方式嵌入，通过ref的方式取到iframe的对象，获得其相应的contentWindow，调用暴露出来的全局方法进行通信。

# 操作系统

## 进程线程区别

## 进程间通信机制

* 管道
* 信号量
* 共享内存

## 连续分配存储管理方式

### 分区式存储管理

1. 固定分区

把内存划分为若干个固定大小的连续分区（不一定等分），每个分区只能容纳一个进程

优点：易于实现，开销小

缺点：内碎片造成浪费；分区总数固定，限制了并发执行的程序数目

1. 动态分区

没有内碎片，但有外碎片。分区算法：（从低的一端到高的一端）

1. first-fit：分配和释放的性能较好，但会产生较多小分区
2. next-fit：使空闲分区分布的更均匀，但较大空闲分区不易保留
3. best-fit：会形成较多外碎片，优点是较大的空闲分区可以保留
4. worst-fit：不易形成外碎片，但缺少较大的空闲分区
5. 伙伴系统

伙伴系统是固定分区和动态分区的折中。规定，无论已分配分区或空闲分区，其大小均为 2 的 k 次幂，k 为整数， l≤k≤m。假设系统的可利用空间容量为2^m个字，则系统开始运行时， 整个内存区是一个大小为2^m的空闲分区。在系统运行过中，由于不断的划分，可能会形成若干个不连续的空闲分区，将这些空闲分区根据分区的大小进行分类，对于每一类具有相同大小的所有空闲分区，单独设立一个空闲分区双向链表。这样，不同大小的空闲分区形成了k(0≤k≤m)个空闲分区链表。 空间性能比较好，时间性能差不多。

1. 内存紧缩（解决外部碎片）

将各个占用分区向内存一段移动，然后将各个空闲分区合并成一个空闲分区。

紧缩仅在重定位是动态并在运行时可采用。

紧缩时机：每个分区释放后，或内存分配找不到满足条件的空闲分区时。

堆结构的存储管理的分配算法很简单，只要移动堆指针即可（堆不管什么时刻，可利用空间都是一个地址连续的存储区）

## 覆盖和交换

1. 覆盖

覆盖是为了在较小的可用内存中运行较大的程序。

程序必要部分常驻内存，可选部分需要时装入，不需要时被其他可选部分覆盖。

如在同一时刻，CPU只能执行B，C中某一条。B，C之间就可以做覆盖。

缺点：编程时必须划分程序模块和确定程序模块之间的覆盖关系；从外存转入覆盖文件，以时间换空间。

1. 交换

在多个程序并发访问时，可以将暂时不能执行的程序送到外存中，从而获得空闲内存空间来装入新进程。交换单位为整个进程的地址空间。

优点：增加并发运行的程序数目。

## 页式存储管理

1. 基本原理

将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载时，可将任意一页放人内存中任意一个页框，这些页框不必连续，从而实现了离散分配。该方法需要CPU的硬件支持，来实现逻辑地址和物理地址之间的映射。在页式存储管理方式中地址结构由两部构成，前一部分是页号，后一部分为页内地址w。

优点：

①    没有外碎片，每个内碎片不超过页大小

②    一个程序不必连续存放

③    便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)

缺点：要求程序全部装入内存，没有足够的内存，程序就不能执行。

1. 数据结构

在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。

为了完成上述的功能，—个页式系统中，一般要采用以下数据结构：

进程页表：完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间)的映射。

每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序，如图：

物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况，其数据结构可采用位图和空闲页链表。

对于位图法，即如果该页面已被分配，则对应比特位置1，否置0.

请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换也可以结合到各进程的PCB(进程控制块)里。

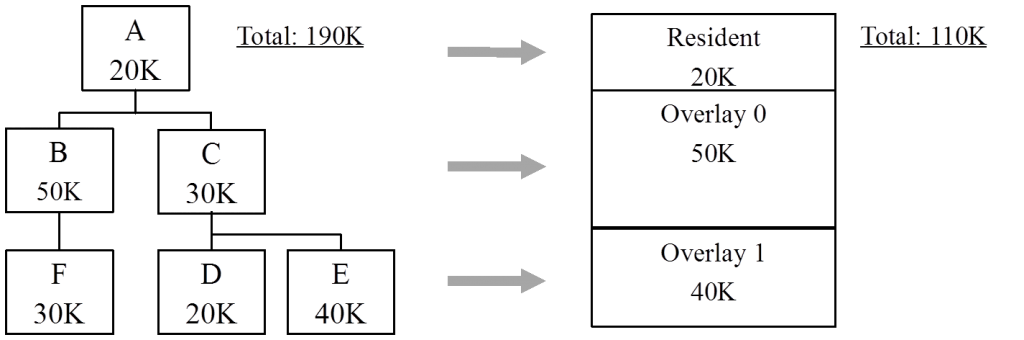
（3）  地址变换

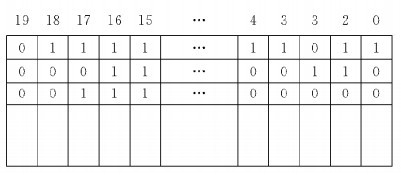
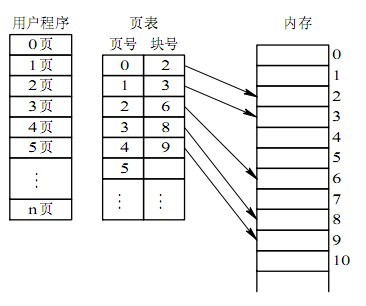
在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。

原理：CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址

上述过程通常由处理器的硬件直接完成，不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存：第一次用来查找页表将操作数的逻辑地址变换为物理地址；第二次完成真正的读写操作。

这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入CPU内部的关联存储器(例如，TLB快表) 中，实现按内容查找。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动将页号送至快表，并将此页号与快表中的所有页号进行比较，而且这 种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由于关联存储器的访问速度比内存的访问速度快得多。





## 段式存储管理

（1）  基本原理

在段式存储管理中，将程序的地址空间划分为若干个段(segment)，这样每个进程有一个二维的地址空间。在段式存储管理系统中，为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。

在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法等方法。

段式存储管理也需要硬件支持，实现逻辑地址到物理地址的映射。

程序通过分段划分为多个模块，如代码段、数据段、共享段：

* + - 可以分别编写和编译
    - 可以针对不同类型的段采取不同的保护
    - 可以按段为单位来进行共享，包括通过动态链接进行代码共享

这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。

优点：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。

缺点与页式存储管理的缺点相同，进程必须全部装入内存。

（2）  数据结构

为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(baseaddress)，即段内地址。在系统中为每个进程建立一张段映射表。

系统段表：系统所有占用段（已经分配的段）。

空闲段表：内存中所有空闲段，可以结合到系统段表中。

（3）  地址变换

在段式管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址。这个过程也是由处理器的硬件直接完成的，操作系统只需在进程切换时，将【进程段表】的首地址装入处理器的特定寄存器当中。这个寄存器一般被称作段表地址寄存器。

## 页式和段式的区别

页式和段式系统有许多相似之处。比如，两者都采用离散分配方式，且都通过地址映射机构来实现地址变换。但概念上两者也有很多区别，主要表现在：

① 需求：页是信息的物理单位，分页是为了实现离散分配方式，以减少内存的碎片，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了更好地满足用户的需要。

一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处。

② 大小：页大小固定且由系统决定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的。段的长度不固定，且决定于用户所编写的程序，通常由编译系统在对源程序进行编译时根据信息的性质来划分。

③ 逻辑地址表示：页式系统地址空间是一维的，即单一的线性地址空间，程序员只需利用一个虚拟地址即可确定物理地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

④ 比页大，因而段表比页表短，可以缩短查找时间，提高访问速度。

# 算法与数据结构

## 排序算法

参考资料： <https://www.cnblogs.com/onepixel/articles/7674659.html>

<https://github.com/francistao/LearningNotes/blob/master/Part3/Algorithm/Sort/%E9%9D%A2%E8%AF%95%E4%B8%AD%E7%9A%84%2010%20%E5%A4%A7%E6%8E%92%E5%BA%8F%E7%AE%97%E6%B3%95%E6%80%BB%E7%BB%93.md>

* 比较类排序：通过比较来决定元素间的相对次序。时间复杂度不能超过O(nlogn)
* 非比较类排序：突破基于比较排序的时间下界，以线性时间运行
* 稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面。
* 不稳定：如果a原本在b的前面，而a=b，排序之后 a 可能会出现在 b 的后面。

### 冒泡排序

可以在最好情况实现O（n）复杂度的代码：

public void bubbleSort(int arr[]) {

boolean didSwap;

*for* (int i = 0, len = arr.length; i < len - 1; i++) {

didSwap = false;

*for* (int j = 0; j < len - i - 1; j++) {

*if* (arr[j + 1] < arr[j]) {

swap(arr, j, j + 1);

didSwap = true;

}

}

*if* (didSwap == false)

*return*;

}

}

### 选择排序

工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

不稳定，例如5，8，5，2，9

### 插入排序

工作原理：是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

### 希尔排序

第一个突破O(n2)的排序算法，是简单插入排序的改进版。按照增量将序列分成多个组，每个组进行插入排序，然后减小增量重复这个操作。

算法描述：

选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

按增量序列个数k，对序列进行k 趟排序；

每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

### 归并排序

算法描述（分治法）：

把长度为n的输入序列分成两个长度为n/2的子序列；

对这两个子序列分别采用归并排序；

将两个排序好的子序列合并成一个最终的排序序列。

### 快速排序

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

从数列中挑出一个元素，称为 “基准”（pivot）；

重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

public static void quickSort(int[] arr) {

qsort(arr, 0, arr.length-1);

}

private static void qsort(int[] arr, int low, int high) {

*if* (low < high) {

int pivot=partition(arr, low, high); *//将数组分为两部分*

qsort(arr, low, pivot-1); *//递归排序左子数组*

qsort(arr, pivot+1, high); *//递归排序右子数组*

}

}

private static int partition(int[] arr, int low, int high) {

int pivot = arr[low]; *//枢轴记录*

*while* (low < high) {

*while* (low < high && arr[high] >= pivot) --high;

arr[low]=arr[high]; *//交换比枢轴小的记录到左端*

*while* (low < high && arr[low] <= pivot) ++low;

arr[high] = arr[low]; *//交换比枢轴小的记录到右端*

}

*//扫描完成，枢轴到位*

arr[low] = pivot;

*//返回的是枢轴的位置*

*return* low;

}

#### 快排的优化

① 选取基准：三数取中：选择左端、右端和中心位置的三个元素的中值作为枢纽元。

② 对小数组（N<=20）使用插入排序。

③ 两端变三段。

④ 快速排序需要一个递归栈，通常情况下这个栈不会很深，为log(n)级别。但是，如果每次划分的两个数组长度严重失衡，则为最坏情况，栈的深度将增加到O(n)。此时，由栈空间带来的空间复杂度不可忽略。如果加上额外变量的开销，这里甚至可能达到恐怖的O(n^2)空间复杂度。所以，快速排序的最差空间复杂度不是一个定值，甚至可能不在一个级别。为了解决这个问题，我们可以在每次划分后比较两端的长度，并先对短的序列进行排序（目的是先结束这些栈以释放空间），可以将最大深度降回到O(㏒n)级别。

⑤ 快排的非递归实现（用自己实现的栈来模拟递归）

### 堆排序

升序排序用大顶堆，反之使用小顶堆。原因是堆顶元素需要交换到序列尾部。

首先，实现堆排序需要解决两个问题：

1. 如何由一个无序序列键成一个堆？

2. 如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

第一个问题，可以直接使用线性数组来表示一个堆，由初始的无序序列建成一个堆就需要自底向上从第一个非叶元素开始挨个调整成一个堆。从一个无序序列建堆的过程就是一个反复筛选的过程。若将此序列看成是一个完全二叉树，则最后一个非终端节点是n/2取底个元素，由此下滤

第二个问题，怎么调整成堆？首先是将堆顶元素和最后一个元素交换。然后比较当前堆顶元素的左右孩子节点，因为除了当前的堆顶元素，左右孩子堆均满足条件，这时需要选择当前堆顶元素与左右孩子节点的较大者（大顶堆）交换，直至叶子节点。我们称这个自堆顶自叶子的调整成为筛选。

### 计数排序

用待排序的数作为计数数组的下标，统计每个数字的个数。然后依次输出即可得到有序序列。

### 桶排序

假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排）。然后各个桶依照次序输出就是排好序的结果。

映射函数必须做到：如果关键字k1<k2，那么f(k1)<=f(k2)。

对于相同的N，桶数量M越大，其效率越高。

### 基数排序

基数排序又是一种和前面排序方式不同的排序方式，基数排序不需要进行记录关键字之间的比较。基数排序是一种借助多关键字排序思想对单逻辑关键字进行排序的方法。所谓的多关键字排序就是有多个优先级不同的关键字。比如说成绩的排序，如果两个人总分相同，则语文高的排在前面，语文成绩也相同则数学高的排在前面。如果对数字进行排序，那么个位、十位、百位就是不同优先级的关键字，如果要进行升序排序，那么个位、十位、百位优先级一次增加。基数排序是通过多次的收分配和收集来实现的，关键字优先级低的先进行分配和收集。

应用场景

1. 若n较小(如n≤50)，可采用直接插入或直接选择排序。

当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

1. 若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜；
2. 若n较大，则应采用时间复杂度为O(nlgn)的排序方法：快速排序、堆排序或归并排序。

快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况。这两种排序都是不稳定的。

若要求排序稳定，则可选用归并排序。但前面介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子序列，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。

## 洗牌算法

随机生成一个数，将该数取余剩余数组长度，将余数下标所对应的元素与未处理的最后一个元素交换位置，最后一个元素视作已经处理，重复该过程，直到处理所有元素。

## 圆上选n个点在同一个半圆上的概率

P = n/(2^(n – 1))

## O(n)找到右边第一个比它大的数

从左至右将下标压栈，遇到下标对应的数据比当前压栈的小的，则弹出，弹出的下标答案即为当前压栈下标

## JavaScript 实现bind

Function.prototype.bind = function(that) {

if (typeof this !== 'function') {

throw new TypeError("cannot callable");

}

var args = Array.prototype.slice.call(arguments, 1);

var fToBind = this;

var fNOP = function() {};

var fBound = function() {

return fToBind.apply(this instanceof fBound ? this : that, args.concat(Array.prototype.slice.call(arguments)));

};

if (this.prototype) {

fNOP.prototype = this.prototype;

}

fBound.prototype = new fNOP();

return fBound;

}

## JavaScript 实现Promise

function Promise(fn) {

    var state = 'pending',

        value = null,

        callbacks = [];

    this.then = function (onFulfilled, onRejected) {

        return new Promise(function (resolve, reject) {

            handle({

                onFulfilled: onFulfilled || null,

                onRejected: onRejected || null,

                resolve: resolve,

                reject: reject

            });

        });

    };

    function handle(callback) {

        if (state === 'pending') {

            callbacks.push(callback);

            return;

        }

        var cb = state === 'fulfilled' ? callback.onFulfilled : callback.onRejected,

            ret;

        if (cb === null) {

            cb = state === 'fulfilled' ? callback.resolve : callback.reject;

            cb(value);

            return;

        }

        ret = cb(value);

        callback.resolve(ret);

    }

    function resolve(newValue) {

        if (newValue && (typeof newValue === 'object' || typeof newValue === 'function')) {

            var then = newValue.then;

            if (typeof then === 'function') {

                then.call(newValue, resolve, reject);

                return;

            }

        }

        state = 'fulfilled';

        value = newValue;

        execute();

    }

    function reject(reason) {

        state = 'rejected';

        value = reason;

        execute();

    }

    function execute() {

        setTimeout(function () {

            callbacks.forEach(function (callback) {

                handle(callback);

            });

        }, 0);

    }

    fn(resolve, reject);

}

## JavaScript 模拟实现new

function objectFactory() {

var obj = {};

var constructor = [].shift.call(arguments);

obj.\_\_proto\_\_ = constructor.prototype;

var ret = constructor.apply(obj, arguments);

return typeof ret === 'object' ? ret : obj;

}

## JavaScript 实现call 与 apply

Function.prototype.call = function (context, ...args) {

var context = context || window;

context.fn = this;

var result = context.fn(args);

delete context.fn

return result;

}

Function.prototype.call = function(context) {

var context = context || window;

context.fn = this;

var args = [];

for (int i = 1; i < arguments.length; i++)

args.push('arguments[' + i + '] ');

var result = eval('context.fn(' + args + ')');

delete context.fn;

return result;

}

Function.prototype.apply = function (context, arr) {

var context = Object(context) || window;

context.fn = this;

var result;

if (!arr) {

result = context.fn();

}

else {

var args = [];

for (var i = 0, len = arr.length; i < len; i++) {

args.push('arr[' + i + ']');

}

result = eval('context.fn(' + args + ')')

}

delete context.fn

return result;

}

## JavaScript 深拷贝

var array = [

{ number: 1 },

{ number: 2 },

{ number: 3 }

];

function copy (obj) {

var newobj = obj.constructor === Array ? [] : {};

if(typeof obj !== 'object'){

return;

}

for(var i in obj){

newobj[i] = typeof obj[i] === 'object' ?

copy(obj[i]) : obj[i];

}

return newobj

}

var copyArray = copy(array)

copyArray[0].number = 100;

console.log(array); // [{number: 1}, { number: 2 }, { number: 3 }]

console.log(copyArray); // [{number: 100}, { number: 2 }, { number: 3 }]

# Git

# Webpack

## loader plugin

## tree shaking

## HMR

## 如何处理依赖

## 常用的loader

## bundle原理

# JavaScript

## 原型与原型链，继承

普通对象[[prototype]]

函数prototype

*/\*\**

*\* 组合继承*

*\*/*

function Parent(name) {

*this*.name = name;

}

function Child(name, age) {

Parent.call(*this*, name);

*this*.age = age;

}

function inherit(Subtype, SuperType) {

var prototype = Object.create(Supertype.prototype);

prototype.constructe = Subtype;

Subtype.prototype = prototype;

}

inherit(Child, Parent);

*/\*\**

*\* 原型继承*

*\*/*

function object(o) {

function F(){}

F.prototype = o;

*return* new F();

}

## 箭头函数与普通函数的区别

* 箭头函数是普通函数的简写
* 无this argument，箭头函数不绑定this，会捕获其所在的上下文的this值，作为自己的this值
* 不能作为构造函数，不能使用new
* 不能当做Generator函数,不能使用yield关键字

## Promise，async，await，generator

解决异步编程的方案

Generator 函数

Promise对象

https://juejin.im/post/5b5d0ac5f265da0f574df709

## Eventloop

* **macro-task:** script（整体代码）, setTimeout, setInterval, setImmediate, I/O, UI rendering
* **micro-task:** process.nextTick, Promises（这里指浏览器实现的原生 Promise）, Object.observe, MutationObserver

## 数组方法，map，reduce，forEach

## commonJS AMD CMD ES6用的哪一种

1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible。
2. CMD 推崇依赖就近，AMD 推崇依赖前置

## this与call bind apply

this指向调用函数的对象，默认window，严格模式下为undefined

call，apply方法的第一个参数为调用函数的this对象

bind可以绑定作为参数的对象作为this

## 变量提升

JavaScript中，非严格模式下，变量声明会被提升至代码段的开头

## 闭包

闭包是由函数和与其相关的引用环境组合而成的实体

## 绑定事件

let eventListener = Dom.addEventListener(“click”, function(){});

clearEventListener(eventListener);

## 事件代理

将需要绑定在子元素上的事件回调函数，绑定在其父元素上，利用事件冒泡机制，与父元素上进行统一处理，减小内存的消耗。

（适合用事件委托的事件：click，mousedown，mouseup，keydown，keyup，keypress。mouseover和mouseout虽然也有事件冒泡但麻烦；focus，blur之类的没有冒泡事件）

window.onload = function(){  
　　var oUl = document.getElementById("ul1");  
　　oUl.onclick = function(ev){  
　　　　var ev = ev || window.event;  
　　　　var target = ev.target || ev.srcElement;  
　　　　if(target.nodeName.toLowerCase() == 'li'){  
　 　　　　　　 alert(123);  
　　　　　　　  alert(target.innerHTML);  
　　　　}  
　　}  
}

## 冒泡和捕获、捕获能否停止，阻止事件冒泡和事件绑定

IE提出了冒泡流；远景提出了捕获流。

**事件绑定：**

　　1、直接获取元素绑定：

element.onclick = function(e){

// ...

};

　　该方法的优点是：简单和稳定，可以确保它在你使用的不同浏览器中运作一致；处理事件时，this关键字引用的是当前元素，这很有帮助。

　　缺点：只会在事件冒泡中运行；一个元素一次只能绑定一个事件处理函数，新绑定的事件处理函数会覆盖旧的事件处理函数；事件对象参数(e)仅非IE浏览器可用

　　2、直接在元素里面使用事件属性

　　3、W3C方法：

element.addEventListener('click', function(e){

// ...

}, false);

　　优点：该方法同时支持事件处理的捕获和冒泡阶段；事件阶段取决于addEventListener最后的参数设置：false (冒泡) 或 true (捕获)；在事件处理函数内部，this关键字引用当前元素；事件对象总是可以通过处理函数的第一个参数(e)捕获；可以为同一个元素绑定你所希望的多个事件，同时并不会覆盖先前绑定的事件

　　缺点：IE不支持，你必须使用IE的attachEvent函数替代。

　　IE下的方法：

element.attachEvent('onclick', function(){

// ...

});

　　优点：可以为同一个元素绑定你所希望的多个事件，同时并不会覆盖先前绑定的事件。  
　　缺点：IE仅支持事件捕获的冒泡阶段；事件监听函数内的this关键字指向了window对象，而不是当前元素（IE的一个巨大缺点）；事件对象仅存在与window.event参数中；事件必须以ontype的形式命名，比如，onclick而非click；仅IE可用，你必须在非IE浏览器中使用W3C的addEventListener

　　注意：不是意味这低版本的ie没有事件捕获，它也是先发生事件捕获，再发生事件冒泡，只不过这个过程无法通过程序控制。

**解除事件：**

element.removeEventListener('click', function(e){

// ...

}, false);

IE:

element.detachEvent('onclick', function(){

// ...

});

**停止冒泡：**

function stopBubble(e) {

//如果提供了事件对象，则这是一个非IE浏览器

if ( e && e.stopPropagation )

//因此它支持W3C的stopPropagation()方法

e.stopPropagation();

else

//否则，我们需要使用IE的方式来取消事件冒泡

window.event.cancelBubble = true;

}

**停止默认行为：**

//阻止浏览器的默认行为

function stopDefault( e ) {

//阻止默认浏览器动作(W3C)

if ( e && e.preventDefault ) e.preventDefault();

//IE中阻止函数器默认动作的方式

else window.event.returnValue = false;

return false;

}

## 函数节流与防抖

*/\*\**

*\* 防抖*

*\*/*

function debounce(func, wait, immediate) {

var timeout;

*return* function () {

var context = *this*, args = *arguments*;

var later = function () {

timeout = null;

*if* (!immediate) {

func.apply(context, args);

}

};

var callNow = immediate && !timeout;

clearTimeout(timeout);

timeout = setTimeout(later, wait);

*if* (callNow) {

func.apply(context, args);

}

};

}

*// 使用案例：resize 事件监听*

var myEfficientFn = debounce(function () {

*// 这里完成那些复杂的业务功能*

}, 250);

window.addEventListener('resize', myEfficientFn);

*/\*\**

*\* 节流*

*\*/*

function poll(fn, callback, errback, timeout, interval) {

var endTime = Number(new Date()) + (timeout || 2000);

interval = interval || 100;

(function p() {

*// 满足条件时，触发回调*

*if* (fn()) {

callback();

}

*// 条件不满足，并在指定的时间段内，那么延迟一段时间后再次触发检查*

*else* *if* (Number(new Date()) < endTime) {

setTimeout(p, interval);

}

*// 条件不满足并已经超时，触发错误回调*

*else* {

errback(new Error('timed out for ' + fn + ': ' + *arguments*));

}

})();

}

*// 使用案例：确保元素可见*

poll(

function () {

*return* document.getElementById('lightbox').offsetWidth > 0;

},

function () {

*// Done, success callback*

},

function () {

*// Error, failure callback*

}

);

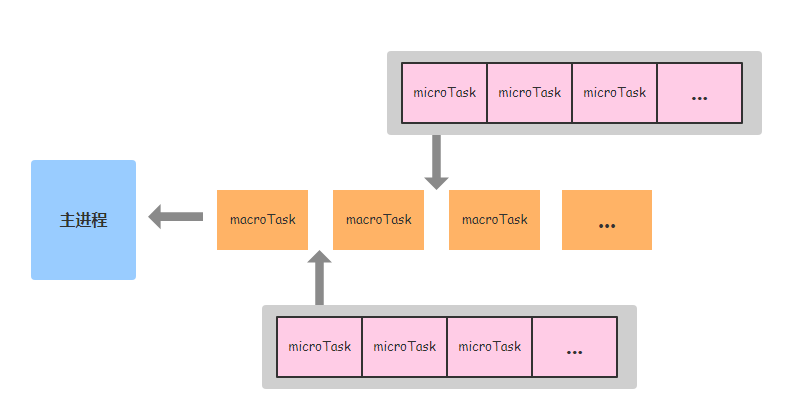
https://juejin.im/entry/58c0379e44d9040068dc952f

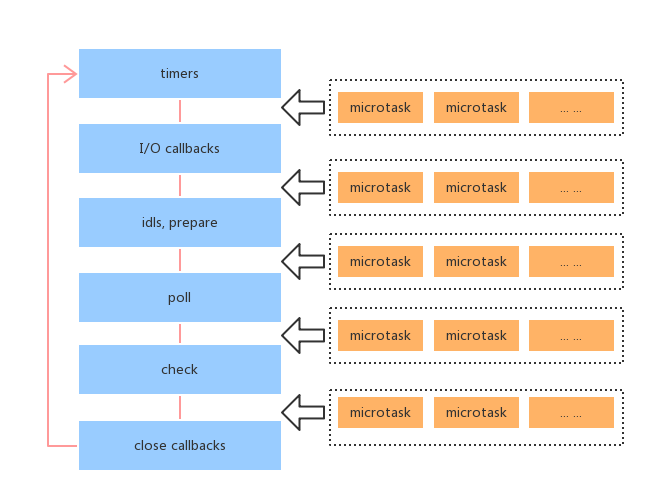
# Node.js

## Node.js 事件循环，定时器和 process.nextTick()

https://nodejs.org/zh-cn/docs/guides/event-loop-timers-and-nexttick/

Express





## Express常用中间件

1. basicAuth

提供基本的访问授权

app.use(connect.basicAuth)();

1. body-parser

npm install --save body-parse;

app.use(require(body-parser))

1. json

json解析模块

1. query

解析查询字符串，并把它编程请求对象上的query属性。（由EXPRESS隐含载入）

1. static

提供对静态（public）文件的支持，这个中间件中间可以连入多次，并可指定不同的目录。

1. express-session
2. cookie-parser

# CSS

## 优先级

！important >> inline Style >> id > class > tag

* Never 绝不要在全站使用!import。
* Only 只在需要覆盖全站或外部 css（例如引用的 ExtJs 或者 YUI ）的特定页面中使用   !important
* Never 永远不要在你的插件中使用 !important
* Always 要优先考虑使用样式规则的优先级来解决问题而不是 !important

权重：

* id：100
* class: 10
* tag: 1

优先级就近原则，同权重情况下样式定义最近者为准;

优先级每个权重不进位，class加到100也不进位，不比id大；

载入样式以最后载入的定位为准;

## 预处理器

## 命名规范

防止组件CSS作用域污染

BEM 是一种真正消除不确定性的命名方式，它使得结构样式更加清晰，我们有足够的信心做任何修改。

block：模块，名字单词间用 - 连接

element：元素，模块的子元素，以 \_\_ 与 block 连接

modifier：修饰，模块的变体，定义特殊模块，以 -- 与 block 连接

## 水平垂直居中

### 行内元素水平居中

行内元素的水平居中，只需要将行内元素包裹在块级父元素内，并且在父元素中设置

#container {

text-align: center;

}

并且适用于文字，链接，及其inline或者inline-block、inline-table和inline-flex

所以之后可以让子元素变为inline-block等类型，再通过这样的方式居中

### 块级元素水平居中

块级元素水平居中只需要将左右外边距设置为auto即可

### 多个块级元素水平居中

1. 让子元素display设为inline-block，再在父元素中text-align: center

.inline-block-center { //父元素  
text-align: center;  
}  
.inline-block-center div { //子元素  
display: inline-block;  
text-align: left;  
}

1. 父元素中设置display: flex，再用justify-content: center居中

.flex-center {//父元素  
display: flex;  
justify-content: center;  
}

1. 以上为水平排列的元素水平居中，如果想要垂直堆栈的元素水平居中使用margin

main div {//子元素  
background: black;  
color: white;  
padding: 15px;  
margin: 5px auto;  
}

## 垂直居中

### 行内或类行内元素

**单行**

1. Padding-top与padding-bottom相等就行，如：padding: 30px 40px;（这里是让多个行内元素的文字垂直居中，一行容不下后会出现重叠）
2. 如果确定文本不会换行，可通过让：line-heigth == height 来实现文字垂直居中

**多行**

1. 多行也能用padding居中，如果不行可以通过设置display，父级为table，子元素为table-cell,然后崽子元素中vertical-align: middle;

.center-table {  
display: table;  
}  
.center-table p {  
display: table-cell;  
vertical-align: middle;  
}

1. 对于父容器为dispaly:flex的元素,每个子元素都是垂直居中的(不只设置display)

.flex-center {  
display: flex;  
flex-direction: column;  
justify-content: center;  
resize: vertical;/\*高度可调；宽度可调(horizontal)；都可以(both)\*/  
overflow: auto;/\*如果被修剪，显示滚动条\*/  
}

1. 如果上述方法都不起作用，那么你就需要使用被称为幽灵元素（ghost element）的非常规解决方式：在垂直居中的元素上添加伪元素，设置伪元素的高等于父级容器的高，然后为文本添加 vertical-align: middle;

.ghost-center {  
position: relative;  
}  
.ghost-center::before {

content: " ";/\*相当于在元素p前立一个根杆子，这样就算p高度不够也会居中\*/  
display: inline-block;  
height: 100%;  
width: 1%;  
vertical-align: middle;  
}  
.ghost-center p {  
display: inline-block;  
vertical-align: middle;  
}

### 块级元素

**已知元素的高度**

main {/\*相当于知道高度后通过top和margin-top上下移动块元素\*/  
height: 300px;  
position: relative;  
}  
main div {  
position: absolute;  
top: 50%;  
height: 100px;  
margin-top: -70px;  
}

**未知元素的高度**

使用transform中的translateY对块级元素进行Y轴方向的比例性移动：

main {  
position: relative;  
}  
main div {  
position: absolute;  
top: 50%;/\*首先上边缘下移到父元素的中间\*/  
transform: translateY(-50%);/\*然后上移子元素自己高度的50%，是的中点重合\*/  
}

注明：translate中的%是基于自己的宽和高的。

**Flexbox**

.parent {  
display: flex;  
flex-direction: column;  
justify-content: center;  
}

## 水平且垂直居中

### 宽高不固定元素

如果无法获取确定的宽高，同样需要设定父级容器为相对定位的容器，设定子元素绝对定位的位置 position: absolute; top: 50%; left: 50%。不同的是，接下来需要使用 transform: translate(-50%, -50%); 实现垂直居中：

.parent {

position: relative;

}

.child {

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%, -50%);

}

如果宽高固定，50%换成对应计算得到的大小即可

### Flexbox

.parent {  
 display: flex;  
 justify-content: center;

align-item: center;  
}

## 盒模型（标准盒模型与IE下的盒模型区别）,BFC

IE盒模型的content包括border、padding

标准盒模型的content就是content

### BFC

* 根元素或包含根元素的元素
* 浮动元素（元素的 float 不是 none）
* 绝对定位元素（元素的 position 为 absolute 或 fixed）
* 行内块元素（元素的 display 为 inline-block）
* 表格单元格（元素的 display为 table-cell，HTML表格单元格默认为该值）
* 表格标题（元素的 display 为 table-caption，HTML表格标题默认为该值）
* 匿名表格单元格元素（元素的 display为 table、table-row、 table-row-group、table-header-group、table-footer-group（分别是HTML table、row、tbody、thead、tfoot的默认属性）或 inline-table）
* overflow 值不为 visible 的块元素
* display 值为 flow-root 的元素
* contain 值为 layout、content或 strict 的元素
* 弹性元素（display为 flex 或 inline-flex元素的直接子元素）
* 网格元素（display为 grid 或 inline-grid 元素的直接子元素）
* 多列容器（元素的 column-count 或 column-width 不为 auto，包括 column-count 为 1）
* column-span 为 all 的元素始终会创建一个新的BFC，即使该元素没有包裹在一个多列容器中（标准变更，Chrome bug）。

块格式化上下文对浮动定位（参见 [float](https://developer.mozilla.org/zh-CN/docs/Web/CSS/float" \o "此页面仍未被本地化, 期待您的翻译!)）与清除浮动（参见 [clear](https://developer.mozilla.org/zh-CN/docs/Web/CSS/clear" \o "clear CSS 属性指定一个元素是否必须移动(清除浮动后)到在它之前的浮动元素下面。clear 属性适用于浮动和非浮动元素。)）都很重要。浮动定位和清除浮动时只会应用于同一个BFC内的元素。浮动不会影响其它BFC中元素的布局，而清除浮动只能清除同一BFC中在它前面的元素的浮动。外边距折叠（[Margin collapsing](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Mastering_margin_collapsing)）也只会发生在属于同一BFC的块级元素之间。

## position

position：absolute，relative

* absolute

生成绝对定位的元素，相对于值不为 static的第一个父元素进行定位。

* fixed （老IE不支持）

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（忽略 top, bottom, left, right z-index 声明）。

* inherit

规定从父元素继承 position 属性的值。

## translate

## 游览器最小显示的字体大小，如何更小

transform: scale(0.5,0.5);

## 移动端兼容

* rem，vw，vh，vmin，vmax等单位

rem 可以通过JavaScript计算当前大小动态设置

也可以通过媒体查询设置

* <meta>标签

width=device-width ：表示宽度是设备屏幕的宽度

initial-scale=1.0：表示初始的缩放比例

minimum-scale=0.5：表示最小的缩放比例

maximum-scale=2.0：表示最大的缩放比例

user-scalable=yes：表示用户是否可以调整缩放比例

如果是想要一打开网页，则自动以原始比例显示，并且不允许用户修改的话，则是：

<meta name="viewport" content="width=device-width,user-scalable=no,initial-scale=1.0, maximum-scale=1.0,minimum-scale=1.0">

## 浮动的理解，清除浮动

1. 父级元素的css中添加： overflow: hidden;
2. 父级元素的</div>前，加一个div标签，css中包含: clear: both;

.clear{ clear:both}

**<div class="clear"></div>**

1. 父级元素设置合适的css高度，这个比较死板。

## CSS reset

HTML标签在浏览器中都有默认的样式，不同的浏览器的默认样式之间存在差别。例如ul默认带有缩进样式，在IE下，它的缩进是由margin实现的，而在Firefox下却是由padding实现的。开发时浏览器的默认样式可能会给我们带来多浏览器兼容性问题，影响开发效率。现在很流行的解决方式是一开始就将浏览器的默认样式全部覆盖掉，这就是css reset。

## 轮播图

## Sprite 图(雪碧图)

## Flex



## grid

## Animation（css3动画相关）

## canvas 和 SVG

# React

## Virtual DOM & Diff

Virtual DOM 相对与原生DOM的好处是能够

## React.setState()

React.setState()不保证同步

如果要保证同步，传入状态计算函数

React.setState((prevState, props) => ({}))

* 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，形式了所谓的“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。

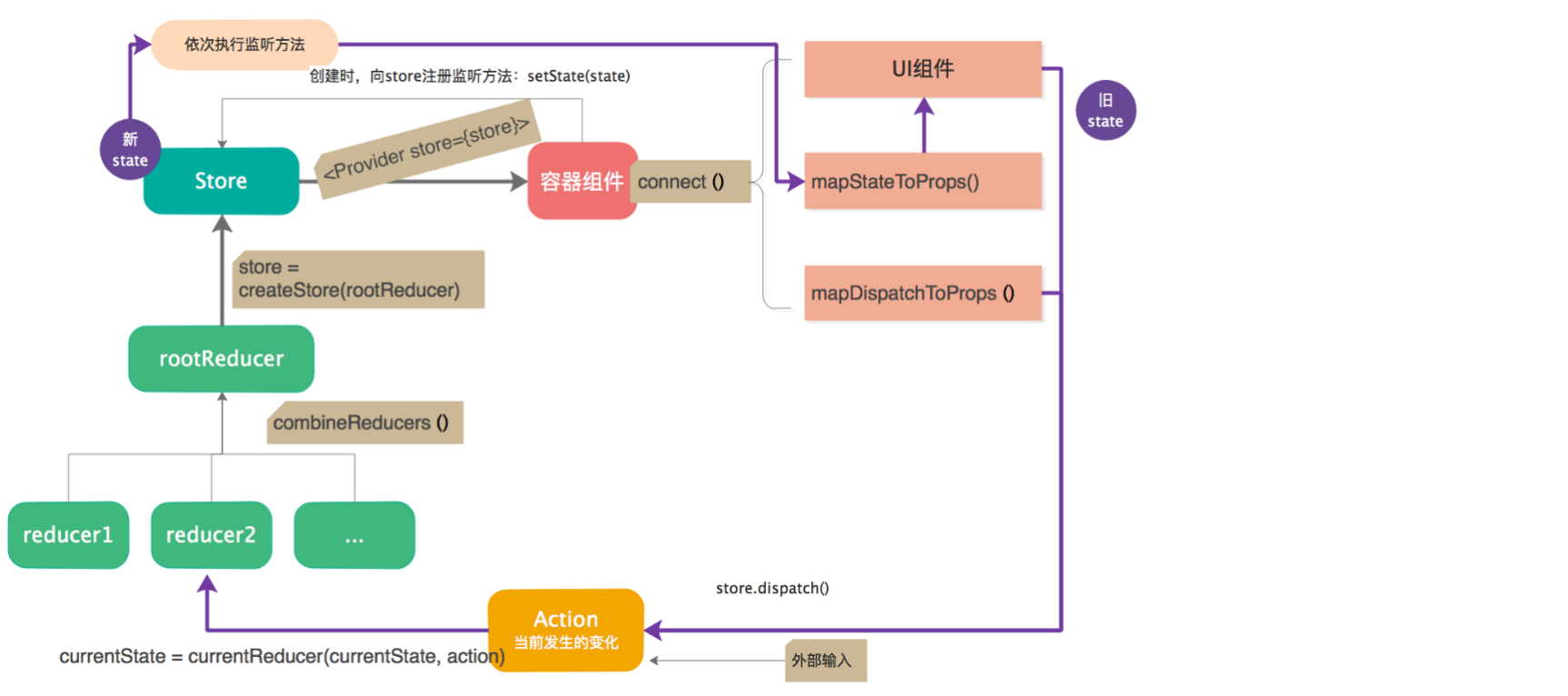
## 无状态组件与PureComponent

## 组件通信，兄弟通信

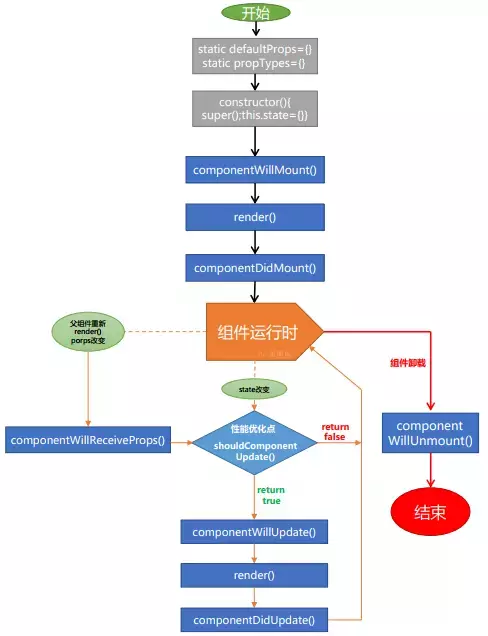
* porps
* 父节点传递回调函数
* 提升改状态到父节点
* 发布-订阅者模式

## Redux

* **react-redux**



## React生命周期



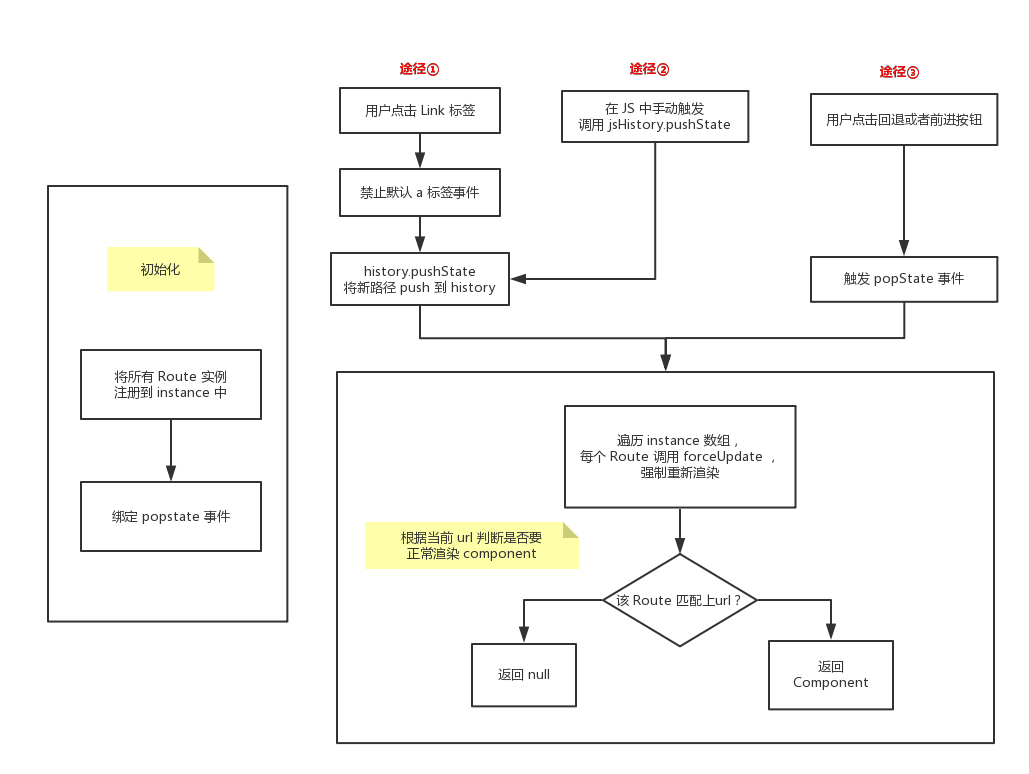
## React如何优化

## react-router

react-router的目的是通过URL切分整个单页应用

要保证URL的变化与组件切换的同步

react-router基于history库，history是一个底层垫片库，为不同的平台提供统一的接口。通过用JavaScript改变URL而不进行页面跳转，并且同时切换需要渲染的组件，保证URL的变化与组件切换进行了同步



## React事件机制

React有自己的一套事件机制，将原生事件合成为自己的事件，每种事件只合成一个，再将所有注册在该事件上的回调函数一次放入数组中保存，事件触发时，依次执行

所有的回调函数都注册在document上，再通过dispatchEvent统一分发

在合成事件的回调函数中调用stopPropagation只能阻止合成事件的分发

# HTML

## 前端路由 history

## Get与Post区别

直观来看

1. Get将参数通过?分割，&分隔写在URL上，Post通过请求体传参

但是HTTP协议并没有规定这样做，只是游览器的实现导致了默认如此实现

2. Get参数长度有限制，Post没有

这也是由于游览器的实现上导致的，IE 2048个字符，firfox 10W以上

3． 有说法Get只发一个数据包，Post会有两个100-continue

这也是实现上导致的，只有当数据包很大的时候，才会发送两个数据包

个人理解：

区分Get与Post应该从语义上来理解，Get表示获取资源，Post表示处理资源

Get是安全的，幂等的，可缓存的

Post是不安全的，不幂等的，通常不可缓存的

所谓安全，应当理解为对于服务器是无害的

所谓幂等，则重复触发多次应当返回相同的效果（重复零次与重复n次的结果一样）

所谓缓存，表示一个方法能否被缓存

## 浏览器缓存机制（强缓存与协商缓存区别，优先级）

1. 浏览器每次发起请求时都会从浏览器缓存中查找请求的结果与缓存标识
2. 浏览器每次拿到返回的请求结果都会将该结果和缓存标识存入浏览器缓存中
3. 浏览器优先使用强缓存，强缓存未命中则使用协商缓存

* 强缓存

强制缓存即向浏览器查找该请求结果，并根据该结果的缓存规则来决定是否使用该缓存

* 1. 不存在缓存结果与缓存标识，直接向服务器请求资源
  2. 存在缓存结果与缓存标识，但是缓存结果已经失效，使用协商缓存
  3. 存在缓存结果与缓存标识，并且缓存结果尚未失效，直接使用浏览器缓存

强制缓存的具体规则

Expires：服务器返回的资源到期时间，HTTP1.1已经弃用

Cache-Control：

* + - * public：所有内容都将被缓存（客户端和代理服务器都可缓存）
      * private：所有内容只有客户端可以缓存，Cache-Control的默认取值
      * no-cache：客户端缓存内容，但是是否使用缓存则需要经过协商缓存来验证决定
      * no-store：所有内容都不会被缓存，即不使用强制缓存，也不使用协商缓存
      * max-age=xxx (xxx is numeric)：缓存内容将在xxx秒后失效
* 协商缓存

协商缓存生效，返回304状态码，协商缓存失效返回200状态码与请求结果

协商缓存有两对字段：last-modified/last-modified-since

Etag/if-none-match

## 状态码 301，302，304

* 2XX——表明请求被正常处理了
  + 200 OK：请求已正常处理
  + 204 No Content：请求处理成功，但没有任何资源可以返回给客户端，一般在只需要从客户端往服务器发送信息，而对客户端不需要发送新信息内容的情况下使用。
  + 206 Partial Content：是对资源某一部分的请求，该状态码表示客户端进行了范围请求，而服务器成功执行了这部分的GET请求。响应报文中包含由Content-Range指定范围的实体内容。
* **3XX——表明浏览器需要执行某些特殊的处理以正确处理请求**
  + 301 Moved Permanently：资源的URL已更新，你也更新下你的书签引用吧。永久性重定向，请求的资源已经被分配了新的URI，以后应使用资源现在所指的URI。
  + 302 Found：资源的URI已临时定位到其他位置了，临时性重定向。和301相似，但302代表的资源不是永久性移动，只是临时性性质的。换句话说，已移动的资源对应的URI将来还有可能发生改变。
  + 303 See Other：资源的URI已更新，你是否能临时按新的URI访问。该状态码表示由于请求对应的资源存在着另一个URL，应使用GET方法定向获取请求的资源。303状态码和302状态码有着相同的功能，但303状态码明确表示客户端应当采用GET方法获取资源，这点与302状态码有区别。

当301，302，303响应状态码返回时，几乎所有的浏览器都会把POST改成GET，并删除请求报文内的主体，之后请求会自动再次发送。

* + 304 Not Modified：资源已找到，但未符合条件请求，协商缓存失效返回304
  + 307 Temporary Redirect：临时重定向。与302有相同的含义。
* **4XX——表明客户端是发生错误的原因所在**
  + 400 Bad Request：服务器端无法理解客户端发送的请求，请求报文中可能存在语法错误。
  + 401 Unauthorized：该状态码表示发送的请求需要有通过HTTP认证（BASIC认证，DIGEST认证）的认证信息。
  + 403 Forbidden：不允许访问那个资源。该状态码表明对请求资源的访问被服务器拒绝了。（权限，未授权IP等）
  + 404 Not Found：服务器上没有请求的资源。路径错误等。
* **5XX——服务器本身发生错误**
  + 500 Internal Server Error：貌似内部资源出故障了。该状态码表明服务器端在执行请求时发生了错误。也有可能是web应用存在bug或某些临时故障。
  + 503 Service Unavailable：抱歉，我现在正在忙着。该状态码表明服务器暂时处于超负载或正在停机维护，现在无法处理请求

## Cookie，sessionStorage，localStorage

三者都是保存在浏览器端、且同源的

Cookie是比较传统的浏览器储存方案，大小存在限制，４K，20条，发起请求时会自动附在HTTP头

LocalStorage 与 sessionStorage一般都是5M，仅在客户端中缓存，不参与服务器的通信

SessionStorage在当前会话有效，游览器窗口打开，刷新重载也存在，但是同一页面不同窗口不存在

Cookie与localStorage同源可见

## HTML5 新标签

|  |  |
| --- | --- |
| <audio> | 定义声音或音乐内容。 |
| <embed> | 定义外部应用程序的容器（比如插件）。 |
| <source> | 定义 <video> 和 <audio> 的来源。 |
| <track> | 定义 <video> 和 <audio> 的轨道。 |
| <video> | 定义视频或影片内容。 |
| <canvas> | 定义使用 JavaScript 的图像绘制。 |
| <svg> | 定义使用 SVG 的图像绘制。 |
| <article> | 定义文档内的文章。 |
| <aside> | 定义页面内容之外的内容。 |
| <bdi> | 定义与其他文本不同的文本方向。 |
| <details> | 定义用户可查看或隐藏的额外细节。 |
| <dialog> | 定义对话框或窗口。 |
| <figcaption> | 定义 <figure> 元素的标题。 |
| <figure> | 定义自包含内容，比如图示、图表、照片、代码清单等等。 |
| <footer> | 定义文档或节的页脚。 |
| <header> | 定义文档或节的页眉。 |
| <main> | 定义文档的主内容。 |
| <mark> | 定义重要或强调的内容。 |
| <menuitem> | 定义用户能够从弹出菜单调用的命令/菜单项目。 |
| <meter> | 定义已知范围（尺度）内的标量测量。 |
| <nav> | 定义文档内的导航链接。 |
| <progress> | 定义任务进度。 |
| <rp> | 定义在不支持 ruby 注释的浏览器中显示什么。 |
| <rt> | 定义关于字符的解释/发音（用于东亚字体）。 |
| <ruby> | 定义 ruby 注释（用于东亚字体）。 |
| <section> | 定义文档中的节。 |
| <summary> | 定义 <details> 元素的可见标题。 |
| <time> | 定义日期/时间。 |
| <wbr> | 定义可能的折行（line-break）。 |

## <mate>标签的作用：

https://www.cnblogs.com/jesse131/p/5334311.html

## <script> 标签中的defer 与 async

普通的script标签，加载执行时会阻塞DOM文档的渲染，script标签可以并发下载，但是解析执行会按照引入顺序进行。当脚本解析完成之后，文档继续渲染。

defer标签，文档解析时遇到了defer标签，会在后台进行下载，但是并不会阻止文档的渲染，当页面渲染结束后，会等到所有的脚本加载完毕，然后按照顺序解析执行

async标签，文档会在加载完成后立即执行，所以可能阻塞文档渲染，并且async不会触发DOMContentLoaded事件

如果标签中的代码依赖页面中的DOM元素，或者被其它脚本依赖，使用defer

如果标签中的代码并不关心页面中的DOM元素，也不会产生其他脚本所需要的数据，使用async

## document.readystate中的状态

* loading

document 仍在加载。

* interactive

文档已经完成加载，文档已被解析，但是诸如图像，样式表和框架之类的子资源仍在加载。

* complete

文档和所有子资源已完成加载。状态表示 load 事件即将被触发。

当这个属性的值变化时，document 对象上的readystatechange 事件将被触发。

## Ajax的请求状态（readyState）

0 (Uninitialized)未初始化。 (XMLHttpRequest)对象已经创建，还没有调用 open() 方法  
1 (Loading)代表正在加载。 open 方法已被调用，但 send() 方法还没有被调用  
2 (Loaded)代表已加载完毕。send 已被调用。请求已经开始  
3 (Interactive)代表交互中。服务器正在发送响应,已经可以接收到部分请求  
4 (Complete)代表完成。响应发送完毕，数据接受完成

## HTTP2.0

### 多路复用与长连接复用

#### 长连接复用

在HTTP/2中，客户端向某个域名的服务器请求页面的过程中，只会创建一条TCP连接，即使这页面可能包含上百个资源。而之前的HTTP/1.x一般会创建6-8条TCP连接来请求这100多个资源。单一的连接应该是HTTP2的主要优势，单一的连接能减少TCP握手带来的时延。并且由于TCP的滑动窗口算法，存在慢启动的过程，复用单一连接能够减少这一过程

#### 多路复用

HTTP1.x对于多个请求都是建立6-8个TCP连接之后，每个连接请求资源都是串行的，而HTTP2能够并行的将数据包发送出去，原理是请求资源切分成不同的包，每个包进行标记，一起发送

### 头部压缩和二进制格式

http1.x的内容都是plain text，除了便于阅读之外，并没有其他的好处。SSL加密之后，这一点好处也不复存在。所以http2使用HPACK压缩头部，将报文中常用的字段与值建立一个静态索引表，用索引值代替原来的字段。并且随着数据传输建立一个动态索引表。对于路径资源之类的不确定的数据，则使用Huffman编码压缩。

头部压缩的原因是，部分重复的字段在每个HTTP请求中都会出现，值得被压缩。此外HTTP请求的大小正在逐步变大，当超过TCP报文大小的时候，就会拆成数个TCP报文进行传输，进行压缩能够有效减小TCP报文个数，减少RTT时间

### 服务端推送Server Push

服务端可以在客户端请求某个资源时，主动推送相应的资源，比如，客户端在请求index.html时，服务端可以主动推送index.css与index.js等文件。在一次请求中，便可以获取到相应的三个文件

服务端推送的目的便是减少HTTP请求次数

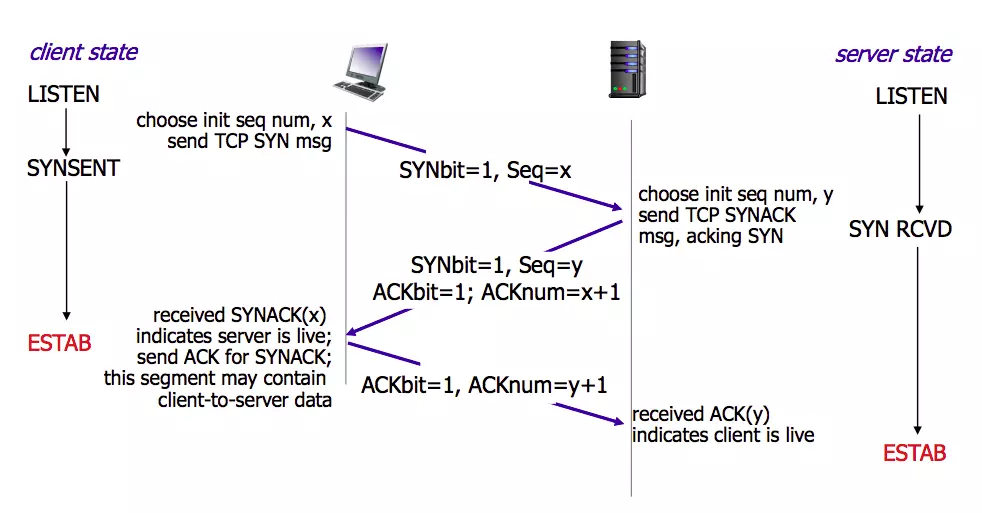
在HTTP2之前，减少HTTP请求次数的优化方式通常是合并外部资源到网页文件中，比如将图片进行Base64编码，或者进行预加载<link rel=“preload“ as=”style“>

# 网络

## TCP三次握手，四次挥手

### 三次握手

* 过程
  1. 客户端发送一个SYN段，并指明客户端的初始序列号，即ISN(c).
  2. 服务端发送自己的SYN段作为应答，同样指明自己的ISN(s)。为了确认客户端的SYN，将ISN(c)+1作为ACK数值。这样，每发送一个SYN，序列号就会加1. 如果有丢失的情况，则会重传。
  3. 为了确认服务器端的SYN，客户端将ISN(s)+1作为返回的ACK数值。



* 目的

三次握手是为了确认服务端与客户端均通信正常

第一次握手，客户端的发送能力正常

第二次握手，服务端的接受能力正常

第三次握手，服务端的发送能力与客户端的接收能力正常

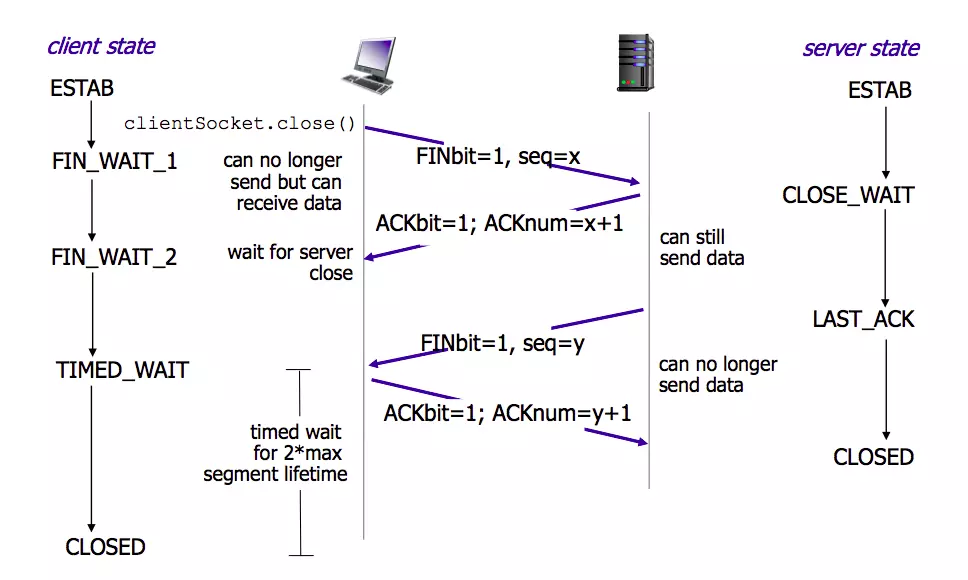
### 四次挥手

* 过程

1. 发起方发送数据包，FIN = 1， seq = x;

当发送 FIN 包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。发送完毕后，（假设发送方式客户端）客户端进入 FIN + WAIT\_1 状态。

1. 服务器确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但是还没准备好关闭连接（理论上：有可能还有数据向客户端传送）。
2. 服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。发送完毕后，服务端进入 LAST\_ACK 状态，等待来自客户端的最后一个 ACK 。
3. 客户端接受到来自服务器的关闭请求，发送一个确认包，并进入 TIME\_WAIT 状态，等待可能出现的要求重传 ACK 包。服务器端接受到这个确认包后，关闭连接，进入 CLOSED 状态。



* 四次挥手时为什么要等待2MSL

A--->B Fin, B--->A ACK ，A属于主动关闭方，收到B的ACK后，A到B的方向连接关闭，即half shutown ，这时A不能再发送数据了。

这种状态下B还是可以单向发送数据的，B的数据发送完毕，也做关闭动作了：

B--->A Fin, A--->B ACK

B收到ACK，关闭连接。

但是A无法知道ACK是否已经到达B，于是开始等待。

假如ACK没有到达B，B会为FIN这个消息超时重传 timeout retransmit

如果A等待时间足够，又收到FIN消息，说明ACK没有到达B，于是再发送ACK，直到在足够的时间内没有收到FIN，说明ACK成功到达。

这个等待时间至少是：B的timeout + FIN的传输时间，为了保证可靠，采用更加保守的等待时间2MSL。

MSL，Maximum Segment Life，这是TCP 对TCP Segment 生存时间的限制。

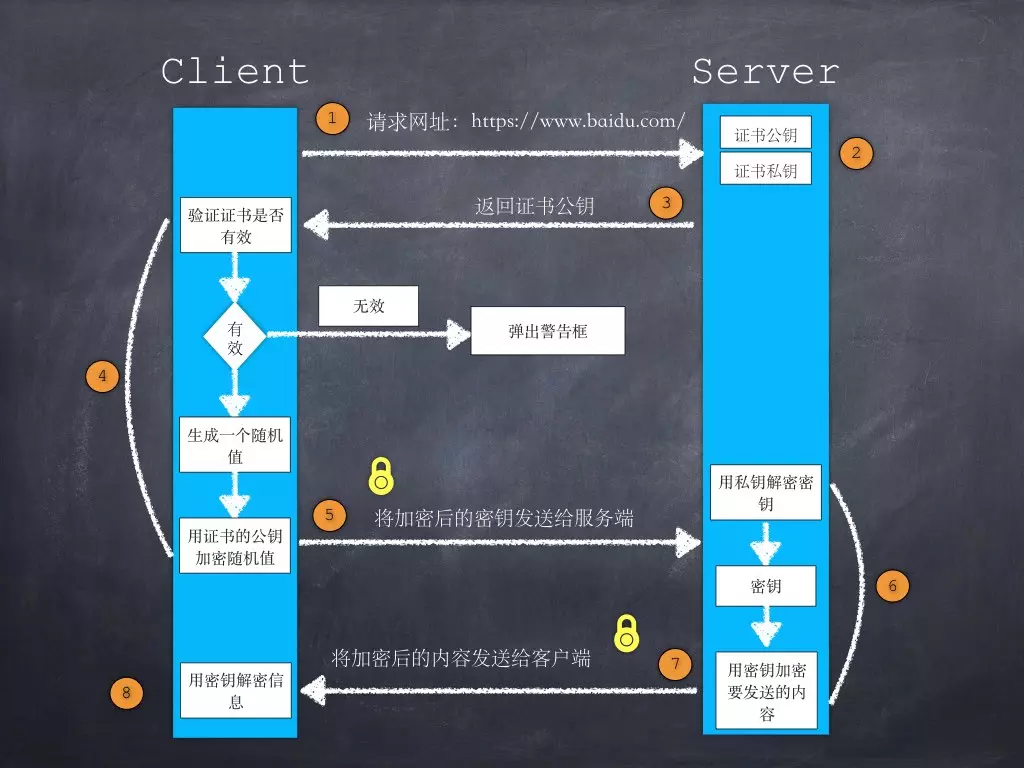
TTL， Time To Live ，IP对IP Datagram 生存时间的限制，255 秒，所以 MSL一般 = TTL = 255秒

A发出ACK，等待ACK到达对方的超时时间 MSL，等待FIN的超时重传，也是MSL，所以如果2MSL时间内没有收到FIN，说明对方安全收到FIN。

综上所述，等待2MSL的目的是为了A最后发送的ACK能最终到达B端。

## HTTPS过程

* 对称与非对称加密



HTTPS的过程在我的认知里便是使用非对称加密传输一个对称加密的密钥

非对称加密一次加密的长度是密钥长度，加密较慢，对称加密的效率较高，但是在密钥分发上存在一定的困难，故在HTTPS中，二者结合使用

获取公钥与证书的过程通过TLS协议，TLS协议基于TCP协议。

证书颁发与验证是一个树状的结构，不断向上回溯，直到确认颁发证书的节点是可信的。

# 其它

## MVVM缺点

1. 数据绑定也使得bug很难被调试。比如你看到页面异常了，有可能是你的View的代码有bug，也可能是你的model的代码有问题。数据绑定使得一个位置的Bug被快速传递到别的位置，要定位原始出问题的地方就变得不那么容易了。
2. 数据双向绑定不利于代码重用。客户端开发最常用的是View，但是数据双向绑定技术，让你在一个View都绑定了一个model，不同的模块model都不同。那就不能简单重用view了
3. 一个大的模块中model也会很大，虽然使用方便了也很容易保证数据的一致性，但是长期持有，不释放内存就造成话费更多的内存。

## Jsonp、CORS跨域

跨域问题来源于JavaScript的同源策略，即只有 协议+主机名+端口号 (如存在)相同，则允许相互访问。

也就是说JavaScript只能访问和操作自己域下的资源，不能访问和操作其他域下的资源。

### 跨域解决办法

1. Jsonp 需要目标服务器配合一个callback函数

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>JSONP 实例</title>

</head>

<body>

<div id="divCustomers"></div>

<script type="text/javascript">

function callbackFunction(result, methodName) {

var html = '<ul>';

*for*(let I = 0; I < result.length; i++) {

html += ‘<li>’ + result[i] + ‘</li>’;

}

html += ‘</ul>’;

document.getElementById(‘divCustomers’).innerHTML = html;

}

</script>

<script type=”text/javascript” src=”http://www.runoob.com/try/ajax/jsonp.php?jsoncallback=callbackFunction”></script>

</body>

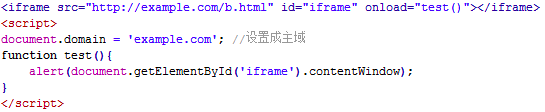
</html>

直接用XMLHttpRequest请求不同域上的数据时，是不可以的。但是，在页面上引入不同域上的js脚本文件却是可以的。通过script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。

jsonp本身非常不安全，如果未处理任何网址都能去获取数据，存在着一个巨大的CSRF漏洞。安全处理：http中添加Referer字段并验证；添加token检验；通过Nginx等方式设置接受地址的范围等。

1. 通过修改document.domain来跨子域

document.domain可以设置为自身或更高一级的父域。这样就可以通过获取iframe的window对象进行跨域的信息传递。



1. 使用window.name + iframe来进行跨域

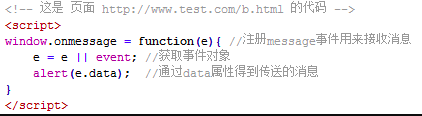
在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的，并不会因新页面的载入而进行重置。可以通过同一窗口的页面跳转实现跨域。

通过iframe可以不页面跳转实现跨域：



1. 跨文档消息传输iframe + window.postMessage





1. Iframe + window.postMessage + localStorage

A、B域都引入C域，通过向C发送，让C存储来实现。

1. localStorage + eventListen(onstorage/addEventListener(storage))
2. 通过CORS解决AJAX跨域，后台设置Access-Control-Allow-Origin

<meta http-equiv="Access-Control-Allow-Origin" content="\*">

<?php header('Access-Control-Allow-Origin: '.$\_SERVER['HTTP\_ORIGIN']); ?>

Nginx: 在nginx.conf里找到server项,并在里面添加

1. 通过Nginx反向代理

## 浏览器线程

游览器是多线程的

常驻线程有：

* GUI 渲染线程

负责渲染浏览器界面HTML元素,当界面需要重绘(Repaint)或由于某种操作引发回流(reflow)时,该线程就会执行。在JavaScript引擎运行脚本期间,GUI渲染线程都是处于挂起状态的,也就是说被“冻结”了.

* JavaScript引擎线程

GUI 渲染线程 与 JavaScript引擎线程互斥！

* 事件触发线程

当一个事件被触发时该线程会把事件添加到待处理队列的队尾，等待JS引擎的处理。这些事件可以是当前执行的代码块，如定时任务、也可来自浏览器内核的其他线程如鼠标点击、AJAX异步请求等，但由于JS的单线程关系所有这些事件都得排队等待JS引擎处理。

* 定时触发器线程

游览器的定时计数器

* 异步http请求线程

XMLHttpRequest在连接后是通过浏览器新开一个线程请求，检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件放到JavaScript引擎的处理队列中等待处理。

## eslint

## docker

## 前端性能优化

* 尽量减少HTTP请求数

这个在HTTP1.x协议下是一个最佳实践，但是HTTP2中应当更加充分的利用缓存

* 减少DNS查找
* 延迟加载组件
* 预加载组件
* 减少DOM元素的数量
* 尽量减少DOM访问
* 把脚本放在底部
* 把JavaScript和CSS放到外面

JavaScript和CSS文件会被缓存在浏览器

* 压缩JavaScript和CSS

## 游览器渲染过程，重绘重排

| **公司** | **浏览器** | **JS引擎** | **渲染引擎** |
| --- | --- | --- | --- |
| Microsoft | IE6-8 | JScritp | Trident |
|  | IE9-11 | Chakra | Trident |
|  | Edge（2015年） | Chakra | Edge |
| Mozilla | Firefox | JagerMonkey | Gecko |
| Google | Chrome | V8 | 以前用的webkit，13年以后用的Blink |
| Apple | Safari | SquirrelFish Extreme | Webkit |
| Opera | Opera12.16+ | Carakan | 13年以后用Blink |

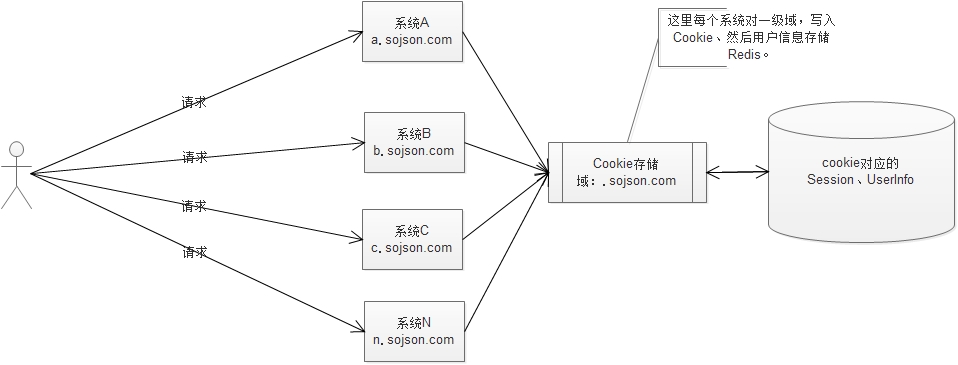
游览器渲染的过程如下:

1. 解析HTML文件生成DOM tree
2. 解析CSS文件生成CSSOM tree
3. 结合DOM 与 CSSOM 生成Render Tree
4. 计算节点在屏幕中的位置，layout
5. 将节点绘制在屏幕上，paint，栅格化数据
6. 浏览器会将各层的信息发送给GPU（GPU进程：最多一个，用于3D绘制等），GPU会将各层合成（composite），显示在屏幕上

## 单点登录

### 同父域，不同子域

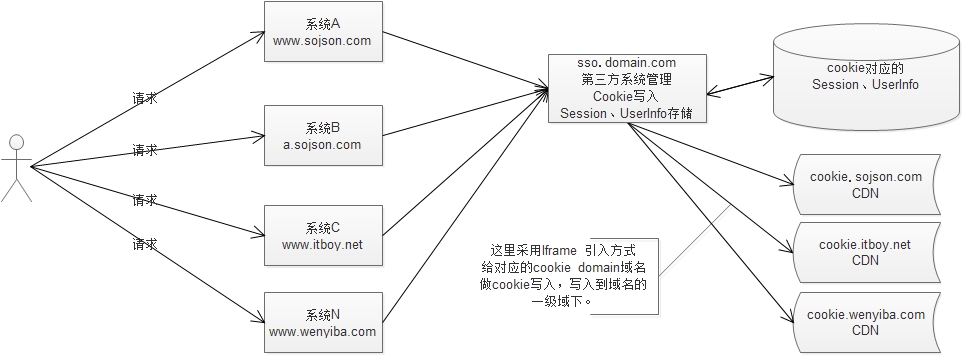
通过设置document.domain = “”，将cookie设置到父域上，则子域也可获取相应的cookie



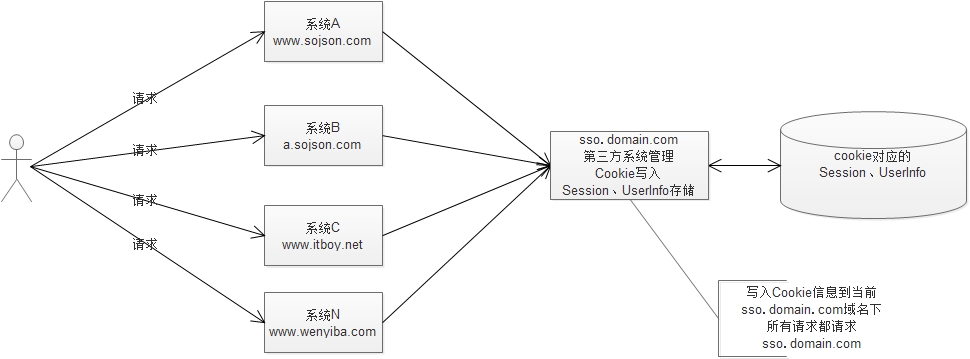
### 不同域父域

单点登录，准备一台专门用作登陆的服务器

* **方案一**

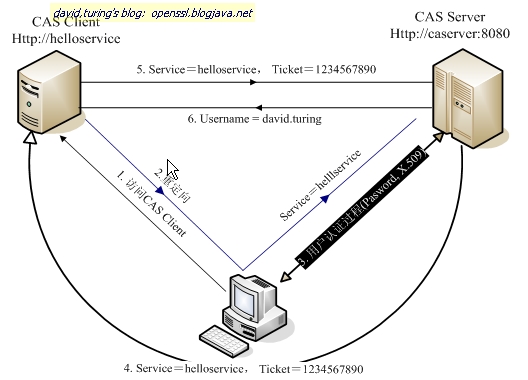


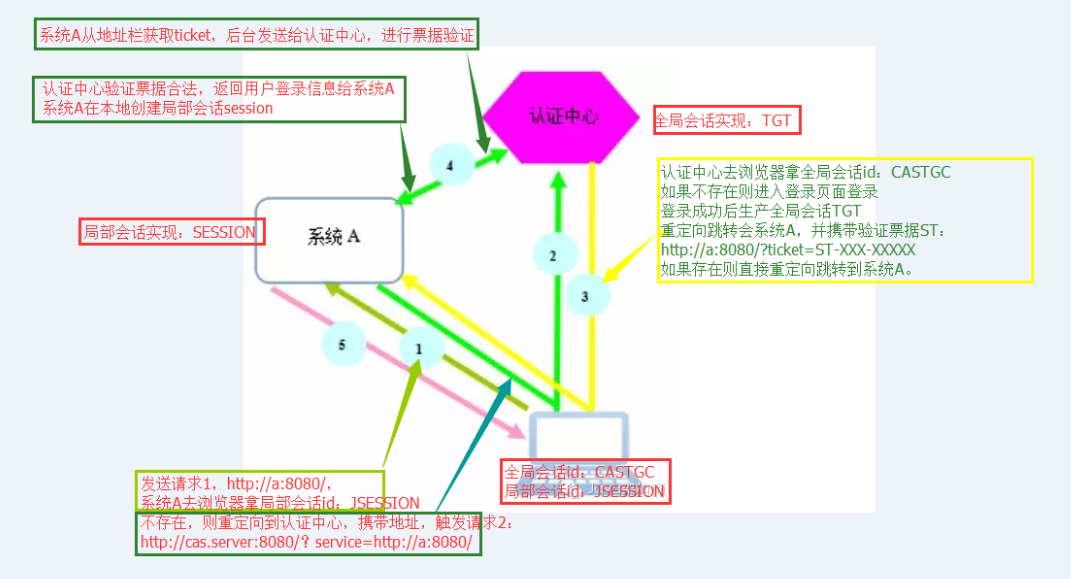
* **方案二**



* **方案三**

**CAS**





## 负载均衡、分布式

* 负载均衡: DNS/Nginx

DNS: 使用RR-DNS（Round-Robin Domain Name System）可以实现平衡负载的功能，向一个主机名发出的入站请求可以被转发到多个IP地址上。

Nginx

1. 轮询
2. 权重轮询
3. IP-hash
4. Fair
5. URL-hash

## CSRF、XSS

* CSRF (Cross Site Request Forgery)

即跨站请求伪造。

假如一家银行用以运行转账操作的URL地址如下： http://www.examplebank.com/withdraw?account=AccoutName&amount=1000&for=PayeeName

那么，一个恶意攻击者可以在另一个网站上放置如下代码：<img src="http://www.examplebank.com/withdraw?account=Alice&amount=1000&for=Badman">

如果有账户名为Alice的用户访问了恶意站点，而她之前刚访问过银行不久，登录信息尚未过期，那么她就会损失1000资金。

1. HTTP头中有一个Referer字段，这个字段用以标明请求来源于哪个地址
2. 在请求地址中添加token并验证（与存在session中的token比较）
3. 在http请求头中自定义属性并验证，相当于把token放在http请求头中

* XSS (Cross Site Scripting)

跨站脚本攻击。为了和层叠样式表(Cascading Style Sheets，CSS)区分开，跨站脚本在安全领域叫做 XSS。攻击者往 Web 页面里注入恶意代码，当用户浏览这些网页时，就会执行其中的恶意代码，可对用户进行盗取 cookie 信息、会话劫持、改变网页内容、恶意跳转等各种攻击。XSS 是常见的 Web 攻击技术之一，由于跨站脚本漏洞易于出现且利用成本低，所以被 OWASP 列为当前的头号 Web 安全威胁。

1. 所有输入都是有害的，对输入进行处理，使用filter处理；
2. http only cookie： 将重要的 cookie 标记为 http only，这样的话当浏览器向服务端发起请求时就会带上 cookie 字段，但是在脚本中却不能访问 cookie
   * JWT放入HTTP Header中的Authorization位，能够解决XSS和XSRF问题