

架构师

ARCHITECT



热点 | Hot

Node.js与io.js那些事儿

深入解析和反思携程宕机事件

推荐文章 | Article

序列化和反序列化

高可用可伸缩架构实用经验谈

专题 | Topic

深入浅出Mesos

细说Executor框架线程池任务执行

观点 | Opinion

架构之重构的12条军规



扫我，码上开启新世界



Geekbang, 有温度的技术社区。Geek是一种精神, 也是一种态度; Geekbang是一个圈子, 也是一种习惯。在这儿, 你要么是Geek, 要么走在成为Geek的路上。



International Architect Summit

全球架构师峰会 2015

中国 · 深圳 · 大梅沙京基海湾大酒店

[深圳站]

2015年7月17-18日



李高峰

华为商城首席架构师

专题演讲:
垂直电商技术的艰辛之路



余庆

易到用车首席架构师

专题演讲:
移动互联网下基于地理位置变化
场景实时搜索引擎



张跃

批改网CEO

专题演讲:
批改网: 基于语言大数据的英语作文
自动批改服务系统建设和运营



陈明

微信高级工程师、朋友圈负责人

专题演讲:
微信朋友圈技术之道



方林

华大基因深圳研发部副院长

专题演讲:
生命科学中的大数据



徐梦云

饿了么数据技术部总监

专题演讲:
数据仓库治理



邓澍军

猿题库研究部总监

专题演讲:
猿题库: 大数据时代的在线教育



曹彬彬

中信证券首席数据应用专家

专题演讲:
证券行业大数据应用探讨

联系方式:

购票热线: 010-89880682

在线咨询(QQ): 2332883546

会务咨询: arch@cn.infoq.com

在线交流(QQ群): 425975960

更多精彩内容, 敬请登陆: sz2015.archsummit.com

Brought by **InfoQ**

QCon上海站

2015年10月15-17日

上海光大会展中心

国际大酒店

www.qconshanghai.com

Brought by **InfoQ**

QCon

全球软件开发大会 International Software Development Conference



QCon是由InfoQ主办的全球顶级技术盛会，每年在伦敦、北京、东京、纽约、圣保罗、上海、旧金山，里约热内卢召开。自2007年3月份首次举办以来，已经有超万名高级技术人员参加过QCon大会。QCon内容源于实践并面向社区，演讲嘉宾依据热点话题，面向5年以上的技术团队负责人、架构师、工程总监、高级开发人员分享技术创新和最佳实践。



www.qconferences.com

云上的基础设施与运维

探讨企业云计算基础设施和架构思路以及高效安全运维解决方案

2015年7月17日(周五)



深圳

大梅沙京基海湾大酒店



报名请扫描二维码
名额有限, 报名从速

数据分析 与企业架构

聚焦IT企业架构设计与研发体系,
构建最优内部数据分析平台

2015年7月18日(周六)

卷首语

Google 推出的 Go 语言这两年火的是一塌糊涂，而同样是亲爹生的 Dart 语言这些年却一直不温不火。什么是 Dart 语言了？我先来简单解释下：

Dart 是 Google 于 2011 年发布的一门开源编程语言，目标是为开发现代 Web 程序提供结构化但又不乏灵活性的编程语言，其实就是弥补 JavaScript 的不足。Dart 在 JavaScript 语言的基础上，改进了编程效率和执行性能，大幅度减少了编程的复杂性。相比 JavaScript，Dart 语言更加简单和高效，它支持类和接口，是一门纯面向对象的语言。Dart 在动态语言的基础上，结合了静态语言的优点，有很多不错的特性，比如可选类型、并发编程、工厂构造函数、级联调用。Dart 代码可以用两种不同方式执行：一是通过原生的虚拟机（可以集成到浏览器）；另一种则是通过 Google 的 Dart2js 编译器将 Dart 代码转换为 JavaScript 代码，然后再执行。

从发布之初，Dart 语言要做的就是颠覆 JavaScript，确实，JavaScript 这门语言缺陷有很多，不过这也可以说理解，因为 JavaScript 从设计到发布仅有几个月的时间，可以说非常仓促。而 Dart 语言在设计时借鉴了很多现代语言的思路，它在性能、易用性等方面都远远超过了 JavaScript。但从现在的情况来看，Dart 语言似乎并没有发展起来。这从最近的新闻里就能看出来，4 月，谷歌确认他们不会再将 Dart VM 集成到 Chrome 中，也就是说，要使用 Dart 语言替换 JavaScript 几乎不可能，因为现在用户只能使用编译为 JavaScript 的方式使用 Dart。

也许 Google 对 Dart 语言的定位早有了变化，所以才宣布在 Chrome 中放弃 Dart。Android 应用基本都是使用 Java 创建的，这俩还打过不少官司，Google 也是吃了不少哑巴亏。5 月初，Google 发布了跨平台框架：Sky。Sky 基于 Dart 语言编写，因为 Dart 本身就是与平台无关的，所以 Sky 的目标是跨平台。最近比较火的跨平台框架是 react-native，Sky 其实和 React 差不多，或者说是参考了 React 的设计哲学，只不过一个使用 JavaScript，一个使用 Dart。

当然，问题又回来了，有了 React，为什么还要用 Sky？JavaScript 已经获得了各个平台的支持，所以 React 推广起来也不费事，但 Dart 又面临的同样的问题，其它平台会支持 Dart 吗？

不管怎么样，Dart 终是迈出了属于自己的一大步。

——郭蕾

目 录

| | |
|---|----|
| 卷首语 | 4 |
| 目录 | 5 |
| 热点 Hot | |
| Node.js 与 io.js 那些事儿 | 6 |
| 深入解析和反思携程宕机事件 | 15 |
| 推荐文章 Article | |
| 序列化和反序列化 | 18 |
| 高可用可伸缩架构实用经验谈 | 34 |
| 专题 Topic | |
| 深入浅出 Mesos (三): 持久化存储和容错 | 39 |
| 深入浅出 Mesos (四): Mesos 的资源分配 | 45 |
| 戏 (细) 说 Executor 框架线程池任务执行全过程 (上) | 50 |
| 戏 (细) 说 Executor 框架线程池任务执行全过程 (下) | 58 |
| 观点 Opinion | |
| Java 20 年: 转角遇到 Go | 67 |
| 架构之重构的 12 条军规 | 70 |
| 封面植物 | 76 |

Node.js 与 io.js 那些事儿

作者 朴灵

去年 12 月，多位重量级 Node.js 开发者不满 Joyent 对 Node.js 的管理，自立门户创建了 io.js。io.js 的发展速度非常快，先是于 2015 年 1 月份发布了 1.0 版本，并且很快就达到了 2.0 版本，社区非常活跃。而[最近 io.js 社区又宣布](#)，这两个项目将合并到 Node 基金会下，并暂时由“Node.js 和 io.js 核心技术团队联合监督”运营。本文将聊一聊 Node.js 项目的一些历史情况，与 io.js 项目之间的恩怨纠葛，他们将来的发展去向。希望能从历史的层面去了解这个开源项目在运营模式上是如何演变和发展的。

Node.js 项目的由来

自从 JavaScript 被 Brendan Eich 创造出来后，除了应用在浏览器中作为重要的补充外，人类从来没有放弃过将 JavaScript 应用到服务端的想法。这些努力从 livewired 项目（1994 年 12 月）开始，就从来没有停止过。如果你不知道 livewired，那应该知道 ASP 中可以使用 JScript 语言（1996 年），或者 Rhino。但直到 2009 年，这些服务端 JavaScript 技术与同样应用在服务端的 Java、PHP 相比，显得相对失色。

谈到 Node.js 的由来，不可避免要聊到它的创始人 Ryan Dahl。在 2009 年时，服务端 JavaScript 迎来了它的拐点，因为 Ryan Dahl 带来了 Node.js，在那之后 Node.js 将服务端 JavaScript 带入了新的境地，大量的 JavaScript 在 GitHub 上被贡献出来，大量的 JavaScript 模块出现，出现了真正的繁荣。

Node.js 不是凭空出现的项目，也不是某个 Web 前端工程师为了完成将 JavaScript 应用到服务端的理想而在实验室里捣鼓出来的。它的出现主要归功于 Ryan Dahl 历时多年的研究，以及一个恰到好处的节点。2008 年 V8 随着 Chrome 浏览器的出世，JavaScript 脚本语言的执行效率得到质的提升，这给 Ryan Dahl 带来新的启示，他原本的研究工作与 V8 之间碰撞出火花，于是带来了一个基于事件的高性能 Web 服务器。

Ryan Dahl 的经历比较奇特，他并非科班出身的开发者，在 2004 年的时候他还在纽约的罗彻斯特大学数学系读博士，期间有研究一些分形、分类以及 p-adic 分析，这些都跟开源和编程没啥关系。2006 年，也许是厌倦了读博的无聊，他产生了『世界那么大，我想去看看』的念头，做出了退学的决定，然后一个人来到智利的 Valparaiso 小镇。那时候他尚不知道找一个什么样的工作来糊口，期间他曾熬夜做了一些不切实际的研究，如如何通过云进行通信。下面是这个阶段他产出的中间产物，与后来苹果发布的 iCloud 似乎有那么点相似。



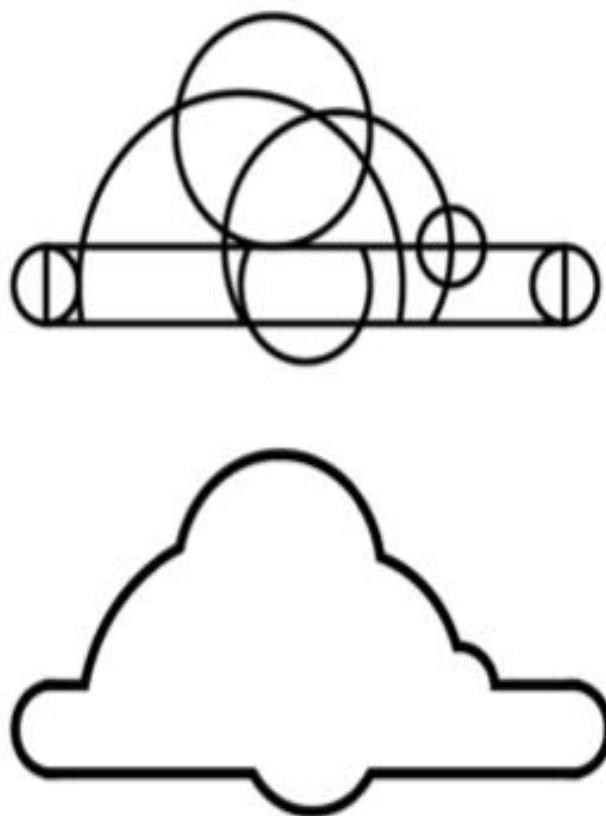
图为 Node.js 创始人

Ryan Dahl

从那起，Ryan Dahl 不知道是否因为生活的关系，他开始学习网站开发了，走上了码农的道路。那时候 Ruby on Rails 很火，他也不例外的学习了它。从那时候开始，Ryan Dahl 的生活方式就是接项目，然后去客户的地方工作，在他眼中，拿工资和上班其实就是去那里旅行。此后他去过很多地方，如阿根廷的布宜诺斯艾利斯、德国的科隆、奥地利的维也纳。

Ryan Dahl 经过两年的工作后，成为了高性能 Web 服务器的专家，从接开发应用到变成专门帮客户解决性能问题的专家。期间他开始写一些开源项目帮助客户解决 Web 服务器的高并发性能问题，尝试过的语言有 Ruby、C、Lua。当然这些尝试都最终失败了，只有其中通过 C 写的 HTTP 服务库 libebbb 项目略有起色，基本上算作 libuv 的前身。这些失败各有各的原因，Ruby 因为虚拟机性能太烂而无法解决根本问题，C 代码的性能高，但是让业务通过 C 进行开发显然是不太现实的事情，Lua 则是已有的同步 I/O 导致无法发挥性能优势。虽然经历了失败，但 Ryan Dahl 大致的感觉到了解决问题的关键是要通过事件驱动和异步 I/O 来达成目的。

在他快绝望的时候，V8 引擎来了。V8 满足他关于高性能 Web 服务器的梦想：



1. 没有历史包袱，没有同步 I/O。不会出现一个同步 I/O 导致事件循环性能急剧降低的情况。
2. V8 性能足够好，远远比 Python、Ruby 等其他脚本语言的引擎快。
3. JavaScript 语言的闭包特性非常方便，比 C 中的回调函数好用。

于是在 2009 年的 2 月，按新的想法他提交了项目的第一行代码，这个项目的名字最终被定名为“node”。

2009 年 5 月，Ryan Dahl 正式向外界宣布他做的这个项目。2009 年底，Ryan Dahl 在柏林举行的 JSConf EU 会议上发表关于 Node.js 的演讲，之后 Node.js 逐渐流行于世。

以上就是 Node.js 项目的由来，是一个专注于实现高性能 Web 服务器优化的专家，几经探索，几经挫折后，遇到 V8 而诞生的项目。

Node.js 项目的组织架构和管理模式

Node.js 随着 JSConf EU 会议等形式的宣传下，一家位于硅谷的创业公司注意到了该项目。这家公司就是 Joyent，主要从事云计算和数据分析等。Joyent 意识到 Node.js 项目的价值，决定赞助这个项目。Ryan Dahl 于 2010 年加入该公司，全职负责 Node.js 项目的开发。此时 Node.js 项目进入了它生命历程里的第二个阶段：从个人项目变成一个公司组织下的项目。

这个阶段可以从 2010 年 Ryan Dahl 加入 Joyent 开始到 2014 年底 Mikeal Rogers 发起 Node Forward 结束，Node 的版本也发展到了 v0.11。这个时期，IT 业中的大多数企业都关注过 Node.js 项目，如微软甚至对于 Node.js 对 Windows 的移植方面做过重要的贡献。

这个时期的组织架构和管理模式可以总结为“Gatekeeper + Joyent”模式。

Gatekeeper 的身份类似于项目的技术负责人，对技术方向的把握是有绝对权威。历任的 Gatekeeper 为：Ryan Dahl、Isaac Z. Schlueter、Timothy J Fontaine，均是在 Node.js 社区具有很高威望的贡献者。项目的法律方面则由 Joyent 负责，Joyent 注册了“Node.js”这个商标，使用其相关内容需要得到法律授权（如笔者《深入浅出 Node.js》上使用了 Node.js 的 Logo，当时是通过邮件的形式得到过授权）。技术方面除了 Gatekeeper 外，还有部分 core contributor。core contributor 除了贡献重要 feature 外，帮助项目进行日常的 patch 提交处理，协助 review 代码和合并代码。项目中知名的 core contributor 有 Ben Noordhuis，Bert Belder、Fedor Indutny、Trevor Norris、Nathan Rajlich 等，这些人大多来自 Joyent 公司之外，他们有各自负责的重要模块。Gatekeeper 除了要做 core contributor 的事情外，还要决定版本的发布等日常事情。

Node.js 成为 Joyent 公司的项目后，Joyent 公司对该项目的贡献非常大，也没有过多的干涉 Node.js 社区的发展，还投入了较多资源发展它，如 Ryan Dahl、Isaac Z. Schlueter、Timothy J Fontaine 等都是 Joyent 的全职员工。

Node.js 社区的分裂

“Gatekeeper + Joyent”模式运作到 2013 年的时候都还工作良好，蜜月期大概中止于第二任 Gatekeeper Isaac Z. Schlueter 离开 Joyent 自行创建 npm inc. 公司时期。前两任 Gatekeeper 期间，Node.js 的版本迭代都保持了较高的频率，大约每个月会发布一个小版本。在 Isaac Z. Schlueter 卸任 Gatekeeper 之后，Node.js 的贡献频率开始下降，主要的代码提交主要来自社区的提交，代码的版本下降到三个月才能发布一个小版本。社区一直期待的 1.0 版本迟迟不能发布。这个时期 Node.js 属于非常活跃的时期，但是对于 Node.js 内核而言却进展缓慢。技术方向上似乎是有了一些不明朗，一方面期待内核稳定下来，一方面又不能满足社区对新 feature 的渴望（如 ES6 的特性迟迟无法引入）。

第三任的 Gatekeeper Timothy J Fontaine 本人也意识到这个问题。从他上任开始，主要的工作方向就是解决该问题。他主要工作是 Node on the road 活动，通过一系列活动来向一些大企业用户获取他们使用 Node.js 的反馈。通过一些调研，他做了个决定，取消了贡献者的 CLA 签证，让任何人可以贡献代码。

尽管 Timothy J Fontaine 的做法对 Node.js 本身是好的，但是事情没有得到更好的改善。这时候 Node.js 项目对社区贡献的 patch 处理速度已经非常缓慢，经常活跃的 core contributor 只有 Fedor Indutny、Trevor Norris。另外还发生了[人称代词](#)的事件，导致 Node.js/libuv 项目中非常重要的贡献者 Ben Noordhuis 离开 core contributor 列表，这件事情被上升到道德层面，迎来了不少人的谩骂。其中 Joyent 的前任 CEO 甚至还致信表示如果是他的员工，会进行开除处理。这致使 Node.js 项目的活跃度更低。Node.js 的进展缓慢甚至让社区的知名 geek TJ Holowaychuk 都选择离开 Node.js 而投入 Go 语言的怀抱。

可以总结这个时期是“Gatekeeper + Joyent”模式的末期。Joyent 对于项目的不作为和其他层面对于社区其他成员的干预，导致项目进展十分缓慢，用蜗牛的速度来形容一点也不为过。尽管 Timothy J Fontaine 试图挽回些什么，也有一些行为来试图重新激活这个项目的活力，但是已经为时已晚。

这时一个社区里非常有威望的人出现了，他就是 Mikeal Rogers。Mikeal Rogers 的威望不是建立在他对 Node.js 项目代码的贡献上，他的威望主要来自于 request 模块和 JSConf 会议。其中 JSConf 是 JavaScript 社区最顶级的会议，他是主要发起人。

在 2014 年 8 月，以 Mikeal Rogers 为首，几个重要 core contributor 一起发起了一个叫做“Node forward”的组织。该组织致力于发起一个由社区自己驱动来提升 Node、JavaScript 和整个生态的项目。



“Node forward”可以视作是 io.js 的前身。这些 core contributor 们在“Node forward”上工作了一段时间，后来因为可能涉及到 Node 这个商标问题，Fedor Indutny 憤而 fork 了 Node.js，改名为 io.js，宣告了 Node.js 社区的正式分裂。

简单点来说这件事情主要在于社区贡献者们对于 Joyent 公司的不满，导致这些主要贡献者们想通过一个更开放的模式进行协作。复杂点来说这是公司开源项目管理模式的问题所在，当社区方向和公司方向一致时，必然对大家都有好处，形同蜜月期，但当两者步骤不一致时，分歧则会暴露出来。这点在 Node.js 项目的后期表现得极为明显，社区觉得项目进展缓慢，而 Joyent 公司的管理层则认为他稳定可靠。

io.js 与 Node.js advisory board

在“Node Forward”的进展期间，社区成员们一起沟通出了一个基本的开放的管理模式。这个模式在 io.js 期间得到体现。

io.js 的开放管理模式主要体现在以下方面：

- 不再有 Gatekeeper。取而代之的是 TC (Technical Committee)，也就是技术委员会。技术委员会基本上是由那些有很多代码贡献的 core contributor 组成，他们来决定技术的方向、项目管理和流程、贡献的原则、管理附加的合作者等。当有分歧产生时（如引入 feature），采用投票的方式来决定，遵循少数服从多数的简单原则。基本上原来由一个人担任的 Gatekeeper 现在由一个技术委员会来执行。如果要添加一个新成员为 TC 成员，需要由一位现任的 TC 成员提议。每个公司在 TC 中的成员不能超过总成员的 1/3。
- 引入 Collaborators。代码仓库的维护不仅仅局限在几个 core contributor 手中，而是引入 Collaborators，也就是合作者。可以理解为有了更多的 core contributor。
- TC 会议。之前的沟通方式主要是分布式的，大家通过 GitHub 的 issue 列表进行沟通。这种模式容易堆积问题，社区的意见被接受和得到处理取决于 core contributor 的情况。io.js 会每周举行 TC 会议，会议的内容主要就 issue 讨论、解决问题、工作进展等。会议通过

Google Hangout 远程进行，由 TC 赞同的委任主席主持。会议视频会发布在 YouTube 上，会议记录会提交为文档放在代码仓库中。

- 成立工作组。在项目中成立一些细分的工作组，工作组负责细分方向上的工作推进。

io.js 项目从 fork 之后，于 2015-01-14 发布了 v1.0.0 版本。自此 io.js 一发不可收拾，以周为单位发布新的版本，目前已经发布到 2.0.2。io.js 项目与 Node.js 的不同在行为上主要体现在以下方面：

- 新功能的激进。io.js 尽管在架构层面依然保持着 Node.js 的样子（由 Ryan Dahl 时确立），但是对于 ECMAScript 6 持拥抱态度。过去在 Node.js 中需要通过 flag 来启用的新功能，io.js 中不再需要这些 flag。当然不用 flag 的前提是 V8 觉得这个 feature 已经稳定的情况下。一旦最新的 Chrome 采用了新版本的 V8，io.js 保持很快的跟进速度。
- 版本迭代。io.js 保持了较高频率的迭代，以底层 API 改变作为大版本的划分，但对于小的改进，保持每周一个版本的频率。只要是改进，io.js 项目的 TC 和 Collaborators 都非常欢迎，大到具体 feature 或 bug，小到文档改进都可以被接受，并很快放出版本。
- issue 反馈。Node.js 的重要的贡献者们都在 io.js 上工作，Node.js 和 io.js 项目的问题反馈速度几乎一致，但是问题处理速度上面 io.js 以迅捷著称，基本在 2-3 天内必然有响应，而 Node.js 则需要 1 个礼拜才有回复。

基本上而言原本应该属于 Node.js 项目的活力现在都在 io.js 项目这里。如果没有其他事情的发生，io.js 可以算作社区驱动开源项目的成功案例了。

当然，尽管在 Node.js 这边进展缓慢，但 Joyent 方面还是做出了他们的努力。在“Node Forward”讨论期间，Joyent 成立了临时的 Node.js 顾问委员会 <https://nodejs.org/about/advisory-board/>。顾问委员会的主要目标与“Node Forward”的想法比较类似，想借助顾问委员会的形式来产出打造一个更加开放的管理模式，以找到办法来平衡所有成员的需要，为各方提供一个平台来投入资源到 Node.js 项目。

顾问委员会中邀请了很多重要的 Contributor 和一些 Node.js 重度用户的参与。开了几次会议来进行探讨和制定新的管理模式。于是就出现了一边是 io.js 如火如荼发布版本，Joyent 这边则是开会讨论的情况。顾问委员会调研了 IBM(Eclipse)、Linux 基金会、Apache 等，决定成立 Node.js 基金会的形式。

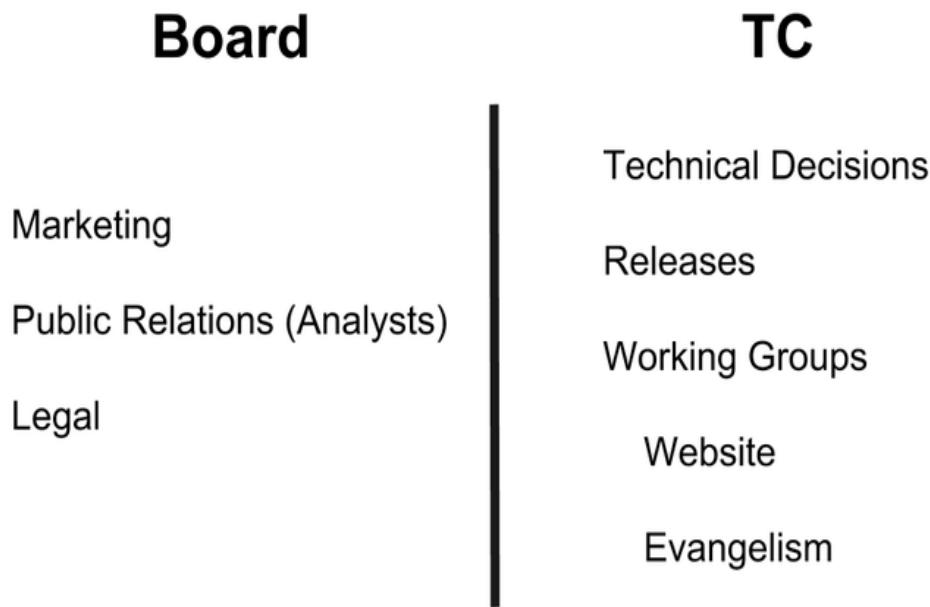
io.js 与 Node.js 基金会

时间来到 2015 年 1 月，临时委员会正式发布通告决定将 Node.js 项目迁移到基金会，并决定跟 io.js 之间进行和解。简单点来说 Node.js 方面除了版本的进展比较缓慢外，确实是在制定一个新的模式来确保 Node.js 项目的下一步发展，Joyent 公司本着开放的原则，也做出相当大的让步，保持着较为和谐的状态。

然而 io.js 动作太快，代码的进展程度远远快于 Node.js 项目，和解的讨论从 2 月开始讨论，到 5 月才做出决定。这时 io.js 已经发布了它的 2.0 版本。

最终的结论是 Node.js 项目和 io.js 项目都将加入 Node.js 基金会。Node.js 基金会的模式与 io.js 较为相似，但是更为健全。Mikeal Rogers 在他的一篇名为“[Growing Up](#)”的文章中提到 io.js 项目需要一个基金会的原因。

io.js 项目在技术方面的成熟度显然要比最初的 Gatekeeper 时代要更为先进，给予贡献者更多的管理权利。然而在市场和法律方面，还略显幼稚。最终无论是顾问委员会，还是 io.js 都选定以基金会的形式存在。这个基金会参考 Linux 基金会的形式，由董事会和技术委员会组成，董事会负责市场和法律方面的事务，技术委员会负责技术方向。

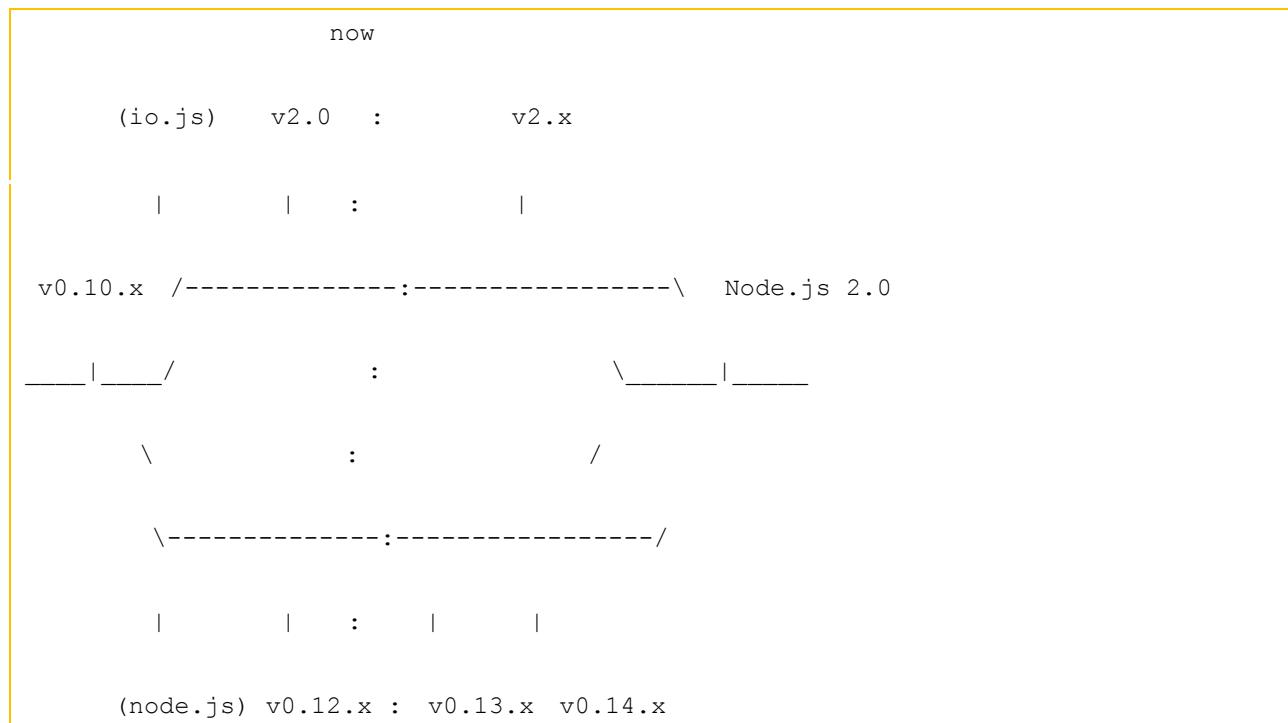


就像《三国演义》所述：天下大势，合久必分，分久必合。Node.js 项目也从 Joyent 公司的怀里走出来，成长为基金会的形式，进入这个项目生命周期里第三个阶段。

后续

从 io.js 的分裂到 Node.js 基金会，从外人看起来似乎如一场闹剧一般，然而这个过程中可以看到一个开源项目自身的成长。尽管 io.js 将归于 Node.js 基金会，像一个离家出走的孩子又回家一般，它的出走可能要被人忘记，但从当初的出发点来说，这场战役，io.js 其实是赢家。穷则思变、不破不立是对 Joyent 较为恰当的形容。如果 Joyent 能提前想到这些，则不会有社区分裂的事情发生。

Node.js 处于停滞状态的开发和 io.js 的活跃情况之间，目前免不了大量的 Merge 工作。作为和解的条件之一，Node.js 基金会之后 Node 版本的发布将基于目前 io.js 的进展来进行。后续的合并工作示意如下：



在未完成合并之前，io.js会继续保持发布。Node.js的下个大版本跨过1.0，直接到2.0。

io.js项目的TC将被邀请加入Node.js基金会的TC，毕竟两者在技术管理方面达成了一致。基金会将在黄金和白银会员中选举出董事、技术委员会成员中选举出技术委员主席。

对于成为Node.js基金会成员方面，企业可以通过赞助的方式注册成为会员。

总结

一个开源项目成长起来之后，就不再是当初创始人个人维护的那个样子了。Node.js项目的发展可以说展现了一个开源项目是如何成长蜕变成成熟项目的。当然我们现在说Node.js基金会是成功的还为时尚早，但是祝福它。

参考文档

- <https://github.com/joyent/node/issues/9295>
- <https://github.com/iojs/io.js/issues/978>
- <https://github.com/iojs/io.js/issues/1336>
- <https://github.com/iojs/io.js/issues/1416>
- <https://github.com/iojs/io.js/labels/meta>
- <http://blog.nodejs.org/2015/05/08/transitions/>

热点| Hot

- <http://blog.nodejs.org/2015/05/08/next-chapter/>
- <https://github.com/iojs/io.js/issues/1664>
- <http://tinyclouds.org/nodeconf2012.pdf>
- <https://www.joyent.com/blog/introducing-the-nodejs-foundation>
- <http://blog.nodejs.org/2015/05/15/node-leaders-are-building-an-open-foundation/>
- <https://medium.com/node-js-javascript/growing-up-27d6cc8b7c53>



The poster features the TalkingData logo (a blue and green pie chart icon) and the text "TalkingData 移动·数据·价值". To its right is the ArchSummit logo (a white square with a blue mountain peak icon). On the right side, there's a vertical column of four blue circular bullet points with corresponding text:

- 分布式
数据收集
- 大数据
计算框架
- 高效的
bitmap存储
- Kafka
统一日志汇集

Below these points is a large yellow arrow pointing upwards and to the right.

Event details:
时间: 2015年7月17~18日
地点: 深圳·大梅沙京基海湾大酒店

TalkingData为您讲述 大数据背后的价值

主讲人: TalkingData 研发副总裁 阎志涛

QR code: www.talkingdata.com

TalkingData (北京腾云天下科技有限公司) 成立于2011年9月，是一家专注于移动互联网综合数据服务的创业公司。TalkingData目前拥有多条数据服务产品线，服务内容从基本的数据统计，到深入的数据分析、挖掘，可以为移动互联网企业提供全方位的大数据解决方案。

深入解析和反思携程宕机事件

作者 智锦

携程网宕机事件还在持续，截止 28 号晚上 8 点，携程首页还是指向一个静态页面，所有动态网页都访问不了。关于事故根源，网上众说纷纭。作为互联网运维老兵，尝试分析原因，谈谈我的看法。

宕机原因分析

网上有各种说法，有说是数据库数据和备份数据被物理删除的。也有说是各个节点的业务代码被删除，现在重新在部署。也有说是误操作，导致业务不可用，还有说是黑客攻击甚至是内部员工恶意破坏的。

先说一下最早传出来的“数据库物理删除”，其实这个提法就很不专业，应该是第一个传播者，试图强调问题之严重和恢复之困难，所以用了一个普通电脑用户比较熟悉的“物理删除”的概念。实际上，任何一个网站的数据库，都分为本地高可用备份、异地热备、磁带冷备三道防线，相应的数据库管理员、操作系统管理员、存储管理员三者的权限是分离的，磁带备份的数据甚至是保存在银行的地下金库中的。从理论上而言，很难有一个人能把所有的备份数据都删除，更不用说这个绘声绘色的物理删除了。

第二个则是黑客攻击和内部员工破坏的说法，这个说法能满足一些围观者猎奇的心理，因此也传播的比较快。但理性分析，可能性也不大。黑客讲究的是潜伏和隐蔽，做这种事等于是自杀性攻击。而内部员工也不太可能，我还是相信携程的运维人员的操守和职业素养，在刑法的威慑下，除非像“法航飞行员撞山”那种极个别案列，正常情况下不太可能出现人为恶意的可能性。

从现象上看，确实是携程的应用程序和数据库都被删除。我分析，最大的可能还是运维人员在正常的批量操作时出现了误操作。我猜测的版本是：携程网被“乌云”曝光了一个安全漏洞，漏洞涉及到了大部分应用服务器和数据库服务器；运维人员在使用 pssh 这样的批量操作执行修复漏洞的脚本时，无意中写错了删除命令的对象，发生了无差别的全局删除，所有的应用服务器和数据库服务器都受到了影响。这个段子在运维圈子中作为笑话流传了很多年，没想到居然真的有这样一天。

为什么恢复的如此缓慢？

从上午 11 点传出故障，到晚上 8 点，携程网站一直没能恢复。所以很多朋友很疑惑：“为什么

网站恢复的如此缓慢？是不是数据库没有备份了？”这也是那个“数据库物理删除”的说法很流行的一个根源。实际上这个还是普通用户，把网站的备份和恢复理解成了类似我们的笔记本的系统备份和恢复的场景，认为只有有备份在，很快就能导入和恢复应用。

实际上大型网站，远不是像把几台应用和数据库服务器那么简单。看似很久都没有变化的一个网站，后台是一个由 SOA（面向服务）架构组成的庞大服务器集群，看似简单的一个页面背后由成百上千个应用子系统组成，每个子系统又包括若干台应用和数据库服务器，大家可以理解为每一个从首页跳转过去的二级域名都是一个独立的应用子系统。这上千的个应用子系统，平时真正经常发布和变更的，可能就是不到 20% 的核心子系统，而且发布时都是做加法，很少完全重新部署一个应用。

在平时的运维过程中，对于常见的故障都会有应急预案。但像携程这次所有系统包括数据库都需要重新部署的极端情况，显然不可能在应急预案的范畴中。在仓促上阵应急的情况下，技术方案的评估和选择问题，不同技术岗位之间的管理协调的问题，不同应用系统之间的耦合和依赖关系，还有很多平时欠下的技术债都集中爆发了，更不用说很多不常用的子系统，可能上线之后就没人动过，一时半会都找不到能处理的人。更要命的是，网站的核心系统，可能会写死依赖了这个平时根本没人关注的应用，想绕开边缘应用只恢复核心业务都做不到。更别说在这样的高压之下，各种噪音和干扰很多，运维工程师的反应也没有平时灵敏。

简单的说，就算所有代码和数据库的备份都存在，想要快速恢复业务，甚至比从 0 开始重新搭建一个携程更困难。携程的工程师今天肯定是一个不眠夜。乐观的估计，要是能在 24 小时之内恢复核心业务，就已经非常厉害了。

天下运维是一家。携程的同行加油，尽快度过难关！

故障根源反思：黑盒运维之殇

携程的这次事件，不管原因是什么，都会成为 IT 运维历史上的一个标志性事件。相信之后所有的 IT 企业和技术人员，都会去认真的反思，总结经验教训。但我相信，不同的人在不同的位置上，看到的东西可能是截然相反的，甚至可能会有不少企业的管理者受到误导，开始制定更严格的规章制度，严犯运维人员再犯事。在此，我想表明一下我的态度：这是一个由运维引发的问题，但真正的根源其实不仅仅在运维，预防和治理更应该从整个企业的治理入手。

长久以来，在所有的企业中，运维部门的地位都是很边缘化的。企业的管理者会觉得运维部门是成本部门，只要能支撑业务就行。业务部门只负责提业务需求，开发部门只管做功能的开发，很多非功能性的问题无人重视，只能靠运维人员肩挑人扛到处救火，可以认为是运维部门靠自己的血肉之躯实现了业务部门的信息化。在这样的场景下，不光企业的管理者不知道该如何评价运维的价值，甚至很多运维从业者都不知道自己除了到处救火外真正应该关注什么，当然也没有时间和精力去思考。

在上文的情况下，传统的运维人员实际上是所谓的“黑盒运维”，不断的去做重复性的操作，时间长了之后，只知道自己管理的服务器能正常对外服务，但是却不知道里面应用的依赖关系，

哪些配置是有效配置、哪些是无效配置，只敢加配置，不敢删配置，欠的技术债越来越多。在这样的情况下，遇到这次携程的极端案列，需要完整的重建系统时候，就很容易一筹莫展了。

对于这样的故障，我认为真正有效的根源解决做法是从黑盒运维走向白盒运维。和 Puppet 这样的运维工具理念一致，运维的核心和难点其实是配置管理，运维人员只有真正的清楚所管理的系统的功能和配置，才能从根源上解决到处救火疲于奔命的情况，也才能真正的杜绝今天携程这样的事件重现，从根本上解决运维的问题。

从黑盒运维走向白盒运维，再进一步实现 DevOps（开发运维衔接）和软件定义数据中心，就是所谓的运维 2.0 了。很显然，这个单靠运维部门自身是做不到的，需要每一个企业的管理者、业务部门、开发部门去思考。因此，我希望今天这个事件，不要简单的让运维来背黑锅，而是让大家真正的从中得到教训和启示。



The banner features the RongCloud logo at the top left, followed by the slogan '引领即时通讯云' (Leading Instant Messaging Cloud) and '开发者首选的即时通讯云第1品牌' (Developer Preferred Instant Messaging Cloud, No. 1 Brand). The central graphic is a large white number '1' inside a cloud, surrounded by various communication icons like messages, location pins, and mobile phones, set against a blue background with a network of nodes and lines.

百姓网等多家亿级用户 APP 首选
iResearch 艾瑞等权威数据显示，融云即时通讯云市场份额稳居第一

<http://www.rongcloud.cn>

序列化和反序列化

作者 刘丁

简介

文章作者服务于美团推荐与个性化组，该组致力于为美团用户提供每天 billion 级别的高质量个性化推荐以及排序服务。从 Terabyte 级别的用户行为数据，到 Gigabyte 级别的 Deal/Poi 数据；从对实时性要求毫秒以内的用户实时地理位置数据，到定期后台 job 数据，推荐与重排序系统需要多种类型的数据服务。推荐与重排序系统客户包括各种内部服务、美团客户端、美团网站。为了提供高质量的数据服务，为了实现与上下游各系统进行良好的对接，序列化和反序列化的选型往往是我们做系统设计的一个重要考虑因素。

本文内容按如下方式组织：

- 第一部分给出了序列化和反序列化的定义，以及其在通讯协议中所处的位置；
- 第二部分从使用者的角度探讨了序列化协议的一些特性；
- 第三部分描述在具体的实施过程中典型的序列化组件，并与数据库组建进行了类比；
- 第四部分分别讲解了目前常见的几种序列化协议的特性，应用场景，并对相关组件进行举例；
- 最后一部分，基于各种协议的特性，以及相关 benchmark 数据，给出了作者的技术选型建议。

一、定义以及相关概念

互联网的产生带来了机器间通讯的需求，而互联互通的双方需要采用约定的协议，序列化和反序列化属于通讯协议的一部分。通讯协议往往采用分层模型，不同模型每层的功能定义以及颗粒度不同，例如：TCP/IP 协议是一个四层协议，而 OSI 模型却是七层协议模型。在 OSI 七层协议模型中展现层（Presentation Layer）的主要功能是把应用层的对象转换成一段连续的二进制串，或者反过来，把二进制串转换成应用层的对象--这两个功能就是序列化和反序列化。一般而言，TCP/IP 协议的应用层对应与 OSI 七层协议模型的应用层，展示层和会话层，所以序列化协议属于 TCP/IP 协议应用层的一部分。本文对序列化协议的讲解主要基于 OSI 七层协议模型。

- 序列化：将数据结构或对象转换成二进制串的过程。
- 反序列化：将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程。

数据结构、对象与二进制串

不同的计算机语言中，数据结构，对象以及二进制串的表示方式并不相同。

数据结构和对象：对于类似 Java 这种完全面向对象的语言，工程师所操作的一切都是对象（Object），来自于类的实例化。在 Java 语言中最接近数据结构的概念，就是 POJO（Plain Old Java Object）或者 Javabean——那些只有 setter/getter 方法的类。而在 C 二进制串：序列化所生成的二进制串指的是存储在内存中的一块数据。C 语言的字符串可以直接被传输层使用，因为其本质上就是以'0'结尾的存储在内存中的二进制串。在 Java 语言里面，二进制串的概念容易和 String 混淆。实际上 String 是 Java 的一等公民，是一种特殊对象（Object）。对于跨语言间的通讯，序列化后的数据当然不能是某种语言的特殊数据类型。二进制串在 Java 里面所指的是 byte[], byte 是 Java 的 8 中原生数据类型之一（Primitive data types）。

二、序列化协议特性

每种序列化协议都有优点和缺点，它们在设计之初有自己独特的应用场景。在系统设计的过程中，需要考虑序列化需求的方方面面，综合对比各种序列化协议的特性，最终给出一个折衷的方案。

通用性

通用性有两个层面的意义。

1. 技术层面，序列化协议是否支持跨平台、跨语言。如果不支持，在技术层面上的通用性就大大降低了。
2. 流行程度，序列化和反序列化需要多方参与，很少人使用的协议往往意味着昂贵的学习成本；另一方面，流行度低的协议，往往缺乏稳定而成熟的跨语言、跨平台的公共包。

健壮性

以下两个方面的原因会导致协议不够强健。

1. 成熟度不够，一个协议从制定到实施，到最后成熟往往是一个漫长的阶段。协议的强健性依赖于大量而全面的测试，对于致力于提供高质量服务的系统，采用处于测试阶段的序列化协议会带来很高的风险。
2. 语言/平台的不公平性。为了支持跨语言、跨平台的功能，序列化协议的制定者需要做大量的工作；但是，当所支持的语言或者平台之间存在难以调和的特性的时候，协议制定者需要做一个艰难的决定--支持更多人使用的语言/平台，亦或支持更多的语言/平台而放弃某个

特性。当协议的制定者决定为某种语言或平台提供更多支持的时候，对于使用者而言，协议的强健性就被牺牲了。

可调试性/可读性

序列化和反序列化的数据正确性和业务正确性的调试往往需要很长的时间，良好的调试机制会大大提高开发效率。序列化后的二进制串往往不具备人眼可读性，为了验证序列化结果的正确性，写入方不得同时撰写反序列化程序，或提供一个查询平台--这比较费时；另一方面，如果读取方未能成功实现反序列化，这将给问题查找带来了很大的挑战--难以定位是由于自身的反序列化程序的 bug 所导致还是由于写入方序列化后的错误数据所导致。对于跨公司间的调试，由于以下原因，问题会显得更严重。

1. 支持不到位，跨公司调试在问题出现后可能得不到及时的支持，这大大延长了调试周期。
2. 访问限制，调试阶段的查询平台未必对外公开，这增加了读取方的验证难度。

如果序列化后的数据人眼可读，这将大大提高调试效率， XML 和 JSON 就具有人眼可读的优点。

性能

性能包括两个方面，时间复杂度和空间复杂度。

1. 空间开销（Verbosity），序列化需要在原有的数据上加上描述字段，以为反序列化解析之用。如果序列化过程引入的额外开销过高，可能会导致过大的网络，磁盘等各方面的压力。对于海量分布式存储系统，数据量往往以 TB 为单位，巨大的额外空间开销意味着高昂的成本。
2. 时间开销（Complexity），复杂的序列化协议会导致较长的解析时间，这可能会使得序列化和反序列化阶段成为整个系统的瓶颈。

可扩展性/兼容性

移动互联时代，业务系统需求的更新周期变得更快，新的需求不断涌现，而老的系统还是需要继续维护。如果序列化协议具有良好的可扩展性，支持自动增加新的业务字段，而不影响老的服务，这将大大提供系统的灵活度。

安全性/访问限制

在序列化选型的过程中，安全性的考虑往往发生在跨局域网访问的场景。当通讯发生在公司之

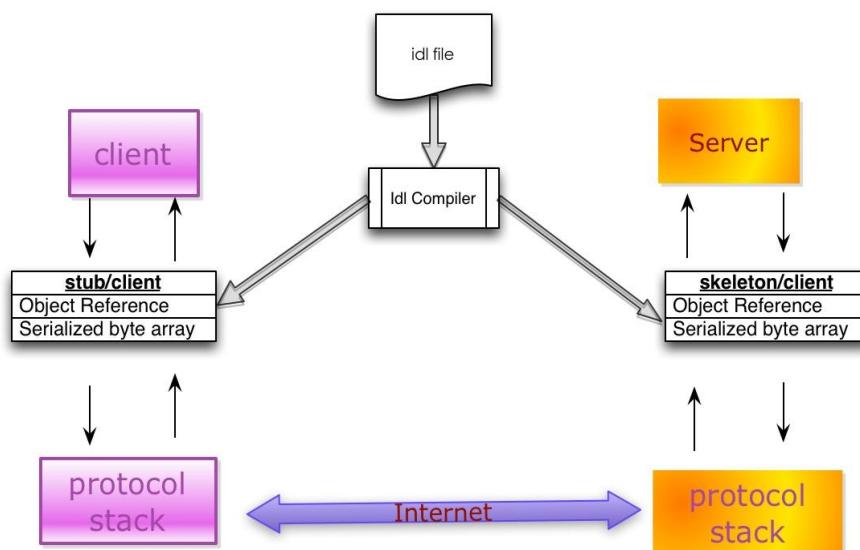
间或者跨机房的时候，出于安全的考虑，对于跨局域网的访问往往被限制为基于 HTTP/HTTPS 的 80 和 443 端口。如果使用的序列化协议没有兼容而成熟的 HTTP 传输层框架支持，可能会导致以下三种结果之一：

1. 因为访问限制而降低服务可用性；
2. 被迫重新实现安全协议而导致实施成本大大提高；
3. 开放更多的防火墙端口和协议访问，而牺牲安全性。

三、序列化和反序列化的组件

典型的序列化和反序列化过程往往需要如下组件。

- IDL (Interface description language) 文件：参与通讯的各方需要对通讯的内容需要做相关的约定 (Specifications)。为了建立一个与语言和平台无关的约定，这个约定需要采用与具体开发语言、平台无关的语言来进行描述。这种语言被称为接口描述语言 (IDL)，采用 IDL 撰写的协议约定称之为 IDL 文件。
- IDL Compiler: IDL 文件中约定的内容为了在各语言和平台可见，需要有一个编译器，将 IDL 文件转换成各语言对应的动态库。
- Stub/Skeleton Lib: 负责序列化和反序列化的工作代码。Stub 是一段部署在分布式系统客户端的代码，一方面接收应用层的参数，并对其序列化后通过底层协议栈发送到服务端，另一方面接收服务端序列化后的结果数据，反序列化后交给客户端应用层；Skeleton 部署在服务端，其功能与 Stub 相反，从传输层接收序列化参数，反序列化后交给服务端应用层，并将应用层的执行结果序列化后最终传送给客户端 Stub。
- Client/Server: 指的是应用层程序代码，他们面对的是 IDL 所生存的特定语言的 class 或 struct。
- 底层协议栈和互联网：序列化之后的数据通过底层的传输层、网络层、链路层以及物理层协议转换成数字信号在互联网中传递。



序列化组件与数据库访问组件的对比

数据库访问对于很多工程师来说相对熟悉，所用到的组件也相对容易理解。下表类比了序列化过程中用到的部分组件和数据库访问组件的对应关系，以便于大家更好的把握序列化相关组件的概念。

| | | | | | | | | | | | | |
|-------|-------|----|-------|-----|-----|-------------|---------|-----------|------------|---------------|-----------------|----------------------------|
| 序列化组件 | 数据库组件 | 说明 | ----- | IDL | DDL | 用于建表或者模型的语言 | DL file | DB Schema | 表创建文件或模型文件 | Stub/Skeleton | lib O/R mapping | 将 class 和 Table 或者数据模型进行映射 |
|-------|-------|----|-------|-----|-----|-------------|---------|-----------|------------|---------------|-----------------|----------------------------|

四、几种常见的序列化和反序列化协议

互联网早期的序列化协议主要有 COM 和 CORBA。

COM 主要用于 Windows 平台，并没有真正实现跨平台，另外 COM 的序列化的原理利用了编译器中虚表，使得其学习成本巨大（想一下这个场景，工程师需要是简单的序列化协议，但却要先掌握语言编译器）。由于序列化的数据与编译器紧耦合，扩展属性非常麻烦。

CORBA 是早期比较好的实现了跨平台，跨语言的序列化协议。CORBA 的主要问题是参与方过多带来的版本过多，版本之间兼容性较差，以及使用复杂晦涩。这些政治经济，技术实现以及早期设计不成熟的问题，最终导致 CORBA 的渐渐消亡。J2SE 1.3 之后的版本提供了基于 CORBA 协议的 RMI-IIOP 技术，这使得 Java 开发者可以采用纯粹的 Java 语言进行 CORBA 的开发。

这里主要介绍和对比几种当下比较流行的序列化协议，包括 XML、JSON、Protobuf、Thrift 和 Avro。

一个例子

如前所述，序列化和反序列化的出现往往晦涩而隐蔽，与其他概念之间往往相互包容。为了更好的让大家理解序列化和反序列化的相关概念在每种协议里面的具体实现，我们将一个例子穿插在各种序列化协议讲解中。在该例子中，我们希望将一个用户信息在多个系统里面进行传递；在应用层，如果采用 Java 语言，所面对的类对象如下所示：

```
class Address
{
    private String city;
    private String postcode;
```

```
private String street;

}

public class UserInfo

{

    private Integer userid;

    private String name;

    private List<address> address;

}

</address>
```

XML&SOAP

XML 是一种常用的序列化和反序列化协议，具有跨机器，跨语言等优点。 XML 历史悠久，其 1.0 版本早在 1998 年就形成标准，并被广泛使用至今。 XML 的最初产生目标是对互联网文档（Document）进行标记，所以它的设计理念中就包含了对于人和机器都具备可读性。但是，当这种标记文档的设计被用来序列化对象的时候，就显得冗长而复杂（Verbose and Complex）。 XML 本质上是一种描述语言，并且具有自我描述（Self-describing）的属性，所以 XML 自身就被用于 XML 序列化的 IDL。标准的 XML 描述格式有两种： DTD (Document Type Definition) 和 XSD (XML Schema Definition)。作为一种人眼可读（Human-readable）的描述语言， XML 被广泛使用在配置文件中，例如 O/R mapping、 Spring Bean Configuration File 等。

SOAP (Simple Object Access protocol) 是一种被广泛应用的，基于 XML 为序列化和反序列化协议的结构化消息传递协议。 SOAP 在互联网影响如此大，以至于我们给基于 SOAP 的解决方案一个特定的名称--Web service。 SOAP 虽然可以支持多种传输层协议，不过 SOAP 最常见的使用方式还是 XML+HTTP。 SOAP 协议的主要接口描述语言（IDL）是 WSDL (Web Service Description Language)。 SOAP 具有安全、可扩展、跨语言、跨平台并支持多种传输层协议。如果不考虑跨平台和跨语言的需求， XML 的在某些语言里面具有非常简单易用的序列化使用方法，无需 IDL 文件和第三方编译器， 例如 Java+XStream。

自我描述与递归

SOAP 是一种采用 XML 进行序列化和反序列化的协议，它的 IDL 是 WSDL. 而 WSDL 的描述文件是 XSD，而 XSD 自身是一种 XML 文件。这里产生了一种有趣的在数学上称之为“递归”的问题，这种现象往往发生在一些具有自我属性（Self-description）的事物上。

IDL 文件举例

采用 WSDL 描述上述用户基本信息的例子如下：

```
<xsd:complexType name='Address'>

  <xsd:attribute name='city' type='xsd:string' />

  <xsd:attribute name='postcode' type='xsd:string' />

  <xsd:attribute name='street' type='xsd:string' />

</xsd:complexType>

<xsd:complexType name='UserInfo'>

  <xsd:sequence>

    <xsd:element name='address' type='tns:Address' />

    <xsd:element name='address1' type='tns:Address' />

  </xsd:sequence>

  <xsd:attribute name='userid' type='xsd:int' />

  <xsd:attribute name='name' type='xsd:string' />

</xsd:complexType>
```

典型应用场景和非应用场景

SOAP 协议具有广泛的群众基础，基于 HTTP 的传输协议使得其在穿越防火墙时具有良好安全特性，XML 所具有的人眼可读（Human-readable）特性使得其具有出众的可调试性，互联网带宽的日益剧增也大大弥补了其空间开销大（Verbose）的缺点。对于在公司之间传输数据量相对小或者实时性要求相对低（例如秒级别）的服务是一个好的选择。由于 XML 的额外空间开销大，序列化之后的数据量剧增，对于数据量巨大序列持久化应用常景，这意味着巨大的内存和磁盘开销，不太适合 XML。另外，XML 的序列化和反序列化的空间和时间开销都比较大，对于对性能要求在 ms 级别的服务，不推荐使用。WSDL 虽然具备了描述对象的能力，SOAP 的 S 代表的也是 simple，但是 SOAP 的使用绝对不简单。对于习惯于面向对象编程的用户，WSDL 文件不直观。

JSON (Javascript Object Notation)

JSON 起源于弱类型语言 Javascript，它的产生来自于一种称之为"Associative array"的概念，其本质是就是采用"Attribute—value"的方式来描述对象。实际上在 Javascript 和 PHP 等弱类型语言中，类的描述方式就是 Associative array。JSON 的如下优点，使得它快速成为最广泛使用的序列化协议之一。

1. 这种 Associative array 格式非常符合工程师对对象的理解。
2. 它保持了 XML 的人眼可读（Human-readable）的优点。
3. 相对于 XML 而言，序列化后的数据更加简洁。来自于的以下链接的研究表明：XML 所产生序列化之后文件的大小接近 JSON 的两倍。
<http://www.codeproject.com/Articles/604720/JSON-vs-XML-Some-hard-numbers-about-verbosity>
4. 它具备 Javascript 的先天性支持，所以被广泛应用于 Web browser 的应用常景中，是 Ajax 的事实标准协议。
5. 与 XML 相比，其协议比较简单，解析速度比较快。
6. 松散的 Associative array 使得其具有良好的可扩展性和兼容性。

IDL 悖论

JSON 实在是太简单了，或者说太像各种语言里面的类了，所以采用 JSON 进行序列化不需要 IDL。这实在是太神奇了，存在一种天然的序列化协议，自身就实现了跨语言和跨平台。然而事实没有那么神奇，之所以产生这种假象，来自于两个原因。

1. Associative array 在弱类型语言里面就是类的概念，在 PHP 和 Javascript 里面 Associative array 就是其 class 的实际实现方式，所以在这些弱类型语言里面，JSON 得到了非常良好的支持。
2. IDL 的目的是撰写 IDL 文件，而 IDL 文件被 IDL Compiler 编译后能够产生一些代码（Stub/Skeleton），而这些代码是真正负责相应的序列化和反序列化工作的组件。但是由于 Associative array 和一般语言里面的 class 太像了，他们之间形成了一一对应关系，这就使得我们可以采用一套标准的代码进行相应的转化。对于自身支持 Associative array 的弱类型语言，语言自身就具备操作 JSON 序列化后的数据的能力；对于 Java 这强类型语言，可以采用反射的方式统一解决，例如 Google 提供的 Gson。

典型应用场景和非应用场景

JSON 在很多应用场景中可以替代 XML，更简洁并且解析速度更快。典型应用场景包括：

1. 公司之间传输数据量相对小，实时性要求相对低（例如秒级别）的服务。

2. 基于 Web browser 的 Ajax 请求。
3. 由于 JSON 具有非常强的前后兼容性，对于接口经常发生变化，并对可调式性要求高的场景，例如 Mobile app 与服务端的通讯。
4. 由于 JSON 的典型应用场景是 JSON+HTTP，适合跨防火墙访问。总的来说，采用 JSON 进行序列化的额外空间开销比较大，对于大数据量服务或持久化，这意味着巨大的内存和磁盘开销，这种场景不适合。没有统一可用的 IDL 降低了对参与方的约束，实际操作中往往只能采用文档方式来进行约定，这可能会给调试带来一些不便，延长开发周期。由于 JSON 在一些语言中的序列化和反序列化需要采用反射机制，所以在性能要求为 ms 级别，不建议使用。

IDL 文件举例

以下是 UserInfo 序列化之后的一个例子：

```
{"userid":1,"name":"messi","address":[{"city":"北京","postcode":"1000000","street":"wangjingdonglu"}]}
```

Thrift

Thrift 是 Facebook 开源提供的一个高性能，轻量级 RPC 服务框架，其产生正是为了满足当前大数据量、分布式、跨语言、跨平台数据通讯的需求。但是，Thrift 并不仅仅是序列化协议，而是一个 RPC 框架。相对于 JSON 和 XML 而言，Thrift 在空间开销和解析性能上有了比较大的提升，对于对性能要求比较高的分布式系统，它是一个优秀的 RPC 解决方案；但是由于 Thrift 的序列化被嵌入到 Thrift 框架里面，Thrift 框架本身并没有透出序列化和反序列化接口，这导致其很难和其他传输层协议共同使用（例如 HTTP）。

典型应用场景和非应用场景

对于需求为高性能，分布式的 RPC 服务，Thrift 是一个优秀的解决方案。它支持众多语言和丰富的数据类型，并对于数据字段的增删具有较强的兼容性。所以非常适用于作为公司内部的面向服务构建（SOA）的标准 RPC 框架。

不过 Thrift 的文档相对比较缺乏，目前使用的群众基础相对较少。另外由于其 Server 是基于自身的 Socket 服务，所以在跨防火墙访问时，安全是一个顾虑，所以在公司间进行通讯时需要谨慎。另外 Thrift 序列化之后的数据是 Binary 数组，不具有可读性，调试代码时相对困难。最后，由于 Thrift 的序列化和框架紧耦合，无法支持向持久层直接读写数据，所以不适合做数据持久化序列化协议。

IDL 文件举例

```
struct Address

{ 1: required string city;

 2: optional string postcode;

 3: optional string street;

} struct UserInfo

{ 1: required string userid;

 2: required i32 name;

 3: optional list<address> address;

}

</address>
```

Protobuf

Protobuf 具备了优秀的序列化协议的所需的众多典型特征。

1. 标准的 IDL 和 IDL 编译器，这使得其对工程师非常友好。
2. 序列化数据非常简洁，紧凑，与 XML 相比，其序列化之后的数据量约为 1/3 到 1/10。
3. 解析速度非常快，比对应的 XML 快约 20-100 倍。
4. 提供了非常友好的动态库，使用非常简介，反序列化只需要一行代码。

Protobuf 是一个纯粹的展示层协议，可以和各种传输层协议一起使用；Protobuf 的文档也非常完善。但是由于 Protobuf 产生于 Google，所以目前其仅仅支持 Java、C### 典型应用场景和非应用场景。Protobuf 具有广泛的用户基础，空间开销小以及高解析性能是其亮点，非常适合于公司内部的对性能要求高的 RPC 调用。由于 Protobuf 提供了标准的 IDL 以及对应的编译器，其 IDL 文件是参与各方的非常强的业务约束，另外，Protobuf 与传输层无关，采用 HTTP 具有良好的跨防火墙的访问属性，所以 Protobuf 也适用于公司间对性能要求比较高的场景。由于其解析性能高，序列化后数据量相对少，非常适合应用层对象的持久化场景。

它的主要问题在于其所支持的语言相对较少，另外由于没有绑定的标准底层传输层协议，在公司间进行传输层协议的调试工作相对麻烦。

IDL 文件举例

```
message Address

{
    required string city=1;

    optional string postcode=2;

    optional string street=3;
}

message UserInfo

{
    required string userid=1;

    required string name=2;

    repeated Address address=3;
}
```

Avro

Avro 的产生解决了 JSON 的冗长和没有 IDL 的问题，Avro 属于 Apache Hadoop 的一个子项目。Avro 提供两种序列化格式：JSON 格式或者 Binary 格式。Binary 格式在空间开销和解析性能方面可以和 Protobuf 媲美，JSON 格式方便测试阶段的调试。Avro 支持的数据类型非常丰富，包括 C#### 典型应用场景和非应用场景 Avro 解析性能高并且序列化之后的数据非常简洁，比较适合于高性能的序列化服务。

由于 Avro 目前非 JSON 格式的 IDL 处于实验阶段，而 JSON 格式的 IDL 对于习惯于静态类型语言的工程师来说不直观。

IDL 文件举例

```
protocol Userservice {

    record Address {
```

```
    string city;

    string postcode;

    string street;

}

record UserInfo {

    string name;

    int userid;

    array<Address> address = [];

}

}
```

所对应的 JSON Schema 格式如下：

```
{
    "protocol" : "Userservice",
    "namespace" : "org.apache.avro.ipc.specific",
    "version" : "1.0.5",
    "types" : [ {
        "type" : "record",
        "name" : "Address",
        "fields" : [ {
            "name" : "city",
            "type" : "string"
        }, {
            "name" : "postcode",
            "type" : "string"
        }, {
            "name" : "street",
            "type" : "string"
        } ]
    } ]
}
```

```
"name" : "postcode",
  "type" : "string"
}, {
  "name" : "street",
  "type" : "string"
} ]
},
{
  "type" : "record",
  "name" : "UserInfo",
  "fields" : [ {
    "name" : "name",
    "type" : "string"
  },
  {
    "name" : "userid",
    "type" : "int"
  },
  {
    "name" : "address",
    "type" : {
      "type" : "array",
      "items" : "Address"
    }
  },
  "default" : [ ]
}
```

```

} ]

} ] ,


"messages" : { }

}

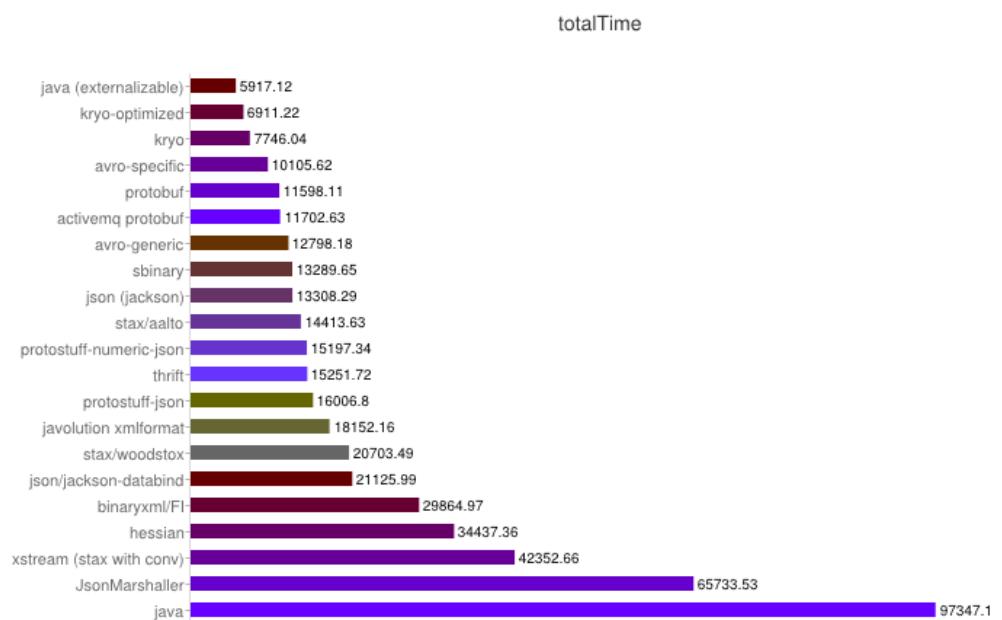
```

五、Benchmark 以及选型建议

Benchmark

以下数据来自 <https://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking>。

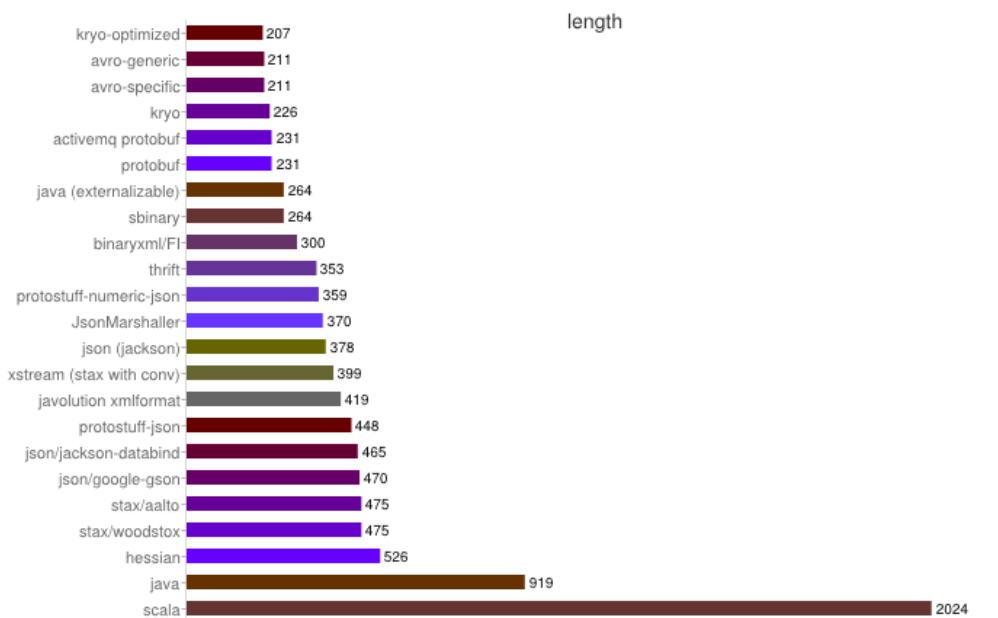
解析性能



序列化之空间开销

从下图可得出如下结论：

1. XML 序列化（Xstream）无论在性能和简洁性上比较差；
2. Thrift 与 Protobuf 相比在时空开销方面都有一定的劣势；
3. Protobuf 和 Avro 在两方面表现都非常优越。



选型建议

以上描述的五种序列化和反序列化协议都各自具有相应的特点，适用于不同的场景。

1. 对于公司间的系统调用，如果性能要求在 100ms 以上的服务，基于 XML 的 SOAP 协议是一个值得考虑的方案。
2. 基于 Web browser 的 Ajax，以及 Mobile app 与服务端之间的通讯，JSON 协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的运用场景，JSON 也是非常不错的选择。
3. 对于调试环境比较恶劣的场景，采用 JSON 或 XML 能够极大的提高调试效率，降低系统开发成本。
4. 当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro 之间具有一定的竞争关系。
5. 对于 T 级别的数据的持久化应用场景，Protobuf 和 Avro 是首要选择。如果持久化后的数据存储在 Hadoop 子项目里，Avro 会是更好的选择。
6. 由于 Avro 的设计理念偏向于动态类型语言，对于动态语言为主的运用场景，Avro 是更好的选择。
7. 对于持久层非 Hadoop 项目，以静态类型语言为主的运用场景，Protobuf 会更符合静态类型语言工程师的开发习惯。
8. 如果需要提供一个完整的 RPC 解决方案，Thrift 是一个好的选择。

9. 如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，Protobuf 可以优先考虑。

参考文献

1. <http://www.codeproject.com/Articles/604720/JSON-vs-XML-Some-hard-numbers-about-verbosity>
2. <https://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking>
3. <http://en.wikipedia.org/wiki/Serialization>
4. <http://en.wikipedia.org/wiki/Soap>
5. <http://en.wikipedia.org/wiki/XML>
6. <http://en.wikipedia.org/wiki/JSON>
7. <http://avro.apache.org/>
8. <http://www.oracle.com/technetwork/java/rmi-iip-139743.html>



ArchSummit
全球架构师峰会

听云应用性能管理(APM) 技术实践专场

[了解更多>>](#)



关注听云官方微信
获取更多信息



扫码报名！
名额有限，报名从速

高可用可伸缩架构实用经验谈

作者 李道兵

移动互联网、云计算和大数据的成熟和发展，让更多的好想法得以在很短的时间内实现为产品。此时，如果用户需求抓得准，用户数量将很可能获得爆发式增长，而不需要像以往一样需要精心运营几年的时间。然而用户数量的快速增长（尤其是短时间内的爆发式增长），通常会让应用开发者有些吃不消，不得不面临一些严峻的技术挑战：如何避免因为单台机器当机导致服务不可用；如何避免在服务容量不足时，用户体验下降，等等。在系统构建之初就采用高可用和可伸缩架构，将能有效避免这些问题。

如何构建高可用和可伸缩架构呢？七牛云存储首席架构师李道兵在 3 月 22 的「开发者最佳实践日」第十期沙龙活动上给出了自己的想法。他结合自己多年的实践经验，针对一些不太复杂的业务场景，从入口层、业务层、缓存层和数据库层四个层面细致讲述了如何构建高可用和可伸缩系统。希望大家读完这篇文章，能觉得高可用和可伸缩不是一个高不可攀的东西，投入不高的成本就能在项目早期把高可用和可伸缩纳入架构设计之中。

如何实现高可用

入口层

入口层，通常指 Nginx 和 Apache 等层面的东西，负责应用（不管是 Web 应用还是移动应用）的服务入口。我们通常会将服务定位在一个 IP，如果这个 IP 对应的服务器当机了，那么用户的访问肯定会中断。此时，可以用 keepalived 来实现入口层的高可用。例如，机器 A 的 IP 是 1.2.3.4，机器 B 的 IP 是 1.2.3.5，那么再申请一个 IP 1.2.3.6（称为心跳 IP），平时绑定在机器 A 上，如果 A 当机，IP 会自动绑定在机器 B 上；如果 B 当机，IP 会自动绑定在机器 A 上。对于这种形式，我们将 DNS 绑定到心跳 IP 上，即可实现入口层的高可用。

但这个方案有一点小问题。第一，它的切换可能会有一到两秒的中断，也就是说，如果不是要求到非常严格的毫秒级就不会有问题。第二，对入口的机器会有些浪费，因为买了两台机器的入口，可能就只有一台机器用上。对一些长连接的应用可能导致服务中断，这时候就需要客户端做配合做一些重新创建连接的工作。简单来说，对于比较普通的业务来说，这个方案就能解决一部分问题。

这里要注意，keepalived 在使用上会有一些限制。

- 两台机器必须在同一个网段，不是在同一个网段，没有办法实现互相抢 IP。
- 内网服务也可以做心跳，但需要注意的是，以前为了安全我们会把内网服务绑定在内网 IP 上，避免出现安全问题。但为了使用 keepalived，必须监听在所有 IP 上（如果监听在心跳 IP 上，那么机器没有持有该 IP 时，服务无法启动），简单的方案是启用 iptables，避免内网服务被外网访问。
- 服务器利用率下降，这时可以考虑做混合部署来改善这一点。

比较常见的一个错误是，如果有两台机器，两个公网 IP，DNS 上把域名同时定位到两个 IP，就觉得已经做了高可用。这完全不是高可用，因为如果一台机器当机，那么就有一半左右的用户无法访问。

除了 keepalive，lvs 也能用来解决入口层的高可用问题。不过，与 keepalived 相比，lvs 会更复杂一些，门槛也会高一些。

业务层

业务层通常是由 PHP、Java、Python、Go 等写的逻辑代码构成的，需要依赖于后台数据库及一些缓存层面的东西。如何实现业务层的高可用呢？最核心的就是，业务层不要有状态，将状态分散到缓存层和数据库。目前大家通常喜欢将以下几种数据放入业务层。

第一个是 session，即用户登录相关的数据，但好的做法是将 session 放在数据库里，或者一个比较稳定的缓存系统中。

第二个是缓存，在访问数据库时，如果一个查询很慢，就希望将这些结果暂时放到进程里，下次再做查询时就不用再访问数据库了。这种做法带来的问题是，当业务层服务器不只一台时，数据很难做到一致，从缓存拿到的数据就可能是错误的。。

一个简单的原则就是业务层不要有状态。在业务层没有状态时，一台业务层服务器当掉了之后，Nginx/Apache 会自动将所有的请求打到另外一台业务层的服务器上。由于没有状态，两台服务器没有任何差异，所以用户完全感受不到。如果把 session 放在业务层里面的话，那么面临的问题是，这个用户以前是登录在一台机器上的，这个进程死掉后，用户就会被登出了。

友情提醒：有一段时间比较流行 cookie session，就是将 session 中的数据加密之后放在客户的 cookie 里，然后下发到客户端，这样也能做到与服务端完全无状态。但这里面有很多坑，如果能绕过这些坑就可以这样使用。第一个坑是怎么保证加密的密钥不泄露，一旦泄露就意味着攻击者可以伪造任何人的身份。第二个坑是重放攻击，如何避免别人通过保存 cookie 去不停地尝试的验证码，当然也还有其他一些攻击手段。如果没有好办法解决这两方面的问题，那么 cookie session 尽量慎用。最好是将 session 放在一个性能比较好的数据库中。如果数据库性能不行，那么将 session 放在缓存中也比放在 cookie 里要好一点。

缓存层

非常简单的架构里是没有缓存这个概念的。但在访问量上来之后，MySQL 之类的数据仓库扛不住了，比如在 SATA 盘里跑 MySQL，QPS 到达 200、300 甚至 500 时，MySQL 的性能会大幅下降，这时就可以考虑用缓存层来挡住绝大部分服务请求，提升系统整体的容量。

缓存层做高可用一个简单的方法就是，将缓存层分得细一点儿。比如说，缓存层就一台机器的话，那么这台机器当了以后，所有应用层的压力就会往数据库里压，数据库扛不住的话，整个网站（或应用）就会随之当掉。而如果缓存层分在四台机器上的话，每台只有四分之一，这台机器当掉了以后，也只有总访问量的四分之一会压在数据库上面，数据库能扛住的话，网站就能很稳定地等到缓存层重新起来。在实践中，四分之一显然是不够的，我们会将它分得更细，以保证单台缓存当机后数据库还能撑得住即可。在中小规模下，缓存层和业务层可以混合部署，这样可以节省机器。

数据库层

在数据库层面实现高可用，通常是在软件层面来做。例如，MySQL 有主从模式(Master-Slave)，还有主主模式(Master-Master)都能满足需求。MongoDB 也有 ReplicaSet 的概念，基本都能满足大家的需求。

总之，要想实现高可用，需要做到这几点：入口层做心跳，业务层服务器无状态，缓存层减小粒度，数据库做一个主从模式。对于这种模式来讲，我们做的高可用不需要太多服务器，这些东西都可以同时部署在两台服务器上。这时，两台服务器就能满足早期的高可用需求了。任何一台服务器当机用户完全无感知。

如何实现可伸缩

入口层

在入口层实现伸缩性，可以通过直接水平扩机器，然后 DNS 加 IP 来实现。但需要注意，尽管一个域名解析到几十个 IP 没有问题，但是很多浏览器客户端只会使用前几个 IP，部分域名供应商对此有优化（如每次返回的 IP 顺序随机），但这个优化效果不稳定。

推荐的做法是使用少量的 Nginx 机器作为入口，业务服务器隐藏在内网（HTTP 类型的业务这种方式居多）。另外，也可以把所有 IP 下发到客户端，然后在客户端做一些调度（特别是非 HTTP 型的业务，如游戏、直播）。

业务层

业务层的伸缩性如何实现?与做高可用时的解决方案一样,要实现业务层的伸缩性,保证无状态是很好的手段。此外,加机器继续水平部署即可。

缓存层

比较麻烦的是缓存层的伸缩性,最简单粗暴的方式是什么呢?趁着半夜量比较低的时候,把整个缓存层全部下线,然后上线新的缓存层。新的缓存层启动起来之后,再等这些缓存慢慢预热。当然这里一个要求,你的数据库能抗住低估期的请求量。如果扛不住呢?取决于缓存类型,下面我们先可以将缓存的类型区分一下。

1. 强一致性缓存:无法接受从缓存拿到错误的数据(比如用户余额,或者会被下游继续缓存这种情形)
2. 弱一致性缓存:能接受在一段时间内从缓存拿到错误的数据(比如微博的转发数)。
3. 不变型缓存:缓存 key 对应的 value 不会变更(比如从 SHA1 推出来的密码,或者其他复杂公式的计算结果)。

那什么缓存类型伸缩性比较好呢?弱一致性和不变型缓存的扩容很方便,用一致性 Hash 即可;强一致性情况稍微复杂一些,稍后再讲。使用一致性 Hash,而不用简单 Hash 的原因是缓存的失效率。如果缓存从 9 台扩容到 10 台,简单 Hash 情况下 90% 的缓存会马上失效,而如果使用一致性 Hash 情况,只有 10% 的缓存会失效。

那么,强一致性缓存会有什么问题?第一个问题是,缓存客户端的配置更新时间会有微小的差异,在这个时间窗内有可能会拿到过期的数据。第二个问题是,如果扩容之后再裁撤节点,会拿到脏数据。比如 a 这个 key 之前在机器 1,扩容后在机器 2,数据更新了,但裁撤节点后 key 回到机器 1,这时候就会拿到脏数据。

要解决问题 2 比较简单,要么保持永不减少节点,要么节点调整间隔大于数据的有效时间。问题 1 可以用如下的步骤来解决:

1. 两套 hash 配置都更新到客户端,但仍然使用旧配置;
2. 逐个客户端改为只有两套 hash 结果一致的情况下会使用缓存,其余情况从数据库读,但写入缓存;
3. 逐个客户端通知使用新配置。

Memcache 设计得比较早,导致在伸缩性高可用方面的考虑得不太周到。Redis 在这方面有不少改进,特别是 @ngaut 团队基于 redis 开发了 codis 这个软件,一次性地解决了缓存层的绝大部分问题。推荐大家考察一下。

数据库

在数据库层面实现伸缩，方法很多，文档也很多，此处不做过多赘述。大致方法为：水平拆分、垂直拆分和定期滚动。

总之，我们可以在入口层、业务层面、缓存层和数据库层四个层面，使用刚才介绍的方法和技术实现系统高可用和可伸缩性。具体为：在入口层用心跳来做到高可用，用平行部署来伸缩；在业务层做到服务无状态；在缓存层，可以减小一些粒度，以方便实现高可用，使用一致性 Hash 将有助于实现缓存层的伸缩性；数据库层的主从模式能解决高可用问题，拆分和滚动能解决可伸缩问题。

| | 高可用 | 可伸缩 |
|-----|-------|---------|
| 入口层 | 心跳 | 平行部署 |
| 业务层 | 服务无状态 | 服务无状态 |
| 缓存层 | 减小粒度 | 一致性Hash |
| 数据库 | 主从模式 | 拆分与滚动 |

本文中分享的这些技巧和方法，主要想帮助不太复杂的业务场景或者中小型应用快速搭建起高可用可伸缩的系统。关于如何构建高可用和可伸缩系统还有很多更为细节的点和实践经验值得探讨，望以后能与大家做更充分的交流。

深入浅出 Mesos (三): 持久化存储和容错

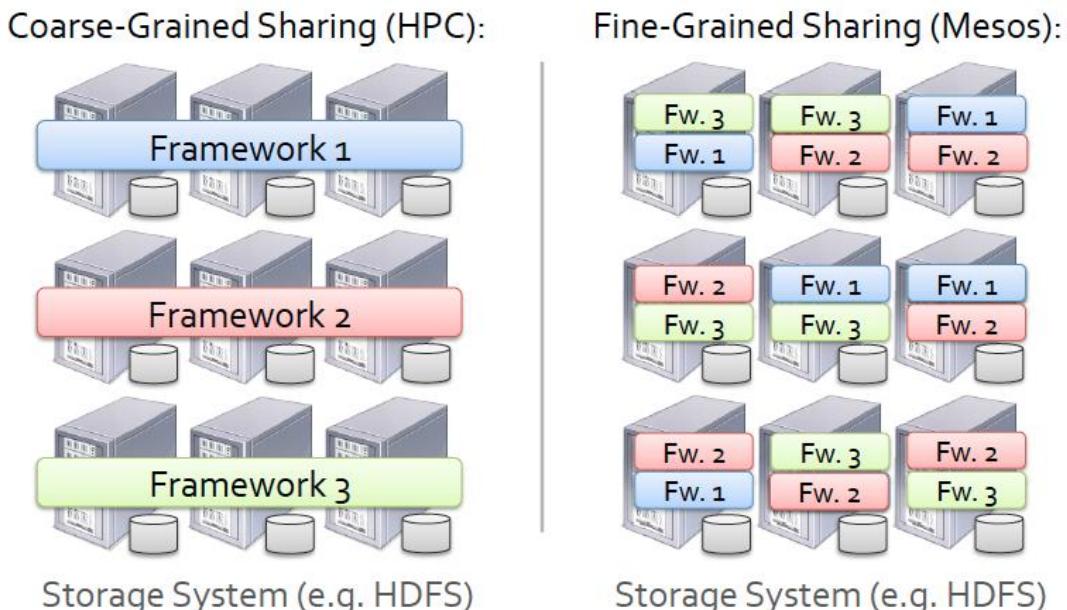
作者 韩陆

【编者按】Mesos 是 Apache 下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos 最初是由加州大学伯克利分校的 AMPLab 开发的，后在 Twitter 得到广泛使用。InfoQ 接下来将会策划系列文章来为读者剖析 Mesos。本文是整个系列的第一篇，简单介绍了 Mesos 的背景、历史以及架构。

注：本文翻译自 [Cloud Architect Musings](#)，InfoQ 中文站在获得作者授权的基础上对文章进行了翻译。

在深入浅出 Mesos 系列的[第一篇文章](#)中，我对相关的技术做了简要概述，在[第二篇文章](#)中，我深入介绍了 Mesos 的架构。完成第二篇文章之后，我本想开始着手写一篇 Mesos 如何处理资源分配的文章。不过，我收到一些读者的反馈，于是决定在谈资源分配之前，先完成这篇关于 Mesos 持久化存储和容错的文章。

持久化存储的问题

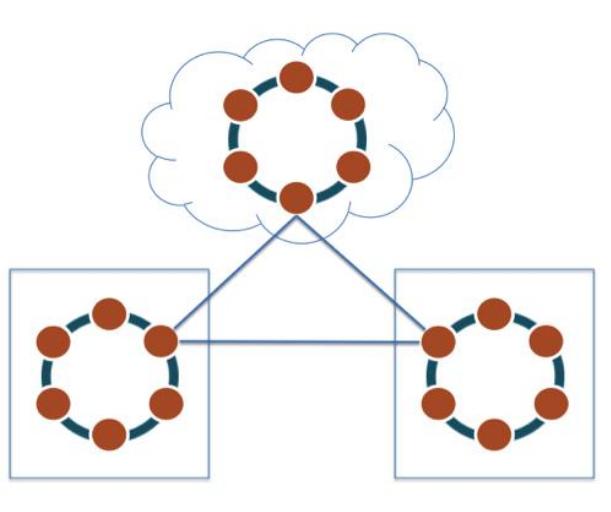


正如我在前文中讨论过的，使用 Mesos 的主要好处是可以在同一组计算节点集合上运行多种类型的应用程序（调度以及通过 Framework 初始化任务）。这些任务使用隔离模块（目

前是某些类型的容器技术)从实际节点中抽象出来,以便它们可以根据需要在不同的节点上移动和重新启动。

由此我们会思考一个问题, Mesos 是如何处理持久化存储的呢? 如果我在运行一个数据库作业, Mesos 如何确保当任务被调度时, 分配的节点可以访问其所需的数据? 如图所示, 在 Hindman 的示例中, 使用 Hadoop 文件系统 (HDFS) 作为 Mesos 的持久层, 这是 HDFS 常见的使用方式, 也是 Mesos 的执行器传递分配指定任务的配置数据给 Slave 经常使用的方式。实际上, Mesos 的持久化存储可以使用多种类型的文件系统, HDFS 只是其中之一, 但也是 Mesos 最经常使用的, 它使得 Mesos 具备了与高性能计算的亲缘关系。其实 Mesos 可以有多种选择来处理持久化存储的问题:

- **分布式文件系统。**如上所述, Mesos 可以使用 DFS (比如 HDFS 或者 Lustre) 来保证数据可以被 Mesos 集群中的每个节点访问。这种方式的缺点是会有网络延迟, 对于某些应用程序来说, 这样的网络文件系统或许并不适合。
- **使用数据存储复制的本地文件系统。**另一种方法是利用应用程序级别的复制来确保数据可被多个节点访问。提供数据存储复制的应用程序可以是 NoSQL 数据库, 比如 Cassandra 和 MongoDB。这种方式的优点是不再需要考虑网络延迟问题。缺点是必须配置 Mesos, 使特定的任务只运行在持有复制数据的节点上, 因为你不会希望数据中心的所有节点都复制相同的数据。为此, 可以使用一个 Framework, 静态地为其预留特定的节点作为复制数据的存储。



- **不使用复制的本地文件系统。**也可以将持久化数据存储在指定节点的文件系统上, 并且将该节点预留给指定的应用程序。和前面的选择一样, 可以静态地为指定应用程序预留节点, 但此时只能预留给单个节点而不是节点集合。后面两种显然不是理想的选择, 因为实质上都需要创建静态分区。然而, 在不允许延时或者应用程序不能复制它的数据存储等特殊情况下, 我们需要这样的选择。

Mesos 项目还在发展中, 它会定期增加新功能。现在我已经发现了两个可以帮助解决持久

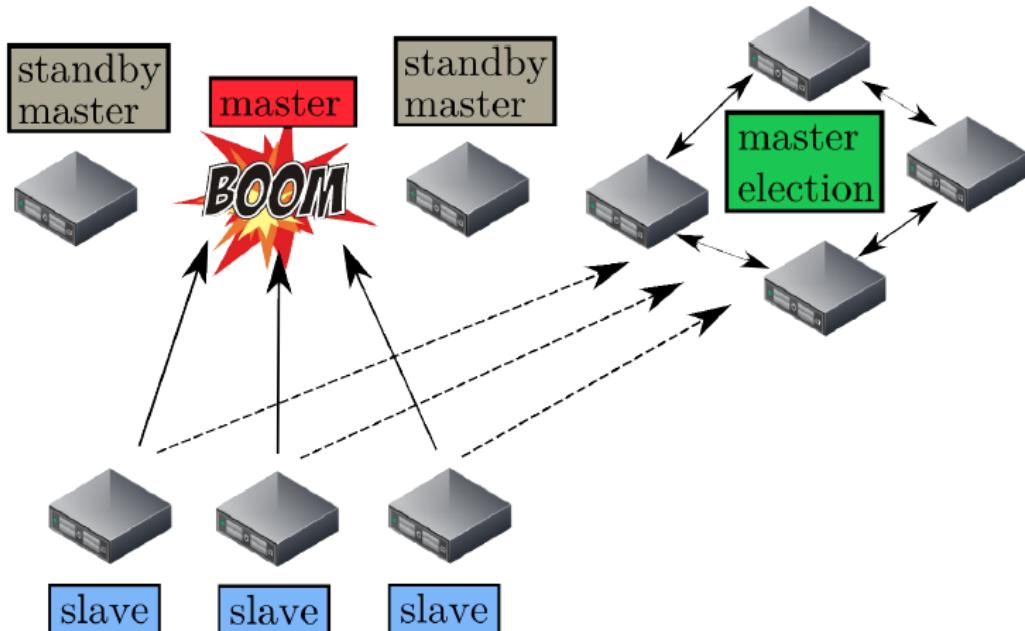
化存储问题的新特性:

- **动态预留。** Framework 可以使用这个功能框架保留指定的资源，比如持久化存储，以便在需要启动另一个任务时，资源邀约只会发送给那个 Framework。这可以在单节点和节点集合中结合使用 Framework 配置，访问永久化数据存储。关于这个建议的功能的更多信息可以从[此处](#)获得。
- **持久化卷。** 该功能可以创建一个卷，作为 Slave 节点上任务的一部分被启动，即使在任务完成后其持久化依然存在。Mesos 为需要访问相同的数据后续任务，提供在可以访问该持久化卷的节点集合上相同的 Framework 来初始化。关于这个建议的功能的更多信息可以从[此处](#)获得。

容错

接下来，我们来谈谈 Mesos 在其协议栈上是如何提供容错能力的。恕我直言，Mesos 的优势之一便是将容错设计到架构之中，并以可扩展的分布式系统的方式来实现。

- **Master。** 故障处理机制和特定的架构设计实现了 Master 的容错。

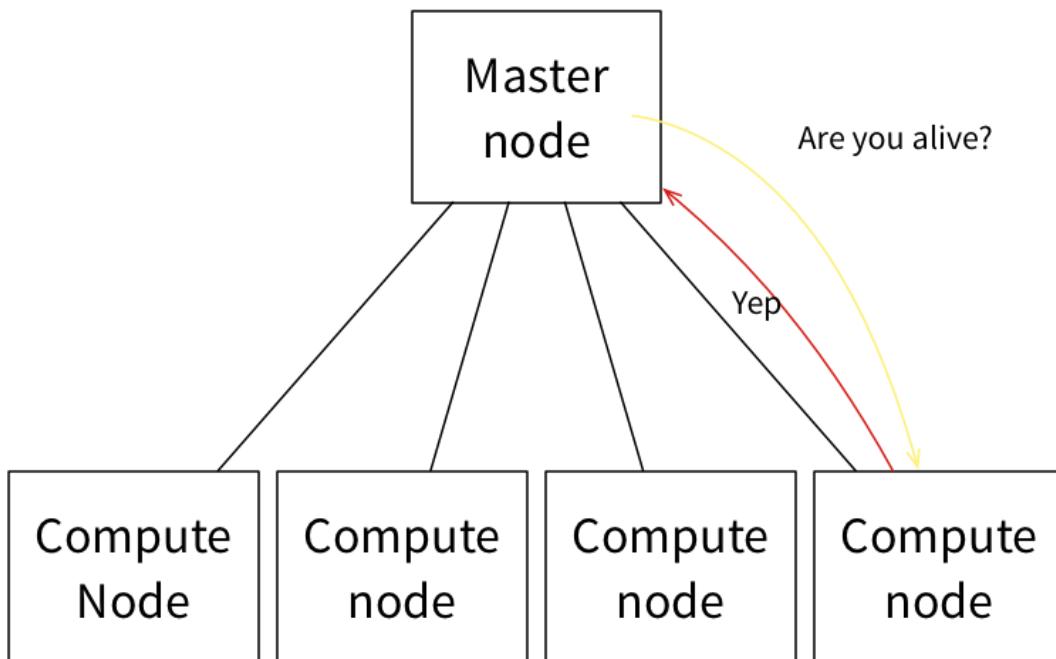


首先，Mesos 决定使用热备份（hot-standby）设计来实现 Master 节点集合。正如 Tomas Barton 对上图的说明，一个 Master 节点与多个备用（standby）节点运行在同一集群中，并由开源软件 Zookeeper 来监控。Zookeeper 会监控 Master 集群中所有的节点，并在 Master 节点发生故障时管理新 Master 的选举。建议的节点总数是 5 个，实际上，生产环境至少需要 3 个 Master 节点。Mesos 决定将 Master 设计为持有软件状态，这意味着当 Master 节点发生故障时，其状态可以很快地在新选举的 Master 节点上重建。Mesos 的状态信息

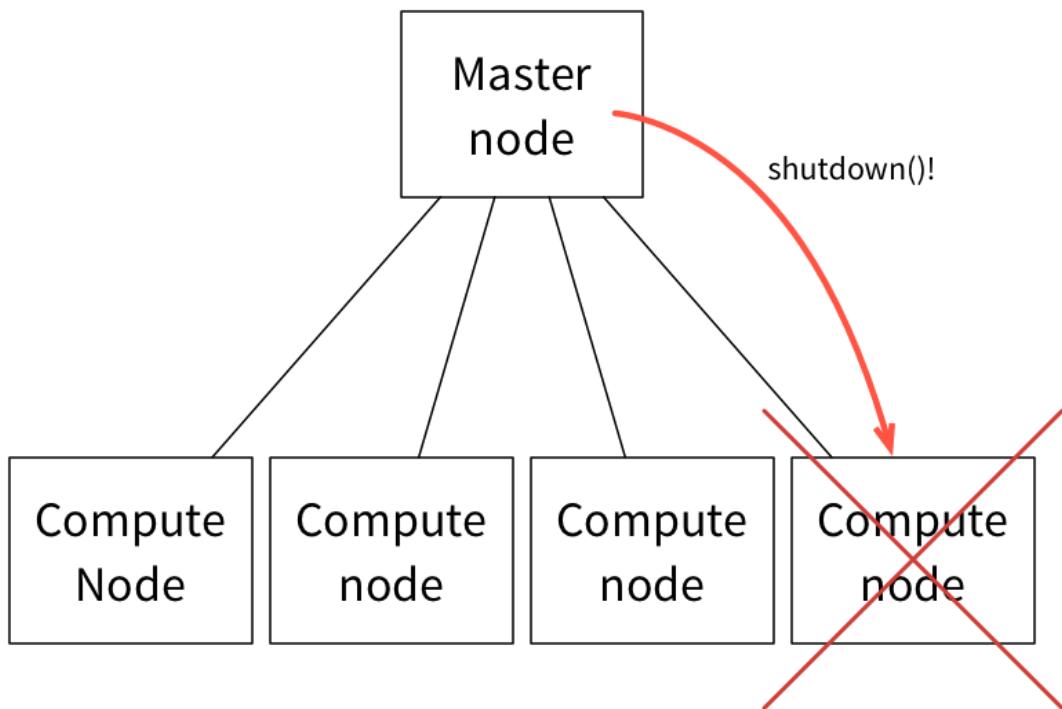
实际上驻留在 Framework 调度器和 Slave 节点集合之中。当一个新的 Master 当选后，Zookeeper 会通知 Framework 和选举后的 Slave 节点集合，以便使其在新的 Master 上注册。彼时，新的 Master 可以根据 Framework 和 Slave 节点集合发送过来的信息，重建内部状态。

- **Framework 调度器。** Framework 调度器的容错是通过 Framework 将调度器注册 2 份或者更多份到 Master 来实现。当一个调度器发生故障时，Master 会通知另一个调度来接管。需要注意的是 Framework 自身负责实现调度器之间共享状态的机制。
- **Slave。** Mesos 实现了 Slave 的恢复功能，当 Slave 节点上的进程失败时，可以让执行器/任务继续运行，并为那个 Slave 进程重新连接那台 Slave 节点上运行的执行器/任务。当任务执行时，Slave 会将任务的监测点元数据存入本地磁盘。如果 Slave 进程失败，任务会继续运行，当 Master 重新启动 Slave 进程后，因为此时没有可以响应的消息，所以重新启动的 Slave 进程会使用检查点数据来恢复状态，并重新与执行器/任务连接。

如下情况则截然不同，计算节点上 Slave 正常运行而任务执行失败。在此，Master 负责监控所有 Slave 节点的状态。



当计算节点/Slave 节点无法响应多个连续的消息后，Master 会从可用资源的列表中删除该节点，并会尝试关闭该节点。



然后，Master 会向分配任务的 Framework 调度器汇报执行器/任务失败，并允许调度器根据其配置策略做任务失败处理。通常情况下，Framework 会重新启动任务到新的 Slave 节点，假设它接收并接受来自 Master 的相应的资源邀约。

- **执行器/任务。**与计算节点/Slave 节点故障类似，Master 会向分配任务的 Framework 调度器汇报执行器/任务失败，并允许调度器根据其配置策略在任务失败时做出相应的处理。通常情况下，Framework 在接收并接受来自 Master 的相应的资源邀约后，会在新的 Slave 节点上重新启动任务。

结论

在接下来的文章中，我将更深入到资源分配模块。同时，我非常期待读者的反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。我非全知，虚心求教，所以期待读者的校正和启示。我也会在 [twitter](#) 响应你的反馈，请关注 @hui_kenneth。

查看英文原文： [DEALING WITH PERSISTENT STORAGE AND FAULT TOLERANCE IN APACHE MESOS](#)

InfoQ 在线课堂

从一个用户到千万用户的云计算架构

时间：2015年7月1日（周三）晚8点



深入浅出 Mesos (四): Mesos 的资源分配

作者 韩陆

【编者按】Mesos 是 Apache 下的开源分布式资源管理框架，它被称为是分布式系统的内核。Mesos 最初是由加州大学伯克利分校的 AMPLab 开发的，后在 Twitter 得到广泛使用。InfoQ 接下来将会策划系列文章来为读者剖析 Mesos。本文是整个系列的第一篇，简单介绍了 Mesos 的背景、历史以及架构。

注：本文翻译自 [Cloud Architect Musings](#)，InfoQ 中文站在获得作者授权的基础上对文章进行了翻译。

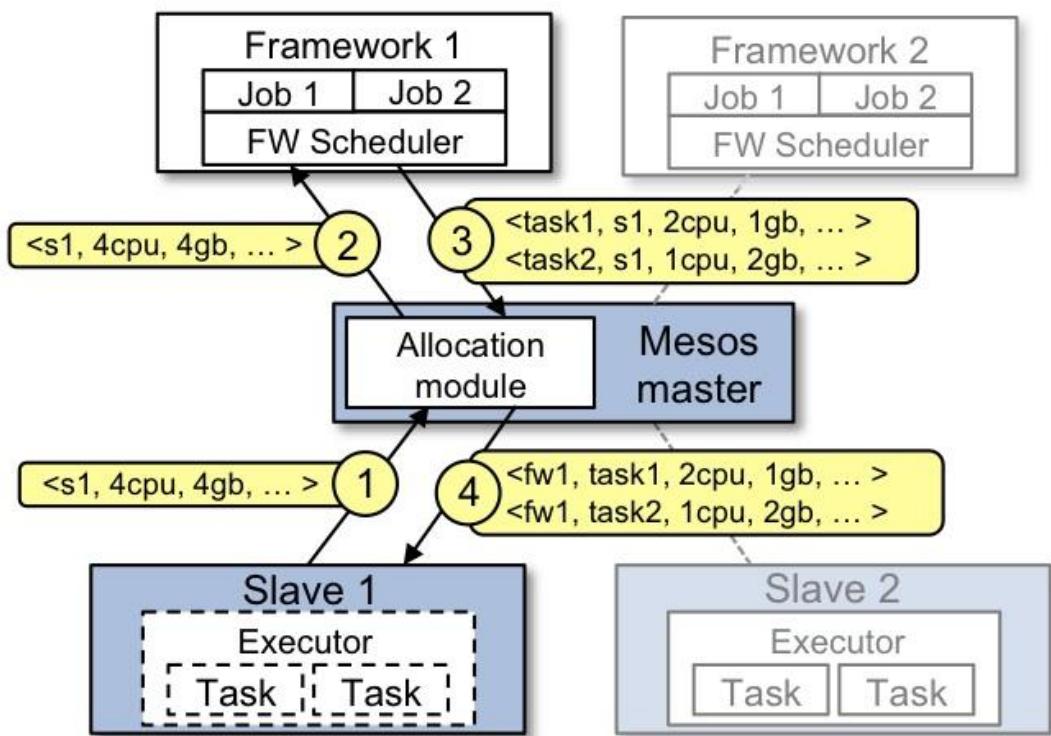
Apache Mesos 能够成为最优秀的数据中心资源管理器的一个重要功能是面对各种类型的应用，它具备像交警一样的疏导能力。本文将深入 Mesos 的资源分配内部，探讨 Mesos 是如何根据客户应用需求，平衡公平资源共享的。在开始之前，如果读者还没有阅读这个系列的前序文章，建议首先阅读它们。第一篇是 [Mesos 的概述](#)，第二篇是 [两级架构的说明](#)，第三篇是 [数据存储和容错](#)。



我们将探讨 Mesos 的资源分配模块，看看它是如何确定将什么样的资源邀约发送给具体哪个 Framework，以及在必要时如何回收资源。让我们先来回顾一下 Mesos 的任务调度过程（见下图）。

从前面提到的 [两级架构的说明](#) 一文中我们知道，Mesos Master 代理任务的调度首先从 Slave 节点收集有关可用资源的信息，然后以资源邀约的形式，将这些资源提供给注册其上的 Framework。

Framework 可以根据是否符合任务对资源的约束，选择接受或拒绝资源邀约。一旦资源邀约被接受，Framework 将与 Master 协作调度任务，并在数据中心的相应 Slave 节点上运行任务。



如何作出资源邀约的决定是由资源分配模块实现的，该模块存在于 Master 之中。资源分配模块确定 Framework 接受资源邀约的顺序，与此同时，确保在本性贪婪的 Framework 之间公平地共享资源。在同质环境中，比如 Hadoop 集群，使用最多的公平份额分配算法之一是最大最小公平算法（max-min fairness）。[最大最小公平算法](#)算法将最小的资源分配最大化，并将其提供给用户，确保每个用户都能获得公平的资源份额，以满足其需求所需的资源；一个简单的例子能够说明其工作原理，请参考[最大最小公平份额算法页面](#)的示例 1。如前所述，在同质环境下，这通常能够很好地运行。同质环境下的资源需求几乎没有波动，所涉及的资源类型包括 CPU、内存、网络带宽和 I/O。然而，在跨数据中心调度资源并且是异构的资源需求时，资源分配将会更加困难。例如，当用户 A 的每个任务需要 1 核 CPU、4GB 内存，而用户 B 的每个任务需要 3 核 CPU、1GB 内存时，如何提供合适的公平份额分配策略？当用户 A 的任务是内存密集型，而用户 B 的任务是 CPU 密集型时，如何公平地为其分配一揽子资源？

因为 Mesos 是专门管理异构环境中的资源，所以它实现了一个可插拔的资源分配模块架构，将特定部署最适合的分配策略和算法交给用户去实现。例如，用户可以实现加权的最大最小公平性算法，让指定的 Framework 相对于其它的 Framework 获得更多的资源。默认情况下，Mesos 包括一个严格优先级的资源分配模块和一个改良的公平份额资源分配模块。严格优先级模块实现的算法给定 Framework 的优先级，使其总是接收并接受足以满足其任务要求的资源邀约。这保证了关键应用在 Mesos 中限制动态资源份额上的开销，但是会潜在其他 Framework 饥饿的情况。

由于这些原因，大多数用户默认使用 DRF（主导资源公平算法 Dominant Resource Fairness），这是 Mesos 中更适合异质环境的改良公平份额算法。

DRF 和 Mesos 一样出自 Berkeley AMPLab 团队，并且作为 Mesos 的默认资源分配策略实现编码。

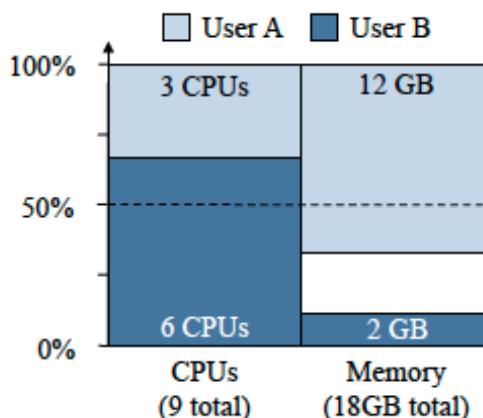
读者可以从[此处](#)和[此处](#)阅读 DRF 的原始论文。在本文中，我将总结其中要点并提供一些例子，相信这样会更清晰地解读 DRF。让我们开始揭秘之旅。

DRF 的目标是确保每一个用户，即 Mesos 中的 Framework，在异质环境中能够接收到其最需资源的公平份额。为了掌握 DRF，我们需要了解主导资源（dominant resource）和主导份额（dominant share）的概念。Framework 的主导资源是其最需的资源类型（CPU、内存等），在资源邀约中以可用资源百分比的形式展示。例如，对于计算密集型的任务，它的 Framework 的主导资源是 CPU，而依赖于在内存中计算的任务，它的 Framework 的主导资源是内存。因为资源是分配给 Framework 的，所以 DRF 会跟踪每个 Framework 拥有的资源类型的份额百分比；Framework 拥有的全部资源类型份额中占最高百分比的就是 Framework 的主导份额。DRF 算法会使用所有已注册的 Framework 来计算主导份额，以确保每个 Framework 能接收到其主导资源的公平份额。

概念过于抽象了吧？让我们用一个例子来说明。假设我们有一个资源邀约，包含 9 核 CPU 和 18GB 的内存。Framework 1 运行任务需要（1 核 CPU、4GB 内存），Framework 2 运行任务需要（3 核 CPU、1GB 内存）

Framework 1 的每个任务会消耗 CPU 总数的 $1/9$ 、内存总数的 $2/9$ ，因此 Framework 1 的主导资源是内存。同样，Framework 2 的每个任务会 CPU 总数的 $1/3$ 、内存总数的 $1/18$ ，因此 Framework 2 的主导资源是 CPU。DRF 会尝试为每个 Framework 提供等量的主导资源，作为他们的主导份额。在这个例子中，DRF 将协同 Framework 做如下分配：Framework 1 有三个任务，总分配为（3 核 CPU、12GB 内存），Framework 2 有两个任务，总分配为（6 核 CPU、2GB 内存）。

此时，每个 Framework 的主导资源（Framework 1 的内存和 Framework 2 的 CPU）最终得到相同的主导份额（ $2/3$ 或 67% ），这样提供给两个 Framework 后，将没有足够的可用资源运行其他任务。需要注意的是，如果 Framework 1 中仅有两个任务需要被运行，那么 Framework 2 以及其他已注册的 Framework 将收到的所有剩余的资源。



那么，DRF 是怎样计算而产生上述结果的呢？如前所述，DRF 分配模块跟踪分配给每个 Framework 的资源和每个框架的主导份额。每次，DRF 以所有 Framework 中运行的任务中最低的主导份额作为资源邀约发送给 Framework。如果有足够的可用资源来运行它的任务，Framework 将接受这个邀约。通过前面引述的 DRF 论文中的示例，我们来贯穿 DRF 算法的每个步骤。为了简单起见，示例将不考虑短任务完成后，资源被释放回资源池中这一因素，我们假设每个 Framework 会有无限数量的任务要运行，并认为每个资源邀约都会被接受。

回顾上述示例，假设有一个资源邀约包含 9 核 CPU 和 18GB 内存。Framework 1 运行的任务需要（1 核 CPU、4GB 内存），Framework 2 运行的任务需要（3 核 CPU、2GB 内存）。Framework 1 的任务会消耗 CPU 总数的 $1/9$ 、内存总数的 $2/9$ ，Framework 1 的主导资源是内存。同样，Framework 2 的每个任务会 CPU 总数的 $1/3$ 、内存总数的 $1/18$ ，Framework 2 的主导资源是 CPU。

| Framework Chosen | Framework 1 | | | Framework 2 | | | CPU Total Allocation | RAM Total Allocation |
|------------------|-----------------|----------------|------------------|-----------------|----------------|------------------|----------------------|----------------------|
| | Resource Shares | Dominant Share | Dominant Share % | Resource Shares | Dominant Share | Dominant Share % | | |
| | 0/9, 0/18 | 0 | 0% | 0/9, 0/18 | 0 | 0% | 0/9 | 0/18 |
| Framework 2 | 0/9, 0/18 | 0 | 0% | 3/9, 1/18 | 1/3 | 33% | 3/9 | 1/18 |
| Framework 1 | 1/9, 4/18 | 2/9 | 22% | 3/9, 1/18 | 1/3 | 33% | 4/9 | 5/18 |
| Framework 1 | 2/9, 8/18 | 4/9 | 44% | 3/9, 1/18 | 1/3 | 33% | 5/9 | 9/18 |
| Framework 2 | 2/9, 8/18 | 4/9 | 44% | 6/9, 2/18 | 2/3 | 67% | 8/9 | 10/18 |
| Framework 1 | 3/9, 12/18 | 2/3 | 67% | 6/9, 2/18 | 2/3 | 67% | 9/9 | 14/18 |

上面表中的每一行提供了以下信息：

- Framework chosen——收到最新资源邀约的 Framework。
- Resource Shares——给定时间内 Framework 接受的资源总数，包括 CPU 和内存，以占资源总量的比例表示。
- Dominant Share（主导份额）——给定时间内 Framework 主导资源占总份额的比例，以占资源总量的比例表示。
- Dominant Share %（主导份额百分比）——给定时间内 Framework 主导资源占总份额的百分比，以占资源总量的百分比表示。
- CPU Total Allocation——给定时间内接受的所有 Framework 的总 CPU 资源。
- RAM Total Allocation——给定时间内接受的所有 Framework 的总内存资源。

注意，每个行中的最低主导份额以粗体字显示，以便查找。

最初，两个 Framework 的主导份额是 0%，我们假设 DRF 首先选择的是 Framework 2，当然我们也可以假设 Framework 1，但是最终的结果是一样的。

1. Framework 2 接收份额并运行任务，使其主导资源成为 CPU，主导份额增加至 33%。
2. 由于 Framework 1 的主导份额维持在 0%，它接收共享并运行任务，主导份额的主导资源（内存）增加至 22%。

3. 由于 Framework 1 仍具有较低的主导份额，它接收下一个共享并运行任务，增加其主导份额至 44%。
4. 然后 DRF 将资源邀约发送给 Framework 2，因为它现在拥有更低的主导份额。
5. 该过程继续进行，直到由于缺乏可用资源，不能运行新的任务。在这种情况下，CPU 资源已经饱和。
6. 然后该过程将使用一组新的资源邀约重复进行。

需要注意的是，可以创建一个资源分配模块，使用加权的 DRF 使其偏向某个 Framework 或某组 Framework。如前面所提到的，也可以创建一些自定义模块来提供组织特定的分配策略。

一般情况下，现在大多数的任务是短暂的，Mesos 能够等待任务完成并重新分配资源。然而，集群上也可以跑长时间运行的任务，这些任务用于处理挂起作业或行为不当的 Framework。

值得注意的是，在当资源释放的速度不够快的情况下，资源分配模块具有撤销任务的能力。Mesos 尝试如此撤销任务：向执行器发送请求结束指定的任务，并给出一个宽限期让执行器清理该任务。如果执行器不响应请求，分配模块就结束该执行器及其上的所有任务。

分配策略可以实现为，通过提供与 Framework 相关的保证配置，来阻止对指定任务的撤销。如果 Framework 低于保证配置，Mesos 将不能结束该 Framework 的任务。

我们还需了解更多关于 Mesos 资源分配的知识，但是我将戛然而止。接下来，我要说点不同的东西，是关于 Mesos 社区的。我相信这是一个值得考虑的重要话题，因为开源不仅包括技术，还包括社区。

说完社区，我将会写一些关于 Mesos 的安装和 Framework 的创建和使用的，逐步指导的教程。在一番实操教学的文章之后，我会回来做一些更深入的话题，比如 Framework 与 Master 是如何互动的，Mesos 如何跨多个数据中心工作等。

与往常一样，我鼓励读者提供反馈，特别是关于如果我打标的地方，如果你发现哪里不对，请反馈给我。我非全知，虚心求教，所以非常期待读者的校正和启示。我们也可以在 [twitter](#) 上沟通，请关注 @hui_kenneth。

查看英文原文：[PLAYING TRAFFIC COP: RESOURCE ALLOCATION IN APACHE MESOS](#)

戏（细）说 Executor 框架线程池任务执行全过程（上）

作者 张超盟

一、前言

1.5 后引入的 Executor 框架的最大优点是把任务的提交和执行解耦。要执行任务的人只需把 Task 描述清楚，然后提交即可。这个 Task 是怎么被执行的，被谁执行的，什么时候执行的，提交的人就不用关心了。具体点讲，提交一个 Callable 对象给 ExecutorService（如最常用的线程池 ThreadPoolExecutor），将得到一个 Future 对象，调用 Future 对象的 get 方法等待执行结果就好了。

经过这样的封装，对于使用者来说，提交任务获取结果的过程大大简化，调用者直接从提交的地方就可以等待获取执行结果。而封装最大的效果是使得真正执行任务的线程们变得不为人知。有没有觉得这个场景似曾相识？我们工作中当老大的老大（且称作 LD²）把一个任务交给我们老大（LD）的时候，到底是 LD 自己干，还是转过身来拉来一帮苦逼的兄弟加班加点干，那 LD² 是不管的。LD² 只用把人描述清楚提及给 LD，然后喝着咖啡等着收 LD 的 report 即可。等 LD 一封邮件非常优雅地报告 LD²report 结果时，实际操作中是码农 A 和码农 B 干了一个月，还是码农 ABCDE 加班干了一个礼拜，大多是不用体现的。这套机制的优点就是 LD² 找个合适的 LD 出来提交任务即可，接口友好有效，不用为具体怎么干费神费力。

二、一个最简单的例子

看上去这个执行过程是这个样子。调用这段代码的是老大的老大了，他所需要干的所有事情就是找到一个合适的老大（如下面例子中 laodaA 就荣幸地被选中了），提交任务就好了。

```
// 一个有 7 个作业线程的线程池，老大的老大找到一个管 7 个人的小团队的老大
```

```
ExecutorService laodaA = Executors.newFixedThreadPool(7);
```

```
// 提交作业给老大，作业内容封装在 Callable 中，约定好了输出的类型是 String。
```

```
String outputs = laodaA.submit(
```

```
new Callable<String>() {  
  
    public String call() throws Exception  
  
    {  
  
        return "I am a task, which submitted by the so called  
laoda, and run by those anonymous workers";  
  
    }  
  
    //提交后就等着结果吧，到底是手下 7 个作业中谁领到任务了，老大是不关心的。  
  
}).get();  
  
System.out.println(outputs);
```

使用上非常简单，其实只有两行语句来完成所有功能：创建一个线程池，提交任务并等待获取执行结果。

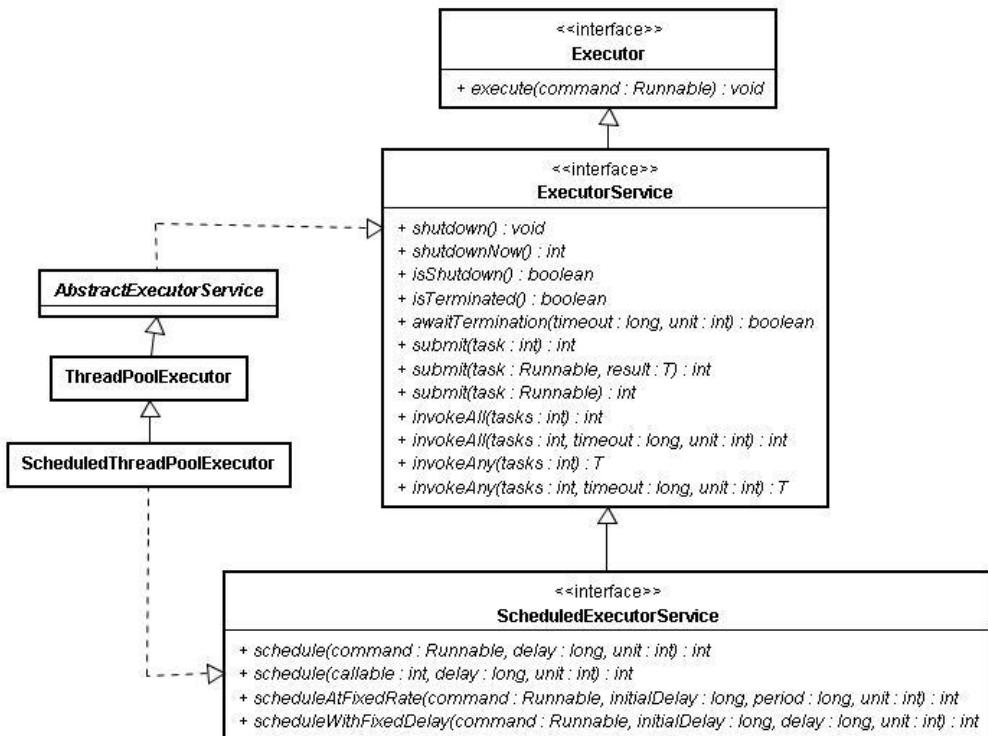
例子中生成线程池采用了工具类 Executors 的静态方法。除了 newFixedThreadPool 可以生成固定大小的线程池，newCachedThreadPool 可以生成一个无界、可以自动回收的线程池，newSingleThreadScheduledExecutor 可以生成一个单个线程的线程池。newScheduledThreadPool 还可以生成支持周期任务的线程池。一般用户场景下各种不同设置要求的线程池都可以这样生成，不用自己 new 一个线程池出来。

三、代码剖析

这套机制怎么用，上面两句话就做到了，非常方便和友好。但是 submit 的 task 是怎么被执行的？是谁执行的？如何做到在调用的时候只有等待执行结束才能 get 到结果。这些都是 1.5 之后 Executor 接口下的线程池、Future 接口下的可获得执行结果的任务，配合 AQS 和原有的 Runnable 来做到的。在下文中我们尝试通过剖析每部分的代码来了解 Task 提交，Task 执行，获取 Task 执行结果等几个主要步骤。为了控制篇幅，突出主要逻辑，文章中引用的代码片段去掉了异常捕获、非主要条件判断、非主要操作。文中只是以最常用的 ThreadPoolExecutor 线程池举例，其实 ExecutorService 接口下定义了很多功能丰富的其他类型，有各自的特点，但风格类似。本文重点是介绍任务提交的过程，过程中涉及的 ExecutorService、ThreadPoolExecutor、AQS、Future、FutureTask 等只会介绍该过程中用到的内容，不会对每个类都详细展开。

1、任务提交

从类图上可以看到，接口 ExecutorService 继承自 Executor。不像 Executor 中只定义了一个方法来执行任务，在 ExecutorService 中，正如其名字暗示的一样，定义了一个服务，定义了完整的线程池的行为，可以接受提交任务、执行任务、关闭服务。抽象类 AbstractExecutorService 类实现了 ExecutorService 接口，也实现了接口定义的默认行为。



AbstractExecutorService 任务提交的 submit 方法有三个实现。第一个接收一个 Runnable 的 Task，没有执行结果；第二个是两个参数：一个任务，一个执行结果；第三个一个 Callable，本身就包含执任务内容和执行结果。 submit 方法的返回结果是 Future 类型，调用该接口定义的 get 方法即可获得执行结果。V get() 方法的返回值类型 V 是在提交任务时就约定好了的。

除了 submit 任务的方法外，作为对服务的管理，在 ExecutorService 接口中还定义了服务的关闭方法 shutdown 和 shutdownNow 方法，可以平缓或者立即关闭执行服务，实现该方法的子类根据自身特征支持该定义。在 ThreadPoolExecutor 中，维护了 RUNNING、SHUTDOWN、STOP、TERMINATED 四种状态来实现对线程池的管理。线程池的完整运行机制不是本文的重点，重点还是关注 submit 过程中的逻辑。

- 看 AbstractExecutorService 中代码提交部分，构造好一个 FutureTask 对象后，调用 execute()方法执行任务。我们知道这个方法是顶级接口 Executor 中定义的最重要的方

法。。FutureTask 类型实现了 Runnable 接口，因此满足 Executor 中 execute()方法的约定。同时比较有意思的是，该对象在 execute 执行后，就又作为 submit 方法的返回值返回，因为 FutureTask 同时又实现了 Future 接口，满足 Future 接口的约定。

```
public <T> Future<T> submit(Callable<T> task) {  
  
    if (task == null) throw new NullPointerException();  
  
    RunnableFuture<T> ftask = newTaskFor(task);  
  
    execute(ftask);  
  
    return ftask;  
}
```

2. Submit 传入的参数都被封装成了 FutureTask 类型来 execute 的，对应前面三个不同的参数类型都会封装成 FutureTask。

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {  
  
    return new FutureTask<T>(callable);  
}
```

3. Executor 接口中定义的 execute 方法的作用就是执行提交的任务，该方法在抽象类 AbstractExecutorService 中没有实现，留到子类中实现。我们观察下子类 ThreadPoolExecutor，使用最广泛的线程池如何来 execute 那些 submit 的任务的。这个方法看着比较简单，但是线程池什么时候创建新的作业线程来处理任务，什么时候只接收任务不创建作业线程，另外什么时候拒绝任务。线程池的接收任务、维护工作线程的策略都要在其中体现。

作为必要的预备知识，先补充下 ThreadPoolExecutor 有两个最重要的集合属性，分别是存储接收任务的任务队列和用来干活的作业集合。

```
//任务队列  
  
private final BlockingQueue<Runnable> workQueue;  
  
//作业线程集合  
  
private final HashSet<Worker> workers = new HashSet<Worker>();
```

其中阻塞队列 `workQueue` 是来存储待执行的任务的，在构造线程池时可以选择满足该 `BlockingQueue` 接口定义的 `SynchronousQueue`、`LinkedBlockingQueue` 或者 `DelayedWorkQueue` 等不同阻塞队列来实现不同特征的线程池。

关注下 `execute(Runnable command)` 方法中调用到的 `addIfUnderCorePoolSize`, `workQueue.offer(command)`, `ensureQueuedTaskHandled(command)`, `addIfUnderMaximumPoolSize(command)` 这几个操作。尤其几个名字较长的 `private` 方法，把方法名的驼峰式的单词分开，加上对方法上下文的了解就能理解其功能。

因为前面说到的几个方法在里面即是操作，又返回一个布尔值，影响后面的逻辑，所以不大方便在方法体中为每条语句加注释来说明，需要大致关联起来看。所以首先需要把 `execute` 方法的主要逻辑说明下，再看其中各自方法的作用。

- 如果线程池的状态是 `RUNNING`，线程池的大小小于配置的核心线程数，说明还可以创建新线程，则启动新的线程执行这个任务。
- 如果线程池的状态是 `RUNNING`，线程池的大小小于配置的最大线程数，并且任务队列已经满了，说明现有线程已经不能支持当前的任务了，并且线程池还有继续扩充的空间，就可以创建一个新的线程来处理提交的任务。
- 如果线程池的状态是 `RUNNING`，当前线程池的大小大于等于配置的核心线程数，说明根据配置当前的线程数已经够用，不用创建新线程，只需把任务加入任务队列即可。如果任务队列不满，则提交的任务在任务队列中等待处理；如果任务队列满了则需要考虑是否要扩展线程池的容量。
- 当线程池已经关闭或者上面的条件都不能满足时，则进行拒绝策略，拒绝策略在 `RejectedExecutionHandler` 接口中定义，可以有多种不同的实现。

上面其实也是对最主要思路的解析，详细展开可能还会更复杂。简单梳理下思路：构建线程池时定义了一个额定大小，当线程池内工作线程数小于额定大小，有新任务进来就创建新工作线程，如果超过该阈值，则一般就不创建了，只是把接收任务加到任务队列里面。但是如果任务队列里的任务实在太多了，那还是要申请额外的工作线程来帮忙。如果还是不够用就拒绝服务。这个场景其实也是每天我们工作中会碰到的场景。我们管人的老大，手里都有一定 `HC` (`Head Count`)，当上面老大有活分下来，手里人不够，但是不超过 `HC`，我们就自己招人；如果超过了还是忙不过来，那就向上门老大申请借调人手来帮忙；如果还是干不完，那就没办法了，新任务咱就不接了。

```
public void execute(Runnable command) {  
  
    if (command == null)  
  
        throw new NullPointerException();  
  
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {  
  
        if (runState == RUNNING && workQueue.offer(command)) {  
    }  
}
```

```
        if (runState != RUNNING || poolSize == 0)

            ensureQueuedTaskHandled(command);

    }

    else if (!addIfUnderMaximumPoolSize(command))

        reject(command); // is shutdown or saturated

    }

}
```

4. `addIfUnderCorePoolSize` 方法检查如果当前线程池的大小小于配置的核心线程数，说明还可以创建新线程，则启动新的线程执行这个任务。

```
private boolean addIfUnderCorePoolSize(Runnable firstTask) {

    Thread t = null;

    //如果当前线程池的大小小于配置的核心线程数，说明还可以创建新线程

    if (poolSize < corePoolSize && runState == RUNNING)

        // 则启动新的线程执行这个任务

        t = addThread(firstTask);

    return t != null;

}
```

5. 和上一个方法类似，`addIfUnderMaximumPoolSize` 检查如果线程池的大小小于配置的最大线程数，并且任务队列已经满了（就是 `execute` 方法试图把当前线程加入任务队列时不成功），说明现有线程已经不能支持当前的任务了，但线程池还有继续扩充的空间，就可以创建一个新的线程来处理提交的任务。

```
private boolean addIfUnderMaximumPoolSize(Runnable firstTask) {

    // 如果线程池的大小小于配置的最大线程数，并且任务队列已经满了（就

    是 execute 方法中试图把当前线程加入任务队列 workQueue.offer(command) 时候不成功
```

) , 说明现有线程已经不能支持当前的任务了, 但线程池还有继续扩充的空间

```
if (poolSize < maximumPoolSize && runState == RUNNING)

//就可以创建一个新的线程来处理提交的任务

t = addThread(firstTask);

return t != null;

}
```

6. 在 ensureQueuedTaskHandled 方法中, 判断如果当前状态不是 RUNING, 则当前任务不加入到任务队列中, 判断如果状态是停止, 线程数小于允许的最大数, 且任务队列还不空, 则加入一个新的工作线程到线程池来帮助处理还未处理完的任务。

```
private void ensureQueuedTaskHandled(Runnable command) {

    // 如果当前状态不是 RUNING, 则当前任务不加入到任务队列中, 判断如

果状态是停止, 线程数小于允许的最大数, 且任务队列还不空

    if (state < STOP &&

        poolSize < Math.max(corePoolSize, 1) &&

        !workQueue.isEmpty())

        //则加入一个新的工作线程到线程池来帮助处理还未处理完的任务

        t = addThread(null);

        if (reject)

            reject(command);

    }
}
```

7. 在前面方法中都会调用 adThread 方法创建一个工作线程, 差别是创建的有些工作线程上面关联接收到的任务 firstTask, 有些没有。该方法为当前接收到的任务 firstTask 创建 Worker, 并将 Worker 添加到作业集合 HashSet<Worker> workers 中, 并启动作业。

```
private Thread addThread(Runnable firstTask) {  
  
    //为当前接收到的任务 firstTask 创建 Worker  
  
    Worker w = new Worker(firstTask);  
  
    Thread t = threadFactory.newThread(w);  
  
    w.thread = t;  
  
    //将 Worker 添加到作业集合 HashSet<Worker> workers 中，并启动作业  
  
    workers.add(w);  
  
    t.start();  
  
    return t;  
  
}
```

至此，任务提交过程简单描述完毕，并介绍了任务提交后 ExecutorService 框架下线程池的主要应对逻辑，其实就是接收任务，根据需要创建或者维护管理线程。

维护这些工作线程干什么用？先不用看后面的代码，想想我们老大每月辛苦地把老板丰厚的薪水递到我们手里，定期还要领着大家出去 happy 下，又是定期的关心下个人生活，所有做的这些都是为什么呢？木讷的代码工不往这边使劲动脑子，但是猜还是能猜的到的，就让干活呗。本文想着重表达细节，诸如线程池里的 Worker 是怎么工作的，Task 到底是不是在这些工作线程中执行的，如何保证执行完成后，外面等待任务的老大拿到想要结果，我们将在[下篇文章](#)中详细介绍。

戏（细）说 Executor 框架线程池任务执行全过程（下）

作者 张超盟

[上一篇文章](#)中通过引入的一个例子介绍了在 Executor 框架下，提交一个任务的过程，这个过程就像我们老大的老大要找个老大来执行一个任务那样简单。并通过剖析 ExecutorService 的一种经典实现 ThreadPoolExecutor 来分析接收任务的主要逻辑，发现 ThreadPoolExecutor 的工作思路和我们带项目的老大的工作思路完全一致。在本文中我们将继续后面的步骤，着重描述下任务执行的过程和任务执行结果获取的过程。会很容易发现，这个过程我们更加熟悉，因为正是每天我们工作的过程。除了 ThreadPoolExecutor 的内部类 Worker 外，对执行内容和执行结果封装的 FutureTask 的表现是这部分着重需要了解的。

为了连贯期间，内容的编号延续上篇。

2. 任务执行

其实应该说是任务被执行，任务是宾语。动宾结构：execute the task，执行任务，无论写成英文还是中文似乎都是这样。那么主语是 who 呢？明显不是调用 submit 的那位（线程），那是哪位呢？上篇介绍 ThreadPoolExecutor 主要属性时提到其中有一个 HashSet<Worker> workers 的集合，我们有说明这里存储的就是线程池的工作队列的集合，队列的对象是 Worker 类型的工作线程，是 ThreadPoolExecutor 的一个内部类，实现了 Runnable 接口：

```
private final class Worker implements Runnable
```

- 看作业线程干什么当然是看它的 run 方法在干什么。如我们所料，作业线程就是在一直调用 getTask 方法获取任务，然后调用 runTask(task)方法执行任务。看到没有，是在 while 循环里面，就是不干完不罢休的意思！在加班干活的苦逼的朋友，有没有遇见战友的亲切感觉？

```
public void run() {  
  
    try {  
  
        Runnable task = firstTask;
```

```
//循环从线程池的任务队列获取任务

while (task != null || (task = getTask()) != null) {

    //执行任务

    runTask(task);

    task = null;

}

} finally {

    workerDone(this);

}

}
```

然后简单看下 `getTask` 和 `runTask(task)` 方法的内容。

9. `getTask` 方法是 `ThreadPoolExecutor` 提供给其内部类 `Worker` 的方法。作用就是一个，从任务队列中取任务，源源不断地输出任务。有没有想到老大手里拿的总是满满当当的 project，也是源源不断的。

```
Runnable getTask() {

    for (;;) {

        //从任务队列的头部取任务

        r = workQueue.take();

        return r;

    }

}
```

- 10) `runTask(Runnable task)` 是工作线程 `Worker` 真正处理拿到的每个具体任务。看到这里才可用确认我们的猜想，之前提到[\[y1\]](#) 的“执行任务”这个动宾结构前面的主语正是这些 `Worker` 呀。唠叨了半天（看主要方法都看到了整整第 10 个了），前面都

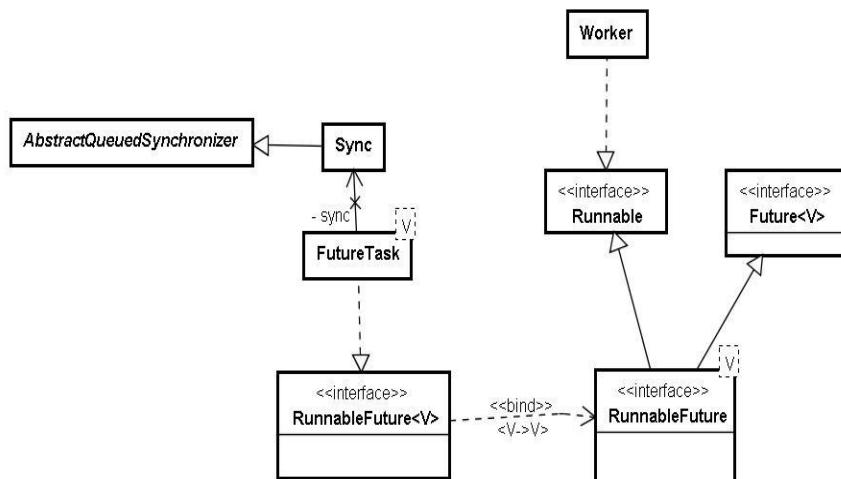
是派活，这里才是干活。和我们的工作何其相似！老大（LD），老大的老大（LD²），老大的老大（LDⁿ）非常辛苦，花了很多时间、精力在会议室、在 project 上想着怎么生成和安排任务，然而真的轮到咱哥们干活，可能花了不少时间，但看看流程就是这么简单。三个大字：“Just do it”。

```
private void runTask(Runnable task) {
    // 调用任务的 run 方法，即在 Worker 线程中执行 Task 内定义内容。
    task.run();
}
```

需要注意的地方出现了，调用的其实是 task 的 run 方法。看下 FutureTask 的 run 方法做了什么事情。

这里插入一个 FutureTask 的类图。可以看到 FutureTask 实现了 RunnableFuture 接口，所以 FutureTask 即有 Runnable 接口的 run 方法来定义任务内容，也有 Future 接口中定义的 get、cancel 等方法来控制任务执行和获取执行结果。Runnable 接口自不用说，Future 接口的伟大设计，就是使得实现该接口的对象可以阻塞线程直到任务执行完毕，也可以取消任务执行，检测任务是执行完毕还是被取消了。想想在之前我们使用 Thread.join() 或者 Thread.join(long millis) 等待任务结束是多么苦涩啊。

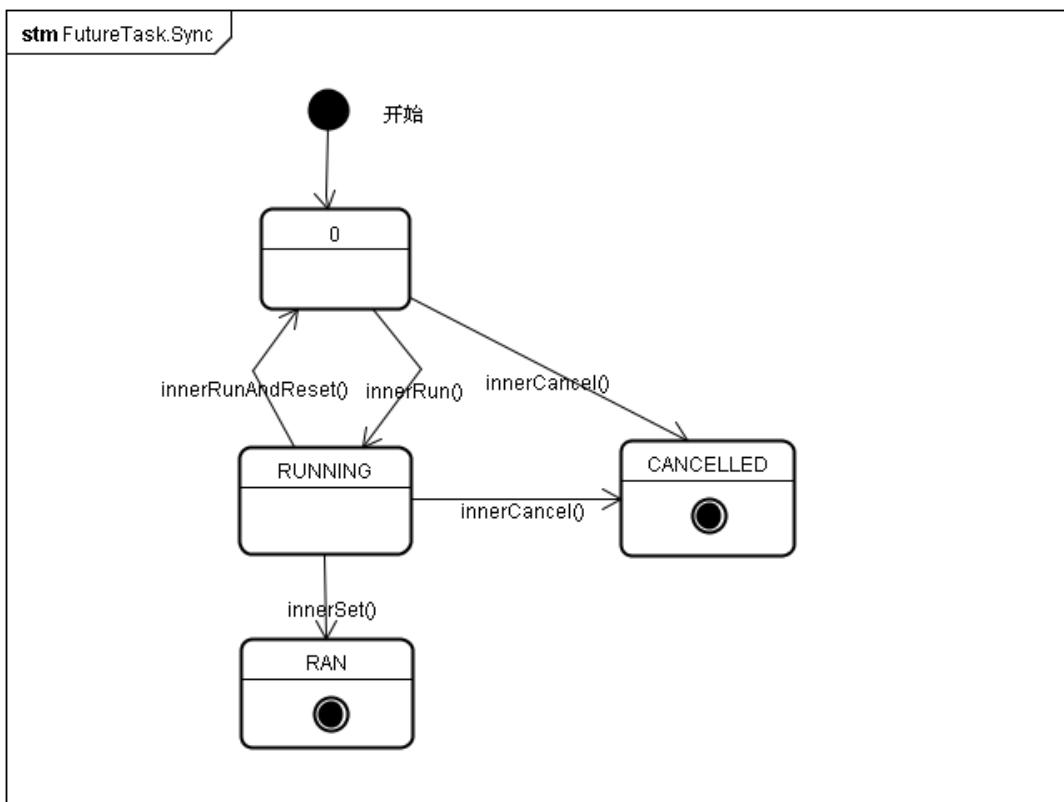
FutureTask 内部定义了一个 Sync 的内部类，继承自 AQS，来维护任务状态。关于 AQS 的设计思路，可以参照参考 Doug Lea 大师的原著 [The java.util.concurrent Synchronizer Framework](#)。



11. 和其他的同步工具类一样，FutureTask 的主要工作内容也是委托给其定义的内部类 Sync 来完成。

```
public void run() {  
  
    //调用 Sync 的对应方法  
  
    sync.innerRun();  
  
}
```

12. FutureTask.Sync.innerRun(), 这样做的目的就是为了维护任务执行的状态，只有当执行完后才能够获得任务执行结果。在该方法中，首先设置执行状态为 RUNNING 只有判断任务的状态是运行状态，才调用任务内封装的回调，并且在执行完成后设置回调的返回值到 FutureTask 的 result 变量上。在 FutureTask 中，innerRun 等每个“写”方法都会首先修改状态位，在后续会看到 innerGet 等“读”方法会先判断状态，然后才能决定后续的操作是否可以继续。下图是 FutureTask.Sync 中几个重要状态的流转情况，和其他的同步工具类一样，状态位使用的也是父类 AQS 的 state 属性。



```
void innerRun() {  
  
    //通过对 AQS 的状态位 state 的判断来判断任务的状态是运行状态，则调用任务内封装的  
    //回调，并且设置回调的返回值  
  
    if (getState() == RUNNING)  
  
        innerSet(callable.call());  
  
}  
  
  
  
  
void innerSet(V v) {  
  
    for (;;) {  
  
        int s = getState();  
  
        //设置运行状态为完成，并且把回调的执行结果设置给 result 变量  
  
        if (compareAndSetState(s, RAN)) {  
  
            result = v;  
  
            releaseShared(0);  
  
            done();  
  
            return;  
  
        }  
  
    }  
}
```

至此工作线程执行 Task 就结束了。提交的任务是由 Worker 工作线程执行，正是在该线程上调用 Task 中定义的任务内容，即封装的 Callable 回调，并设置执行结果。下面就是最重要的部分：调用者如何获取执行的结果。让你加班那么久，总得把成果交出来吧。老大在等，因为老大的老大在等！

3. 获取执行结果

前面说过，对于老大的老大这样的使用者来说，获取执行结果这个过程总是最容易的事情，只需调用 FutureTask 的 get()方法即可。该方法是在 Future 接口中就定义的。get 方法的作用就是等待执行结果。（Waits if necessary for the computation to complete, and then retrieves its result.）Future 这个接口命名得真好，虽然是在将来，但是定义有一个 get()方法，总是“可以掌控的未来，总是有收获的未来！”实现该接口的 FutureTask 也应该是这个意思，在未来要完成的任务，但是一样要有结果哦。

13. FutureTask 的 get 方法同样委托给 Sync 来执行。和该方法类似，还有一个 V
get(long timeout, TimeUnit unit)，可以配置超时时间。

```
public V get() throws InterruptedException, ExecutionException {  
  
    return sync.innerGet();  
  
}
```

14. 在 Sync 的 innerGet 方法中，调用 AQS 父类定义的获取共享锁的方法 acquireSharedInterruptibly 来等待执行完成。如果执行完成了则可以继续执行后面的代码，返回 result 结果，否则如果还未完成，则阻塞线程等待执行完成。[\[bd2\]](#) 再大的老大要想获得结果也得等老子干完了才行！可以看到调用 FutureTask 的 get 方法，进而调用到该方法的一定是想要执行结果的线程，一般应该就是提交 Task 的线程，而这个任务的执行是在 Worker 的工作线程上，通过 AQS 来保证执行完毕才能获取执行结果。该方法中 acquireSharedInterruptibly 是 AQS 父类中定义的获取共享锁的方法，但是到底满足什么条件可以成功获取共享锁，这是 Sync 的 tryAcquireShared 方法内定义的。[\[bd3\]](#) 具体说来，innerIsDone 用来判断是否执行完毕，如果执行完毕则向下执行，返回 result 即可；如果判断未完成，则调用 AQS 的 doAcquireSharedInterruptibly 来挂起当前线程，一直到满足条件。这种思路在其他的几种同步工具类 [Semaphore](#)、[CountDownLatch](#)、[ReentrantLock](#)、[ReentrantReadWriteLock](#) 也广泛使用。借助 AQS 框架，在获取锁时，先判断当前状态是否允许获取锁，若是允许则获取锁，否则获取不成功。获取不成功则会阻塞，进入阻塞队列。而释放锁时，一般会修改状态位，唤醒队列中的阻塞线程。每个同步工具类的自定义同步器都继承自 AQS 父类，是否可以获取锁根据同步类自身的功能要求覆盖 AQS 对应的 try 前缀方法，这些方法在 AQS 父类中都是只有定义没有内容。可以参照《[源码剖析 AQS 在几个同步工具类中的使用](#)》来详细了解。

突然想到想想那些被称为老大的，是不是整个 career 流程就是只干两件事情：submit a task, then wait and get the result。不对，还有一件事情，不是等待，而是催。“完了没，完了没？schedule 很紧的，抓点紧啊，要不要适当加点班啊……”

```

V innerGet() throws InterruptedException, ExecutionException {
    //获得锁，表示执行完毕，才能获得后执行结果，否则阻塞等待执行完成再获取执行结果
    acquireSharedInterruptibly(0);

    return result;
}

protected int tryAcquireShared(int ignore) {
    return innerIsDone() ? 1 : -1;
}

```

至此，获得执行结果，圆满完成任务！

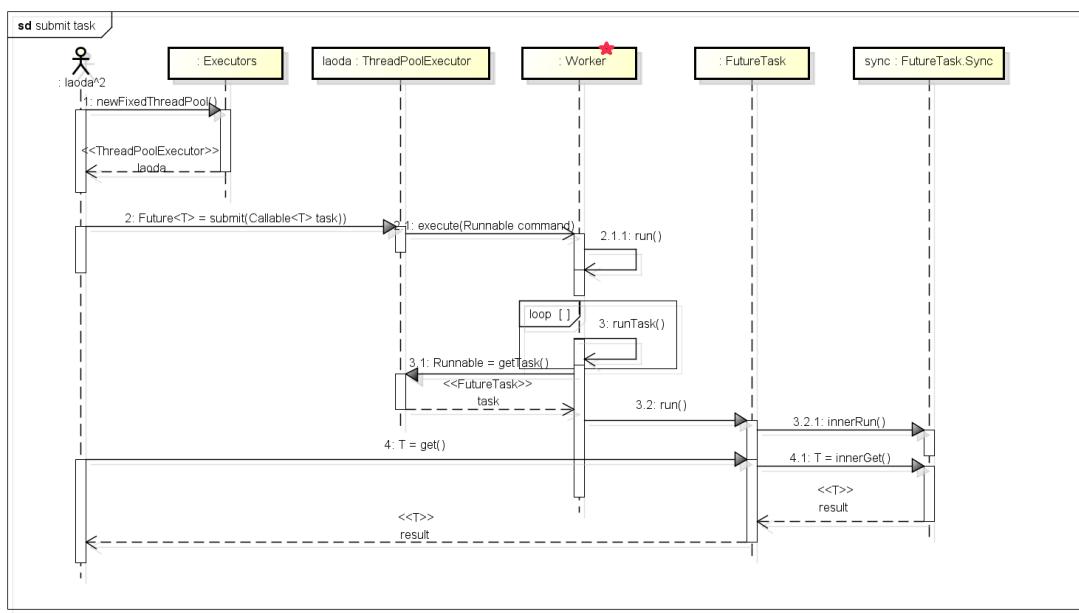
老大的老大，拍着咱们老大的肩膀（或者深情的抚摸着咱们老大唏嘘胡茬的脸庞）说：“亲，你这活干的漂亮！”而隔壁桌座位的几个兄弟，刚熬了几个晚上加班交付完这波 task 后，发现任务队列里又有新任务了，俺们老大又从他的另外一个老大手里接来的任务了。每个人都按照这样的角色进行着，依照这样的角色安排和谐愉快地进行着……

| 角色名 | 任务用户 | 任务管理者 | 任务执行者 |
|------|--|-------------------------------|--|
| 角色属性 | 任务的甲方 | 任务的乙方 | 乙方的工具 |
| 角色说明 | 选择合适的任务执行服务，如可以根据需要选择 ThreadPoolExecutor 还是 ScheduledThreadPoolExecutor，并定制 ExecutorService 的配置。 定义好任务的工作内容和结果类型，提交任务，等待任务的执行结果 | 接收提交的任务；维护执行服务内部管理；配置工作线程执行任务 | 每个工作线程一直从任务执行服务获取待执行的任务，保证任务完成后返回执行结果。 |

| | | | |
|---------------------|---|--|---|
| Executor 中对应 | 创建获取 ExecutorService、并提交 Task 的外部接口 | ExecutorService 的各种实现。如经典的 ThreadPoolExecutor, ScheduledThreadPoolExecutor | 执行服务内定义的配套的 Worker 线程。如 ThreadPoolExecutor.Worker |
| 主要接口方法 | submit(Callable task) | execute(Runnable command) | runTask(Runnable task) |
| 现实角色映射 | 手里有活的大老大 | 领人干活的老大 | 真正干活的码农 |
| 主要工作伪代码 | taskService = createService() future=taskService.submit Task() future.get() | executeTask() { addTask() createThread() } | while(true) { getTask() runTask() } |

四、总结

从时序图上看主要的几个角色是这样配合完成任务提交、任务执行、获取执行结果这几个步骤的。



1. 外面需要提交任务的角色（如例子中老大的老大），首先创建一个任务执行服务 ExecutorService，一般使用工具类 Executors 的若干个工厂方法 创建不同特征的线程池 ThreadPoolExecutor，例子中是使用 newFixedThreadPool 方法创建有 n 个固定工作线程的线程池。
2. 线程池是专门负责从外面接活的老大。把任务封装成一个 FutureTask 对象，并根据输入定义好要获得结果的类型，就可以 submit 任务了。
3. 线程池就像我们团队里管人管项目的老大，各个都有一套娴熟、有效的办法来对付输入的任务和手下干活的兄弟一样，内部有一套比较完整、细致的任务管理办法，工作线程管理办法，以便应付输入的任务。这些逻辑全部在其 execute 方法中体现。
4. 线程池接收输入的 task，根据需要创建工作线程，启动工作线程来执行 task。
5. 工作线程在其 run 方法中一直循环，从线程池领取可以执行的 task，调用 task 的 run 方法执行 task 内定义的任务。
6. FutureTask 的 run 方法中调用其内部类 Sync 的 innerRun 方法来执行封装的具体任务，并把任务的执行结果返回给 FutureTask 的 result 变量。
7. 当提及任务的角色调用 FutureTask 的 get 方法获取执行结果时，Sync 的 innerGet 方法被调用。根据任务的执行状态判断，任务执行完毕则返回执行结果；未执行完毕则等待。

还记得我们费了半天劲试图找出任务执行时那个动宾结构的主语吗？从示例上看更像是线程池在向外提供任务执行的服务。就像我们的老大在代表我们接收任务、执行任务、提交执行结果。明显我们这些真正的 Worker 成了延伸，有点搞不懂到底我们是主语，还是主语延伸的工具，就像定义 ThreadPoolExecutor 的内部类 Worker 一样。我们只是工具，不是主语，是状语： execute the task by workers。突然想到毛主席当年的“数风流人物，还看今朝”，说的应该是这些 Worker 的劳苦大众吧，怎么都今朝这么久了，俺们这些 Woker 们还是风流不起来呢？风骚的作者居然在上面严肃的时序图上加了个风骚的小星星，向同行的 Worker 们致敬！

作者简介

张超盟，an ExTrender，CS 数据管理方向工学硕士。与妻儿蜗居于钱江畔，就职一初创安全公司任数据服务团队负责人，做数据(存储、挖掘、服务)方面研发。爱数据，爱代码，爱技术，爱豆吧！(idouba.net)。

Java 20 年：转角遇到 Go

作者 郭蕾

1995 年，横空出世的 Java 语言以其颠覆式的特性迅速获得了开发者的关注。跨平台、垃圾回收、面向对象，这在当时都是不可思议的事情，而 Java 却完美地在一门语言中实现了这一特性。可以说，Java 将编程语言设计带领到一个新的高度。20 年后的今天，当年的那些新特性已经不再是什么新鲜词。同时，又会有一些新的语言宣称自己有一些颠覆性的特性，其中 Go 语言就是新语言的一个代表，它部署简单、并发性好，在语言设计上确实优于 Java。为了了解 Java 和 Go 语言的发展现状与趋势，InfoQ 采访了 Go 语言大牛郝林。

InfoQ：今年的 5 月 23 日是 Java 的第 20 岁生日，转眼间，Java 已经走过了 20 年，版本号也已经更新到 Java 8。你怎么看 Java 这门语言？在这 20 年里，有哪些对你印象比较深刻的 Java 事件？

郝林：我觉得 Java 语言一路走来赚足了眼球也惹来了众多非议。就拿它随着 Sun 公司的没落被流转到 Oracle 公司来说吧。我记得当时有一大批 Java 程序员在网上扬言要摒弃 Java 语言，并且一部分人真的这么做了。但事实证明，Oracle 更好地发展了 Java。我认为从 Java 7 开始这门语言相当于迎来了第二春，在发展上增速了不少，各种新鲜特性和类库层出不穷。Java 8 给我印象最深刻的就是对 Lambda 表达式的支持。这使得 Java 真正地对函数式编程提供了支持。这是质的改变。也终将使 Java 语言走得更远。

InfoQ：从版本迭代的角度看，你认为 Java 的发展经历了哪几个阶段？

郝林：我是从 Java 1.3 的末期开始接触它的。所以在我看来 Java 1.3 之前就属于萌芽期吧（虽然那时它已被广泛使用了）。从 1.4 开始，Java 语言有了很多改观，比如 NIO、更多的垃圾回收器、性能上的提升、Java EE 规范的逐步简化，等等。所以我认为从此 Java 进入了第一个高速发展期（也许有上一个但我没赶上）。到了 Java 6 的时候，发展速度其实已经减缓不少了。这也可能是由于 Java 正处于被交接阶段的缘故。不过，我不得不说，Oracle 的调整动作很快，在几乎没有断档的情况下，Java 的发展又开始“跑”起来了。这也是我在前一个回答中说的“第二春”。

InfoQ：JVM 的普及促使相关周边语言不断涌现，你怎么看这些 JVM 语言？

郝林：这就是 Java 真正牛的地方。它不单单是一门语言，更是一个平台。到目前为止，JVM 语言已经有很多了，但是发展最好的是 Scala。它解决了一些 Java 在程序开发方面的问题。但是，我认为它的方向有所偏颇。我觉得“简化”往往比“丰富”来得更直接，效果也会更好。相比之下，Clojure 语言就做得很好。但是由于它是一个 Lisp 语言的方言，编码方式和思维方式与 Java 的面向对象思想相去甚远，所以仅仅被一小部分 Java 程序员接受。总之，JVM 语言让 Java 更加流行了。它们虽不完美，但却功不可没。

InfoQ: 很多人都在唱衰 Java，您能结合 Java 的发展现状和趋势谈谈 Java 的前景吗？

郝林: 任何一个流行的技术都会有人唱衰，更何况 Java 已经发展了 20 年了，中间又经历了种种坎坷。我觉得 Java 9 又会是一个里程碑式的版本。我很期待。我认为在我可预见的未来 Java 不会没落。实际上，Java 语言在企业级软件领域的霸主地位是不可动摇的。在互联网软件领域，它虽然受到了各种开发成本更低的语言（比如 Ruby 和 Python）的不断侵蚀，但是仍然占有一席之地。这正说明了 Java 生命力的顽强。不过，相比于 Java 语言，我更看好 Java 作为一个平台的前景。

InfoQ: 你什么时候开始接触 Go 语言的？相比于 Java 语言，它有哪些优势？

郝林: 我接触 Go 语言实际上并不算早，大约在 2013 年的上半年。那时候 Go 语言的版本是 1.0，1.1 版本正处于开发期。Go 语言给我的第一印象就是支持多种编程范式、提供了给力的程序构建和发布工具，以及在并发编程方面的极度简化。在当时，我认为 Java 语言的不足恰恰就包括了这几个方面。所以我义无反顾的开始学习并使用 Go 语言。事实证明，Go 语言虽属于新兴语言，但它却是一种革新。另外，与 Java 语言一样，Go 语言的向后兼容做的很好。并且，为了以防万一，它提供了一个命令用于自动地把旧版本的 Go 语言程序源码调整为当前版本的源码。诸如此类的“便捷大法”还有很多。许多在 Java 世界中只能依靠额外的类库或工具才能完成的事情，在 Go 语言看来却是手到擒来。当然，这种实实在在的优势也有诞生时间不同的缘故。正是由于 Java 已历经了太多，所以在很多方面都很难改变。我觉得这是所有编程语言都应该正视的问题。显然，Go 语言的创造者们已经意识到了这一点。

InfoQ: 出色的并发性能是 Go 语言区别于其他语言的一大特色。相比于 Java 的并发编程，它有哪些显著性的优势？

郝林: 说到并发，Go 语言给人们的第一印象就是便捷。在这便捷之下，Go 语言权衡了各方面利弊，做了大量的工作，使得我们用极低的开发成本就可以编写出拥有超高运行性能的 Go 语言并发程序。其中最大的亮点就是，Go 语言把“激活”需要并发执行的代码块的操作内置了。我们仅通过一个关键字“go”就可以轻易地完成这项操作。

还记得我们在 Java 中为此需要编写的代码是多么的冗长吗？侵入式的接口实现声明和类继承声明、复杂的匿名内部类，以及困难重重的线程间协调和调度。这些都是不可忽视的程序开发维护成本。我们在编写和修改这样的并发程序时都要保持头脑和思路的绝对清晰，否则就会埋下祸根，搞出不易察觉和定位的 Bug。另一方面，如果透过表象看本质的话，我们就可以看到 Go 语言为了程序员的方便而做的大量的工作。

笼统地讲，Go 语言把对内核线程的使用和调度操作都内置到其运行时系统中了。但是，它远远要比一个线程池复杂得多。Java 线程与内核线程之间关系是 1:1 的。而 Go 语言的 Goroutine（可以看做是 Go 语言中执行并发代码块的实体）与内核线

程之间的关系是 M:N 的。这让我们可以使用成千上万个 Goroutine 去执行并发代码块而仅仅耗费极少的内核线程。关于 Go 并发编程更详细的介绍，大家可以参看我著的“图灵原创”图书《Go 并发编程实战》。

InfoQ：Java 和 Go 语言的使用场景是不是不一样？

郝林：Java 语言与 Go 语言在使用场景方面其实有很多相似之处。例如，它们都适用于服务端程序的构建，并且可以很容易地编写出页面模板文件。又例如，它们在桌面软件方面都比较捉襟见肘。有意思的是，就本身而言，Go 语言在适用领域的优势更强，而在不适用领域的劣势也更加明显。优势方面我就不再赘述了，下面说说劣势。比如，用 Java 编写桌面程序起码还有 Swing 和 JavaFX 可选，但是 Go 语言官方至今还没有一个成熟的解决方案。当然，这仍旧与诞生时间有关。另外，我们还可以用 Java 语言编写 Android 应用程序。Go 语言目前虽然已经涉足，但还不完美。不过我在这里爆料一下，我很期待能用 Go 语言编写 iOS 应用程序。实际上，Go 语言在这方面已经有所进展了。总之，两种语言在适用领域方面有所重叠但又有些不同。在很多情况下，我们可以混用这两种语言。

InfoQ：现在的开发语言特别多，Java、Go、PHP、Rust、Python 等，你认为未来语言的发展趋势是怎么样的？

郝林：的确，现在的编程语言层出不穷、多如牛毛。但是编程语言的兴衰是有规律可循的。第一个规律是顺应时代的语言才能有更好的发展。正如 Objective-C 因 iPhone 和 iPad 的诞生而变得火热至极那样。而 Java 也因 Google 公司的“横插一足”而在移动程序开发领域占领了制高点。当今的计算机世界正处于“云”的时代，而从处理器的角度看也正处于多核时代。谁能够更好地把握住这些时代标签，谁就会在发展上更具优势。当然，这里说的“把握住”是需要有真功夫的。只喊不练不起任何作用，而且还会遭人唾弃。第二个规律是能够解决问题的语言就是好语言。对于任何场景都是如此。我相信每个技术团队都会在选择编程语言时进行一番权衡。哪种编程语言能更快更好地解决问题（这也涉及到开发和维护成本），它就肯定会胜出。从这方面看，编程语言并没有好坏之分。它们都必有独特的优势和擅长做的事情，否则就根本不会诞生出来了。而问题的解决能力几乎是发展趋势的唯一评判标准。“多快好省”就是选择编程语言的要诀。这也会从侧面预示一个编程语言的发展趋势。说了这么多，我另一个想要表达的意思是：对于它们的未来，我无法预知。

受访嘉宾介绍

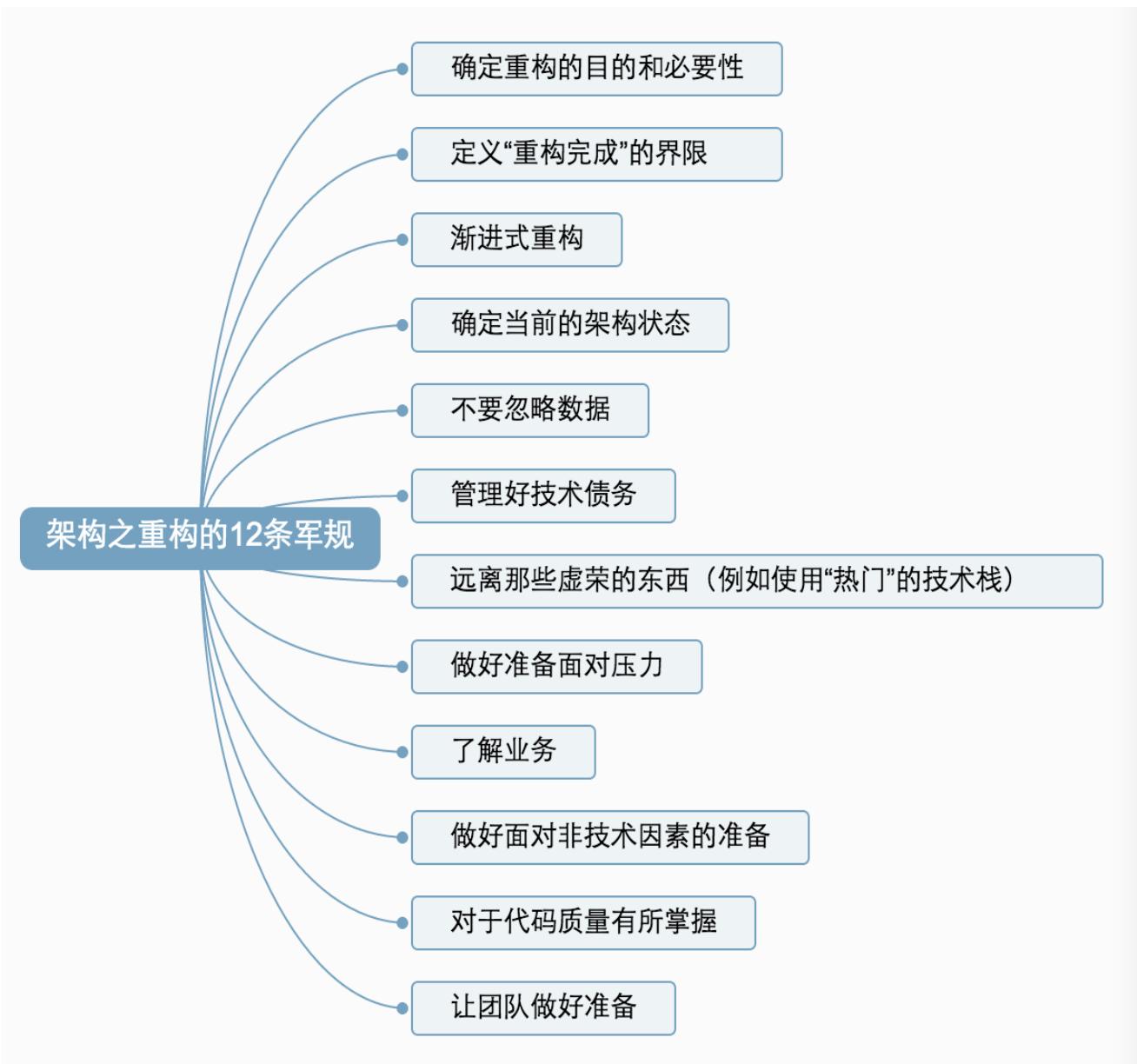
郝林，软件工程师，从事软件开发工作 9 年有余。既搞过企业级软件项目，也堆过互联网软件系统。近期在使用和推广 Go 语言，著有“图灵原创”图书《Go 并发编程实战》，以及在线免费教程《Go 语言第一课》和《Go 命令教程》。

架构之重构的 12 条军规

作者 崔康

对于开发者来说，架构设计是软件研发过程中最重要的一环，所谓没有图纸，就建不了房子。在遍地 App 的互联网时代，架构设计有了一些比较成熟的模式，开发者和架构师也可以经常借鉴。

但是，随着应用的不断发展，最初的架构往往面临着各种问题，比如无法满足客户的需求、无法实现应用的扩展、无法实现新的特性等等。在这种情况下，我们如何避免一些坑，尽量比较成功地实现架构的重构，是很多开发者和架构师亟需解决的问题。



在这里，跟大家分享一下 Uber 的工程主管 Raffi Krikorian 的 12 条规则，并附上一些解读，希望对大家有所启发。

确定重构的目的和必要性

看起来这个规矩有些多余，但是请不要忽略。每一次架构的重构都是“伤筋动骨”，就像做手术一样，即使再成功，也会伤元气，所以决策者们首先要分析架构重构的理由和其他备选方案，明确重构的目的是为了满足业务需求，并且是不得不做的最佳方案，然后再考虑其他问题。有时候，经过分析就会发现，也许还有其他解决方案，比如增加计算资源，或者重构的目的不是为了业务需求，那就没有必要做了。

检查清单：

- 架构重构的原因是什么，是为了满足业务的需要还是只是觉得架构不好看？
- 除了架构重构之外，还有其他备选方案吗？是否都分析过这些方案的利弊？

定义“重构完成”的界限

如果确定要重构，那么要把目标明确下来，也就是重构的边界条件，怎么才算是“完成”了重构，目标要有数据量化，或者有能够测试的办法。这也是一个需求分析的过程，如果需求不明确，那么规格说明书没法写清楚，负责重构的团队也没有明确的目标，不能以重构的时间或者主观的判断为结束的依据。前几天和一朋友聊天，他最近在负责系统的性能优化，也要做一些重构的事情，开始的时候团队的目标不明确，大家不知道优化到什么程度，所以不敢下手。如果目标是提高 10%，那么可以从细节处着手；如果是提高 50%，那可能要搞大动作才能实现了。后来目标明确之后，团队才找到合适的办法。

检查清单：

- 重构的目标可以量化，或者说可以测试吗？
- 重构完成的标准是什么？得到业务部门或者领导的认可了吗？

渐进式重构

现在软件研发最流行的就是快速迭代、持续交付、尽早反馈。这同样可以用在架构的重构上，重构过程的难度不亚于构建一个新产品，所以在设计重构的时候，要引入持续交付的流程，每一个重构步骤或者模块都要快速部署并得到反馈，以便评估重构的效果，及时作出策略调整。有的读者会说，我们的架构重构是釜底抽薪型的，没法渐进，只能一蹴而就。如果是这种情况，可以考虑在另外一套拷贝的系统中做重构，经过谨慎测试之后，将数据和业务迁移过去。

检查清单：

- 能否把重构过程分成小的迭代，每一次改进都能尽快得到反馈？
- 重构过程中的效果能够定期展示给业务部门或者领导吗？

确定当前的架构状态

在启动重构之前，团队要对当前的架构状态有清晰的了解，也就是设定好基准，以便评估重构的效果。据我的经验，负责重构的架构师或者开发者，往往还没有搞清楚现有的架构设计，就开始重构了，结果经常出现这样的情况：重构到某个阶段，发现行不通，然后一拍脑袋说，哦，原来这块的架构是这个样的，是为了达到某某业务需求啊，这块不能动，得想别的办法。类似的例子在研发团队中时有发生，也提醒我们要慎重小心。记得有位哲人说过，了解别人很容易，了解自己很难。

检查清单：

- 你了解当前的架构设计吗？它的设计初衷和之前的选型方案知道吗？
- 你能给架构设定一个基准状态吗？

不要忽略数据

数据的重要性不言而喻，业务都是以数据流为载体的，所以架构重构的本质就是对于数据流的重构。数据对重构的重要性主要体现在两个方面：在重构设计时，需要考虑业务数据的需求，重构之后的系统对于数据的存储、处理、分析等功能是否有影响；在重构过程中，考虑依靠数据甚至是实际的数据来验证重构的效果，提供评估的支持。

检查清单：

- 业务数据的需求在重构设计中有体现吗？
- 重构过程中能否通过实际数据来验证效果？

管理好技术债务

技术债务在平常的软件研发过程中也是比较突出的问题，现在单独拿出来强调是希望提醒开发者们：架构重构往往是为了偿还技术债务，所以请不要在偿还技术债务的过程中制造技术债务了。技术债务就像信用卡一样，会有很高的利息率，就如同给团队留下了大量的帐务开销。组织应该培养一种保证设计质量的文化。应当鼓励重构、同时也应当鼓励持续设计以及其它有关代码质量的实践。在开发时间中应当专门抽出一部分以解决技术债

务。如果没有合适的照料，那么真实世界中的代码会变得越来越复杂难懂。

检查清单：

- 团队对技术债务有跟踪和备忘录机制吗？还是开发人员可以随意的产生债务？
- 针对技术债务有定期的培训、回顾机制吗？

远离那些虚荣的东西（例如使用“热门”的技术栈）

架构的重构过程应该是以目标为导向，换句话说“注重实效”。对于技术人员来说，一个经常被轻视的问题在于，喜欢追逐新鲜的热门技术，这其实是个好事情，说明技术人员勇于创新，不断接受新技术。但是对于架构的重构这样的关键性任务来说，是不是新技术并不重要，重要的是能不能实现重构的目标。对于新技术来说，虽然热度大，但是人才储备还不足，大家踩过的坑还不多，积累的失败教训和成功经验还不够，在这种情况下，建议大家不要头脑一热就上马新技术，应该客观冷静地评估新技术和成熟技术对架构重构的影响和效果，以数据和经验来说话，而不要追赶时髦。

检查清单：

- 重构的技术选型是否有详实的数据和专家评估？
- 选用的技术是否有良好的人才积累和足够的经验支持？你是不是实验小白鼠？
- 在技术选型时，是否至少有两个方案待评估？有没有成熟的技术方案？

做好准备面对压力

这条军规更像是对架构师们的心理建议，软件开发过程中，压力无处不在。对于架构重构来说，压力来源于多个方面：管理层、团队成员、同级部门等等。说白了，架构重构对个人来说往往是一件出力不讨好的事情。和做一个新产品能够取得很高的赞赏相比，重构的成绩往往并不受领导重视，而且出了问题还要承担很大的责任。从软件开发角度看，做新产品是从 0 到 1，而架构重构是从 -1 到 1，复杂性和难度通常更大。因此，重构的负责人要提前做好心理准备，舒缓压力的一个技巧是，设置好里程碑，将重构的成果量化，并且和业务的变化关联起来，定期向利益相关各方同步状态，得到大家的理解和支持。

检查清单：

- 架构的重构是否得到了管理层（特别是最高管理层）的支持？他们是否对重构的时间、任务量有直接的认识？

- 你的重构计划中是否包含了一些可以量化的成果？是否定期向管理层展示这些成果？

了解业务

虽然看起来像是一句废话，但是我想 Raffi Krikorian 特意把这条提出来一定是有理由的。架构重构的最终目的是改进业务，所以对于业务的了解将有助于架构师和技术人确定重构目标的优先级和关键路径。比如，我们需要知道哪些关键业务的架构是不能碰的，哪些业务之间是互相关联的，哪些业务的架构是需要优先重构的.....等等。除了了解业务本身，我们还需要了解“人”，表面上管理层是重构目标的裁决者，但实际上业务部门的人才是。技术人需要了解他们的业务需求，并将其转化为重构目标。通过这种方式，架构重构的意义才能得到具体的体现。

检查清单：

- 是否与业务部门就架构重构所能实现的业务目标进行过充分的讨论和确认？
- 是否对关键业务和优先重构的业务进行了确认？

做好面对非技术因素的准备

恩.....这又是一个不那么让人舒服的建议。不管你是否愿意相信，技术在架构重构（以及其他很关键的公司决策中）的影响因素中并不是最高的，我们还会涉及到商业利益、管理层偏好、大客户影响、办公室 zhengzhi、站队问题等等，对于架构师和技术人来说，这些因素往往不是他们所能掌控的。我们能做的就是，与利益相关者设定重构目标，然后，根据不同的影响因素，调整目标。请记住，不要死扛这个目标，当有人提出不同的意见时，要坦诚地和他们交流，并告知他们如何采纳意见，那么重构目标会有变化，然后让其他利益相关者也知道这些变化。非技术因素的影响是客观存在的，而且从商业层面来说也是合理的，所以对于技术人来说要学会适应。

检查清单：

- 当非技术因素影响架构的重构时，你是否对目标做了调整并告知了利益相关各方？
- 你是否准备以开放而不是抵制的心态来对待非技术因素的影响？

对于代码质量有所掌握

这和上篇中所提到的“管理好技术债务”有异曲同工之处。架构的重构对代码质量要求很高，一方面是重构过程对 bug 的容忍性比新产品的研发更低，另一方面也决定了下一次重构的难易程度。关于代码质量的书籍和文章已经有很多，在这里只想提醒大家一点：代

码审查是一个非常好的办法。代码审查是软件开发过程中的必要步骤，既可以帮助被审查者提到代码质量，又可以让审查者加深对产品的理解。不论团队多忙，一定要保证代码提交之前，是经过其他成员审核过的，短期来看会占用团队的时间，长期来看是事半功倍的好事。

检查清单：

- 团队成员是否对代码质量有足够的重视？是否有奖惩措施？
- 团队内部是否有代码质量的标准文档和审查流程？

让团队做好准备

这是 Raffi Krikorian 列举的最后一条军规，是对之前所有建议的总结，我在这里不做解读了，请大家自我感觉吧。

结尾

关于架构的重构，Raffi Krikorian 给了很好的建议，不过到底有没有效果，还是要实践中检验。尽信书不如无书，来源于实践中的经验是最有价值的经验，为技术人所用才有意义。

封面植物

竹



原产自中国。品种繁多，多年生禾本科竹亚科植物，茎为木质，是禾本科的一个分支，学名 *Bambusoideae* (*Bambusaceae* 或 *Bamboo*)，分布在热带、亚热带地区，东亚、东南亚和印度洋及太平洋岛屿上分布最集中，种类很多，有的低矮似草，有的高如大树，生长迅速。通常通过地下匍匐的根茎成片生长，也可以通过开花结籽繁衍，种子被称为竹米。有些种类的竹笋可以食用。竹枝杆挺拔，修长，四季青翠，凌霜傲雨，倍受中国人喜爱，与梅、兰、菊并称为四君子，与梅、松并称为岁寒三友，古今文人墨客，爱竹咏竹者众多。

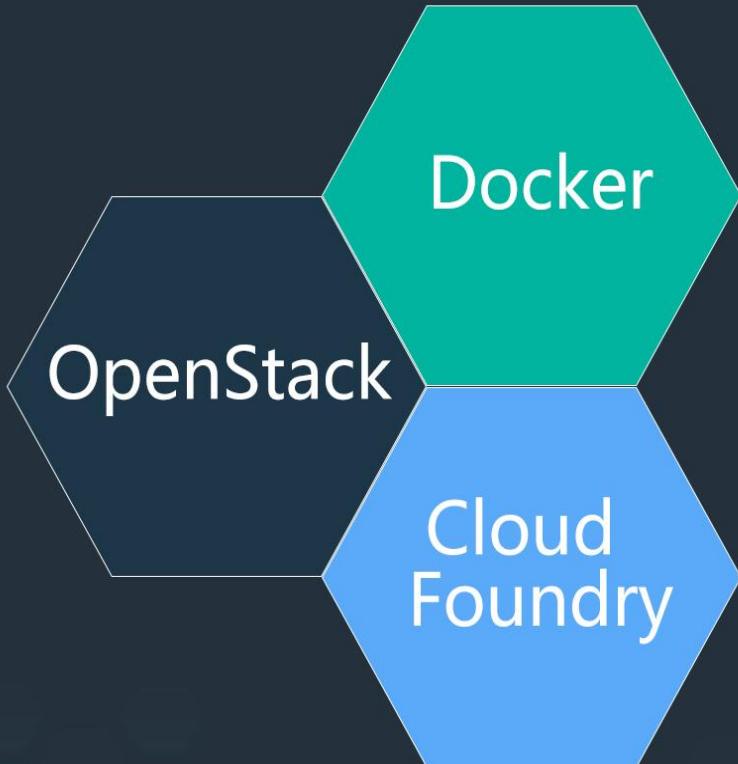
竹是高大乔木状禾草类植物。记载有 70 余属，1000 多种，但其中许多是同物异名。竹为高大、生长迅速的禾草类植物，茎为木质。分布于热带、亚热带至暖温带地区。东亚、东南亚和印度洋及太平洋岛屿上分布最集中，种类也最多。竹的地上茎木质而中空（我们称为竹杆），是从竹的地下茎（根状茎）成簇状生出来的。竹的分类体系不完善。由于生长特性，竹并不经常开花，外形又非常相似，所以现行竹的分类是根据竹笋外包的箬壳来区别分类的。但是箬壳的性状并不稳定，所以，在许多时候会引致歧义，使一些竹种群之间的界限无法唯一性的定论。



IBM Bluemix

下一代数字创新平台

用你想要的方式开发应用





有云™
世界更精彩

www.unitedstack.com

这是一个天翻地覆的时代
每一个梦想都有闪光的机会
我们,是云时代的创新者
两年的奋斗时光
回首
我们欣慰已完美踏出第一步
今天的**UnitedStack**日新月异
一个全新的我,
期待一个全新的你



UnitedStack 有云
openstack [cloud] services



架构师 2015 年 6 月刊

每月 8 号出版

本期主编：魏星

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.com

InfoQ 中文站新浪微博：<http://weibo.com/infoqchina>

商务合作：sales@cn.infoq.com 15810407783



本期主编

魏星，InfoQ 技术编辑，热爱互联网安全与开源，信奉技术的力量。致力于做一些有价值的事，并且越来越慢。人生理想：壕，不犹豫。