

Table of Contents

Introduction	1.1
第一部分-概述	1.2
前端简史	1.2.1
JavaScript语言发展概况	1.2.2
JavaScript编译原理	1.2.2.1
前端工程化	1.2.3
前端组件化	1.2.4
Welcome on Board	1.2.5
电脑、操作系统	1.2.5.1
终端与常用命令行工具	1.2.5.2
Node.js环境	1.2.5.3
编辑器	1.2.5.4
第二部分-工程化实践	1.3
模块系统	1.3.1
刀耕火种时代	1.3.1.1
CommonJS模块系统	1.3.1.2
AMD模块系统	1.3.1.3
UMD模块系统	1.3.1.4
ECMAScript 模块系统	1.3.1.5
前端资源构建	1.3.2
Gulp	1.3.2.1
Babel	1.3.2.2
Webpack	1.3.2.3
代码的组织与管理	1.3.3
开发与调试	1.3.4
前端测试	1.3.5
第三部分-组件化开发	1.4
组件化历史	1.4.1
面向对象的组件化开发	1.4.2
web components技术	1.4.3
自定义元素	1.4.3.1
Shadow DOM	1.4.3.2
HTML template	1.4.3.3
HTML Import	1.4.3.4
Polymer	1.4.4
React	1.4.5
基本使用	1.4.5.1

虚拟DOM原理	1.4.5.2
组件间通信	1.4.5.3
服务端渲染	1.4.5.4
React与函数式编程	1.4.5.5
Vue	1.4.6
微信小程序组件化开发实践	1.4.7
第四部分-前端开发中的常见话题	1.5
标准库	1.5.1
编码规范	1.5.2
重构	1.5.3
前端程序员的自我成长	1.5.4
附录	1.6
Git 基本使用	1.6.1
Web 浏览器工作原理	1.6.2
Lambda 演算	1.6.3
不可变数据（Immutable Data）	1.6.4
参考资料	1.7

大纲&目录

书名：Web组件化开发与前端工程化实践

英文书名：Practice In Web Component Development And Front End Engineering

选题产生的背景、市场需求情况及产品定位：

目前无论是 to B 产品还是 to C 产品，业务复杂度都在不断提升，前端开发人员要保证页面、组件的良好交互体验，提供更好的性能，同时又要保证自己所写代码的可维护性，这是前端工程化命题的一个背景；另外，近几年前端飞速发展，各种框架层出不穷，它们各有不同，然而也有着诸多相似的设计理念。web 框架设计原理、前端工程化、web 组件化开发，这三个主题是 web 前端开发人员在汹涌澎湃的技术浪潮中应该关注的焦点。

目前市面上多为介绍框架开发实践，涉及框架原理的较少，而关注前端工程化、组件化开发的书籍，则更为稀少。国内外都如此。这与相关标准或者社区规范一直欠缺有很大关系。而随着 ES6 模块，以及 Web Components 标准的提出，这些问题也都有了大量的实践，已经可以总结出一些经验与问题，供大家思考。

本书面向的读者是具备一定经验的初中级Web前端开发人员，最好对 JavaScript、CSS 以及 Node.js 有一定的了解。

内容简介（50-200字）

本书简要介绍了当前主流的 Web 前端框架设计原理，然后基于 React 技术栈并结合若干示例来介绍 Web 组件化开发的相关实践，给读者带来一些直观的认识；最后以 webpack 这样的前端开发打包工具为例来介绍前端工程化所要解决的问题，原理与实践。

图书特色

本书的特色是理论与实践并重。理论方面：深入浅出，对组件化、工程化与主流框架设计的核心概念、原理进行剖析；实践方面：结合作者在工作中遇到的具体案例，讲解组件化开发的思路，具备较好的可操作性。

目录

目录中这些是核心内容。不过它们对读者的技术水平要求可能比较高，所以，写作中为了做到“深入浅出”，会对相关的知识进行介绍，通过增加章节、附录等形式。可能还会涉及的其他知识点：设计模式；编译原理；前端性能优化。

第0章 Get Started

第一章 前端组件化、框架与原理概述

- 1.1 组件化历史
- 1.2 常见的前端框架及其特点
 - 1.2.1 AngularJS
 - 1.2.2 ReactJS
 - 1.2.3 Vue
 - 1.2.4 Web Components 与 Polymer
- 1.3 MDV 与变化侦测
- 1.4 虚拟 DOM

第二章 React 组件化开发入门与实践

- 2.1 简介
- 2.2 React 入门示例
- 2.3 组件生命周期

2.4 高性能组件

第三章 React 组件状态管理

- 3.1 组件间通信问题
- 3.2 无状态组件
- 3.3 Redux
- 3.4 局部状态与全局状态
- 3.5 RxJS

第四章 前端工程化原理与实践

- 4.1 历史
- 4.1.1 Node.js 与 CommonJS 规范
- 4.1.2 从 Gulp 到 Webpack
- 4.2 webpack 基本使用
- 4.3 webpack 原理

附录I ECMAScript6

- 附录II 函数式编程
- 附录III RxJS
- 附录IV ESLint 代码规范

关键词

web前端开发

web组件化

前端工程化

JavaScript

请用200~400字概述您最想让读者了解的有关本书的重要信息：

随着国内互联网的飞速发展，前端开发人员的需求量也在不断增长，而国内高校很少会专门开设Web前端开发类课程，大多数同学都是靠自学来获得前端开发相应技能。已经有很多专门讲述开发语言的书籍，详细本书的读者也都有所涉猎。但是在实际的工作中，只掌握编程语言技能，是不足以应对每天的开发工作的。现在的前端技术日趋复杂，这对我们的开发效率、代码质量、可维护性等都提出了更高的要求。本书会从工程实践的角度出发，给你讲解可以应对大型项目开发的技能。

概述

本章会对前端、JS、HTML、CSS 分别进行发展历史的介绍并对未来进行展望。

因为很多我们当前采用的工程化、组件化方案，可能会随着技术的发展而逐渐在语言层面上标准化，那么很有必要关注前端核心技术的未来发展趋势。

前端简史

世界上第一个网站

世界上第一个网站是由在英国 CERN（欧洲核子研究中心，法语为 Conseil Européen pour la Recherche Nucléaire，因此缩写为 CERN，对应的英文名称是 European Council for Nuclear Research）工作的一位科学家，蒂姆·伯纳斯-李（Tim Berners-Lee），在 1989 年创建的，页面服务由蒂姆的 NeXT 电脑提供（这台机器目前仍在 CERN）。我们现在依然可以在 <http://info.cern.ch/> 看到它相关的原始页面。如果打开 [World Wide Web](#)，观察它的响应头，可以看到该文件的最后修改时间（Last-Modified Time）是 1992 年 12 月 3 日：

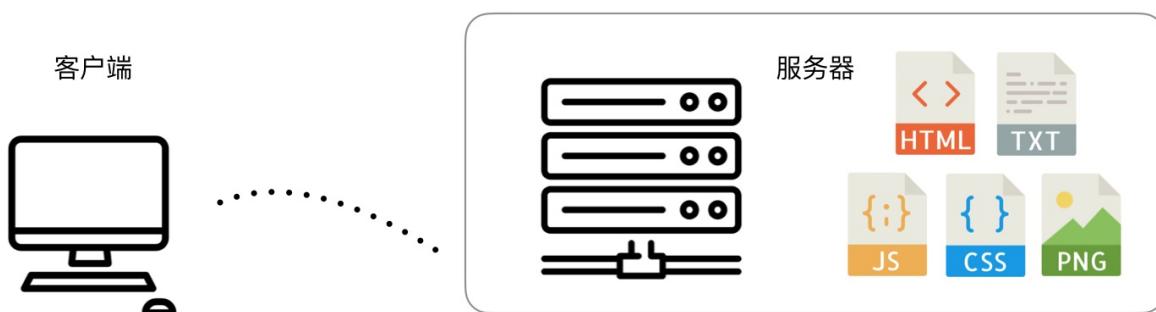
```
# http://info.cern.ch/hypertext/WWW/TheProject.html 请求的 Response Headers
HTTP/1.1 200 OK
Date: Thu, 08 Feb 2018 15:21:53 GMT
Server: Apache
Last-Modified: Thu, 03 Dec 1992 08:37:20 GMT
ETag: "40521e06-8a9-291e721905000"
Accept-Ranges: bytes
Content-Length: 2217
Connection: close
Content-Type: text/html
```

时过境迁，1952年创建的 CERN 的最初目标，是探究原子内部的结构（这也是为什么它的名字里有“核子”，即 nuclear）。而到了今天，人类对于物质的理解已经不仅仅停留在原子核层面，CERN 的研究内容也更为深入，主要为粒子物理领域（particle physics）——研究物质的基础组成及它们之间的相互作用。

World Wide Web (WWW) 项目的初衷是让世界各高校、研究机构的科学家可以分享信息。1993 年 8 月 30 日，CERN 将 WWW 软件公开到了公网中，软件里包括 Web 服务软件、一个简单的浏览器以及一份代码库。随后，CERN 声明这些软件都可自由获取与使用。

万维网从此脱离实验室，“飞入寻常百姓家”。在接下来的二十几年里，不计其数的计算机、智能设备，加快了全球的信息交换速度。信息科技成为21世纪初人类文明进步的最大推手，CERN 的物理学家们功不可没。

静态页面时期



在静态页面时期，Web 服务器就是最简单的文件服务器。用户主要请求 HTML 文件为入口，然后服务器会响应后续的脚本、样式、图片等资源请求。这些文件由网站开发人员、设计师、内容管理人员一起维护。客户端如果想要与网站进行通信怎么办？发邮件！

这个时期主要以博客为主。

动态网页（DHTML）时期

1994年，雅虎时代开始。雅虎对当时的几乎所有网站人工进行分类，其数据库中的注册网站无论是在形式上还是内容上质量都非常高。

后端主导的 MVC 模式时期

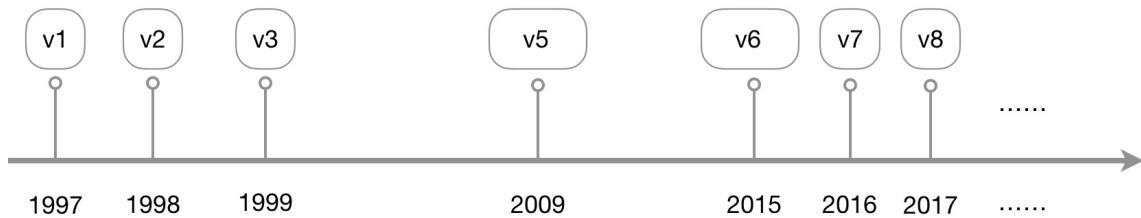
前后端分离时期

谁提出的前后端分离解决方案？？

前端应用时期

JavaScript 语言发展概况

ECMAScript-262 版本历史



JavaScript 编译原理

<https://zhuanlan.zhihu.com/p/32189701>

前端工程化的定义

20 多年前，Web 刚刚兴起的时候，并没有 Web 前端开发工程师这样的职位，当时的网页开发工作应该被称为网页设计，切图、布局、样式是其主要工作。随着互联网的不断发展，前端交互、性能、数据处理、安全问题一个个成为网页设计人员的关注点，这群人也分为了两个领域：交互设计师，前端程序员。软件开发中的大部分工程问题，前端程序员都需要涉及。

Node.js 诞生后，前端开发人员凭借自身的 JavaScript 语言优势，也可以逐渐接管部分原本由后端工程师承担的工作，例如页面服务器端渲染、网站路由管理。以常见的 MVC (Model-View-Controller) 架构为例，也就是 View 和 Controller 两个环节。

苹果公司的 iPhone 手机（2007年）引领的智能设备革命，使得前端代码的运行环境变得日益复杂。时至今日，Web 页面的“一处编写，随处运行”天然优势促使 Android、iOS 等系统中的大量应用采取了 Hybrid 开发模式。而且不仅仅是浏览器，微信还推出了小程序这样的运行环境，使得 JavaScript 可以编写出接近原生应用的“页面”。

到了现在，我们可以认为 Web 前端工程化主要包含以下范畴：

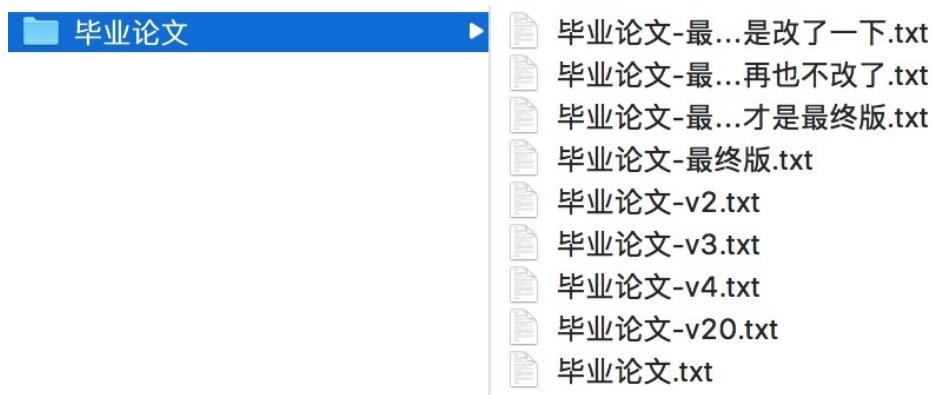
- 版本控制
- 模块化开发
- 前端资源构建&持续集成
- 服务器运维
- 性能优化

版本控制

版本控制系统 (VCS, Version Control System) 通常用于管理软件开发过程中的源代码文件，好的版本控制系统应该具备多人并行开发、代码变更记录、文件冲突处理机制、分支策略等特性。版本控制系统经历过三代演进：

本地版本控制 (Local Version Control System)

最原始的本地备份式版本控制。这种版本控制策略，本质上只是文件备份，最常用的方式就是将当期文件复制到另一个目录（可能会添加时间戳）。不少人也许有过类似下面这样命名文件的经历，这其实也是一种本地版本控制策略：



集中式版本控制系统 (Centralized Version Control System)

然后是以 Subversion (通常被叫做 svn) 为代表的中央版本控制系统。

分布式版本控制系统 (Distributed Version Control System)

再后来，出现了以 Git 为代表的分布式版本控制系统。

Subversion (2000年)、Git (2005年) 都是免费开源软件，Git 出现的更晚一些，也吸收了前者的很多优点，两者各有优缺点。不过随着 Github 网站的崛起，以及自由软件开发的持续热浪，越来越多的程序员新人更倾向于使用 Git 作为自己主要的版本控制工具，对前端开发人员来说更是如此——JavaScript 向来是 Github 上面最受欢迎的编程语言 (GitHub Universe, 2017)。

Git 的使用入门、原理请参加附录A。读者如果希望更深入地学习 Git 知识，那么 Scott Chacon 和 Ben Straub 所著的 Pro Git (2nd Edition, 2014) 可能是目前最好的 Git 学习资料，任何人都可以在网上免费阅读此书：<https://git-scm.com/book/en/v2>。此外，乔恩·罗力格 (Jon Loeliger) 与马修·麦卡洛 (Matthew McCullough) 合著的《Git版本控制管理（第2版）》也不错。

模块系统与模块化开发

Web 前端一开始是没有模块化开发概念的：简单地写一些 JavaScript 脚本，放到一个文件里，然后在 HTML 文件中通过 `<script>` 标签引入它就可以了。不过后来每个页面所需的 JavaScript 代码量不断增长，即便可以容忍通过同一个文件加载 JavaScript，源代码在同一个文件中进行开发也是非常难受的。我们希望一个页面有一个入口 JavaScript 文件，不妨叫做 `index.js`，然后它在里面去声明、加载自己依赖的其他 JavaScript 脚本——这便是模块化开发的初衷。模块化开发的其他好处还有代码易于维护、方便多人开发、减小系统耦合度、提升代码复用能力等。

2015年6月正式发布的 ECMAScript 6.0 (以下简称 ES6) 里对 JavaScript 模块标准进行了定义，称为 ECMAScript Module (简称 ESM)。在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。CommonJS 模块系统主要用于服务器端，例如 Node.js 一开始就是采用了 CommonJS 模块系统。Node.js 在 8.8 版本之后，开始原生支持 ESM。

除了 JavaScript，我们的 CSS 资源也有各种模块化开发的解决方案。例如 less/sass/stylus/postcss 等。这些 CSS 预处理语言的目标大致都是要提升代码复用、降低维护成本。

模块化开发也涉及代码的目录结构组织方式。Web 前端开发最初按照资源类型分别创建目录，例如通常一个网站下会有 `js`、`css`、`img`、`fonts` 等标志性的目录。在 `js` 目录下存放各个页面要用到的脚本文件，例如：

```
.
└── js
    ├── page-1.js
    ├── page-2.js
    └── ...
└── css
└── img
```

后来则倾向于将相关资源就近放置，以 `page` 为单位，每个页面有自己的目录，下面存放自己所需要的所有资源，与单个页面无关的资源 (即公共资源) 则另外设置目录进行存放。

```
pages
└── page-1
    ├── index.js
    ├── style.css
    └── index.html
└── page-2
└── page-3
```

再到现在，组件化的开发设计理念之下，页面也被视为组件，那么一切资源都以组件为基本单位进行目录组织，实际上就更为自然、合理了。可以说，模块化是组件化开发的雏形，组件化开发是我们的最终目标。

前端资源构建与持续集成

我们的代码按照一定的模块化、组件化方式进行组织之后，需要分析处理各个组件的依赖，进行代码分析与整合、根据入口文件逐个进行打包，最后进行压缩或者合并。例如为了使得生产环境下的代码体积最小，我们会使用诸如 `uglify.js` 这样的工具对打包后的脚本文件进行一次混淆压缩。CSS 文件也是类似。

在 `Bootstrap.css` 的官方 Github 代码仓库里 (<https://github.com/twbs/bootstrap>)，可以看到其对 CSS 源码的组织方式。

TODO => 持续集成

服务端运维

Node.js 服务端运维关注这些问题：

- 如何充分利用服务器的多核 CPU。由于每个 Node.js 进程实例使用一个核心且每个实例中的 JavaScript 代码只能是单线程的，这就限制了单个 Node.js 进程的计算能力。现实中的服务器往往具有多个核心，Node.js 应用应该尽可能发挥多核 CPU 的计算能力、提高系统的并发性能。
- 如何保证服务的稳定性，发布部署时也应尽量避免服务不可用，即做到平滑发布
- 如何管理服务器端 JavaScript 应用产生的大量日志数据
- 应该有可靠的性能、服务监控系统

Web 组件化开发概述

前端工程化主要关注非运行时的开发问题（文件处理、代码编译等），组件化则关注前端代码在运行时的功能实现方式及功能模块之间的关系。

Welcome on Board

你好，

欢迎你加入猫眼前端组，感谢你对我们的认可，也向我们证明了你的实力。“既往不恋，纵情向前”，是美团的口号，今天送给你，让我们一起扬帆起航。那么接下来开启你在美团的第一天吧！

我加入猫眼前端团队的第一天，组长分享了一篇题目为“Welcome on board”的内部 wiki，上面正是开头的一段话。

这篇 wiki 给出了入职第一天的员工需要进行的开发环境配置、软件安装等准备工作清单。正所谓“工欲善其事必先利其器”，其中很多工具都可以提升前端工程师的开发效率，所以在这里，我们也从配置你的开发环境开始。

电脑&操作系统

操作系统

操作系统的设计有所不同，造就了该系统下软件开发大大小小、方方面面的风格。……总的来说，与不同操作系统相关的设计和编程风格可以追溯出三个源头：

- 操作系统设计者的意图
- 成本和编程环境的限制对设计的均衡影响
- 文化随机漂移，传统无非就是先入为主

—— Eric S. Raymond, 《Raymond. UNIX 编程艺术》

不同的操作系统存在的纯用户转变为开发者的门槛有所不同。Unix 操作系统都会默认安装编译器和脚本工具，并且随时默认就是（或者可以随时切换到）命令行界面（CLI, Command Line Interface），诸如此类的特点，使得人们常说 Unix 是程序员写给程序员的。

Windows 在网络支持方面的落后，导致2000年后，其在服务器操作市场领域完全输给了 Unix 阵营。这也是为什么很多 Web 开发者（Web 前端工程师也不例外）选择在 Unix 操作系统下进行软件开发工作。除了上面提到的那些优点，也是因为绝大部分 Web 服务是由运行了 Unix 系统的服务器提供的，开发时尽可能保持开发环境与生产环境一致，可以减少很多兼容性问题。

推荐读者在 Unix 系列的操作系统下编程，包括 Linux、MacOS X、BSD、Solaris 等，其独特的编程艺术、优雅的设计哲学，会启示你写出更好的软件。

苹果电脑因为拥有极佳的硬件配置，并且其操作系统 Darwin 是 Unix 的一个发行版，Linux 的特点它基本都具备，所以成为了国内不少互联网公司程序员的标配。

本书所有的代码、命令都在安装了 MacOS@10.12.6 的 Macbook Pro 机器上面执行通过。

如果你没有 Mac，那么推荐安装任何一款你喜欢的 Linux 操作系统。Ubuntu 有着友好的界面，通常是入门者的首选；而 CentOS 由于在服务器端的应用更为广泛，也是非常适合开发者安装的系统。

XCode

用 Mac 进行开发，需要用到 XCode 里的一个子工具：Command Line Tools for Xcode。

参考资料

1. Eric S. Raymond. UNIX编程艺术 [M]. 北京: 电子工业出版社, 2012. 2.

终端与常用命令行工具

Iterm2

无论开发或者运维，亦或平时进行一些简单的任务，使用终端（Terminal）都比 GUI 界面程序高效得多。MacOS 虽然自带了一个终端，却并不好用，如果不借助于 Oh-my-zsh 这样的工具，甚至都无法进行文件名称补全提示。此外，MacOS 的终端也不支持窗口分隔为多面板（Split Panes）。

怀着打造一个好用的 MacOS 终端的理念，Iterm2 诞生了，它是用 Objective-C 为主要编程语言开发的终端模拟程序（terminal emulator）。它有许多方便的功能：

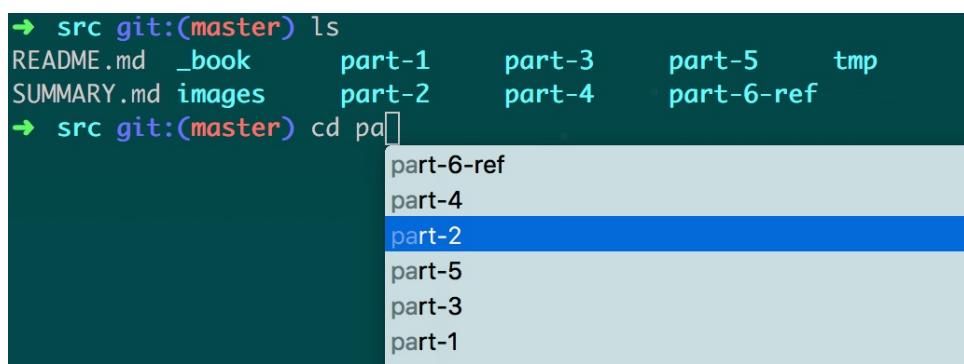
多个窗口

Iterm2 最常用的功能，是将当前标签分割为多个面板。例如我在写这本书的时候，将标签分为四个面板：



自动补全（Autocomplete）

输入单词的开头部分，然后按 `Cmd + S` 和 `;`，就可以出现自动补全提示选择框，通过上下箭头或者鼠标，可以直接选择自己想要输入的词语。如下图所示。



随时随地召唤 Iterm2 (Hotkey Window)

在 Iterm2 的偏好设置里（`Preferences -> Keys -> Create a Dedicated Hotkey Window`），可以设置一个能够通过快捷键随时打开的悬浮半透明终端窗口。这个功能对于随时想敲上几句命令行的开发者来说很有帮助。好比我正在浏览鸟哥的Linux私房菜网站，看到有个命令很有意思，想尝试一下。就可以通过这种方式快速“召唤”出 Iterm2（如果希望在全屏模式下也可以悬浮显示 Iterm2，需要设置其打开的方式为 `Floating Window`）：

The screenshot shows a terminal window with the following session:

```

→ ~ last | grep 'root'
root      console  子龙
root      console  学习  Thú Aug 17 00:14 - crash (00:04)
→ ~  cn.linux.vbird.org/linux_basic/0320bash_6.php
应用  Bookmarks  Google  MT  效率-办公  小短  翻墙  常去网站  我的网站  WEB前端技术  NodeJS
* grep

刚刚的 cut 是将一行信息当中，取出某部分我们想要的，而 grep 则是分析一行信息，若当中有我们所需要的信息，就将该行拿出来~简单的语法是这样的：

[root@www ~]# grep [-acinv] [-color=auto] '搜寻字符串' filename
选项与参数：
-a : 将 binary 文件以 text 文件的方式搜寻数据
-c : 计算找到 '搜寻字符串' 的次数
-i : 忽略大小写的不同，所以大小写视为相同
-n : 顺便输出行号
-v : 反向选择，亦即显示出没有 '搜寻字符串' 内容的那一行！
-color=auto : 可以将找到的关键词部分加上颜色的显示！

范例一：将 last 当中，有出现 root 的那一行就取出来；
[root@www ~]# last | grep 'root'

范例二：与范例一相反，只要没有 root 的就取出！
[root@www ~]# last | grep -v 'root'

范例三：在 last 的输出信息中，只要有 root 就取出，并且仅取第一栏
[root@www ~]# last | grep 'root' | cut -d ' ' -f1
# 在取出 root 之后，利用上个命令 cut 的处理，就能够仅取得第一栏啰！

```

其他

此外，Iterm2 还有选择文本并复制的快捷方式，保留粘贴历史，强大的搜索，以及命令人回溯功能。可以在其官网^[1]获得更详细的使用说明。

Oh-my-zsh

Oh-my-zsh 是一个 Zsh 配置管理框架。安装 Oh-my-zsh 前要确保 Zsh 已经装好（Mac 预装了 Zsh，所以很方便）。运行下面的脚本可以安装 Oh-my-zsh：

```
$ sh -c "$(curl -fsSL https://raw.githubusercontent.com/robbyrussell/oh-my-zsh/master/tools/install.sh)"
```

1990年，Paul Falstad 还在普林斯顿大学读书时，写出了 Zsh 的第一版。程序的名字“zsh”来源于当时的一位助理教授（如今已经是耶鲁大学的教授）Zhong Shao 的账号 ID，Paul Falstad 用“zsh”命名这个程序向他致敬。

Homebrew

Homebrew 也许是 MacOS 上最好的开源软件管理工具，最初由 Max Howell 在 2009 年用 Ruby 语言开发，现在则有十多个开发者一起维护其核心代码。用户不需要安装 Ruby，因为它早已在 MacOS 中预装好了。通过在终端运行下面的命令，可以安装 Homebrew：

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Homebrew 会将程序安装到其自己的目录下（在 Mac 中，通常位于 `/usr/local/Cellar` 目录下），然后将其可执行文件链接到 `/usr/local` 目录下。以 `wget` 程序为例，安装时执行：

```
$ brew install wget
```

`wget` 会被解压到 `/usr/local/Cellar/wget` 中：

```
Cellar
└─ wget
```

```

└── 1.18
    ├── README
    ├── bin
    |   └── wget
    └── ...

```

而在 `/usr/local/bin` 目录中，会创建一个软链接 `wget` 指向 `/usr/local/Cellar/wget/1.18/bin/wget`，可以像下面那样来查看链接情况：

```

$ cd /usr/local
$ ls -l bin | grep wget
wget -> ../../Cellar/wget/1.18/bin/wget

```

Git



Git 是一款强大的分布式版本管理系统，Linus Torvalds 在 2005 年 3 月开始开发，最初是用于管理 Linux 的内核代码。2005 年 6 月，Linus 将 Git 项目的维护权转交给了 Junio Hamano，后者向 Git 贡献了最多的源码。

我们上面已经安装好了 Homebrew，那么安装 Git 就轻而易举了：

```

# 安装 git
$ brew install git

# 查看当前系统里的 git 版本，以确认其是否已经安装好
$ git version
git version 2.11.0 (Apple Git-81)

```

如果系统里已经安装了 git，那么可以检查一下其是否为最新版本，如果不是，可以按照下面的方法来升级 git：

```

$ brew outdated
$ brew upgrade git

```

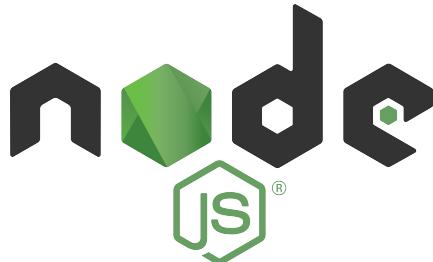
除了使用 Homebrew 安装，还可以直接在 Git 官网下载适合自己系统的二进制包进行安装。

参考资料

1. [Iterm2 官网](#)
2. [Git 官网](#)

Node.js 环境

Node.js^[1, 2]是一个跨平台的 JavaScript 运行时环境，由 Ryan Dahl 在 2009 年开发出来。在过去的8年里，Node.js 及其生态圈的发展可以用突飞猛进、日新月日来形容。前端开发者通常用 Node.js 来进行文件构建（例如 gulp、webpack 一类的构建工具）、提供页面服务（例如 express.js、koa.js 这样的服务器端框架），或者写一些网络、计算类的程序（例如爬虫）。



用 NVM 管理 Node.js

我们可以直接下载安装 Node.js，这样的话操作系统里共用一个 Node.js 程序。但是 Node.js 社区异常活跃，其开发迭代的速度非常快，更好的方式是使用 NVM（Node.js Version Manager，Node 版本管理工具）来管理系统里的 Node.js，可以同时安装任意多个版本，然后在使用时指定需要的 Node.js 版本即可。安装 NVM（可以在其官网上找到具体的命令，地址：<https://github.com/creationix/nvm>）：

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh | bash
```

安装好 NVM 后，就可以方便地使用 nvm 命令安装或切换当前会话使用的 Node.js 版本。NVM 的常用命令：

```
# 从服务器查询目前所有可安装的 Node 版本
nvm ls-remote

# 查看当前安装的 nvm 程序的版本，如果安装正确，会打印 0.33.8
$ nvm --version

# 下载并安装某个版本的 Node.js
$ nvm install 9.7.0

# 查看当前系统已安装的所有 Node.js 版本
$ nvm ls

# 在当前的会话中，使用某个已安装了的 Node.js 版本
$ nvm use 9.7.0

# 查看当前所用 Node 版本
nvm current

# 设置默认的 Node 版本。这个命令会在 /.nvm/alias/default 文件中写入默认的 Node 版本号
nvm alias default 9.7.0

# 如果某个版本的 Node 因为太古老等原因，你不再使用它，可以这样卸载掉
nvm uninstall 0.10
```

NVM 工作原理

TODO

NVM 实际上是一系列 shell 脚本的集合。

.nvmrc 描述文件

在项目的根目录下，可以通过 `.nvmrc` 文件来指定当前项目希望使用的 Node.js 版本。例如：

```
# 在项目的根目录下运行
$ echo "9.7.0" > .nvmrc
$ nvm use
Found '/path/to/project/.nvmrc' with with version <9.7.0>
Now using node v9.7.0 (npm v5.6.0)
```

nvm 的更多用法可以参考其官网^[3]，或者运行 `man nvm` 来获得帮助。

NPM - Node.js 包管理工具

参考资料

1. [Node.js 官网](#)
2. [Node.js | wikipedia](#)
3. [nvm | github](#)

代码编辑器

工欲善其事，必先利其器。

——孔子，《论语·卫灵公》

Sublime Text



Sublime Text 是一套跨平台的文本编辑器，支持基于Python的插件。Sublime Text 是专有软件（意味着在使用、修改上有所限制），可通过包（Package）扩充本身的功能。大多数的包使用自由软件授权发布，并由社区建置维护。Sublime Text 由 Jon Skinner 开发与维护，初始版本于 2008 年发布，目前最新的版本是 Sublime Text 3。

英文 `sublime` 意为“卓越的，令人惊叹的”。正如其名，Sublime Text 有着许多让人喜欢的功能：

- “Goto Anything”功能：可快速跳至文件、符号或行数
- “Command Palette”功能：弹性快捷键功能
- 多行编辑：用户可一次选择多行并进行同步编辑。
- 基于 Python 语言的外挂 API
- 针对个别项目使用不同的编辑器设置
- 通过 JSON 文件自定义设置值
- 多面板编辑
- 高性能
- 跨平台（Windows、Linux 和 Mac OS X）
- 代码摘要：用户可替常用的代码片段指定关键字快速插入。

Vim



Vim 是从 vi (Bill Joy) 发展出来的一个具有 TUI (Text User Interface，文本用户界面，与 GUI，即图形用户界面相对应) 的编辑器。代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用。Vim 的第一个版本由布莱姆·米勒 (Bram Moolenaar) 在 1991 年发布，源码主要由 C 语言写成。最初的简称是 Vi IMitation，随着功能的不断增加，正式名称改成了 Vi IMproved。现在是在开放源代码方式下发行的自由软件。^[4]

Vim 对重复性操作进行了很好的优化。“对于 Vim 高手来说，Vim 能以思考的速度编辑文本”，这是对 Vim 最流行的描述。即便不把它作为自己的常用编辑器，在登录到服务器上的时候，Vim 很可能就是你唯一可用的编辑器。所以，Vim 应当作为前端程序员的必备技能之一。

MacOS X 和 Linux 系统都预装了一个可以开箱即用的轻量级 Vim。可以通过输入 `vim` 命令查看当前所用 Vim 的版本。当前的最新版本是7.4，如果要想获得更多的功能，最好还是重新安装一下：

```
# Mac OS X
```

```
$ brew install vim

# Linux Ubuntu
$ sudo apt-get update
$ sudo apt-get install vim
```

Vim 有四种常见模式 (mode) :

- 普通模式——Vim 的默认状态，在这里可以进行高效率的操作
- 插入模式——一种所见即所得的状态。在普通模式下，按 `i`、`a`、`A` 等键均可进入插入模式；而按 `esc` 键，或者按组合键 `ctrl + [` 则可以退出插入模式，回到普通模式。
- 可视模式——可视模式允许我们选中一块文本区域并在其上进行操作。
- 命令行模式——在命令行模式中可以输入会被解释并执行的文本。例如执行命令 (`":"`键)，搜索 (`"/"`和`"?"`键) 或者过滤命令 (`"!"`键)。

在不同的模式下，即便相同的按键，也可能有不同的结果

Emacs



Emacs 在 1970 年代诞生于 MIT 人工智能实验室 (MIT AI Lab)，源码主要是使用 ELisp 写成。Emacs 不仅仅是一个编辑器，它本质上是一个集成环境，进入到编辑器后，用户会感觉如同进入了一个功能完备的操作系统中去，因为你可以在这里收发电子邮件、通过 Telnet 登录远程主机、上 Twitter、玩游戏以及浏览网站等等。

注：在 Unix 文化里，Emacs 是程序员们关于编辑器之战的两大主角之一，另一个主角是 Vim/vi。

VSCode

VSCode 全称 Visual Studio Code，于 2016 年正式发布。微软在互联网浪潮中曾一度被大家遗忘，近些年则十分热衷于开源项目。不但开源了自家最新的浏览器 Edge 的所有代码，还精心打造了 VSCode 这样一款非常适合前端开发人员的编辑器。与 Sublime Text、Vim 相比，VSCode 是经典的集成开发环境 (IDE)：

参考资料

1. [Sublime Text 官网](#)
2. [Sublime Text | wikipedia](#)
3. [Vim 官网](#)
4. [Vim | wikipedia](#)
5. [Emacs | wikipedia](#)
6. [VSCode | wikipedia](#)

第二篇

恩格斯说：“社会一旦有技术上的需要，则这种需要就会比十所大学更能把科学推向前进”。这句话用来描述 Web 前端技术近十年的发展，恰到好处。

调研内容

- HTML
- CSS
- JavaScript
- web components 标准
- Web 组件化历史
- Web 工程化历史

成书风格

技术/科学

商业

人文

模块系统概述

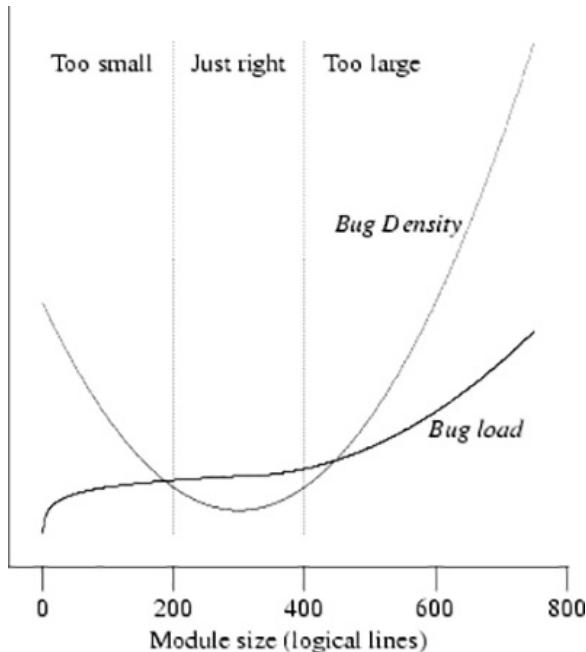
JavaScript 模块系统旨在解决大中型前端应用脚本管理与加载问题。

- 刀耕火种时代
- CommonJS
- AMD 时代
- ESM 时代

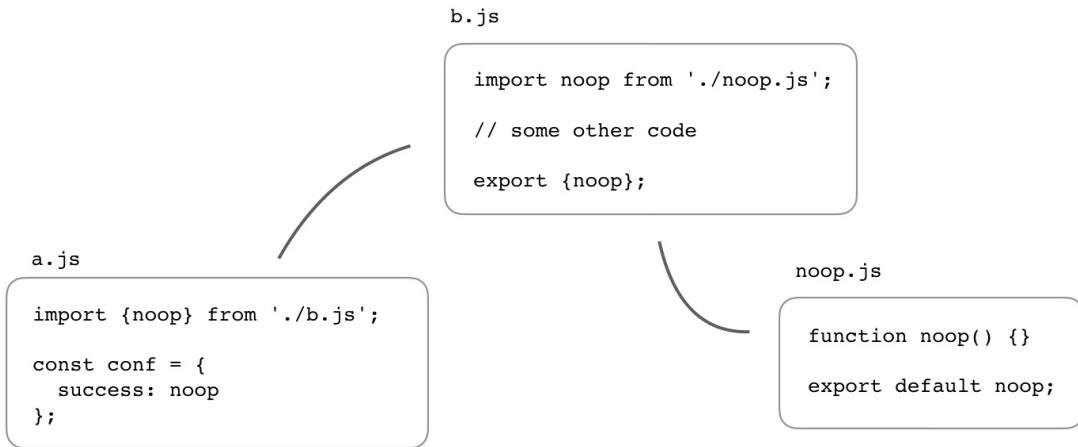
模块系统的重要性

减少 bug 率

Hatton^[1] 研究统计了代码行数与出现的 bug 数量的关系，给出了下面的曲线。结论之一是，一个模块的代码逻辑行数在200~400行之间是最佳状态，此时缺陷的密度最低。而且，Hatton 的研究发现，这个行数与所使用的编程语言无关。如果考虑到必要的注释、空行等，一个可读性好、思路清晰、易于维护的模块文件代码总行数通常在400~800行之间。



模块的代码行数太小，也会带来 bug 密度的提升，主要是因为模块与模块之间的 API 成本相应增加导致。例如，在一个模块中，当你需要通过追溯多个模块才能发现某个方法、依赖的来源时，这种成本就可以理解为模块间API的成本。



过多的跨模块查找，形成了模块间“沟通”成本。这会干扰你的注意力，使得 bug 产生的概率增加。

提升复用性

Don't Repeat Yourself. (不要重复自身)

-- The Pragmatic Programmer. 《程序员修炼之道》

《Unix编程艺术》里提到模块的正交性特点：在纯粹的正交性设计中，任何操作均无副作用；每个操作只改变一件事情；要修改任何一个属性，有且仅有一个方法可以使用。

“正交”是非常优雅的一个词语。在欧几里得空间里，两个向量点积为零则为正交；在物理学中，正交表现为运动的独立性。

参考资料

1. IEEE Software. Les Hatton. "Re-examining the Defect-Density versus Component Size Distribution". March/April 1997.

刀耕火种时代

CommonJS 模块系统

简史

JavaScript 是一门非常强大的面向对象语言，拥有多个执行最快的动态语言解释器。官方的 JavaScript 标准定义了许多便于开发浏览器应用的对象 API。JavaScript 是如此灵活而强大，很多人期望可以用它做更多的事情，但是当时的这个官方 JavaScript 标准未能给出明确的解决方案。

2009 年 1 月，Kevin Dangoor 在一篇题为 What Server Side JavaScript needs [2] 的博客文章里提到，他已经在 Google Groups 上面创建了名为 ServerJS 的讨论区，希望大家就 JavaScript 的模块系统、文件系统、包管理系统等对于打造良好的开发者生态非常重要的系统的代码、API设计等进行讨论。这个讨论区很火热，8天内就有 224 人发表了 653 条消息。ServerJS 规范的发展也非常迅速，2009年3月，推出了 CommonJS API 0.1，同年 8 月，这个组织正式改名为 CommonJS，以表示其对通用性的期望。

长远来看，CommonJS 希望自己能够建立起像 Python、Ruby 和 Java 那样丰富的标准库。使用兼容 CommonJS 的系统，开发者可以写出这些类型的应用[1]：

- 服务器端 JavaScript 应用
- 命令行工具
- 桌面级的 GUI 应用
- 混合式应用程序（例如 Titanium、Adobe AIR）

基本概念

参考资料

[1] CommonJS官网, <http://www.commonjs.org/> [2] [What Server Side JavaScript needs]
(<http://www.blueskyonmars.com/2009/01/29/what-server-side-javascript-needs/>)

AMD 模块系统

AMD 模块系统的经典实现库是 require.js(<http://requirejs.org/>)。)

基本使用

首先我们来看一下 require.js 的使用。考虑如下目录结构的静态页面服务：

```
root-dir
├── amd-demo      // 示例页面
│   ├── index.css
│   ├── index.html // 入口页面
│   └── main.js    // 入口 JS
└── common.css
└── js             // 公共脚本目录
    ├── jquery-3.2.1.js
    ├── jquery-3.2.1.min.js
    ├── lodash.min.js
    └── require.js // 模块加载器
```

amd-demo 目录是我们的示例页面目录， index.html 如下：

```
<!DOCTYPE html>
<html>
<head>
  <title>AMD 模块系统示例</title>
  <meta charset="utf-8">
</head>
<body>
  <h1>AMD 模块系统示例</h1>
  <pre id="console"></pre>
  <div data-role="jquery-container"></div>
  <script data-main="/amd-demo/main.js" src="/js/require.js"></script>
</body>
</html>
```

```
/***
 * /amd-demo/main.js
 */

requirejs.config({
  paths: {
    'jquery': '/js/jquery-3.2.1'
  }
});

require([
  '/js/lodash.min.js',
  'jquery',
], function(_, $) {
  var str = 'I am using main.js, and I am loaded.';

  if (_ && (typeof _.chunk === 'function')) {
    var res = _.chunk(['a', 'b', 'c', 'd'], 2);
    str += '\nLodash.js seems fine too. \n_.chunk gives this result: ' + JSON.stringify(res);
  }

  console.log(str);
  document.getElementById('console').innerHTML = str;
});
```

```
if (typeof $ === 'function') {
    $('[data-role="jquery-container"]').html('jQuery is loaded too.');
}
});
```

利用 require.js 进行模块加载的关键代码是：

```
<script data-main="/amd-demo/main.js" src="/js/require.js"></script>
```

这行代码在浏览器里运行，会先后发生这些事情：

1. 它首先根据指定的目录 `/js/require.js` 加载 `require.js`
2. 加载完成后，`require.js` 会在所有的 `script` 标签里寻找 `data-main` 属性指定的路径文件作为下一个将要加载的脚本，在这里，就是 `/amd-demo/main.js`
3. 开始加载 `/amd-demo/main.js`，加载后，解析并执行它，如果发现它有其他依赖，那么先加载所有的依赖（递归）并完成后，再执行 `/amd-demo/main.js` 里面 `require` 的回调函数。

关键技术与原理

脚本加载

依赖分析、任务队列与递归加载

UMD 模块系统

鉴于存在 CommonJS、AMD 等不同的模块系统，为了让代码能够同时支持它们，社区提出了一种统一模块定义（Universal Module Definition，UMD）来解决不兼容的问题。

示例

一个常见的 UMD 模块声明实际上是一个立即执行函数表达式。模块的主体在一个工厂方法里面，其返回值作为模块最终暴露的对象。例如下面的模块暴露了一个构造函数 `Time`：

```
;(function (global, factory) {
  typeof exports === 'object' && typeof module !== 'undefined' ? module.exports = factory() :
  typeof define === 'function' && define.amd ? define(factory) :
  global.Time = factory();
}(this, function () {
  // 模块工厂方法开始
  'use strict';
  var _private = '';

  function Time(param) {
    this._date = new Date(param);
  }

  return Time;
  // 模块工厂方法结束
}));
```

分析

通常，如果一个变量在取右值时未定义，会发生引用错误（Reference Error），例如

```
// 标识符 an_undefined_token 不在当前作用域链上
console.log(an_undefined_token);

// 会报如下错误
// Uncaught ReferenceError: an_undefined_token is not defined
```

但是 `typeof` 运算符有所不同，`typeof an_undefined_token` 并不会报任何错，而是输出 `undefined`。利用 JS 的这个运算符，我们可以在脚本加载后立即执行模块头部代码，利用特性检测来判断环境中存在的是哪种模块系统。

- 如果 `exports` 是个对象，而且 `module` 也存在，那么运行工程函数，拿到其返回值，然后像任何一个 NodeJS 模块一样，将返回值赋给 `module.exports`。
- 如果 `define` 是个函数，而且 `define` 上面存在 `amd` 属性（RequireJS 的特性，表示这是一个 AMD 加载器），那么使用 `define` 函数将模块工厂函数加到队列里。
- 如果上面两个特性检测都失败，就在全局对象上面挂载该模块的返回值。

UMD 模块试图对当前最流行的那些脚本加载器（例如 RequireJS）提供足够好的兼容性。很多情况下，它使用 AMD 为基础，并对特殊情况处理以提供 CommonJS 兼容性。

-- 译自 <https://github.com/umdjs/umd>

ECMAScript 模块系统

令人激动的是，ECMAScript 6 中点点滴滴的变化全都致力于解决开发者实际工作中遇到的问题。—— Nicholas C. Zakas

前面所提到的所有模块化解决方案，都是利用 JavaScript 语言本身的特性，实现的封装。而鉴于模块系统的重要性、必要性，TC39 委员会也对其标准化极为上心。2015 年推出的 ECMAScript 6 标准正式定义了 JavaScript 的模块系统。

工作原理

模块文件只加载、执行一次。

ESM 的实现状态

Node.js

2017年9月份，Node.js 发布了 8.5.0 版本。从这个版本开始^[1]，开发者可以通过开启试验特性 `--experimental-modules` 来使用原生的 ESM 模块系统，而且该 JS 文件必须以 `.mjs` 作为自己的文件扩展名。官方的计划是，预计到 Node.js 10 LTS 版本会默认支持 ESM，开发者就不必再借助于命令行参数来开启它了。

CommonJS 模块的 `__dirname`、`__filename` 也不会作为全局变量提供给模块。

Node.js 对 CommonJS 与 ESM 进行了较严格的区分，CommonJS 的 `require()` 不可用于加载 ESM 文件。

```
# 不支持的写法
require('./foo.mjs');
```

Node.js ESM 示例

文件目录：

```
.
├── lib.mjs
└── my-app.mjs
```

`lib.mjs` 的内容：

```
console.log('lib module is loaded into memory.');

export var name = '阿珂';

export function setName(n) {
    name = n;
}

export function add(x, y) {
    return x + y;
}
```

`my-app.mjs` 的内容：

```
import {name} from './lib.mjs';
import {setName, add} from './lib.mjs';
```

```
console.log(name);
setName('李白');
console.log(name);

console.log('sum: ' + add(2, 3));
```

执行：

```
$ node --experimental-modules my-app.mjs
# 输出的结果:
lib module is loaded into memory.
阿珂
李白
sum: 5
```

从上面的输出结果，我们可以发现有两个特点：

- 虽然 `lib.mjs` 模块被引用了两次，但是它内部只被解析、执行了一次（只打印了一次 `lib module is loaded into memory.`）。
- 在 `my-app.mjs` 里调用 `setName` 方法，修改的实际上是 `lib.mjs` 里的 `name` 变量。`my-app.mjs` 里的 `setName` 只是对 `lib.mjs` 里的 `setName` 的引用。

浏览器

浏览器中的 ESM 模块不必以 `.mjs` 作为扩展名。但必须有正确的文件类型描述（`text/javascript` 或 `application/javascript`）。

未来可以期待的特性

动态加载

```
import()
```

参考资料

1. [Using ES modules natively in Node.js](#)

概述

依赖分析

从入口文件开始进行依赖分析，分析的结果通常是 JavaScript AST。

持续集成

Gulp

Babel

Facebook 推出的 ECMAScript 编译器 Babel^[1]名字源于巴别塔（Tower of Babel）的故事，这个故事则起源于《圣经·旧约·创世记》（Book of Genesis）第11章：

大洪水过后，人类使用着同一种语言，当时人们联合起来兴建希望能通往天堂的高塔；为了阻止人类的计划，上帝让人类说不同的语言，使人类相互之间不能沟通，计划因此失败，人类自此各散东西。此故事试图为世上出现不同语言和种族提供解释。



Babel.js 的愿景如传说中那座塔一样，使得我们可以直接按照最新的 ECMAScript 标准书写代码，然后由它来翻译为浏览器已普遍支持的 ECMAScript 5。本质上讲，Babel 就是一款 JavaScript 编译器。

Babel 并非第一个将源码编译为具有良好兼容性的 JavaScript 代码的编译器。在它之前，就有 CoffeeScript 这样的 ECMAScript 方言及配套预处理器存在^{[2][3]}。CoffeeScript 诞生于2009年，它借鉴了 Ruby、Python 与 Haskell 等语言中许多优秀的语法，例如箭头函数（Arrow Functions）、解构赋值（Destructuring Assignment）、异步函数（Async Functions）等等，增强了 ECMAScript 的简洁性和可读性。但是随着 ECMAScript 第6版在 2015 年正式发布，CoffeeScript 大部分特性都在标准中得到了支持，这款小巧而美的 ECMAScript 方言也算完成了它最重要的历史使命。

体验 Babel

可以在 Babel 的官网 (<https://babeljs.io/repl/>) 中体验编译前后代码的差异。

例如输入这样的一段代码：

```
const arr = [1, 2, 3, 4];
const brr = arr.map(ele => ele * ele);
```

设置输入语言为 es2016，目标代码的运行环境为浏览器（例如 > 2%，ie 11, safari > 9），那么 Babel 会将其编译为：

```
"use strict";  
  
var arr = [1, 2, 3, 4];  
var brr = arr.map(function (ele) {  
    return ele * ele;  
});
```

如果试图重新给常量 `arr` 赋值，那么 Babel 会在编译阶段进行报错，然后中断编译过程：

```
repl: "arr" is read-only  
1 | const arr = [1, 2, 3, 4];  
2 | const brr = arr.map(ele => ele * ele);  
> 3 | arr = 2;  
   | ^
```

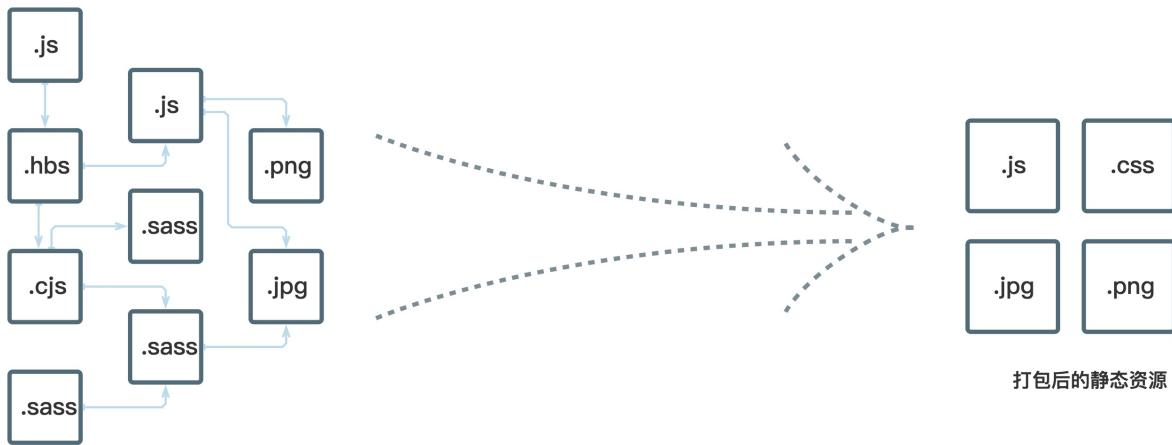
基本使用

工作原理

参考资料

1. [Babel 官网](#)
2. [CoffeeScript 官网](#)
3. [CoffeeScript | 维基百科](#)

Webpack



相互依赖的模块

Webpack 是现代 JavaScript 应用的静态模块打包工具。

代码的组织与管理

本章我们要讨论的问题是，如何组织、管理我们的所有前端代码（HTML、JavaScript、CSS）与资源（图片、字体、多媒体资源等），包括目录结构的设计，以及 CDN 网络的使用等等。

前端开发与调试

前端移动开发调试工具

- browser-sync
- eruda

safari + simulator

前端测试

Web 组件化开发实践

本部分介绍当前各种 web 组件化开发解决方案。

目录

- 基本概念与设计理念
- MPA 配置
- SPA 配置
- 原理

Web 应用分类

多页面应用（Multi Page Application, MPA）是最常见的 web 形式。MPA 产品又有两大类：一类是具有复杂的数据及界面交互的胖客户端类型，例如面向企业内部的 B 端产品；另一类是对响应速度要求高、客户端轻交互、重 SSR（Server Side Render，服务器端渲染）的轻客户端类型，通常为面向大众用户的 C 端产品（尤其是移动端）。

单页面应用（Single Page Application, SPA）

只打包浏览器端代码

无论代码组织结构如何，需要进行打包构建的，其实只有浏览器端代码。

面向对象 vs 函数式 以及代码复用

Joe Armstrong 认为，缺少复用是面向对象编程语言造成的，而不是函数式语言造成的。

因为面向对象编程语言的问题在于总是得到语言运行环境的所有隐含信息。你要的是香蕉，但看到的却是香蕉拿在大猩猩手里，并且后面还有整个丛林。如果代码具备引用透明性，如果是纯函数，即所有的数据都来自输入参数，所有的东西在离开时都不会留下任何痕迹，那么会达到惊人的复用效果。

MDV

MDV 是指模型驱动的视图变更模式（Model Driven View）。

面向对象的组件化开发

在各种前端 MVC/MVVM 框架出现之前，Web 前端开发人员可以通过 jQuery + AMD + 面向对象的设计方式，进行初级的组件化开发。

时钟示例

我们这里以一个简单的时钟组件为例，展示基于 jQuery 与 AMD 的组件化开发方式是怎样的。示例共有4个文件：

```
jquery-amd-clock
├── clock.js
├── index.css
├── index.html
└── index.js
```

HTML 文件：

```
<!DOCTYPE html>
<html>
<head>
    <title>jQuery+AMD组件化时钟示例</title>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css" href="/jquery-amd-clock/index.css">
</head>
<body>
    <h1>jQuery+AMD组件化时钟示例</h1>

    <div id="clock-container"></div>

    <script data-main="/jquery-amd-clock/index.js" src="/js/require.js"></script>
    <script type="text/javascript">
        requirejs.config({
            paths: {
                'jquery': '/js/jquery-3.2.1'
            }
        });
    </script>
</body>
</html>
```

下面是主 JavaScript 入口文件 `index.js`。做的事情比较简单：加载 `jquery`、`clock.js`，然后调用 `clock` 函数显示时钟：

```
define([
    'jquery',
    '/jquery-amd-clock/clock.js',
], function($, Clock) {
    var cl = Clock({
        container: $('#clock-container')[0],
    });
});
```

时钟组件 `clock.js` 文件：

```
define([], function() {
    /**
     * 如果num小于10，则在其左侧填充0
     */
```

```

    * @param {Number} num
    * @return {String|Number}
    */
    function leftPadding(num) {
        if (num < 9) {
            return '0' + num;
        }
        return num;
    }

    /**
     * 解析当前时间
     */
    function getTimeStr() {
        var time = new Date();
        var hour = leftPadding(time.getHours());
        var minute = leftPadding(time.getMinutes());
        var second = leftPadding(time.getSeconds());

        var year = time.getFullYear();
        var month = leftPadding(time.getMonth() + 1);
        var day = leftPadding(time.getDate());
        var str = `${hour}:${minute}:${second}`;

        var weekZNArry = ['日', '一', '二', '三', '四', '五', '六'];
        var weekDesc = '周' + weekZNArry[time.getDay()];

        return {
            weekDesc: weekDesc,
            date: `${year}-${month}-${day}`,
            time: str,
        };
    }

    /**
     * 最终被输出的构造函数
     * @param {Object} [props] 形如:
     * {
     *   container: DOMElement,
     * }
     */
    function Clock(props) {
        if (!props || !props.container) {
            throw new Error('请指定时钟的容器元素');
            return;
        }
        var container = props.container;

        var $div = document.createElement('div');

        function display() {
            var now = getTimestr();

            // 这里我们借助 ES6 的模板字符串，来声明所需要的 HTML 片段
            $div.innerHTML =
                `

<div>${now.time}</div>
                    <div style="font-size: 12px; font-weight: normal; padding-top: 5px;">${now.date} + ' ' + now.weekDesc

`;
        }

        display();
        container.appendChild($div);

        setInterval(function() {
            display();
        }, 500);
    }

```

```

    console.log('时钟已创建');
}

return Clock;
});

```

最后是 CSS :

```

.my-clock {
  display: inline-block;
  border: 1px solid #ccc;
  border-radius: 5px;
  padding: 10px;
  margin: 10px;
  font-size: 24px;
  font-weight: bold;
  font-family: monospace;
}

```

效果:



分析

`clock.js` 模块输出的是一个函数 `clock`，要在文档里显示一个时钟，用户只需要调用这个函数，传入要显示时钟的容器元素对象就可以了。使用者不需要关心如何操作 DOM，这个细节由 `clock` 函数完成了。

组件 `clock.js` 做到了基本的逻辑功能封装（数据变更、模板生成、视图渲染等），不过数据到视图的渲染过程，依然脱离不了对 DOM 的操作。在应用程序比较简单的时期，这种方案似乎总是能够满足开发者的需求。然而随着前端应用的复杂度提升，大量的 DOM 操作交织错杂，使得这种方式开发的组件并不好维护。

Web Components 概述

2011 年，Alex Russell 在 Fronteers 大会上首次提出“Web 组件”的概念。要知道，使用基于组件的 UI 库进行开发客户端应用，一直就是标准方法。而在 web UI 开发领域，原生的 HTML 组件功能非常有限。像 React/Vue 是借助于 JavaScript、在有限的 HTML 标记下实现的组件化。Web Components 则有所不同，它期望能够在 DOM 层面实现组件化，开发者可以扩展 HTML 标记。

Web Components 是 W3C 正在向 HTML 和 DOM 规范添加的一套功能^[1]，它允许在 Web 文档和 Web 应用程序中创建可重用的小部件或组件，开发者可以在更高的层次上组装自己的 web 应用程序。Web Components 规范实际上包括四大部分，它们可以单独或者组合使用。到目前为止，它们都仍然只是草稿阶段，没有成为标准。

四个子规范分别是：

- Custom Elements — 定义新 HTML 元素的 API
- Shadow DOM — 封装的 DOM 和样式，配以组合化。是 Web Components 的核心
- HTML Templates — 允许文档包含惰性的 DOM 块
- HTML Imports — 将 HTML 文档导入其他文档的声明方法

浏览器支持进展

自 2011 年 web 组件概念提出后，各浏览器厂商就已经纷纷开始着手进行相关功能的开发了。比往年好的事情是，各厂商之间出现了较多的合作。

Chrome：谷歌在这件事上依然走在最前面，Chrome 几乎完全实现了所有的 Web Components 规范，也侧面反映出它对 Web Components 标准化进程是多么上心。四个子规范都在 Chrome 中得到了原生的支持，不必借助任何 polyfill 方案。

Firefox 实现了 HTML Templates，并且允许开发者模式下开启 Shadow DOM 和 Custom Elements。FireFox 对 HTML 导入有所顾虑，因为他们觉得这个功能与 ES6 的模块化有太多重叠之处，打算观望一阵子。Regardless of this, Wilson Page from Mozilla concluded in a June blog post that “we’re optimistic the end is near. All major vendors are on board, enthusiastic, and investing significant time to help resolve the remaining issues.”

Safari (WebKit) 目前原生支持 HTML 模板、Shadow DOM 以及自定义元素 API。至于 HTML Imports，WebKit 与 FireFox 观点一致，即认为模板导入应该交给 ES6 的模块系统来处理，因此他们目前没有着手支持此特性。

微软的 Edge 13 浏览器开始支持 HTML 模板。

框架

Google 在 2013 年发布了一个基于 Web 组件的程序库“Polymer”^[3]。

参考资料

1. <https://fronteers.nl/congres/2011/sessions/web-components-and-model-driven-views-alex-russell>
2. Web Components W3C 进展. https://www.w3.org/standards/techs/components#w3c_all
3. <https://www.polymer-project.org/>
4. <https://vaadin.com/blog/web-components-in-production-use-are-we-there-yet->

自定义元素 (customElements)

纯展示组件：一个时钟挂件

我们首先来看一个极简单的组件，它由一个 HTML 文件（主文档）和一个组件脚本组成：

```
└── clock
    ├── index.html
    └── my-clock.js
```

我们的目标是使用自定义元素 (customElements) API，获得一个可以展示当前时间的日期组件，效果如下图所示：



时钟组件 `clock.js` 文件的内容如下：

```
/**
 * 时钟组件示例-v1
 */
function getTimeStr() {
  var time = new Date();
  return time.toString();
}

class MyClock extends HTMLElement {
  constructor() {
    super();

    var shadow = this.attachShadow({mode: 'open'});

    var text = document.createElement('span');
    text.textContent = getTimeStr();

    shadow.appendChild(text);

    setInterval(function() {
      text.textContent = getTimeStr();
    }, 250);
  }
}

customElements.define('my-clock', MyClock, {extends: 'p'});
```

在这段代码中，`customElements` 方法的语法是这样的：

```
customElements.define(name, constructor, options);
```

其中，

- `name` 表示自定义的标签名
- `constructor` 是每个新标签在创建的时候执行的构造函数
- `options` 可选，是一个对象，用于指定元素创建的参数。目前只支持一个，即 `extends`，用于表示希望自定义的元素继承哪个内置元素

我们可以对上面的组件JS代码进行优化，例如丰富时间展示的内容，使用 IIFE 来生成一个块作用域，以及在 Shadow DOM 内使用模板一次性插入更多的 HTML 片段：

```

    /**
     * 时钟组件示例-v2
     */
    (function() {
    /**
     * 如果num小于10，则在其左侧填充0
     * @param {Number} num
     * @return {String|Number}
     */
    function leftPadding(num) {
        if (num < 9) {
            return '0' + num;
        }
        return num;
    }

    /**
     * 解析当前时间
     */
    function getTimeStr() {
        var time = new Date();
        var hour = leftPadding(time.getHours());
        var minute = leftPadding(time.getMinutes());
        var second = leftPadding(time.getSeconds());

        var year = time.getFullYear();
        var month = leftPadding(time.getMonth() + 1);
        var day = leftPadding(time.getDate());
        var str = `${hour}:${minute}:${second}`;

        var weekZNArry = ['日', '一', '二', '三', '四', '五', '六'];
        var weekDesc = '周' + weekZNArry[time.getDay()];

        return {
            weekDesc: weekDesc,
            date: `${year}-${month}-${day}`,
            time: str,
        };
    }

    /**
     * 用于创建自定义元素的构造函数
     */
    class MyClock extends HTMLElement {
        constructor() {
            super();
            var shadow = this.attachShadow({mode: 'open'});

            var $div = document.createElement('div');

            var display = () => {
                var now = getTimestr();
                this.value = now;

                $div.innerHTML = `
                    <div>${now.time}</div>
                    <div style="font-size: 12px; font-weight: normal; padding-top: 5px;">${now.date} ${now.weekDesc}</div>
                `;
            }

            display();
            shadow.appendChild($div);

            setInterval(function() {

```

```

        display();
    }, 500);
}

getValue() {
    return this.value;
}
}

customElements.define('my-clock', MyClock, {extends: 'div'});
})();

```

使用时，只需在 `index.html` 里引入 `my-clock.js`，然后直接使用标签 `<my-clock></my-clock>`，或者 `<my-clock/>` 就可以了。如下所示。

```

<!DOCTYPE html>
<html>
<head>
    <title>纯展示组件：一个时钟挂件</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
    <style type="text/css">
        my-clock {
            display: inline-block;
            border: 1px solid #ccc;
            border-radius: 5px;
            padding: 10px;
            margin: 10px;
            font-size: 24px;
            font-weight: bold;
            font-family: monospace;
        }
    </style>
</head>
<body>
    <my-clock id="first-clock"></my-clock>

    <script type="text/javascript" src="../my-clock.js"></script>
    <script type="text/javascript">
        var firstClock = document.getElementById('first-clock');
        var val = firstClock.getValue();

        // 这两行都会打印形如 {weekDesc: "周日", date: "2018-02-25", time: "10:23:02"} 的对象
        console.log(val);
        console.log(firstClock.value);
    </script>
</body>
</html>

```

上面的 HTML 文件比较简单，我们使用了自定义的 `<my-clock></my-clock>` 标签，并且引用了定义了该标签对应的组件的 JavaScript 文件 `my-clock.js`。而且，由于在上面的 `clock.js v2` 版本里，我们给 `<my-clock>` 元素添加了自定义的属性 `value` 和方法 `getValue`，因此，在 HTML 在主脚本里，我们可以获取到 `<my-clock id="first-clock"></my-clock>` DOM 元素 `firstClock`，然后直接访问相应的自定义属性或方法。

自定义事件

TODO

浏览器支持情况

Custom Elements v1 是指 Supports "Autonomous custom elements" but not "Customized built-in elements"

Firefox (Gecko)	Chrome	IE	Safari	Opera
No support	59.0	No support	10.1	47
Android	iOS Safari			
62	10.3			

问题

- 样式是否被父文档修改? CSS
- 父文档如何与组件进行交互? 函数式

Shadow DOM

基本概念

Shadow DOM 是指，浏览器可以渲染一系列 DOM 元素，而不必把它们插入到主文档的 DOM 树结构中。

基于 Shadow DOM，可以实现基于组件的应用。它可以为网络开发中的常见问题提供解决方案：

- DOM 隔离：组件的 DOM 是独立的（例如，`document.querySelector()` 不会返回组件 shadow DOM 中的节点）。这意味着在主文档里，通过 `querySelectorAll`、`getElementsByName` 等方法，无法获取到 shadow DOM 内的任何元素。
- 样式隔离：shadow DOM 内部定义的 CSS 在其作用域内。样式规则不会泄漏至组件外，页面样式也不会渗入。
- 组合：为组件设计一个声明性、基于标记的 API。
- 简化 CSS：作用域 DOM 意味着您可以使用简单的 CSS 选择器，更通用的 id/类名称，而无需担心命名冲突。
- 效率：将应用看成是多个 DOM 块，而不是一个大的（全局性）页面。

示例

```
<html>
<head>
  <title>Shadow DOM</title>
</head>
<body>

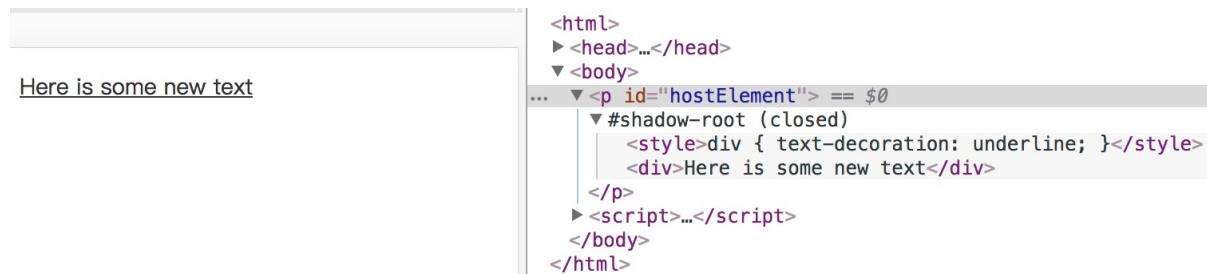
<p id="hostElement"></p>
<input type="text" name="hello">

<script>
  // 以 <p> 元素为根，创建 shadow DOM
  const p = document.querySelector('#hostElement'); // ①
  const shadowRoot = p.attachShadow({mode: 'open'}); // ②

  // ③
  shadowRoot.innerHTML =
    <style>div { text-decoration: underline; }</style>
    <div>Here is some new text</div>
  ;
  console.log('shadowRoot: ', shadowRoot);
</script>
</body>
</html>
```

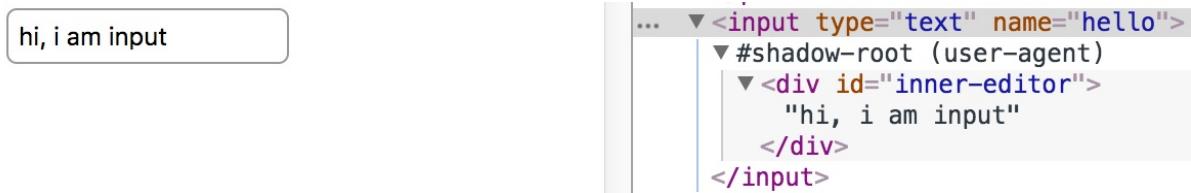
上面的代码里，做了这些事情：

- ①先获取到主文档里 `id` 为 `hostElement` 的 `<p>` 元素
- ②然后调用 `attachShadow` 方法，将其设置为一个影子树的根节点。
- ③在这个影子根节点下，插入 HTML 字符串，交给浏览器去生成对应的 DOM 结构。可以看到执行的效果：



可用元素列表

并非所有的 HTML 元素都可以托管影子树，例如常见的表单元素（`<input>`、`<textarea>`），实际上浏览器很早就使用它们进行了影子树的托管。



可以看到其节点信息里标注的影子树为 `#shadow-root (user-agent)`，意为浏览器自己实现的影子树。如果在这些元素上面执行 `attachShadow`，那么浏览器会提示错误：

```

document.getElementsByTagName('input')[0].attachShadow({mode: 'open'});
// Failed to execute 'attachShadow' on 'Element': This element does not support attachShadow

```

此外，让某些元素托管 shadow DOM 毫无意义，例如图片元素 ``。

<slot> 元素

HTML 的 `<slot>` 元素是 Web Components 技术的重要组成部分。`<slot>` 是组件内部的占位符，用户可以使用自己的标记来填充它，这样一来，我们可以创建独立的 DOM 树，然后将他们整合到一起。

浏览器支持情况

Shadow DOM v0 版本在 Chrome/Opera 浏览器中得到了支持，其他浏览器厂商则跳过 v0，直接开始实现 v1 版本的 Shadow DOM。

IE 11 及更早的 IE 浏览器均不支持此特性。微软的 Edge 浏览器对 Shadow DOM 的支持还处于考虑阶段。在 [1] 这里可以看到其对于此特性的描述与状态。

Firefox 对此特性的支持正在开发中。

Opera: Support since 40 Safari: partial support, 10.1+

iOS Safari, partial support, 10.2+

参考资料

1. <https://developer.microsoft.com/en-us/microsoft-edge/platform/status/shadowdom/>
2. <http://w3c.github.io/webcomponents/spec/shadow/>
3. Shadow DOM v1：独立的网络组件. <https://developers.google.cn/web/fundamentals/web-components/shadowdom?hl=zh-cn>
4. <https://webkit.org/blog/4096/introducing-shadow-dom-api/>

HTML Template

Web Components 规范里的 HTML 模板（HTML Template）是指浏览器新增对标签 `<template>` 的支持，以便用户能够在其中声明任意的 HTML 片段。

为何需要 `<template>` ?

模板最初是服务端技术栈的一部分，例如 PHP, Django, Ruby on Rails，都有各自的模板系统，用于将数据与HTML结构组合为浏览器端可识别的HTML文档。即便是后来的 Node.js，也有不少模板系统可用，例如 nunjucks、esj、jade、handlebars 等等。

目前的技术栈下，后端服务（不包括Node.js这样的视图类后端服务）越来越专注于数据处理，客户端则渐渐地承担更多的用户交互与视图渲染。各类前端 MVC 框架（例如 Angular.js, Backbone.js）都重度使用前端模板进行页面或组件渲染。在没有 `<template>` 的时候，声明模板有下面3种方式常见方式。

使用 `<div>` 标签

第一种方式是声明一个不可见的 `<div>` 标签，将模板放置于其中，如下所示。这个方式有个致命问题：其内部如果有图片、样式、脚本，那么都会被浏览器解析并且按相应规则加载或执行，即使这些资源都还没有用到。

```
<!-- 模板容器 div 内联样式设为 'display:none' 以避免显示在页面中 -->
<div style="display:none;">
<div>
  <h1>Web Components</h1>
  
</div>
</div>
```

使用 `<script>` 标签

另外一个办法是使用 `<script>` 标签并且将其 `type` 声明为 `text/template` 而非 `text/javascript`。

In the following example, the template content is stored inside of a script tag. The down side of this approach is that the templates will be converted into DOM elements using `.innerHTML`, which could introduce a cross site scripting vulnerability if an adequate sanity check is not performed.

```
<script type="text/template">
<div>
  
</script>
```

JavaScript字符串

这个大家再也熟悉不过了，就是在 JavaScript 代码里进行大量的字符串拼接，生成 HTML 字符串，然后通过设置 `innerHTML` 的方式将其注入到 DOM 树中。

示例

下面是一个使用 `<template>` 编程的示例。

```
<div id="host">加载中.....</div>
```

```

<template id="template">
  <style>
    img {
      border-radius: 4px;
      width: 200px;
    }
  </style>
  <div>
    <h2>模板内容-《指环王》海报</h2>
    
  </div>
</template>

<script>
  var template = document.querySelector('#template');

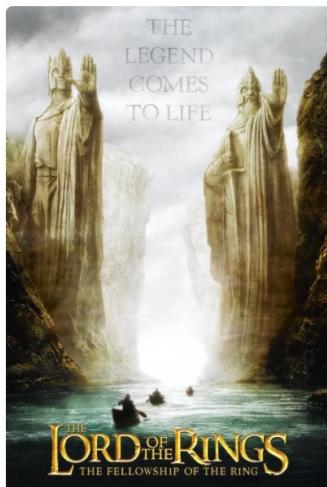
  setTimeout(function() {
    var clonedNode = document.importNode(template.content, true); // ①
    var host = document.querySelector('#host');
    host.innerHTML = '';
    host.appendChild(clonedNode);
  }, 2000);

  // ② 查看 template 的类型描述
  console.log(String(template));           // "[object HTMLTemplateElement]"
  console.log(String(template.content)); // "[object DocumentFragment]"
</script>

```

上面的代码在浏览器中解析执行后，会先显示“加载中.....”文字，约2秒后，模板里的内容被插入到主文档里进行渲染。可以看到 `<style>` 会对主文档有影响。

模板内容-《指环王》海报



```

<!DOCTYPE html>
<html>
  <head>...</head>
  ...<body> == $0
    <div id="host">
      <style>
        img {border-radius: 4px; width: 200px;}
      </style>
      <div>
        <h2>模板内容-《指环王》海报</h2>
        
      </div>
    </div>
    <template id="template">
      <#document-fragment>
        <style>
          img {border-radius: 4px; width: 200px;}
        </style>
        <div>
          <h2>模板内容-《指环王》海报</h2>
          
        </div>
      </template>
    <script>...</script>
  </body>
</html>

```

① 处用到的 `document.importNode()` 方法，会接收模板内容节点，然后返回一个该节点的深拷贝（第二个参数 `true` 表明了要使用深拷贝）。这有点类似于 `document.createElement()`。

② 处我们通过调用 `template` 和 `template.content` 的 `toString()` 原型方法（`string(some_object)` 的作用就是如此），来查看该对象的字符串描述。可以了解到，模板元素 `<template>` 继承自 `HTMLElement`，而 `<template>` 的属性 `content` 则继承自 `DocumentFragment`（这个类的细节可以参考^[2]）。这个细节也可以从 W3C 的接口定义中一窥大概（参考资料^[3]）：

```
// W3C 使用接口定义语言（Interface Definition Language, IDL）来描述元素的定义
```

```
[Exposed=Window,
HTMLConstructor]
interface HTMLOptionsElement : HTMLElement {
  readonly attribute DocumentFragment content;
};
```

参考模板元素的IDL，我们可以用很简单的方式来判断浏览器是否支持 `<template>`：

```
function ifSupportTemplate() {
  return typeof HTMLOptionsElement === 'function';
}
```

关于数据绑定

大多数前端 MVC/MVVM 框架的模板都支持数据绑定，对开发非常方便。`<template>` 则不支持这样的特性，它只是提供了模板内容的隔离而已。

浏览器支持情况

HTML template 是四个子规范中完成最早的，已经是 2014 年发布的 HTML5 标准的一部分。所以到目前，总体来说，`<template>` 元素在各浏览器中的支持度很不错^[4]。除了 IE/Edge，其他的浏览器的近期版本都可以放心地使用 `<template>` 特性。

Firefox (Gecko)	Chrome	IE	Safari	Opera
22	35	均不支持	10.1	22
Android	iOS Safari	Edge		
4.4	9.2	13, 部分支持		

参考资料

1. [Introduction to the template elements | webcomponents.org](#)
2. [Interface DocumentFragment | w3c](#)
3. [HTML5.2-template](#)
4. [caniuse template](#)

HTML Import

HTML Import (HTML 导入) 可以将外部的 HTML 文档引入到当前页面中，并对当前页面提供完全的 DOM 访问。

基本用法

目录结构：

```
static          # 静态文件服务根目录
└── html-import
    ├── index.html   # 主文档
    └── part.html    # 外部文档
```

HTML Import 需要通过声明了 `rel="import"` 属性的 `<link>` 元素来导入外部 HTML 文档。而且无法通过文件协议 (`file:///`) 访问，必须是 HTTP 或者 HTTPS。

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
  <title>HTML Imports</title>
  <link rel="import" href="./other.html">
</head>
<body>
  <h1 style="padding: 20px;">HTML Imports Demo</h1>
  <div id="part-container"></div>
</body>
</html>

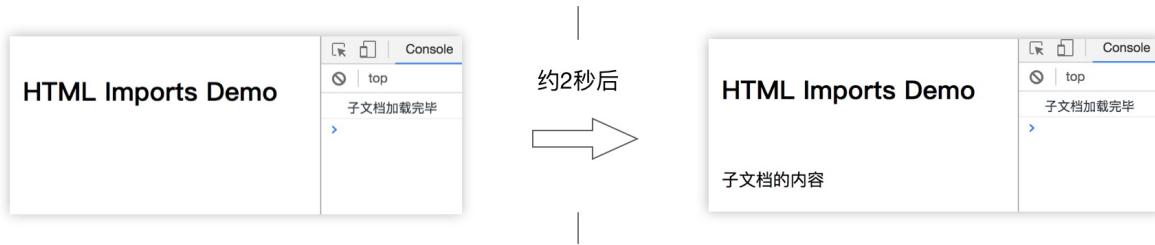
<script type="text/javascript">
(function() {
  var linkEleList = document.querySelectorAll('link[rel="import"]');
  var extDocOther = linkEleList[0].import;

  setTimeout(function() {
    document.getElementById('part-container').appendChild(
      extDocOther.querySelector('div').cloneNode(true)
    );
  }, 2000);
})();
</script>
```

被引用的 HTML 文档：

```
<!DOCTYPE html>
<html>
<head>
  <title>一个子文档</title>
</head>
<body>
  <div style="padding: 20px;">子文档的内容</div>
</body>
</html>
<script type="text/javascript">
(function() {
  console.log('子文档加载完毕');
})();
</script>
```

在本地的服务中访问 `index.html` 页面，观察控制台的输出，可以看到立即会打印出“子文档加载完毕”，约2秒后，主文档中显示出“子文档的内容”。效果如下图所示。



跨域引用

除了使用相对路径来引入同一个域下的 HTML 页面，还可以使用 URL 引入其他域下的文档，例如：

```
<link rel="import" href="http://borninsummer.com/">
```

样式影响

目前的草案里，有一章节是“9. Style processing with Imports”，描述了被引用的文档的样式要对主文档产生影响。这种样式的变化通常出乎意料，可能会对主文档造成较严重影响。因此，草案的撰写人们计划将这一章节从 HTML Imports 提案中移除 (<https://github.com/TakayoshiKochi/deprecate-style-in-html-imports>) 。

在上面的示例中，如果 `other.html` 中用 `<style>` 元素声明了一些样式规则，例如

```
<style type="text/css">
/* 给所有的文字添加阴影效果 */
body {text-shadow: 0 0 2px #333;}
</style>
```

那么主文档也会立即在引入 `other.html` 文档后被应用其中的样式规则（在 Chrome65 版本中测试）：



TakayoshiKochi 称，在 Chrome 67 版本之后，被引入的文档的样式将不会再应用于主文档^[1]。

事件

浏览器支持情况

Firefox (Gecko)	Chrome	IE	Safari	Opera
No support	61.0	No support		Support
Android	iOS Safari			

参考资料

1. <https://github.com/TakayoshiKochi/deprecate-style-in-html-imports/issues/5>

Polymer

React 组件化开发

React 诞生于 2011 年，由 Facebook 的一位员工 Jordan Walke 发明。在 Facebook 的新闻流产品中使用了一段时间后，Instagram 也使用该框架进行了部分业务构建。2013 年 3 月，在美国 JS 开发者大会上，Facebook 将 React 的代码正式开源。React 只处理用户界面相关的逻辑，对应着 MVC（Model-View-Controller）架构里的 V（View）。在众多框架中，React 利用纯粹的 JavaScript 践行着组件化的开发理念。

React 具有这些著名的特性：

- 单向数据流
- 虚拟DOM
- JSX
- 不仅可以用于浏览器中
- React Native

起源

Jordan Walke 创造 React 的灵感来自 Facebook 的 PHP 框架 XHP^[1]。XHP 扩展了 PHP 的语法，使得在 PHP 代码中可以直接使用 XML 字面量来表达 HTML 模板/元素等，也可以很方便地自定义可复用的 HTML 元素。Facebook 将其在 GitHub 上进行了开源。后来统一交给 HHVM 团队进行维护。

使用传统的 PHP 语法，一个简单的页面可能是这样子：

```
<?php  
$href = 'http://www.facebook.com';  
echo "<a href=$href>Facebook</a>";
```

但是如果借助 XHP 扩展，页面可以这样写：

```
<?php  
$href = 'http://www.facebook.com';  
echo <a href={$href}>Facebook</a>;
```

区别主要有两点：

- 输出 HTML 片段时，不使用字符串，而是直接使用 XML
- HTML 片段里的变量，使用大括号 {} 进行引用

XHP 并非这种语法扩展的原创。E4X（ECMAScript for XML）可以说是在编程语言中使用 XML 的最早鼻祖，它在 2004 年进行了标准化，并且一度在各主流浏览器中得到了实现。不过，由于其规则过于复杂，在 JS 引擎中实现 E4X 功能经常出现问题，因此，这样的功能特性逐渐在浏览器中消亡。2014 年，Mozilla 将 E4X 标准彻底废除^[2]。

XHP 的这个特性实际上体现了服务器端开发对于可复用的自定义元素的诉求。这本应该是前端开发的范畴。后来，Facebook 的开发人员们也不遗余力，借助同样的语法扩展理念，推出了 JSX，以便在 JavaScript 中直接使用 XML 来声明或定义 HTML 元素。

JSX

Douglas Crockford 在《JavaScript语言精粹》一书里，指出了 JavaScript 的诸多设计缺陷，整理出一个“优雅”的子集。E4X 虽然已经不复存在，但是有些好的特性依然影响着后来的开发者。React 的 JSX 就是这样的产物：它不再期望被各浏览器厂商进行原生实现，而只是规定了一些方便书写 HTML 元素或自定义元素的语法接口。

鉴于 JSX 的通用性，2014年9月，Facebook 的官方开发团队发布了一份 JSX 规范。该规范的开头如此声明：

JSX 是基于 XML 风格对 ECMAScript 进行的语法扩展。它不打算被任何引擎或浏览器实现，也不必被纳入到 ECMAScript 标准当中。它应当通过各种预处理器（编译器，transpilers）转换为标准的 ECMAScript 代码。^[3]

在 React 中，下面这样的 JSX 代码，

```
var bar = <div className="sidebar"><strong>Hello World.</strong></div>;
```

将被编译为标准的 ECMAScript 代码（使用 babel 编译，需要设置 `babel-preset-react`）：

```
"use strict";

var bar = React.createElement(
  "div", // 标签名
  {className: "sidebar"}, // 属性
  React.createElement( // 子元素
    "strong", // 标签名
    null, // 属性
    "Hello World." // 子元素/内容
  )
);
```

关于 JSX 与 HTML/XML 的关系，Facebook 的 JSX 规范里如此解释：

这个规范（指 JSX）并不打算去遵循任何 XML 或者 HTML 规范。JSX 是作为 ECMAScript 的一种特性来设计的。只是为了让大家便于上手，才设计得为 XML 语法风格。^[3]

参考资料

1. [XHP: Introduction | HHVM](#)
2. [ECMAScript for XML | wikipedia](#)
3. [Draft: JSX Specification](#)
4. [WTF is JSX](#)

基本使用

虚拟DOM

Another notable feature is the use of a "virtual Document Object Model", or "virtual DOM". React creates an in-memory data structure cache, computes the resulting differences, and then updates the browser's displayed DOM efficiently.[14] This allows the programmer to write code as if the entire page is rendered on each change, while the React libraries only render sub components that actually change.

天生就慢的 DOM

从 JSX 到 DOM

在表达树形结构时，XML 风格的语法要比 JavaScript 的多层对象表达方式清晰地多。

组件间通信

服务端渲染

有时候，出于 SEO 等因素的考虑，需要在服务端进行页面渲染（即 Server Side Render, SSR）。借助虚拟 DOM，React 支持使用 `ReactDOMServer`（来自 `react-dom/server`）将组件树渲染为字符串。此时，组件就只会调用位于 `render` 之前的那些组件生命周期方法，即：

- `constructor`
- `componentWillMount`

原理

基本使用

我们来看一个最简单的服务器端应用示例。目录结构如下：

```
server
├── index.js          // koa 搭建的简单 HTTP 服务器
└── pages
    └── hello-ssr
        ├── index.js  // 页面入口
        └── part.js   // 一个组件
```

首先，我们用 Koa@2.* 构建一个简单的 HTTP 服务器。值得注意的是，Koa 依赖 node v7.6.0 或 ES2015 及更高版本和 `async` 方法支持。

`koa` 是由 Express 原班人马打造的，致力于成为一个更小、更富有表现力、更健壮的 Web 框架。

```
/**
 * 一个简单的 HTTP 服务器
 * 文件: server/index.js
 * node@9.4.0
 * koa@2.5.0
 */
const Koa = require('koa');
const React = require('react');
const Router = require('koa-router');
const ReactDOMServer = require('react-dom/server');

const app = new Koa();
const router = new Router();

router.get('/hello-ssr', (ctx, next) => {
  const PageFunc = require('../pages/hello-ssr/index.js').default;
  ctx.body = ReactDOMServer.renderToString(<PageFunc />);
});

app
  .use(router.routes())
  .use(router.allowedMethods());

const port = 7002;
app.listen(port);
```

我们的页面入口文件，`./pages/hello-ssr/index.js` 里可以像浏览器中组件的写法一样去定义或引用页面/组件：

```
import React from 'react';
```

```

import Part from './part.js';

export default class HelloWorldPage extends React.Component {
  constructor() {
    super();
    console.log('\n组件生命周期方法: constructor');
  }

  componentWillMount() {
    console.log('组件生命周期方法: componentWillMount');
  }

  componentDidMount() {
    console.log('组件生命周期方法: componentDidMount');
  }

  render() {
    return (
      <div style={{padding: '5px'}}>
        <h2>Hello World!</h2>
        <p>本页面使用 React 服务端渲染而成。</p>
        <Part />
      </div>
    );
  }
}

```

上面的页面引入了一个简单的组件， `part.js`：

```

import React from 'react';
import moment from 'moment';

export default class Part extends React.Component {
  constructor() {
    super();
  }

  render() {
    return (
      <div style={{border: '1px solid #ccc', borderRadius: '4px', padding: '5px'}}>
        <div>这是一个小组件。显示渲染页面时的服务器时间: </div>
        <strong>{moment().format('YYYY-MM-DD HH:mm:ss')}</strong>
      </div>
    );
  }
}

```

启动服务：

```

# 我们使用了 babel-node 来启动服务，它可以先将代码进行编译，然后在使用 node 运行
# 这里仅仅用于演示，生产环境中并不推荐直接使用
$ babel-node ./server/index.js

```

在 `http://localhost:7002/` 可以访问到服务端渲染出的页面：

Hello World!

本页面使用 React 服务端渲染而成。

这是一个小程序。显示渲染页面时的服务器时间：

2018-02-25 21:21:59

常见问题

组件生命周期

在组件中发起异步请求

性能评估

参考资料

1. 示例代码, <https://github.com/zilong-thu/react-ssr-intro>

React 与函数式编程

何为函数式编程

为什么函数式编程至关重要？

React 中的函数式编程思想

参考资料

1. John Hughes[瑞典]. Why Functional Programming Matters. 1984.
<http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html>
2. BYVoid. 函数式程序设计为什么至关重要
3. Side Effect | wikipedia, [https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

微信小程序组件化开发

前端开发中的常见话题

JavaScript 社区标准库

许多编程语言都有其官方推出的标准库，例如 C++ 有 C++ STL，Java 有 lang/util/io/socket/net/concurrent 包等各种基础包。标准库的意义在于它们对那些常见的高级数据结构进行了高质量的封装，使开发者得以开箱即用。

严格来讲，JavaScript 是不存在官方的标准库的。W3C 负责定义语言本身的标准。而由于 JavaScript 又不涉及 I/O，因此也不会像 C++/Java 那样有不少 I/O 库。但是，JavaScript 在浏览器里的应用规模如此巨大，社区基于那些最常出现的应用场景，提炼了许多优秀的 JavaScript 库，例如 jQuery。它们应用范围之广，已经可以称为“JavaScript 社区标准库”了。我们这里就列举一些常见的库。

- jQuery
- lodash.js
- moment.js
- RxJS
- d3.js
- three.js

编码规范与编码风格

有多个指标可以衡量我们编写的代码的质量：

- 正确性
- 可读性
- 性能

按照 Douglas Crockford 的理念，可读性一定比代码的速度更重要，可读性处在与正确性不相上下的地位。为了提高前端代码的可读性从而提高其可维护性，不少团队都会约定内部的编码规范，例如 Airbnb 在 Github 上推出的 JavaScript Style Guide。

要想在团队内统一编码规范也许并非易事，这很有可能会掀起关于个人编码风格的争论，但这些争论通常毫无意义。

我也看到过有人就风格的正确与否争论了半天，但这些解释都毫无意义，因为他们真正争论的是自己在学校里用的是什么、在第一份工作中又用到了什么风格，或者是影响过他的某人使用了哪种风格，那这种风格就是正确的，而其他则是错误的。

.....更重要的是每个人都能达成共识。

Douglas Crockford, 《编程人生》（上卷）

K&R 风格。

注释

编写可维护代码的关键点之一是注释。许多人都认同“自说明式代码”，意思是通过良好的变量命名、清晰的分支路线等让代码清晰。但这其实往往只适用于小范围的代码。就大型的函数或者模块的话，通过文档或文档式的注释来说明会更好（Brendan Eich 也持有类似的观点^[1]）。

ESLint

React/JSX 编码规范

Airbnb React/JSX Style Guide：

A mostly reasonable approach to React and JSX.

重构

第二系统综合征 (Second-System Effect)

设计一个系统有两种方式：一种是尽量简单，这样显然不会有什么问题；另外一种是，尽量复杂，这样没什么显然的问题。第一种方法其实更难。它需要从复杂的自然现象中发现简单的物理规律的那种技能、投入、洞察力，甚至是那种灵感，同时还需要你能接受你的目标受限于物理、逻辑和科技的约束，以及在目标间有冲突的时候可以妥协。—— Tony Hoar, 图灵奖获奖感言

在《编程人生》中，Joshua Bloch 有这样的观点：

在大多数情况下，程序员的时间比计算机的时间更宝贵。但是当你的程序运行在成千上万台机器上的时候，就完全不同了。所以我们写的有些程序（例如谷歌的索引服务器的内循环程序），使用那些可能不那么安全的语言，榨出每一点值得榨出的性能。

我要重点说说一类人，用 Kevin Bourrillion 的话来说，他们“缺乏感同身受的基因”。如果你不能把自己想象成使用你的 API、你的语言的普通程序员，那么你就没有办法做一个好的API或者语言设计者。

前端程序员的自我成长

关于学习技术

开发与学习过程中，会接触与使用到各种技术。它们在预期使用寿命、紧迫性等方面存在诸多差异。

预期寿命

先说一下所谓的预期使用寿命。这也与2015年底讨论的比较热的话题有点关系。

摩尔定律指出计算机性能大概每18个月会提高一倍。除了硬件，摩尔定律似乎也眷顾了软件开发领域：

每过 18 个月，就有一半的知识会过时。

有些工具、类库、方言从诞生到流行到凋亡，只有很短的时间。例如打包工具 Grunt，流行了一年后，社区基本上就转到 Gulp 麾下；而 Gulp 流行了一年多后，Webpack 又迅速壮大。Coffeescript 在一些开发者中也非常流行，它提供了非常简洁的编程语法，然后将其编译为（Coffeescript 程序员们认为笨拙的）原生 JavaScript。不过，随着 ECMAScript 2015 标准的正式发布，Coffeescript 其实算是完成了它的历史使命，即以社区力量推动标准化进程。

这一类技术，就是预期使用寿命比较短的技术——它们因标准或生产工具的不完善而生，最终要么因标准化而消亡，要么因为更优秀的替代者的出现而逐渐无人问津。

因此，在选择一个技术的时候，就要考虑它的预期使用寿命了：是否过一段时间，就要抛弃它，转向它的替代品？不可否认程序员是需要不断学习的一群孩子。可是如果经常要学习各种寿命短的相似工具，这个成本还是比较大的。

预期使用寿命长的，无非就是那些大家称为“基础”的技能点：例如 Git, JavaScript。Git 自 2005 年诞生至今 11 年，JavaScript 从 1996 年诞生至今已有 20 年，相信它们都会被社区长久地维护下去。那么这些就是应该作为基本功，认真去学的，最好是达到手到擒来的境界。

对前端程序员来说，在个人的自由时间里，应该优先选择学习这样的技术：

- 所有程序员应当具备的计算机科学核心知识，例如数学、算法、数据结构、系统设计等
- 已正式标准化或者即将标准化的，例如 ECMAScript 5/6/7, HTML5, CSS3
- 社区或者基金会长期维护的，例如 Git, Node.js, Linux
- 大公司长期维护的，例如 React（背后有 Facebook），AngularJS（背后有 Google 支撑）

紧迫性（或优先级）

这是跟业务开发相关的。项目中用到的那些技术，如果没有掌握，会成为影响开发进度、开发效率的因素，那就是应该具有较高的学习优先级。好比我刚来的时候，懂点JS，懂点Git操作，会点Mac操作，但就是没有接触过 React，而项目乃至整个团队都是完全基于 React 进行开发的，那么 React 自然具有最高优先级。

没有需求，也许就没有了程序员的价值。

如何提升

不断修正自己的认知

人不能两次踏进同一条河流

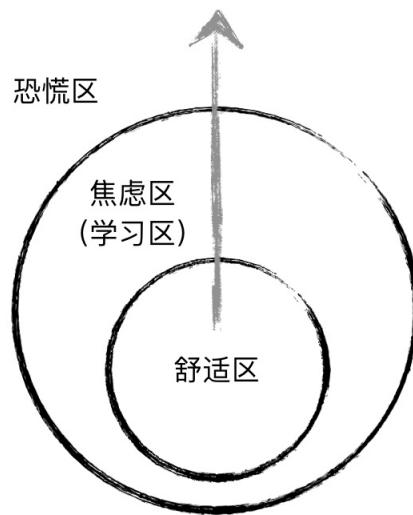
—— 古希腊哲学家赫拉克利特（约公元前530年~前470年）

程序员的编程行为是严谨、严肃的，容不得马马虎虎与模棱两可。自己的错误认知可能会给系统带来潜在的问题，甚至造成致命的打击。如果你的同事指出了他认为你理解有误的地方，那么应该去做调研、验证，而非停下来，展开一场基于已有认知的辩论。

不必因为自己过去理解错了而自惭形秽。最重要的是要敢于接受正确的理解，承认那时的自己是错误的。倘若心里有点不好受，可以这样安慰自己：那时的我不是现在的我，就如同那时的河流并非此时的河流，毕竟：“万物皆流，无物常住”。

时不时跳出舒适区

人的认知领域可以划分为这三个区域：舒适区（comfort zone），学习区（或焦虑区，learning zone），以及恐慌区（panic zone，也叫做危险区）。如下图所示。



人人都喜欢舒适区，这里阳光明媚，温度适宜，空气清新，工作起来非常高效率，甚至都不怎么用脑子。在这里的知识都是这个人掌握、运用得非常好的。就像一个学了一两年前端开发的同学A，对 JavaScript 的各种特性肯定是如数家珍。舒适区听上去很不错。但是停留在舒适区太久，水平原地踏步，绝对不是件好事。

最外面的是新领域，叫做“恐慌区”，这里的知识对这个人来说都是陌生的，完全没有掌握甚至没有接触的。就像那个同学A（假定TA就像我一样，是自学编程，因此只会前端开发），如果让他去用 Java 写后台，那很可能会很痛苦、无从下手。

稍微往舒适区外走几步，你会感觉到这里有些让人焦虑：有些技术，你听说过名字，但其实还没有完全理解它的工作原理，或者用得不熟练，然而项目中马上就要用到了。你感觉到一丝冷风吹过，天空好像没有那么明媚了。不过，好在你稍微了解过大概，可以花一两天的时间达到熟练的程度，看起来，可以应付得来。那么这个区域就是你的学习区，在这里，适当的未知引发你的焦虑，会刺激你学习新东西。征服了这个区域，阳光会很快明媚起来。好比如果这个同学懂 JavaScript，又要学后端开发，那么直接上手 Node.js，可能比先学 Java，然后开始做后端开发要稍微少点门槛（之所以说“稍微少点”，是因为后端开发不只是编程语言层面上的东西，编程语言只是服务器端开发的一个子集，理论上任何一门编程语言都可以实现一整套的服务器端逻辑，这个时候变的是语言，不变的是诸如HTTP、TCP、并发处理、页面渲染、文件分发、认证授权、会话管理、路由设计、并发处理等等）。

很多新技术、新工具，通常都会提供一个“学习区”，叫做“Get started”，或者“快速入门”，或者“Hello world”。通过看这些东西，就可以知道这个是不是处在自己的学习区内。

想要成为一个更厉害的编程高手，就要时不时跳出自己的舒适区，不断地去学习新东西。

不要放过技术细节

揪住一个点，挖下去，很可能多挖几层就碰到问题的根源了。而这正是提升自己的一个好方法。

读书

编程本身是严肃的工程行为，优秀的程序员应当具备相关开发领域足够多的知识。读书是建立系统的认知和理论体系的正统之道。选择书目的时候，不一定只关注纯技术类的话题，最好还要广泛涉猎计算机文化、艺术、人物、历史、未来等等方面。例如读《黑客与画家》以了解黑客文化；读《UNIX编程艺术》以领略 Unix 文化的魅力；读《代码的未来》可以让你了解今后编程语言可能的发展趋势；读《编程人生》（Coders at Work）则可以使你了解那些世界顶级的程序员们的成长经历、人生观念。

Stackoverflow 网站上有个著名的问题，“程序员最应该读的书有哪些？”^[1]。投票数最多的回答给出了一个长长的书单，前十名是：

1. Code Complete (2nd edition) by Steve McConnell. 《代码大全》
2. The Pragmatic Programmer. 《程序员修炼之道：从小工到专家》
3. Structure and Interpretation of Computer Programs. 《计算机程序的构造和解释》
4. The C Programming Language by Kernighan and Ritchie. 《C程序设计语言》
5. Introduction to Algorithms by Cormen, Leiserson, Rivest & Stein. 《算法导论（原书第2版）》
6. Design Patterns by the Gang of Four. 《设计模式》
7. Refactoring: Improving the Design of Existing Code. 《重构》
8. The Mythical Man Month. 《人月神话》
9. The Art of Computer Programming by Donald Knuth. 《计算机程序设计艺术》
10. Compilers: Principles, Techniques and Tools by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. 《编译原理》
(龙书)

这份书单对所有程序员都有参考意义。而在前端领域，个人推荐下列几本必读书目。

1. 《JavaScript高级程序设计（第三版）》
2. 《JavaScript权威指南》
3. 《JavaScript语言精粹》
4. 《高性能网站建设指南》
5. 《高性能网站建设进阶指南》
6. 《CSS权威指南》
7. 《HTTP权威指南》
8. 《深入理解ES6》
9. 《JavaScript忍者秘籍》
10. 《你不知道的JavaScript》系列

参考资料

1. [What is the single most influential book every programmer should read? | Stackoverflow 2.](#)

附录

附录A：Git 原理与使用基础

=> TODO: git 的原理、使用、工作流. 3000 字

Git 工作原理

基本使用

Git 工作流

在多人协同开发时，Git 工作流程就显得尤为重要。Vincent Driessen 在2010年的一篇文章《A successful Git branching model》^[1] 介绍了在这样的场景下，如何使用 Git 进行高效开发。

良好的提交信息

准确恰当的提交信息对于回溯项目发展历程、寻找特定功能的代码片段都很有帮助。

真正到了要写提交信息的时候，很多人可能会图省事，使用 `...`，或者 `abc` 这种方式蒙混过去。Git 的提交信息应该简明扼要。可以使用一些可以描述所做改动的关键词作为前缀然后书写详细的信息。

关键词有两类，一类是动词，表示进行的动作；另一类是名词，通常与业务相关。

下面是一些常见的动词类关键词：

- `add`，表示添加新功能
- `fix`，缺陷修复
- `update`，对已有功能进行更改或优化
- `remove`，删除部分代码、功能
- `refactor`，重构了一些代码、功能
- `workflow`，工作流程更改
- `chore`，琐碎的修改

```
# 新增了支持用户上传文件的功能
$ git commit -m 'add: 上传文件功能'

# 修复了一个线上问题
$ git commit -m 'fix: **页面数字取值问题'
```

此外，对于每个项目，可以用业务相关的名词作为关键词前缀。例如：

```
# docs 表示这是对文档进行的修改
$ git commit -m 'docs: 更新接口文档'
```

参考资料

1. Vincent Driessen. A successful Git branching model. <http://nvie.com/posts/a-successful-git-branching-model/>

Web 浏览器组成原理

Web 浏览器应该是大家再也熟悉不过的软件了。几乎任何具有 GUI 的操作系统都可以运行 Web 浏览器。浏览器主要由这些功能模块组成：

- 网络引擎
- 渲染引擎
- JavaScript 引擎
- 数据存储引擎

网络引擎

浏览器是实现 HTTP 规范的应用程序。

渲染引擎

- 构建 DOM 树 (DOM Tree)
- 构建 CSS 规则
- 构建 DOM 渲染树 (DOM Render Tree)
- 布局 (Layout)
- 绘制 (Paint)

```
var ele = document.getElementsByTagName('script')[0];
var type = Object.prototype.toString.call(ele);
console.log('type => ', type);
```

参考资料

- 张成文, 现代前端技术解析[M]. 北京: 电子工业出版社, 中国工信出版集团, 2017.

不可变数据 (Immutable Data)

Facebook 工程师 Lee Byron 花费 3 年时间打造了 Immutable.js，这个库深受 Clojure、Scala、Haskell 等函数式编程语言的影响。它提供了使用纯 JavaScript 实现不可变数据的解决方案。

应当视不可变集合为值而非对象。“对象”描述的事物是会随着时间变化的，“值”则明确地代表了该事物在某时刻的确切状态。

——Immutable.js 官网^[1]

Immutable.js 内部使用了 trie 数据结构来存储数据，只要两个对象的 hashCode 相等

参考资料

1. Immutable.js 官网. <https://facebook.github.io/immutable-js/>
2. Immutable 详解及 React 中实践. <https://segmentfault.com/a/1190000003910357>

参考资料

1. Peter Seibel. 《编程人生》（上卷）, 北京: 人民邮电出版社, 2015.
2. Sandeep Kumar Patel [印] 著. 范洪春 译. 《Web Component 实战》, 北京: 人民邮电出版社, 2015.