聊聊 Node.js 的历史

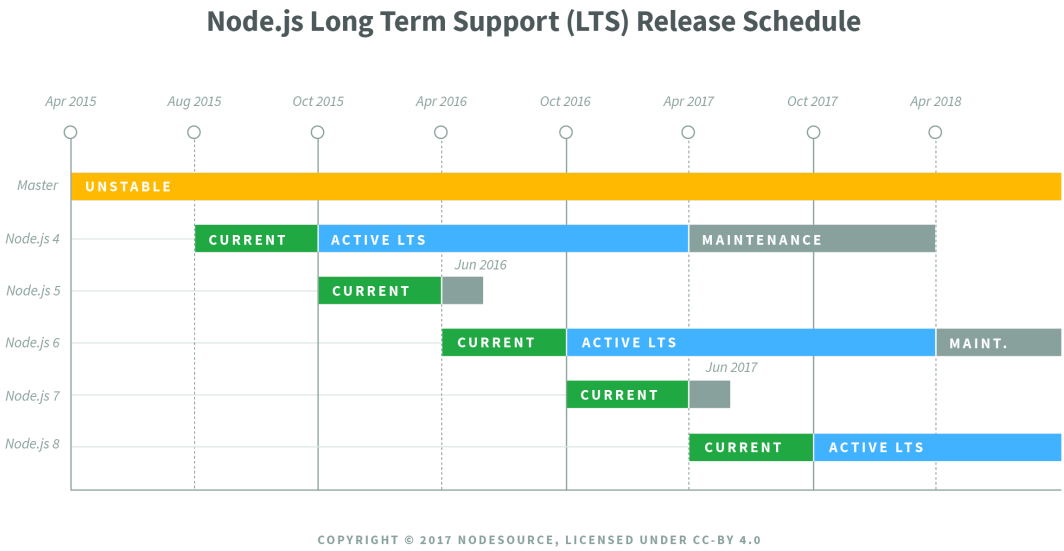
> 狼叔将在6月底发布新书《更了不起的 Node.js：将下一代 Web 框架 Koa 进行到底》，于是借本场 Chat，我将顺便梳理了一下 Node.js 的8年历史、大事件（掌门人）、各种梗（io.js与Node基金会、版本帝等），以及演变引发的思考（各种黑、框架、流程控制、语言特性、段子等）。从此让你做一个不糊涂的Node开发者。 你关注的和没关注到的，相信狼叔比你更清楚。

版本帝？

Chrome浏览器已经蹦到57版本了，是名副其实的版本帝，作为兄弟的Node.js也一样，1.0之前等了6年，而从1.0到8.0，只用了2年时间，这世界到底怎么了？

我们就数一下：

- 从v0.1到0.12用了6年
- 2015-01-14发布了v1.0.0版本（io.js）
- 2.x（io.js）
- 3.x（io.js）
- 2015年09月Node.js基金会已发布Node.js V4.0版 与io.js合并后的第一个版本
- 2015年10月Node.jsv4.2.0将是首个lts长期支持版本
- 2016年底发布到4.2.4 && 5.4.0
- 2016年3月20日v4.4.0 LTS（长期支持版本）和v5.9.0 Stable（稳定版本）
- 2016 年底 v6.0 支持95%以上的es6特性 ， v7.0通过flag支持async函数，99%的es6特性
- 2017年2月发布v7.6版本，可以不通过flag使用async函数



整体来说趋于稳定：

- 成立了Node.js基金会，能够让Node.js在未来有更好的开源社区支持。
- 发布了LTS版本，意味着api稳定。
- 快速发版本，很多人吐槽这个，其实换个角度看，这也是社区活跃的一个体现，但如果大家真的看CHANGELOG，其实都是小改进，而且是边边角角的改进，也就是说Node.js的core（核心）已经非常稳定了，可以大规模使用。

已无性能优势？

Node.js在2009年横空出世，可以说是纯异步获得高性能的功劳。所有语言几乎没有能够和它相比的，比如Java、PHP、Ruby都被啪啪的打脸。但是山一程，水一程，福祸相依，因为性能太出众，导致很多语言、编程模型上有更多探索，比如go语言产生、php里的swolo和vm改进等，大家似乎都以不支持异步为耻辱。后来的故事大家都知道了，性能都提到非常高，c10问题已经没人再考虑，只是大家实现早晚而产生的性能差距而已。

> 编程语言的性能趋于一样的极限，所以剩下的选择，只有喜好。

那么在这种情况下，Node.js还有优势么？

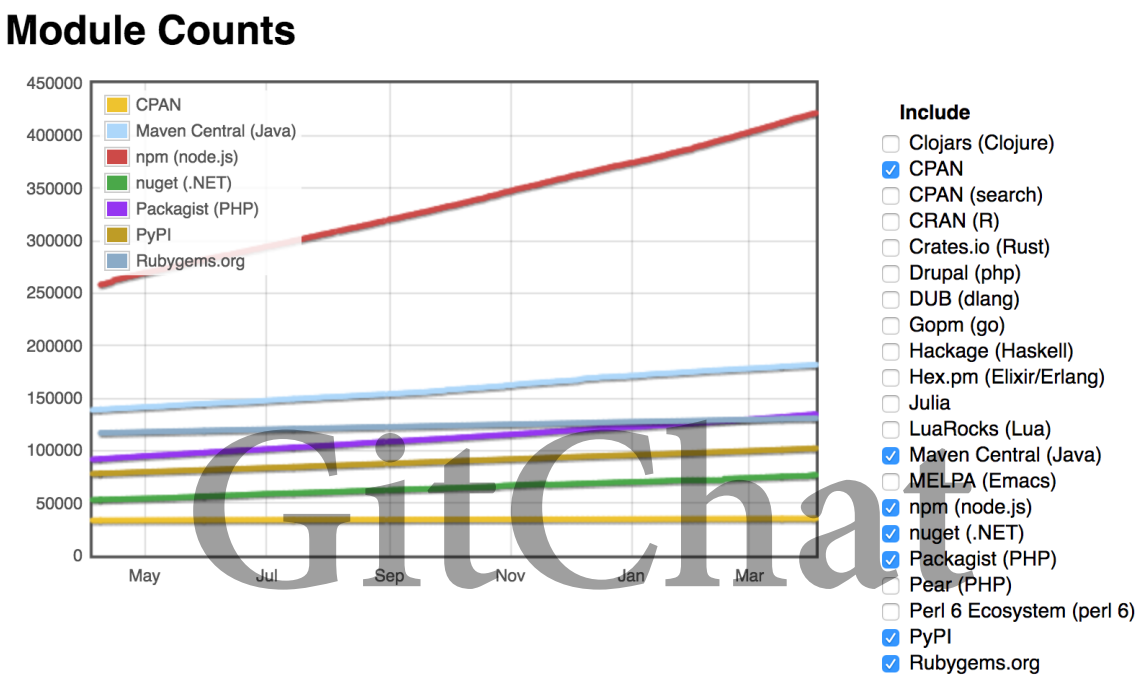
- 实现成本：Node.js除了异步流程控制稍复杂外，其他的都非常简单，比如写法，你可以面向过程、面向对象、函数式，根据自己的解决选择就好了。不要因为它现在变化快，就觉得自己跟不上潮流。尤其是后端程序员转Node.js几乎是2周以内的成本，某些语言光熟悉语法习惯也不止2周吧？
- 调优成本：Node.js即使不优化，它的性能也非常好，如果优化，也比其他语言更简单。

- 学习成本：是否必须用，如果是必须要用，那就少学一样是一样，人生有限，不能都花在写hello world上。我想问，大前端离得开js么？

> 误读：Node.js已无性能优势，它现在最强大的是基于npm的生态。

上面是成本上的比较，其实大家把关注点都转移到基于npm的生态上，截止2017年2月，在npm上有超过45万个模块，秒杀无数。npm是所有的开源的包管理里最强大的，我们说更了不起的Node.js，其实npm居功甚伟，后面会有独立的章节进行阐述。

来自www.modulecounts.com的各个包管理模块梳理的比较：



npm生态是Node的优势不假，可是说“Node.js没有性能优势”真的对么？这其实就是误读，Node.js的性能依然很好呀，而且它有npm极其强大的生态，可谓性能与生态双剑合璧，你说你死不死？

异步和回调地狱？

> 天生异步，败也异步，成也异步。

正因为异步导致了api设计方式只能采用error-first风格的回调，于是大家硬生生的把callback写成了callback hell。于是各种黑粉就冒出来，无非是一些浅尝辄止之辈。但也正因为回调地狱是最差实践，所以大家才不得不求变，于是thunk、promise等纷沓而至。虽然Promise/A+不完美，但对于解决回调地狱是足够的了。而且随着ES6等规范实现，引入generator、co等，让异步越来越近于同步。当async函数落地的时候，Node已经站在了同C#、Python一样的高度上，大家还有什么理由黑呢？

本小节先科普一下异步流程里的各种概念，后面会有独立章节进行详细讲解。

名称	说明
callback	Node.js API天生就是这样的
thunk	参数的求值策略
promise	最开始是Promise/A+规范，随后成为ES6标准
generator	ES6种的生成器，用于计算，但tj想用做流程控制
co	generator用起来非常麻烦，故而tj写了co这个generator生成器，用法更简单
async 函数	原本计划进入es7规范，结果差一点，但好在v8实现了，所以node 7就可以使用，无须等es7规范落地

> 有时，将一件事儿做到极致，也许能有另一种天地。

应用场景

MEAN是一个Javascript平台的现代Web开发框架总称，它是MongoDB + Express +AngularJS + NodeJS四个框架的第一个字母组合。它与传统LAMP一样是一种全套开发工具的简称。在2014和2015年喜欢讲这个，并且还有MEAN.js等框架，但今天已经过时，Node.js有了更多的应用场景。

《Node.js in action》一书里说，Node所针对的应用程序有一个专门的简称：DIRT。它表示数据密集型实时（data-intensive real-time）程序。因为Node自身在I/O上非常轻量，它善于将数据从一个管道混排或代理到另一个管道上，这能在处理大量请求时持有很多开放的连接，并且只占用一小部分内存。它的设计目标是保证响应能力，跟浏览器一样。

这话不假，但在今天来看，DIRT还是范围小了。其实DIRT本质上说的I/O处理的都算，但随着大前端的发展，Node.js已经不再只是I/O处理相关，而是更加的“Node”！

这里给出Node.js的若干使用场景：

- 网站（如express/koa等）
- im即时聊天(socket.io)
- api（移动端，pc，h5）
- HTTP Proxy（淘宝、Qunar、腾讯、百度都有）
- 前端构建工具(grunt/gulp/bower/webpack/fis3...)
- 写操作系统（NodeOS）
- 跨平台打包工具（PC端的electron、 nw.js，比如钉钉PC客户端、微信小程序IDE、微信客户端，移动的cordova，即老的Phonegap，还有更加有名的一站式开发框架ionicframework）
- 命令行工具（比如cordova、shell.js）
- 反向代理（比如anyproxy，node-http-proxy）

- 编辑器Atom、VSCode等

可以说目前大家能够看到的、用到的软件都有Node.js身影，当下最流行的软件写法也大都是基于Node.js的，比如PC客户端luin/medis采用electron打包，写法采用React+Redux。我自己一直的实践的【Node全栈】，也正是基于这种趋势而形成的。在未来，Node.js的应用场景会更加的广泛。更多参见sindresorhus/awesome-nodejs。

Web框架

演进时间线大致如下：

- 2010年tj写的Express。
- 2011年Derby.js开始开发，8月5日，WalmartLabs的一位成员Eran Hammer提交了Hapi的第一次commit。Hapi原本是Postmile的一部分，并且最开始是基于Express构建的。后来它发展成自己自己的框架。
- 2012年1月21日，专注于rest api的restify发布1.0版本，同构的meteor开始投入开发，最像rails的sails也开始了开发。
- 2013年tj开始玩generator，编写co这个generator执行器，并开始了Koa。2013年下半年李成银开始ThinkJS，参考ThinkPHP。
- 2014年，4月9日，express发布4.0，进入4.x时代持续到今天，MEAN.js开始随着MEAN架构的提出开始开发，意图大一统，另外total.js开始，最像PHP's Laravel 或 Python's Django 或 ASP.NET MVC的框架。
- 2015年8月22日，下一代Web框架Koa发布1.0，可以在node 0.12下面，通过co + generator实现同步逻辑，那时候co还是基于thunkfy的，2015.10.30 ThinkJS发布了Es2015+ 特性开发的v 2.0版本。
- 2016年09月，蚂蚁金服的eggjs，在JSConf China 2016上亮相并宣布开源。
- 2017年2月，下一代Web框架Koa发布2.0。

我们可以根据框架的特性进行分类

框架名称	特性	点评
Express	简单、实用，路由中间件等五脏俱全	最著名的Web框架
Derby.js && Meteor	同构	前后端都放到一起，模糊了开发便捷，看上去更简单，实际上上对开发来说要求更高
Sails、Total	面向其他语言，Ruby、PHP等	借鉴业界优秀实现，也是Node.js成熟的一个标志
MEAN.js	面向架构	类似于脚手架，又期望同构，结果只是蹭了热点
Hapi 和 Restfy	面向Api && 微服务	移动互联网时代Api的作用被放大，故而独立分类。尤其是对于微服务开发更是利器
ThinkJS	面向新特性	借鉴ThinkPHP，并慢慢走出自己的一条路，对于Async函数等新特性支持，无出其右
Koa	专注于异步流程改进	下一代Web框架

对于框架选型：

- 业务场景、特点，不必为了什么而什么，避免本末倒置
- 自身团队能力、喜好，有时候技术选型决定团队氛围的，需要平衡激进与稳定
- 出现问题的时候，有人能Cover的住，Node.js虽然8年历史，但模块完善程度良莠不齐，如果不慎踩到一个坑里，需要团队在无外力的情况能够搞定，否则会影响进度

> 个人学习求新，企业架构求稳，无非喜好与场景而已。

我猜大家能够想到的场景，大约如下：

- 前端工具，比如gulp、grunt、webpack等
- 服务器，做类似于Java、PHP的事儿

如果只是做这些，和Java、PHP等就没啥区别了。如果再冠上更了不起的Node.js，就有点名不符实了。所以这里我稍加整理，看看和大家想的是否一样。

技术栈演进

自从ES 2015（俗称ES 6）在Node.js 落地之后，整个Node.js开发都发生了翻天覆地的变化。自从0.10开始，Node.js就逐渐的加入了ES 6特性，比如0.12就可以使用generator，才导致寻求异步流程控制的tj写出了co这个著名的模块，继而诞生了Koa框架。但是在4.0之前，一直都是要通过flag才能开启generator支持，故而Koa 1.0迟迟未发布，在Node 4.0发布才发布的Koa 1.0。

> 2015年，成熟的传统，而2016年，变革开始。

核心变更：es语法支持

- 使用Node.js 4.x或5.x里的es6特性，如果想玩更高级的，可以使用babel编译支持es7特性，或者typescript。
- 合理使用standard 或者 xo 代码风格约定。
- 适当的引入ES 6语法，只要Node.js SDK支持的，都可以使用。
- 需要大家重视OO（面向对象）写法的学习和使用，虽然ES 6的OO机制不健全，但这是大方向，以后会一直增强。OO对于大型软件开发更好。这其实也是我看好typescript的原因。

对比一下变革前后的技术栈选型，希望读者能够从中感受到其中的变化。

分类	2015年	2016年	选型原因
Web框架	express 4.x	koa 1.0 && 2.0（koa2.0刚发布不久，喜欢折腾的可以考虑）	主要在流程控制上的便利，异步毕竟要时刻注意，心累
数据库	mongoose（mongodb）	mongoose（mongodb）	对mongodb和mysql支持都一样，不过是mongodb更简单，足以应付绝大部分场景
异步流程控制	bluebird（Promise/A+实现）	bluebird（Promise/A+ 实现）1）Koa 1.0 使用co + generator 2）Koa 2.0 使用async函数	流程控制演进路线，从promise到async函数，无论如何，promise都是基石，必要掌握的
模板引擎（视图层）	ejs && jade	jade && nunjucks	给出了2种，一种可读性好，另一种简洁高效，都是非常好的
测试	mocha	ava	mocha是Node.js里著名的测试框架，但对新特性的支持没有ava那么好，而ava基于babel安装也要大上好多
调试	node-inspector	VSCode	在Node 6和7出来之后，node-inspector支持的不是那么好，相反VSCode可视化，简单，文件多时也不卡，特别好用

预处理器

前端预处理可分3种：

- 模板引擎
- css预处理器
- js友好语言

这些都离不开Node.js的支持，对于前端工程师来说，使用Node.js来实现这些是最方便不过的。

名称	实现	描述
模板引擎	art\mustache\ejs\hbs\jade ….	上百种之多，自定义默认，编译成html，继而完成更多操作
css预处理器	less\sass\scss\rework\postcss	自定义语法规则，编译成css
js友好语言	coffeescript、typescript	自定义语法规则、编译成js

跨平台

跨平台指的是PC端、移动端、Web/H5

平台	实现	点评
Web/H5	纯前端	不必解释
PC客户端	nw.js和electron	尤其是atom和vscode编辑器最为著名，像钉钉PC端，微信客户端、微信小程序IDE等都是这样的，通过web技术来打包成PC客户端
移动端	cordova（旧称PhoneGap），基于cordova的ionicframework	这种采用h5开发，打包成ipa或apk的应用，称为Hybrid开发（混搭），通过webview实现所谓的跨平台，应用的还是非常广泛的

构建工具

说起构建工具，大概会想到make、 ant、 rake、 gradle等，其实Node.js里有更多实现

名称	介绍	点评
jake	基于coffeescript的大概都熟悉这个，和make、 rake类似	经典传统
grunt	dsl风格的早期著名框架	配置非常麻烦
gulp	流式构建，不会产生中间文件，利用Stream机制，处理大文件和内存有优势，配置简单，只有懂点js就能搞定	grunt 的替代品
webpack + npm scripts	说是构建工具有点过，但二者组合勉强算吧，loader和plugin机制还是非常强大的	流行而已

构建工具都不会特别复杂，所以Node.js世界里有非常多的实现，还有人写过node版本的make呢，玩的很嗨

HTTP Proxy

- 请求代理
- SSR && PWA
- Api Proxy

1）请求代理

对于http请求复杂定制的时候，你是需要让Node.js来帮你的，比如为了兼容一个历史遗留需求，在访问某个CSS的时候必须提供HEADER才可以，如果放到静态server或cdn上是做不到的。

2）SSR && PWA

SSR是服务器端渲染，PWA是渐进式Web应用，都是今年最火的技术。如果大家用过，一定对Node.js不陌生。比如React、 Vuejs都是Node.js实现的ssr。至于pwa的service-worker也是Node.js实现的。那么为啥不用其他语言实现呢？不是其他语言不能实现，而是使用Node.js简单、方便、学习成本低，轻松获得高性能，如果用其他语言，我至少还得装环境

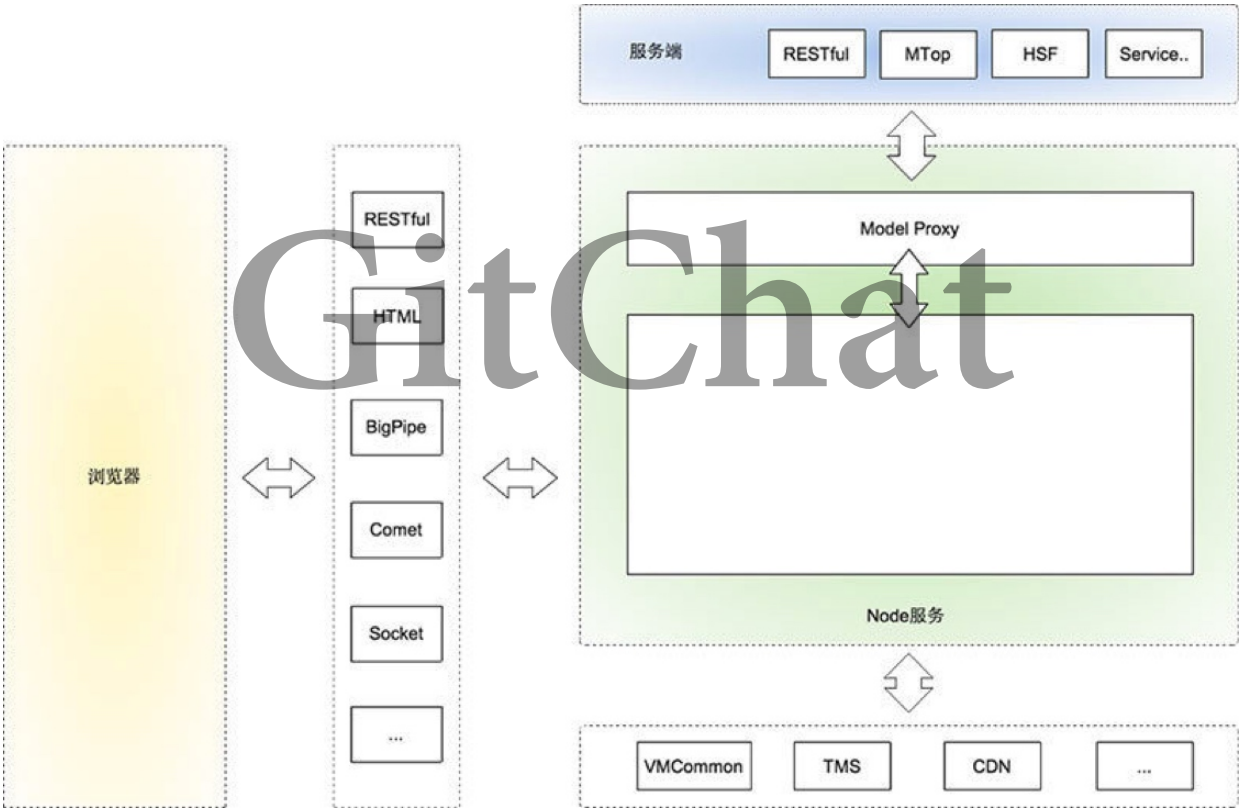
3）Api Proxy

产品需要应变，后端不好变，一变就要设计到数据库、存储等，可能引发事故。而在前端相对更容易，前端只负责组装服务，而非真正对数据库进行变动，所以只要服务api粒度合适，在前端来处理是更好的。

Api的问题：

- 一个页面的Api非常多
- 跨域，Api转发
- Api返回的数据对前端不友好，后端讨厌（应付）前端，几种api都懒得根据ui/ue去定制，能偷懒就偷懒
- 需求决定Api，Api不一定给的及时

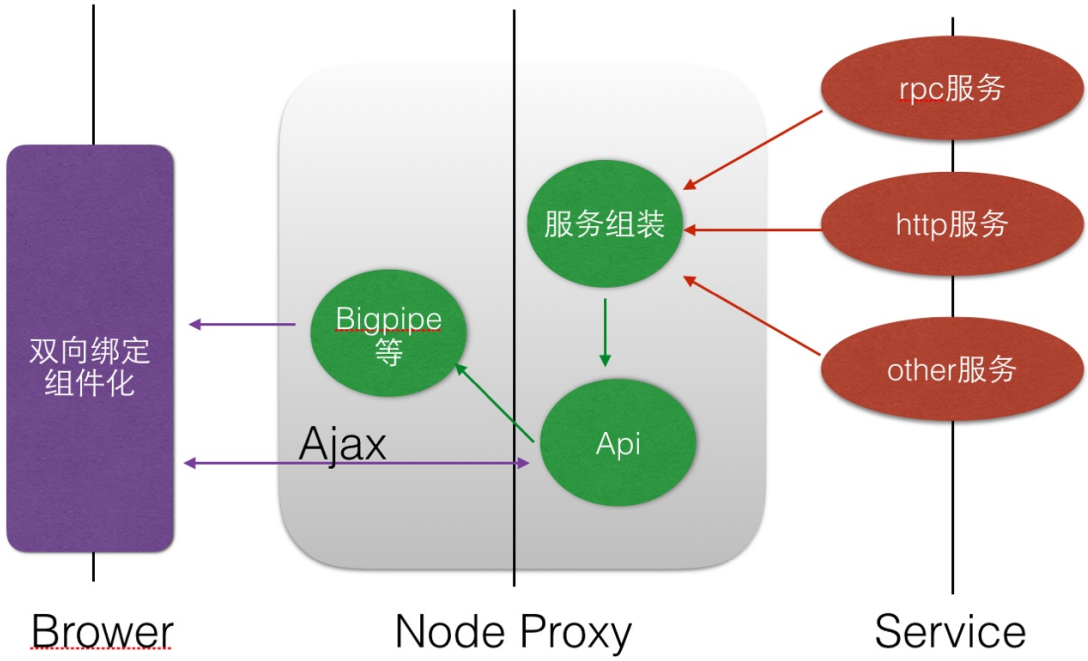
所以，在前端渲染之余，加一层Api Proxy是非常必要的。淘宝早起曾公开过一张架构图，在今天看来，依然不过时。



- 左侧半边，浏览器和Node.js Server通信可以有多种协议，HTML、RESTfull、BigPipe、Comet、Socket等，已经足够我们完成任何想做的事儿了。
- 右侧半边，是Node.js实现的WebServer，Node服务分了2个部分：
  - 常规的Http服务，即大块部分二。
  - ModelProxy指的是根据Server端的服务，组成并转化成自身的Model层。磨蹭用于为Http服务提供更好的接口。

这里的Model Proxy其实就是我们所说的Api Proxy，这张图里只是说了结果，把聚合的服务转成模型，继而为HTTP服务提供Api。

下面我们再深化一下Api Proxy的概念：



这里的Node Proxy做了2件事儿，Api和渲染辅助。

- 前端的异步ajax请求，可以直接访问Api。
- 如果是直接渲染或者bigpipe等协议的，需要在服务器端组装api，然后再返回给浏览器。

所以Api后面还有一个服务组装，在微服务架构流行的今天，这种服务组装放到Node Proxy里的好处尤其明显。既可以提高前端开发效率，又可以让后端更加专注于服务开发。甚至如果前端团队足够大，可以在前端建一个API小组，专门做服务集成的事儿。

API服务

说完了Proxy，我们再看看利益问题。Node.js向后端延伸，必然会触动后端开发的利益。那么Proxy层的事儿，前后端矛盾的交界处，后端不想变，前端又求变，那么长此以往，Api接口会变得越来越恶心。后端是愿意把Api的事儿叫前端的，对后端来说，只要你不动我的数据库和服务就可以。

但是Node.js能不能做这部分呢？答案是能的 ，这个是和Java、PHP类似的，一般是和数据库连接到一起，处理带有业务逻辑的。目前国内大部分都是以Java、PHP等为主，所以要想吃到这部分并不容易。

- 小公司，创业公司，新孵化的项目更倾向于Node.js，简单，快速，高效。
- 微服务架构下的某些服务，使用Node.js开发，是比较合理的。

国内这部分一直没有做的很好，所以Node.js在大公司还没有很好的被应用，安全问题、生态问题、历史遗留问题等，还有很多人对Node.js的误解。

- 单线程很脆弱，这是事实，但单线程不等于不能多核并发，而且你还有集群呢。
- 运维，其实很简单，比其他语言之简单，日志采集、监控也非常简单。

- 模块稳定性，对于Mongodb、 MySQL、 Redis等还是相当不错，但其他的数据库支持可能没那么好。
- 安全问题。

这些对于提供Api服务来说已经足够了。

其他

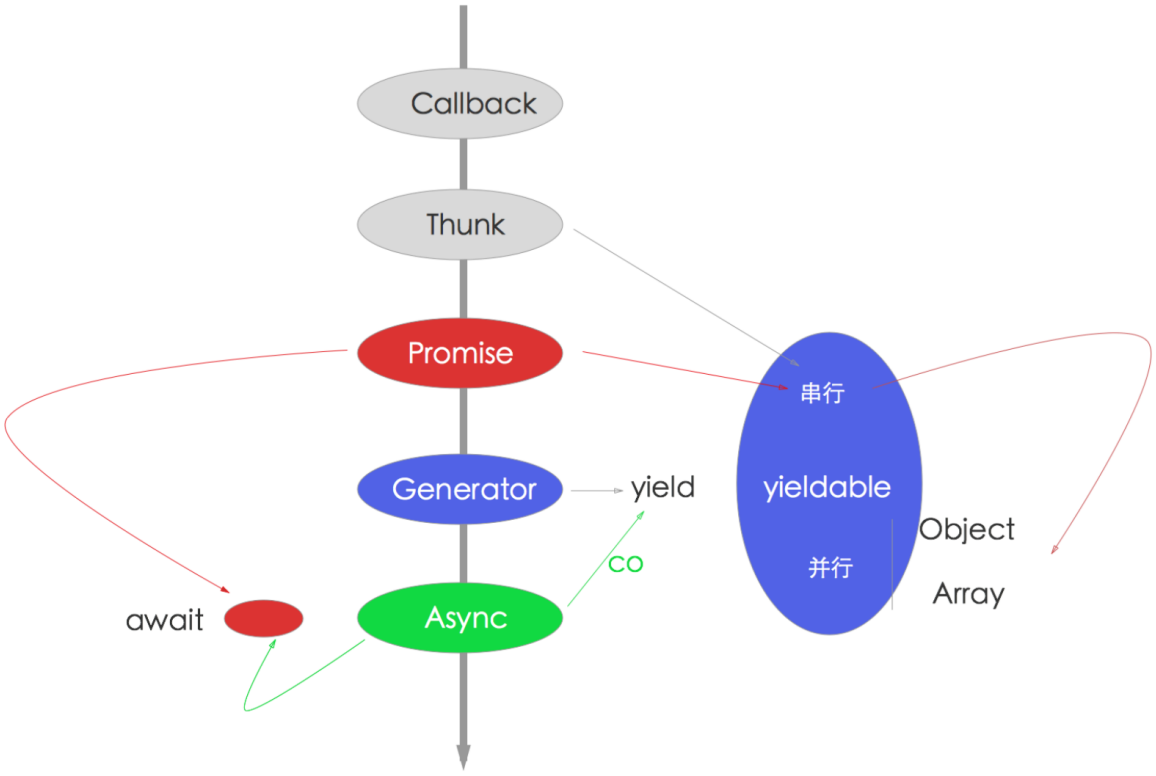
用途	说明	前景
爬虫	抢了不少Python的份额，整体来说简单，实用	看涨
命令行工具	写工具、提高效率，node+npm真是无出其右	看涨
微服务与RPC	Node做纯后端不好做，但在新项目和微服务架构下，必有一席之地	看涨
微信公众号开发	已经火了2年多了，尤其是付费阅读领域，还会继续火下去，gitchat就是实用Node.js做的，而且还在招人	看涨
反向代理	Node.js可以作为nginx这样的反向代理，虽然线上我们很少这样做，但它确确实实可以这样做。比如node-http-proxy和anyproxy等，其实使用Node.js做这种请求转发是非常简单的	看涨

更好的写法

Async函数与Promise

- Async函数是趋势，Chrome 52. v8 5.1已经支持Async函数。(https://github.com/nodejs/CTC/issues/7)了，Node.js 7.0+支持还会远么？
- Async和Generator函数里都支持promise，所以promise是必须会的。
- Generator和yield异常强大，不过不会成为主流，所以学会基本用法和promise就好了，没必要所有的都必须会。
- co作为Generator执行器是不错的，它更好的是当做Promise 包装器，通过Generator支持yieldable，最后返回Promise，是不是有点无耻？

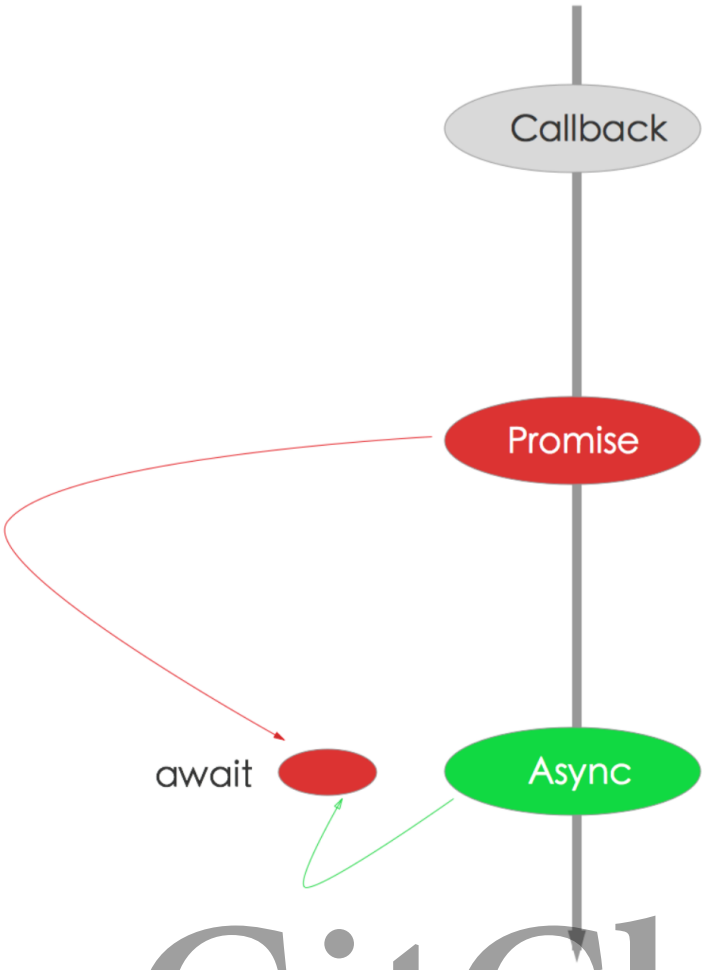
我整理了一张图，更直观一些。



- 红色代表Promise，是使用最多的，无论async还是generator都可用
- 蓝色是Generator，过度货
- 绿色是Async函数，趋势

结论：Promise是必须会的，那你为什么不顺势而为呢？

推荐：使用Async函数 + Promise组合，如下图所示。



实践

合理的结合Promise和Async函数是可以非常高效的，但也要因场景而异。

- Promise更容易做promisefyAll（比如使用bluebird）。
- Async函数无法批量操作。

那么，在常见的Web应用里，我们总结的实践是，dao层使用Promise比较好，而service层，使用Async/Await更好。

dao层使用Promise：

- crud
- 单一模型的方法多
- 库自身支持Promise

这种用promisefyAll基本几行代码就够了，一般单一模型的操作，不会特别复杂，应变的需求基本不大。

而service层一般是多个Model组合操作，多模型操作就可以拆分成多个小的操作，然后使用Await来组合，看起来会更加清晰，另外对需求应变也是非常容易的。

ES.next

> Node.js + ES.next = ♥

Flow && TypeScript

> Type Systems Will Make You a Better JavaScript Developer

ES6模块

现在ES6自带了模块标准，也是JS第一次支持module（之前的CommonJS、AMD、CMD都不算），但目前的所有Node.js版本都没有支持，目前只能用用Traceur、BabelJS，或者TypeScript把ES6代码转化为兼容ES5版本的js代码，ES6模块新特性非常吸引人，下面简要说明。

ES6 模块的目标是创建一个同时兼容CommonJS和AMD的格式，语法更加紧凑，通过编译时加载，使得编译时就能确定模块的依赖关系，效率要比 CommonJS 模块的加载方式高。而对于异步加载和配置模块加载方面，则借鉴AMD规范，其效率、灵活程度都远远好于CommonJS写法。

- 语法更紧凑
- 结构更适于静态编译（比如静态类型检查，优化等）
- 对于循环引用支持更好

ES6 模块标准只有2部分，它的用法更简单，你根本不需要关注实现细节:

- 声明式语法：模块导入import、导出export，没有require了。
- 程式化加载API：可以配置模块是如何加载，以及按需加载。

多模块管理器：Lerna

> A tool for managing JavaScript projects with multiple packages.

https://lernajs.io/

在设计框架的时候，经常做的事儿是进行模块拆分，继而提供插件或集成机制，这样是非常好的做法。但问题也随之而来，当你的模块模块非常多时，你该如何管理你的模块呢？

- 法1：每个模块都建立独立的仓库
- 法2：所有模块都放到1个仓库里

法1虽然看起来干净，但模块多时，依赖安装，不同版本兼容等，会导致模块间依赖混乱，出现非常多的重复依赖，极其容易造成版本问题。这时法2就显得更加有效，对于测试，代码管理，发布等，都可以做到更好的支持。

Lerna就是基于这种初衷而产生的专门用于管理Node.js多模块的工具，当然，前提是你有很多模块需要管理。

你可以通过npm全局模块来安装Lerna，官方推荐直接使用Lerna 2.x版本。

更好的NPM替代品：Yarn

Yarn是开源JavaScript包管理器，由于npm在扩展内部使用时遇到了大小、性能和安全等问题，Facebook携手来自Exponent、Google和Tilde的工程师，在大型JavaScript框架上打造和测试了Yarn，以便其尽可能适用于多人开发。Yarn承诺比各大流行npm包的安装更可靠，且速度更快。根据你所选的工作包的不同，Yarn可以将安装时间从数分钟减少至几秒钟。Yarn还兼容npm注册表，但包安装方法有所区别。其使用了lockfiles和一个决定性安装算法，能够为参与一个项目的所有用户维持相同的节点模块（node_modules）目录结构，有助于减少难以追踪的bug和在多台机器上复制。

Yarn还致力于让安装更快速可靠，支持缓存下载的每一个包和并行操作，允许在没有互联网连接的情况下安装（如果此前有安装过的话）。此外，Yarn承诺同时兼容npm和Bower工作流，让你限制安装模块的授权许可。

2016年10月份, Yarn在横空出世不到一周的时间里，github上的star数已经过万，可以看出大厂及社区的活跃度，以及解决问题的诚意，大概无出其右了！

替换的原因：

- 在Facebook的大规模 npm 都工作的不太好。
- npm拖慢了公司的ci工作流。
- 对一个检查所有的模块也是相当低效的。
- npm被设计为是不确定性的，而Facebook工程师需要为他们的DevOps工作流提供一直和可依赖的系统。

与hack npm限制的做法相反，Facebook编写了Yarn。

- Yarn 的本地缓存文件做的更好
- Yarn 可以并行它的一些操作，这加速了对新模块的安装处理
- Yarn 使用lockfiles，并用确定的算法来创建一个所有跨机器上都一样的文件
- 出于安全考虑，在安装进程里，Yarn 不允许编写包的开发者去执行其他代码

> Yarn, which promises to even give developers that don’t work at Facebook’s scale a major performance boost, still uses the npm registry and is essentially a drop-in replacement for the npm client.

很多人说和ruby的gem机制类似，都生成lockfile。确实是一个很不错的改进，在速度上有很大改进，配置cnpm等国内源来用，还是相当爽的。

友好语言

- 过气的Coffeescript，不多说。
- Babel - also an ES6 to ES5 transpiler that's growing in popularity possibly because it also supports React's JSX syntax. As of today it supports the most ES6 features at a somewhat respectable 73%.
- TypeScript - a typed superset of JavaScript that not only compiles ES6 to ES5 (or even ES3) but also supports optional variable typing. TypeScript only supports 53% of ES6 features.

总结

> 坦诚的力量是无穷的。

Node.js是为异步而生的，它自己把复杂的事儿做了（高并发，低延时），交给用户的只是有点难用的Callback写法。也正是坦诚的将异步回调暴露出来，才有更好的流程控制方面的演进。也正是这些演进，让Node.js从DIRT（数据敏感实时应用）扩展到更多的应用场景，今天的Node.js已经不只是能写后端的JavaScript，已经涵盖了所有涉及到开发的各个方面，而Node全栈更是热门种的热门。

直面问题才能有更好的解决方式，Node.js你值得拥有！