目录


**个人、工作等主观问题**

- 个人简历
- 自我介绍
  - 专业/研究方向简介：
- 如何看待前端
- 为什么选择前端
- 如何学习前端
- 书中印象深刻点
- 公司了解
- 未来规划
- 前沿知识 / 最近学习
- 反问

**基础知识**

- 计算机网络
  - 进程与线程
  - http1，1.1，2.0，3.0，https，tcp，udp
    - http1.0与http1.1的区别
    - http1.1与http2.0的区别
    - http3.0
    - http队首阻塞
    - http与https的区别
    - https安全性实现原理
    - http与tcp 的联系
    - tcp 与 udp 的区别及应用场景
  - http 请求头
  - http返回状态码
  - http缓存机制
  - OSI七层模型
  - 三次握手，四次挥手
  - 三次握手四次挥手中的常见问题
  - 在地址栏里输入URL到页面呈现中间的流程
  - 页面加载性能优化
  - Cookie，sessionStorage，localStorage的区别
  - csrf和xss的网络攻击及防范
  - 设计模式
  - HTML5 新特性
- CSS
  - css盒子模型
  - 行内元素，块元素
  - css选择器
    - 常见选择器
    - 伪类选择器
    - 伪类与伪元素的区别
  - position 定位
  - css动画

    - transition
    - animation
  - 重排，重绘
  - 实现元素的居中
  - Flex 布局
  - 双栏布局
  - margin外边距折叠
  - 清除浮动
  - BFC
  - CSS hack
  - 省略号替代文字超出部分
  - less，sass，stylus区别
  - em，rem，px区别
    - 响应式布局基础 —— rem
    - 移动端自适应方案 —— flexible.js
    - vw + vh + rem 响应式布局
  - 移动前端开发之viewport
  - css3 新特性
- JavaScript
  - JS数据类型
    - 基本数据类型
    - 引用数据类型
    - 两种数据类型的区别
    - Undefined 和 Null 的区别
    - 数据类型的判断
    - 数据类型的转换
    - js中的假值
    - Array数组常用方法
    - String字符串常用方法
  - this指向
  - 箭头函数与普通函数的区别
  - 任务队列，事件循环，微任务，宏任务
    - 为什么setTimeout()不准时
  - 事件模型, 事件
  - 事件委托
  - 异步编程
  - js 延迟加载
  - get，post 区别
  - Ajax
  - Promise，Promise/A+ 规范
  - Generator 函数
  - async，await 函数
  - 深拷贝，浅拷贝
  - 闭包
  - 作用域链
  - 原型，原型链
  - 继承
  - new 操作符做什么
  - 跨域

- 防抖与节流
- 严格模式
- js面向对象的理解
- 模块化开发
- 函数式编程
- es6 新特性

**项目及项目技术**

MES项目描述

项目简介

职责描述

项目难点

物理车间可视化的实现

逻辑车间可视化的展示

echarts的使用

车间基础单位管理 —— 原生js实现拖拽

前端优化

项目技术

- Vue

基础知识

Angularjs，React，Vue

vue.js原理

单页应用与多页应用

MVVM / MVC / MVP 模式

双向绑定

2.0双向绑定的缺陷及3.0的优化

虚拟DOM

Diff 算法

生命周期

父子组件生命周期顺序

组件参数传递

data为什么是函数

computed 与 watchd 的差异

v-if 与 v-show 的区别

v-for key属性的作用

keepalive

nextTick

vue-router

原理

导航守卫

$router 和 $route 的区别

vuex

axio

- Webpack
- Babel
- NPM
- Echarts
- Node.js

Express

- Git

- 正则表达式
  - 常用正则表达式
- 项目收获

**高频手撕代码**

- JS
  - 使用闭包实现定时输出
  - 防抖和节流
  - 浅拷贝、深拷贝
  - 数组扁平化
  - 数组去重
  - 单例模式
  - 手写promise，promise.all 和 promise.race
  - 将原生的ajax封装成promise
  - 限制Promise“并发”的数量
  - 实现sleep函数
  - 模拟实现new
  - 实现 call / apply / bind
  - 函数柯里化
  - 模拟 Object.create() 的实现
  - 数字千分位分隔符
  - 当前周月
- CSS
  - 实现三角形
  - 实现元素的居中
  - 实现三栏布局 / 双栏布局

**算法**

- 数据结构
  - 数组
  - 链表
- 算法解题框架
  - 动态规划解题套路框架
  - 回溯算法解题套路框架
  - BFS（广度优先搜索）算法解题套路框架
- 常见排序算法
  - 冒泡排序
  - 选择排序
  - 插入排序
  - 快速排序
  - 堆排序
- TOP K
  - 快速排序法
  - 堆排序法

**参考资料**


# 个人、工作等主观问题

## 个人简历

# 王鹏

23 / 男 / 前端开发工程师

电话: 136 5984 5944
QQ: 731046400
邮箱: wellpeng97@qq.com
Github: github.com/1pone

## 教育经历

- 硕士 / 华中科技大学 / 工业工程 / 保研 / 研究方向：企业信息化
  *2019.09 – 至今*
- 本科 / 武汉科技大学 / 机械电子工程 *2015.09 – 2019.06*

## 项目经验

### 星光印刷有限公司包装印刷车间星光 MES 团队项目 在线展示

*2019.11 – 至今*

- 相关技术： `Vue` `Element-ui` `Echarts` `Heatmap.js`
- 项目描述：一个包装印刷车间的制造执行系统
- 主要工作：
  - 完成系统车间数据可视化模块的前端开发，前后端分离提高前后端开发效率
  - 定义了通用的车间设备信息数据结构，适用于车间可视化各个功能模块
  - 使用Echarts完成逻辑车间的制造工艺、物料流动等可视化展示
  - 使用Heatmap.js以热力图形式完成车间设备利用率的可视化展示
  - 封装可复用组件及功能函数类，方便项目中重复使用

---

### 基于物联网的智能精准烘烤系统 团队项目 在线展示

*2020.07 – 2020.08*

- 相关技术： `Vue` `Element-ui` `Echarts`
- 项目描述：一个基于物联网的烤烟监控管理系统

- 主要工作：
    - 使用Element-UI进行页面搭建布局
    - 使用Echarts烘烤过程数据的图形可视化展示
    - 封装可复用组件方便项目中重复使用

---

### 电商后台管理系统 个人项目 在线展示 *2019.10 – 2019.12*

- 相关技术： `Vue` `Element-ui` `Webpack` `Git`
- 项目描述：一个拥有用户权限管理，商品管理，订单管理等功能的电商后台管理系统
- 主要工作：
    - 使用Vue-cli对项目进行初始化及Webpack、Babel、Eslint相关初始化配置
    - 使用Vue-Router配置页面路由
    - 使用Element-UI进行页面搭建布局
    - 使用路由守卫在路由配置生效前对用户身份进行token验证
    - 完成系统各功能模块的开发
    - 使用Git对项目进行版本控制

## 技能描述

### Web前端

- 熟练使用 HTML / CSS 进行页面搭建和布局，熟悉 HTML5 / CSS3 新特性
- 熟练使用 JavaScript 语言，熟悉原生 DOM / BOM / Ajax 等
- 熟练使用 Vue / Webpack / Element-UI 等技术链进行前端开发
- 熟练使用 Echarts 进行数据可视化展示
- 有 大屏可视化 / 响应式设计 的相关经验

### Web后端

- 使用过 Node.js ，能够通过 Express 框架搭建后端程序并进行数据处理和数据库交互
- 使用过 Java 进行后端业务设计，能通过 Mybaties 对数据库进行操作
- 有使用 Mysql / MongoDB 等数据库的经验

### 其他技能

- 英语通过 CET6，能够流畅阅读英文文档
- 能够使用 Photoshop / Illustrator 完成简单的图像处理和平面设计

## 获奖描述

- 2019 - 2020 年度华中科技大学硕士研究生奖学金一等奖
- 2019 年武汉科技大学优秀毕业生
- 2018 年湖北省大学生机械创新设计大赛一等奖
- 2017-2018 年度武汉科技大学优秀学生奖学金一等奖

## 自我评价

- 待人真诚，对待工作认真负责
- 能很好落实上级交代的任务，与团队合作协同工作
- 有较好的编程习惯
- 善于探索前沿技术，有较好的学习能力和动手能力
- 热爱生活，有对艺术的追求向往，始终相信好的产品应当能让用户有如同艺术品般的美好享受

「 感谢您在百忙之中在我的简历上停留数秒，相信这数秒的时间对于我们而言都是一次难得的机遇。」

查看在线简历 →

# 自我介绍

**（1'50'' + 为什么选择这家公司） 待完善**

面试官您好，我是王鹏，来自浙江台州，目前是华中科技大学在读研究生，专业是工业工程，研究方向是企业信息化。

熟练使用 HTML / CSS 进行页面搭建和布局，熟悉 HTML5 / CSS3 新特性；熟练使用 JavaScript 语言 ； 熟练使用 Vue / Webpack / Element-UI 等技术链进行前端开发；熟练使用 Echarts 进行数据可视化展示。

目前正在跟着我们实验室团队做一个包装印刷企业MES系统的项目，我的主要职责是参与车间数据可视化模块的前端开发，同时与负责该模块的后台人员进行协同开发，该项目前端采用的是vue框架，选用的ui框架是element-ui，其中在我的模块中还主要使用了echarts和heatmap.js进行可视化的开发。

（为什么选择这家公司）

以上是我的介绍，谢谢。

# 专业/研究方向简介：

**企业信息化：**

**企业信息化建设**是指企业利用计算机技术、网络技术等一系列现代化技术，通过对信息资源的深度开发和广泛利用，不断提高生产、经营、管理、决策的效率和水平，从而提高企业经济效益和企业竞争力的过程。从内容上看，企业信息化主要包括企业产品设计的信息化、企业生产过程的信息化、企业产品销售的信息化、经营管理信息化、决策信息化以及信息化人才队伍的培养等多个方面。

意义：

(1)企业信息化建设利于增强企业的核心竞争力.加快业务流程重组，有利于组织结构优化，有效降低成本，扩大企业竞争范围，激发生产、技术创新.推动研发项目进展，从而提高企业经济效益促进企业竞争，加速企业发展。

(2)企业信息化建设有利于迎接加入WTO后所带来的挑战，适应国际化竞争。加入WTO以后，企业更直接地面对国际竞争的挑战，与国际经济接轨。在全球知识经济和信息化高速发展的今天，信息化是决定企业成败的关键因素，也是企业实现跨地区、跨行业、跨所有制(特别是跨国经营)的重要前提。

(3)企业信息化建设为企业建立了信息门户。这些"门户"让拥有它的企业能够及时的掌握行业动态、市场变化，从而迅速做出反应，抓住占领市场的先机。这些门户已经超出r传统的管理信息系统概念，已成为企业管理信息系统与电子商务两大应用的结合点：这些门户对高科技技术及信息技术的应用。必将会带来信息时代技术革命的飞跃。

(4)企业信息化建设有利于实现国有企业改革和困难企业脱困。

他们可以通过资源共享.现代信息技术利用.科研开发结合为一体，有效地开发、利用信息资源，寻求到合适的合作伙伴和项目，改善管理，早日走出困境。

(5)企业信息化建设实现了企业全部生产经营活动的运营自动化、管理网络化、决策智能化。有利于抓住新世纪的良好发展机遇。使企业在知识经济迅速崛起.在全球信息时化代迅速发展。

**ERP：**

ERP系统是**企业资源计划** (Enterprise Resource Planning) 的简称，是指建立在信息技术基础上，集信息技术与先进管理思想于一身，以系统化的管理思想，为企业员工及决策层提供决策手段的管理平台。它是从MRP（物料需求计划）发展而来的新一代集成化管理信息系统，它扩展了MRP的功能，其核心思想是供应链管理。它跳出了传统企业边界，从供应链范围去优化企业的资源，优化了现代企业的运行模式，反映了市场对企业合理调配资源的要求。它对于改善企业业务流程、提高企业核心竞争力具有显著作用。**通过软件把企业的人、财、物、产、供销及相应的物流、信息流、资金流、管理流、增值流等紧密地集成起来实现资源优化和共享。**

**MES：**

MES系统是制造执行系统的简称，是一套面向制造企业车间执行层的生产信息化管理系统。MES可以为企业提供包括制造数据管理、计划排程管理、生产调度管理、库存管理、质量管理、人力资源管理、工作中心/设备管理、工具工装管理、采购管理、成本管理、项目看板管理、生产过程控制、底层数据集成分析、上层数据集成分解等管理模块，为企业打造一个扎实、可靠、全面、可行的制造协同管理平台。

MESA在MES定义中强调了以下三点：

1、MES是对整个车间制造过程的优化，而不是单一的解决某个生产瓶颈；

2、 MES必须提供实时收集生产过程中数据的功能，并作出相应的分析和处理。

3、MES需要与计划层和控制层进行信息交互，通过企业的连续信息流来实现企业信息全集成。

ERP 与 MES 的区别与联系（除上述定义外）

MES是对ERP计划的监控和反馈

ERP——业务计划系统——解决生产什么的问题；MES——制造执行系统——解决如何生产的问题。

ERP是业务管理级的系统，而MES是现场作业级的系统。MES是底层车间和上层ERP系统之间的协调信息系统。它提供了ERP系统所不能提供的生产车间信息的透明性；提供了连通上层管理系统（如ERP）与底层车间（操作终端与设备）的可靠数据界面等。

MES系统在产品从工单发出到成品产出的过程中，扮演着生产活动的信息传递者。 ERP 使MES系统的生产计划更合理；MES使ERP系统的数据更及时有效，工作效率更高。

---

# 如何看待前端

有一种说法，说是在程序员圈子内也存在着一条鄙视链，做算法的看不上做开发的，做后端开发的看不上做前端开发的，这样看来做前端似乎处于鄙视链的最底端，产生这种偏见的原因，我想大概是因为人们往往会觉得什么岗位越接近计算机科学本身，他就越高端，什么岗位的学习成本和门槛越高就越，他就越牛，乍听之下似乎有几分道理，但是这仅仅是从技术角度片面的看待问题，而不是从用户价值创造的角度，对于用户来说，流畅友好的交互，精致的外观可能是直接打动他们的原因，而不一定是那些看不见摸不着的各种高大上的系统。

我认为前端是很有意义，因为它的工作是直接面向用户的，是将虚拟无形的数据、逻辑实体化并友好地呈现给用户，让用户能够很自然（舒适、流畅）地体验到产品的服务，感受到其中的美好。

同时前端也是很有挑战的，因为前端既要注重用户体验，但与此同时有不能忽视代码结构与性能的重要性。另外，前端的技术发展也是日新月异，这也就要求我们保持一个不断学习的状态。

最后，我认为前端也是很有前景的，目前前端这个行业正在处于一个高速发展的阶段，前端的含义也被进行了拓展，出现了大前端的概念，比如web前端、安卓、iOS、微信小程序、pc桌面端等等，这些客户端的开发都被纳入到了大前端的范畴。前端的未来发展一定是更加全面化、专业化与工程化。

---

# 为什么选择前端

**（添习工业设计、学习钢笔画，体现对艺术、美的追求） 待完善**

选择前端，有这么一个主要的原因：**离成为设计师的目标更进一步**。

说实话，我的内心深处都一直有一个成为设计师的梦想。在这边我可以简单的分享一下我之前的心路历程。

我在大三保研消息还没有定的时候我曾有过考工业设计的念头，我自学了手绘，学习了近代设计史，因为可能那时候觉得工业设计就很有设计师的感觉，在做好一个产品的外形设计的同时，也要注意一个产品的性能设计以及与人的交互设计，就像工业设计大师迪特拉姆斯提出的好设计的十个原则中提到的，好的设计不仅是美的也是实用的。不过最后处于种种原因我还是选择了保研读了工业工程。

然后就是读研后我的研究方向是MES研发，MES指的是制造执行系统，通俗的讲就是一个企业的信息化生产制造管理系统，将传统的手工化、纸质化生产管理模式转变为自动化、电子化，那这个系统最后设计好之后就要靠代码来实现，我就是负责我们这个项目里面可视化模块的设计和前端开发，在不断地学习前端知识的过程中，我意识到前端开发工程师在某种程度上来说也是设计师。

这时候我所理解的“设计师”，已经是广泛意义上的设计师。他们做一些创意的工作，以此来创造一些令人啧啧称赞的作品。这些作品不仅仅可以是一件雕塑，一幅画，还能是一个 Idea，一段代码。

**当你是一个前端工程师的时候，你是一个程序员，还是一个设计师。**

程序员本身也是设计师。虽然程序已经代替了相当数量的手工操作，要想代替程序员则需要更多的时日。然而，程序员手艺的好坏是有相当大的差异的。初学编程的时候，总会看到各种“程序设计”这样高大上的字眼，设计才是程序的核心。这就意味着，写代码的时候，就是在设计作品。设计是一门脑力活，也是一门模式活，从中能看出一个人的风格，从而了解一个人的水平。

因为我认为，前端工程师还应该懂得设计。我便花费了很多时间去：学习绘画，熟悉一些原型设计软件，了解各种配色原理。以指望我可以像一个设计师一样，做好前端网页的设计。毕竟代码和大部分艺术作品一样，容易被临摹、复制，而要复制思想则不是一件容易的事。

当然我的意思不是说前端开发就是纯粹的设计，那是ui界面设计师，我们前端工程师的岗位定位当然还是开发人员，接触的更多还是代码，再好的设计无法实现还是白搭，我的意思是我们在前端工作中不能忽视设计的重要性。

而到了今天，我的设计能力还是有待商榷。幸运的是，我可以使用各种工具搭建功能完善、界面美观、交互友好的页面，熟练地使用各种可视化工具，将繁杂庞大的数据与逻辑清晰地进行展示，而这也正是我选择前端的意义。

---

# 如何学习前端

我学习前端还是有蛮多途径的，比如说在我们实验室会有针对于项目进行不定期的开发培训，我们的培训老师对对我们开发模式和用到的相关技术有比较全面的讲解。然后像我之前在学js、vue的时候我会在网上找一些附带案例的教学视频，跟着视频边学边做加深记忆；或者就是在看一些网上的博客、GitHub、掘金、阮一峰老师这上面一些别人学习总结的知识，这些都是非常精华的内容，看完之后都会很有收获然后就是看一些相关的书籍；当然我也会看一些相关方面的书籍，因为网上别人的总结或是视频都是别人一次加工后的知识，可能会包含重要的知识点用来初步了解学习是没有问题的，但是一些细节的知识点往往是缺失的，所以看书还是会学习的更加全面深入一些，像我最近在看《JavaScript语言精粹》和《你不知道的JavaScript》。还有就是学习某一个特定技术或者框架比如说我在学vue的时候我回去他的官网查看文档说明，像这种有官网文档的话我会优先去看官网文档。以上基本就是我学习前端的一些途径，总的来说基本渠道有很多基本靠自学为主。

---

# 书中印象深刻点

那就举个例子就拿我最近在看的《你不知道的JavaScript》里面有一部分试讲强制类型转换的，这个强转是分显示强转和隐式强转，显示强转往往会比较明显，比如说用构造函数String、Number、Boolean等进行类型转换；但是隐式强转往往会比较隐蔽甚至晦涩。我举个例子，就拿显式...

---

# 公司了解

---

# 未来规划

*写给前端应届生的职业规划建议*

---

# 前沿知识 / 最近学习

我可以分享一下最近国庆假期我学习的一个编程语言。

Processing是关于数字艺术的编程语言，专用于生成以及编辑图像，支持跨平台，语言本身是一个类Java语言，Processing最大特点是把视觉形式、动画、交互与软件中的概念关联在一起，在可视化的开发环境下进行编程，而且它拥有功能丰富的类库，包括物理逻辑库、流体库、粒子运动库、图像视觉库、GUI库等等，可以让在短时间内轻松实现功能强大的程序。因为Processing创建初衷就是面向让艺术家，设计师，教育工作者和编程初学者，很轻松地学习和使用程序设计，并且用它创造出的极富创意的视觉听觉作品。

说了这么多Processing的优点好处，我再说说为什么想要学这个语言，或者说它和我现在所做的或是我将来将要从事的前端有什么关联。那这里就是要提到P5.js。

P5是Processing语言的一个JS移植版本，是一个JavaScript的函数库，使其能在Web中工作。它完全使用JavaScript来实现Processing语言相同的功能（，但并不会动态翻译Processing语言代码，这一点和Processing.js不同。它在制作之初就和Processing有同样的目标。）用P5.js可以讲很多有趣的东西比如艺术创作，做资料数据的可视化，做交互艺术等等，那将这些很具有视觉表现力的艺术效果呈现在浏览器上我认为会有一个非常好的效果，这也是正是我学习Processing的原因。

于此同时还了解到three.js以及CG、新媒体艺术方面的知识。

- Three.js 是一款运行在浏览器中的 3D 引擎，你可以用它创建各种三维场景，包括了摄影机、光影、材质等各种对象。你可以在它的主页上看到许多精采的演示。
- 新媒体艺术不同于现成品艺术、装置艺术、身体艺术、大地艺术等现代艺术。新媒体艺术是一种以光学媒介和电子媒介为基本语言的新艺术学科门类，它建立在数字技术的核心基础上，亦称数码艺术。其表现手段主要为电脑图像CG(computer graph)。新媒体艺术的范畴具有"与时俱进"的确定性，眼下它主要是指那些利用录象、计算机、网络、数字技术等最新科技成果作为创作媒介的艺术品。新媒体艺术已经在不经意中，深入到当代艺术的各个领域中去了。

**Vue3.0**

**1，压缩包体积更小**

当前最小化并被压缩的 Vue 运行时大小约为 20kB（2.6.10 版为 22.8kB）。Vue 3.0捆绑包的大小大约会减少一半，即只有10kB！

**2，Object.defineProperty -> Proxy**

Object.defineProperty是一个相对比较昂贵的操作，因为它直接操作对象的属性，颗粒度比较小。将它替换为es6的Proxy，在目标对象之上架了一层拦截，代理的是对象而不是对象的属性。这样可以将原本对对象属性的操作变为对整个对象的操作，颗粒度变大。

javascript引擎在解析的时候希望对象的结构越稳定越好，如果对象一直在变，可优化性降低，proxy不需要对原始对象做太多操作。

**3，Virtual DOM 重构**

vdom的本质是一个抽象层，用javascript描述界面渲染成什么样子。react用jsx，没办法检测出可以优化的动态代码，所以做时间分片，vue中足够快的话可以不用时间分片。

**传统vdom的性能瓶颈：**

- 虽然 Vue 能够保证触发更新的组件最小化，但在单个组件内部依然需要遍历该组件的整个

vdom 树。

- 传统 vdom 的性能跟模版大小正相关，跟动态节点的数量无关。在一些组件整个模版内只有少量动态节点的情况下，这些遍历都是性能的浪费。
- JSX 和手写的 render function 是完全动态的，过度的灵活性导致运行时可以用于优化的信息不足

**那为什么不直接抛弃vdom呢?**

- 高级场景下手写 render function 获得更强的表达力
- 生成的代码更简洁
- 兼容2.x

vue的特点是底层为Virtual DOM，上层包含有大量静态信息的模版。为了兼容手写 render function，最大化利用模版静态信息，vue3.0采用了动静结合的解决方案，将vdom的操作颗粒度变小，每次触发更新不再以组件为单位进行遍历，主要更改如下

- 将模版基于动态节点指令切割为嵌套的区块
- 每个区块内部的节点结构是固定的
- 每个区块只需要以一个 Array 追踪自身包含的动态节点

**vue3.0将 vdom 更新性能由与模版整体大小相关提升为与动态内容的数量相关**

**4, 更多编译时优化**

- Slot 默认编译为函数：父子之间不存在强耦合，提升性能
- Monomorphic vnode factory：参数一致化，给它children信息，
- Compiler-generated flags for vnode/children types

**5，选用Function_based API**

---

# 反问

- 你觉得我这次面试有什么不足之处呢？刚刚的编程题如何解决?
- 我们公司或者我们部门的前端团队规模有多大?
- 主要的业务方向是什么?
- 我们公司前端开发用的框架是Vue还是React呢？常用到的技术栈有哪一些呢?
- 会涉及到移动端web开发以及移动端app开发或混合开发么？会涉及到小程序开发么?
- 如果有幸入职，请问新员工的培训大概是个什么模式?
- 上下班时间，加班文化
- 企业文化

---

# 基础知识

## 计算机网络

# ●进程与线程

**进程是资源分配的最小单位，线程是CPU调度的最小单位**

做个简单的比喻：进程=火车，线程=车厢

- 线程在进程下行进（单纯的车厢无法运行）
- 一个进程可以包含多个线程（一辆火车可以有多个车厢）
- 不同进程间数据很难共享（一辆火车上的乘客很难换到另外一辆火车，比如站点换乘）
- 同一进程下不同线程间数据很易共享（A车厢换到B车厢很容易）
- 进程要比线程消耗更多的计算机资源（采用多列火车相比多个车厢更耗资源）
- 进程间不会相互影响，一个线程挂掉将导致整个进程挂掉（一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢）
- 进程可以拓展到多机，进程最多适合多核（不同火车可以开在多个轨道上，同一火车的车厢不能在行进的不同的轨道上）
- 进程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。（比如火车上的洗手间）－"互斥锁"
- 进程使用的内存地址可以限定使用量（比如火车上的餐厅，最多只允许多少人进入，如果满了需要在门口等，等有人出来了才能进去）－“信号量”

---

# ● http1，1.1，2.0，3.0，https，tcp，udp

HTTP（超文本传输协议）是互联网上应用最为广泛的一种网络协议，是用于从WWW服务器传输超文本到本地浏览器的传输协议。HTTP客户端发起一个请求，建立一个到服务器指定端口（默认是80端口）的***TCP（拓展三次握手四次挥手、七层架构）***连接。HTTP连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

## http1.0与http1.1的区别

1. 长连接

   HTTP 协议老的标准是HTTP/1.0，为了提高系统的效率，HTTP 1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。（性能缺陷：如包含图片、js文件、css文件链接的网页需要建立更多的连接，耗时且影响客户机和服务器的性能）。HTTP 1.0需要使用keep-alive参数来告知服务器端要建立一个长连接。而HTTP1.1默认支持长连接。

2. 节约宽带

   HTTP 1.1支持只发送header信息(不带任何body信息)，如果服务器认为客户端有权限请求服务器，则返回100，否则返回401。客户端如果接受到100，才开始把请求body发送到服务器。

   这样当服务器返回401的时候，客户端就可以不用发送请求body了，节约了带宽。

   另外HTTP还支持传送内容的一部分。这样当客户端已经有一部分的资源后，只需要跟服务器请求另外的部分资源即可。这是支持文件断点续传的基础。

3. HOST域

   现在可以web server，例如tomat，设置虚拟站点是非常常见的，也即是说，web server上的多个虚拟站点可以共享同一个ip和端口。

   HTTP1.0是没有host域的，HTTP1.1才支持这个参数。

## http1.1与http2.0的区别

1. 多路复用

   HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。

   当然HTTP1.1也可以多建立几个TCP连接，来支持处理更多并发的请求，但是创建TCP连接本身也是有开销的。

   TCP连接有一个预热和保护的过程，先检查数据是否传送成功，一旦成功过，则慢慢加大传输速度。因此对应瞬时并发的连接，服务器的响应就会变慢。所以最好能使用一个建立好的连接，并且这个连接可以支持瞬时并发的请求。

2. 数据压缩

   HTTP1.1不支持header数据的压缩，HTTP2.0使用HPACK头部压缩算法对header的数据进行压缩，这样数据体积小了，在网络上传输就会更快。

3. 服务器推送

   意思是说，当我们对支持HTTP2.0的web server请求数据的时候，服务器会顺便把一些客户端需要的资源一起推送到客户端，免得客户端再次创建连接发送请求到服务器端获取。这种方式非常合适加载静态资源。

   服务器端推送的这些资源其实存在客户端的某处地方，客户端直接从本地加载这些资源就可以了，不用走网络，速度自然是快很多的。

## http3.0

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。 由于多个数据流使用同一个 TCP 连接，遵守同一个流量状态控制和拥塞控制。 只要一个数据流遭遇到拥塞，剩下的数据流就没法发出去，这样就导致了后面的所有数据都会被阻塞。HTTP/2 出现的这个问题是由于其使用 TCP 协议的问题， 与它本身的实现其实并没有多大关系。

由于 TCP 本身存在的一些限制，Google 就开发了一个基于 UDP 协议的 QUIC 协议，并且使用在了 HTTP/3 上。 QUIC 协议在 UDP 协议上实现了多路复用、有序交付、重传等等功能

## http队首阻塞

> 队首阻塞：就是需要排队，队首的事情没有处理完的时候，后面的人都要等着。

1. http1.0的队首阻塞

   对于同一个tcp连接，所有的http1.0请求放入队列中，只有前一个请求的响应收到了，然后才能发送下一个请求。

   可见，http1.0的队首组塞发生在客户端。

2. http1.1的队首阻塞

   对于同一个tcp连接，http1.1允许一次发送多个http1.1请求，也就是说，不必等前一个响应收到，就可以发送下一个请求，这样就解决了http1.0的客户端的队首阻塞。但是，http1.1规定，服务器端的响应的发送要根据请求被接收的顺序排队，也就是说，先接收到的请求的响应也要先发送。这样造成的问题是，如果最先收到的请求的处理时间长的话，响应生成也慢，就会阻塞已经生成了的响应的发送。也会造成队首阻塞。

   可见，http1.1的队首阻塞发生在服务器端。

3. http2是怎样解决队首阻塞的

   http2无论在客户端还是在服务器端都不需要排队，在同一个tcp连接上，有多个stream，由各个stream发送和接收http请求，各个steam相互独立，互不阻塞。

   只要tcp没有人在用那么就可以发送已经生成的requst或者reponse的数据，在两端都不用等，从而彻底解决了http协议层面的队首阻塞问题（但是tcp层仍存在问题）。

## http与https的区别

HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全。HTTPS 是基于 HTTP 协议的，不过它采用SSL协议（现采用TLS协议）对HTTP协议传输的数据进行加密，第三方没有办法窃听。并且它提供了一种校验机制，信息一旦被篡改，通信的双方会立刻发现。~~它还配备了身份证书，防止身份被冒充的情况出现，~~故相比于HTTP协议安全性得到保障。

## https安全性实现原理

TLS 的握手过程主要用到了三个方法来保证传输的安全。 首先是对称加密的方法，对称加密的方法是，双方使用同一个秘钥对数据进行加密和解密。但是对称加密的存在一个问题，就是如何保证秘钥传输的安全性，因为秘钥还是会通过网络传输的，一旦秘钥被其他人获取到，那么整个加密过程就毫无作用了。这就要用到非对称加密的方法。

非对称加密的方法是，我们拥有两个秘钥，一个是公钥，一个是私钥。公钥是公开的，私钥是保密的。用私钥加密的数据，只有对应的公钥才能解密，用公钥加密的数据，只有对应的私钥才能解密。我们可以将公钥公布出去，任何想和我们通信的客户，都可以使用我们提供的公钥对数据进行加密，这样我们就可以使用私钥进行解密，这样就能保证数据的安全了。但是非对称加密有一个缺点就是加密的过程很慢，因此如果每次通信都使用非对称加密的方式的话，反而会造成等待时间过长的问题。

因此我们可以使用对称加密和非对称加密结合的方式，因为对称加密的方式的缺点是无法保证秘钥的安全传输，因此我们可以非对称加密的方式来对对称加密的秘钥进行传输，然后以后的通信使用对称加密的方式来加密，这样就解决了两个方法各自存在的问题。

**中间人攻击**

**但是现在的方法也不一定是安全的**，因为我们没有办法确定我们得到的公钥就一定是安全的公钥。可能存在一个中间人，截取了对方发给我们的公钥，然后将他自己的公钥发送给我们，当我们使用他的公钥加密后发送的信息，就可以被他用自己的私钥解密。然后他伪装成我们以同样的方法向对方发送信息，这样我们的信息就被窃取了，然而我们自己还不知道。

为了解决这样的问题，我们可以使用数字证书的方式，首先我们使用一种 Hash 算法来对我们的公钥和其他信息进行加密生成一个信息摘要，然后让有公信力的认证中心（简称 CA）用它的私钥对消息摘要加密，形成签名。最后将原始的信息和签名合在一起，称为数字证书。当接收方收到数字证书的时候，先根据原始信息使用同样的 Hash 算法生成一个摘要，然后使用公证处的公钥来对数字证书中的摘要进行解密，最后将解密的摘要和我们生成的摘要进行对比，就能发现我们得到的信息是否被更改了。这个方法最要的是认证中心的可靠性， 一般浏览器里会内置一些顶层的认证中心的证书，相当于我们自动信任了他们，只有这样我们才能保证数据的安全。

## http与tcp 的联系

TCP是底层通讯协议，定义的是数据传输和连接方式的规范 HTTP是应用层协议，定义的是传输数据的内容的规范 （七层架构模型） HTTP协议中的数据是利用TCP协议传输的，所以支持HTTP也就一定支持TCP

HTTP支持的是www服务 TCP/IP是网络中使用的基本的通信协议，它是Internet国际互联网络的基础，它包括上百个各种功能，如：远程登录、文件传输和电子邮件等，而TCP协议和IP协议是保证数据完整传输的两个基本的重要协议。通常说TCP/IP是Internet协议族，而不单单是TCP和IP

## tcp 与 udp 的区别及应用场景

- TCP和UDP 两者都是通信协议，TCP和UDP都是传输层协议，但是他们的通信机制和应用场景不同。
    - TCP TCP（Transmission Control Protocol）又叫传输控制协议，TCP是面向连接的，并且是一种可靠的协议，在基于TCP进行通信时，通信双方需要建立TCP连接，建立连接需要经过三次握手，握手成功才可以通信。
    - UDP UDP是一种面向无连接，切不可靠的协议，在通信过程中，它并不像TCP那样需要先建立一个连接，只要目的地址，端口号，源地址，端口号确定了，就可以直接发送信息报文，并且不需要一定能收到或者完整的数据。它仅仅提供了校验和机制来保障报文是否完整，若校验失败，则直接将报文丢弃，不做任何处理。
- TCP，UDP的优缺点
    - TCP优点 可靠，稳定 TCP的可靠性体现在传输数据之前，三次握手建立连接（四次挥手断开连接），并且在数据传递时，有确认，窗口，重传，拥塞控制机制，数据传完之后断开连接来节省系统资源。
    - TCP缺点 慢，效率比较低，占用系统资源，容易被攻击 传输数据之前建立连接，这样会消耗时间，而且在消息传递时，确认机制，重传机制和拥塞机制都会消耗大量的时间，而且要在每台设备上维护所有的传输连接。而且每一个连接都会占用系统的CPU，内存等硬件软件资源。并且TCP的取而机制，三次握手机制导致TCP容易被人利用，实现DOS，DDOS攻击。
    - UDP优点 快，比TCP安全 UDP没有TCP的握手，确认窗口，重传，拥塞机制。UDP是一个无状态的传输机制，所以在传输数据时非常快。UDP没有TCP这些机制，相应被利用的漏洞就少一点。但是UDP的攻击也是存在的，比如：UDP 的flood攻击。
    - UDP缺点 不可靠，不稳定 因为UDP没有TCP的那些可靠机制，在网络质量不好的时候容易发生丢包。
- 应用场景
    - TCP应用场景 当对网络通信质量有要求时，比如：整个数据要准确无误的传递给对方，这往往对于一些要求可靠的应用，比如HTTP,HTTPS,FTP等传输文件的协议，POP,SMTP等邮件的传输协议。常见使用TCP协议的应用： 1.浏览器使用的：HTTP 2.FlashFXP:FTP 3.Outlook:POP，SMTP 4.QQ文件传输
    - UDP 文件传输协议 对当前网络通讯质量要求不高的时候，要求网络通讯速度尽量的快，这时就使用UDP 日常生活中常见使用UDP协议： 1.QQ语音 2.QQ视频 3.TFTP

## • http 请求头

通常HTTP消息包括客户机向服务器的请求消息和服务器向客户机的响应消息。客户端向服务器发送一个请求，请求头包含请求的方法、URI、协议版本、以及包含请求修饰符、客户信息和内容的类似于MIME的消息结构。服务器以一个状态行作为响应，相应的内容包括消息协议的版本，成功或者错误编码加上包含服务器信息、实体元信息以及可能的实体内容。

- Host :主机和端口号
- Connection ： 连接类型
- Upgrade-Insecure-Requests：升级为https请求
- User-Agent:浏览器名称
- Accept：传输文件类型
- Referer：页面跳转处
- Accept-Encoding：文件编解码格式
- Cookie：Cookie
- x-requested-with :XMLHttpRequest(是Ajax异步请求)

---

## • http返回状态码

> HTTP状态码分类
>
> 1**： 临时响应并需要请求者继续执行操作
>
> 2**：请求成功。操作被成功接收接收并处理
>
> 3**：重定向代码，需要进一步的操作以完成请求
>
> 4**：请求错误，请求包含语法错误或者无法完成请求
>
> 5**：服务器错误，服务器在处理请求的过程中发生错误

| 状态码 | 状态码英文名称 | 中文描述 |
| --- | --- | --- |
| 100 | Continue | 继续。客户端应继续其请求 |
| 101 | Switching Protocols | 切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到HTTP的新版本协议 |
| 200 | OK | 请求成功。一般用于GET与POST请求 |
| 201 | Created | 已创建。成功请求并创建了新的资源 |
| 202 | Accepted | 已接受。已经接受请求，但未处理完成 |
| 203 | Non-Authoritative Information | 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本 |
| 204 | No Content | 无内容。服务器成功处理，但未返回内容。在未更新网页的情况下，可 |

| | | |
|---|---|---|
| | | 确保浏览器继续显示当前文档 |
| 205 | Reset Content | 重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域 |
| 206 | Partial Content | 部分内容。服务器成功处理了部分GET请求 |
| 300 | Multiple Choices | 多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择 |
| 301 | Moved Permanently | 永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替 |
| 302 | Found | 临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI |
| 303 | See Other | 查看其它地址。与301类似。使用GET和POST请求查看 |
| 304 | Not Modified | 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源 |
| 305 | Use Proxy | 使用代理。所请求的资源必须通过代理访问 |
| 306 | Unused | 已经被废弃的HTTP状态码 |
| 307 | Temporary Redirect | 临时重定向。与302类似。使用GET请求重定向 |
| 400 | Bad Request | 客户端请求的语法错误，服务器无法理解 |
| 401 | Unauthorized | 请求要求用户的身份认证 |
| 402 | Payment Required | 保留，将来使用 |
| 403 | Forbidden | 服务器理解请求客户端的请求，但是拒绝执行此请求 |
| 404 | Not Found | 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面 |
| 405 | Method Not Allowed | 客户端请求中的方法被禁止 |
| 406 | Not Acceptable | 服务器无法根据客户端请求的内容特性完成请求 |
| 500 | Internal Server Error | 服务器内部错误，无法完成请求 |
| | Not | |

| | | |
|---|---|---|
| 501 | Implemented | 服务器不支持请求的功能，无法完成请求 |
| 502 | Bad Gateway | 作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应 |
| 503 | Service Unavailable | 由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中 |
| 504 | Gateway Time-out | 充当网关或代理的服务器，未及时从远端服务器获取请求 |
| 505 | HTTP Version not supported | 服务器不支持请求的HTTP协议的版本，无法完成处理 |

## • http缓存机制

> HTTP缓存机制是根据HTTP报文的缓存标识进行的，所以在分析浏览器缓存机制之前，我们先简单介绍一下HTTP报文，HTTP报文分为两种:
>
> 1. HTTP请求(Request)报文，报文格式为：请求行 – HTTP头(通用信息头，请求头，实体头) – 请求报文主体(只有POST才有报文主体)
> 2. HTTP响应(Response)报文，报文格式为：状态行 – HTTP头(通用信息头，响应头，实体头) – 响应报文主体

**浏览器缓存分类**

浏览器缓存分为**强缓存**和**协商缓存**，浏览器加载一个页面的简单流程如下：

1. 浏览器先根据这个资源的http头信息来判断是否命中强缓存。如果命中则直接加载缓存中的资源，并不会将请求发送到服务器。
2. 如果未命中强缓存，则浏览器会将资源加载请求发送到服务器。服务器来判断浏览器本地缓存是否失效。若可以使用，则服务器并不会返回资源信息，浏览器继续从缓存加载资源。
3. 如果未命中协商缓存，则服务器会将完整的资源返回给浏览器，浏览器加载新资源，并更新缓存。

**强缓存**

强缓存是利用http的返回头中的Expires或者Cache-Control两个字段来控制的，用来表示资源的缓存时间。

**Expires**

缓存过期时间，用来指定资源过期的时间，在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。是**服务器端**的具体的时间点，是**绝对时间**，存在一些缺陷，比如时差等问题会引起缓存混乱。

**Cache-Control**

Cache-Control是一个**相对时间**，并且都是与**客户端**时间比较，所以服务器与客户端时间偏差也不会导致问题。与Expires同时启用的时候Cache-Control**优先级**高。

请求时的缓存指令包括no-cache、no-store、max-age、 max-stale、min-fresh、only-if-cached，响应消息中的指令包括public、private、no-cache、no- store、no-transform、must-revalidate、proxy-revalidate、max-age。

**协商缓存**

若未命中强缓存，则浏览器会将请求发送至服务器。服务器根据http头信息中的Last-Modify/If-Modify-Since或Etag/If-None-Match来判断是否命中协商缓存。如果命中，则http返回304状态码表示未修改，则不会返回资源，浏览器从缓存中加载资源。

**Last-Modify/If-Modify-Since**

浏览器第一次请求一个资源的时候，服务器返回的header中会加上Last-Modify，Last-modify是一个时间标识该资源的最后修改时间。

当浏览器再次请求该资源时，发送的请求头中会包含If-Modify-Since，该值为缓存之前返回的Last-Modify。服务器收到If-Modify-Since后，根据资源的最后修改时间判断是否命中缓存。

如果命中缓存，则返回http304，并且不会返回资源内容，并且不会返回Last-Modify。由于对比的服务端时间，所以客户端与服务端时间差距不会导致问题。但是有时候通过最后修改时间来判断资源是否修改还是不太准确（资源变化了最后修改时间也可以一致）。于是出现了ETag/If-None-Match。

**ETag/If-None-Match**

与Last-Modify/If-Modify-Since不同的是，Etag/If-None-Match返回的是一个校验码（ETag: entity tag）。ETag可以保证每一个资源是唯一的，资源变化都会导致ETag变化。ETag值的变更则说明资源状态已经被修改。服务器根据浏览器上发送的If-None-Match值来判断是否命中缓存。

**既生Last-Modified何生Etag?**

1. Last-Modified标注的最后修改只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间

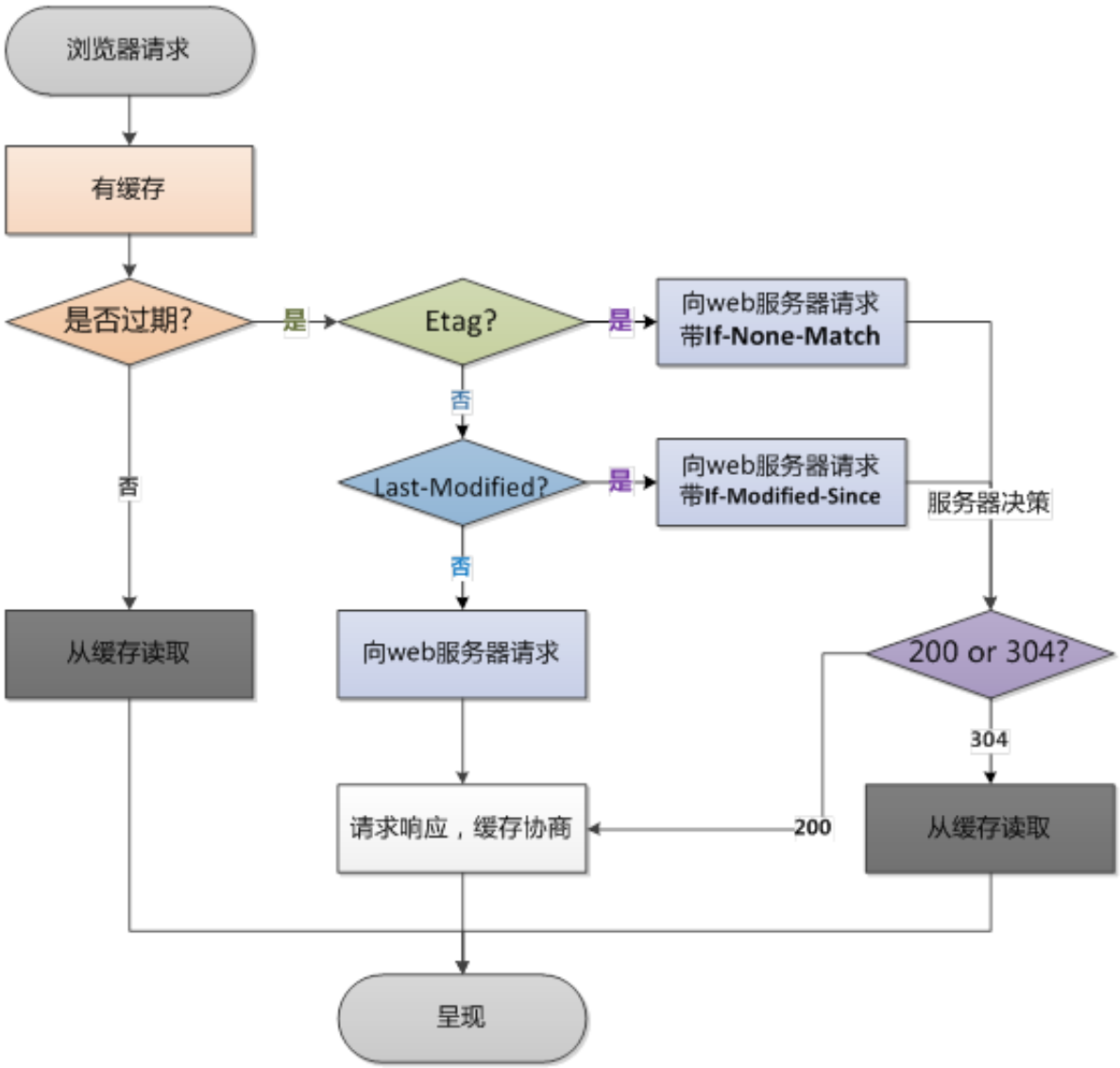
2. 如果某些文件会被定期生成，当有时内容并没有任何变化，但Last-Modified却改变了，导致文件没法使用缓存

3.有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形

Etag是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符，能够更加准确的控制缓存。Last-Modified与ETag是可以一起使用的，服务器会优先验证ETag，一致的情况下，才会继续比对Last-Modified，最后才决定是否返回304。

**用户行为与缓存**

| 用户操作 | Expires/Cache-Control | Last-Modified/Etag |
| --- | --- | --- |
| 地址栏回车 | 有效 | 有效 |
| 页面链接跳转 | 有效 | 有效 |
| 新开窗口 | 有效 | 有效 |
| 前进、后退 | 有效 | 有效 |
| F5刷新 | 无效 | 有效 |
| Ctrl+F5刷新 | 无效 | 无效 |



## • OSI七层模型

从上到下分别是：

应用层：文件传输，常用协议HTTP, FTP ,

表示层：数据格式化，数据加密，代码转换

会话层：建立，解除会话

传输层：提供端对端的接口，tcp,udp

网络层：为数据包选择路由，IP，icmp

数据链路层：传输有地址的帧

物理层：二进制的数据形式在物理媒体上传输数据

| OSI七层网络模型 | TCP/IP四层概念模型 | 对应网络协议 |
| --- | --- | --- |
| 应用层（Application） | 应用层 | HTTP、TFTP, FTP, NFS, WAIS、SMTP |
| 表示层（Presentation） | 应用层 | Telnet, Rlogin, SNMP, Gopher |
| 会话层（Session） | 应用层 | SMTP, DNS |
| 传输层（Transport） | 传输层 | TCP, UDP |
| 网络层（Network） | 网络层 | IP, ICMP, ARP, RARP, AKP, UUCP |
| 数据链路层（Data Link） | 数据链路层 | FDDI, Ethernet, Arpanet, PDN, SLIP, PPP |
| 物理层（Physical） | 数据链路层 | IEEE 802.1A, IEEE 802.2到IEEE 802.11 |

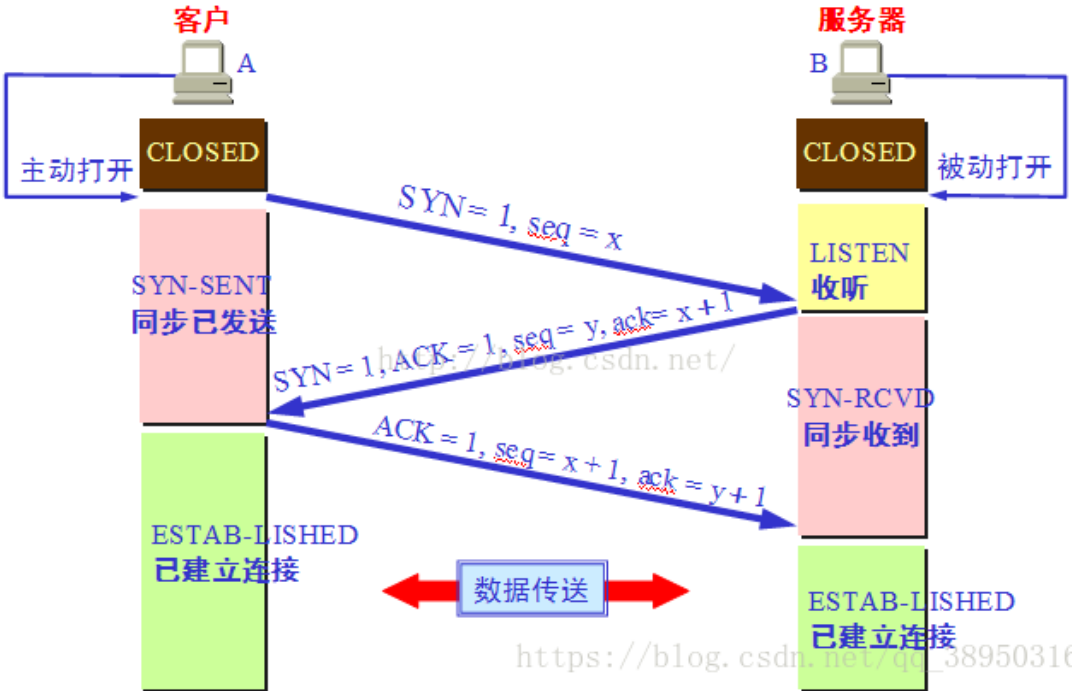
---

## • 三次握手，四次挥手

第一次握手：建立连接时，客户端发送syn包（syn=x）到服务器，并进入**SYN_SENT**状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入**SYN_RECV**状态；

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




（1）首先客户端想要释放连接，向服务器端发送一段TCP报文，其中：

标记位为FIN，表示“请求释放连接”；序号为Seq=U；随后客户端进入**FIN-WAIT-1**阶段，即半关闭阶段。并且停止在客户端到服务器端方向上发送数据，但是客户端仍然能接收从服务器端传输过来的数据。（注意：这里不发送的是正常连接时传输的数据(非确认报文)，而不是一切数据，所以客户端仍然能发送ACK确认报文。）

（2）服务器端接收到从客户端发出的TCP报文之后，确认了客户端想要释放连接，随后服务器端结束ESTABLISHED阶段，进入**CLOSE-WAIT**阶段（半关闭状态）并返回一段TCP报文，其中：

标记位为**ACK**，表示“接收到客户端发送的释放连接的请求”；序号为Seq=V；（确认号为Ack=U+1，表示是在收到客户端报文的基础上，将其序号Seq值加1作为本段报文确认号Ack的值；）随后服务器端开始准备释放服务器端到客户端方向上的连接。客户端收到从服务器端发出的TCP报文之后，确认了服务器收到了客户端发出的释放连接请求，随后客户端结束FIN-WAIT-1阶段，进入**FIN-WAIT-2**阶段

> 前"两次挥手"既让服务器端知道了客户端想要释放连接，也让客户端知道了服务器端了解了自己想要释放连接的请求。于是，可以确认关闭客户端到服务器端方向上的连接了

（3）服务器端自从发出ACK确认报文之后，经过**CLOSED-WAIT**阶段，做好了释放服务器端到客户端方向上的连接准备，再次向客户端发出一段TCP报文，其中：

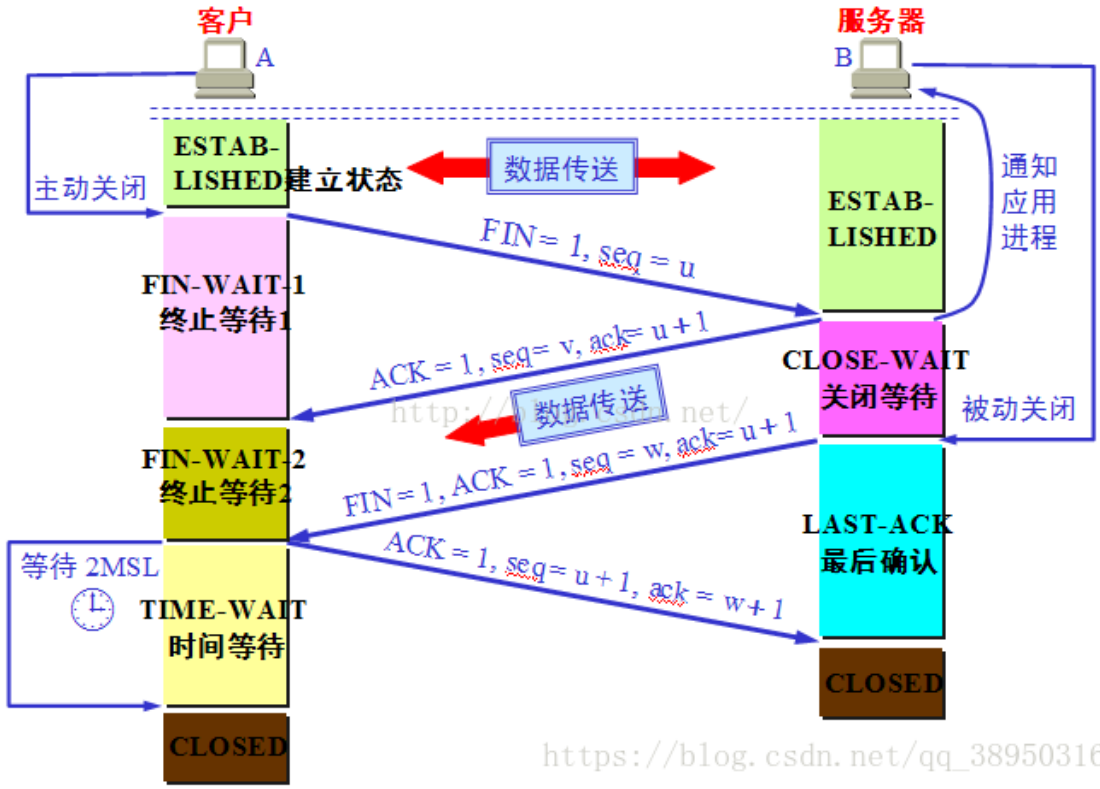
标记位为**FIN，ACK**，表示“已经准备好释放连接了”。（注意：这里的ACK并不是确认收到服务器端报文的确认报文。序号为Seq=W；确认号为Ack=U+1；表示是在收到客户端报文的基础上，将其序号Seq值加1作为本段报文确认号Ack的值。随后服务器端结束CLOSE-WAIT阶段，进入**LAST-ACK阶段**。并且停止在服务器端到客户端的方向上发送数据，但是服务器端仍然能够接收从客户端传输过来的数据。）

（4）客户端收到从服务器端发出的TCP报文，确认了服务器端已做好释放连接的准备，结束FIN-WAIT-2阶段，进入**TIME-WAIT**阶段，并向服务器端发送一段报文，其中：

标记位为ACK，表示“接收到服务器准备好释放连接的信号”。序号为Seq=U+1；（表示是在收到了服务器端报文的基础上，将其确认号Ack值作为本段报文序号的值。确认号为Ack=W+1；表示是在收到了服务器端报文的基础上，将其序号Seq值作为本段报文确认号的值。）随后客户端开始在**TIME-WAIT阶段等待2MSL**

服务器端收到从客户端发出的TCP报文之后结束LAST-ACK阶段，进入**CLOSED**阶段。由此正式确认关闭服务器端到客户端方向上的连接。

客户端**等待完2MS**L之后，结束TIME-WAIT阶段，进入**CLOSED**阶段，由此完成“四次挥手”。




## • 三次握手四次挥手中的常见问题

1. 为什么连接的时候是三次握手，关闭的时候却是四次握手?

答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，**很可能并不会立即关闭SOCKET**，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

2. 为什么TIME_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态?

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假想网络是不可靠的，有可以最后一个ACK丢失。所以**TIME_WAIT状态就是用来重发可能丢失的ACK报文**。在Client发送出最后的ACK回复，但该ACK可能丢失。**Server如果没有收到ACK，将不断重复发送FIN片段**。所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指**一个片段在网络中最大的存活时间**，2MSL就是一个发送和一个回复所需的最大时间。**如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接**。

3. 为什么不能用两次握手进行连接?

答：3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S 是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，C认为连接还未建立成功，将忽略S发来的任何数据分组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

4. 如果已经建立了连接，但是客户端突然出现故障了怎么办?

答：TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

---

## ● 在地址栏里输入URL到页面呈现中间的流程

> - DNS解析
> - TCP连接
> - 发送HTTP请求
> - 服务器处理请求并返回HTTP报文
> - 浏览器解析渲染页面
> - 连接结束

输入url后，首先需要找到这个url域名的**服务器ip**，为了寻找这个ip，浏览器首先会寻找缓存，查看缓存中是否有记录，缓存的查找记录为：浏览器缓存 ->系统缓存 -> 路由器缓存，缓存中没有则查找系统的hosts文件中是否有记录，如果没有则查询DNS服务器，得到服务器的ip地址后，浏览器根据这个ip以及相应的端口号，构造一个**http请求**，这个请求报文会包括这次请求的信息，主要是请求方法，请求说明和请求附带的数据，并将这个http请求封装在一个tcp包中，这个tcp包会依次经过传输层，网络层，数据链路层，物理层到达服务器，服务器解析这个请求来作出响应，返回相应的**html**给浏览器，因为html是一个树形结构，浏览器根据这个html来构建DOM树，在dom树的构建过程中如果遇到**JS脚本和外部JS链接**，则会停止构建DOM树来执行和下载相应的代码，这会造成阻塞（这就是为什么推荐JS代码应该放在html代码的后面），之后根据外部样式，内部样式，内联样式构建一个**CSS对象模型树CSSOM树**，构建完成后和DOM树合并为**渲染树**，这里主要做的是**排除非视觉节点**，比如script，meta标签和排除display为none的节点，之后进行**布局**，布局主要是确定各个元素的位置和尺寸，之后是**渲染**页面（***重排与重绘***），因为html文件中会含有图片，视频，音频等资源，在解析DOM的过程中，遇到这些都会进行并行下载（浏览器对每个域的并行下载数量有一定的限制，一般是4-6个）。当然在这些所有的请求中我们还需要关注的就是**缓存**。（后面部分缓一手回答，引出缓存机制）~~缓存一般通过Cache-Control、Last-Modify、Expires等头部字段控制。（Cache-Control和Expires的区别在于Cache-Control使用相对时间，Expires使用的是基于服务器端的绝对时间，因为存在时差问题，一般采用Cache-Control，）在请求这些有设置了缓存的数据时，会先查看是否过期，如果没有过期则直接使用本地缓存，过期则请求并在服务器校验文件是否修改，如果上一次响应设置了ETag值会在这次请求的时候作为If-None-Match的值交给服务器校验，如果一致，继续校验Last-Modified，没有设置ETag则直接验证Last-Modified，再决定是否返回304。~~

## • 页面加载性能优化

关于性能优化涉及的方向太广，从网络请求到数据库整条链路都有其可优化的地方。这里从两个个维度进行讨论：

- **网络请求的优化**

浏览器渲染网页的前提是下载相关的资源，html文档、css文档、图片资源等。这些资源是客户端基于HTTP协议，通过网络请求从服务器端请求下载的，大家都知道，有网络，必定有延迟，而资源加载的网络延迟，是页面缓慢的一个重要因素。所以，如何使资源更快、更合理的加载，是性能优化的必修课。

**1. 静态资源**

**1）拼接、合并、压缩、制作雪碧图：**

由于HTTP的限制，在建立一个tcp请求时需要一些耗时，所以，我们对资源进行合并、压缩，其目的是减少http请求数和减小包体积，加快传输速度。

- 拼接、合并、压缩： 在现代的前端工程化开发流程中，相信大家都有使用webpack或者gulp等打包工具对资源（js、css、图片等）进行打包、合并、去重、压缩。在这基础上，我们需要根据自身的业务，合理的对公共代码，公共库，和首屏代码进行单独的打包压缩，按需加载；
- 雪碧图：对于图片资源，我们可以制作雪碧图，即对一些页面上的icon和小图标，集成到一张图片上，css使用背景图定位来使用不同的icon，这样做可以有效的减少图片的请求数，降低网络延迟。而它的缺点也很明显，由于集成在同一张图片上，使用其中的一个图标，就需要将整张图片下载下来，所以，雪碧图不能盲目的使用。

**2）CDN资源分发：**

将一些静态资源文件托管在第三方CDN服务中，一方面可以减少服务器的压力，另一方面，**CDN的优势在于，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上，保证资源的加载速度和稳定性**。

**3）缓存：**

缓存的范围很广，比如协议层的DNS解析缓存、代理服务器缓存，到客户端的浏览器本地缓存，再到服务端的缓存。一个网络链路的每个环节都有被缓存的空间。缓存的目的是简化资源的请求路径，比如某些静态资源在客户端已经缓存了，再次请求这个资源，只需要使用本地的缓存，而无需走网络请求去服务端获取。

**4）分片：**

分片指得是将资源分布到不同的主机，这是为了突破浏览器对同一台主机建立tcp连接的数量限制，一般为6~8个。现代网站的资源数量有50~100个很常见，所以将资源分布到不同的主机上，可以建立更多的tcp请求，降低请求耗时，从而提升网页速度。

**5）升级协议：**

可以升级我们的网络协议，比如使用HTTP2，quic 之类的，代替之前的http1.1，从协议层优化资源的加载。可以参考我之前的文章。

**2. 业务数据**

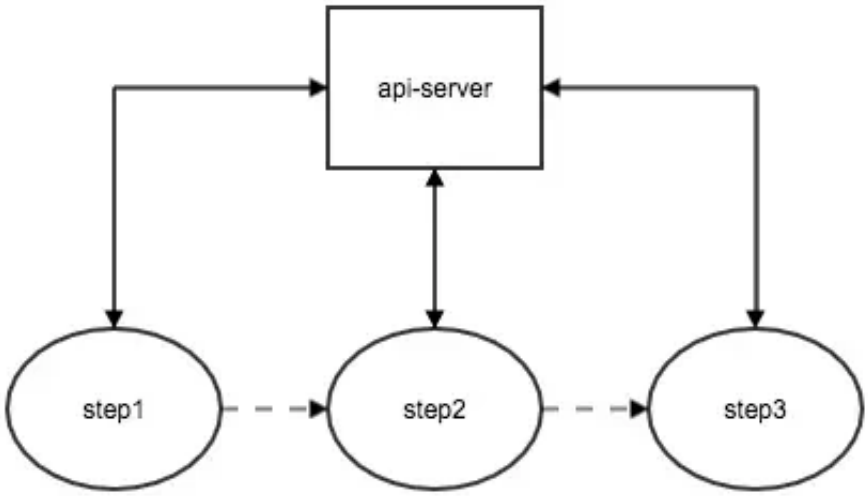
虽然做好了静态数据的加载优化，但是还是会出现一种情景，即静态数据已经加载完毕，但页面还是在转菊花，页面还没有进入可交互状态，这是因为现如今的网站开发模式，前后端分离已经成为主流，不再由php或jsp服务端渲染前端页面，而是前端先加载静态数据，再通过ajax异步获取服务器的数据，进而重新渲染页面。这就导致了异步从接口获取数据也是网页的一个性能瓶颈。响应缓慢，不稳定的接口，会导致用户交互体验极差，页面渲染速度也不理想。比如点击一个提交数据的按钮，接口速度慢，页面上菊花需要转好久才能交换完数据。

**1）首屏直出**

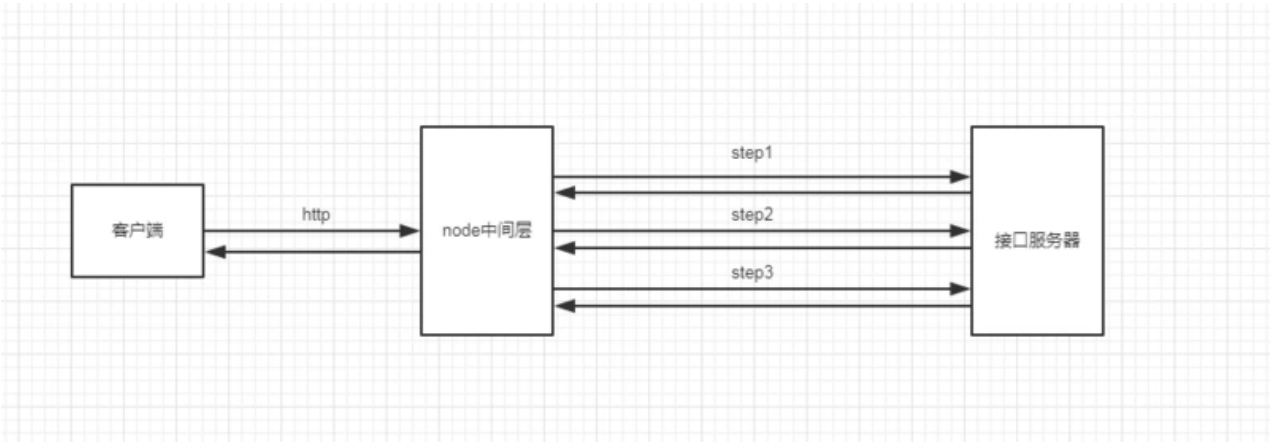
为了提升用户体验，我们认为首屏的渲染速度是极为重要的，用户进来页面，首页可见区域的加载可以由服务端渲染，保证了首屏加载速度，而不可见的部分则可以异步加载，甚至做到子路由页面的预加载。业界已经有很多同构直出的方案，比如vue的nuxt， react的beidou等。

**2）接口合并**

前端经常有这样的场景，完成一个功能需要先请求第一个接口获得数据，然后再根据数据请求第二个接口获取第二个数据，然后第三、第四...前端通常需要通过promise或者回调，一层一层的then下去，这样显然是很消耗性能的




通常后台接口都按一定的粒度存在的，不可能一个接口满足所有的场景。这是不可避免的，那么如何做到只发送一个请求就能实现功能呢？有一种不错的方案是，代理服务器实现请求合并，即后台的接口只需要保证健壮和分布式，而由nodejs（当然也可以使用其他语言）建设一层代理中间层，流程如下图所示：



前端只需要按照约定的规则，向代理服务器发起一次请求，由代理服务器向接口服务器发起三次请求，再将目标数据返回给客户端。这样做的好处是：**一方面是代理服务器代替前端做了接口合并，减少了前端的请求数量；另一方面代理服务器可以脱离HTTP的限制，使用更高效的通信协议与服务器通信**

- **页面渲染性能的优化**

**1. 防止阻塞渲染**

页面中的css 和 js 会阻塞html的解析，因为他们会影响dom树和render树。为了避免阻塞，我们可以做这些优化：

- css 放在首部，提前加载，这样做的原因是： 通常情况下 CSS 被认为是阻塞渲染的资源，在 CSSOM 构建完成之前，页面不会被渲染，放在顶部让样式表能够尽早开始加载。但如果把引入样式表的 link 放在文档底部，页面虽然能立刻呈现出来，但是页面加载出来的时候会是没有样式的，是混乱的。当后来样式表加载进来后，页面会立即进行重绘，这也就是通常所说的闪烁了。
- js文件放在底部，防止阻塞解析
- 一些不改变dom和css的js 使用 `defer` 和 `async` 属性告诉浏览器可以异步加载，不阻塞解析

**2. 减少重绘和回流**

重绘和回流在实际开发中是很难避免的，我们能做的就是尽量减少这种行为的发生。

- js尽量少访问dom节点和css 属性
- 尽可能的为产生动画的 HTML 元素使用 `fixed` 或 `absolute` 的 `position` ，那么修改他们的 CSS 是不会 Reflow 的。
- img标签要设置高宽，以减少重绘重排
- 把DOM离线后修改，如将一个dom脱离文档流，比如`display: none` ，再修改属性，这里只发生一次回流。
- 尽量用 `transform` 来做形变和位移，不会造成回流

**3. 提高代码质量**

这最能体现一个前端工程师的水平了，高性能的代码能在实现功能的同时，还兼顾性能。下面是一些好的实践：

**1）html：**

- dom的层级尽量不要太深，否则会增加dom树构建的时间，js访问深层的dom也会造成更大的负担。
- meta标签里需要定义文档的编码，便于浏览器解析

**2）css：**

- 减少 CSS 嵌套层级和选择适当的选择器，可参考如何提高css选择器性能
- 对于首屏的关键css 可以使用style标签内联。可参考什么是关键css

**3）js：**

- 减少通过JavaScript代码修改元素样式，尽量使用修改class名方式操作样式或动画
- 访问dom节点时需要对dom节点转存，防止循环中重复访问dom节点造成性能损耗。
- 慎用 定时器 和 计时器， 使用完后需要销毁。
- 用于复杂计算的js代码可以放在worker进程中运行

- 对于一些高频的回调需要对其节流和消抖，就是 `debounce` 和 `throttle` 这两个函数。比如 `scroll` 和 `touch` 事件

---

## Cookie，sessionStorage，localStorage的区别

> SessionStorage, LocalStorage, Cookie这三者都可以被用来在浏览器端存储数据，而且都是字符串类型的键值对。 区别在于前两者属于WebStorage，创建它们的目的便于客户端存储数据。而Cookie早在网景公司的浏览器中就开始支持，最初目的是为了保持HTTP的状态。

共同点：都是用来在浏览器端存储数据，而且都是字符串类型的键值对，且同源的。

区别：

- cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递（可能会浪费带宽）。而sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。
- cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下。
- 存储大小限制也不同，cookie数据不能超过4k，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如会话标识。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。
- 数据有效期不同，sessionStorage：仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持；localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie只在设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭。
- 作用域不同，sessionStorage**不在**不同的浏览器窗口中共享，即使是同一个页面；localStorage 在所有同源窗口中都是共享的；cookie也是在所有同源窗口中都是共享的。
- Web Storage 支持事件通知机制，可以将数据更新的通知发送给监听者。
- Web Storage 的 api 接口使用更方便。

应用场景：

cookie：

- 判断用户是否登陆过网站，以便下次登录时能够实现自动登录（或者记住密码）。如果我们删除cookie，则每次登录必须从新填写登录的相关信息。
- 保存上次登录的时间等信息。
- 保存上次查看的页面
- 浏览计数

| 特性 | cookie | sessionStorage | localStorage |
| --- | --- | --- | --- |
| 数据生命期 | 生成时就会被指定一个maxAge值，这就是cookie的生存周期，在这个周期内cookie有效，默认关闭浏览器失效 | 页面会话期间可用 | 除非数据被清除，否则一直存在 |
| 存放数据大小 | 4K左右（因为每次http请求都会携带cookie） | 一般5M或更大详细看这(需科学上网) | |
| 与服务器通信 | 由对服务器的请求来传递，每次都会携带在HTTP头中，如果使用cookie保存过多数据会带来性能问题 | 数据不是由每个服务器请求传递的，而是只有在请求时使用数据，不参与和服务器的通信 | |
| 易用性 | cookie需要自己封装setCookie，getCookie | 可以用源生接口，也可再次封装来对Object和Array有更好的支持 | |
| 共同点 | 都是保存在浏览器端，和服务器端的session机制不同（这里有一篇很好的介绍cookie和session的文章） | | |

## • csrf和xss的网络攻击及防范

CSRF：跨站请求伪造(Cross—Site Request Forgery)，可以理解为攻击者盗用了用户的身份，以用户的名义发送了恶意请求，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作，这时候CSRF就产生了。比如这个制造攻击的网站使用一张图片，但是这种图片的链接却是可以修改数据库的，这时候攻击者就可以以用户的名义操作这个数据库，防御方式：检查https头部的refer，使用token，在http头中自定义属性并验证

XSS：跨站脚本攻击(Cross Site Scripting)，是说攻击者通过注入恶意的脚本，在用户浏览网页的时候进行攻击，比如获取cookie，或者其他用户身份信息，XSS 的本质是因为网站没有对恶意代码进行过滤，与正常的代码混合在一起了，浏览器没有办法分辨哪些脚本是可信的，从而导致了恶意代码的执行。XSS 可以分为存储型和反射型，存储型是攻击者输入一些数据并且存储到了数据库中，其他浏览者看到的时候进行攻击，反射型的话不存储在数据库中，往往表现为将攻击代码放在url地址的请求参数中攻击者事先制作好攻击链接, 需要欺骗用户自己去点击链接才能触发XSS代码。防御方式：cookie设置httpOnly属性，对用户的输入进行检查，进行特殊字符过滤。

## • 设计模式

1. 单例模式

   确保一个类仅有一个实例，并提供一个访问它的全局访问点。一般用一个变量来标志当前的类已经创建过对象，或者利用闭包，如果下次获取当前类的实例时，直接返回之前创建的对象，否则重新创建

```
var singleton = function( fn ){
    var result;
    return function(){
        return result || ( result = fn .apply( this, arguments ) );
    }
}
var createMask = singleton( function(){
return document.body.appendChild( document.createElement('div') );
 })
```

2. 工厂模式

   提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。 工厂就是把成员对象的创建工作转交给一个外部对象，好处在于消除对象之间的耦合(何为耦合？就是相互影响)。通过使用工厂方法而不是new关键字及具体类，可以把所有实例化的代码都集中在一个位置，有助于创建模块化的代码，这才是工厂模式的目的和优势。

```
var XMLHttpFactory =function(){};           //这是一个简单工厂模式
     XMLHttpFactory.createXMLHttp =function(){
         var XMLHttp = null;
         if (window.XMLHttpRequest){
           XMLHttp = new XMLHttpRequest()
         }elseif (window.ActiveXObject){
            XMLHttp = new ActiveXObject("Microsoft.XMLHTTP")
         }
    return XMLHttp;
    }
    //XMLHttpFactory.createXMLHttp()这个方法根据当前环境的具体情况返回一个XHR对象。
    var AjaxHander =function(){
       var XMLHttp = XMLHttpFactory.createXMLHttp();
              ...
    }
```

3. 抽象工厂模式

   将其成员对象的实列化推迟到子类中，子类可以重写父类接口方法以便创建的时候指定自己的对象类型。 父类就变成了一个抽象类，但是父类可以执行子类中相同类似的方法，具体的业务逻辑需要放在子类中去实现。

4. 策略模式

   定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换 使用策略模式重构代码，可以消除程序中大片的条件分支语句。在实际开发中，我们通常会把算法的含义扩散开来，使策略模式也可以用来封装一系列的“业务规则”。只要这些业务规则指向的目标一致，并且可以被替换使用，我们就可以使用策略模式来封装他们

   普通模式

```
  var awardS = function (salary) {
```

```
  return salary * 4
};

var awardA = function (salary) {
  return salary * 3
};

var awardB = function (salary) {
  return salary * 2
};

var calculateBonus = function (level, salary) {
  if (level === 'S') {
    return awardS(salary);
  }
  if (level === 'A') {
    return awardA(salary);
  }
  if (level === 'B') {
    return awardB(salary);
  }
};

calculateBonus('A', 10000);
```

策略模式

```
var strategies = {
  'S': function (salary) {
    return salary * 4;
  },
  'A': function (salary) {
    return salary * 3;
  },
  'B': function (salary) {
    return salary * 2;
  }
}

var calculateBonus = function (level, salary) {
  return strategies[level](salary);
}

calculateBonus('A', 10000);
```

5. 模板方法模式

模板方法模式使用了原型链的方法，封装性好，复用性差 模板方法模式由二部分组成，第一部分是抽象父类，第二部分是具体实现的子类，一般的情况下是抽象父类封装了子类的算法框架，包括实现一些公共方法及封装子类中所有方法的执行顺序，子类可以继承这个父类，并且可以在子类中重写父类的方法，从而实现自己的业务逻辑。 模板方法是基于继承的设计模式，可以很好的提高系统的扩展性。 java中的抽象父类、子类 模板方法有两部分结构组成，第一部分是抽象父类，第二部分是具体的实现子类。

6. 职责链模式

   职责链，一系列可能处理请求的对象被连接成一条链，请求在这些对象之间依次传递，直到遇到一个可以处理它的对象，减少了很多重复代码。 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系，将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

7. 发布订阅者模式（观察者模式）

   发布订阅模式，顾名思义，就是一个发布消息，一个监听消息，当有消息接收时处理消息。

```
function Publisher(name) {
    this.name = '报纸' + name
    // 订阅了本出版社的读者列表
    this.subscribers = [];
}

// 添加一个迭代方法，遍历所有投递列表
Publisher.prototype.deliver = function(data) {
    // 遍历当前出版社对象所有的订阅过的方法
    const context = this
    this.subscribers.forEach(function(fn) {
        //data用于传参数给内部方法
        fn.fire(context.name, data);
    });
    return this;
}

// 观察者
function Observe(callback) {
    this.fire = callback;
}

// 给予订阅者阅读的能力
Observe.prototype.subscribe = function(publisher) {

    var that = this;
    // some如果有一个返回值为true则为true
    var alreadyExists = publisher.subscribers.some(function(el){
        // 这里的el指的是函数对象
        return el.fire === that.fire // 查看是否已经订阅了，如果订阅了就不再
订阅
    });
```

```
    // 如果存在这个函数对象则不进行添加
    if (!alreadyExists) {
        publisher.subscribers.push(this);
    }
    return this;
};

// 给予观察者退订的能力
Observe.prototype.unsubscribe = function(publisher) {

    var that = this;

    // 过滤，将返回值为true的重组成数组
    publisher.subscribers = publisher.subscribers.filter(function(el) {

        // 这里的el.fire指的是观察者传入的callback
        // that.fire就是当前对象对callback的引用
        return !(el.fire === that.fire) // 删除掉订阅
    })
    return this;
};

var publisher1 = new Publisher('xixi');

// 实例化的读者，这个需要改进
var observer1 = new Observe(function(publisher, text) {
        console.log('得到来自'+publisher +'的订阅消息：'+ text + ',哈哈哈')
});
var observer2 = new Observe(function(publisher, text) {
    console.log('得到来自'+publisher +'的订阅消息：'+ text + ',嘻嘻嘻')
});

// 读者订阅了一家报纸,在报社可以查询到该读者已经在订阅者列表了,
// publisher1.subscribers->[observer1]
observer1.subscribe(publisher1);
observer2.subscribe(publisher1)

// 分发报纸，执行所有内在方法
publisher1.deliver(13);// 输出123

// 退订
observer1.unsubscribe(publisher1);
publisher1.deliver(12388);
```

8. 桥接模式
9. 适配器模式

适配器模式（Adapter）是将一个类（对象）的接口（方法或属性）转化成客户希望的另外一个接口（方法或属性），适配器模式使得原本由于接口不兼容而不能一起工作的那些类（对象）可以一些工作。

10. 代理模式

    代理模式的定义是把对一个对象的访问, 交给另一个代理对象来操作

## • HTML5 新特性

首先html5为了更好的实践web语义化，增加了header、footer、nav、aside、section等语义化标签，在表单方面，为了增强表单，为input增加了color、emial、data、range等类型，在存储方面，提供了sessionStorage，localStorage,和离线存储，通过这些存储方式方便数据在客户端的存储和获取，在多媒体方面规定了音频和视频元素audio和vedio，另外还有地理定位，canvas画布，拖放API，多线程编程的web worker和websocket协议

---

# CSS

## • css盒子模型

就是用来装页面上的元素的矩形区域。CSS中的盒子模型包括IE盒子模型和标准的W3C盒子模型。

在CSS3中引入了box-sizing属性：box-sizing:content-box;表示标准的盒子模型，box-sizing:border-box表示的是IE盒子模型，box-sizing:padding-box,这个属性值的宽度包含了左右padding+width

（记忆：包含什么，width就从什么开始算起。）

---

## • 行内元素，块元素

- 行内元素：
    - HTML4 中，元素被分成两大类: inline (内联元素) 与 block(块级元素)。一个行内元素只占据它对应标签的边框所包含的空间。
    - 常见的行内元素有 a b span img strong sub sup button input label select textarea
- 块元素：
    - 块级元素占据其父元素(容器)的整个宽度，因此创建了一个“块”。 常见的块级元素有 div ul ol li dl dt dd h1 h2 h3 h4 h5 h6 p
- 行内元素与块级元素的区别?
    - 格式上，默认情况下，行内元素不会以新行开始，而块级元素会新起一行。
    - 内容上，默认情况下，行内元素只能包含文本和其他行内元素。而块级元素可以包含行内元素和其他块级元素。
    - 行内元素与块级元素属性的不同，主要是盒模型属性上:行内元素设置 width 无效，height 无效(可以设置 line-height)，设置 margin 和 padding 的**上下**不会对其他元素产生影响。

---

## • css选择器

| 名称 | 举例 |
| --- | --- |
| ID | #id |
| class | .class |
| 标签 | p |
| 通用 | * |
| 属性 | [type="text"] |
| 伪类 | :hover |
| 伪元素 | ::first-line |
| 子选择器、相邻选择器 | :first-child |

权重计算规则：

1. !important
2. 第一等：代表内联样式，如: style=””，权值为1000。
3. 第二等：代表ID选择器，如：#content，权值为0100。
4. 第三等：代表类，伪类和属性选择器，如.content，权值为0010。
5. 第四等：代表类型选择器和伪元素选择器，如div p，权值为0001。
6. 通配符、子选择器、相邻选择器等的。如*、>、+,权值为0000。
7. 继承的样式没有权值。

### 常见选择器

1. 标签选择器 写法： HTML标签名{} 作用：选中页面中所有与选择器同名的HTML标签

```
li{
      color: red;
}
```

2. 类选择器（class选择器）

   写法：.class名{}

   调用：在需要应用这套样式的标签上，使用class="class名"调用选择器

   优先级：class选择器>标签选择器

```
.ji{
      color: blue;
}
```

3. ID选择器

写法：#ID名{}

调用：在需要应用这套样式的标签上，使用id="ID名"调用选择器

优先级：ID选择器>class选择器

```
#two{
      color: yellow;
}

    /* 【class选择器和ID选择的区别】
     * 1. 写法不同：class选择器用.声明，ID选择器用#声明；
     * 2. 优先级不同：ID选择器>class选择器
     * 3. 调用次数不同： class选择器可以调用多次； ID选择器只能调用一次(同一页面，不能出现同名ID)。
     */
```

4. 通用选择器

   写法：*{}

   作用：作用于页面中所有的HTML标签

   优先级：最低！低于标签选择器

```
*{
      background-color: yellow;
      color: pink;
}
```

5. 并集选择器

   写法： 选择器1,选择器2,.....选择器n{} 多个选择器之间逗号分隔

   生效规则： 多个选择器取并集，只要标签满足其中任何一个选择器，样式即可生效

   (其实相当于多个选择器拆开写)

```
li,.ji{
      color: red;
}
```

6. 交集选择器

   写法： 选择器1选择器2.....选择器n{} 多个选择器之间紧挨着，没有分隔

   生效规则： 多个选择器取交集，必须满足所有选择器的要求，样式才能生效。

```
li.ji{
      color: red;
}
```

7. 后代选择器

选择器1 选择器2 ..... 选择器n{}

生效规则： 只要满足后一个选择器是前一个选择器的后代，样式即可生效（后代包括子代、孙代。。。）

```
div .ji{
      color: red;
}
```

8. 子代选择器

   选择器1>选择器2>.....>选择器n{}

   生效规则： 必须满足后一个选择器是前一个选择器的**直接**子代，样式才能生效。

```
div>ul>.ji{
      color: red;
}
```

## 伪类选择器

1. 写法： 伪类选择器，在选择器的后面，用:分隔，并紧接伪类状态；

   eg: a:hover{}

2. 子元素选择器
   - :first-child 匹配父元素的第一个子元素的是该元素元素

```
<html>
<head>
<style type="text/css">
p:first-child {
  color: red;
  }
</style>
</head>

<body>
  <p>some text</p> <!--该p元素为body元素的第一个子元素，即被选中-->
  <p>some text</p>
</body>
</html>
```

```
<html>
<head>
<style type="text/css">
p:first-child {
  color: red;
  }
 /* p元素的父元素的第一个子元素为a元素，故不匹配 */
```

```
</style>
</head>

<body>
  <a>aaa</a>
  <p>some text</p>
  <p>some text</p>
</body>
</html>
```

**提示：**最常见的错误是认为 p:first-child 之类的选择器会选择 p 元素的第一个子元素，正确的元素群体为指定元素的父元素的子元素群体。

- :last-child 匹配父元素的最后一个子元素的元素
- :nth-child(n) 选择器匹配属于其父元素的第 N 个子元素，不论元素的类型

3. 超链接的伪类状态：
   - :link - 未访问状态
   - :visited - 已访问状态
   - :hover - 鼠标指上状态
   - :active - 激活选定状态(鼠标点下未松开)

   注意： 当超链接多种伪类状态共存时，必须按照link-visited-hover-active 的顺序排列，否则会导致某些状态不能生效。
4. input的常用伪类状态：
   - :hover :focus - 获得焦点的状态
   - :active

   注意： 当input多种状态共存时，必须按照上述顺序
5. 其他标签，基本只有:hover事件

```
.a:link{
  color: yellow;
}
.a:visited{
  color: red;
}
.a:hover{
  color: blue;
}
.a:active{
  color: green;
}

input:hover{
  color: red;
}
input:focus{
  color: blue;
```

```
}
input:active{
  color: yellow;
}
```

### 伪类与伪元素的区别

css 引入伪类和伪元素概念是为了格式化文档树以外的信息。也就是说，伪类和伪元素是用来修饰不在文档树中的部分。

伪类用于当已有的元素处于某个状态时，为其添加对应的样式，这个状态是根据用户行为而动态变化的。比如说，当用户悬停在指定的元素时，我们可以通过:hover 来描述这个元素的状态。

伪元素用于创建一些不在文档树中的元素，并为其添加样式。它们允许我们为元素的某些部分设置样式。比如说，我们可以通过::before 来在一个元素前增加一些文本，并为这些文本添加样式。虽然用户可以看到这些文本， 但是这些文本实际上不在文档树中。

---

## • position 定位

- 相对定位relative：

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

- 绝对定位absolute：

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

- 固定定位fixed：

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

- 粘性定位sticky：

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

- 默认定位Static：

  默认值。没有定位，元素出现在正常的流中（忽略top, bottom, left, right 或者 z-index 声明）。

- 继承定位inherit:

  规定应该从父元素继承position 属性的值。

---

- ## css动画

### transition

transition 属性是一个简写属性，用于设置四个过渡属性：

```
transition: property duration timing-function delay;
```

| 值 | 描述 |
| --- | --- |
| transition-property | 规定设置过渡效果的 CSS 属性的名称。 |
| transition-duration | 规定完成过渡效果需要多少秒或毫秒。 |
| transition-timing-function | 规定速度效果的速度曲线。 |
| transition-delay | 定义过渡效果何时开始。 |

```
img{
    height:15px;
    width:15px;
    transition: 1s; /* 指定状态变化所需要的时间 */
}

img:hover{
    height: 450px;
    width: 450px;
}
```

**transition的局限**

transition的优点在于简单易用，但是它有几个很大的局限。

```
1. transition需要事件触发，所以没法在网页加载时自动发生。

2. transition是一次性的，不能重复发生，除非一再触发。

3. transition只能定义开始状态和结束状态，不能定义中间状态，也就是说只有两个状态。

4. 一条transition规则，只能定义一个属性的变化，不能涉及多个属性。
```

### animation

animation 属性是一个简写属性，用于设置六个动画属性：

```
animation: name duration timing-function delay iteration-count direction;
```

| 值 | 描述 |
| --- | --- |
| *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-fill-mode* | 规定动画结束时保持的状态。 |
| *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。 |

```
div:hover {
  animation: 1s rainbow infinite;
}

@keyframes rainbow {
  0% { background: #c00; }
  50% { background: orange; }
  100% { background: yellowgreen; }
}
```

- keyframes的写法

  keyframes关键字用来定义动画的各个状态，它的写法相当自由。

  ```
  @keyframes rainbow {
    0% { background: #c00 }
    50% { background: orange }
    100% { background: yellowgreen }
  }
  ```

  0%可以用from代表，100%可以用to代表，因此上面的代码等同于下面的形式。

  ```
  @keyframes rainbow {
    from { background: #c00 }
    50% { background: orange }
    to { background: yellowgreen }
  }
  ```

  如果省略某个状态，浏览器会自动推算中间状态。

  甚至，可以把多个状态写在一行。

  另外一点需要注意的是，浏览器从一个状态向另一个状态过渡，是平滑过渡。steps函数可以实现分步过渡。

```
div:hover {
  animation: 1s rainbow infinite steps(10);
}
```

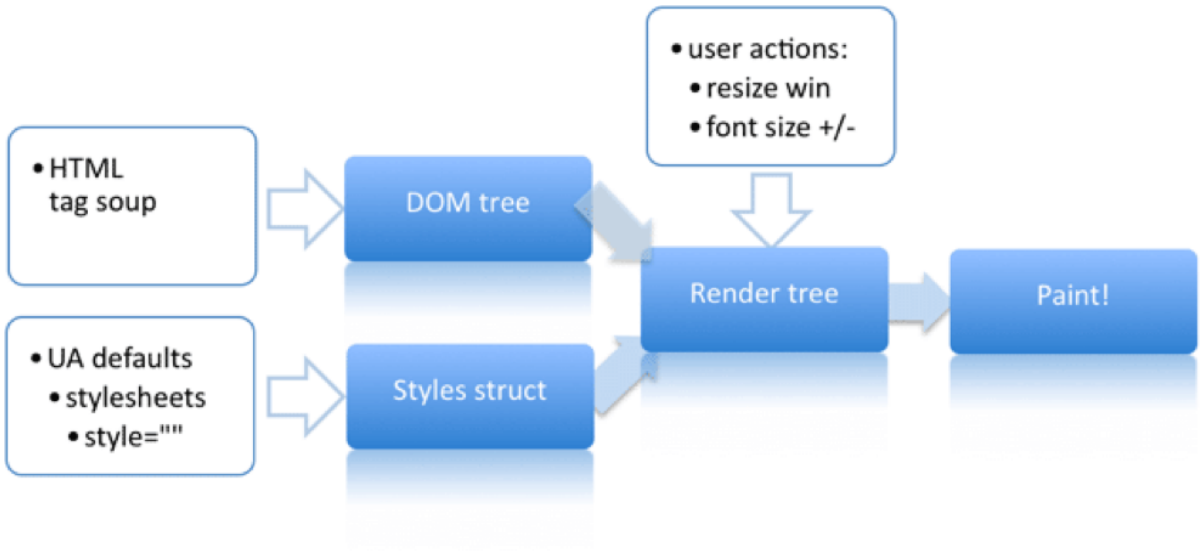
> *一个用animation实现的打字机动效*

```
@keyframes typing { from { width: 0; } }
@keyframes blink-caret { 50% { border-color: transparent; } }

h1 {
  font: bold 200% Consolas, Monaco, monospace;
  border-right: .1em solid;
  width: 16.5em; /* fallback */
  width: 30ch; /* # of chars */
  margin: 2em 1em;
  white-space: nowrap;
  overflow: hidden;
  animation: typing 20s steps(30, end), /* # of steps = # of chars
*/
             blink-caret .5s step-end infinite alternate;
}
```

---

- **重排，重绘**

> **浏览器对页面呈现的处理：**




1. 浏览器把获取到的HTML代码解析成1个DOM树，HTML中的每个标签都是DOM树中的1个节点，根节点就是我们常用的document对象。DOM树里包含了所有HTML标签，包括display:none;隐藏的，还有用JS动态添加的元素等。
2. 浏览器把所有样式(用户定义的CSS和用户代理)解析成样式结构体，在解析的过程中会去掉浏览器不能识别的样式，比如IE会去掉-moz开头的样式，而FF会去掉_开头的样式。
3. DOM Tree 和样式结构体组合后构建render tree, render tree类似于DOM tree，但区别

> 很大，render tree能识别样式，render tree中每个NODE都有自己的style，而且render tree不包含隐藏的节点 (比如display:none的节点，还有head节点)，因为这些节点不会用于呈现，而且不会影响呈现的，所以就不会包含到 render tree中。注意visibility:hidden隐藏的元素还是会包含到 render tree中的，因为visibility:hidden 会影响布局(layout)，会占有空间。根据CSS2的标准，render tree中的每个节点都称为Box (Box dimensions)，理解页面元素为一个具有填充、边距、边框和位置的盒子。

**重绘（repaint或redraw）：**

重绘是指当 DOM 元素的属性发生变化 (如 color) 时, 浏览器会通知 render 重新描绘相应的元素, 此过程称为重绘。

**重排（重构）：**

重排是指某些元素变化涉及元素布局 (如width), 浏览器则抛弃原有属性, 重新计算，此过程称为重排。（重排一定会重绘，重绘不一定重排）

**回流（reflow）：**

重排好的结果，传递给render以重新描绘页面元素, 此过程称为回流

**什么会影响回流?**

1. 添加或者删除可见的DOM元素；
2. 元素位置改变；
3. 元素尺寸改变——边距、填充、边框、宽度和高度；
4. 内容改变——比如文本改变或者图片大小改变而引起的计算值宽度和高度改变；
5. 页面渲染初始化；
6. 浏览器窗口尺寸改变——resize事件发生时；
7. 操作：offsetTop/offsetLeft/offsetWidth/offsetHeight/scrollTop/Left/Width/Height/clientTop/Left/Width/Height/width/height/getComputedStyle()/currentStyle ...

**优化**

重排和重绘会不断触发，这是不可避免的。但是，它们非常耗费资源，是导致网页性能低下的根本原因。

提高网页性能，就是要降低"重排"和"重绘"的频率和成本，尽量少触发重新渲染。

```
div.style.color = 'blue';
div.style.marginTop = '30px';
```

上面代码中，div元素有两个样式变动，但是浏览器只会触发一次重排和重绘。

```
div.style.color = 'blue';
var margin = parseInt(div.style.marginTop);
div.style.marginTop = (margin + 10) + 'px';
```

上面代码对div元素设置背景色以后，第二行要求浏览器给出该元素的位置，所以浏览器不得不立即重排。

**优化方案：**

1. DOM的多个读操作（或多个写操作），应该放在一起。不要两个读操作之间，加入一个写操作。
2. 如果某个样式是通过重排得到的，那么最好缓存结果。避免下一次用到的时候，浏览器又要重排。

**js 代码：**

```
// 别这样写，大哥
for(循环) {
el.style.left = el.offsetLeft + 5 + "px";
el.style.top = el.offsetTop + 5 + "px";
}
// 这样写好点
var left = el.offsetLeft,
top = el.offsetTop,
s = el.style;
for (循环) {
left += 10;
top += 10;
s.left = left + "px";
s.top = top + "px";
}
```

3. 不要一条条地改变样式，而要通过改变class，或者csstext属性，一次性地改变样式。
4. 尽量使用离线DOM，而不是真实的网面DOM，来改变元素样式。比如，使用 cloneNode() 方法，在克隆的节点上进行操作，然后再用克隆的节点替换原始节点。
5. 先将元素设为display: none（需要1次重排和重绘），然后对这个节点进行100次操作，最后再恢复显示（需要1次重排和重绘）。这样一来，你就用两次重新渲染，取代了可能高达100次的重新渲染。
6. position属性为absolute或fixed的元素，重排的开销会比较小，因为不用考虑它对其他元素的影响。
7. 只在必要的时候，才将元素的display属性为可见，因为不可见的元素不影响重排和重绘。另外，visibility: hidden的元素只对重绘有影响，不影响重排。
8. 使用虚拟DOM的脚本库，比如Vue等。

---

## ● 实现元素的居中

1. **使用margin进行固定长度的偏移**

```
/*关键样式代码*/
#father{
  overflow: hidden;
}
#son{
  margin:0 auto;/*水平居中*/
  margin-top: 50px;
}
```

对父元素使用overflow：hidden，子元素使用margin:0 auto;进行水平方向居中，margin-top进行下移实现垂直居中。

缺点：需要明白子元素的具体的尺寸，当子元素的尺寸有改变时，布局就会被破坏，维护性也极差

2. **使用绝对定位并进行偏移**

```
/*关键样式代码*/
#father{
         position:relative;
       }
   #son{
         position: absolute;
         left:50%;
         margin-left: -50px;
         top:50%;
         margin-top: -50px;
        }
```

在使用绝对定位进行百分比偏移时他是将子元素整体偏移到另半边，而不是将子元素的中轴线对准父元素的中轴线，所以还要使用margin-left：-50px;将子元素的一半偏移回来。

```
/*优化代码（使用css样式中的计算公式）*/
#son{
        position: absolute;
        left:calc(50% - 50px);
        top:calc(50% - 50px);
        }
```

计算公式中的运算符前后都需要有空格隔开！

缺点：当子元素的width和height未知时，无法通过设置 `margin-left:-width/2` 和 `margin-top:-height/2` 来实现，这时候可以设置子元素的 `transform: translate(-50%,-50%)` 。

> 绝对定位、相对定位属性加上 top、left ，会影响offsetTop和 offsetLeft 的值；使用 translate 的offsetTop和 offsetLeft 与没有产生位移的元素没有区别，即无论 translate 的值为多少，offsetTop和 offsetLeft 的值都是固定不变的。

3. **使用绝对定位并margin自适应进行居中**

```
/*关键样式代码*/
#father{
         position:relative;
       }
   #son{
         position: absolute;
         left: 0;
         top: 0;
         right: 0;
         bottom: 0;
         margin:auto;
        }
```

这种居中方式采用的是流体自适应居中，将left/top/bottom/right四个属性都设置为0表示子模块相对于四条边的偏移量都是零，这时再放进去一个margin:auto;，在满足前边四个属性的条件下进行margin的auto布局，就可以实现垂直居中。

4. **使用table-cell进行居中显示**

```
/*关键样式代码*/
 #father{
            display: table-cell;
            vertical-align: middle;
        }
    #son{
            margin: 0 auto;
        }
```

将父元素display:设置为table-cell，此时就可以使用vertical-align: middle对内部的子元素进行垂直居中。之后在子元素样式中使用margin: 0 auto;实现水平居中

5. **使用弹性盒子(Flex布局)来实现居中**

```
/*关键样式代码*/
 #father{
           display: flex;
           justify-content: center;
           align-items: center;
        }
```

设置display为flex，同时在默认情况下flex-direction为row，即主轴线为水平方向，副轴为垂直方向。设置justify-content为center，可以将元素在主轴（水平）方向上居中显示，设置align-items可以将元素在副轴（垂直）方向上居中显示。(推荐)

## • Flex 布局

Flex 是 FlexibleBox 的缩写，意为"弹性盒子布局"，是 CSS3 新增的一种布局方式，可以通过将一个元素的 display 属性值设置为 flex 从而使它成为一个 flex 容器，它的所有子元素都会成为它的项目。 一个容器默认有两条轴，一个是水平的主轴，一个是与主轴垂直的交叉轴。我们可以使用 flex-direction 来指定主轴的方向。 我们可以使用 justify-content 来指定元素在主轴上的排列方式，使用 align-items 来指定元素在交叉轴上的排列方式。还可以使用 flex-wrap 来规定当一行排列不下时的换行方式。对于容器中的项目，我们可以使用 order 属性来指定项目的排列顺序，可以使用 flex-grow 来指定当排列空间有剩余的时候， 项目的放大比例。可以使用 flex-shrink 来指定当排列空间不足时，项目的缩小比例。可以使用 flex-basis 定义了在分配多余空间之前，项目占据的主轴空间。

以下 6 个属性设置在容器上。

- flex-direction: row | row-reverse | column | column-reverse; 属性决定主轴的方向（即项目的排列方向）
- flex-wrap: nowrap | wrap | wrap-reverse;属性定义，如果一条轴线排不下，如何换行
- flex-flow: | | ;属性是 flex-direction 属性和 flex-wrap 属性的简写形式，默认值为 row nowrap
- justify-content: flex-start | flex-end | center | space-between | space-around; 属性定义了项目在主轴上的对齐方式
- align-items: flex-start | flex-end | center | baseline | stretch; 属性定义项目在交叉轴上如何对齐
- align-content: flex-start | flex-end | center | space-between | space-around | stretch 属性定义了多根轴线的对齐方式。如果项目只有一根轴线，该属性不起作用

以下 6 个属性设置在项目上。

- order 属性定义项目的排列顺序。数值越小，排列越靠前，默认为 0
- flex-grow 属性定义项目的放大比例，默认为 0，即如果存在剩余空间，也不放大
- flex-shrink 属性定义了项目的缩小比例，默认为 1，即如果空间不足，该项目将缩小
- flex-basis 属性定义了在分配多余空间之前，项目占据的主轴空间。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为 auto，即项目的本来大小
- flex 属性是 flex-grow，flex-shrink 和 flex-basis 的简写，默认值为 0 1 auto

> 当 `flex` 取值为 `none`，则计算值为 0 0 auto； 当 `flex` 取值为 `auto`，则计算值为 1 1 auto； 当 `flex` 取值为一个非负数字，则该数字为 `flex-grow` 值， `flex-shrink` 取 1， `flex-basis` 取 0% 故 flex: 1 等价于 flex: 1 1 0%

- align-self 属性允许单个项目有与其他项目不一样的对齐方式，可覆盖 align-items 属性。默认值为 auto，表示继承父元素的 align-items 属性，如果没有父元素，则等同于 stretch。

---

## • 双栏布局

1. 将左侧div浮动，右侧div设置margin-left

```
.c{overflow: hidden;}
.l{float: left;width: 200px;}
.r{margin-left: 200px;}
```

2. 固定区采用绝对定位，自适应区设置margin

```
.c{position: relative;}
.l{position: absolute;left: 0;top: 0;width: 200px}
.r{margin-left: 200px;}
```

3. table布局

```
.c{display: table;}
.l{display: table-cell;width: 200px;}
.r{display: table-cell;}
```

4. 双float + calc()计算属性

```
.c{overflow: hidden;}
.l{float: left;width: 200px;}
.r{float: left;width: calc(100% - 200px);} /* 100vw - 200px */
```

5. float + BFC方法

```
.c{overflow: hidden;}
.l{float: left;width: 200px;}
.r{overflow: auto}
```

6. flex

```
.c{display: flex;}
.l{flex: 0 0 200px;}
.r{flex: 1;}
```

---

## • margin外边距折叠

多个相邻（兄弟或者父子关系）普通流的块元素垂直方向marigin会重叠

**折叠边距计算：**

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。

**防止外边距重叠解决方案：**

1. 外层元素padding代替
2. 外层元素 overflow:hidden;
3. 内层元素绝对定位 postion:absolute:
4. 内层元素 加float:left;或display:inline-block;
5. 内层元素padding:1px;
6. 内层元素透明边框 border:1px solid transparent;

## 清除浮动

> 在非IE浏览器（如Firefox）下，当容器的高度为auto，且容中仅有浮动（float为left或right）的元素，在这种情况下，容器的高度不能自动伸长以适应内容的高度，使得内容溢出到容器外面而影响（甚至破坏）布局的现象。这个现象叫浮动溢出，为了防止这个现象的出现而进行的CSS处理，就叫CSS清除浮动。

1. 使用带clear属性的空元素

   在浮动元素后使用一个空元素如

   ，并在CSS中赋予.clear{clear:both;}属性即可清理浮动。亦可使用
   或

   ---

   来进行清理。

2. 使用CSS的overflow属性

   给浮动元素的容器添加overflow:hidden;或overflow:auto;可以清除浮动，另外在 IE6 中还需要触发 hasLayout ，例如为父元素设置容器宽高或设置 zoom:1。

   当元素设置了overflow样式且值部位visible时，该元素就构建了一个BFC，BFC在计算高度时，内部浮动元素的高度也要计算在内，也就是说即使BFC区域内只有一个浮动元素，BFC的高度也不会发生塌缩，所以达到了清除浮动的目的。

3. 给浮动的元素的容器添加浮动

   给浮动元素的容器也添加上浮动属性即可清除内部浮动，但是这样会使其整体浮动，影响布局，不推荐使用。

4. 使用邻接元素处理

   什么都不做，给浮动元素后面的元素添加clear类，并在CSS中赋予.clear{clear:both;}。

5. 使用CSS的:after伪元素

   结合:after 伪元素（注意这不是伪类，而是伪元素，代表一个元素之后最近的元素）和 IEhack ，可以完美兼容当前主流的各大浏览器，这里的 IEhack 指的是触发 hasLayout。

   给浮动元素的容器添加一个clearfix的class，然后给这个class添加一个:after伪元素实现元素末尾添加一个看不见的块元素（Block element）清理浮动。

   ```
   .clearfix:after{
     content: "020";
     display: block;
     height: 0;
     clear: both;
     visibility: hidden;
   }
   ```

**总结**

通过上面的例子，我们不难发现清除浮动的方法可以分成两类：

一是利用 clear 属性，包括在浮动元素末尾添加一个带有 clear: both 属性的空 div 来闭合元素，其实利用 :after 伪元素的方法也是在元素末尾添加一个内容为一个点并带有 clear: both 属性的元素实现的。

二是触发浮动元素父元素的 BFC (Block Formatting Contexts, 块级格式化上下文)，使到该父元素可以包含浮动元素。

**推荐**

在网页主要布局时使用:after伪元素方法并作为主要清理浮动方式；在小模块如ul里使用overflow:hidden;；如果本身就是浮动元素则可自动清除内部浮动，无需格外处理；正文中使用邻接元素清理之前的浮动。

最后可以使用相对完美的:after伪元素方法清理浮动，文档结构更加清晰。

---

## • BFC

BFC也就是常说的块格式化上下文，这是一个独立的渲染区域，规定了内部如何布局，并且这个区域的子元素不会影响到外面的元素。

**触发条件：**

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

**规则：**

- 浮动的元素会被父级计算高度（父级触发了BFC）
- 非浮动元素不会覆盖浮动元素位置（非浮动元素触发了BFC）
- margin不会传递给父级（父级触发了BFC），两个相邻元素上下margin会重叠（给其中一个元素增加一个父级，然后让他的父级触发BFC）

**巧妙用法：**

- 清除浮动
- 非浮动元素盖住浮动元素，可依靠触发BFC来解决
- margin合并情况，在其中一个元素父级触发BFC，将不会合并margin

---

## • CSS hack

由于不同的浏览器，比如IE6,IE7,Firefox等，对CSS的解析认识不一样，因此会导致生成的页面效果不一样，得不到我们所需要的页面效果。 这个时候我们就需要针对不同的浏览器去写不同的CSS，让它能够同时兼容不同的浏览器，能在不同的浏览器中也能得到我们想要的页面效果。

简单的说，CSS hack的目的就是使你的CSS代码兼容不同的浏览器。当然，我们也可以反过来利用CSS hack为不同版本的浏览器定制编写不同的CSS效果。

---

## • 省略号替代文字超出部分

```
/* 单行文本 */
overflow: hidden; /*超出文本不显示*/
text-overflow:ellipsis; /*超出部分文本用省略号替代*/
white-space: nowrap; /*强制不换行*/

/* 多行文本（3行） */
display: -webkit-box
-webkit-box-orient:vertical
-webkit-line-clamp:3
overflow:hidden
```

---

## • less，sass，stylus区别

> CSS 预处理器是一种语言用来为 CSS 增加一些编程的的特性，无需考虑浏览器的兼容性问题，例如你可以在 CSS 中使用变量、简单的程序逻辑、函数等等在编程语言中的一些基本技巧，可以让CSS 更见简洁，适应性更强，代码更直观等诸多好处。

- **基本语法**

Less 的基本语法属于「CSS 风格」，Sass、Stylus 相比之下激进一些，利用缩进、空格和换行来减少需要输入的字符，不过区别在于 Sass、Stylus 同时也兼容「CSS 风格」代码

- **基本语法**

Less & SCSS：

```
/* Less,Scss */
.box {
  display: block;
}
```

```
/* Sass */
.box
  display: block
```

```
/* Stylus */
.box
  display: block
```

- **嵌套**

三者的嵌套语法都是一致的，甚至连引用父级选择器的标记 & 也相同。

区别只是 Sass 和 Stylus 可以用没有大括号的方式书写

```
/* Less */
.a {
  &.b {
    color: red;
  }
}
```

- **变量**

less使用@进行变量声明和引用，sass使用$进行变量声明和引用，stylus不需要额外的标志符

- **@import**

Less 中可以在字符串中进行插值：

```
@device: mobile;
@import "styles.@{device}.css";
```

Sass 中只能在使用 url() 表达式引入时进行变量插值：

```
$device: mobile;
@import url(styles.#{$device}.css);
```

Stylus 中在这里插值不管用，但是可以利用其字符串拼接的功能实现：

```
device = "mobile"
@import "styles." + device + ".css"
```

- **mixin混入**

混入（mixin）是预处理器最精髓的功能之一了，它提供了 CSS 缺失的最关键的东西：样式层面的抽象。

```
/* Less */
.alert {
  font-weight: 700;
}

.highlight(@color: red) {
  font-size: 1.2em;
  color: @color;
}

.heads-up {
  .alert;
  .highlight(red);
}
```

```
/* Sass */
@mixin large-text {
  font: {
    family: Arial;
    size: 20px;
    weight: bold;
  }
  color: #ff0000;
}

.page-title {
  @include large-text;
  padding: 4px;
  margin-top: 10px;
}
```

- 继承

```
/* Stylus,Scss */
.message
  padding: 10px
  border: 1px solid #eee

.warning
  @extend .message
  color: #e2e21e
```

```
/* Less */
.message {
  padding: 10px;
  border: 1px solid #eee;
}

.warning {
  &:extend(.message);
  color: #e2e21e;
}
```

```
/* Sass */
.active {
   color: red;
}
button.active {
   @extend .active;
}
```

- 函数

三种预处理器都自带了诸如色彩处理、类型判断、数值计算等内置函数

```
/* Stylus */
@function golden-ratio($n) {
  @return $n * 0.618;
}

.golden-box {
  width: 200px;
  height: golden-ratio(200px);
}
```

**附：** Scss 与 Sass 是什么,他们的区别在哪里？ （文件拓展名、语法书写方式 - sass严格缩进不带大括号分号）

---

## • em，rem，px区别

1. **px（像素）**

   px单位的名称为像素，它是一个固定大小的单元，像素的计算是针对（电脑/手机）屏幕的，一个像素（1px）就是（电脑/手机）屏幕上的一个点，即屏幕分辨率的最小分割。由于它是固定大小的单位，单独用它来设计的网页，如果适应大屏幕（电脑），在小屏幕（手机）上就会很不友好，做不到自适应的效果。

   **px是固定长度单位，不随其它元素的变化而变化**

2. **em（相对长度单位）**

   em单位的名称为相对长度单位，它是用来设置文本的字体尺寸的，它是相对于当前对象内文本的字体尺寸；一般浏览器默认1em=16px，通过设置font-size大小来代表如：16px*0.625=10px，其则代表1em=10px。

   **em是相对于父级元素的单位，会随父级元素的属性（font-size或其它属性）变化而变化**

3. **rem（css3新增的相对长度单位）**

   rem是css3新增的一个相对长度单位，它的出现是为了解决em的缺点，em可以说是相对于父级元素的字体大小，当父级元素字体大小改变时，又得重新计算。rem出现就可以解决这样的问题，rem只相对于根目录，即HTML元素。所以只要在html标签上设置字体大小，文档中的字体大小都会以此为参照标准，一般用于自适应布局。

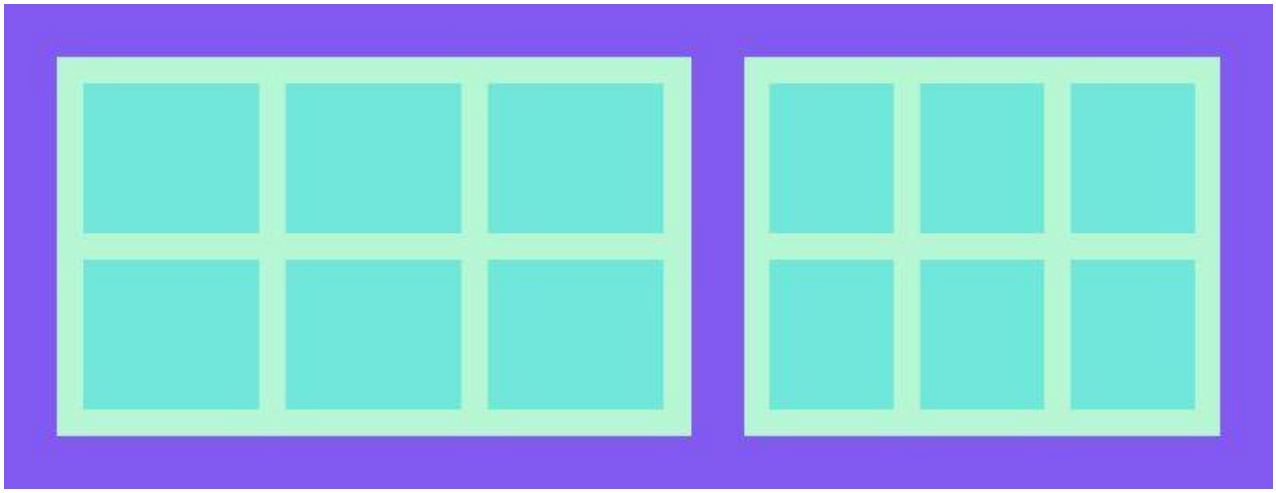
   **rem是相对于根目录（HTML元素）的，所有它会随HTML元素的属性（font-size）变化而变化**

***引出响应式布局、移动端布局***

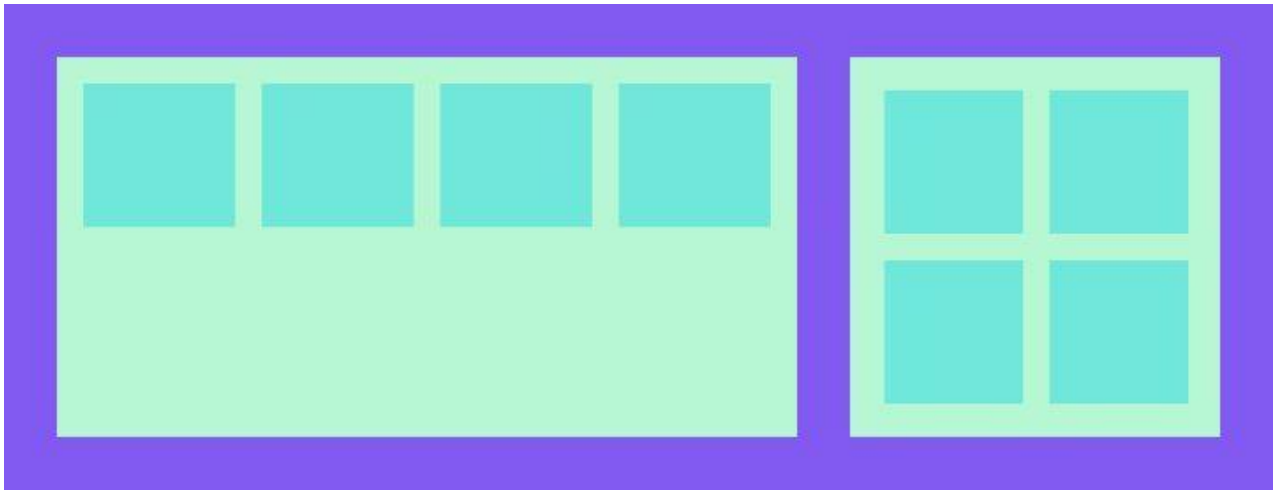
## 响应式布局基础 —— rem

**响应式布局准则**

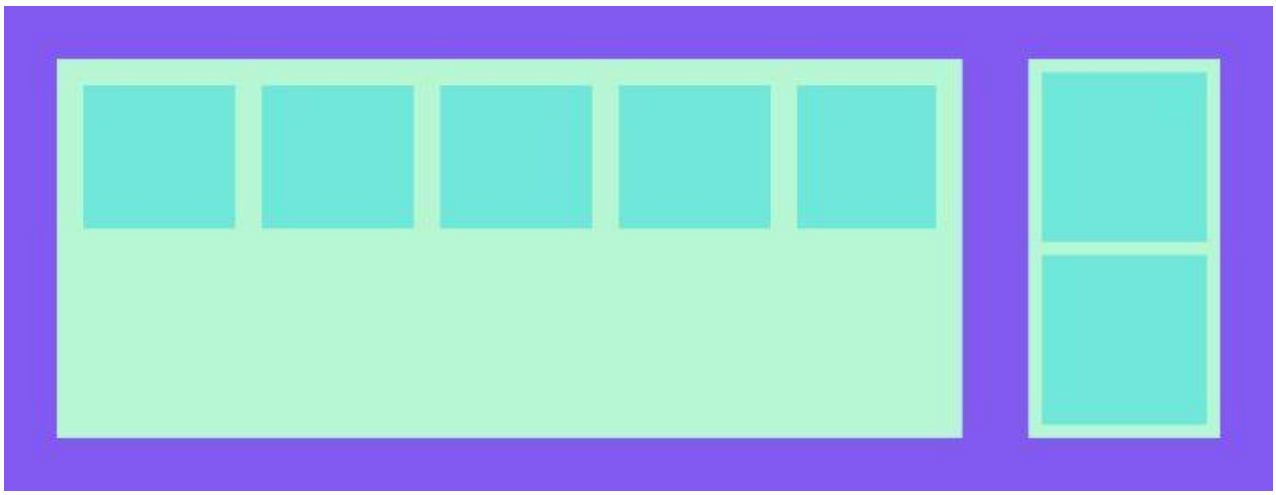
1 . **内容区块**可伸缩：内容区块的在一定程度上能够自动调整，以确保填满整个页面



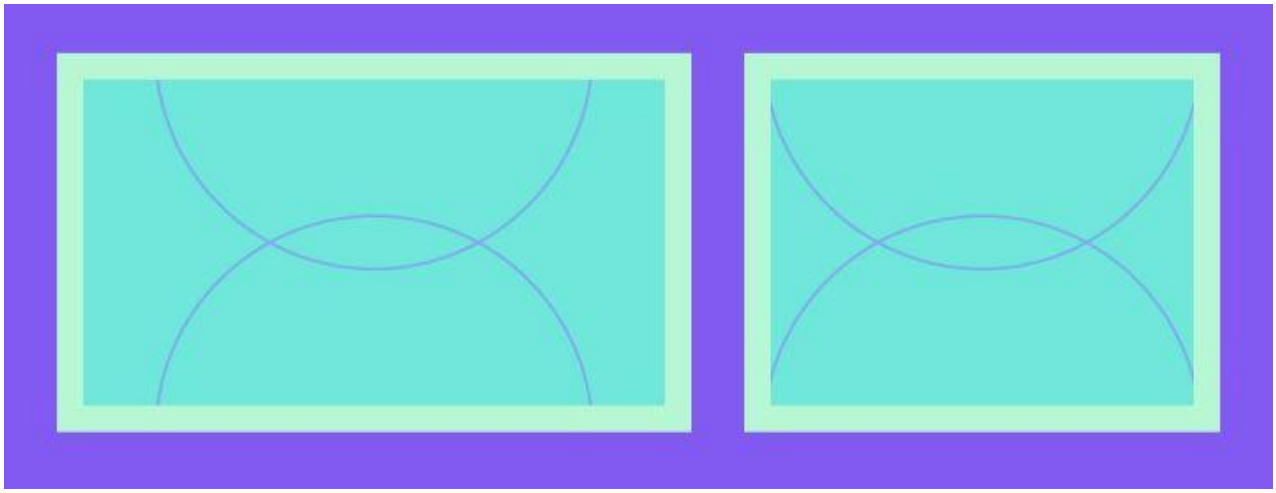
2 . **内容区块**可自由排布：当页面尺寸变动较大时，能够减少/增加排布的列数



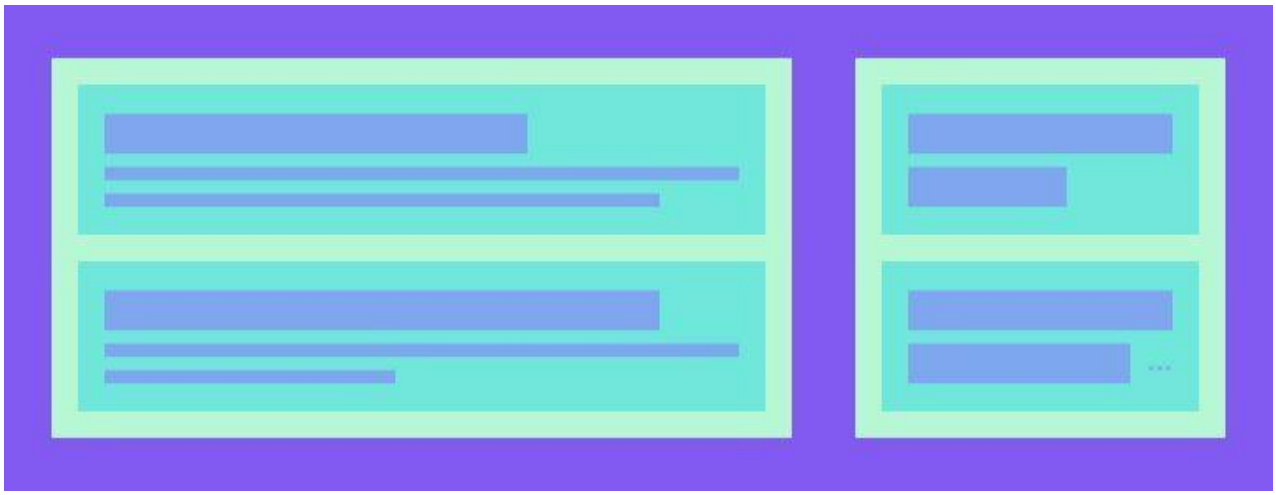
3 .**边距**适应：到页面尺寸发生更大变化时，区块的边距也应该变化



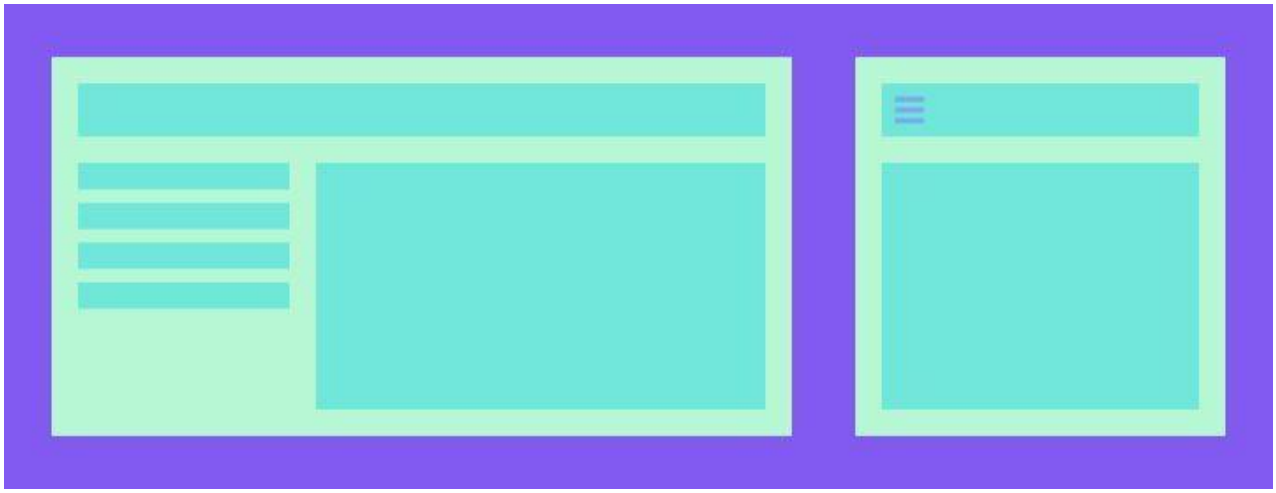
4 .**图片**适应：对于常见的宽度调整，图片在隐去两侧部分时，依旧保持美观可用



5 .**内容**能够自动隐藏/部分显示：如在电脑上显示的的大段描述文本，在手机上就只能少量显示或全部隐藏



6 .**导航和菜单**能自动折叠：展开还是收起，应该根据页面尺寸来判断



7 .**字体**比例缩放时，字体也保持缩放

**rem的应用**

> 选择性使用rem作为单位，这样在不同尺寸的设备上，通过修改根节点的`font-size`大小，实现等比例缩放

1 . 给根元素设置字体大小，并在body元素校正

```
html{font-size:100px;}
body{font-size:14px;}
```

如上设置后，使用rem代替px，直接除100即可

```
.menu li{
    display: table-cell;
    padding: .1rem .3rem;/*相当于10px 30px*/
}
```

2 . 绑定监听事件，dom加载后和尺寸变化时改变font-size

**注意把代码中的 `1536` 修改为实际开发时页面的宽度**

```
//改变font-size
(function(doc,win){
    var docEI = doc.documentElement,
    resizeEvt = 'orientationchange' in window?'orientataionchange':'resize',
    recalc = function(){
        var clientWidth = docEI.clientWidth;
        if(!clientWidth) return;
        //100是字体大小，1536是开发时浏览器窗口的宽度，等比计算
        docEI.style.fontSize = 100*(clientWidth/1536)+'px';
    }

    if(!doc.addEventListener) return;
    win.addEventListener(resizeEvt, recalc, false);
    doc.addEventListener('DOMContentLoaded', recalc, false);
})(document,window);
```

## 移动端自适应方案 —— flexible.js

> flexible.js由手机淘宝团队开发，现已停止维护，应该采用vw+vh+rem方案

在页面中引入flexible.js后，flexible会在标签上增加一个data-dpr属性和font-size样式，js首先会获取设备型号，然后根据不同设备添加不同的data-dpr值，比如说1、2或者3。

另外，页面中的元素用rem单位来设置，rem就是相对于根元素的font-size来计算的，flexible.js能根据页面元素的盒模型大小计算出适合的font-size。这样就意味着我们只需要在根元素确定一个px字号，因此来算出各元素的宽高，从而实现屏幕的适配效果。

### vw + vh + rem 响应式布局

> vw（Viewport Width）、vh(Viewport Height)是基于视图窗口的单位，是css3的一部分，基于视图窗口的单位，除了vw、vh还有vmin、vmax。
>
> - vw:1vw 等于视口宽度的1%

> - Vh:1vh 等于视口高度的1%
> - vmin: 选取 vw 和 vh 中最小的那个,即在手机竖屏时，1vmin=1vw
> - vmax:选取 vw 和 vh 中最大的那个 ,即在手机竖屏时，1vmax=1vh
>
> 通过 Media Queries 实现的布局需要配置多个响应断点，布局在响应断点范围内的分辨率下维持不变，而在响应断点切换的瞬间，布局带来断层式的切换变化，带来的体验也对用户十分的不友好。
>
> 采用rem单位的动态计算的弹性布局，则是需要在头部内嵌一段脚本来进行监听分辨率的变化来动态改变根元素字体大小，使得 CSS 与 JS 耦合了在一起。
>
> 通过利用视口单位实现适配的页面，是既能解决响应式断层问题，又能解决脚本依赖的问题的。

- **做法一：仅使用vw作为CSS单位：**

  在仅使用 vw 单位作为唯一应用的一种 CSS 单位的这种做法下，我们遵守：

1. 对于设计稿的尺寸转换为vw单位，我们使用Sass函数编译
2. 无论是文本还是布局高宽、间距等都使用 vw 作为 CSS 单位
3. 物理像素线（也就是普通屏幕下 1px ，高清屏幕下 0.5px 的情况）采用 transform 属性 scale 实现。
4. 对于需要保持高宽比的图，应改用 padding-top 实现

- **做法二：搭配vw和rem，布局更优化：**

  rem 弹性布局的核心在于动态改变根元素大小，结合rem单位来实现布局：

1. 给根元素大小设置随着视口变化而变化的 vw 单位，这样就可以实现动态改变其大小。
2. 限制根元素字体大小的最大最小值，配合 body 加上最大宽度和最小宽度

```
// rem 单位换算：定为 75px 只是方便运算，750px-75px、640-64px、1080px-108px，如此类推
$vm_fontsize: 75; // iPhone 6尺寸的根元素大小基准值
@function rem($px) {
     @return ($px / $vm_fontsize ) * 1rem;
}
// 根元素大小使用 vw 单位
$vm_design: 750;
html {
    font-size: ($vm_fontsize / ($vm_design / 2)) * 100vw;
    // 同时，通过Media Queries 限制根元素最大最小值
    @media screen and (max-width: 320px) {
        font-size: 64px;
    }
    @media screen and (min-width: 540px) {
        font-size: 108px;
    }
}
// body 也增加最大最小宽度限制，避免默认100%宽度的 block 元素跟随 body 而过大过小
body {
```

```
    max-width: 540px;
    min-width: 320px;
}
```

---

- ## 移动前端开发之viewport

1. **viewport的概念**

   通俗的讲，移动设备上的viewport就是设备的屏幕上能用来显示我们的网页的那一块区域，在具体一点，就是浏览器上(也可能是一个app中的webview)用来显示网页的那部分区域，但viewport又不局限于浏览器可视区域的大小，它可能比浏览器的可视区域要大，也可能比浏览器的可视区域要小。在默认情况下，一般来讲，移动设备上的viewport都是要大于浏览器可视区域的，这是因为考虑到移动设备的分辨率相对于桌面电脑来说都比较小，所以为了能在移动设备上正常显示那些传统的为桌面浏览器设计的网站，移动设备上的浏览器都会把自己默认的viewport设为980px或1024px，但带来的后果就是浏览器会出现横向滚动条，因为浏览器可视区域的宽度是比这个默认的viewport的宽度要小的。下图列出了一些设备上浏览器的默认viewport的宽度。

2. **css中的1px并不等于设备的1px**

   css中的像素只是一个抽象的单位，在不同的设备或不同的环境中，css中的1px所代表的设备物理像素是不同的。在为桌面浏览器设计的网页中，我们无需对这个津津计较，但在移动设备上，必须弄明白这点。在早先的移动设备中，屏幕像素密度都比较低，如iphone3，它的分辨率为320x480，在iphone3上，一个css像素确实是等于一个屏幕物理像素的。后来随着技术的发展，移动设备的屏幕像素密度越来越高，从iphone4开始，苹果公司便推出了所谓的Retina屏，分辨率提高了一倍，变成640x960，但屏幕尺寸却没变化，这就意味着同样大小的屏幕上，像素却多了一倍，这时，一个css像素是等于两个物理像素的。其他品牌的移动设备也是这个道理。例如安卓设备根据屏幕像素密度可分为ldpi、mdpi、hdpi、xhdpi等不同的等级，分辨率也是五花八门，安卓设备上的一个css像素相当于多少个屏幕物理像素，也因设备的不同而不同，没有一个定论。

   还有一个因素也会引起css中px的变化，那就是用户缩放。例如，当用户把页面放大一倍，那么css中1px所代表的物理像素也会增加一倍；反之把页面缩小一倍，css中1px所代表的物理像素也会减少一倍。关于这点，在文章后面的部分还会讲到。

   在移动端浏览器中以及某些桌面浏览器中，window对象有一个devicePixelRatio属性，它的官方的定义为：设备物理像素和设备独立像素的比例，也就是 devicePixelRatio = 物理像素 / 独立像素。css中的px就可以看做是设备的独立像素，所以通过devicePixelRatio，我们可以知道该设备上一个css像素代表多少个物理像素。

3. **移动设备的三个viewport**

   移动设备上的viewport分为***layout viewport*** 、 ***visual viewport*** 和 ***ideal viewport*** 三类，其中的ideal viewport是最适合移动设备的viewport，ideal viewport的宽度等于移动设备的屏幕宽度，只要在css中把某一元素的宽度设为ideal viewport的宽度(单位用px)，那么这个元素的宽度就是设备屏幕的宽度了，也就是宽度为100%的效果。ideal viewport 的意义在于，无论在何种分辨率的屏幕下，那些针对ideal viewport 而设计的网站，不需要用户手动缩放，也不需要出现横向滚动条，都可以完美的呈现给用户。

4. **利用meta标签对viewport进行控制**

```
<meta name="viewport" content="width=device-width, initial-scale=1.0,
maximum-scale=1.0, user-scalable=0">
```

该meta标签的作用是让当前viewport的宽度等于设备的宽度，同时不允许用户手动缩放。也许允不允许用户缩放不同的网站有不同的要求，但让viewport的宽度等于设备的宽度，这个应该是大家都想要的效果，如果你不这样的设定的话，那就会使用那个比屏幕宽的默认viewport，也就是说会出现横向滚动条。

---

## • css3 新特性

### 1. css3的新选择器

- `E:nth-child(n)` 选择器匹配其父元素的第n个子元素，不论元素类型，n可以使数字，关键字，或公式
- `E:nth-of-type(n)` 选择与之其匹配的父元素的第N个子元素
- `E:frist-child` 相对于父级做参考，“所有”子元素的第一个子元素，并且“位置”要对应
- `E:frist-of-type` 相对于父级做参考，“特定类型”（E）的第一个子元素
- `E:empty` 选择没有子元素的每个E元素
- `E:target` 选择当前活动的E元素
- `::selection` 选择被用户选取的元素部分
- 属性选择器

  `E[abc*="def"]` 选择adc属性值中包含子串"def"的所有元素

### 2. 文本

- `text-shadow:2px 2px 8px #000;`参数1为向右的偏移量，参数2为向左的偏移量，参数3为渐变的像素，参数4为渐变的颜色
- `text-overflow`：规定当文本溢出包含元素时发生的事情 text-overflow:ellipsis(省略号)
- `text-wrap`：规定文本换行的规则
- `word-break` 规定非中日韩文本的换行规则
- `word-wrap`：对长的不可分割的单词进行分割并换行到下一行
- `white-space`：规定如何处理元素中的空白 white-space:nowrap 规定段落中的文本不进行换行

### 3. 边框

- `border-raduis` 边框的圆角
- `border-image` 边框图片

```
.border-image {
    border-image-source:url(images/border.png);
    boder-image-slice:27;
    border-image-width:10px;
    border-iamge-repeat:round; /* (round平铺) 平铺效果不作用于四角，只适应与四边 */
}
```

#### 4. 背景

- `rgba`
- `backgrounnd-size:cover/contain`，其中background-size：cover，会使“最大”边进行缩放，另一边同比缩放，铺满容器，超出部分会溢出。background-size:contain，会使“最小”边进行缩放，另一边同比缩放，不一定铺满容器，会完整显示图片

#### 5. 渐变

- `linear-gradient`

```
background-image:linear-gradient(90deg,yellow 20%,green 80%)
```

- `radial-gradient`

```
background-iamge:radial-gradient(120px at center center,yellow,green)
```

#### 6. 多列布局

- `column-count`
- `column-width`
- `column-gap`
- `column-rule`

#### 7. 过渡

- `transition`
- `transition-property:width` //property为定义过渡的css属性列表，列表以逗号分隔
- `transition-duration:2s;` //过渡持续的时间
- `transition-timing-function:ease;`
- `transition-delay:5s` //过渡延迟5s进行

#### 8. 动画、旋转

- `animation`
- `transform ： translate (x,y) rotate(deg) scale(x,y)`
- `translate`
- `scale`
- `rotate`
- `skew` （倾斜）

#### 9. flex布局

**10. @media媒体查询**

# JavaScript

## • JS数据类型

### 基本数据类型

js 一共有六种基本数据类型，分别是 Undefined、Null、Boolean、Number、String，还有 在 ES6 中新增的 Symbol 类型，代表创建后独一无二且不可变的数据类型，它的出现我认为主要是为了解决可能出现的全局变量冲突的问题。

### 引用数据类型

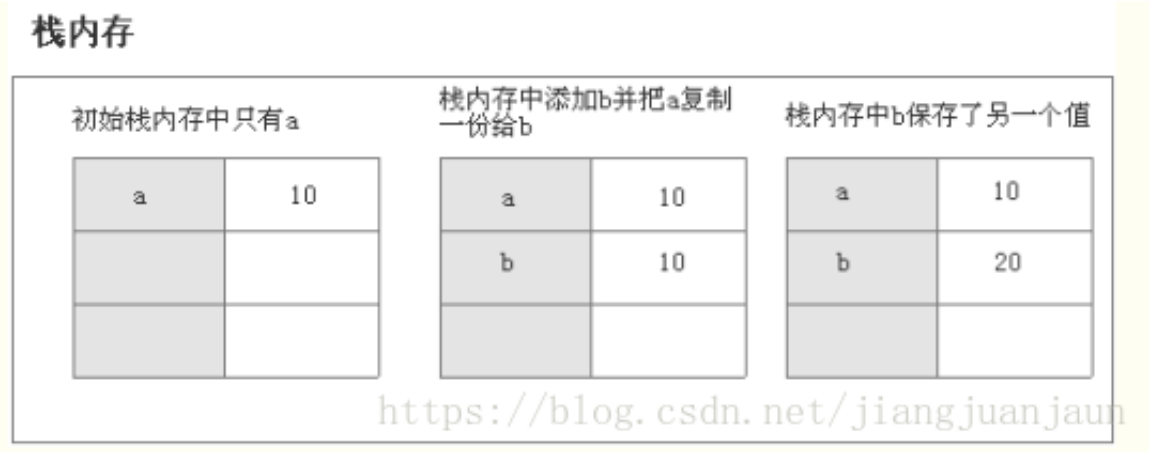
对象、数组、函数

### 两种数据类型的区别

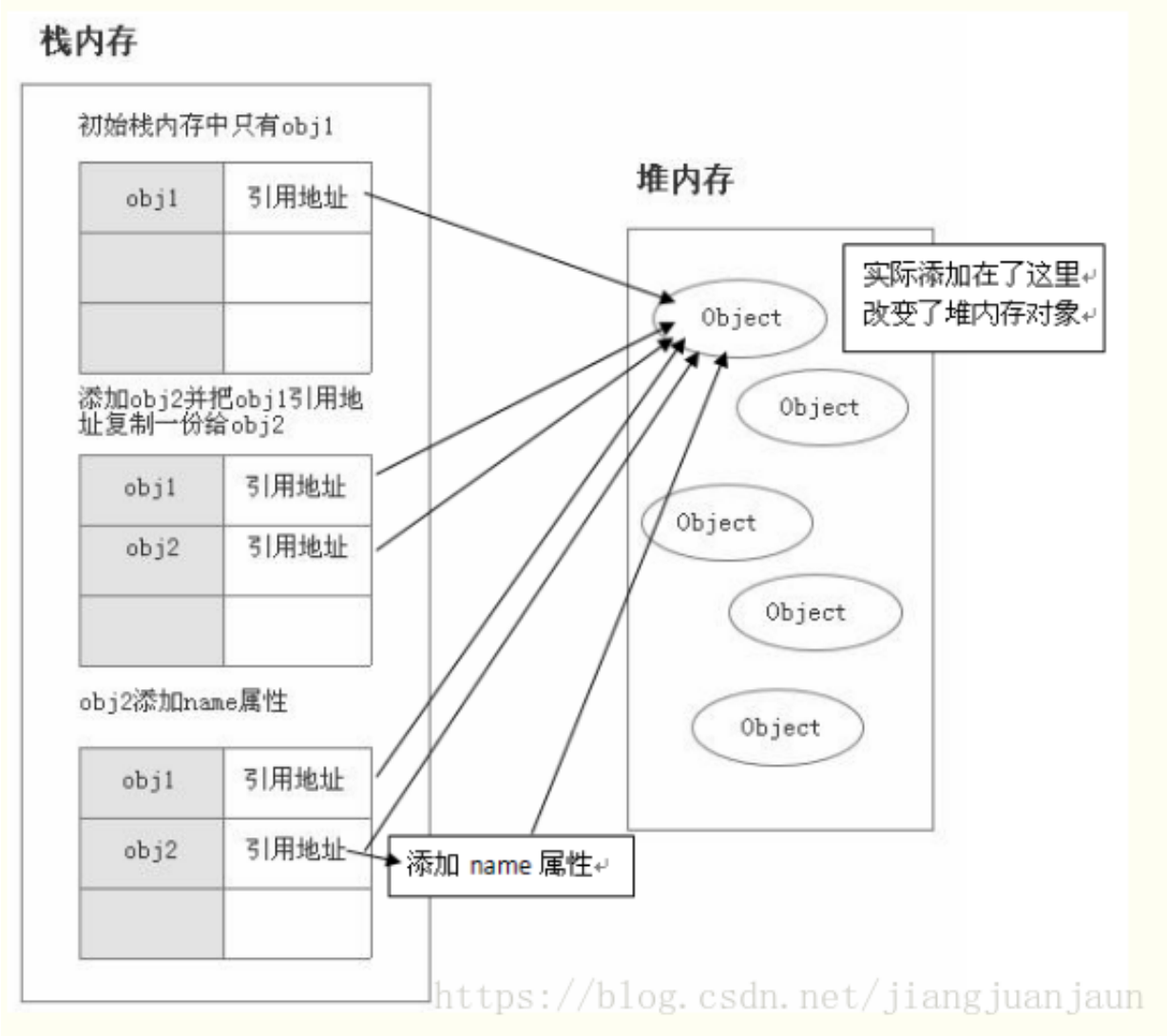
两种类型的区别是：存储位置不同。

基本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。




基本数据类型存储在栈




引用数据类型存储在堆

## Undefined 和 Null 的区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值， 就是 undefined 和 null。undefined 代表的含义是未定义（更准确来说是已声明未赋值），null 代表的含义是空对象。一 般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。undefined 在 js 中不是一个保留字，这意味着我们可以使用 undefined 来作为一个变量名，这样的做法是非常危险的，它会影响我们对 undefined 值的判断。但是我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。当我们对两种 类型使用 typeof 进行判断的时候，Null 类型化会返回 “object”，这是一个历史遗留的问题。 当我们使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

## 数据类型的判断

- **typeof**

> typeof 是一元操作符，放在其单个操作数的前面，操作数可以是任意类型。返回值为表示操作数类型的一个字符串。
>
> 那我们都知道，在 ES6 前，JavaScript 共六种数据类型，分别是：
>
> Undefined、**Null**、Boolean、Number、String、**Object**

> 当我们使用 typeof 对这些数据类型的值进行操作的时候，返回的结果却不是一一对应，分别是：
>
> undefined、**object**、boolean、number、string、**object**
>
> 注意以上都是小写的字符串。Null 和 Object 类型都返回了 object 字符串。
>
> 尽管不能一一对应，但是 typeof 却能检测出函数类型：

```
function a() {}
console.log(typeof a); // function
```

> 所以 typeof 能检测出六种类型的值，但是，除此之外 Object 下还有很多细分的类型呐，如 Array、Function、Date、RegExp、Error 等... ...
>
> 解决办法：**Object.prototype.toString**
>
> 调用 Object.prototype.toString 会返回一个由 "[object " 和 class 和 "]" 组成的字符串，而 class 是要判断的对象的内部属性

```
console.log(Object.prototype.toString.call(undefined)) // [object
Undefined]
console.log(Object.prototype.toString.call(null)) // [object Null]

var date = new Date();
console.log(Object.prototype.toString.call(date)) // [object Date]
```

> 由此我们可以看到这个 class 值就是识别对象类型的关键！

```
function getType(value) {
    // 判断数据是 null 的情况
    if (value === null) {
        return value + "";
    }
    // 判断数据是引用类型的情况
    if (typeof value === "object") {
        let valueClass = Object.prototype.toString.call(value),
            type = valueClass.split(" ")[1].split("");
        type.pop();
        return type.join("").toLowerCase();
        // return valueClass.slice(8, -1).toLowerCase();
    } else {
        // 判断数据是基本数据类型的情况和函数的情况
        return typeof value;
    }
}
```

- **instanceof**

> 判断该对象是谁的实例，同时我们也就知道instanceof是对象运算符。这里的实例就牵扯到了对象的继承，它的判断就是根据原型链进行搜寻，在对象obj1的原型链上如果存在另一个对象obj2的原型属性，那么表达式（obj1 instanceof obj2）返回值为true；否则返回false。

typeof和instanceof都是用来判断变量类型的，两者的区别在于：

- typeof判断所有变量的类型，返回值有number，boolean，string，function，object，undefined。
- typeof对于丰富的对象实例，只能返回"Object"字符串。
- instanceof用来判断对象，代码形式为obj1 instanceof obj2（obj1是否是obj2的实例），obj2必须为对象，否则会报错！其返回值为布尔值。
- instanceof可以对不同的对象实例进行判断，判断方法是根据对象的原型链依次向下查询，如果obj2的原型属性存在obj1的原型链上，（obj1 instanceof obj2）值为true。

## 数据类型的转换

- **其他类型到字符串的转换规则**

  规范的 9.8 节中定义了抽象操作 ToString ，它负责处理非字符串到字符串的强制类型转换。

  1. Null 和 Undefined 类型 ，null 转换为 "null"，undefined 转换为 "undefined"，
  2. Boolean 类型，true 转换为 "true"，false 转换为 "false"。
  3. Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。
  4. Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。
  5. 对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString()（Object.prototype.toString()）来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。
- **其他类型到数字类型的转换规则**

  有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义 了抽象操作 ToNumber。

  1. Undefined 类型的值转换为 NaN。
  2. Null 类型的值转换为 0。
  3. Boolean 类型的值，true 转换为 1，false 转换为 0。
  4. String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。
  5. Symbol 类型的值不能转换为数字，会报错。
  6. 对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值， 则再遵循以上规则将其强制转换为数字。为了将值转换为相应的基本类型值，抽象操作ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有 valueOf() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。
- **其他类型到布尔类型的转换规则**

  以下这些是假值：

- undefined
- null
- false
- +0、-0 和 NaN
- ""

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

## js中的假值

ToBoolean() —— 数据类型转换

### 7.1.2 ToBoolean ( argument )

The abstract operation ToBoolean converts *argument* to a value of type Boolean according to Table 10:

**Table 10 — ToBoolean Conversions**

| Argument Type | Result |
|---|---|
| Completion Record | If *argument* is an abrupt completion, return *argument*. Otherwise return ToBoolean(*argument*.[[value]]). |
| Undefined | Return **false**. |
| Null | Return **false**. |
| Boolean | Return *argument*. |
| Number | Return **false** if *argument* is **+0**, **−0**, or **NaN**; otherwise return **true**. |
| String | Return **false** if *argument* is the empty String (its length is zero); otherwise return **true**. |
| Symbol | Return **true**. |
| Object | Return **true**. |

直接了当的可以看到，undefined、null直接返回false； Symbol、Object永远返回true；本身就是Boolean型直接返回原本值；Number型的除+0、-0和NaN返回false以外，其余都返回true；String型中空字符串(即长度为0)返回false，其他情况返回true。

总结一下自身进行判断(toBoolean)，假值的有6种情况：null、undefined、""（空字符串）、+0、-0、NaN。

## Array数组常用方法

## String字符串常用方法

## this指向

> this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模式来判断。

1. 第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。
2. 第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。
3. 第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。
4. 第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入 一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指 向除了使用 new 时会被改变，其他情况下都不会改变。

> 这四种方式，使用构造器调用模式的优先级最高，然后是 apply 、call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

---

## 箭头函数与普通函数的区别

***ES6 系列之箭头函数***

1. this 指向不同：

- 普通函数 this 指向为方法调用的对象，可以通过bind，call，apply，改变this指向
- 箭头函数比函数表达式更简洁，箭头函数不会创建自己的this,它只会从自己的作用域链的上一层继承this。bind，call，apply只能调用传递参数，不可修改this指向

```
var obj = {
  a: 10,
  b: () => {
    console.log(this.a); // undefined
    console.log(this); // Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ,
frames: Window, …}
  },
  c: function() {
    console.log(this.a); // 10
    console.log(this); // {a: 10, b: ƒ, c: ƒ}
  }
}
obj.b();
obj.c();
```

> 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。 任何方法都改变不了其指向

```
var obj = {
  a: 10,
  b: function(){
    console.log(this.a); // 10
  },
  c: function() {
     return ()=>{
           console.log(this.a); // 10
     }
  }
}
obj.b();
obj.c()();
```

> 箭头函数通过 call() 或 apply() 方法调用一个函数时，只传入了一个参数，对 this 并没有影响。
>
> 补充：call,aplly,bind: 它们在功能上是没有区别的，都是改变this的指向，它们的区别主要是在于方法的实现形式和参数传递上的不同。call和apply方法都是在调用之后立即执行的。而bind调用之后是一个函数，需要再调用一次才行
>
> ①：函数.call(对象,arg1,arg2....) ②：函数.apply(对象，[arg1,arg2,...]) ③：var ss=函数.bind(对象,arg1,arg2,....)

```
let obj2 = {
    a: 10,
    b: function(n) {
        let f = (n) => n + this.a;
        return f(n);
    },
    c: function(n) {
        let f = (n) => n + this.a;
        let m = {
            a: 20
        };
        return f.call(m,n);
    }
};
console.log(obj2.b(1));  // 11
console.log(obj2.c(1)); // 11
```

2. 箭头函数没有原型,

```
var a = ()=>{
  return 1;
}

function b(){
  return 2;
}

console.log(a.prototype);  // undefined
console.log(b.prototype);   // {constructor: ƒ}
```

3. 箭头函数不能绑定arguments，取而代之用rest参数...解决

```
function A(a){
  console.log(arguments);
}
A(1,2,3,4,5,8);
// [1, 2, 3, 4, 5, 8, callee: ƒ, Symbol(Symbol.iterator): ƒ]
let C = (...c) => {
  console.log(c);
}
C(3,82,32,11323);
// [3, 82, 32, 11323]
```

4. 箭头函数是匿名函数，不能作为构造函数，不能使用new

> 无法实例化的原因： 没有自己的this，无法调用call，apply 没有prototype属性，而new命令执行的时候需要将构造函数的prototype赋值给新的对象的**_proto**

5. 箭头函数不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。
6. 箭头函数体内的this对象（继承的），就是定义时所在的对象，而不是使用时所在的对象。

---

## • 任务队列，事件循环，微任务，宏任务

JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

因为 js 是单线程运行的，在代码执行的时候，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。在执行同步代码的时候，如果遇到了异步事件，js 引擎并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务。当异步事件执行完毕后，再将异步事件对应的回调加入到与当前执行栈中不同的另一个任务队列中等待执行。任务队列可以分为宏任务对列和微任务对列，当当前执行栈中的事件执行完毕后，js 引擎首先会判断微任务对列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。当微任务对列中的任务都执行完成后再去判断宏任务对列中的任

务。

微任务包括了 promise 的回调、node 中的 process.nextTick 、对 Dom 变化监听的 MutationObserver。

宏任务包括了 script 脚本执行，setTimeout ，setInterval 一类的定时事件，ajax回调函数，还有如 I/O 操作、UI 渲染等。

### 为什么setTimeout()不准时

因为 JavaScript 是一个单线程序的解释器，因此一定时间内只能执行一段代码。为了控制要执行的代码，就有一个 JavaScript 任务队列。这些任务会按照将它们添加到队列的顺序执行。setTimeout() 的第二个参数告诉 JavaScript 再过多长时间把当前任务添加到队列中。如果队列是空的，那么添加的代码会立即执行；如果队列不是空的，那么它就要等前面的代码执行完了以后再执行。

---

## ● 事件模型, 事件

HTML中与javascript交互是通过事件驱动来实现的，例如鼠标点击事件onclick、页面的滚动事件onscroll等等，可以向文档或者文档中的元素添加事件侦听器来预订事件，现代浏览器一共有三种事件模型。

事件流，就是用户触发事件后，事件的传递，称之为事件流。

第一种事件模型是最早的 DOM0 级模型，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，它可以在网页中直接定义监听函数，也可以通过 js 属性来指定监听函数。这种方式是所有浏览器都兼容的。

第二种事件模型是 IE 事件模型，在该事件模型中，一次事件共有两个过程，事件处理阶段，和事件冒泡阶段。事件处理阶段会首先执行目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。

第三种是 **DOM2** 级事件模型，在该事件模型中，一次事件共有三个过程，第一个过程是事件捕获阶段。捕获指的是事件从 document 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，事件绑定的函数是 addEventListener，其中第三个参数是布尔值参数如果是true，表示在捕获阶段调用事件处理程序；如果是false，表示在冒泡阶段调用事件处理程序。

---

## ● 事件委托

事件委托本质上是利用了浏览器事件冒泡的机制。因为事件在冒泡过程中会上传到父节点， 并且父节点可以通过事件对象获取到目标节点，因此可以把子节点的监听函数定义在父节点上， 由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件代理。使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的监听函数来处理。

---

## • 异步编程

[深入掌握 ECMAScript 6 异步编程]

> 异步与同步：
>
> 同步指的是当一个进程在执行某个请求的时候，如果这个请求需要等待一段时间才能返回，那么这个进程会一直等待下去，直到消息返回为止再继续向下执行。
>
> 异步指的是当一个进程在执行某个请求的时候，如果这个请求需要等待一段时间才能返回，这个时候进程会继续往下执行，不会阻塞等待消息的返回，当消息返回时系统再通知进程进行处理。
>
> 所谓"异步"，简单说就是一个任务分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。比如，有一个任务是读取文件进行处理，异步的执行过程就是下面这样。
>
> 
> 
>
> 上图中，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。
>
> **这种不连续的执行，就叫做异步。**相应地，连续的执行，就叫做同步。
>
> 
> 
>
> 上图就是同步的执行方式。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。

js 中的异步机制可以分为以下几种：

第一种最常见的是使用回调函数的方式，所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成**回调函数地狱**，代码不是纵向发展，而是横向发展，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

第二种是 Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

第三种是使用 generator 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部我们还可以将执行权转移回来。当我们遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕的时候我们再将执行权给转移回来。因此我们在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式我们需要考虑的问题是何时将函数的控制权转移回来，因此我们需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

第四种是使用 async 函数的形式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此我们可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

理解 JavaScript 的 async/await

*TODO 代码执行顺序题*

---

# • js 延迟加载

js 延迟加载，也就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提 高页面加载速度。

一般有以下几种方式:

- defer 属性
- async 属性
- 动态创建 DOM 方式
- 使用 setTimeout 延迟方法
- 让 JS 最后加载

js 的加载、解析和执行会阻塞页面的渲染过程，因此我们希望 js 脚本能够尽可能的延迟 加载，提高页面的渲染速度。我了解到的几种方式是:

- 第一种方式是我们一般采用的是将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。
- 第二种方式是给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析**同步解析**，然后在**文档解析完成后再执行这个脚本文件***，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是**顺序执行**的，但是在一些浏览器中可能不是这样。
- 第三种方式是给 js 脚本添加 async 属性，这个属性会使脚本**异步加载**，不会阻塞页面的解析过程，但是当脚本加载完成后**立即执行 js 脚本**，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。
- 第四种方式是动态创建 DOM 标签的方式，我们可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。

---

# • get，post 区别

Post 和 Get 是 HTTP 请求的两种方法。

1. get参数通过url传递，post放在request body中。
2. get请求在url中传递的参数是有长度限制的，而post没有。
3. get比post更不安全，因为参数直接暴露在url中，所以不能用来传递敏感信息。

4. get请求只能进行url编码，而post支持多种编码方式
5. get请求会浏览器主动cache，而post支持多种编码方式。
6. get请求参数会被完整保留在浏览历史记录里，而post中的参数不会被保留。
7. GET和POST本质上就是TCP链接，并无差别。但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。
8. GET产生一个TCP数据包；POST产生两个TCP数据包。

---

## • Ajax

我对 ajax 的理解是，它是一种异步通信的方法，通过直接由 js 脚本向服务器发起 http 通信，然后根据服务器返回的数据，更新网页的相应部分，而不用刷新整个页面的一种方法。创建一个 ajax 有这样几个步骤：

首先是创建一个 XMLHttpRequest 对象。然后在这个对象上使用 open 方法创建一个 http 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。 在发起请求前，我们可以为这个对象添加一些信息和监听函数。比如说我们可以通过 setRequestHeader 方法来为请求添加头信息。我们还可以为这个对象添加一个状态监听函数。

一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 onreadystatechange 事件，我们可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候我们可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候我们就可以通过 response 中的数据来对页面进行更新了。当对象的属性和监听函数设置完成后，最后我们调用 sent 方法来向服务器发起请求，可以传入参数作为发送的数据体。

**readyState 5个状态**

| 状态 | 名称 | 描述 |
| --- | --- | --- |
| 0 | Uninitialized | 初始化状态。XMLHttpRequest 对象已创建或已被 abort() 方法重置。 |
| 1 | Open | open() 方法已调用，但是 send() 方法未调用。请求还没有被发送。 |
| 2 | Sent | Send() 方法已调用，HTTP 请求已发送到 Web 服务器。未接收到响应。 |
| 3 | Receiving | 所有响应头部都已经接收到。响应体开始接收但未完成。 |
| 4 | Loaded | HTTP 响应已经完全接收。 |

一般实现：

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", SERVER_URL, true);
xhr.send(data);
// 设置状态监听函数
xhr.onreadystatechange = function () {
  if (this.readyState !== 4) return;
  // 当请求成功时
```

```
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};

// 将原生的ajax封装成promise
var myNewAjax = function(url){
  return new Promise(function(resolve,reject){
    var xhr = new XMLHttpRequest();
    xhr.open('get',url);
    xhr.send(data);
    xhr.onreadystatechange=function(){
      if(xhr.status==200&&readyState==4){
        var json=JSON.parse(xhr.responseText);
        resolve(json)
      }else if(xhr.readyState==4&&xhr.status!=200){
        reject('error');
      }
    }
  })
}
```

---

## • Promise，Promise/A+ 规范

> 地狱回调：指由于在异步请求中大量地嵌套了回调函数导致代码结构复杂的现象

Promise 对象是异步编程的一种解决方案。Promises/A+ 规范是 JavaScript Promise 的标准，规定了一个 Promise 所必须具有的特性。

Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由 pending 转变 resolved 或者 rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。状态的改变是通过 resolve() 和 reject() 函数来实现的， 我们可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

**手写Promise**:

- 简单版Promise

首先我们应该知道Promise是通过构造函数的方式来创建的(new Promise( executor ))，并且为 executor 函数传递参数:

```
function Promi(executor) {
  executor(resolve, reject);
  function resolve() {}
  function reject() {}
}
```

再来说一下Promise的三种状态: pending-等待, resolve-成功, reject-失败, 其中最开始为pending状态, 并且一旦成功或者失败, Promise的状态便不会再改变,所以根据这点:

```
function Promi(executor) {
  let _this = this;
  _this.$$status = 'pending';
  executor(resolve.bind(this), reject.bind(this));
  function resolve() {
    if (_this.$$status === 'pending') {
      _this.$$status = 'full'
    }
  }
  function reject() {
    if (_this.$$status === 'pending') {
      _this.$$status = 'fail'
    }
  }
}
```

其中$$status来记录Promise的状态,只有当promise的状态为pending时我们才会改变它的状态为'full'或者'fail', 因为我们在两个status函数中使用了this,显然使用的是Promise的一些属性,所以我们要绑定resolve与reject中的this为当前创建的Promise; 这样我们最最最基础的Promise就完成了(只有头部没有四肢...)

- Promise高级 --> .then

接着,所有的Promise实例都可以用.then方法,其中.then的两个参数,成功的回调和失败的回调也就是我们所说的resolve和reject:

```
function Promi(executor) {
    let _this = this;
  _this.$$status = 'pending';
  _this.failCallBack = undefined;
  _this.successCallback = undefined;
  _this.error = undefined;
  executor(resolve.bind(_this), reject.bind(_this));
  function resolve(params) {
    if (_this.$$status === 'pending') {
      _this.$$status = 'success'
      _this.successCallback(params)
    }
  }
```

```
  function reject(params) {
    if (_this.$$status === 'pending') {
      _this.$$status = 'fail'
      _this.failCallBack(params)
    }
  }
}

Promi.prototype.then = function(full, fail) {
  this.successCallback = full
  this.failCallBack = fail
};
Promise.prototype.catch = function(fn){
    return this.then(null,fn);
}

// 测试代码
new Promi(function(res, rej) {
  setTimeout(_ => res('成功'), 30)
}).then(res => console.log(res))
```

讲一下这里： 可以看到我们增加了failCallBack和successCallback，用来储存我们在then中回调,刚才也说过,then中可传递一个成功和一个失败的回调,当P的状态变为resolve时执行成功回调,当P的状态变为reject或者出错时则执行失败的回调,但是具体执行结果的控制权没有在这里。但是我们知道一定会调用其中的一个。

executor任务成功了肯定有成功后的结果，失败了我们肯定也拿到失败的原因。所以我们可以通过params来传递这个结果或者error reason（当然这里的params也可以拆开赋给Promise实例）其实写到这里如果是面试题,基本上是通过了,也不会有人让你去完整地去实现

**error**:用来存储，传递reject信息以及错误信息

- Promise.all

  Promise.all可以将多个Promise实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。

  具体代码如下：

  ```
  let p1 = new Promise((resolve, reject) => {
    resolve('成功了')
  })

  let p2 = new Promise((resolve, reject) => {
    resolve('success')
  })

  let p3 = Promse.reject('失败')
  ```

```
Promise.all([p1, p2]).then((result) => {
  console.log(result)               //['成功了', 'success']
}).catch((error) => {
  console.log(error)
})

Promise.all([p1,p3,p2]).then((result) => {
  console.log(result)
}).catch((error) => {
  console.log(error)      // 失败了，打出 '失败'
})
```

Promse.all在处理多个异步处理时非常有用，比如说一个页面上需要等两个或多个ajax的数据回来以后才正常显示，在此之前只显示loading图标。

代码模拟：

```
let wake = (time) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`${time / 1000}秒后醒来`)
    }, time)
  })
}

let p1 = wake(3000)
let p2 = wake(2000)

Promise.all([p1, p2]).then((result) => {
  console.log(result)       // [ '3秒后醒来', '2秒后醒来' ]
}).catch((error) => {
  console.log(error)
})
```

需要特别注意的是，Promise.all获得的成功结果的数组里面的数据顺序和Promise.all接收到的数组顺序是一致的，即p1的结果在前，即便p1的结果获取的比p2要晚。这带来了一个绝大的好处：在前端开发请求数据的过程中，偶尔会遇到发送多个请求并根据请求顺序获取和使用数据的场景，使用Promise.all毫无疑问可以解决这个问题。

- Promise.race

  顾名思义，Promse.race就是赛跑的意思，意思就是说，Promise.race([p1, p2, p3])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。

```
let p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('success')
  },1000)
})
```

```
let p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('failed')
  }, 500)
})

Promise.race([p1, p2]).then((result) => {
  console.log(result)
}).catch((error) => {
  console.log(error)  // 打开的是 'failed'
})
```

- Promise.retry

  实现函数myGetData，也返回一个promise，但是有失败重试功能

```
function myGetData(getData, times, delay) {//retry函数
    return new Promise(function (resolve, reject) {
        function attempt() {
            getData().then(resolve).catch(function (erro) {
                console.log(`还有 ${times} 次尝试`)
                if (0 == times) {
                    reject(erro)
                } else {
                    times--
                    setTimeout(attempt(), delay)
                }
            })
        }
        attempt()
    })
}
```

---

# • Generator 函数

- Generator函数的概念

Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。

```
function* gen(x){
var y = yield x + 2;
return y;
}
```

上面代码就是一个 Generator 函数。它不同于普通函数，是可以暂停执行的，所以函数名之前要加星号，以示区别。

整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用 yield 语句注明。Generator 函数的执行方法如下。

```
var g = gen(1);
g.next() // { value: 3, done: false }
g.next() // { value: undefined, done: true }
```

上面代码中，调用 Generator 函数，会返回一个内部指针（即遍历器 ）g 。这是 Generator 函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针 g 的 next 方法，会移动内部指针（即执行异步任务的第一段），指向第一个遇到的 yield 语句，上例是执行到 x + 2 为止。

换言之，next 方法的作用是分阶段执行 Generator 函数。每次调用 next 方法，会返回一个对象，表示当前阶段的信息（ value 属性和 done 属性）。value 属性是 yield 语句后面表达式的值，表示当前阶段的值；done 属性是一个布尔值，表示 Generator 函数是否执行完毕，即是否还有下一个阶段。

- Generator 函数的数据交换和错误处理

Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。

next 方法返回值的 value 属性，是 Generator 函数向外输出数据；next 方法还可以接受参数，这是向 Generator 函数体内输入数据。

```
function* gen(x){
var y = yield x + 2;
return y;
}

var g = gen(1);
g.next() // { value: 3, done: false }
g.next(2) // { value: 2, done: true }
```

上面代码中，第一个 next 方法的 value 属性，返回表达式 x + 2 的值（3）。第二个 next 方法带有参数2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量 y 接收。因此，这一步的 value 属性，返回的就是2（变量 y 的值）。

Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。

```
function* gen(x){
try {
var y = yield x + 2;
} catch (e){
console.log(e);
}
return y;
}

var g = gen(1);
g.next();
g.throw ('出错了') ;
// 出错了
```

上面代码的最后一行，Generator 函数体外，使用指针对象的 throw 方法抛出的错误，可以被函数体内的 try ... catch 代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的。

- Generator 函数的用法

下面看看如何使用 Generator 函数，执行一个真实的异步任务。

```
var fetch = require('node-fetch');

function* gen(){
var url = 'https://api.github.com/users/github';
var result = yield fetch(url);
console.log(result.bio);
}
```

上面代码中，Generator 函数封装了一个异步操作，该操作先读取一个远程接口，然后从 JSON 格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了 yield 命令。

执行这段代码的方法如下。

```
var g = gen();
var result = g.next();

result.value.then(function(data){
return data.json();
}).then(function(data){
g.next(data);
});
```

上面代码中，首先执行 Generator 函数，获取遍历器对象，然后使用 next 方法（第二行），执行异步任务的第一阶段。由于 Fetch 模块返回的是一个 Promise 对象，因此要用 then 方法调用下一个 next 方法。

可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便（即何时执行第一阶段、何时执行第二阶段）。

## • async，await 函数

**一句话，async 函数就是 Generator 函数的语法糖。一比较就会发现，async 函数就是将 Generator 函数的星号（*）替换成 async，将 yield 替换成 await，仅此而已。**

async 函数对 Generator 函数的改进，体现在以下三点。

**（1）内置执行器。** Generator 函数的执行必须靠执行器，所以才有了 co 函数库，而 async 函数自带执行器。也就是说，async 函数的执行，与普通函数一模一样，只要一行。

```
var result = asyncReadFile();
```

**（2）更好的语义。** async 和 await，比起星号和 yield，语义更清楚了。async 表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。

**（3）更广的适用性。** co 函数库约定，yield 命令后面只能是 Thunk 函数或 Promise 对象，而 async 函数的 await 命令后面，可以跟 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

## • 深拷贝，浅拷贝

浅拷贝指的是将一个对象的属性值复制到另一个对象，如果有的属性的值为引用类型的话，那么会将这个引用的地址复制给对象，因此两个对象会有同一个引用类型的引用。浅拷贝可以使用 Object.assign 和展开运算符来实现。

> `Object.assign`方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。
>
> ```
> const target = { a: 1 };
> const source1 = { b: 2 };
> const source2 = { c: 3 };
>
> Object.assign(target, source1, source2);
> target // {a:1, b:2, c:3}
> ```
>
> `Object.assign`方法的第一个参数是目标对象，后面的参数都是源对象。
>
> `Object.assign`常见用途：
>
> - 为对象添加属性
> - 为对象添加方法
> - 克隆对象 `Object.assign({}, origin);`
> - 合并多个对象
> - 为属性指定默认值 （同名覆盖）

深拷贝相对浅拷贝而言，如果遇到属性值为引用类型的时候，它新建一个引用类型并将对应的值复制给它，因此对象获得的一个新的引用类型而不是一个原有类型的引用。深拷贝对于一些对象可以使用 JSON 的两个函数来实现，但是由于 JSON 的对象格式比 js 的对象格式更加严格， 所以如果属性值里边出现函数或者 Symbol 类型的值时，会转换失败。

```
// 浅拷贝的实现;
function shallowCopy(object) {
  // 只拷贝对象
  if (!object || typeof object !== "object") return object;
  // 根据 object 的类型判断是新建一个数组还是对象
  let newObject = Array.isArray(object) ? [] : {};
  // 遍历 object，并且判断是 object 的属性才拷贝
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] = object[key];
    }
  }
  return newObject;
}
// 简单的深拷贝的实现（object类型只支持数组和函数）;
function deepCopy(object) {
  if (!object || typeof object !== "object") return object;
  let newObject = Array.isArray(object) ? [] : {};
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] =
        typeof object[key] === "object" ? deepCopy(object[key]) : object[key];
    }
  }
  return newObject;
}

// 完整的深拷贝实现。
//支持 String,Number,Boolean,null,undefined,Object,Array,Date,RegExp,Error 类型
/**
 * 变量类型判断
 * @param {String} type - 需要判断的类型
 * @param {Any} value - 需要判断的值
 * @returns {Boolean} - 是否该类型
 */
const IsType = (type, value) => Object.prototype.toString.call(value) ===
`[object ${type}]`;
/**
 * 深拷贝变量-递归算法(recursive algorithm)
 * 支持 String,Number,Boolean,null,undefined,Object,Array,Date,RegExp,Error 类型
 * @param {Any} arg - 需要深拷贝的变量
 * @returns {Any} - 拷贝完成的值
 */
```

```
const DeepCopyRA = arg => {
  const newValue = IsType("Object", arg) // 判断是否是对象
    ? {}
    : IsType("Array", arg) // 判断是否是数组
    ? [] // 三元表达式嵌套
    : IsType("Date", arg) // 判断是否是日期对象
    ? new arg.constructor(+arg)
    : IsType("RegExp", arg) || IsType("Error", arg) // 判断是否是正则对象或错误对象
    ? new arg.constructor(arg)
    : arg;
  // 判断是否是数组或对象
  if (IsType("Object", arg) || IsType("Array", arg)) {
    // 循环遍历
    for (const key in arg) {
      // 判断是否为自身的属性，防止原型链的值
      Object.prototype.hasOwnProperty.call(arg, key) && (newValue[key] =
DeepCopyRA(arg[key]));
      /**
      * 逻辑与运算的应用 http://c.biancheng.net/view/5452.html
      * 逻辑与是一种短路逻辑，如果左侧表达式为 false，则直接短路返回结果，不再运算右侧表达
式。
      * 运算逻辑如下：
      * 第 1 步：计算第一个操作数（左侧表达式）的值。
      * 第 2 步：检测第一个操作数的值。如果左侧表达式的值可转换为 false（如 null、
undefined、NaN、0、""、false），那么就会结束运算，直接返回第一个操作数的值。
      * 第 3 步：如果第一个操作数可以转换为 true，则计算第二个操作数（右侧表达式）的值。
      * 第 4 步：返回第二个操作数的值。
      */
    }
  }
  return newValue;
};
```

## • 闭包

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是在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包有两个常用的用途。闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，我们可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。 闭包的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

```
// 使用闭包实现隔一秒打印1，2，3，4
for (var i = 0; i < 5; i++) {
  (function(i) {
     setTimeout(function() {
```

```
        console.log(i);
      }, i * 1000);
  })(i);
}

// 用let块级作用域
for(let i=0;i<5;i++){
  setTimeout(function(){
    console.log(i)
  }, 1000 * i)
}

// 请手写一个函数，这个函数名字是a，使得a被执行之后有如下效果。
// a(); // 函数返回值为1
// a(); // 函数返回值为2
// a(); // 函数返回值为3
// ...以此类推，每调用一次a函数，函数返回
var a = (function(){ // 使用闭包保存变量，使用立即执行函数返回+1函数
    var count = 0
    return function(){
            return ++count;
        }
})()

console.log(a()) // 1
console.log(a()) // 2
console.log(a()) // 3
```

---

## • 作用域链

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，我们可以访问到外层环境的变量和函数。 作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。 当我们查找一个变量时，如果当前执行环境中没有找到，我们可以沿着作用域链向后查找。作用域链的创建过程跟执行上下文的建立有关...

---

## • 原型，原型链

在 js 中我们是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性值，这个属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当我们使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值（在 ES5 中这个指针被称为对象的原型。一般来说我们是不应该能够获取到这个值的，但是现在浏览器中都实现了 **proto** 属性来让我们访问这个属性，但是我们最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，我们可以通过这个方法来获取对象的原型）。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，那么它就

会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是Object.prototype 所以这就是我们新建的对象为什么能够使用toString() 等方法的原因。

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

---

## ● 继承

1、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。

2、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

3、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

4、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）

5、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

6、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

---

## ● new 操作符做什么

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

```
/**
* new的具体步骤
* 创建一个空对象 var obj = {}
* 修改obj.__proto__=Dog.prototype
* 只改this指向并且把参数传递过去,call和apply都可以
* 根据规范，返回 null 和 undefined 不处理，依然返回obj
*/
function _new(fn, ...args){
  const obj = Object.create(fn.prototype)
  const ret = fn.apply(obj, args)
  return ret instanceof Object ? ret : obj
}
```

- 使用new操作符时，构造函数内的this就指向相应的实例化对象；
- 未使用new操作符时，为普通函数调用，全局函数内的this指向window。

---

## 跨域

> **同源政策**
>
> 一个域下的 js 脚本在未经允许的情况下不能够访问另一个域的内容。这里的同源的指的是两个域的协议、域名、端口号必须相同，否则不属于同一个域。
>
> 同源政策主要限制了三个方面：
>
> - 当前域下的 js 脚本不能够访问其他域下的 cookie、localStorage 和 indexDB。
> - 当前域下的 js 脚本不能够操作访问操作其他域下的 DOM。
> - 当前域下的 ajax 无法发送跨域请求。
>
> 同源政策的目的主要是为了保证用户的信息安全，它只是对 js 脚本的一种限制，并不是对浏览器的限制，对于一般的 img、或者 script 脚本请求都不会有跨域的限制，这是因为这些操作都不会通过响应结果来进行可能出现安全问题的操作。

**解决跨域问题**

1. 使用 jsonp 来实现跨域请求，它的主要原理是通过动态构建 script 标签来实现跨域请求，因为浏览器对 script 标签的引入没有跨域的访问限制 。通过在请求的 url 后指定一个回调函数，然后服务器在返回数据的时候，构建一个 json 数据的包装，这个包装就是回调函数，然后返回给前端，前端接收到数据后，因为请求的是脚本文件，所以会直接执行，这样我们先前定义好的回调函数就可以被调用，从而实现了跨域请求的处理。这种方式只能用于 get 请求。
2. 使用 CORS 的方式，CORS 是一个 W3C 标准，全称是"跨域资源共享"。CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，因此我们只需要在服务器端配置就行。浏览器将 CORS 请求分成两类：简单请求和非简单请求。对于简单请求，浏览器直接发出 CORS 请求。具体来说，就是会在头信息之中，增加一个 Origin 字段。Origin 字段用来说明本次请求来自哪个源。服务器根据这个值，决定是否同意这次请求。对于如果 Origin 指定的源，不在许可范围内，服务器会返回一个正常的 HTTP 回应。浏览器发现，这个回应的头信息没有包含 Access-Control-Allow-Origin 字段，就知道出错了，从而抛出一个错误，ajax 不会收到响应信息。如果成功的话会包含一些以 Access-Control- 开头的字段。非简单请求是那种对服务器有

特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

> CORS与JSONP的使用目的相同，但是比JSONP更强大。
>
> JSONP只支持GET请求，CORS支持所有类型的HTTP请求。JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。

> **options请求**
>
> 在正式跨域的请求前，浏览器会根据需要，发起一个“PreFlight”（也就是Option请求），用来让服务端返回允许的方法（如get、post），被跨域访问的Origin（来源，或者域），还有是否需要Credentials(认证信息）
>
> - 当跨域请求是简单请求时不会进行preflight request,只有复杂请求才会进行preflight request。
>
> 跨域请求分两种：简单请求、复杂请求；
>
> 符合以下任一情况的就是复杂请求：
>
> 1.使用方法put或者delete;
>
> 2.发送json格式的数据（content-type: application/json）
>
> 3.请求中带有自定义头部；
>
> 除了满足以上条件的复杂请求其他的就是简单请求
>
> - 为什么跨域的复杂请求需要preflight request?
>
> 复杂请求可能对服务器数据产生副作用。例如delete或者put,都会对服务器数据进行修改,所以在请求之前都要先询问服务器，当前网页所在域名是否在服务器的许可名单中，服务器允许后，浏览器才会发出正式的请求，否则不发送正式请求。

3. 将 document.domain 设置为主域名，来实现相同子域名的跨域操作，这个时候主域名下的 cookie 就能够被子域名所访问。同时如果文档中含有主域名相同，子域名不同的 iframe 的话，我们也可以对这个 iframe 进行操作。
4. 使用 location.hash 的方法，我们可以在主页面动态的修改 iframe 窗口的 hash 值， 然后在 iframe 窗口里实现监听函数来实现这样一个单向的通信。因为在 iframe 是没有办法访问到不同源的父级窗口的，所以我们不能直接修改父级窗口的 hash 值来实现通信，我们可以在 iframe 中再加入一个 iframe ，这个 iframe 的内容是和父级页面同源的，所以我们可以 window.parent.parent 来修改最顶级页面的 src，以此来实现双向通信。
5. 使用 window.name 的方法，主要是基于同一个窗口中设置了 window.name 后不同源的页面也可以访问，所以不同源的子页面可以首先在 window.name 中写入数据，然后跳转到一个和父级同源的页面。这个时候级页面就可以访问同源的子页面中 window.name 中的数据了，这种方式的好处是可以传输的数据量大。
6. 使用 postMessage 来解决的方法，这是一个 h5 中新增的一个 api。通过它我们可以实现多窗口间的信息传递，通过获取到指定窗口的引用，然后调用 postMessage 来发送信息， 在窗口中我们通过对 message 信息的监听来接收信息，以此来实现不同源间的信息交换。如果是像解决 ajax 无法提交跨域请求的问题，我们可以使用 jsonp、cors、websocket 协议、 服务器

代理来解决问题。

7. 使用 websocket 协议，这个协议没有同源限制。
8. 使用服务器来代理跨域的访问请求，就是有跨域的请求操作时发送请求给后端，让后端代为请求，然后最后将获取的结果发返回。

---

## • 防抖与节流

> 函数防抖： 在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。
>
> 函数节流： 规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。

函数防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。 这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。 函数节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

**结合应用场景**

- 防抖(debounce)
    - search搜索联想，用户在不断输入值时，用防抖来节约请求资源。
    - window触发resize的时候，不断的调整浏览器窗口大小会不断的触发这个事件，用防抖来让其只触发一次
- 节流(throttle)
    - 鼠标不断点击触发，mousedown(单位时间内只触发一次)
    - 监听滚动事件，比如是否滑到底部自动加载更多，用throttle来判断

**函数防抖的实现**

```
function debounce(fn, wait) {
  var timer = null;
  return function () {
    var context = this,
      args = arguments;
    // 如果此时存在定时器的话，则取消之前的定时器重新记时
    clearTimeout(timer);
    // 设置定时器，使事件间隔指定事件后执行
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, wait);
  };
};
```

**函数节流的实现**

```
// 1. 利用闭包保存时间进行判断
```

```
function throttle(fn, delay) {
var preTime = Date.now();
return function () {
  var context = this,
    args = arguments,
    nowTime = Date.now();
  // 如果两次时间间隔超过了指定时间，则执行函数。
  if (nowTime - preTime >= delay) {
    preTime = Date.now();
    return fn.apply(context, args);
  }
};

// 2. 利用闭包保存计时器进行判断
function throttle(func, delay){//节流
  let timer = null;
  return function(){
    let context = this;
    let args    = arguments;
    if(!timer){
      timer = setTimeout(function(){
        func.apply(context, args);
        timer = null;
      }, delay); // 定时器只能靠真正执行的那一次自动清空，保证了期间其他的调用无法被执行
    }
  }
}
```

---

## ● 严格模式

严格模式指的是使js在更为严格的条件下运行。严格模式通过在脚本或函数的头部添加 **use strict**;表达式来声明。

为什么使用严格模式:

- 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
- 消除代码运行的一些不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的Javascript做好铺垫。

"严格模式"体现了Javascript更合理、更安全、更严谨的发展方向，包括IE 10在内的主流浏览器，都已经支持它，许多大项目已经开始全面拥抱它。

另一方面，同样的代码，在"严格模式"中，可能会有不一样的运行结果；一些在"正常模式"下可以运行的语句，在"严格模式"下将不能运行。掌握这些内容，有助于更细致深入地理解Javascript，让你变成一个更好的程序员。

严格模式特点:

1. 不允许使用隐式声明的变量，会报错。

2. 不允许函数有相同的参数，对象有相同的属性。
3. 不允许对只读属性赋值:
4. 禁止this关键字指向全局对象，也就是说this不能指向window顶层对象。
5. 不允许使用保留关键字（implements, interface, let, package, private, protected, public, static, yield）作为变量名。js一直处于发展中，这些保留关键字将来可能用来实现相应的功能。所以不能使用。
6. 创设eval作用域，在作用域 eval() 创建的变量不能在外部被调用

---

## • js面向对象的理解

> 在不会面向对象编程之前，我们都是采用面向过程编程的。按照传统流程编写一个个的函数来解决需求的这种方式就是过程编程。
>
> 面向对象编程就是将你的需求抽象成一个对象，然后针对这个对象分析其特征(属性)与动作(方法)。而这个对象我们就称之为 **类**。
>
> js是一个伪面向对象的语言，没有完整的面向对象的体系，至少es5及之前没有面向对象的体系，但是也有各种办法模拟其功能，到es6或以后，js的面向对象逐渐的展露头角。

- 面向对象编程 在es5中对象的构造函数用function创建，变量名首字母大写进行标识与普通函数区别。在构造函数用this.变量名设置对象实例的属性和方法，用构造函数.prototype将方法挂载在原型上。而在es6中引入了class、extends等关键字。和java的面向对象很像，js的class也是直接定义类名，加大括号，拥有构造函数constructor，实例的时候参数直接传递到构造函数中。js的继承使用extends关键字。
- 面向对象的三大特征 三大特征分别为：封装，继承和多态。这些特征本来是Java等面向对象语言的最大特征，但随着js的发展，不断吸收其他语言的优良特性，现在js也拥有了面向对象的特征。
- 1：封装

  我们平时所用的方法和类都是一种封装，当我们在项目开发中，遇到一段功能的代码在好多地方重复使用的时候，我们可以把他单独封装成一个功能的方法，这样在我们需要使用的地方直接调用就可以了。

  2：继承

  继承在我们的项目开发中主要使用为子类继承父类，下面是es6继承的书写方法

  三：多态

  多态的具体表现为方法重载和方法重写：

  方法重载：重载是指不同的函数使用相同的函数名，但是函数的参数个数或类型不同。调用的时候根据函数的参数来区别不同的函数

  方法重写：重写（也叫覆盖）是指在派生类中重新对基类中的虚函数（注意是虚函数）重新实现。即函数名和参数都一样，只是函数的实现体不一样
- 三大特征的优点：

  封装：封装的优势在于定义只可以在类内部进行对属性的操作，外部无法对这些属性指手画脚，要想修改，也只能通过你定义的封装方法；

> 继承：继承减少了代码的冗余，省略了很多重复代码，开发者可以从父类底层定义所有子类必须有的属性和方法，以达到耦合的目的；
>
> 多态：多态实现了方法的个性化，不同的子类根据具体状况可以实现不同的方法，光有父类定义的方法不够灵活，遇见特殊状况就捉襟见肘了

---

## ● 模块化开发

模块化是指将一个复杂的程序依据一定的规则（规范）或特定功能封装成几个块（文件）并进行组合。模块的内部数据的实现是私有的，只是向外部暴露一些接口（方法）与外部其他模块通信。这则就是模块化。模块化可以降低代码耦合度，减少重复代码，提高代码重用性，并且在项目结构上更加清晰，便于维护。

在最开始的时候，js 只实现一些简单的功能，所以并没有模块的概念，但随着程序越来越复杂，代码的模块化开发变得越来越重要。由于函数具有独立作用域的特点，最原始的写法是使用函数来作为模块，几个函数作为一个模块，但是这种方式容易造成全局变量的污染，并且模块间没有联系。

后面提出了对象写法，通过将函数作为一个对象的方法来实现，这样解决了直接使用函数作为模块的一些缺点，但是这种办法会暴露所有的所有的模块成员，外部代码可以修改内部属性的值。现在最常用的是立即执行函数的写法，通过利用闭包来实现模块私有作用域的建立，同时不会对全局作用域造成污染。

- **四种常见的模块化规范**

js 中现在比较成熟的有四种模块加载方案。

第一种是 CommonJS 方案，它通过 require 来引入模块，通过 module.exports 定义模块的输出接口。这种模块加载方案是服务器端的解决方案，它是以同步的方式来引入模块的，因为在服务端文件都存储在本地磁盘，所以读取非常快，所以以同步的方式加载没有问题。但如果是在浏览器端，由于模块的加载是使用网络请求，因此使用异步加载更加合适。

第二种是 AMD 方案，这种方案采用异步加载的方式来加载模块，模块的加载不影响后面语句的执行，所有依赖这个模块的语句都定义在一个回调函数里，等到加载完成后再执行回调函数。 require.js 实现了 AMD 规范。

第三种是 CMD 方案，这种方案和 AMD 方案都是为了解决异步模块加载的问题，sea.js 实现 了 CMD 规范。

> AMD推崇**依赖前置**：在定义模块的时候要先声明其依赖的模块，所以JS可以及其轻巧地知道某个模块依赖的模块是哪一个，因此可以立即加载那个模块，也就是**提前执行** 。
>
> CMD推崇**依赖就近**：它只要依赖的模块在附近就可以了 ，因此它要等到所有的模块变为字符串，解析一遍之后才知道他们之间的依赖关系，CMD加载完某个模块后没有立即执行而是等到遇到require语句的时再执行，也就是**延迟执行** 。
>
> 他们两者的目的都是为了 JavaScript 的模块化开发，但其不同导致各自的优点是：
>
> AMD用户体验好，因为模块提前执行了；
>
> CMD性能好，因为只有用户需要的时候才执行。

第四种方案是 ES6 提出的方案，使用 import 和 export 的形式来导入导出模块。这种方案和上面三种方案都不同。

## • 函数式编程

- 与面向对象编程（Object-oriented programming）和过程式编程（Procedural programming）并列的编程范式。
- 最主要的特征是，函数是第一等公民。
- 强调将计算过程分解成可复用的函数，典型例子就是`map`方法和`reduce`方法组合而成MapReduce 算法。
- 只有纯的、没有副作用的函数，才是合格的函数。
- 深入理解函数式编程

## • es6 新特性

### 一、 let 和 const

与var不同，let和const都是用于命名局部变量，都是块级作用域。具体可参考阮一峰老师的文章。

这三者的用法区别如下：

```
var val = "全局变量";

{
  let val = "局部变量";
  console.log(val);      // 局部变量
}

console.log(val);        // 全局变量

const val = "常量";
val = "123";             // Uncaught TypeError: Assignment to constant variable.
```

前面说const声明的是常量，一旦声明就不可再进行修改。但是当用const声明对象时，又会出现一种新情况，举个栗子：

```
const person = {name: "Peter", age: "22"};
person.age = 23;                        // 不会报错，person的age变量会被改成23
person = {name: "Lily", age: "18"};     // 报错
```

如果用const声明一个对象，对象所包含的值是可以被修改的。换一句话来说，只要对象指向的地址不被修改，就是允许的。

关于let和const有几个小tips：

1. let、const 关键词声明的变量不具备变量提升特性；
2. 使用let、const命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”
3. let 和 const 声明只在最靠近的一个块中有效；
4. 当使用常量 const 声明时，请使用大写变量，如：CAPITAL_CASING；

5. const 在声明时必须被赋值，否则会报错。

**二、 模版字符串**

在过去我们想要将字符串和变量拼接起来，只能通过运算符“+”来实现，若内容过多还要用“\”来表示换行，如：

```
var person = {name: "Peter", age: 22, career: "student"};
$(".introduction").html("Hello, my name is " + person.name + ", and my career
is " + person.career + ".");
```

而在ES6中，可以将反引号（``）将内容括起来，在反引号中，可以使用${}来写入需要引用到的变量。如：

```
var person = {name: "Peter", age: 22, career: "student"};
$(".introduction").html(`Hello, my name is ${person.name}, and my career is
${person.career}.`);
```

所以在ES6中，我们可以更方便地将字符串和变量连接起来。

**三、 箭头函数**

在ES6中引入了一种新的函数表达方式，它是函数的一种简写方法，有以下三个特点：

1. 不需要用关键字function来定义函数；
2. 一般情况下可以省略return；
3. 在箭头函数内部，this并不会跟其他函数一样指向调用它的对象，而是继承上下文的this指向的对象。

在上面的三点中，第三点尤为重要，初学者在使用时经常会忽略这一点。

下面举几个箭头函数的使用方法：

```
/*
** 对应上面所说的第3点
** 在箭头函数中，this的指向与它的上下文有关
** 在本例中，箭头函数fun的上下文是window，所以this指向的也是window
*/
window.val = 12;
let fun = () => {
    let val = 13;
    console.log(val);               // 13
    console.log(this.val);          // 12
    console.log(this == window);    // true
}
fun();

/*
** 普通函数使用
*/
```

```
let add = function(a, b){
    return a + b;
}

/*
** 箭头函数使用
*/
let add1 = (a, b) => a + b;

// 当参数只有一个时，可以将括号省略
let sqrt = a => a*a;
```

**四、 函数可以设置默认参数值**

在这之前，我们想要在函数中设置默认值，只能通过以下方法进行设置：

```
function printText(text){
    var text = text || "hello world!";
    console.log(text);
}

printText("My name is Peter");          // "My name is Peter";
printText();                            // "hello world!";
```

但是在ES6中定义了一种新方法，开发者可以直接使用如下方法设置函数的参数默认值：

```
function printText(text = "hello world!") {
    console.log(text);
}

printText("My name is Peter");  // "My name is Peter";
printText();                    // "hello world!";
```

**五、 Spread / Rest 操作符**

Rest运算符用于获取函数调用时传入的参数。举个栗子：

```
let fun = function(...args) {
    console.log(args);
}

const list = ["Peter", "Lily", "Tom"];
fun(list);    // ["Peter", "Lily", "Tom"]
```

Spread运算符用于数组的构造，析构，以及在函数调用时使用数组填充参数列表。再举个栗子：

```
/*
```

```
** 使用Spread运算符合并数组
*/
const list1 = ["Peter", "Tom"];
const list2 = ["Lily", "Mary"];
const list = [...list1, ...list2];
console.log(list); // ["Peter", "Tom", "Lily", "Mary"]]

/*
** 使用Spread运算符析构数组
*/
const [person, ...list3] = list;
console.log(person);        // Peter
console.log(list3);         // ["Tom", "Lily", "Mary"]
```

更多关于Rest和Spread运算法的使用方法，可以参考一下阮一峰老师的文章，es6之扩展运算符 三个点（...）

**六、 二进制和八进制的字面量**

ES6支持二进制和八进制的字面量，通过在数字前面增加0o或者0O可以将数字转换为八进制。

```
let val1 = 0o10;
console.log(val1);      // 8，八进制的0o10对应十进制的8

let val2 = 0b10;
console.log(val2);      // 2，二进制的0b10对应十进制的2
```

**七、 对象和数组解构**

ES6可以将对象中的属性或者数组中的元素进行解构，操作方式与前面所提到的Rest和Spread操作符类似，看一下下面这个栗子：

```
let person = {
    name: "Peter",
    age: 22,
    career: "student"
}

const {name, age, career } = person;
console.log(`Hello, my name is ${name}, and my career is ${career}.`);
//Hello, my name is Peter, and my career is student.
```

**八、 允许在对象中使用super方法**

super方法应该都不陌生，在java中用来代表调用父类的构造函数。由于js不是面向对象语言，所以也没有继承这以说法。但是在ES6中，可以通过调用setPrototypeOf()方法来设置一个对象的prototype对象，与面向对象语言中的继承有相似之处，所以也可以理解成这是js中用来实现继承的方法。（这段话纯属个人理解，如果有误请指出。）所以，在ES6中，通过使用super可以调用某个对象的prototype对象的方法或获取参数。栗子如下：

```
var father = {
    text: "Hello from the Father.",
    foo() {
        console.log("Hello from the Father.");
    }
}

var son = {
    foo() {
        super.foo();
        console.log(super.text);
        console.log("Hello from the Son.");
    }
}

/*
** 将father设置成son的prototpe
** 当在son中调用super时，可以直接调用到它的prototype对象，即father的方法和变量
*/
Object.setPrototypeOf(son, father);
son.foo();
// Hello from the Fater.
// Hello from the Fater.
// Hello from the Son.
```

### 九、 迭代器iterator、for...of和for...in

首先你要理解什么是iterator。

了解完iterator之后，便可以来深入了解一下for...of和for...in这两种方法了。用一句话来总结就是，无论是for...in还是for...of语句，都是用来迭代数据，它们之间的最主要的区别在于它们的迭代方式不同。

- for...in 语句以原始插入顺序迭代对象的可枚举属性，简单理解就是for...in是用来循环遍历属性，遍历出的是自身和原型上的可枚举非symbol属性，但是遍历不一定按照顺序（tips：for...in在ES5中就已经出现了）
- for...of 语句遍历可迭代对象定义要迭代的数据，也就是说，for...of只可以循环可迭代对象的可迭代属性，不可迭代属性在循环中被忽略了。（tips：for...of是ES6才提出来的）

关于for...in和for...of的用法，可以看以下栗子：

```
Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];
iterable.foo = 'hello';
//for in 会继承
for (let i in iterable) {
    console.log(i); // 依次打印 0, 1, 2, "foo", "arrCustom", "objCustom"
```

```
}

for (let i in iterable) {
    if (iterable.hasOwnProperty(i)) {
        console.log(i); // 依次打印 0, 1, 2, "foo"
    }
}

// for of
for (let i of iterable) {
    console.log(i); // 依次打印 3, 5, 7
}
```

`for...in`循环有几个缺点。

- 数组的键名是数字，但是`for...in`循环是以字符串作为键名“0”、“1”、“2”等等。
- `for...in`循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。
- 某些情况下，`for...in`循环会以任意顺序遍历键名。

总之，`for...in`循环主要是为遍历对象而设计的，不适用于遍历数组。

`for...of`循环相比上面几种做法，有一些显著的优点。

```
for (let value of myArray) {
  console.log(value);
}
```

- 有着同`for...in`一样的简洁语法，但是没有`for...in`那些缺点。
- 不同于`forEach`方法，它可以与`break`、`continue`和`return`配合使用。
- 提供了遍历所有数据结构的统一操作接口。

### 十、 class

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过`class`关键字，可以定义类。

基本上，ES6 的`class`可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的`class`写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

---

# 项目及项目技术

## MES项目描述

## 项目简介

一个应用于包装印刷企业的MES（制造执行）系统，主要的功能有对企业的生产资源进行数字化建模；制定车间级派工计划，并实现计划追踪；在关键工艺点设置数据采集站并完成对物料、设备、人员、工艺流程等资源等数据采集和信息管理，实现产品生产全过程的追踪和追溯；实现各层级的看板，对企业生产数据进行可视化展示，实时掌握车间计划的执行情况等。

## 职责描述

主要负责系统中的车间数据可视化功能模块的设计与前端开发，并与后端人员协同进行模块开发。该模块的主要功能有物理车间展示、逻辑车间展示、设备人员利用率展示、车间单位管理等。具体工作有 · 定义了通用的车间设备信息数据结构，适用于车间可视化各个功能模块； · 借助echarts完成逻辑车间的制造工艺、物料流动等可视化展示； · 借助heatmap.js以热力图形式完成车间设备利用率的可视化展示； · 封装可复用组件及功能函数类，方便项目中重复使用。

相关技术栈：Vue，Element - UI，Webpack，Echarts

---

# 项目难点

## 物理车间可视化的实现

车间物理可视化模块的主要功能是展示某一车间实时的生产信息，包括设备与人员的生产信息与详细信息，关键生产事件的推送，库存信息以及生产指标和设备状态汇总的图表展示。

其中最为关键的部分就是设备与人员的信息展示，因为这一部分的展示是基于实际的车间布局图，要求我们在布局图中能够用方框代表展示出设备人员的实际物理位置与大小，也就是一个物理车间的映射，同时点击每一个设备或人员要弹出他的一个详细信息界面。

考虑到这一个模块展示的不只是一个车间，以及设备在车间中的位置大小信息都是存储在数据库当中，所以我们不能将代表设备人员的方框直接写死，而应该通过请求获取当前车间的设备人员数据，动态的渲染到页面当中。那么用何种数据结构来存储设备、人员的信息，以及用何种方式将这种数据结构渲染到页面的车间布局图当中，这两个是这个页面中最大的难点。

1. 数据渲染

   先说数据渲染方式的确定，因为这个影响到后面的数据结构。我采用的方案是 绝对百分比定位 + 百分比大小 。通过绝对定位来确定设备方框与布局图上方和左侧的距离top / left，方框的长款width/height 来确定设备的大小，而考虑到相应式设计的要求；车间布局图的尺寸是auto撑满容器的，也就是说设备方框的容器尺寸不确定，故采用百分比进行定位和大小，也就是设备相对于车间的百分比定位和尺寸，进而实现响应式设计的要求。

   刚刚说的都是设备的渲染方式，人员相对来说比较简单，因为根据实际情况我们得知，人员是与设备绑定的，也就是说人员的出现一定是跟附着设备，这样的话我们就可以将人员数据放到相应的设备对象当中，让设备框成为人员框的一个容器，这样以来就可以省略人员框的位置确定。

   最后我们通过 v-for 遍历设备生产数据，v-if 如果设备携带人员数据就把设备框座位容器渲染人员框。最终实现能够展示实际物理车间布局的可视化效果。

2. 数据结构 明确了渲染方式之后数据结构也就清晰了，主体数据对象是一个数组，包含了这个车间的所有设备对象，每一个设备对象包含两个关键的对象：位置大小信息，生产人员信息，以及一些其他的一些设备基础信息比如设备编号、名称等等。

还涉及到的技术点：

1. 设备人员边框的冒泡问题，需要点击查看相应的详细信息 --- vue中取消事件冒泡 @click.stop()
2. 手写了一个推送框缩放效果 --- 事件绑定更换类样式

# 逻辑车间可视化的展示

# echarts的使用

**直角坐标热力图** -> heatmap.js

**切换其他组件图表时出现卡顿**

- 原因：每一个图例在没有数据的时候它会创建一个定时器去渲染气泡，页面切换后，echarts图例是销毁了，但是这个echarts的实例还在内存当中，同时它的气泡渲染定时器还在运行。这就导致Echarts占用CPU高，导致浏览器卡顿，当数据量比较大时甚至浏览器崩溃
- 解决方法：在mounted()方法和destroy()方法之间加一个beforeDestroy()方法释放该页面的chart资源，clear()方法则是清空图例数据，不影响图例的resize，而且能够释放内存，切换的时候就顺畅了

```
beforeDestroy () {
  this.chart.clear()
}
```

---

# 车间基础单位管理 —— 原生js实现拖拽

> 首先是三个事件，分别是 mousedown，mousemove，mouseup 当鼠标点击按下的时候，需要一个 tag 标识此时已经按下，可以执行 mousemove 里面的具体方法。 clientX，clientY 标识的是鼠标的坐标，分别标识横坐标和纵坐标，并且我们用 offsetX 和 offsetY 来表示元素的元素的初始坐标，移动的举例应该是： 鼠标移动时候的坐标 - 鼠标按下去时候的坐标。 也就是说定位信息为： 鼠标移动时候的坐标 - 鼠标按下去时候的坐标 + 元素初始情况下的 offetLeft.

在车间可视化模块中，最常用的可视化模式为以车间布局图为背景，以车间基础单位（设备、线边库等）的位置大小信息进行定位作为详细信息的载体，以此作为可视化模版，在此基础之上用不同的可视化方案对车间不同维度的信息进行展示。由此可见，车间基础单位（设备、线边库等）的位置大小信息作为可视化模块的基础通用数据，应该进行数据抽离，同时应该有一个统一的GUI界面进行管理维护。这里我的方案是先选择车间，在以车间布局图为背景进行车间基础单位的信息管理，采用鼠标拖拽生成矩形框的形式预览车间基础单位的位置大小信息，依次替换信息的人工录入，使用户操作更加便捷高效。

**拖拽生成矩形框的实现**

一个元素的拖拽过程，我们可以分为三个步骤，第一步是鼠标按下目标元素，第二步是鼠标保持 按下的状态移动鼠标，第三步是鼠 标抬起，拖拽过程结束。 这三步分别对应了三个事件，mousedown 事件，mousemove 事件和 mouseup 事件。只有在鼠 标按下的状态移动鼠标我们才会执行拖拽事件，因此我们需要在 mousedown 事件中设置一个状态来标识鼠标已经按下，然后在 mouseup 事件中再取消这个状 态。在 mousedown 事件中我们首先应该判断，目标元素是否为拖拽元素，如果是拖拽元素，我们就设置状态并且保存这个时候鼠 标的位置。然后在 mousemove 事件中，我们通过判断鼠标现在的位置和以前位置的相对移动， 来确定拖拽元素在移动中的坐标。最后 mouseup 事件触发后，清除状态，结束拖拽事件。

**优化 —— HTML5的拖放API**

在HTML5之前，如果要实现拖放效果，一般会使用mousedown、mousemove和mouseup三个事件进行组合来模拟出拖拽效果，比较麻烦。而HTML5规范实现了原生拖放功能，使得元素拖放的实现更加方便和高效。

- dragstart：事件主体是被拖放元素，在开始拖放被拖放元素时触发
- drag：事件主体是被拖放元素，在正在拖放被拖放元素时触发
- dragenter：事件主体是目标元素，在被拖放元素进入目标元素时触发
- dragover：事件主体是目标元素，在被拖放元素在目标元素内移动时触发
- dragleave：事件主体是目标元素，在被拖放元素移出目标元素时触发
- drop：事件主体是目标元素，在目标元素完全接受被拖放元素时触发
- dragend：事件主体是被拖放元素，在整个拖放操作结束时触发

## 前端优化

降低请求量：合并资源，减少HTTP 请求数，minify / gzip 压缩，webP，lazyLoad。

加快请求速度：预解析DNS，减少域名数，并行加载，CDN 分发。

缓存：HTTP 协议缓存请求，离线缓存 manifest，离线数据缓存localStorage。

渲染：JS/CSS优化，加载顺序，服务端渲染，pipeline。

---

# 项目技术

## • Vue

> 推荐阅读：
>
> 1. Vue常见面试问题
> 2. Vue基础语法介绍

### 基础知识

**Angularjs，React，Vue**

- vue 对比 angularjs
    1. vue在设计之初参考了很多angularjs的思想（angjularjs != angular）
    2. vue相对比与angular比较简单
    3. vue相对比与angular比较小巧，运行速度快

4. vue与angular数据绑定都可以用 {{ }}
5. vue指令用v－xxx angularjs用ng－xxx
6. vue数据放在data对象里面，angular数据绑定到$scope对象上

- vue 对比 react

1. vue与react都使用 virtual DOM(虚拟DOM)
2. vue与react都提供了组件化的视图组件
3. vue与react将注意力集中保持在核心库，有丰富的插件库
4. react使用jsx渲染页面，vue使用更简单的模版
5. vue比react运行速度更快

**vue.js原理**

JS感知URL变化，当URL发生变化后，使用JS动态把当前的页面内容清除掉，再把下一个页面的内容挂载到页面上。此时的路由就不是后端来做了，而是前端来做，判断页面到底显示哪一个组件，再把以前的组件清除掉使用新的组件。就不会每一次跳转都请求HTML文件。页面跳转不需要去做HTML文件的请求，节约HTTP请求发送的时延。

**单页应用与多页应用**

**多页面应用：每次页面跳转，后台都会返回一个新的HTML文档，就是多页面应用。**

在以往传统开发的应用（网站）大多都是多页面应用，路由由后端来写。

- 首屏时间快？访问页面，服务器只需要返回一个HTML文件，这个过程就经历了一个HTTP请求，请求响应回来，页面就能被展示出来。
- SEO（搜索引擎排名）效果好？搜索引擎能识别HTML的内容，根据内容进行排名。
- 页面切换慢：每一次切换页面都需要发起一个HTTP请求，假设网络较慢就会出现卡顿情况。

**单页应用：用vue写的项目是单页应用，刷新页面会请求一个HTML文件，切换页面的时候，并不会发起新的请求一个HTML文件，只是页面内容发生了变化**

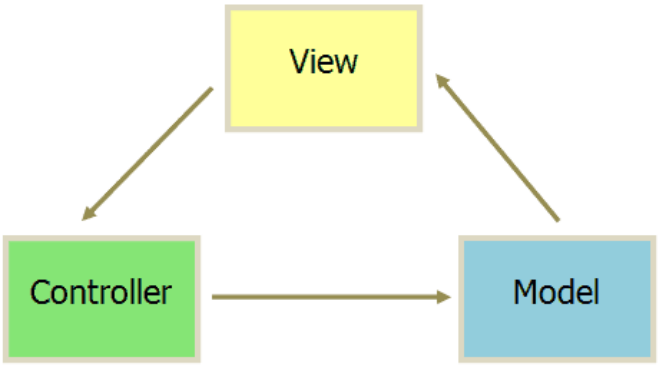
- 页面跳转不需要去做HTML文件的请求，节约HTTP请求发送的时延。
- SEO差？搜索引擎只认识HTML内容不认识JS内容。单页应用的渲染都是靠JavaScript渲染出来的。搜索引擎不好识别排名。

**MVVM / MVC / MVP 模式**

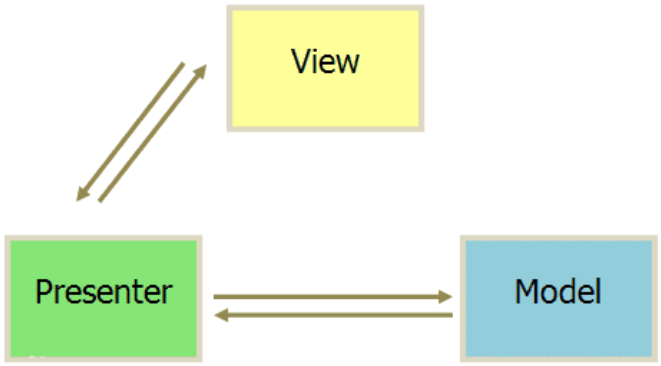
MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化我们的开发效率。

比如说我们实验室在以前项目开发的时候，使用单页应用时，往往一个路由页面对应了一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，当时一旦项目变得复杂，那么整个文件就会变得冗长，混乱，这样对我们的项目开发和后期的项目维护是非常不利的。

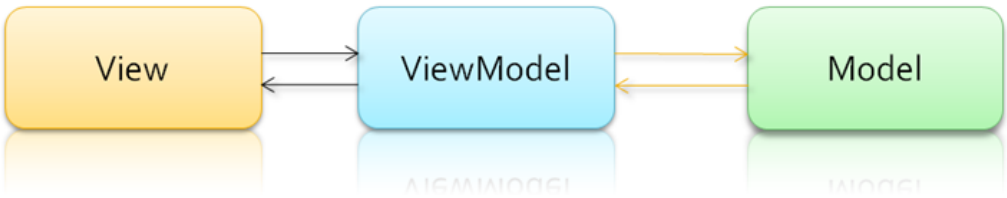
MVC 通过分离 Model、View 和 Controller 的方式来组织代码结构。其中 View 负责页面的显示逻辑，Model 负责存储页面的业务数据，以及对相应数据的操作。并且 View 和 Model 应用了观察者模式，当 Model 层发生改变的时候它会通知有关 View 层更新页面。Controller 层是 View 层和 Model 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，Controller 中的事件触发器就开始工作了，通过调用 Model 层，来完成对 Model 的修改，然后 Model 层再去通知 View 层更新。




MVP 模式与 MVC 唯一不同的在于 Presenter 和 Controller。在 MVC 模式中我们使用观察者模式，来实现当 Model 层数据发生变化的时候，通知 View 层的更新。这样 View 层和 Model 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代 码的复用性造成一些问题。MVP 的模式通过使用 Presenter 来实现对 View 层和 Model 层 的解耦。MVC 中的 Controller 只知道 Model 的接口，因此它没有办法控制 View 层的更新，MVP 模式中，View 层的接口暴露给了 Presenter 因此我们可以在 Presenter 中将 Model 的变化和 View 的 变化绑定在一起，以此来实现 View 和 Model 的同步更新。这样就实现了对 View 和 Model 的解耦，Presenter 还包含了其他的响应逻辑。




MVVM 模式中的 VM，指的是 ViewModel，它和MVP 的思想其实是相同的，不过它通过双向的数据绑定，将 View 和 Model 的同步更新给自动化了。当 Model 发生变化的时候，ViewModel 就会自动更新；ViewModel 变化了，View 也会更新。这样就将 Presenter 中的工作给自动化了。我了解过一点双向数据绑定的原理，比如 vue 是通过使用数据劫持和发布订阅者模式来实现的这一功能。

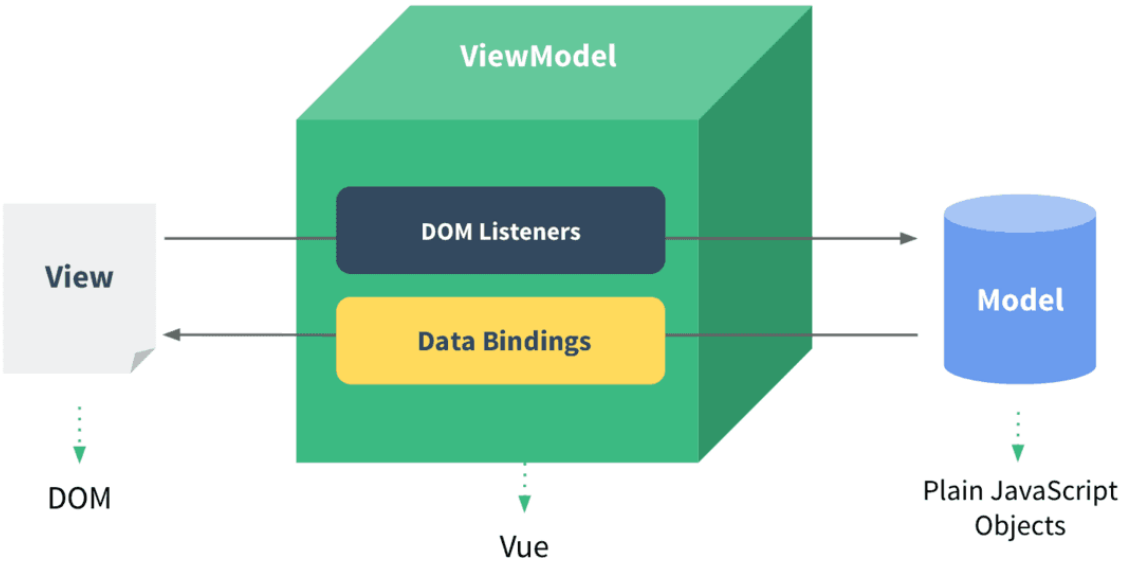


**双向绑定**

vue的双向绑定首先联系到的就是MVVM(Model-View-ViewModel)模式

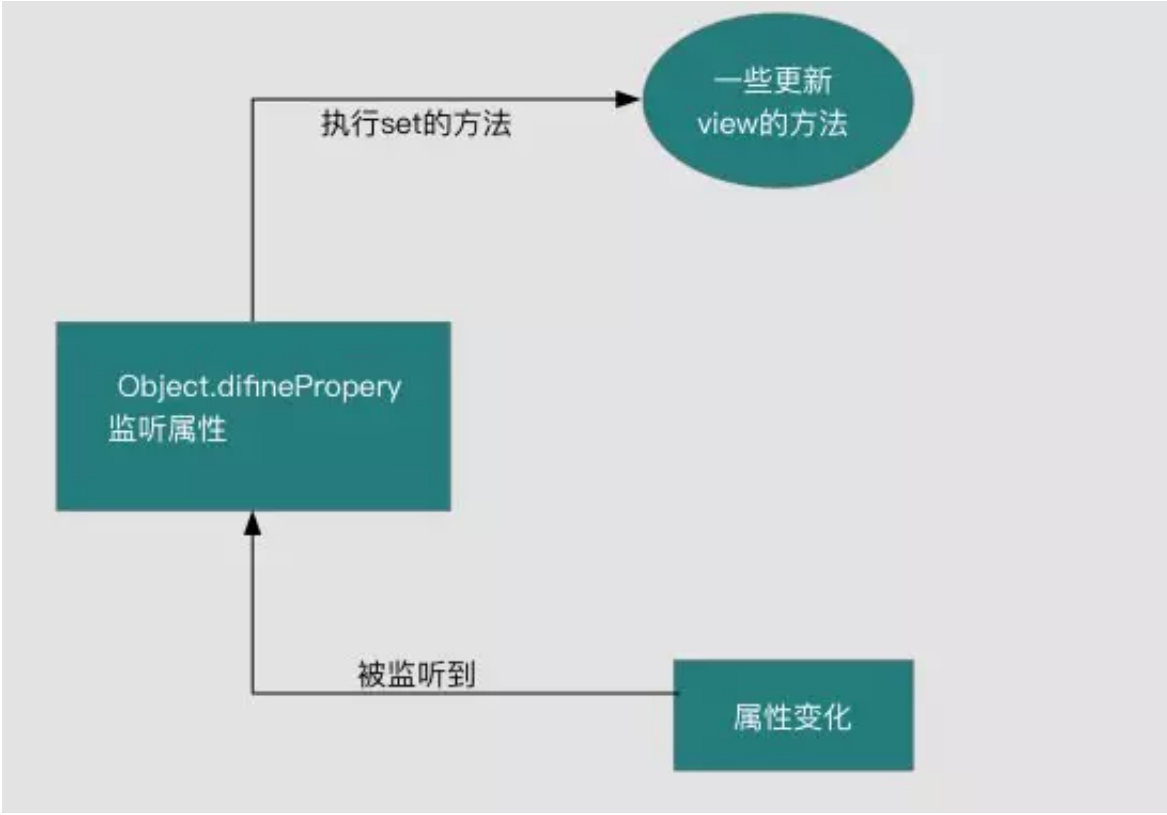
MVVM模式是通过以下三个核心组件组成，每个都有它自己独特的角色：

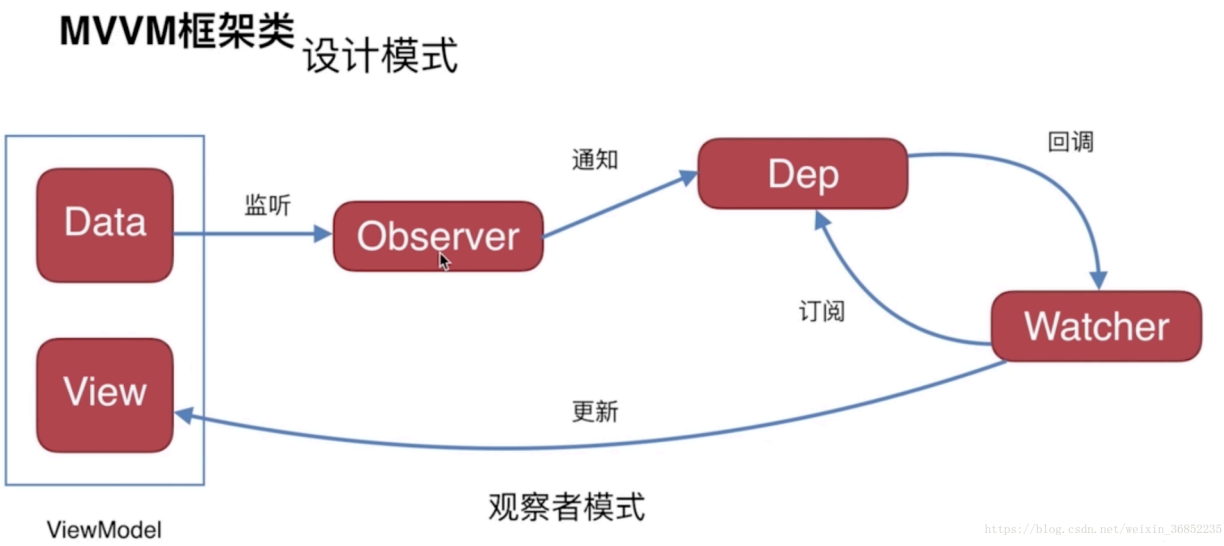
- **Model** - 包含了业务和验证逻辑的数据模型
- **View** - 定义屏幕中View的结构，布局和外观
- **ViewModel** - 它相当于中间枢纽的作用,连接着model 和 view
- 当前台显示的view发生变化了，它会实时反应到viewModel上，如果有需要，viewModel 会通过ajax等方法将改变的数据传递给后台model
- 同时从后台model获取过来的数据，通过vm将值响应到前台UI上




vm的核心是 view 和 data

- 当 data 有变化的时候它通过 Object.defineProperty() 方法中的 set 方法进行监控，并调用在此之前已经定义好 data 和 view 的关系的回调函数，来通知 view 进行数据的改变
- 而 view 发生改变则是通过底层的input 事件来进行data的响应更改



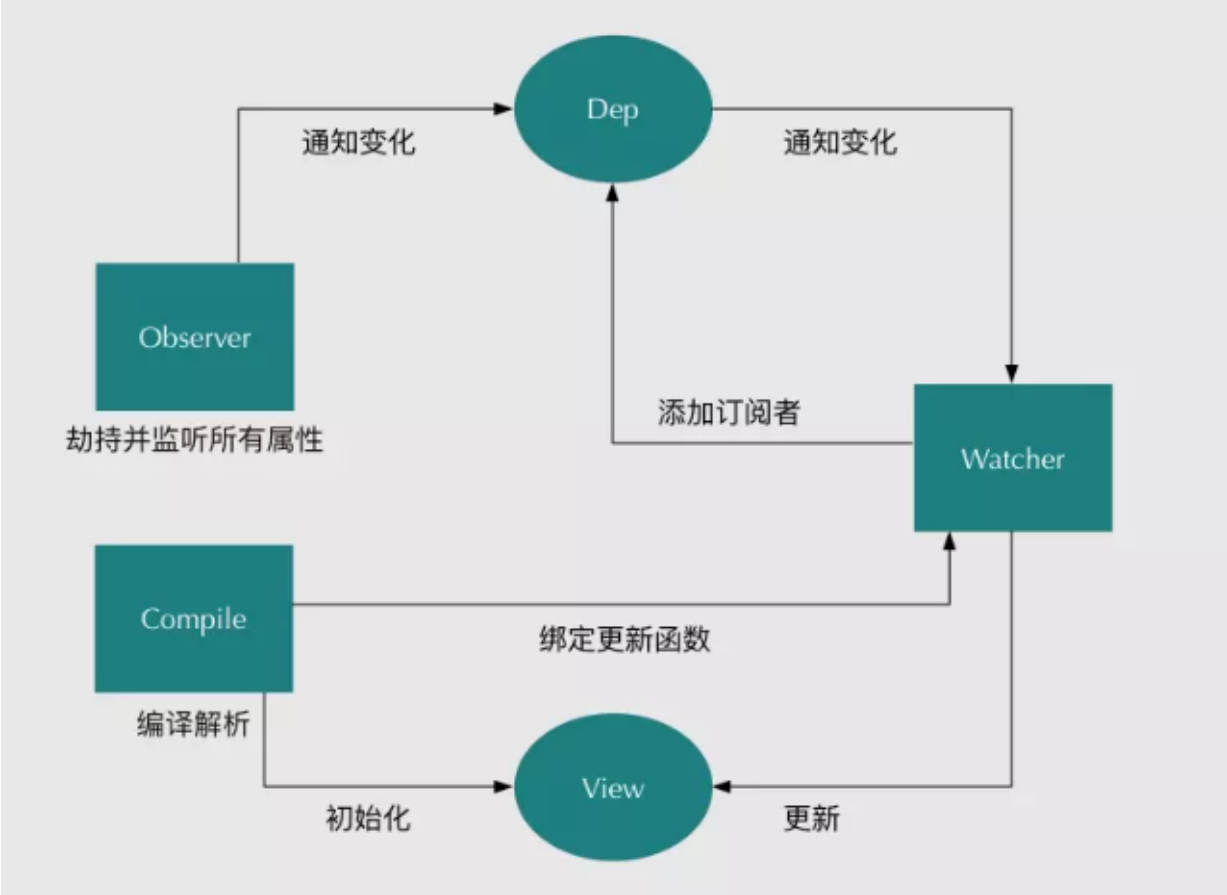

vue的数据双向绑定是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

具体实现：

- 第一步：需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter和getter 这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化
- 第二步：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
- 第三步：Watcher订阅者是Observer和Compile之间通信的桥梁，主要做的事情是:
    1. 在自身实例化时往属性订阅器(dep)里面添加自己
    2. 自身必须有一个update()方法

3. 待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

- 第四步：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数据变化，通过Compile来解析编译模板指令，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变更的双向绑定效果。




以上的图片可以具，体归纳为：

1. 实现一个监听器Observer，用来劫持并监听所有属性，如果有变动的，就通知订阅者。
2. 实现一个订阅者Watcher，可以收到属性的变化通知并执行相应的函数，从而更新视图。
3. 实现一个解析器Compile，可以扫描和解析每个节点的相关指令，并初始化模板数据以及初始化相应的订阅器。

**2.0双向绑定的缺陷及3.0的优化**

- Object.defineProperty 的缺陷

1. 无法检测到新的属性添加/删除

   在开发过程中，我们可能会遇到这样一种情况：当在 Vue 实例中的 data 里边声明或者已经赋值过的对象或者数组（数组里边的值是对象）时，向对象中添加新的属性，如果更新此属性的值，是不会更新视图的。

   根据当时的官方文档定义：**如果在 Vue 实例创建之后，添加新的属性到实例上，是不会触发视图更新的**。

当你把一个普通的 JavaScript 对象传入 Vue 实例作为 data 选项，Vue 将遍历此对象所有的属性，并使用 Object.defineProperty() 把这些属性全部转为 getter/setter，从而实现对 data 的监听，从而实现一旦 data 改变就刷新视图的效果。但是如果是后来追加的属性，则是不会调用 Object.defineProperty() 对新属性进行监听的，如果新属性改变了，虽然数据在内存中的值会改变，但是 Vue 实例是监听不到新的属性的，所以不会调用 render() 函数对视图进行刷新。对于这种情况，Vue 2.x 提出的解决办法是 Vue.$set()

2. 无法监听数组的变化

   Object.defineProperty() 只能对属性进行数据劫持，不能对整个对象进行劫持，同理无法对数组进行劫持，但是我们在使用 Vue 框架中都知道，Vue 通过遍历属性或者数组的项进行观察，实现数据劫持，但是这个劫持是有缺陷的。

   即无法监听数组根据 index 对于元素的赋值（**因为数组是在实例化过程中遍历数组进行 observe 的**），如果这样赋值就相当于把 index 的指向变成了一个新的位置，但是这个位置的对象是没有进行 observe 的。

   Vue 能够监听数组变化的场景：（1）通过赋值的形式改变正在被监听的数组；（2）通过 splice(index,num,val)的形式改变正在被监听的数组；（3）通过数组的 push() 的形式改变正在被监听的数组；

   Vue 无法监听的数组变化的场景：（1）通过数组索引改变数组元素的值；（2）改变数组的长度；

   解决无法监听数组变化的方法：（1）this.$set(arr, index, newVal)；（2）通过 splice(index，num，val)；使用临时变量作为中转，重新赋值数组；

   补充：Vue 2.x 针对数组是在源码中单独进行处理的，将数组的原始方法进行变异：push()，pop()，shift()，unshift()，splice()，sort()，reverse()。（这就是为什么你在控制台打印一个数组，数组最后会多一个 observer 项）。

3. 需要深度遍历，浪费内存

   Vue 通过遍历属性或者数组的项进行观察，实现数据劫持，如果在嵌套比较深的情况下，是需要深度遍历，这样会浪费内存（对象深度遍历）。

- 3.0 的数据双向绑定原理

基于上述的问题，Vue 3.0 进行了改进：使用 ES6 的 Proxy 对象，通过 reactive() 函数给每一个对象都包一层 Proxy，通过 Proxy 监听属性的变化，从而实现对数据的监控。

reactive()源码中声明了两个缓存，toProxy 和 toRaw，前者用于监测属性是否被监听过，如果已经被监听了就直接返回之前存储在缓存中的 observed。第二个缓存，如果传入的是 observed，从 toRaw 的 WeakMap 里面直接查找，如果存在就可以不进行代理，这样可以避免重复代理。

Vue 3.0 版本相比 2.0 增加了 Proxy 之后，就可以直接拦截所有对象类型数据的操作，对数组也是支持的，同时具有一个缓存的机制，多层对象嵌套的时候就可以使用懒代理。

| **Vue 2.0 版本存在的问题** | **Vue 3.0 版本增加 Proxy 后** |
| --- | --- |
| 无法监测新属性的 添加/删除 | 允许框架拦截对象上的操作 |
| 无法监听数组的一些变化 | Proxy 代理默认支持数组 |
| 多层对象嵌套，需要深度遍历，浪费内存 | 多层对象嵌套的时候就可以使用懒代理 |

**虚拟DOM**

我对 Virtual DOM 的理解是

首先对我们将要插入到文档中的 DOM 树结构进行分析，使用 js 对象将其表示出来，比如一个元素对象，包含 TagName、props 和 Children 这些属性。然后我们将这个 js 对象树给保存下来，最后再将 DOM 片段插入到文档中。

当页面的状态发生改变，我们需要对页面的 DOM 的结构进行调整的时候，我们首先根据变更的状态，重新构建起一棵对象树，然后将这棵新的对象树和旧的对象树进行比较，记录下两棵树的的差异。

最后将记录的有差异的地方应用到真正的 DOM 树中去，这样视图就更新了。

我认为 Virtual DOM 这种方法对于我们需要有大量的 DOM 操作的时候，能够很好的提高我们的操作效率，通过在操作前确定需要做的最小修改，尽可能的减少 DOM 操作带来的重流和重绘的影响。其实 Virtual DOM 并不一定比我们真实的操作 DOM 要快，这种方法的目的是为了提高我们开发时的可维护性，在任意的情况下，都能保证一个尽量小的性能消耗去进行操作。

**Diff 算法**

- **diff算法的时间复杂度**

  两个树的完全的 diff 算法是一个时间复杂度为 O(n^3) , Vue 进行了优化O(n^3) 复杂度的问题基于两个假设转换成 O(n) 复杂度的问题(只比较同级不考虑跨级问题)

  > 1. 两个相同的组件产生类似的DOM结构，不同的组件产生不同的DOM结构。
  > 2. 同一层级的一组节点，他们可以通过唯一的id进行区分。

  因为在前端操作dom的时候了，不会把当前元素作为上一级元素或下一级元素，很少会跨越层级地移动Dom元素，常见的都是同级的比较。 所以当页面的数据发生变化时，Diff算法只会比较同一层级Virtual Dom的节点：

  1. **如果节点类型不同，直接干掉前面的节点，再创建并插入新的节点，不会再比较这个节点以后的子节点了。**
  2. **如果节点类型相同，则会重新设置该节点的属性，从而实现节点的更新。**

- **vue中diff算法的原理**

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

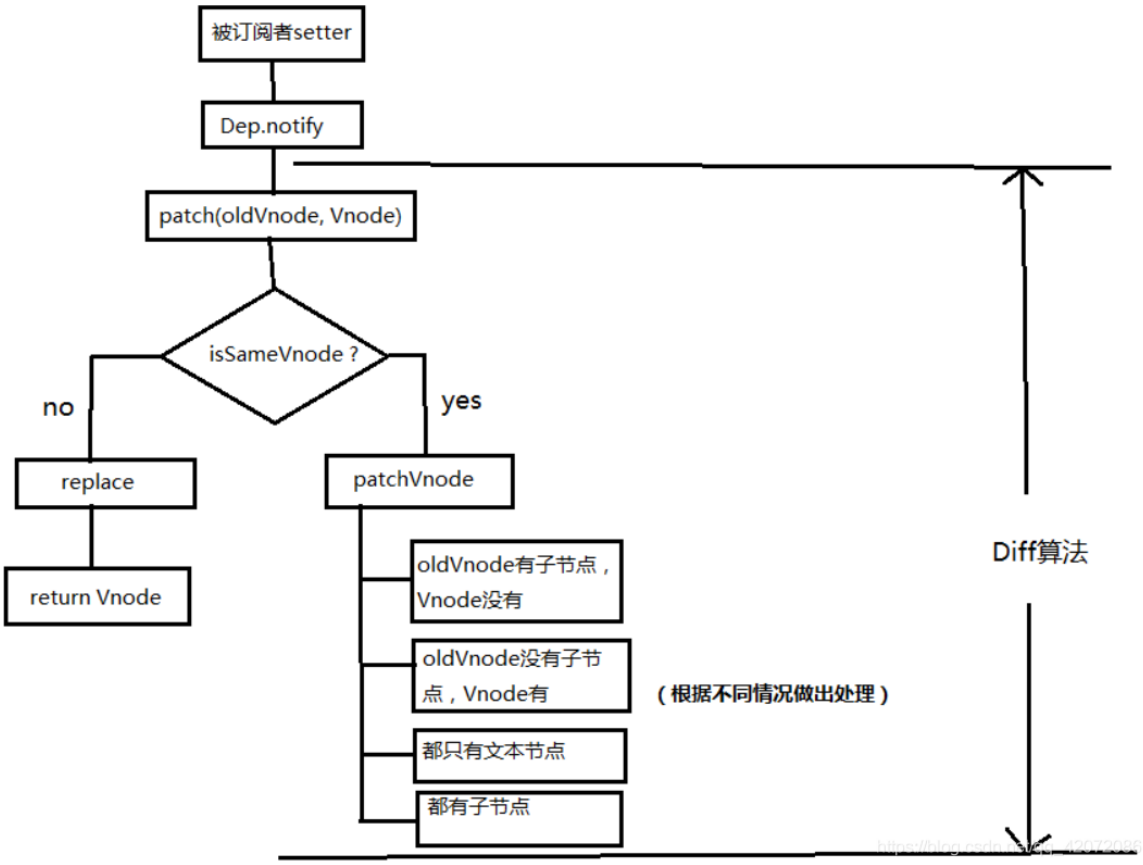
  - 原理简述：

  （1）先去同级比较，然后再去比较子节点

  （2）先去判断一方有子节点一方没有子节点的情况

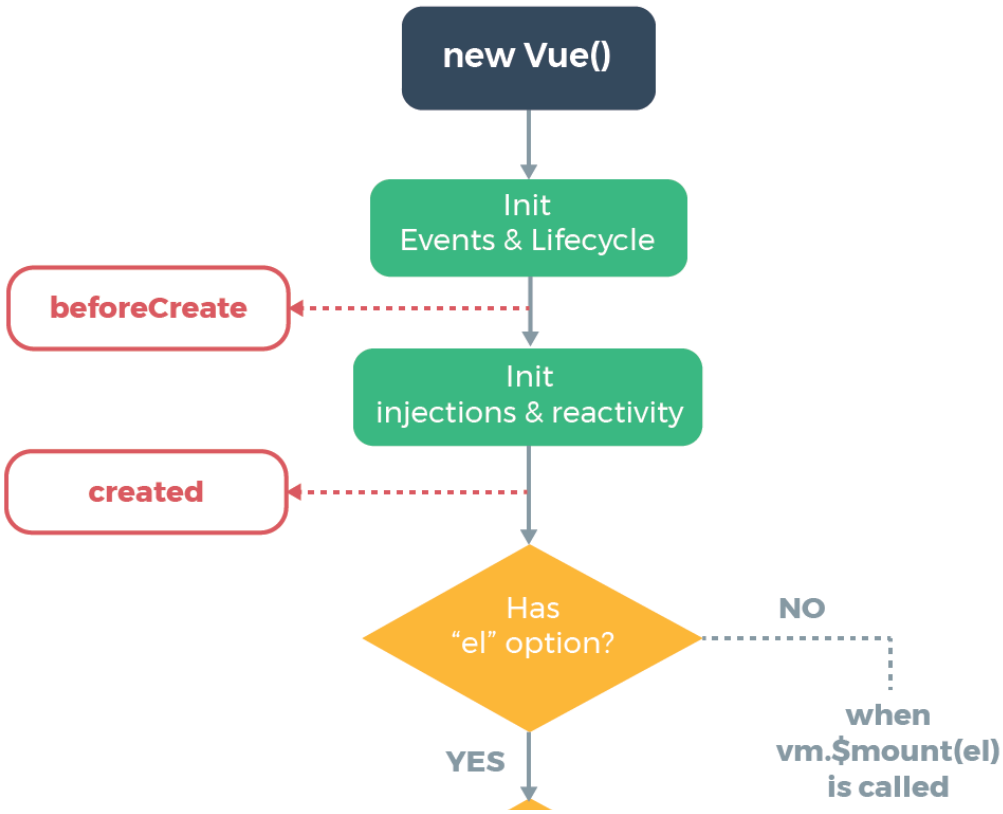
  （3）比较都有子节点的情况

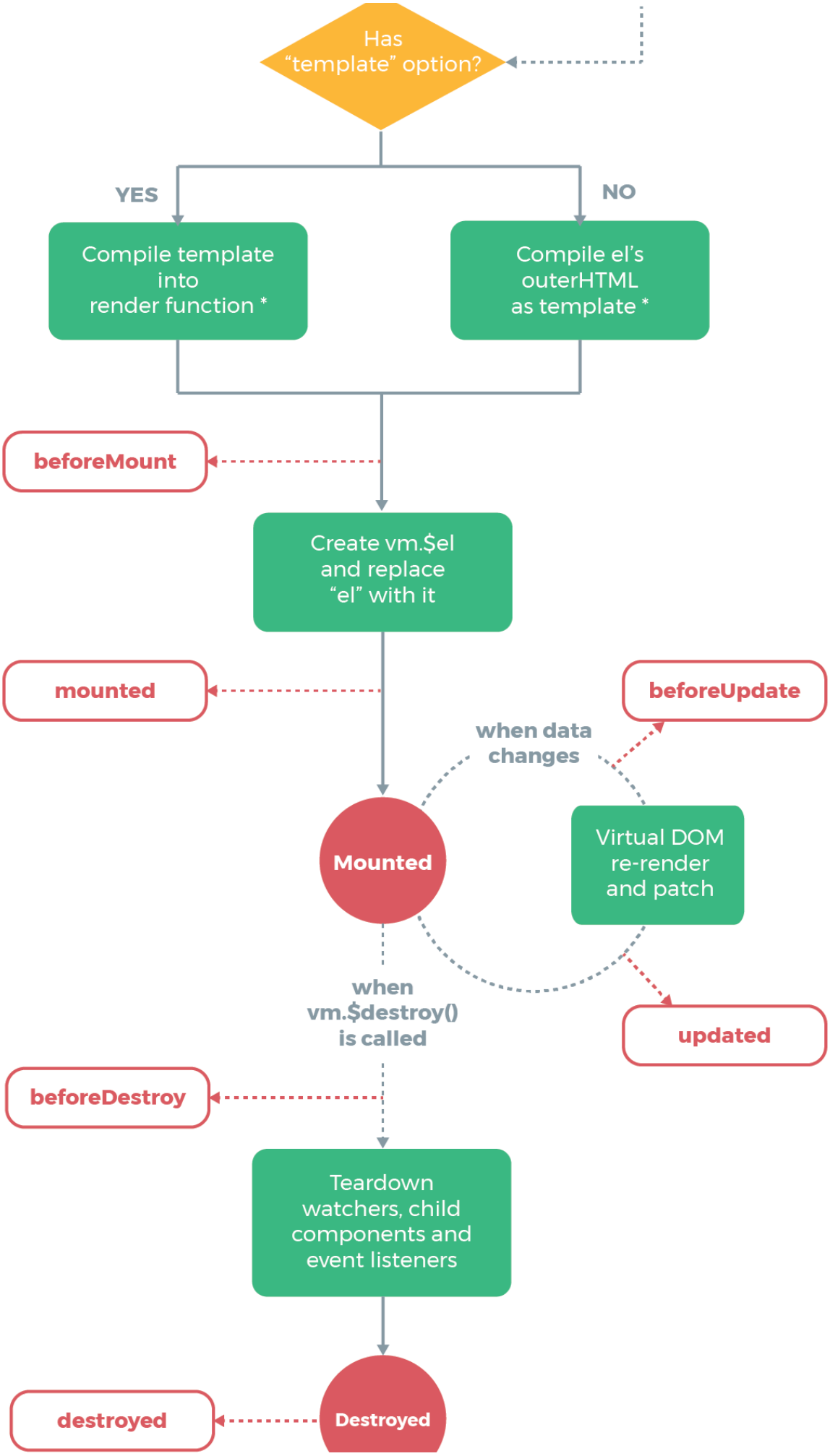
  （4）递归比较子节点




**生命周期**

> Vue 的生命周期指的是组件从创建到销毁的一系列的过程，被称为 Vue 的生命周期。通过提供的 Vue 在生命周期各个阶段的钩子函数，我们可以很好的在 Vue 的各个生命阶段实现一些操作。




Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data
changes
Mounted
Virtual DOM
re-render
and patch
when
vm.$destroy()
is called
updated
beforeDestroy
Teardown
watchers, child
components and
event listeners
destroyed
Destroyed

* template compilation is performed ahead-of-time if using
a build step, e.g. single-file components

Vue 一共有 8 个生命阶段，分别是创建前、创建后、加载前、加载后、更新前、更新后、销毁 前和销毁后，每个阶段对应了一个生命周期的钩子函数。

1. beforeCreate 钩子函数，在实例初始化之后，在数据监听和事件配置之前触发。因此在 这个事件中我们是获取不到 data 数据的。
2. created 钩子函数，在实例创建完成后触发，此时可以访问 data、methods 等属性。但 这个时候组件还没有被挂载到页面中去，所以这个时候访问不到 $el 属性。一般我们可以在这 个函数中进行一些页面初始化的工作，比如通过 ajax 请求数据来对页面进行初始化。
3. beforeMount 钩子函数，在组件被挂载到页面之前触发。在 beforeMount 之前，会找到对应的 template，并编译成 render 函数。
4. mounted 钩子函数，在组件挂载到页面之后触发。此时可以通过 DOM API 获取到页面中 的 DOM 元素。
5. beforeUpdate 钩子函数，在响应式数据更新时触发，发生在虚拟 DOM 重新渲染和打补丁之前，这个时候我们可以对可能会被移除的元素做一些操作，比如移除事件监听器。
6. updated 钩子函数，虚拟 DOM 重新渲染和打补丁之后调用。
7. beforeDestroy 钩子函数，在实例销毁之前调用。一般在这一步我们可以销毁定时器、 解绑全局事件等。
8. destroyed 钩子函数，在实例销毁之后调用，调用后，Vue 实例中的所有东西都会解除 绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

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、编译模板、挂载Dom、渲染→更新→渲染、销毁等一系列过程，我们称这是Vue的生命周期。通俗说就是Vue实例从创建到销毁的过程，就是生命周期。

每一个组件或者实例都会经历一个完整的生命周期，总共分为三个阶段：初始化、运行中、销毁。

实例、组件通过new Vue() 创建出来之后会初始化事件和生命周期，然后就会执行beforeCreate钩子函数，这个时候，数据还没有挂载呢，只是一个空壳，无法访问到数据和真实的dom，一般不做操作。

挂载数据，绑定事件等等，然后执行created函数，这个时候已经可以使用到数据，也可以更改数据，在这里更改数据不会触发updated函数，在这里可以在渲染前倒数第二次更改数据的机会，不会触发其他的钩子函数，一般可以在这里做初始数据的获取 。

接下来开始找实例或者组件对应的模板，编译模板为虚拟dom放入到render函数中准备渲染，然后执行beforeMount钩子函数，在这个函数中虚拟dom已经创建完成，马上就要渲染，在这里也可以更改数据，不会触发updated，在这里可以在渲染前最后一次更改数据的机会，不会触发其他的钩子函数，一般可以在这里做初始数据的获取。

接下来开始render，渲染出真实dom，然后执行mounted钩子函数，此时，组件已经出现在页面中，数据、真实dom都已经处理好了，事件都已经挂载好了，可以在这里操作真实dom等事情。

当组件或实例的数据更改之后，会立即执行beforeUpdate，然后vue的虚拟dom机制会重新构建虚拟dom与上一次的虚拟dom树利用diff算法进行对比之后重新渲染，一般不做什么事儿。

当更新完成后，执行updated，数据已经更改完成，dom也重新render完成，可以操作更新后的虚拟dom。

当经过某种途径调用$destroy方法后，立即执行beforeDestroy，一般在这里做一些善后工作，例如清除计时器、清除非指令绑定的事件等等。

组件的数据绑定、监听...去掉后只剩下dom空壳，这个时候，执行destroyed，在这里做善后工作也可以。

当我们使用 **keep-alive** 的时候，还有两个钩子函数，分别是 **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 actived 钩子函数。

**父子组件生命周期顺序**

**1. 加载渲染过程**

- 同步引入时生命周期顺序为： 父组件的beforeCreate、created、beforeMount --> 所有子组件的beforeCreate、created、beforeMount --> 所有子组件的mounted --> 父组件的mounted 总结：父组件先创建，然后子组件创建；子组件先挂载，然后父组件挂载 *若有孙组件呢？* 父组件先beforeCreate => created => beforeMount , 然后子组件开始beforeCreate => created => beforeMount ，然后孙组件beforeCreate => created => beforeMount => mounted，孙组件挂载完成了，子组件mounted，父组件再mounted
- 异步引入时生命周期顺序为： 父组件的beforeCreate、created、beforeMount、mounted --> 子组件的beforeCreate、created、beforeMount、mounted

  总结：父组件创建，父组件挂载；子组件创建，子组件挂载。

**2. 子组件更新过程** 父beforeUpdate->子beforeUpdate->子updated->父updated

**3.父组件更新过程** 父beforeUpdate->父updated

**4.销毁过程** 父beforeDestroy->子beforeDestroy->子destroyed->父destroyed

**组件参数传递**

- 父子组件间通信
    - 第一种方法是子组件通过 props 属性来接受父组件的数据，然后父组件在子组件上注册监听事件，子组件通过 emit 触发事件来向父组件发送数据。
    - 第二种是通过 ref 属性给子组件设置一个名字。父组件通过 `$refs` 组件名来获得子组件，子组件通过 `$parent` 获得父组件，这样也可以实现通信。
    - 第三种是只要在父组件中通过 provider 来提供变量，那么不论子组件有多深都可以通过 inject 向父组件 provider 中注入数据。
- 兄弟组件间通信
    - 第一种是使用 eventBus 的方法，它的本质是通过创建一个空的 Vue 实例来作为消息传递的对象，组件引入这个实例，通信的组件通过在这个实例上监听和触发事件，来实现消息的传递。
    - 第二种是通过 `$parent.$refs` 来获取到兄弟组件，也可以进行通信。

- 任意组件之间

  使用 eventBus ，其实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。

  如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这一些方法可能不利于项目的维护。这个时候可以使用 **vuex** ，vuex 的思想就是将这一些公共的数据抽离出来，将它作为一个全局的变量来管理，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

**data为什么是函数**

vue 是由组件组成的，data 定义了组件中的数据。组件是可复用的`vue`实例，一个组件被创建好之后，就可能被用在各个地方，而组件不管被复用了多少次，组件中的`data`数据都应该是相互隔离，互不影响的，基于这一理念，组件每复用一次，`data`数据就应该被复制一次，之后，当某一处复用的地方组件内`data`数据被改变时，其他复用地方组件的`data`数据不受影响。javascipt只有函数构成作用域，data是一个函数时，每个组件实例都有自己的作用域，每个实例相互独立,不会相互影响。所以可以说这都是因为js本身的特性带来的，跟vue本身设计无关

**computed 与 watchd 的差异**

1. computed 是计算一个新的属性，并将该属性挂载到 Vue 实例上，而 watch 是监听已经存在且已挂载到 Vue 实例上的数据，所以用 watch 同样可以监听 computed 计算属性的变化。
2. computed 本质是一个惰性求值的观察者，具有缓存性，只有当依赖变化后，第一次访问 computed 属性，才会计算新的值。而 watch 则是当数据发生变化便会调用执行函数。
3. 从使用场景上说，computed 适用一个数据被多个数据影响，而 watch 适用一个数据影响多个数据。

**v-if 与 v-show 的区别**

`v-if` **VS** `v-show`

`v-if` 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

`v-if` 也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

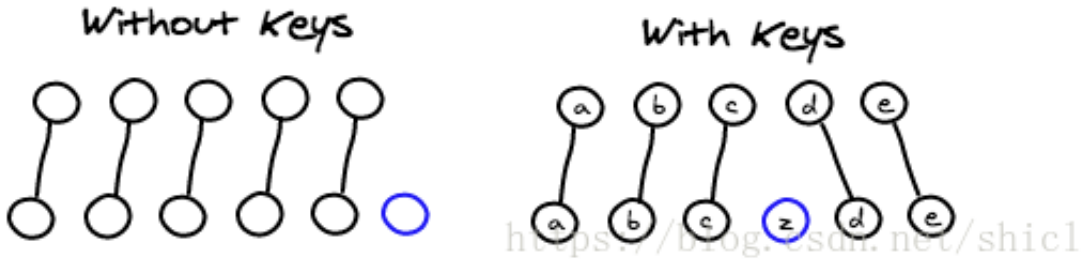
相比之下，`v-show` 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。

一般来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，如果需要非常频繁地切换，则使用 `v-show` 较好；如果在运行时条件很少改变，则使用 `v-if` 较好。

**v-for key属性的作用**

> vue中在进行列表渲染的时候，会默认遵守**就地复用**策略：（与diff算法相关）
>
> 当在进行列表渲染的时候，vue会直接对已有的标签进行复用，不会整个的将所有的标签全部重新删除和创建,只会重新渲染数据，然后再创建新的元素直到数据渲染完为止

key的作用主要是为了高效的更新虚拟DOM。另外vue中在使用相同标签名元素的过渡切换时，也会使用到key属性，其目的也是为了让vue可以区分它们，否则vue只会替换其内部属性而不会触发过渡效果。（比如没有key时在一群相同节点中插入节点会依次改变指向最后创建新节点，有了key则可以正确的找到区间插入新的节点）





**keepalive**

我们在平时开发中，总有部分组件没必要多次 Init,我们需要将组件进行持久化，使组件状态维持不变，在下一次展示时，也不会重新init keepalive 音译过来就是保持活跃，所以在vue中我们可以使用keepalive来进行组件缓存 基本使用

```
//keepalive包含的组件会被进行缓存
<keep-alive>
  <component />
</keep-alive>
```

上面提到被keepalive包含的组件不会被再次init，也就意味着不会重新走生命周期函数，但是平时工作中很多业务场景是希望我们缓存的组件在再次渲染时能做一些事情，vue为keepalive提供了两个额外的hook,

- activated 当keepalive包含的组件再次渲染的时候触发
- deactived 当keepalive包含的组件销毁的时候触发 keepalive可以接收三个属性作为参数进行匹配对应的组件进行缓存
- include 包含的组件
- exclude 排除的组件
- max 缓存组件的最大值 其中include,exclude可以为字符，数组，以及正则表达式max类型为字符或者数字 代码理解

```
//只缓存组件name为a或者b的组件
<keep-alive include="a,b">
 <component :is="currentView"/>
</keep-alive>
//组件名为c的组件不缓存
<keep-alive exclude="c">
 <component :is="currentView"/>
</keep-alive>
// 如果同时使用include,exclude,那么exclude优先于include， 下面的例子也就是只缓存a组件
<keep-alive include="a,b" exclude="b">
 <component :is="currentView"/>
</keep-alive>
```

```
// 如果缓存的组件超过了max设定的值5，那么将删除第一个缓存的组件
<keep-alive exclude="c" max="5">
 <component :is="currentView"/>
</keep-alive>
```

配合router使用

```
//意思就是$router.meta.keepalive值为真是将组件进行缓存
<keep-alive>
  <router-view v-if="$router.meta.keepalive"></router-view>
</keep-alive>
//router配置
new Router({
  router:[
    {
        name:'a',
        path:'/a',
        component:A,
        meta:{
          keepAlive:true
        },
        {
          name:'b',
          path:'/b',
          component:B
        }
      }
    ]
  })
```

总结 keepalive是一个抽象组件，缓存vnode，缓存的组件不会被mounted，为此提供activated 和 deactivated 钩子函数, 使用props max 可以控制缓存组件个数

**nextTick**

定义：在下次 DOM 更新循环结束之后执行延迟回调。简单的理解是当数据更新了，在dom中渲染后，自动执行该函数。

应用场景：

- 在Vue生命周期的`created()`钩子函数进行的DOM操作一定要放在`Vue.nextTick()`的回调函数中

  在`created()`钩子函数执行的时候DOM 其实并未进行任何渲染，而此时进行DOM操作无异于徒劳，所以此处一定要将DOM操作的js代码放进`Vue.nextTick()`的回调函数中。与之对应的就是`mounted()`钩子函数，因为该钩子函数执行时所有的DOM挂载和渲染都已完成，此时在该钩子函数中进行任何DOM操作都不会有问题 。

- 在数据变化后要执行的某个操作，而这个操作需要使用随数据改变而改变的DOM结构的时候，这个操作都应该放进`Vue.nextTick()`的回调函数中。

- 在使用某个第三方插件时，希望在vue生成的某些dom动态发生变化时重新应用该插件，也会用到该方法，这时候就需要在 $nextTick 的回调函数中执行重新应用插件的方法。

使用原理：

Vue是异步执行dom更新的，一旦观察到数据变化，Vue就会开启一个队列，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。如果这个watcher被触发多次，只会被推送到队列一次。这种缓冲行为可以有效的去掉重复数据造成的不必要的计算和DOM操作。而在下一个事件循环时，Vue会清空队列，并进行必要的DOM更新。 当你设置 vm.someData = 'new value'，DOM 并不会马上更新，而是在异步队列被清除，也就是下一个事件循环开始时执行更新时才会进行必要的DOM更新。如果此时你想要根据更新的 DOM 状态去做某些事情，就会出现问题。为了在数据变化之后等待 Vue 完成更新 DOM ，可以在数据变化之后立即使用 Vue.nextTick(callback) 。这样回调函数在 DOM 更新完成后就会调用。

## vue-router

> **单页面**，即 第一次进入页面的时候会请求一个html文件，刷新清除一下。切换到其他组件，此时路径也相应变化，但是并没有新的html文件请求，页面内容也变化了。
>
> 原理是：JS会感知到url的变化，通过这一点，可以用js动态的将当前页面的内容清除掉，然后将下一个页面的内容挂载到当前页面上，这个时候的路由不是后端来做了，而是前端来做，判断页面到底是显示哪个组件，清除不需要的，显示需要的组件。这种过程就是单页应用，每次跳转的时候不需要再请求html文件了。
>
> **多页面**，即 每一次页面跳转的时候，后台服务器都会给返回一个新的html文档，这种类型的网站也就是多页网站，也叫做多页应用。 原理是：传统的页面应用，是用一些超链接来实现页面切换和跳转的
>
> 总结： 单页面模式：相对比较有优势，无论在用户体验还是页面切换的数据传递、页面切换动画，都可以有比较大的操作空间
>
> 多页面模式：比较适用于页面跳转较少，数据传递较少的项目中开发，否则使用cookie，localstorage进行数据传递，是一件很可怕而又不稳定的无奈选择
>
> 在vue中便是依靠 vue-route 实现单页面跳转

**原理**

原理核心就是**更新视图但不重新请求页面**。

vue-router实现单页面路由跳转，提供了两种方式：hash模式、history模式（、abstract模式），根据mode参数来决定采用哪一种方式。

> 路由模式
>
> vue-router 提供了三种运行模式： ● hash: 使用 URL hash 值来作路由。默认模式。 ● history: 依赖 HTML5 History API 和服务器配置。查看 HTML5 History 模式。 ● abstract: 支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的API，路由会自动强制进入这个模式。

- **Hash模式**

hash模式背后的原理是`onhashchange`事件，可以在`window`对象上监听这个事件：

```
window.onhashchange = function(event){
    console.log(event.oldURL, event.newURL);
    let hash = location.hash.slice(1);
    document.body.style.color = hash;
}
```

上面的代码可以通过改变hash来改变页面字体颜色，虽然没什么用，但是一定程度上说明了原理。

更关键的一点是，因为hash发生变化的url都会被浏览器记录下来，从而你会发现浏览器的前进后退都可以用了，同时点击后退时，页面字体颜色也会发生变化。这样一来，尽管浏览器没有请求服务器，但是页面状态和url一一关联起来，后来人们给它起了一个霸气的名字叫**前端路由**，成为了单页应用标配。

- **History模式**

随着history api的到来，前端路由开始进化了，前面的**hashchange，你只能改变#后面的url片段，而history api则给了前端完全的自由**

history api可以分为两大部分：切换和修改

**（1）切换历史状态**

包括`back、forward`、`go`三个方法，对应浏览器的前进，后退，跳转操作，有同学说了，(谷歌)浏览器只有前进和后退，没有跳转，嗯，在前进后退上长按鼠标，会出来所有当前窗口的历史记录，从而可以跳转(也许叫跳更合适)：

```
history.go(-2);//后退两次
history.go(2);//前进两次
history.back(); //后退
hsitory.forward(); //前进
```

**（2）修改历史状态**

包括了`pushState、replaceState`两个方法，这两个方法接收三个参数：stateObj，title，url

```
history.pushState({color:'red'}, 'red', 'red')

window.onpopstate = function(event){
    console.log(event.state)
    if(event.state && event.state.color === 'red'){
        document.body.style.color = 'red';
    }
}

history.back();

history.forward();
```

通过pushstate把页面的状态保存在state对象中，当页面的url再变回这个url时，可以通过event.state取到这个state对象，从而可以对页面状态进行还原，这里的页面状态就是页面字体颜色，其实滚动条的位置，阅读进度，组件的开关的这些页面状态都可以存储到state的里面。

通过history api，我们丢掉了丑陋的#，但是它也有个毛病：

不怕前进，不怕后退，就怕**刷新**，**f5**，（如果后端没有准备的话），因为刷新是实实在在地去请求服务器的。

在hash模式下，前端路由修改的是#中的信息，而浏览器请求时是不带它玩的，所以没有问题。但是在history下，你可以自由的修改path，当刷新时，如果服务器中没有相应的响应或者资源，会分分钟刷出一个404来。

**（3）popstate实现history路由拦截，监听页面返回事件**

当活动历史记录条目更改时，将触发popstate事件。

1、如果被激活的历史记录条目是通过对 history.pushState() 的调用创建的，或者受到对 history.replaceState() 的调用的影响，popstate事件的state属性包含历史条目的状态对象的副本。

2、需要注意的是调用 history.pushState() 或 history.replaceState() 用来在浏览历史中添加或修改记录，不会触发popstate事件；

只有在做出浏览器动作时，才会触发该事件，如用户点击浏览器的回退按钮（或者在Javascript代码中调用history.back()）

- ~~**abstract模式**~~

abstract模式是使用一个不依赖于浏览器的浏览历史虚拟管理后端。 根据平台差异可以看出，在 Weex 环境中只支持使用 abstract 模式。 不过，vue-router 自身会对环境做校验，如果发现没有浏览器的 API，vue-router 会自动强制进入 abstract 模式，所以在使用 vue-router 时只要不写 mode 配置即可，默认会在浏览器环境中使用 hash 模式，在移动端原生环境中使用 abstract 模式。（当然，你也可以明确指定在所有情况下都使用 abstract 模式）。

**导航守卫**

- 全局的钩子函数 beforeEach 和 afterEach beforeEach 有三个参数，to 代表要进入的路由对象，from 代表离开的路由对象。next 是一个必须要执行的函数，如果不传参数，那就执行下一个钩子函数，如果传入 false，则终止跳转，如果传入一个路径，则导航到对应的路由，如果传入 error ，则导航终止，error 传入错误的监听函数。
- 单个路由独享的钩子函数 beforeEnter，它是在路由配置上直接进行定义的。
- 组件内的导航钩子主要有这三种：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave。它们是直接在路由组件内部直接进行定义的。

**$router 和 $route 的区别**

**router**是VueRouter的一个实例，通过Vue.use(VueRouter)和VueRouter构造函数得到一个router的实例对象，所以它是一个全局对象，包含了所有的路由的许多关键对象和属性。

举例：history对象

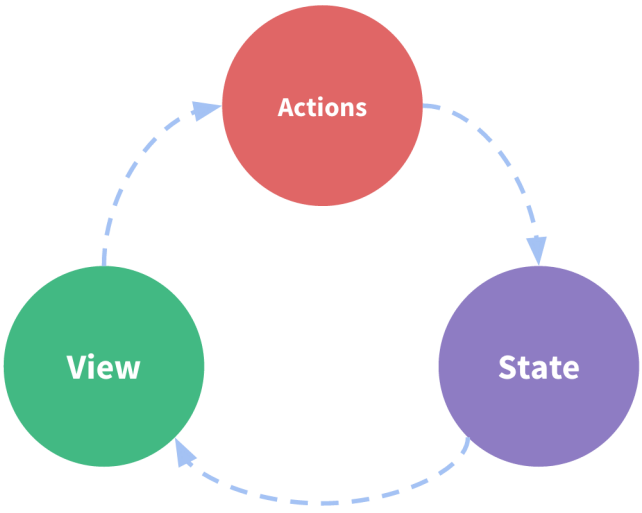
$router.push({path:'home'});本质是向history栈中添加一个路由，在我们看来是切换路由，但本质在添加一个history记录

$router.replace({path:'home'});//替换路由，没有历史记录

**route**是一个跳转的路由对象，每一个路由都会有一个route对象，是一个局部的对象，可以获取对应的name, path, params, query等。

- $route.path 字符串，等于当前路由对象的路径，会被解析为绝对路径，如 `"/home/news"` 。
- $route.params 对象，包含路由中的动态片段和全匹配片段的键值对
- $route.query 对象，包含路由中查询参数的键值对。例如，对于 `/home/news/detail/01?favorite=yes` ，会得到`$route.query.favorite == 'yes'` 。
- $route.router 路由规则所属的路由器（以及其所属的组件）。
- $route.matched 数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象。
- $route.name 当前路径的名字，如果没有使用具名路径，则名字为空。

## vuex

> Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式**，换句话说就是用于管理页面的数据状态、提供统一数据操作的生态系统，相当于数据库mongoDB，MySQL等，任何组件都可以存取仓库中的数据。
>
> Vuex规定所有的数据必须通过action—>mutaion—>state这个流程进行来改变状态的。再结合Vue的数据视图双向绑定实现页面的更新。统一页面状态管理，可以让复杂的组件交互变的简单清晰（，同时在调试时也可以通过DEVtools去查看状态。）
>
> 
> 
>
> 在当前前端的spa模块化项目中不可避免的是某些变量需要在全局范围内引用，此时父子组件的传值，兄弟组件间的传值成了我们需要解决的问题。虽然vue中提供了props（父传子）commit（子传父）兄弟间也可以用localstorage和sessionstorage。但是这种方式在项目开发中带来的问题比他解决的问题（难管理，难维护，代码复杂，安全性低）更多。vuex的诞生也是为了解决这些问题，从而大大提高我们vue项目的开发效率。

vue中的data 、methods、computed，可以实现响应式。与之类似，vuex中也有四个属性值：state、getters、mutations、actions。

在没有actions的情况下vuex与vue的类比：

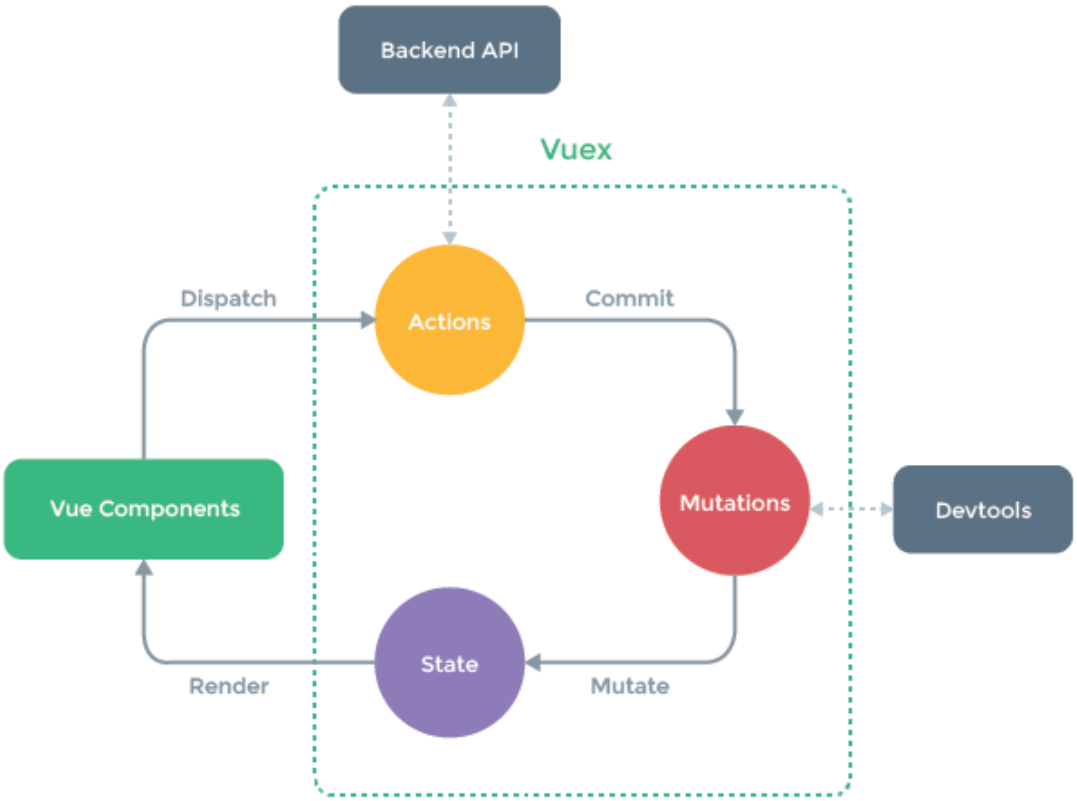
- 数据：state --> data
- 获取数据：getters --> computed
- 更改数据：mutations --> methods

**视图通过点击事件，触发mutations中方法，可以更改state中的数据，一旦state数据发生更改，getters把数据反映到视图。**

那么actions,可以理解处理异步，而单纯多加的一层。

既然提到了mutions actions这时候 就不得不提**commit，dispatch**这两个有什么作用呢?

在vue例子中，通过click事件，触发methods中的方法。当存在异步时，而在vuex中需要dispatch来触发actions中的方法，actions中的commit可以触发mutations中的方法。同步，则直接在组件中commit触发vuex中mutations中的方法。




---

## axio

Axios 是一个基于 promise 的 HTTP 库，简单的讲就是可以发送get、post请求。

**Axios特性：**

1. 可以在浏览器中发送 XMLHttpRequests
2. 可以在 node.js 发送 http 请求
3. 支持 Promise API
4. 拦截请求和响应

5. 转换请求数据和响应数据
6. 能够取消请求
7. 自动转换 JSON 数据
8. 客户端支持保护安全免受 XSRF 攻击

---

## ● Webpack

webpack是一个前端模块化方案，更侧重模块打包，我们可以把开发中的所有资源（图片、js文件、css文件等）都看成模块，通过loader（加载器）和plugins（插件）对资源进行处理，打包成符合生产环境部署的前端资源，这样的好处是可以减少用户请求的资源大小和数量。

我当时使用 webpack 的一个最主要原因是为了简化页面依赖的管理，并且通过将其打包为一个 文件来降低页面加载时请求的资源数。

Webpack 具有四个核心的概念，分别是 Entry（入口）、Output（输出）、Loader（加载器） 和 Plugins （插件）。

Entry 是 webpack 的入口起点，它指示 webpack 应该从哪个模块开始着手，来作为其构建内部依赖图的开始。

Output 属性告诉 webpack 在哪里输出它所创建的打包文件，也可指定打包文件的名称，默认位置为 ./dist。

loader 可以理解为 webpack 的编译器，它使得 webpack 可以处理一些非 JavaScript 文件。 在对 loader 进行配置的时候，test 属性，标志有哪些后缀的文件应该被处理，是一个正则表达式。use 属性，指定 test 类型的文件应该使用哪个 loader 进行预处理。常用的 loader 有 css-loader、style-loader 等。

Plugins可以用于执行范围更广的任务，包括打包、优化、压缩、搭建服务器等等，要使用一个插件， 一般是先使用 npm 包管理器进行安装，然后在配置文件中引入，最后将其实例化后传递给 plugins 数组属性。

**常用Loaders**

```
less-loader, sass-loader
    处理样式

url-loader, file-loader
    两个都必须用上。否则超过大小限制的图片无法生成到目标文件夹中

babel-loader, babel-preset-es2015, babel-preset-react
    js处理，转码

expose?*
    eg:
    {
        test: require.resolve('react'),
        loader: 'expose?React'
    }
```

```
expose-loader
    将js模块暴露到全局，如果
```

**常用插件Plugin**

- config类

```
    NormalModuleReplacementPlugin
        匹配resourceRegExp，替换为newResource

    ContextReplacementPlugin
        替换上下文的插件

    IgnorePlugin
        不打包匹配文件

    PrefetchPlugin
        预加载的插件，提高性能

    ResolverPlugin
        替换上下文的插件

    ContextReplacementPlugin
        替换上下文的插件
```

- optimize

```
    DedupePlugin
        打包的时候删除重复或者相似的文件

    MinChunkSizePlugin
        把多个小模块进行合并，以减少文件的大小

    LimitChunkCountPlugin
        限制打包文件的个数

    MinChunkSizePlugin
        根据chars大小，如果小于设定的最小值，就合并这些小模块，以减少文件的大小

    OccurrenceOrderPlugin
        根据模块调用次数，给模块分配ids，常被调用的ids分配更短的id，使得ids可预测，降低文件大小，该模块推荐使用

    UglifyJsPlugin
        压缩js
```

```
    ngAnnotatePlugin
        使用ng-annotate来管理AngularJS的一些依赖

    CommonsChunkPlugin
        多个 html共用一个js文件(chunk)
```

- dependency injection

```
    DefinePlugin
        定义变量，一般用于开发环境log或者全局变量

    ProvidePlugin
        自动加载模块，当配置（$:'jquery'）例如当使用$时，自动加载jquery
```

- other

```
    HotModuleReplacementPlugin
        模块热替换,如果不在dev-server模式下，需要记录数据，recordPath，生成每个模块的热更新模块

    ProgressPlugin
        编译进度

    NoErrorsPlugin
        报错但不退出webpack进程

    HtmlWebpackPlugin
        生成html
```

# • Babel

javascript在不断的发展，各种新的标准和提案层出不穷，但是由于浏览器的多样性，导致可能几年之内都无法广泛普及，babel可以让你提前使用这些语言特性，他是一种用途很多的javascript编译器，他把最新版的javascript编译成当下可以执行的版本，简言之，利用babel就可以让我们在当前的项目中随意的使用这些新最新的es6，甚至es7的语法。说白了就是把各种javascript千奇百怪的语言统统专为浏览器可以认识的语言。

babel的核心概念就是利用一系列的plugin来管理编译案列，通过不同的plugin，他不仅可以编译es6的代码，还可以编译react JSX语法或者别的语法，甚至可以使用还在提案阶段的es7的一些特性，这就足以看出她的可扩展性。

---

- **NPM**

---

- **Echarts**

> 百度团队开发的，提供了一些直观，易用的交互方式以便于对展示数据进行挖掘.提取.修正或整合，拥有互动图形用户界面的深度数据可视化工具，支持的图表类型有折线图（区域图）、柱状图（条状图）、散点图（气泡图）、K线图、饼图（环形图）、雷达图（填充雷达图）、和弦图、力导向布局图、地图、仪表盘、漏斗图、事件河流图等12类图表

- **简述数据可视化技术**

1. 什么是数据可视化技术

   借助图形化的数段，清晰有效的传递和沟通信息，以视觉的方式展现数据，便于用户的认知，偏于图表的样式，相对于文字说明更加直观

   - 科学可视化（出现最早，最成熟）
     - 处理科学数据，面向科学和工程数据方面，研究带有空间坐标和几何信息的三维空间，如何呈现数据中的几何特征
     - 主要面向自然科技中产生数据的建模操作和处理
     - 应用于医疗（透析，CT），科研，航天，天气，生物等技术
   - 信息可视化（更常见，接触更多）
     - 科学可视化演变而来，主要处理非结构化，非几何的数据
     - 金融交易，社交网络，文本数据展示
     - 减少视觉混淆对有用数据的干扰，把无用的数据过滤掉，而非简单信息的堆叠（数据加工，提取可用信息）
     - 更倾向于展示信息
   - 可视化分析（前两者的结合）
     - 分析数据导向进行展示，需要了解具体的事物逻辑

2. 数据可视化技术优点
   - 分析出数据的趋势
   - 进行精准的广告投放
   - 信息快人一步，优先获取信息就有更大的优势

- **echarts的基本用法**

1. 初始化类

   Html里面创建一个id为box1的div，并初始化echarts绘图实例

   ```
   var myChart = echarts.init(document.getElementById('box1'))
   ```

2. 样式配置option

- title： 标题
- tooltip： 鼠标悬停气泡
- xAxis : 配置横轴类别，type类型为category类别

- series：销量数据，data参数与横轴一一对应，如果想调样式，也可以简单调整，比如每个条形图的颜色可以通过函数进行数组返回渲染

3. 渲染图展示表

```
myChart.setOption(option);
```

4. 图表自适应

   echart图表本身是提供了一个`resize`的函数的，常见有两个方法实现。

   - 给window添加resize的事件监听器

     ```
     window.addEventListener('resize', function() {
       myChart.resize()
     })
     ```

   - 用jquery的 resize() 事件图表div，当发生resize事件的时候，让其触发echart的resize事件，重绘canvas。

   > 注：jquery有resize()事件，但直接调用没有起作用，引入jquery.ba-resize.js文件

   ```
     <div class="chart">
         <div class="col-md-3" style="width:73%;height:270px"
   id="chartx">   </div>
     </div>
     <script src="/static/assets/scripts/jquery.ba-resize.js">
   </script>
     <script>
       var myChartx =
   echarts.init(document.getElementById('chartx'));
       $('.chart').resize(function(){
         myChartx.resize();
       })
   </script>
   ```

- **echarts使用中的一点问题**

  **切换其他组件统计图时，出现卡顿问题如何解决**

  1. 原因：每一个图例在没有数据的时候它会创建一个定时器去渲染气泡，页面切换后，echarts图例是销毁了，但是这个echarts的实例还在内存当中，同时它的气泡渲染定时器还在运行。这就导致Echarts占用CPU高，导致浏览器卡顿，当数据量比较大时甚至浏览器崩溃
  2. 解决方法：在mounted()方法和destroy()方法之间加一个beforeDestroy()方法释放该页面的chart资源，clear()方法则是清空图例数据，不影响图例的resize，而且能够释放内存，切换的时候就顺畅了

```
beforeDestroy () {
  this.chart.clear()
}
```

---

# • Node.js

> 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. 轻量、可伸缩，适于实时数据交互应用
> 6. 单进程，单线程

对Node 的优点和缺点提出了自己的看法：

- （优点）因为Node 是基于事件驱动和无阻塞的，轻量的，所以非常适合处理高并发请求， 用于搭建高性能的web服务器。 此外，与Node 代理服务器交互的客户端代码是由javascript 语言编写的， 因此客户端和服务器端都用同一种语言编写，这是非常美妙的事情。
- （缺点）Node 是一个相对新的开源项目，所以不太稳定，它总是一直在变， 而且缺少足够多的第三方库支持。（目前来看已经比较丰富）

## Express

express是一个基于node.js平台的极简，灵活的web应用开发框架，它提供一系列强大的特征，帮助你创建各种web和移动设备应用

express框架核心特征：

1. 可以设置中间件来响应HTTP请求
2. 定义了路由表用于执行不同的HTTP请求动作（url=资源）映射
3. 可以通过向模板传递参数来动态渲染HTML页面

---

# • Git

- ## git 与 svn 的区别

  git 和 svn 最大的区别在于 git 是分布式的，而 svn 是集中式的。因此我们不能再离线的情况下使用 svn。如果服务器出现问题，我们就没有办法使用 svn 来提交我们的代码。

  svn 中的分支是整个版本库的复制的一份完整目录，而 git 的分支是指针指向某次提交，因此 git 的分支创建更加开销更小并且分支上的变化不会影响到其他人。svn 的分支变化会影响到所有的人。

  svn 的指令相对于 git 来说要简单一些，比 git 更容易上手。

- **git 常见命令**

```
git init // 新建 git 代码库
git add // 添加指定文件到暂存区
git rm // 删除工作区文件，并且将这次删除放入暂存区
git commit -m [message] // 提交暂存区到仓库区
git branch // 列出所有分支
git checkout -b [branch] // 新建一个分支，并切换到该分支
git status // 显示有变更的文件
```

- **git 提交冲突**

  开发过程中，我们都有自己的特性分支，所以冲突发生的并不多，但也碰到过。诸如公共类的公共方法，我和别人同时修改同一个文件，他提交后我再提交就会报冲突的错误。 发生冲突，在IDE里面一般都是对比本地文件和远程分支的文件，然后把远程分支上文件的内容手工修改到本地文件，然后再提交冲突的文件使其保证与远程分支的文件一致，这样才会消除冲突，然后再提交自己修改的部分。特别要注意下，修改本地冲突文件使其与远程仓库的文件保持一致后，需要提交后才能消除冲突，否则无法继续提交。必要时可与同事交流，消除冲突。 发生冲突，也可以使用命令。

  - 通过git stash命令，把工作区的修改提交到栈区，目的是保存工作区的修改；
  - 通过git pull命令，拉取远程分支上的代码并合并到本地分支，目的是消除冲突；
  - 通过git stash pop命令，把保存在栈区的修改部分合并到最新的工作空间中；

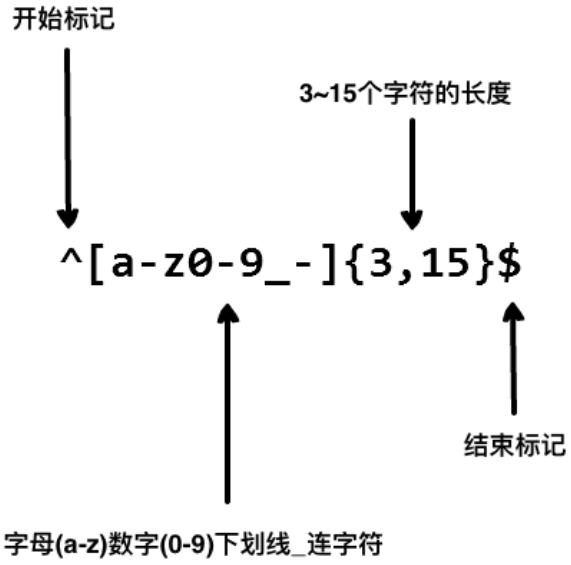
---

- **正则表达式**

  - **什么是正则表达式?**

    一个正则表达式是一种从左到右匹配主体字符串的模式。 “Regular expression”这个词比较拗口，我们常使用缩写的术语“regex”或“regexp”。 正则表达式可以从一个基础字符串中根据一定的匹配模式替换文本中的字符串、验证表单、提取字符串等等。

    想象你正在写一个应用，然后你想设定一个用户命名的规则，让用户名包含字符、数字、下划线和连字符，以及限制字符的个数，好让名字看起来没那么丑。 我们使用以下正则表达式来验证一个用户名：




以上的正则表达式可以接受 `john_doe`、`jo-hn_doe`、`john12_as`。但不匹配`Jo`，因为它包含了大写的字母而且太短了。

- **1. 基本匹配**

  正则表达式其实就是在执行搜索时的格式，它由一些字母和数字组合而成。例如：一个正则表达式 `the`，它表示一个规则：由字母`t`开始，接着是`h`，再接着是`e`。

  ```
  "the" => The fat cat sat on the mat.
  ```

[在线练习]

正则表达式`123`匹配字符串`123`。它逐个字符的与输入的正则表达式做比较。

正则表达式是大小写敏感的，所以`The`不会匹配`the`。

```
"The" => The fat cat sat on the mat.
```

[在线练习]

- **2. 元字符**

  正则表达式主要依赖于元字符。元字符不代表他们本身的字面意思，他们都有特殊的含义。一些元字符写在方括号中的时候有一些特殊的意思。以下是一些元字符的介绍：

| 元字符 | 描述 |
| --- | --- |
| . | 句号匹配任意单个字符除了换行符。 |
| [ ] | 字符种类。匹配方括号内的任意字符。 |
| [^ ] | 否定的字符种类。匹配除了方括号里的任意字符 |
| * | 匹配>=0个重复的在*号之前的字符。 |
| + | 匹配>=1个重复的+号前的字符。 |
| ? | 标记?之前的字符为可选. |
| {n,m} | 匹配num个大括号之前的字符或字符集 (n <= num <= m). |
| (xyz) | 字符集，匹配与 xyz 完全相等的字符串. |
| \| | 或运算符，匹配符号前或后的字符. |
| \ | 转义字符,用于匹配一些保留的字符 `[ ] ( ) { } . * + ? ^ $ \ \|` |
| ^ | 从开始行开始匹配. |
| $ | 从末端开始匹配. |

- **3. 简写字符集**

正则表达式提供一些常用的字符集简写。如下:

| 简写 | 描述 |
| --- | --- |
| . | 除换行符外的所有字符 |
| \w | 匹配所有字母数字，等同于 `[a-zA-Z0-9_]` |
| \W | 匹配所有非字母数字，即符号，等同于： `[^\w]` |
| \d | 匹配数字： `[0-9]` |
| \D | 匹配非数字： `[^\d]` |
| \s | 匹配所有空格字符，等同于： `[\t\n\f\r\p{Z}]` |
| \S | 匹配所有非空格字符： `[^\s]` |
| \f | 匹配一个换页符 |
| \n | 匹配一个换行符 |
| \r | 匹配一个回车符 |
| \t | 匹配一个制表符 |
| \v | 匹配一个垂直制表符 |
| \p | 匹配 CR/LF（等同于 `\r\n`），用来匹配 DOS 行终止符 |

- **4. 零宽度断言（前后预查）**

  先行断言和后发断言都属于**非捕获簇**（不捕获文本，也不针对组合计进行计数）。先行断言用于判断所匹配的格式是否在另一个确定的格式之前，匹配结果不包含该确定格式（仅作为约束）。

  例如，我们想要获得所有跟在 `$` 符号后的数字，我们可以使用正后发断言 `(?<=\$)[0-9\.]*`。这个表达式匹配 `$` 开头，之后跟着 `0,1,2,3,4,5,6,7,8,9,.` 这些字符可以出现大于等于 0 次。

  零宽度断言如下：

| 符号 | 描述 |
| --- | --- |
| ?= | 正先行断言-存在 |
| ?! | 负先行断言-排除 |
| ?<= | 正后发断言-存在 |
| ?<! | 负后发断言-排除 |

- **4.1 `?=...` 正先行断言**

`?=...` 正先行断言，表示第一部分表达式之后必须跟着 `?=...` 定义的表达式。

返回结果只包含满足匹配条件的第一部分表达式。 定义一个正先行断言要使用 `()`。在括号内部使用一个问号和等号： `(?=...)`。

正先行断言的内容写在括号中的等号后面。 例如，表达式 `(T|t)he(?=\sfat)` 匹配 `The` 和 `the`，在括号中我们又定义了正先行断言 `(?=\sfat)` ，即 `The` 和 `the` 后面紧跟着 `(空格)fat`。

```
"(T|t)he(?=\sfat)" => The fat cat sat on the mat.
```

在线练习

- **4.2 `?!...` 负先行断言**

负先行断言 `?!` 用于筛选所有匹配结果，筛选条件为 其后不跟随着断言中定义的格式。 `正先行断言` 定义和 `负先行断言` 一样，区别就是 `=` 替换成 `!` 也就是 `(?!...)`。

表达式 `(T|t)he(?!\sfat)` 匹配 `The` 和 `the`，且其后不跟着 `(空格)fat`。

```
"(T|t)he(?!\sfat)" => The fat cat sat on the mat.
```

在线练习

- **4.3 `?<= ...` 正后发断言**

正后发断言 记作`(?<=...)` 用于筛选所有匹配结果，筛选条件为 其前跟随着断言中定义的格式。例如，表达式 `(?<=(T|t)he\s)(fat|mat)` 匹配 `fat` 和 `mat`，且其前跟着 `The` 或 `the`。

```
"(?<=(T|t)he\s)(fat|mat)" => The fat cat sat on the mat.
```

在线练习

- **4.4 `?<!...` 负后发断言**

负后发断言 记作 `(?<!...)` 用于筛选所有匹配结果，筛选条件为 其前不跟随着断言中定义的格式。 例如，表达式 `(?<!(T|t)he\s)(cat)` 匹配 `cat`，且其前不跟着 `The` 或 `the`。

```
"(?<!(T|t)he\s)(cat)" => The cat sat on cat.
```

在线练习

- **5. 标志**

标志也叫模式修正符，因为它可以用来修改表达式的搜索结果。 这些标志可以任意的组合使用，它也是整个正则表达式的一部分。

| 标志 | 描述 |
| --- | --- |
| i | 忽略大小写。 |
| g | 全局搜索。 |
| m | 多行修饰符：锚点元字符 `^` `$` 工作范围在每行的起始。 |

- **6. 贪婪匹配与惰性匹配 (Greedy vs lazy matching)**

正则表达式默认采用贪婪匹配模式，在该模式下意味着会匹配尽可能长的子串。我们可以使用 `?` 将贪婪匹配模式转化为惰性匹配模式。

```
"/(.*at)/" => The fat cat sat on the mat.
```

在线练习

```
"/(.*?at)/" => The fat cat sat on the mat.
```

在线练习

## 常用正则表达式

```
// (1) 匹配 16 进制颜色值
var regex = /#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g;
// (2) 匹配日期，如 yyyy-mm-dd 格式
var regex = /^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/;
// (3) 匹配 qq 号
var regex = /^[1-9][0-9]{4,10}$/g;
// (4) 手机号码正则
var regex = /^1[34578]\d{9}$/g;
// (5) 用户名正则
var regex = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4,16}$/;
```

使用正则表达式将浮点数点左边的数每三位添加一个逗号

```
function format(number) {
return number && number.replace(/(?!^)(?=(\d{3})+\.)/g, ",");}
```

```
'1000000.00'.replace(/(\d)(?=(?:\d{3})+\.)/g, '$1,');
```

字符串驼峰和下划线互相转换

```
// 下划线转换驼峰
function toHump(name) {
  return name.replace(/\_(\w)/g, function (all, letter){
    return letter.toUpperCase();
  });
}
// 驼峰转换下划线
function toLine(name) {
    return name.replace(/([A-Z])/g, "_$1").toLowerCase();
}
```

js正则表达式获取url参数

```
let reg =/([^?&+#]+)=([^?&+#]+)/g;  // 简化 reg =/([^&?]+)=([^&#]+)/g
let obj={};
let href = 'http://www.runoob.com/jquery/misc-trim.html?
channelid=12333&name=xiaoming&age=23';
href.replace(reg,($0,$1,$2)=>obj[$1]=$2)
console.log(obj);
```

返回字符串中最长连续相同字串的长度

```
let r = 'AAABBCCAAAA DDE666FF'

// 匹配模式：单个字符+第一个括号中的匹配值( * —— 有零个或者多个)
let match = r.match(/(\w)\1*/g) // 正则中\1的用法---反向引用
console.log('match :', match)   // match : [ 'AAA', 'BB', 'CC', 'AAAA', 'DD',
'E', '666', 'FF' ]

match.sort((a, b) => {
    return a.length < b.length  // 从大到小
})
console.log('match :', match)   // match : [ 'AAAA', 'AAA', '666', 'BB', 'CC',
'DD', 'FF', 'E' ]
console.log('连续相同字串长度 :', match[0].length) // 连续相同字串长度 : 4
```

## 项目收获

# 高频手撕代码

# JS

## • 使用闭包实现定时输出

```
// 使用闭包实现隔一秒打印1，2，3，4
for (var i = 0; i < 5; i++) {
  (function(i) {
     setTimeout(function() {
        console.log(i);
     }, i * 1000);
  })(i);
}

// 用let块级作用域
for(let i=0;i<5;i++){
  setTimeout(function(){
    console.log(i)
  }, 1000 * i)
}

// 请手写一个函数，这个函数名字是a，使得a被执行之后有如下效果。
// a(); // 函数返回值为1
// a(); // 函数返回值为2
// a(); // 函数返回值为3
// ...以此类推，每调用一次a函数，函数返回
var a = (function(){ // 使用闭包保存变量，使用立即执行函数返回+1函数
    var count = 0
    return function(){
            return ++count;
        }
})()

console.log(a()) // 1
console.log(a()) // 2
console.log(a()) // 3
```

## • 防抖和节流

- 函数防抖的实现

```
function debounce(fn, delay) {//防抖
  // 维护一个 timer，用来记录当前执行函数状态
  let timer = null;

  return function() {
    // 通过 ‘this’ 和 ‘arguments’ 获取函数的作用域和变量
```

```
    let context = this;
    let args = arguments;
    // 清理掉正在执行的函数，并重新执行
    clearTimeout(timer);
    timer = setTimeout(function() {
      fn.apply(context, args);
    }, delay);
  }
}
let flag = 0; // 记录当前函数调用次数
// 当用户滚动时被调用的函数
function foo() {
  flag++;
  console.log('Number of calls: %d', flag);
}

// 在 debounce 中包装我们的函数，过 2 秒触发一次
document.body.addEventListener('scroll', debounce(foo, 2000));
```

- 函数节流的实现

```
function throttle(func, delay){//节流
  let timer = null;

  return function(){
    let context = this;
    let args    = arguments;
    if(!timer){
      timer = setTimeout(function(){
        func.apply(context, args);
        timer = null;
      }, delay);
    }
  }
}
```

## 浅拷贝、深拷贝

数据类型判断的常用方法：

- typeof
- instanceof

```
(type, value) => Object.prototype.toString.call(value) === `[object ${type}]`
```

```
/**  浅拷贝的实现; */
function shallowCopy(object) {
  // 非引用类型直接返回
  if (!object || typeof object !== "object") return object;
  // 根据 object 的类型判断是新建一个数组还是对象
  let newObject = Array.isArray(object) ? [] : {};
  // 遍历 object，并且判断是 object 的属性才拷贝
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] = object[key];
    }
  }
  return newObject;
}

/** 最简单的深拷贝 JSON.parse(JSON.stringify())
* 缺点:
* 1.如果obj里面有时间对象，则JSON.stringify后再JSON.parse的结果，时间将只是字符串的形式，而不是对象的形式
* 2.如果obj里有RegExp(正则表达式的缩写)、Error对象，则序列化的结果将只得到空对象;
* 3、如果obj里有函数，undefined，则序列化的结果会把函数或 undefined丢失;
* 4、如果obj里有NaN、Infinity和-Infinity，则序列化的结果会变成null 等等
*
* 总结:
* 用法简单，然而使用这种方法会有一些隐藏的坑：因为在序列化JavaScript对象时，所有函数和原型成员会被有意忽略。
* 通俗点说，JSON.parse(JSON.stringfy(X))，其中X只能是Number, String, Boolean, Array, 扁平对象，即那些能够被 JSON 直接表示的数据结构。
*/

/** 简单的深拷贝的实现，利用浅拷贝的递归（object类型只支持数组和函数）; */
function deepCopy(object) {
  if (!object || typeof object !== "object") return object;
  let newObject = Array.isArray(object) ? [] : {};
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] =
        typeof object[key] === "object" ? deepCopy(object[key]) : object[key];
    }
  }
  return newObject;
}

/**完整的深拷贝实现,支持
String,Number,Boolean,null,undefined,Object,Array,Date,RegExp,Error 类型 */
// 引用类型判断函数
const IsType = (type, value) => Object.prototype.toString.call(value) ===
`[object ${type}]`;
```

```
// 深拷贝主函数
const DeepCopyRA = arg => {
  const newValue = IsType("Object", arg) // 判断是否是对象
    ? {}
    : IsType("Array", arg) // 判断是否是数组
    ? [] // 三元表达式嵌套
    : IsType("Date", arg) // 判断是否是日期对象
    ? new arg.constructor(+arg)
    : IsType("RegExp", arg) || IsType("Error", arg) // 判断是否是正则对象或错误对象
    ? new arg.constructor(arg)
    : arg;
  // 判断是否是数组或对象
  if (IsType("Object", arg) || IsType("Array", arg)) {
    // 循环遍历
    for (const key in arg) {
      // 判断是否为自身的属性，防止原型链的值
      Object.prototype.hasOwnProperty.call(arg, key) && (newValue[key] =
DeepCopyRA(arg[key]));
    }
  }
  return newValue;
};
```

## ● 数组扁平化

> 数组扁平化是指将一个多维数组变为一维数组 `[1, [2, 3, [4, 5]]] ------> [1, 2, 3, 4, 5]`
>
> 核心思路：遍历数组arr，若arr[i]为数组则递归遍历，直至arr[i]不为数组然后与之前的结果concat。

1. **reduce**

遍历数组每一项，若值为数组则递归遍历，否则concat。

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

reduce是数组的一种方法，它接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

reduce包含两个参数：

- 回调函数 ( 上一次调用返回的值或者是初始值，当前值，<当前值在对象中的索引>，<原数组> )
- 传给total的初始值

```
// 求数组的各项值相加的和:
arr.reduce((total, item)=> {  // total为之前的计算结果，item为数组的各项值
    return total + item;
}, 0);
```

2. **toString & split**

调用数组的toString方法，将数组变为字符串然后再用split分割还原为数组

```
function flatten(arr) {
    return arr.toString().split(',').map(function(item) {
        return Number(item);
    })
}
```

因为split分割后形成的数组的每一项值为字符串，所以需要用一个map方法遍历数组将其每一项转换为数值型

3. **join & split**

和上面的toString一样，join也可以将数组转换为字符串

```
function flatten(arr) {
    return arr.join(',').split(',').map(function(item) {
        return parseInt(item);
    })
}
```

4. **递归**

递归的遍历每一项，若为数组则继续遍历，否则concat

```
function flatten(arr) {
    var res = [];
    arr.map(item => {
        if(Array.isArray(item)) {
            res = res.concat(flatten(item));
        } else {
            res.push(item);
        }
    });
    return res;
}
```

5. **扩展运算符**

es6的扩展运算符能将二维数组变为一维

```
[].concat(...[1, 2, 3, [4, 5]]);  // [1, 2, 3, 4, 5]
```

根据这个结果我们可以做一个遍历，若arr中含有数组则使用一次扩展运算符，直至没有为止。

```
function flatten(arr) {
    while(arr.some(item=>Array.isArray(item))) {
        arr = [].concat(...arr);
    }
    return arr;
}
```

- **数组去重**

> 总结两类方法：
>
> - 两层循环 / 递归法
> - 利用语法自身键不可重复性

- 两层循环法

1. 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;
}
//NaN和{}没有去重，两个null直接消失了
```

2. 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;
}
```

- 利用语法自身键不可重复性

1. 利用ES6 Set去重（ES6中最常用）

```
arr => Array.from(new Set(arr))
// 或
arr => [...new Set(arr)]
```

不考虑兼容性，这种去重的方法代码最少，但是只能去除字符串和数字的重复，不能去除对象的重复。

2. 利用 indexOf / includes 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。

```
function unique(arr) {
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (array.indexOf(arr[i]) === -1) { //
!array.includes(arr[i])
            array.push(arr[i])
        }
    }
    return array;
}
//NaN、{}没有去重
```

3. 利用对象属性不能相同的特点作为HashMap进行去重 (有点问题，true对象直接去掉)

```
function unique(arr) {
    var arrry= [];
     var  obj = {};
    for (var i = 0; i < arr.length; i++) {
        if (!obj[arr[i]]) {
            arrry.push(arr[i])
            obj[arr[i]] = 1
        }
    }
    return arrry;
}
```

4. 利用sort()

   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，总是添加相同元素的最后一个值。

```
 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;
}
//NaN、{}没有去重
```

5. 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) === index;
  });
}
```

- ## 单例模式

确保一个类仅有一个实例，并提供一个访问它的全局访问点。一般用一个变量来标志当前的类已经创建过对象，或者利用闭包，如果下次获取当前类的实例时，直接返回之前创建的对象，否则重新创建

```
var singleton = function( fn ){
    var result;
    return function(){
        return result || ( result = fn .apply( this, arguments ) );
    }
}
var createMask = singleton( function(){
return document.body.appendChild( document.createElement('div') );
 })
```

- ## 手写promise，promise.all 和 promise.race

**手写promise**

```
function Promi(executor) {
    let _this = this;
  _this.$$status = 'pending';
  _this.failCallBack = undefined;
  _this.successCallback = undefined;
  _this.error = undefined;
  executor(resolve.bind(_this), reject.bind(_this));
  function resolve(params) {
    if (_this.$$status === 'pending') {
      _this.$$status = 'success'
      _this.successCallback(params)
    }
  }
  function reject(params) {
    if (_this.$$status === 'pending') {
      _this.$$status = 'fail'
      _this.failCallBack(params)
    }
  }
}

Promi.prototype.then = function(full, fail) {
  this.successCallback = full
  this.failCallBack = fail
};
Promise.prototype.catch = function(fn){
    return this.then(null,fn);
}
```

```
// 测试代码
new Promi(function(res, rej) {
  setTimeout(_ => res('成功'), 30)
}).then(res => console.log(res))
```

**手写promise.all, promise.race**

```
//promise.all()
  function myall(proArr) {
    return new Promise((resolve, reject) => {
      let ret = []
      let count = 0
      let done = (i, data) => {
        ret[i] = data
        if(++count === proArr.length) resolve(ret)
      }
      for (let i = 0; i < proArr.length; i++) {
        proArr[i].then(data => done(i,data) , reject)
      }
    })
  }


//promise.race();这么简单得益于promise的状态只能改变一次，即resolve和reject都只被能执行一次
 function myrace(proArr) {
    return new Promise(function (resolve, reject) {
      for(let i=0;i<proArr.length;i++){
        proArr[i].then(resolve,reject);
      }
    })
  }
```

- **将原生的ajax封装成promise**

```
// 原生js实现
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", SERVER_URL, true);
// 设置状态监听函数
xhr.onreadystatechange = function () {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
```

```
  }
};

// 将原生的ajax封装成promise
var myNewAjax = function(url){
  return new Promise(function(resolve,reject){
    var xhr = new XMLHttpRequest();
    xhr.open('get',url);
    xhr.send(data);
    xhr.onreadystatechange=function(){
      if(xhr.status==200&&readyState==4){
        var json=JSON.parse(xhr.responseText);
        resolve(json)
      }else if(xhr.readyState==4&&xhr.status!=200){
        reject('error');
      }
    }
  })
}
```

- **限制Promise“并发”的数量**

```
class LimitPromise {
  constructor (max) {
    this._max = max // 异步任务“并发”上限
    this._count = 0 // 当前正在执行的任务数量
    this._taskQueue = [] // 等待执行的任务队列
  }

  /**
   * 调用器，将异步任务函数和它的参数传入
   * @param caller 异步任务函数，它必须是async函数或者返回Promise的函数
   * @param args 异步任务函数的参数列表
   * @returns {Promise<unknown>} 返回一个新的Promise
   */
  call (caller, ...args) {
    return new Promise((resolve, reject) => {
      const task = this._createTask(caller, args, resolve, reject)
      if (this._count >= this._max) {
        // console.log('count >= max, push a task to queue')
        this._taskQueue.push(task)
      } else {
        task()
      }
    })
  }

  /**
```

```
   * 创建一个任务
   * @param caller 实际执行的函数
   * @param args 执行函数的参数
   * @param resolve
   * @param reject
   * @returns {Function} 返回一个任务函数
   * @private
   */
  _createTask (caller, args, resolve, reject) {
    return () => {
      // 实际上是在这里调用了异步任务，并将异步任务的返回（resolve和reject）抛给了上层
      caller(...args)
        .then(resolve)
        .catch(reject)
        .finally(() => {
          // 任务队列的消费区，利用Promise的finally方法，在异步任务结束后，取出下一个任务执行
          this._count--
          if (this._taskQueue.length) {
            // console.log('a task run over, pop a task to run')
            let task = this._taskQueue.shift()
            task()
          } else {
            // console.log('task count = ', count)
          }
        })
      this._count++
      // console.log('task run , task count = ', count)
    }
  }
}
```

- **实现sleep函数**

```
async function test() {
    console.log('开始');
    await sleep(4000);
    console.log('结束');
}

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

## • 模拟实现new

new 运算符是创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例，其创建过程如下：

- 创建一个空的简单JavaScript对象（即{}）
- 链接该对象（即设置该对象的构造函数）到另一个对象
- 将第一步新创建的对象作为this的上下文
- 如果该函数没有返回对象，则返回this

```
function newObject(){
    var obj = new Object();
    Constructor = [].shift.call(arguments);
    obj.__proto__ = Constructor.prototype;
    var demo = Constructor.apply(obj,arguments);
    return typeof demo === 'object' ? demo : obj;
}
```

## • 实现 call / apply / bind

> apply.call.bind 都是为了改变函数运行时上下文(this指向)而存在的。
>
> - 三兄弟接收的第一个参数都是 要绑定的this指向.
> - apply的第二个参数是一个参数数组,call和bind的第二个及之后的参数作为函数实参按顺序传入。
> - bind不会立即调用,其他两个会立即调用。

- call的简易模拟实现(es6)

```
/**
* 先将传入的指定执行环境的对象 context 取到
* 将需要执行的方法(调用call的对象) 作为 context 的一个属性方法fn
* 处理传入的参数， 并执行 context的属性方法fn， 传入处理好的参数
* 删除私自定义的 fn 属性
* 返回执行的结果
*/
Function.prototype.defineCall = function (context) {
    context = context || window;
    context.fn = this; //this指向 sayName函数实例
    let args = [];
    for (let i = 1; i < arguments.length; i++) { //i从1开始
        args.push(arguments[i]);
    } //或者args = [...arguments].slice(1);
    let result = context.fn(args.join(','));
    delete context.fn; // 删除该方法，不然会对传入的对象造成污染（新增方法）
    return result;
}
```

- apply的简易模拟实现(es6)

```
Function.prototype.defineApply = function (context, arr) {
    context = context || window;
    context.fn = this;
    let result;
    if (!arr) { // 第二个参数不传
        result = context.fn();
    } else { // 第二个参数是数组类型
        let args = [];
        for (let i = 0; i < arr.length; i++) { //i从0开始
            args.push(arr[i]);
        }
        result = context.fn(args.join(','));
    }
    delete context.fn;
    return result;
}
```

- bind的简易模拟实现(es6)

```
//用call、apply模拟实现bind
Function.prototype.bind = function (context) {
    let self = this; // 保存函数的引用
    return function () { // 返回一个新的函数
        // console.log(arguments);
        // return self.apply(context, arguments);
        return self.call(context, arguments);
    }
};
```

## 函数柯里化

> 函数柯里化指的是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

```
// es6 实现
function curry(fn, ...args) {
  return fn.length <= args.length ? fn(...args) : curry.bind(null, fn,
...args);
}

// es5 实现
function curry(fn, args) {
  // 获取函数需要的参数长度
  let length = fn.length;
  args = args || [];
  return function () {
    let subArgs = args.slice(0);
```

```
    // 拼接得到现有的所有参数
    for (let i = 0; i < arguments.length; i++) {
      subArgs.push(arguments[i]);
    }
    // 判断参数的长度是否已经满足函数所需参数的长度
    if (subArgs.length >= length) {
      // 如果满足，执行函数
      return fn.apply(this, subArgs);
    } else {
      // 如果不满足，递归返回科里化的函数，等待参数的传入
      return curry.call(this, fn, subArgs);
    }
  };
}
```

- **模拟 Object.create() 的实现**

```
function _create(obj){
    function C(){}
    C.prototype = obj;
    return new C();
}

var obj1 = {name: "Lilei"};
var lilei = _create(obj1);
lilei; // {}
lilei.name; // "Lilei"
```

- **数字千分位分隔符**

```
// 每次取末三位子字符串放到一个新的空字符串里并拼接上之前的末三位，原本数组不断截掉后三位直到长度小于三个，最后把剥完的原数组拼接上新的不断被填充的数组
// 专门处理正整数部分代码，完整实现需要做正负号和小数部分的前置处理
function toQfw(){
  var str_n=n.toString();
  var result="";
  while(str_n.length>3){
    result=","+str_n.slice(-3)+result;
    str_n=str_n.slice(0,str_n.length-3)
  }
  return str_n + result
};
// 正则表达式实现
'1000000.00'.replace(/(\d)(?=(?:\d{3})+\.)/g, '$1,');
```

- **当前周月**
  - 获取当前时间是本月第几周和年的第几周

```
var getMonthWeek = function (a, b, c) {
        /**
        * a = d = 当前日期
        * b = 6 - w = 当前周的还有几天过完(不算今天)
        * a + b 的和在除以7 就是当天是当前月份的第几周
        */
        var date = new Date(a, parseInt(b) - 1, c),
            w = date.getDay(),
            d = date.getDate();
        if(w==0){
            w=7;
        }
        var config={
            getMonth:date.getMonth()+1,
            getYear:date.getFullYear(),
            getWeek:Math.ceil((d + 6 - w) / 7),
        }
        return config;
    };
    var getDate=getMonthWeek("2018", "12", "31");
    console.log("今天是 " + getDate.getYear + " 年的第 "+
getDate.getMonth + " 月的第 " + getDate.getWeek + " 周");
```

  - 获取年的第几周

```
var getYearWeek = function(a, b, c)
{
    /*
    date1是当前日期
    date2是当年第一天
    d是当前日期是今年第多少天
    用d + 当前年的第一天的周差距的和在除以7就是本年第几周
    */
        var date1 = new Date(a, parseInt(b) - 1, c),
            date2 = new Date(a, 0, 1),
            d = Math.round((date1.valueOf() - date2.valueOf()) /
86400000);
        return Math.ceil((d + ((date2.getDay() + 1) - 1)) / 7);
};
```

  - 二维数组顺时针旋转90度

```
// 另外开辟空间
var arr = [[1,2,3],[4,5,6],[7,8,9]]
```

```
function rotateArr (arr) {
  // body...
  var len1 = arr.length, // 二维数组长度
      len2 = arr[0].length; // 子元素长度
  var newArr = new Array(len2).fill(0).map(i => new
Array(len1).fill(0))
  for(var i = 0; i < len1; i++){
    for(var j = 0; j < len2; j++){
      newArr[j][len1 - 1 - i] = arr[i][j]
    }
  }
  return newArr
}

// 在n * n原数组上进行旋转
// 1. 寻找最小旋转单位：以中心点为旋转对称中心的四个点
// 2. 判断遍历区间，保证遍历最多只改变一次值：不包含中心点的四分之一扇区
function rotateArrSitu (arr) {
  var n = arr.length
  var newArr = new Array(n).fill(0).map(i => new Array(n).fill(0))
  var limit = parseInt(n / 2)
  for(var i = 0; i <= limit; i++) {
    for(var j = i; j < n - 1 - i; j++){
      var temp = arr[i][j]
      arr[i][j] = arr[n-1-j][i]
      arr[n-1-j][i] = arr[n-1-i][n-1-j]
      arr[n-1-i][n-1-j] = arr[j][n-1-i]
      arr[j][n-1-i] = temp
    }
  }
  return arr
}
```

# CSS

## • 实现三角形

边框的均分原理

```
div {
    width:0px; /*设置宽高为0，所以div的内容为空，从才能形成三角形尖角*/
    height:0px;
    /*或者*/
    border-top:10px solid red;
    border-right:10px solid transparent; /*transparent 表示透明*/
    border-bottom:10px solid transparent;
    border-left:10px solid transparent;
    /*或者*/
    border-width: 100px;
    border-style: solid;
    border-color:red transparent transparent transparent;
}
```

- [实现元素的居中]

- ## 实现三栏布局 / 双栏布局

三列布局又分为两种，两列定宽一列自适应，以及两侧定宽中间自适应 两列定宽一列自适应：

1. 使用float+margin：

   给div设置float：left，left的div添加属性margin-right：left和center的间隔px,right的div添加属性margin-left：left和center的宽度之和加上间隔

2. 使用float+overflow：

   给div设置float：left，再给right的div设置overflow:hidden。这样子两个盒子浮动，另一个盒子触发bfc达到自适应

3. 使用float+calc：

   三栏float浮动（或者设置inline），中间栏使用calc动态计算宽度 calc(100% - 2 * 边栏宽度)

```
<style type="text/css">
    .container {
      position: relative;
      width: 100%;
      height: 600px;
    }
    .middle {
      height: 100%;
      float: left;
      width: calc(100% - 2 * 200px);
      background-color: pink;
    }
    .left, .right {
      float: left;
      width: 200px;
      height: 100%;
      background-color: yellow;
```

```
    }
    .right {
      background-color: gray;
    }
</style>
<div class="container">
    <div class="left"></div>
    <div class="middle"></div>
    <div class="right"></div>
</div>
```

4. 使用position：

   父级div设置position：relative，三个子级div设置position：absolute，这个要计算好盒子的宽度和间隔去设置位置，兼容性比较好

```
<style type="text/css">
    .container {
      position: relative;
      width: 100%;
      height: 600px;
    }
    .middle {
      height: 100%;
      padding: 0 200px;
      background-color: pink;
    }
    .left, .right {
      position: absolute;
      width: 200px;
      height: 100%;
      top: 0;
      left: 0;
      background-color: yellow;
    }
    .right {
      left: auto;
      right: 0;
      background-color: gray;
    }
</style>
<div class="container">
    <div class="middle"></div>
    <div class="left"></div>
    <div class="right"></div>
</div>
```

5. 使用table实现：

   父级div设置display：table，设置border-spacing：10px//设置间距，取值随意,子级div设置display:table-cell，这种方法兼容性好，适用于高度宽度未知的情况，但是margin失效，设计间隔比较麻烦，

6. flex实现：

   parent的div设置display：flex；left和center的div设置margin-right；然后right 的div设置flex：1；这样子right自适应，但是flex的兼容性不好

```
<style type="text/css">
    .container {
      display: flex;
      justify-content: space-between;
      width: 100%;
      height: 600px;
    }
    .middle {
      flex: 1;
      height: 100%;
      background-color: pink;
    }
    .left, .right {
      float: left;
      width: 200px;
      height: 100%;
      background-color: yellow;
    }
    .right {
      background-color: gray;
    }
</style>
<div class="container">
    <div class="left"></div>
    <div class="middle"></div>
    <div class="right"></div>
</div>
```

7. grid实现：

   parent的div设置display：grid，设置grid-template-columns属性，固定第一列第二列宽度，第三列auto，

对于两侧定宽中间自适应的布局，对于这种布局需要把center放在前面，可以采用双飞翼布局：圣杯布局，来实现，也可以使用上述方法中的grid，table，flex，position实现

---

# 算法

## 数据结构

> **数据结构的存储方式只有两种：数组（顺序存储）和链表（链式存储）**。散列表、栈、队列、堆、树、图等都属于「上层建筑」，而数组和链表才是「结构基础」。因为那些多样化的数据结构，究其源头，都是在链表或者数组上的特殊操作，API 不同而已。

## 数组

**一、数组特点：**

**所谓数组**，就是相同数据类型的元素按一定顺序排列的集合；数组的存储区间是连续的，占用内存比较大，故空间复杂的很大。但数组的二分查找时间复杂度小，都是O(1)；数组的特点是：查询简单，增加和删除困难；

- 在内存中，数组是一块连续的区域
- 数组需要预留空间

  在使用前需要提前申请所占内存的大小，如果提前不知道需要的空间大小时，预先申请就可能会浪费内存空间，即数组的空间利用率较低。注：数组的空间在编译阶段就需要进行确定，所以需要提前给出数组空间的大小(在运行阶段是不允许改变的)
- 在数组起始位置处，插入数据和删除数据效率低。

  插入数据时，待插入位置的元素和他后面的所有元素都需要向后搬移

  删除数据时，待删除位置后面的所有元素都需要向前搬移。
- 随机访问效率很高，时间复杂度可以达到O(1)

  因为数组的内存是连续的，想要访问那个元素，直接从数组的首地址向后偏移就可以访问到了。
- 数组开辟的空间，在不够使用的时候需要进行扩容；扩容的话，就涉及到需要把旧数组中的所有元素向新数组中搬移。
- 数组的空间是从栈分配的。（栈：先进后出）

**二、数组的优点：**

- 随机访问性强,查找速度快，时间复杂度是0（1）

**三、数组的缺点：**

- 从头部删除、从头部插入的效率低，时间复杂度是o(n),因为需要相应的向前搬移和向后搬移。
- 空间利用率不高
- 内存空间要求高，必须要有足够的连续的内存空间。
- 数组的空间大小是固定的，不能进行动态扩展。

# 链表

**一、链表的特点：**

所谓链表，链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。 相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到O(1)的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是O(logn)和O(1)。

链表:链表存储区间离散，占用内存比较宽松，故空间复杂度很小，但时间复杂度很大，达O（N）。链表的特点是：查询相对于数组困难，增加和删除容易。

- 在内存中，元素的空间可以在任意地方，空间是分散的，不需要连续。
- 链表中的元素有两个属性，一个是元素的值，另一个是指针，此指针标记了下一个元素的地址。

每一个数据都会保存下一个数据的内存地址，通过该地址就可以找到下一个数据

- 查找数据时间效率低，时间复杂度是o(n)

  因为链表的空间是分散的，所以不具有随机访问性，如果需要访问某个位置的数据，需要从第一个数开始找起，依次往后遍历，知道找到待查询的位置，故可能在查找某个元素时，时间复杂度是o(n)

- 空间不需要提前指定大小，是动态申请的，根据需求动态的申请和删除内存空间，扩展方便，故空间的利用率较高
- 任意位置插入元素和删除元素时间效率较高，时间复杂度是o(1)
- 链表的空间是从堆中分配的。（堆：先进先出，后进后出）

**二、链表的优点**

- 任意位置插入元素和删除元素的速度快，时间复杂度是o(1)
- 内存利用率高，不会浪费内存
- 链表的空间大小不固定，可以动态拓展。

**三、链表的缺点**

- 随机访问效率低，时间复杂度是o(n)

总之：

**对于想要快速访问数据，不经常有插入和删除元素的时候，选择数组**

**对于需要经常的插入和删除元素，而对访问元素时的效率没有很高要求的话，选择链表**

---

# 算法解题框架

数据结构的基本存储方式就是链式和顺序两种，基本操作就是增删查改，遍历方式无非迭代和递归。

## ● 动态规划解题套路框架

> - 菲波那契数列
> - 零钱兑换
> - 扔鸡蛋

```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)
```

动态规划问题的一般形式就是求最值，求解动态规划的核心问题是穷举。

首先，动态规划的穷举有点特别，因为这类问题存在**「重叠子问题」**，如果暴力穷举的话效率会极其低下，所以需要「备忘录」或者「DP table」来优化穷举过程，避免不必要的计算。

而且，动态规划问题一定会**具备「最优子结构」**，才能通过子问题的最值得到原问题的最值。

另外，虽然动态规划的核心思想就是穷举求最值，但是问题可以千变万化，穷举所有可行解其实并不是一件容易的事，只有列出**正确的「状态转移方程」**才能正确地穷举。

- 菲波那契数列

```
var fib = function(N) {
    if (N == 0 || N == 1) return N
    var prev = 0,curr = 1
    for(var i = 2; i <= N; i++){
        // [prev, curr] = [curr, param + curr]
        var temp = curr
        curr += prev
        prev  = temp
    }
    return curr
}
```

- 零钱兑换

```
var coinChange = function(coins, amount) {
    // 维护一个最小硬币组成表，用amount+1代表无穷大做初始化
    var arr = new Array(amount + 1).fill(amount + 1)
    if(coins == 0) return
    arr[0] = 0
    for(var i = 1; i < arr.length; i++) {
        for(coin of coins) {
            if(i < coin) continue
            arr[i] = Math.min(arr[i], 1 + arr[i - coin])
        }
    }
    return arr[amount] == amount + 1 ? -1 : arr[amount]
};
```

- 扔鸡蛋

```
let superEggDrop = (K, N)=> {
    let dp = Array(K+1).fill(0)
    let cnt = 0
    while (dp[K] < N){
        cnt++
        for (let i=K; i>0; i--){
            dp[i] = dp[i-1] + dp[i] + 1
        }
    }
    return cnt
};
```

## 回溯算法解题套路框架

> - 全排列问题
> - N皇后问题

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

其核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」(即回溯)

**解决一个回溯问题，实际上就是一个决策树的遍历过程**。你只需要思考 3 个问题：

1. 路径：也就是已经做出的选择。
2. 选择列表：也就是你当前可以做的选择。
3. 结束条件：也就是到达决策树底层，无法再做选择的条件。

- 全排列

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var res = []
var permute = function(nums) {
    res = []
    var track = []
    backtrack(nums, track)
    return res
};
// 回溯函数
// 路径记录：  track
// 路径选择：  nums中不在track的数
// 终点判断：  track长度达到num长度
function backtrack(nums, track) {
    // 触发结束条件
    if(track.length == nums.length) {
        res.push([...track])
        return
    }
    // 做选择
    for(choice of nums) {
        // 排除路径记录中的选择
        if(track.includes(choice)) continue
        track.push(choice)
        backtrack(nums, track)
        track.pop()
    }
}
```

- N皇后

## • BFS（广度优先搜索）算法解题套路框架

> - 二叉树的最小深度
> - 打开转盘锁

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

> 队列 `q` 就不说了，BFS 的核心数据结构；`cur.adj()` 泛指 `cur` 相邻的节点，比如说二维数组中，`cur` 上下左右四面的位置就是相邻节点；`visited` 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 `visited`。

- 二叉树的最小深度

```
var minDepth = function(root) {
    if(root == null) return 0
    var queue = [root],
        deep = 1;
    while(queue.length) {
        var len = queue.length
        for(var i = 0; i < len; i++){
```

```
            var curr = queue.shift()
            if(curr.left == null && curr.right == null) return deep
            if(curr.left) queue.push(curr.left)
            if(curr.right) queue.push(curr.right)
        }
        deep++
    }
    return -1
};
```

## 常见排序算法

| 排序方法 | 时间复杂度（平均） | 时间复杂度（最坏） | 时间复杂度（最好） | 空间复杂度 | 稳定性 |
| --- | --- | --- | --- | --- | --- |
| 插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 希尔排序 | $O(n^{1.3})$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 不稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 堆排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(1)$ | 不稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 快速排序 | $O(nlog_2n)$ | $O(n^2)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | 不稳定 |
| 归并排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(n)$ | 稳定 |
| | | | | | |
| 计数排序 | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | 稳定 |
| 桶排序 | $O(n+k)$ | $O(n^2)$ | $O(n)$ | $O(n+k)$ | 稳定 |
| 基数排序 | $O(n*k)$ | $O(n*k)$ | $O(n*k)$ | $O(n+k)$ | 稳定 |

- **冒泡排序**

思路：外层循环从1到n-1，内循环从当前外层的元素的下一个位置开始，依次和外层的元素比较，出现逆序就交换，通过与相邻元素的比较和交换来把小的数交换到最前面。

```
// 冒泡排序
function bubbleSort(arr) {
    let len = arr.length-1;
    for (let i=0; i<len; i++) {
        for (let j=0; j<len-i; j++) {
            if (arr[j] > arr[j+1]) { // 大于则交换两个的位置
                let temp = arr[j];
```

```
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    return arr;
}
```

## • 选择排序

思路：冒泡排序是通过相邻的比较和交换，每次找个最小值。选择排序是：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。

```
// 选择排序
function selectionSort(arr) {
    let len = arr.length;
    let minIndex, temp;
    for (let i=0; i<len; i++) {
        minIndex = i;
        for (let j=i+1; j<len; j++) {
            if (arr[minIndex] > arr[j]) {
                minIndex = j; // 保存最小值索引
            }
        }
        // 进行互换位置
        temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
    return arr;
}
```

## • 插入排序

适用与规模小、有序的数据3w- 排序思想：将数组分成两个，一个是已排序好的，一个是待排序的，将待排序的元素和已排序好的元素进行比较，插入适当位置。

```
// 插入排序
function insertionSort(arr) {
    let len = arr.length;
    let prev, cur;

    for (let i = 1; i < len; i++) {
        prev = i - 1;
        cur = arr[i];
        while (prev>=0 && arr[prev]>cur) {
            arr[prev+1] = arr[prev];
```

```
            prev--;
        }
        arr[prev+1] = cur;
    }
    return arr;
}
```

## • 快速排序

排序思想：取一个基准值，比基准值小的在左边，大的在右边；左右在继续这样的操作，最后合并。

```
function quickSort (arr) {
    if (arr.length < 2) { // 数组元素只有一个的时候
        return arr;
    }
    let pivotIndex = Math.floor(arr.length/2);
    let pivot = arr.splice(pivotIndex,1)[0]; // 基准值
    let left = [], // 存放比基准值小的
        right = []; // 存放比基准值大的
    arr.forEach(item=>{
      item > pivot ? right.push(item) : left.push(item);
    })
    return quickSort(left).concat([pivot], quickSort(right));
}
```

## • 堆排序

排序思想：先构建一个最大堆，然后循环数组，依次将最大的元素放到末尾

```
function heapSort(arr) {
  let len = arr.length;
  function maxHeapify(i) {
    let left = 2 * i + 1;
    let right = 2 * i + 2;
    let largest = i;

    if (left < len && arr[left] > arr[largest]) {
      largest = left;
    }
    if (right < len && arr[right] > arr[largest]) {
      largest = right;
    }
    if (largest != i) {
      swap(i, largest);
      maxHeapify(largest);
    }
  }
```

```
  function swap(i, j) {
    let t = arr[i];
    arr[i] = arr[j];
    arr[j] = t;
  }
  // 构建堆
  for (let i = Math.floor(len/2) - 1; i >= 0; i--) {
    maxHeapify(i);
  }
  // 大-> 小
  for (let i = arr.length - 1; i > 0; i--) {
    swap(0, i);
    len--;
    maxHeapify(0);
  }
  /* 小->大
  for (let i = 0; i < len; i++) {
    maxHeapify(i);
  }
  */
  return arr;
}
```

# TOP K

## 快速排序法

核查pivot-left+1和k大小比较 如果大于k那么topK就在pivot左侧的这些数里面 如果等于k那么topK就是pivot所在位置的值 如果小于k那么寻找pivot右侧的topk-(pivot-left+1)即可d

```
let quickTopK = function (arr, k) {
    if(k==0)return []
    if (arr.length < 2) return arr
    let midValue = arr.splice(0, 1), left = [], right = []
    arr.forEach((el) => {
        el > midValue ? left.push(el) : right.push(el)
    });
    if (left.length == k) {
        return left
    } else if (left.length > k) {
        return quickTopK(left, k)
    } else {
        return left.concat(midValue, quickTopK(right, k - left.length - 1))
    }
}
console.log(quickTopK([6, 0, 1, 4, 9, 7, -3, 1, -4, -8, 4, -7, -3, 3, 2, -3, 9, 5, -4, 0], 8))
////[ 9, 7, 9, 6, 5, 4, 4, 3 ]
```

## 堆排序法

1. 首先把数组中的前k个值用来建一个小根堆(不是大根堆)。
2. 之后的其他数拿到之后进行判断，如果遇到比小顶堆的堆顶的值大的，将堆顶元素删除，将该元素放入堆中。
3. 最终的堆会是数组中最大的K个数组成的结构，小根堆的顶部又是结构中的最小数，因此把堆顶的值弹出即可得到Top-K。

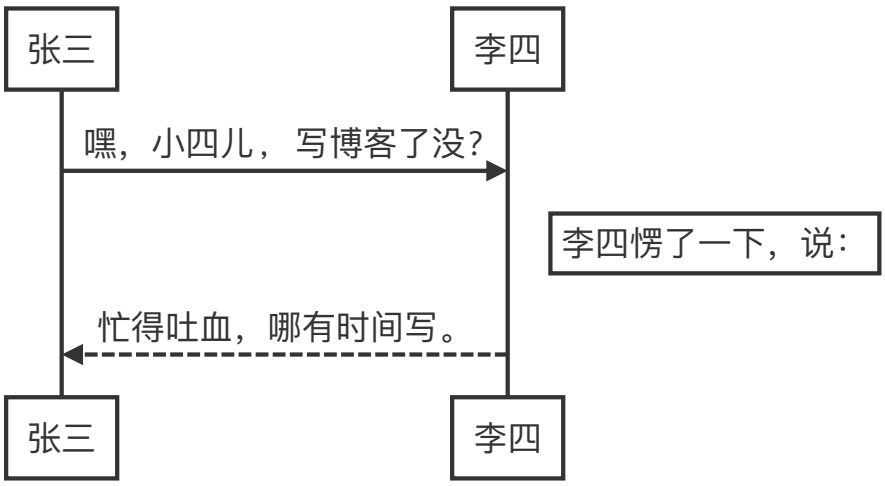
```
function swap(A, i, j) {
    let temp = A[i]
    A[i] = A[j]
    A[j] = temp
}

function shiftDown(A, i, length) {
    let temp
    for (let j = 2 * i + 1; j < length; j = 2 * j + 1) {
        temp = A[i]
        if (j + 1 < length && A[j] > A[j + 1]) {
            j++
        }
        if (temp > A[j]) {
            swap(A, i, j)
            i = j
        } else {
            break
        }
    }
}
function heapTopK(arr, k) {
    let A = arr.splice(0, k)
    for (let i = Math.floor(A.length / 2 - 1); i >= 0; i--) {
        shiftDown(A, i, A.length)
    }
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] > A[0]) {
            A[0] = arr[i]
            for (let i = Math.floor(A.length / 2 - 1); i >= 0; i--) {
                shiftDown(A, i, A.length)
            }
        }
    }
    return A
}
```

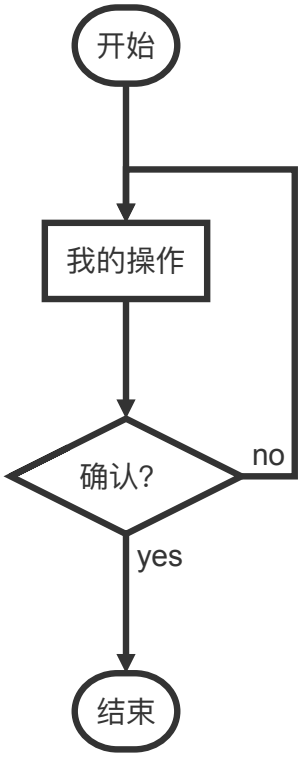
# 参考资料


1. 牛客前端面试宝典
2. 面试高频手撕题
3. Vue官网
4. Vue基础语法介绍
5. 浅谈浏览器多进程与JS线程
6. TypeScript - 一种思维方式


$$\Gamma(n) = (n-1)!\quad \forall n \in \mathbb{N}$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$




开始
我的操作
确认?
no
yes
结束