

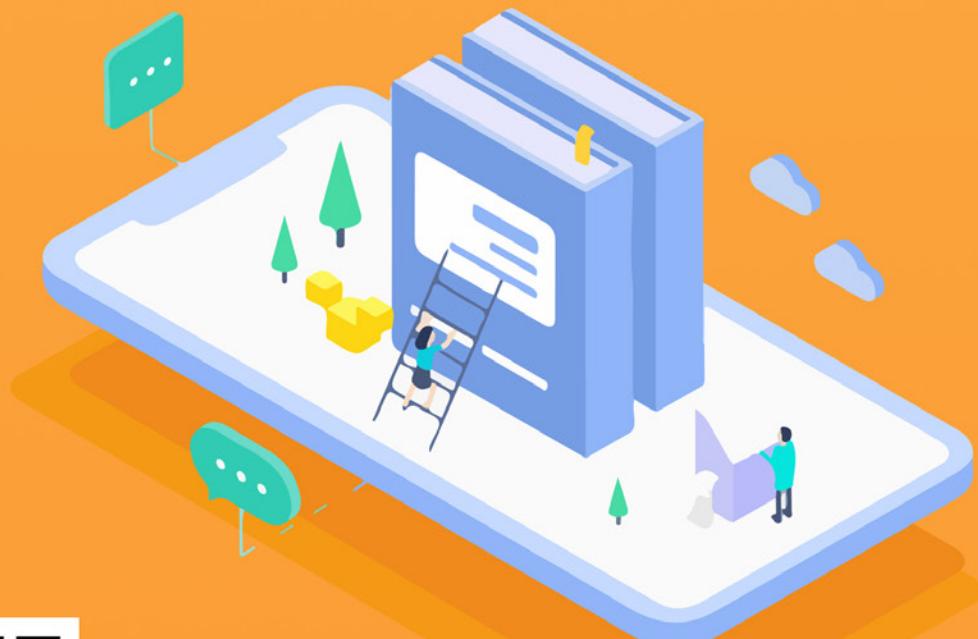
架构师

ARCHITECT

6月刊 |

热点

Google力推的那些前端技术，
最近有何进展？



CONTENTS / 目录

热点 | Hot

Google 力推的那些前端技术，最近有何进展？

理论派 | Theory

如何做好文本关键词提取？从达观数据应用的三种算法说起

推荐文章 | Article

优秀架构师必须掌握的架构思维

观点 | Opinion

区块链没有未来，是时候抛弃它了

专题 | Topic

一个可供中小团队参考的微服务架构技术栈

特别专栏 | Column

实现生产级的 Migrate 操作



架构师
2018 年 6 月刊

本期主编 徐川

提供反馈 feedback@cn.infoq.com

流程编辑 丁晓昀

商务合作 hezuo@geekbang.org

发行人 霍泰稳

内容合作 editors@cn.infoq.com

卷首语

学不动了

作者 徐川

最近，前端圈见证了一场“落伍焦虑”的集中爆发。

Node 之父 Ryan Dahl 在 Github 新建了一个项目 Deno，意在打造一个更好的、更安全的 TypeScript 运行时。本来到这里画风还好好的，前端圈三天两头的造轮子，不单只有这一个。

然而，随后画风突变，一个 Github 用户在 Deno 的 issue 区留言说：“求你别更新了，老子学不动了”，然后 issue 区被中文垃圾留言所淹没，逼得项目的管理者只能暂时限制非相关者的留言。

是的，在所有互联网技术里，前端技术的更新迭代是最快的，所以有这样的焦虑，造成这样的爆发，虽在意料之外，也在情理之中。

但是对不起，学不动了，还要学。

现在的时代本来就是终身学习的时代，不光是领域知识，连常识都在不断更新，你不去更新自己的知识，那只有落伍。

不过，学习有技巧，可以学的快一些，学的轻松一些。

这个技巧指的是两方面：

- 一个是学会学习，你去网上搜，教人学习的书有一大把。当然每个人都有自己的学习习惯，你需要从别人的经验中选择适合自己的。
- 另一个是掌握核心知识，每个领域的核心知识没有那么容易被替代，虽然很多人都认为jQuery落伍了，但用它的人照样还有很多，有些几十年前的编程语言也仍有自己的用武之地。掌握了核心知识以及一两个框架，你可以不学习。当然，前提是你能克服焦虑。

焦虑是这个时代的常态，学会与它为伍，这是你第一个也是终身都需要去学习的。

区块链技术与落地场景 大比拼

核心技术

分布式账本、共识算法、P2P网络等技术如何演进和创新？

区块链金融

金融被视为区块链最重要的应用场景，如今又有哪些技术应用和实践？

智能合约

基于智能合约我们能做哪些事情？如何开发智能合约的Dapp应用？

区块链安全

如何利用区块链解决一些传统和伴随新兴技术出现的安全问题？

区块链游戏

如何将区块链应用到游戏当中，如何设计和开发基于区块链的游戏？

区块链即服务

分布式账本、共识算法、P2P网络等技术如何演进和创新？

8折优惠限时抢购

抢票热线：13269076283
商务合作：13426412029

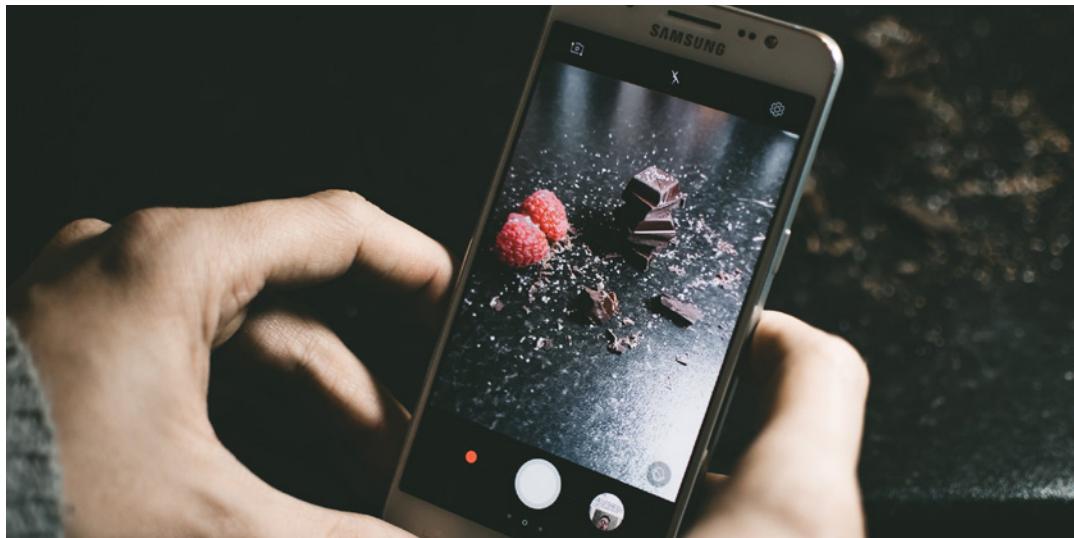
千位区块链工程师在北京等你！

扫码二维码，立即报名



Google 力推的那些前端技术，最近有何进展？

编译 覃云



Google I/O 2018 已于上周落下帷幕，普通民众看的是新产品，开发者们关注的是新技术。透过这次大会，我们不难发现，Google 已经从 mobile first 转向 AI first，AI 之后，就是移动和前端技术了，移动无非是 Android P 和 Flutter 等，前端涵盖的技术从 Web 框架到 Web 工具，包括 Angular、PWA、polymer、AMP 等，下面让我来为大家捋一捋 Google 力推的这些前端技术最近都有哪些进展。

Angular

Angular 是前端三大框架之一，它与 React、Vue 的“争斗”一直都没有

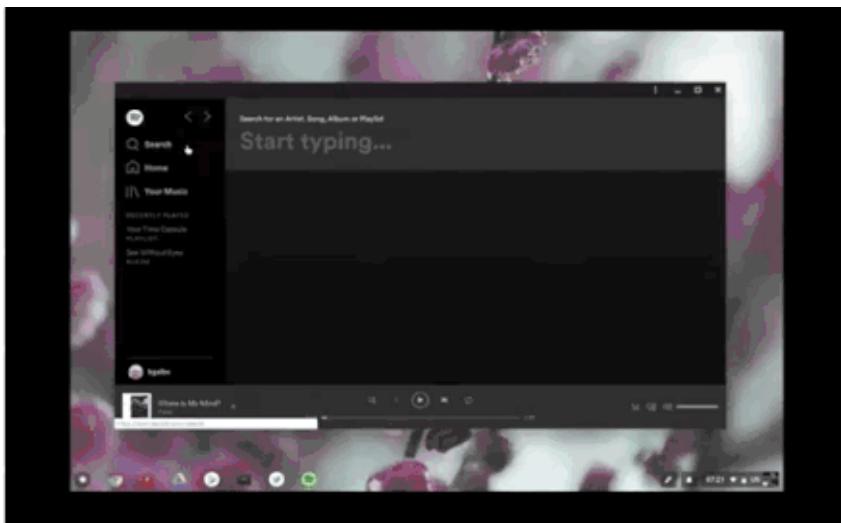
停止过，经过 Angular 团队和开发者们的努力，Angular 已经建立了拥有上百万开发者规模的社区和生态系统。在今年 I/O 大会开始的前几天，Angular 6 正式发布，添加了 ng update 和 ng add 这样的新功能，让你的应用程序保持最新的状态，帮助 Angular 开发者加快创新的步伐。

Angular 渲染器 Project Ivy 也有很大的改进，它能使 Angular 调试更容易，以更快地速度编译和运行，它还可以与现有的应用程序一起使用，Angular 团队还在小型 Hello World 应用程序做了演示，不使用的 Angular 功能将自动从应用的 JavaScript bundle 自动删除。

PWA

PWA 应该是这两年前端最火的技术之一了吧。Google 声称世界各地各行业在 PWA 的构建上都获得了很大的成功，星巴克在推出 PWA 网站后，日活跃用户数量增加了 2 倍，他们对广告网站进行测试，发现当一个网站切换到 PWA 时，平均转化率提高了 20%。

PWA近期动态：



- 早期的PWA主要专注于移动设备，但是由于平台限制，iOS无法支持，但是今年年初，终于迎来了好消息，苹果宣布iOS 11.3将迎

来PWA，这使得开发跨平台的PWA成为了可能。

- 2月，微软着手在 Windows 商店中增加PWA，宣布渐进增强式 Web应用将在 Win10 系统上线。
- 2月底，PC端的Chrome正式支持PWA，具体实现方式可参考本篇文章。
- I/O大会上，Google宣布今年6月，Chrome 67将支持PWA“安装”到桌面上，同时保留在浏览器中查询的功能，如在页面中查找、共享网址、Google Cast支持等，下图为Spotify部署桌面PWA后的部分体验。

Service Worker

Service Worker 是近年来 Web 最大的改进之一，它是 Chrome 团队力推的一个 Web API，它将开发人员从页面的生命周期中解放出来，运行于浏览器后台，可以控制打开作用范围下的所有页面请求，使 Web 应用程序能够脱机工作。

今年三月，苹果宣布 iOS 和 MacOS 上的 Safari 11.1 支持 Service Worker，4 月底，微软也宣布 Microsoft Edge 也将支持 Service Worker，这意味着现在所有的主流浏览器都支持 Service Worker 了。

为了使用过程更简便，Service Worker 开发团队创建了 Workbox 库，它能将许多常用的、强大的 service worker 模式封装到易于使用的 API 中。

Workbox: <https://developers.google.com/web/tools/workbox/>

WebAssembly

WebAssembly 使网站能够运行用 C 或 C ++ 等语言编写的高性能低级代码，为 Web 打开了新世界，今年 3 月，来自 Autodesk 的 AutoCAD 就采用了 35 年前的代码库，并用 WebAssembly 编译让其直接在浏览器中运行，这意味着，无论你的设备或操作系统如何，你都可以直接在浏览器中用 CAD 绘图。

Polymer

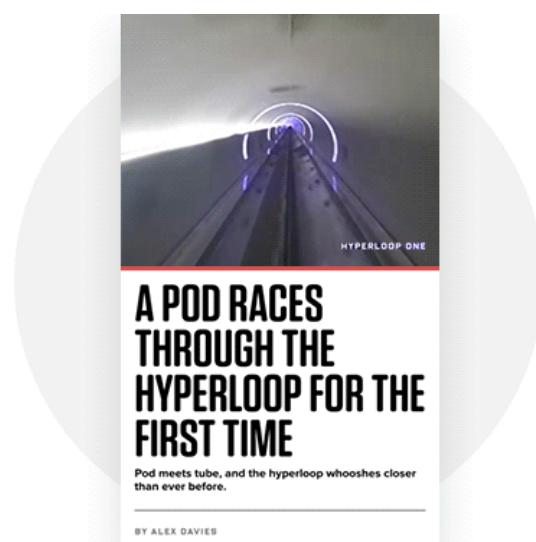
Polymer 是 Google 主推的一个 JavaScript 库，它可帮助你创建自定义的可重用 HTML 元素，并使用它们来构建高性能、可维护的 App。

在 I/O 大会上，Google 推出了 Polymer 3.0，Polymer 3.0 致力于将 Web 组件的生态系统从 HUML Imports 转移到 ES Modules，包管理系统将支持 npm，这使你更容易将基于 Polymer 的 web 组件和你喜欢的工具、框架协同使用。

AMP

AMP 是 Google 推出的一种为静态内容构建 Web 页面，提供可靠和快速渲染，加快页面加载速度的 Web 组件库。目前为止，来自 4600 万个域名的网页已经超过 60 亿个，他们在 Google 搜索的平均加载速度不超过 1 秒。

由于移动内容消费正在往全屏、简练的排版风格转变，为了满足内容发布商的需求，AMP 团队日前宣布开发了 AMP stories，它是一套为 mobile-first storytelling 开发的 Web 组，它支持原生视频和图像，具有丰富的视觉效果，可通过动画和可点击的交互方式来讲故事，这对于突发事件



的新闻报道具有重要作用，因为有时候图片能更直观和快速地向读者表达意思。

Lighthouse

Lighthouse 是一个分析网络质量的工具，为你提供网站性能衡量指标和指导，它可以直接从 Chrome DevTools 内部进行访问，从命令行运行或与其他开发产品集成，仅在 2018 年，就有 50 万开发人员在他们的网站上运行 Lighthouse。

本月初，Lighthouse 3.0 发布，最新的版本对网站的审核速度会更快，具有全新的报告界面，更多更新内容，可进入 Lighthouse 官网查看。

链接：<https://developers.google.com/web/updates/2018/05/lighthouse3>

如何做好文本关键词提取？ 从达观数据应用的三种算法说起

作者 韩伟



韩伟：达观数据数据挖掘工程师，负责达观数据文本方面的挖掘与应用。
主要参与达观数据标签提取与文本分类系统的构建与实现，对深度学习，
**NLP 数据挖掘领域有浓厚兴趣。

0 简介

在自然语言处理领域，处理海量的文本文件最关键的是要把用户最关心的问题提取出来。而无论是对于长文本还是短文本，往往可以通过几个关键词窥探整个文本的主题思想。同时，不管是基于文本的推荐还是基于文本的搜索，对于文本关键词的依赖也很大，关键词提取的准确程度直接关系到推荐系统或者搜索系统的最终效果。因此，关键词提取在文本挖掘

领域是一个很重要的部分。

关于文本的关键词提取方法分为有监督、半监督和无监督三种：

有监督的关键词抽取算法是将关键词抽取算法看作是二分类问题，判断文档中的词或者短语是或者不是关键词。既然是分类问题，就需要提供已经标注好的训练数据，利用训练语料训练关键词提取模型，根据模型对需要抽取关键词的文档进行关键词抽取。

半监督的关键词提取算法只需要少量的训练数据，利用这些训练数据构建关键词抽取模型，然后使用模型对新的文本进行关键词提取，对于这些关键词进行人工过滤，将过滤得到的关键词加入训练集，重新训练模型。

无监督的方法不需要人工标注的语料，利用某些方法发现文本中比较重要的词作为关键词，进行关键词抽取。

有监督的文本关键词提取算法需要高昂的人工成本，因此现有的文本关键词提取主要采用适用性较强的无监督关键词抽取。其文本关键词抽取流程如下：



图 1 无监督文本关键词抽取流程图

无监督关键词抽取算法可以分为三大类，基于统计特征的关键词抽取、基于词图模型的关键词抽取和基于主题模型的关键词抽取。

1、基于统计特征的关键词抽取算法

基于统计特征的关键词抽取算法的思想是利用文档中词语的统计信息抽取文档的关键词。通常将文本经过预处理得到候选词语的集合，然后采用特征值量化的方式从候选集合中得到关键词。基于统计特征的关键词抽取方法的关键是采用什么样的特征值量化指标的方式，目前常用的有三类：

1) 基于词权重的特征量化

基于词权重的特征量化主要包括词性、词频、逆向文档频率、相对词频、词长等。

2) 基于词的文档位置的特征量化

这种特征量化方式是根据文章不同位置的句子对文档的重要性不同的假设来进行的。通常，文章的前 N 个词、后 N 个词、段首、段尾、标题、引言等位置的词具有代表性，这些词作为关键词可以表达整个的主题。

3) 基于词的关联信息的特征量化

词的关联信息是指词与词、词与文档的关联程度信息，包括互信息、hits 值、贡献度、依存度、TF-IDF 值等。

我们介绍几种常用的特征值量化指标。

1.1 词性

词性时通过分词、语法分析后得到的结果。现有的关键词中，绝大多数关键词为名词或者动名词。一般情况下，名词与其他词性相比更能表达一篇文章的主要思想。但是，词性作为特征量化的指标，一般与其他指标结合使用。

1.2 词频

词频表示一个词在文本中出现的频率。一般我们认为，如果一个词在文本中出现的越是频繁，那么这个词就越有可能作为文章的核心词。词频简单地统计了词在文本中出现的次数，但是，只依靠词频所得到的关键词有很大的不确定行，对于长度比较长的文本，这个方法会有很大的噪音。

1.3 位置信息

一般情况下，词出现的位置对于词来说有着很大的价值。例如，标题、摘要本身就是作者概括出的文章的中心思想，因此出现在这些地方的词具有一定的代表性，更可能成为关键词。但是，因为每个作者的习惯不同，写作方式不同，关键句子的位置也会有所不同，所以这也是一种很宽泛的得到关键词的方法，一般情况下不会单独使用。

1.4 互信息

互信息是信息论中概念，是变量之间相互依赖的度量。互信息并不局限于实值随机变量，它更加一般且决定着联合分布 $p(X,Y)$ 和分解的边缘分布的乘积 $p(X)p(Y)$ 的相似程度。互信息的计算公式如下：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

其中， $p(x,y)$ 是 X 和 Y 的联合概率分布函数， $p(x)$ 和 $p(y)$ 分别为 X 和 Y 的边缘概率分布函数。

当使用互信息作为关键词提取的特征量化时，应用文本的正文和标题构造 PAT 树，然后计算字符串左右的互信息。

1.5 词跨度

词跨度是指一个词或者短语字文中首次出现和末次出现之间的距离，词跨度越大说明这个词对文本越重要，可以反映文本的主题。一个词的跨度计算公式如下：

$$span_i = \frac{last_i - first_i + 1}{sum}$$

其中 $last_i$ ，表示词 i 在文本中最后出现的位置， $first_i$ 表示词 i 在文本中第一次出现的位置， sum 表示文本中词的总数。

词跨度被作为提取关键词的方法是因为在现实中，文本中总是有很多噪声（指不是关键词的那些词），使用词跨度可以减少这些噪声。

1.6 TF-IDF 值

一个词的 TF 是指这个词在文档中出现的频率，假设一个词 w 在文本中出现了 m 次，而文本中词的总数为 n ，那么

$$TF_w = \frac{m}{n}$$

一个词的 IDF 是根据语料库得出的，表示这个词在整个语料库中出现的频率。假设整个语料库中，包含词 w 的文本一共有 M 篇，语料库中的文本一共有 N 篇，则

$$IDF_w = \log_2 \frac{N}{M}$$

由此可得词 w 的 TF-IDF 值为：

$$TFIDF_w = TF_w \times IDF_w$$

TF-IDF 的优点是实现简单，相对容易理解。但是，TFIDF 算法提取关键词的缺点也很明显，严重依赖语料库，需要选取质量较高且和所处理文本相符的语料库进行训练。另外，对于 IDF 来说，它本身是一种试图抑制噪声的加权，本身倾向于文本中频率小的词，这使得 TF-IDF 算法的精度不高。TF-IDF 算法还有一个缺点就是不能反应词的位置信息，在对关键词进行提取的时候，词的位置信息，例如文本的标题、文本的首句和尾句等含有较重要的信息，应该赋予较高的权重。

基于统计特征的关键词提取算法通过上面的一些特征量化指标将关键词进行排序，获取 TopK 个词作为关键词。

基于统计特征的关键词的重点在于特征量化指标的计算，不同的量化指标得到的记过也不尽相同。同时，不同的量化指标作为也有其各自的优缺点，在实际应用中，通常是采用不同的量化指标相结合的方式得到 Topk 个词作为关键词。

2 基于词图模型的关键词抽取算法

基于词图模型的关键词抽取首先要构建文档的语言网络图，然后对语言进行网络图分析，在这个图上寻找具有重要作用的词或者短语，这些短语就是文档的关键词。语言网络图中节点基本上都是词，根据词的链接方式不同，语言网络的主要形式分为四种：共现网络图、语法网络图、语义网络图和其他网络图。

在语言网络图的构建过程中，都是以预处理过后的词作为节点，词与词之间的关系作为边。语言网络图中，边与边之间的权重一般用词之间的关联度来表示。在使用语言网络图获得关键词的时候，需要评估各个节点的重要性，然后根据重要性将节点进行排序，选取 TopK 个节点所代表的词作为关键词。节点的重要性计算方法有以下几种方法。

2.1 综合特征法

综合特征法也叫社会网络中心性分析方法，这种方法的核心思想是节点中重要性等于节点的显著性，以不破坏网络的整体性为基础。此方法就是从网络的局部属性和全局属性角度去定量分析网络结构的拓扑性质，常用的定量计算方法如下。

2.1.1 度

节点的度是指与该节点直接向量的节点数目，表示的是节点的局部影响力，对于非加权网络，节点的度为：

$$d_i = k_i$$

对于加权网络，节点的度又称为节点的强度，计算公式为：

$$SC_i = \sum_j w_{ij}$$

2.1.2 接近性

节点的接近性是指节点到其他节点的最短路径之和的倒数，表示的是信息传播的紧密程度，其计算公式为：

$$C_i = \frac{N-1}{\sum_j d_{ij}}$$

2.1.3 特征向量

特征向量的思想是节点的中心化测试值由周围所有连接的节点决定，即一个节点的中心化指标应该等于其相邻节点的中心化指标之线性叠加，表示的是通过与具有高度值的相邻节点所获得的间接影响力。特征向量的计算公式如下：

$$EC_i = \frac{1}{\lambda} \sum_j a_{ij} x_{ij}$$

2.1.4 集聚系数

节点的集聚系数是它的相邻的节点之间的连接数与他们所有可能存在来链接的数量的比值，用来描述图的顶点之间阶级成团的程度的系数，计算公式如下：

$$C_i = \frac{2 |e_{jk}|}{k_i(k_i - 1)}$$

2.1.5 平均最短路径

节点的平均最短路径也叫紧密中心性，是节点的所有最短路径之和的平均值，表示的是一个节点传播信息时对其他节点的依赖程度。如果一个节点离其他节点越近，那么他传播信息的时候也就越不需要依赖其他人。一个节点到网络中各点的距离都很短，那么这个点就不会受制于其他节点。

计算公式如下：

$$CC_i = \frac{\sum_j d_{ij}}{N}$$

因为每个算法的侧重方向的不同，在实际的问题中所选取的定量分析方法也会不一样。同时，对于关键词提取来说，也可以和上一节所提出的统计法得到的词的权重，例如词性等相结合构建词搭配网络，然后利用上述方法得到关键词。

2.2 系统科学法

系统科学法进行中心性分析的思想是节点重要性等于这个节点被删除后对于整个语言网络图的破坏程度。重要的节点被删除后会对网络的连通性等产生变化。如果我们在网络图中删除某一个节点，图的某些指定特性产生了改变，可以根据特性改变的大小获得节点的重要性，从而对节点进行筛选。

2.3 随机游走法

随机游走算法时网络图中一个非常著名的算法，它从给定图和出发点，随机地选择邻居节点移动到邻居节点上，然后再把现在的节点作为出发点，迭代上述过程。

随机游走算法一个很出名的应用是大名鼎鼎的 PageRank 算法，PageRank 算法是整个 google 搜索的核心算法，是一种通过网页之间的超链接来计算网页重要性的技术，其关键的思想是重要性传递。在关键词提取领域，Mihalcea 等人所提出的 TextRank 算法就是在文本关键词提取领域借鉴了这种思想。

PageRank 算法将整个互联网看作一张有向图，网页是图中的节点，

而网页之间的链接就是图中的边。根据重要性传递的思想，如果一个大型网站 A 含有一个超链接指向了网页 B，那么网页 B 的重要性排名会根据 A 的重要性来提升。网页重要性的传递思想如下图所示。

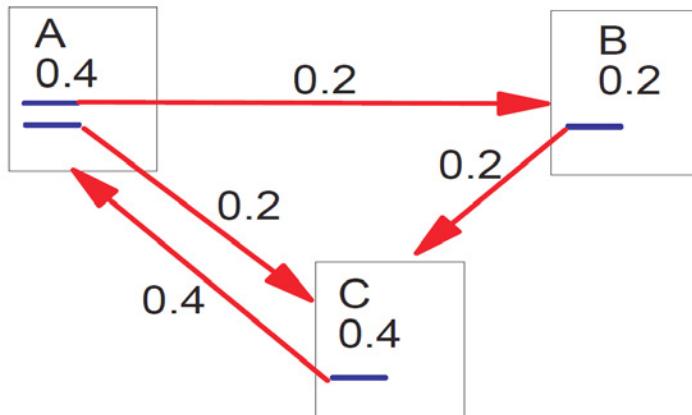


图 2 PageRank 简单描述（来自 PageRank 论文）

在 PageRank 算法中，最主要的是对于初始网页重要性 (PR 值) 的计算，因为对于上图中的网页 A 的重要性我们是无法预知的。但是，在原始论文中给出了一种迭代方法求出这个重要性，论文中指出，幂法求矩阵特征值与矩阵的初始值无关。那么，就可以为每个网页随机给一个初始值，然后迭代得到收敛值，并且收敛值与初始值无关。

PageRank 求网页 i 的 PR 值计算如下：

$$S(V_i) = (1 - d) + d \times \sum_{j \in In(V_i)} \frac{1}{|Out(V_j)|} S(V_j)$$

其中， d 为阻尼系数，通常为 0.85。 $\in (v_i)$ 是指向网页 i 的网页集合。 $out(v_j)$ 是指网页 j 中的链接指向的集合， $|out(v_j)|$ 是指集合中元素的个数。

TextRank 在构建图的时候将节点由网页改成了句子，并为节点之间的边引入了权值，其中权值表示两个句子的相似程度。其计算公式如下：

$$WS(V_i) = (1 - d) + d \times \sum_{j \in In(V_i)} \frac{w_{ji}}{\sum_{k \in Out(V_j)} w_{jk}} WS(V_j)$$

公式中的 $w_{\{ji\}}$ 为图中节点 v_i 和 v_j 的边的权重。其他符号与 PageRank 公式相同。

TextRank 算法除了做文本关键词提取，还可以做文本摘要提取，效果不错。但是 TextRank 的计算复杂度很高，应用不广。

3 基于主题模型的关键词抽取

基于主题关键词提取算法主要利用的是主题模型中关于主题的分布的性质进行关键词提取。算法步骤如下：

1. 从文章中获取候选关键词。即将文本分词，也可以再根据词性选取候选关键词。
2. 根据大规模预料学习得到主题模型。
3. 根据得到的隐含主题模型，计算文章的主题分布和候选关键词分布。
4. 计算文档和候选关键词的主题相似度并排序，选取前 n 个词作为关键词。

算法的关键在于主题模型的构建。主题模型是一种文档生成模型，对于一篇文章，我们的构思思路是先确定几个主题，然后根据主题想好描述主题的词汇，将词汇按照语法规则组成句子，段落，最后生成一篇文章。主题模型也是基于这个思想，它认为文档是一些主题的混合分布，主题又是词语的概率分布，pLSA 模型就是第一个根据这个想法构建的模型。同样地，我们反过来想，我们找到了文档的主题，然后主题中有代表性的词就能表示这篇文档的核心意思，就是文档的关键词。

pLSA 模型认为，一篇文档中的每一个词都是通过一定概率选取某个主题，然后再按照一定的概率从主题中选取得到这个词语，这个词语的计算公式为：

$$P\left(\frac{\text{词语}}{\text{文档}}\right) = \sum_{\text{主题}} P\left(\frac{\text{词语}}{\text{主题}}\right) \times P\left(\frac{\text{主题}}{\text{文档}}\right)$$

一些贝叶斯学派的研究者对于 pLSA 模型进行了改进，他们认为，文章对应主题的概率以及主题对应词语的概率不是一定的，也服从一定的概率，于是就有了现阶段常用的主题模型 --LDA 主题模型。

LDA 是 D.M.Blei 在 2003 年提出的。LDA 采用了词袋模型的方法简化

了问题的复杂性。在 LDA 模型中，每一篇文档是一些主题的构成的概率分布，而每一个主题又是很多单词构成的一个概率分布。同时，无论是主题构成的概率分布还是单词构成的概率分布也不是一定的，这些分布也服从 Dirichlet 先验分布。

文档的生成模型可以用如下图模型表示：

其中 α 和 η 为先验分布的超参数， β 为第 k 个主题下的所有单词的分布， θ 为文档的主题分布， w 为文档的词， z 为 w 所对应的主题。

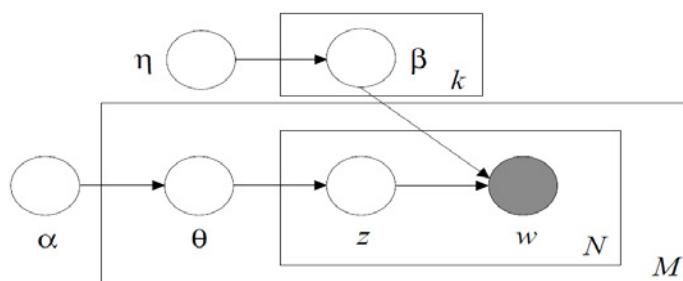


图 3 Blei 在论文中的图模型

LDA 挖掘了文本的深层语义即文本的主题，用文本的主题来表示文本的也从一定程度上降低了文本向量的维度，很多人用这种方式对文本做分类，取得了不错的效果。具体 LDA 的算法在请参考。

LDA 关键词提取算法利用文档的隐含语义信息来提取关键词，但是主题模型提取的关键词比较宽泛，不能很好的反应文档主题。另外，对于 LDA 模型的时间复杂度较高，需要大量的实践训练。

4 应用

现在阶段，文本的关键词提取在基于文本的搜索、推荐以及数据挖掘领域有着很广泛的应用。同时在实际应用中，因为应用环境的复杂性，对于不同类型的文本，例如长文本和短文本，用同一种文本关键词提取方法得到的效果并不相同。因此，在实际应用中针对不同的条件环境所采用的算法会有所不同，没有某一类算法在所有的环境下都有很好的效果。

相对于上文中所提到的算法，一些组合算法在工程上被大量应用以弥

补单算法的不足，例如将 TF-IDF 算法与 TextRank 算法相结合，或者综合 TF-IDF 与词性得到关键词等。同时，工程上对于文本的预处理以及文本分词的准确性也有很大的依赖。对于文本的错别字，变形词等信息，需要在预处理阶段予以解决，分词算法的选择，未登录词以及歧义词的识别在一定程度上对于关键词突提取会又很大的影响。

关键词提取是一个看似简单，在实际应用中却十分棘手的任务，从现有的算法的基础上进行工程优化，达观数据在这方面做了很大的努力并且取得了不错的效果。

5 总结

本文介绍了三种常用的无监督的关键词提取算法，并介绍了其优缺点。关键词提取在文本挖掘领域具有很广阔的应用，现有的方法也存在一定的问题，我们依然会在关键词提取的问题上继续努力研究，也欢迎大家积极交流。

参考文献

- [1] TextRank 算法提取关键词和摘要 <http://xiaosheng.me/2017/04/08/article49/>
- [2] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: Bringing order to the web[R]. Stanford InfoLab, 1999.
- [3] 刘知远 . 基于文档主题结构的关键词抽取方法研究 [D]. 北京 : 清华大学 , 2011.
- [4] tf-idf, <https://zh.wikipedia.org/zh-hans/Tf-idf>
- [5] 一文详解机器领域的 LDA 主题模型 <http://zhuanlan.51cto.com/art/201712/559686.htm?mobile>
- [6] Blei D M, Ng A Y, Jordan M I. Latent dirichlet allocation[J]. Journal of machine Learning research, 2003, 3(Jan): 993-1022.
- [7] 赵京胜 , 朱巧明 , 周国栋 , 等 . 自动关键词抽取研究综述 [J]. 软件学报 , 2017, 28(9): 2431-2449.

优秀架构师必须掌握的架构思维

作者 杨波



介绍

架构的本质是管理复杂性，抽象、分层、分治和演化思维是我们工程师 / 架构师应对和管理复杂性的四种最基本武器。

最近团队来了一些新人，有些有一定工作经验，是以高级工程师 / 架构师身份进来的，但我发现他们大部分人思维偏应用和细节，抽象能力弱。所以作为团队技术培训的一部分，我整理了这篇文章，希望对他们树立正确的架构设计思维有帮助。我认为，对思维习惯和思考能力的培养，其重要性远远大于对实际技术工具的掌握。

由于文章内容较长，所以我把它分成两篇小文章，在第一篇《优秀架

构师必须掌握的架构思维》中，我会先介绍抽象、分层、分治和演化这四种应对复杂性的基本思维。在第二篇《四个架构设计案例及其思维方式》中，我会通过四个案例，讲解如何综合运用这些思维，分别对小型系统，中型系统，基础架构，甚至是组织技术体系进行架构和设计。

一、抽象思维

如果要问软件研发 / 系统架构中最重要的能力是什么，我会毫不犹豫回答是抽象能力。抽象 (abstraction) 这个词大家经常听到，但是真正理解和能讲清楚什么是抽象的人少之又少。抽象其实是这样定义的：

对某种事物进行简化表示或描述的过程，抽象让我们关注要素，隐藏额外细节。

举一个例子，见下图：



你看到什么？你看到的是一扇门，对不对？你看到的不是木头，也不是碳原子，这个门就是抽象，而木头或者碳原子是细节。另外你可以看到门上有个门把手，你看到的不是铁，也不是铁原子，门把手就是抽象，铁和铁原子是细节。

在系统架构和设计中，抽象帮助我们从大处着眼 (get our mind about big picture)，隐藏细节 (temporarily hide details)。抽象能力的强弱，直接决定我们所能解决问题的复杂性和规模大小。

下图是我们小时候玩的积木，我发现小时候喜欢玩搭积木的，并且搭得快和好的小朋友，一般抽象能力都比较强。

下图右边的积木城堡就是抽象，这个城堡如果你细看的话，它其实还

是由若干个子模块组成，这些模块是子抽象单元，左边的各种形状的积木是细节。搭积木的时候，小朋友脑袋里头先有一个城堡的大图（抽象），然后他 / 她大脑里头会有一个初步的子模块分解（潜意识中完成），然用利用积木搭建每一个子模块，最终拼装出最后的城堡。这里头有一个自顶向下的分治设计，然后自底向上的组合过程，这个分治思维非常重要，我们后面会讲。



我认为软件系统架构设计和小朋友搭积木无本质差异，只是解决问题域和规模不同罢了。架构师先要在大脑中形成抽象概念，然后是子模块分解，然后是依次实现子模块，最后将子模块拼装组合起来，形成最后系统。所以我常说编程和架构设计就是搭积木，优秀的架构师受职业习惯影响，眼睛里看到的世界都是模块化拼装组合式的。

抽象能力不仅对软件系统架构设计重要，对建筑、商业、管理等人类其它领域活动同样非常重要。其实可以这样认为，我们生存的世界都是在抽象的基础上构建起来的，离开抽象人类将寸步难行。

这里顺便提一下抽象层次跳跃问题，这个在开发中是蛮普遍的。有经验的程序员写代码会保持抽象层次的一致性，代码读起来像讲故事，比较清晰易于理解；而没有经验的程序员会有明显的抽象层次跳跃问题，代码读起来就比较累，这个是抽象能力不足造成。

举个例子：



一个电商网站在处理订单时，一般会走这样一个流程：

1. 更新库存 (InventoryUpdate)

2. 打折计算 (Discounting)
3. 支付卡校验 (PaycardVerification)
4. 支付 (Pay)
5. 送货 (Shipping)

上述流程中的抽象是在同一个层次上的，比较清晰易于理解，但是没有经验的程序员在实现这个流程的时候，代码层次会跳，比方说主流程到支付卡校验一块，他的代码会突然跳出一行某银行 API 远程调用，这个就是抽象跳跃，银行 API 调用是细节，应该封装在 PaycardVerification 这个抽象里头。

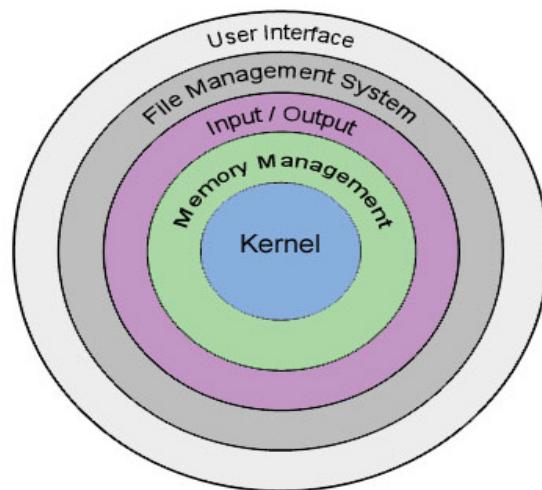
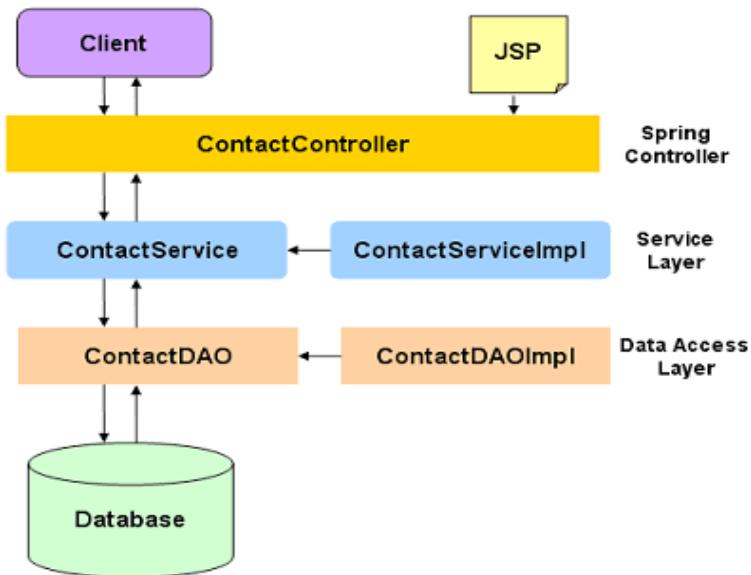
二、分层思维

除了抽象，分层也是我们应对和管理复杂性的基本思维武器，如下图，为了构建一套复杂系统，我们把整个系统划分成若干个层次，每一层专注解决某个领域的问题，并向上提供服务。有些层次是纵向的，它贯穿所有其它层次，称为共享层。分层也可以认为是抽象的一种方式，将系统抽象分解成若干层次化的模块。

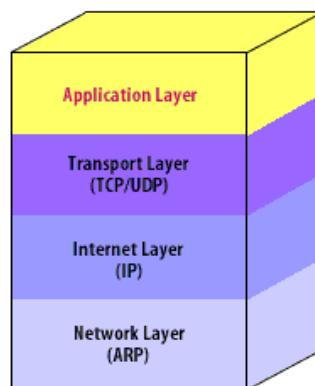


分层架构的案例很多，一个中小型的 Spring Web 应用程序，我们一般会设计成三层架构。

操作系统是经典的分层架构，如下图。TCP/IP 协议栈也是经典的分层架构，如下图。



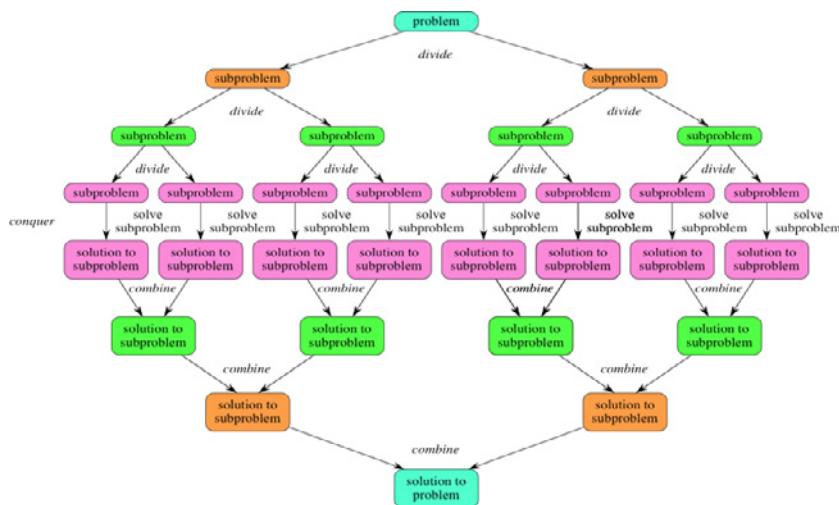
TCP/IP Protocol Layers



如果你关注人类文明演化史，你会发现今天的人类世界也是以分层方式一层层搭建和演化出来的。今天的互联网系统可以认为是现代文明的一个层次，其上是基于互联网的现代商业，其下是现代电子工业基础设施，诸如此类。

三、分治思维

分而治之 (divide and combine 或者 split and merge) 也是应对和管理复杂性的一般性方法，下图展示一个分治的思维流程：



对于一个无法一次解决的大问题，我们会先把大问题分解成若干个子问题，如果子问题还无法直接解决，则继续分解成子子问题，直到可以直接解决的程度，这个是分解 (divide) 的过程；然后将子子问题的解组合拼装成子问题的解，再将子问题的解组合拼装成原问题的解，这个是组合 (combine) 的过程。

面试时为了考察候选人的分治思维，我经常会面一个分治题：给你一台 8G 内存 / 500G 磁盘空间的普通电脑，如何对一个 100G 的大文件进行排序？假定文件中都是字符串记录，一行约 100 个字符。

这是一个典型的分治问题，100G 的大文件肯定无法一次加载到内存直接排序，所以需要先切分成若干小问题来解决。那么 8G 内存的计算机一次大概能排多大的数据量，可以在有限的时间内排完呢？也就是 100G

的大文件要怎么切法，切成多少份比较合适？这个是考察候选人的时间空间复杂度估算能力，需要一定的计算机组织和算法功底，也需要一定实战经验和 sense。实际上 8G 内存的话，操作系统要用掉一部分，如果用 Java 开发排序程序，大致 JVM 可用 2~4G 内存，基于一般的经验值，一次排 1G 左右的数据应该没有问题（我实际在计算机上干过 1G 数据的排序，是 OK 的）。所以 100G 的文件需要先切分成 100 份，每份 1G，这样每个子文件可以直接加载到内存进行排序。对于 1G 数据量的字符串排序，采用 Java 里头提供的快速排序算法是比较合适的。

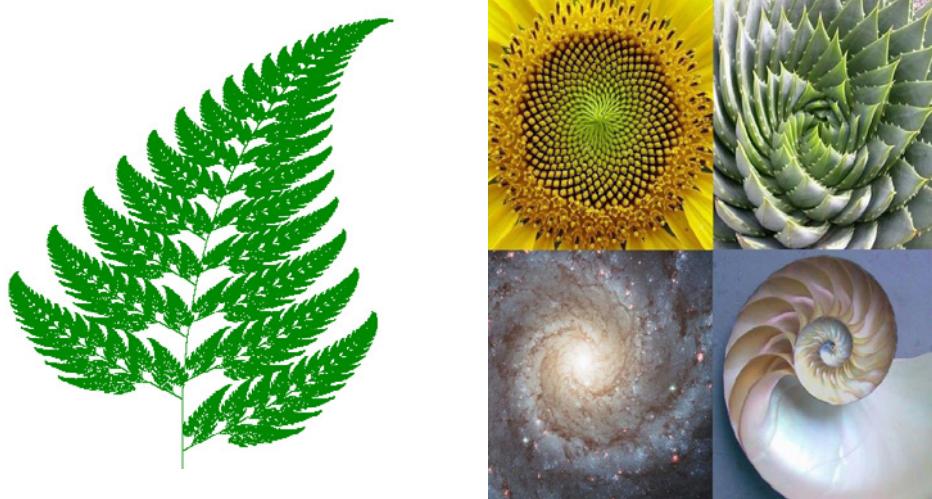
好，经过有限时间的排序（取决于计算机性能，快的一天内能排完），假定 100 个 1G 的文件都已经排好了，相当于现在硬盘上有 100 个已经排好序的文件，但是我们最终需要的是一个排好序的文件，下面该怎么做？这个时候我们需要把已经解决的子问题组合起来，合并成我们需要的最终结果文件。这个时候该采用什么算法呢？这里考察候选人对外排序和归并排序算法的掌握程度，我们可以将 100 个排好序的文件进行两两归并排序，这样不断重复，我们就会得到 50 个排好序的文件，每个大小是 2G。然后再两两归并，不断重复，直到最后两个文件归并成目标文件，这个文件就是 100G 并且是排好序的。因为是外排序 + 归并排序，每次只需要读取当前索引指向的文件记录到内存，进行比较，小的那个输出到目标文件，内存占用极少。另外，上面的算法是两路归并，也可以采用多路归并，甚至是采用堆排序进行优化，但是总体分治思路没有变化。

总体上这是一个非常好的面试题，除了考察候选人的分治思维之外，还考察对各种排序算法（快排，外排序，归并排序，堆排序）的理解，计算的时间空间复杂度估算，计算机的内外存特性和组织，文件操作等等。实际上能完全回答清楚这个问题的候选人极少，如果有幸被我面到一个，我会如获至宝，因为这个人有成长为优秀架构师的潜质。

另外，递归也是一种特殊的分治技术，掌握递归技术的开发人员，相当于掌握了一种强大的编程武器，可以解决一些一般开发人员无法解决的问题。比方说最近我的团队在研发一款新的服务框架，其中包括契约解析

器(parser)，代码生产器(code generator)，序列化器(serializer)等组件，里头大量需要用到递归的思维和技术，没有这个思维的开发人员就干不了这个事情。所以我在面试候选人的时候，一般都会出递归相关的编程题，考察候选人的递归思维。

大自然中递归结构比比皆是，如下图，大家有兴趣不妨思考，大自然



通过递归给我们人类何种启示？

四、演化思维

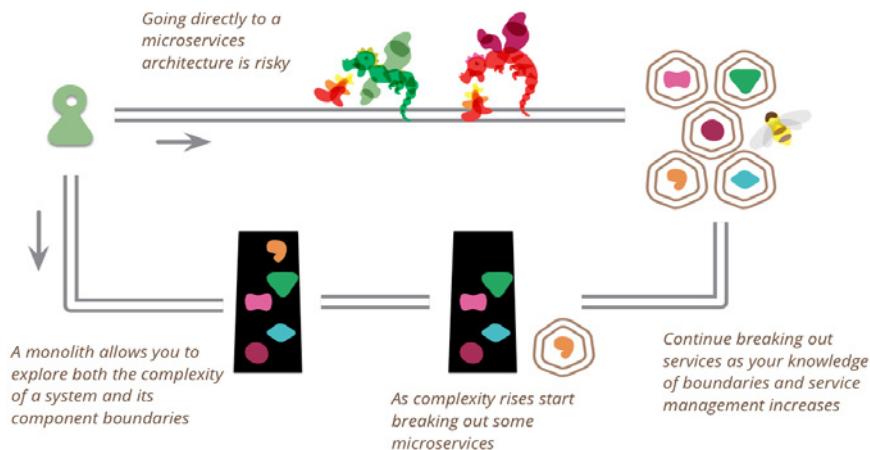
社区里头经常有人在讨论：架构是设计出来的？还是演化出来的？我个人基于十多年的经验认为，架构既是设计出来的，同时也是演化出来的，对于互联网系统，基本上可以说是三分设计，七分演化，而且是在设计中演化，在演化中设计，一个不断迭代的过程。

在互联网软件系统的整个生命周期过程中，前期的设计和开发大致只占三分，在后面的七分时间里，架构师需要根据用户的反馈对架构进行不断的调整。我认为架构师除了要利用自身的架构设计能力，同时也要学会借助用户反馈和进化的力量，推动架构的持续演进，这个就是演化式架构思维。

当然一开始的架构设计非常重要，架构定系统基本就成型了，不容马

虎。同时，优秀的架构师深知，能够不断应对环境变化的系统，才是有生命力的系统，架构的好坏，很大部分取决于它应对变化的灵活性。所以具有演化式思维的架构师，能够在一开始设计时就考虑到后续架构的演化特性，并且将灵活应对变化的能力作为架构设计的主要考量。

当前，社区正在兴起一种新的架构方法学～演化式架构，微服务架构就是一种典型的演化式架构，它能够快速响应市场用户需求的变化，而单块架构就缺乏这种灵活性。马丁·福乐曾经在其博客上给出过一张微服务架构的演化路线图 [附录 8.2]，可以用来解释设计式思维和演化式思维的差异，如下图所示：

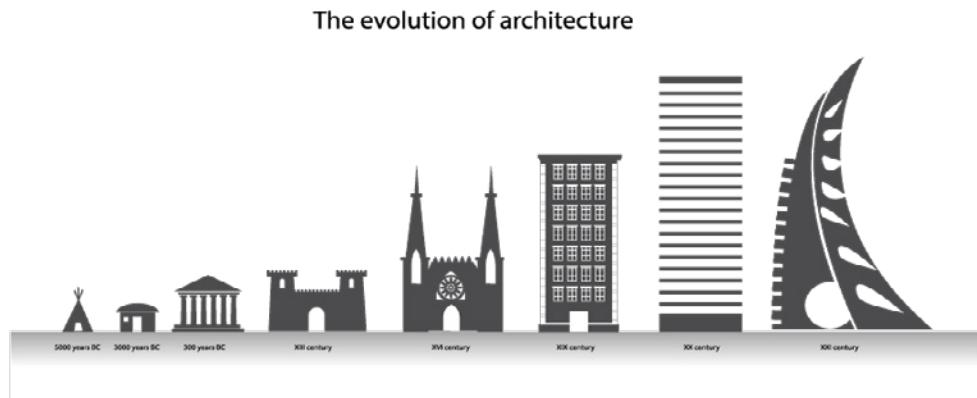


上面的路线是一开始就直奔微服务架构，其实背后体现的是设计式架构的思维，认为架构师可以完全设计整个系统和它的演化方向。马丁认为这种做法风险非常高，一个是成本高昂，另外一个是刚开始架构师对业务域理解不深，无法清晰划分领域边界，开发出来的系统很可能无法满足用户需求。

下面的路线是从单块架构开始，随着架构师对业务域理解的不断深入，也随着业务和团队规模的不断扩大，渐进式地把单块架构拆分成微服务架构的思路，这就是演化式架构的思维。如果你观察现实世界中一些互联网公司（例如 eBay，阿里，Netflix 等等）的系统架构，大部分走得都是演化式架构的路线。

下图是建筑的演化史，在每个阶段，你可以看到设计的影子，但如果

时间线拉得足够长，演化的特性就出来了。



五、如何培养架构设计思维

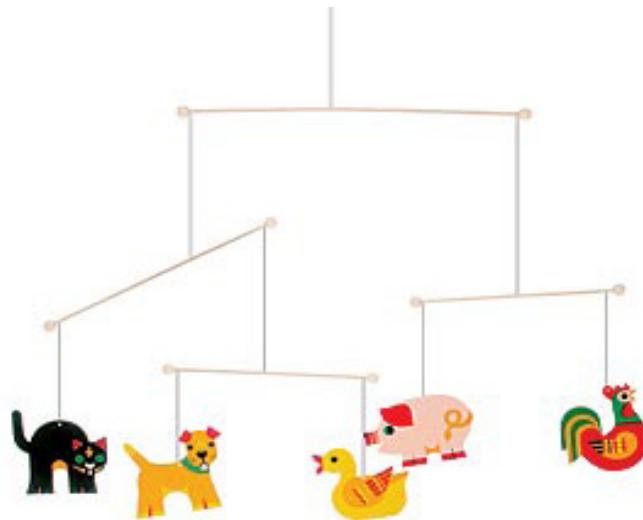
良好的架构设计思维的培养，离不开工作中大量高质量项目的实战锻炼，然后是平时的学习、思考和提炼总结。

另外，基本的架构设计思维，其实在我们大学计算机课程（比如数据结构和算法）中可以找到影子，只不过当时以学习为主，问题域比较小和理想化。所以大学教育其实非常重要，基本的架构设计思维在那个时候就已经埋下种子，后面工程实践中进一步消化和应用，随着经验的积累，我们能够解决的问题域复杂性和规模逐渐变大，但基本的武器还是抽象、分层和分治等思维。

我认为一个架构师的成长高度和他大学期间的思维习惯的养成关系密切。我所知道世界一流的互联网公司，例如谷歌等，招聘工程师新人时，对数据结构和算法的要求可以用苛刻来形容，这个可以理解，谷歌级别公司要解决的问题都是超级复杂的，基本思维功底薄弱根本无法应对。

对于工作经验 <5 年的工程师新手，如果你大学时代是属于荒废型的，建议工作之余把相关课程再好好自学一把。个人推荐参考美国 Berkeley 大学的数据结构课程 CS61B[附录 8.1] 进行学习，对建立抽象编程思维非常有帮助，我本人在研究生阶段自学过这门课程，现在回想起来确实受益匪浅，注意该课程中的所有 Lab/Homework/Project 都要实际动手做一遍，

才有好的效果。



我当年自学的是 CS61B 2006 秋季版的课程，上图是课程 Logo

对于演化设计思维，当前的大学教育其实培养很少，相反，当前大学教育大都采用脱离现实场景的简化理想模型，有些还是固定答案的应试教学，这种方式会造成学生思维确定化，不利于培养演化式设计思维。我个人的体会，演化式设计思维更多在实际工作中通过实战锻炼和培养。

结论

1. 架构的本质是管理复杂性，抽象、分层、分治和演化思维是架构师征服复杂性的四种根本性武器。
2. 掌握了抽象、分层、分治和演化这四种基本的武器，你可以设计小到一个类，一个模块，一个子系统，或者一个中型的系统，也可以大到一个公司的基础平台架构，微服务架构，技术体系架构，甚至是组织架构，业务架构等等。
3. 架构设计不是静态的，而是动态演化的。只有能够不断应对环境变化的系统，才是有生命力的系统。所以即使你掌握了抽象、分层和分治这三种基本思维，仍然需要演化式思维，在设计的同时，借助反馈和进化的力量推动架构的持续演进。

4. 架构师在关注技术，开发应用的同时，需要定期梳理自己的架构设计思维，积累时间长了，你看待世界事物的方式会发生根本性变化，你会发现我们生活其中的世界，其实也是在抽象、分层、分治和演化的基础上构建起来的。另外架构设计思维的形成，会对你的系统架构设计能力产生重大影响。可以说对抽象、分层、分治和演化掌握的深度和灵活应用的水平，直接决定架构师所能解决问题域的复杂性和规模大小，是区分普通应用型架构师和平台型 / 系统型架构师的一个分水岭。

参考

1. Berkeley CS61B <http://datastructur.es/sp17/>
2. 单块优先 <https://www.martinfowler.com/bliki/MonolithFirst.html>

区块链没有未来，是时候抛弃它了

作者 Kai Stinchcombe，译者 无明



区块链不过是一项蹩脚的技术，它根本就没有未来。区块链技术之所以没能被大规模采用，主要是因为建立在信任、规范和监管机构之上的系统比区块链所愿景的无信任机构系统运作得更好。区块链在一开始就走错了方向，所以不管它如何进步，也改变不了错误的事实。

去年 12 月份，我写了一篇关于区块链无法解决实际问题的文章，受到广泛关注。大多数人并没有停留在技术的争论上，而是强调去中心化可以带来正直诚信。

Venmo 提供了免费的汇款服务，但对比特币不免费。去年 12 月，在我写了那篇说比特币没有实际用处的文章之后，有人就站出来，说 Venmo 和 Paypal 正在利用用户的款项来敛财，所以人们应该改用比特币。

可见，在区块链的无用和它的追随者之间存在巨大的鸿沟！事实上，这个人并非比特币的狂热追随者，他只不过是想找到一种既方便又免费的汇款方式，而比特币刚好满足了他的需求。我敢肯定，在现实当中，没有哪一个人会认为区块链是解决他们问题的唯一良方，并因此成为区块链的狂热追随者。

采用数字货币作为支付手段的零售商已经越来越少了，而区块链最大的推动者（如 IBM、NASDAQ、Fidelity、Swift 和沃尔玛）大多数也只是光说不练。即使是声名远播的区块链公司 Ripple 也没有在自己的产品上使用区块链技术。Ripple 公司认为，进行国际汇款最好的方式是不使用 Ripple。

区块链是一项技术，而不是某种隐喻

为什么对某些事物的狂热在实际当中却没有什么用处？

人们“臆想”了一个区块链的未来，比如像谷歌或 Facebook 那样将区块链应用在 AI 上。之所以出现这些臆想，是因为人们误解了区块链。区块链只不过是一种数据结构，一种线性的交易日志，这些日志通常由计算节点的所有者（也叫挖矿者）产生，他们会因为记录交易而受到奖励。

这种数据结构有两点是值得一提的。首先，修改任何一个区块，都会导致其后的那个区块失效，也就是说，我们无法篡改历史交易。其次，我



们只有与其他人同在一条链上才有可能获得奖励，所以每个参与者都倾向于达成共识。

在《黄金罗盘》这部电影里，尘埃弥漫，并经由意识形成了万物，但区块链并不是这样的。

我们最终会得到一个权威的历史记录。而且，因为共识的形成是基于每个人的利益，所以如果有人加入虚假交易就会导致所有人都得不到奖励。这样一来，遵守规则就变成潜移默化的，不需要政府或警察的介入。这是一个非常伟大的想法。

简而言之，区块链就是这样的一种技术：“让我们一起来创建一长串小文件，每个文件里包含了上一个文件的散列值、一些新数据和算法的答案，如果有人愿意在他们的电脑上验证并保存这些文件，就奖励他们一些钱”。

而关于区块链的隐喻是这样描述的：所有人都把自己的记录保存在一个不属于任何人的防篡改仓库里。

2006 年，沃尔玛启动了一个系统，用于追踪香蕉和芒果从产地到商店之间的过程。2009 年，他们停用了这个系统，因为它要求所有人都要往系统里输入数据，体验并不是很好。2017 年，沃尔玛基于区块链技术重新启动了这个系统。如果有人告诉你说，“果农不喜欢输入数据”，你会怎么回答他？你可能会说，“我知道，那就让我们创建一长串小文件，每个文件里包含了上一个文件的散列值”。如果说这样，他们可能一头雾水，但如果说，“让他们把记录保存在一个不属于任何人的防篡改仓库里”，他们可能更容易理解。

基于区块链的信任在实践当中土崩瓦解

人们把区块链看成是“未来正直诚信的沃土”，只要使用区块链来解决你的问题，你的数据就会在瞬间变成正当合法的。人们想让什么变得正当合法，只要用上区块链技术就可以了。

要篡改区块链的数据确实很困难，但区块链并非创建正当合法数据最

有效的方式。

为了说明这个问题，我们先举实例再谈理论。我们以一个大家较为熟悉的区块链应用场景为例——通过“智能”合约购买电子书。区块链在这里起到的作用就是“信任”：你不信任电子书提供商，他们也不信任你（因为你们都只不过是互联网上的互不知晓的个体），但都信任在区块链上进行的交易。

在传统的场景里可能是这样的：你付了钱并等待收货，但商家收到钱之后并没有发货的意思。你唯一能够依靠的是 Visa、亚马逊或者政府，让他们帮你解决这类问题。相反，在区块链系统里，通过往一个不属于任何人的防篡改仓库里添加记录来执行交易，不管是汇款还是购买数字产品，都是直接进行的，不需要中间人仲裁，也没有人能够从中窃取好处。这样不是比传统的方式更好吗？

或许你很懂软件，当作家向你出示智能合约，你花了一两个小时时间，确保合约只会提走与公允的电子书价格等价的款项，然后，你就会收到电子书，而不是一堆文件。

但软件审计是很难做到完美的！即使是审核最严格的智能合约也会出现 bug，而人们通常注意不到，等到他们注意到了，窃贼已经利用它偷走了 5 千万美金。由数字货币的狂热者一起出资 1 亿 5 千万美元所建立起来的基金都不能保证万无一失的审计，你又凭什么对你的电子书审计如此自信？或许你应该要开发自己的买方软件合约，万一电子书作者在他们的合约里藏了后门，然后用它从你的以太坊钱包里偷走你的毕生积蓄呢？

买电子书的过程非常复杂！这其中并不缺乏信任，只是你选择信任软件，而不是人。

再举一个例子：区块链在弱监管国家投票系统上的应用。“把你的投票记录放进一个不属于任何人的防篡改仓库中”这句话似乎是对的，但问题是，选民们是否需要自己从广播节点上下载区块，并通过 Linux 命令行破解默克尔密码，才能知道自己的选票是否被统计进去了？抑或是他们需要使用由负责监管投票的第三方信任机构为他们提供的应用才能知道投票

的真实情况？



“我想看看系统的源代码，确保这个人没有重复投票。”

这些听起来都很傻，电子书作者和选民使用电子“保安”来保护自己，但不法用户或第三方信任机构却用智能合约来偷取你的存款或篡改选票。在区块链世界里，因为没有了信任和监管，个体需要为自己的安全负责。而如果他们所使用的软件出现了 bug，或者遭到木马侵害，那后果就更加严重了！

我们对区块链的认识是错误的

或许大家已经司空见惯了，区块链系统总是被认为是更值得信任的，但事实上它们是世界上最不值得信任的系统。在不到十年的时间里，三个排名靠前的比特币交易平台被黑客攻击，另一个因为出现内幕交易被起诉，模范智能合约项目 DAO 走到了尽头，数字货币的价格虚高且摇摆不定，比特币更是由背后数十亿美元的虚假交易支撑起来的。

区块链系统并不能保证人们输入的数据一定是可信的，它只能保证当中的数据不会被篡改。一个果农在芒果上喷了农药，但他仍然可以在区块链系统里记录说他的芒果是完全有机的。一个腐败的政府可以创建一个区块链系统用于统计选票，并给它的亲信偷偷输入额外的数百万张选票。一个获得数字执照的投资基金会仍然有可能错配资金。

区块链能阻止这个果农往芒果上喷洒农药吗？

那么信任从何而来？

在购买电子书的例子中，即使使用了智能合约，如果不对软件进行审计，那么你只能依赖以下四种“传统”信任机制：你知道智能合约的作者是谁，并认为对方值得信任；电子书零售商有良好的声誉；你或你的朋友在过去曾经向这家零售商成功购买过电子书；你寄希望于这家零售商会诚实守信。对于上述的四种情况，即使在交易中使用了智能合约，你仍然需要依赖中间人或者卖方的信誉。智能合约依然奏效，但它的透明度变得更低了，而不是更高。

选票统计也是一样的。在出现区块链之前，我们只能相信投票管理处会公平处理选票，相信只有合法的选民参与了投票，相信选票是无记名的，不能通过威逼利诱的方式获得，相信公告系统上显示的选票是真实的，相信政治亲信不会获得额外的选票。区块链并不会让这些问题变得更容易解决，相反，它只会增加难度。不过更重要的是，通过区块链解决这些问题需要用到一些手段，而这些手段会破坏投票的核心许诺。事实上，纵观任何一个区块链解决方案，我们都会不可避免地发现，它们都通过某种手段在缺乏信任的世界里建立信任实体。

原始的数字货币系统

如果没有了“传统”因素，完全依赖区块链的自利和自我保护机制来创建真实世界的系统，那么你极有可能会陷入泥潭之中。

八百多年前的欧洲，政府软弱无能，盗贼横行，安全的银行系统只存在于理想之中，而个人的安危更是悬在剑锋上。现在的索马里也是这个样子，而在区块链上进行交易也与此有点相似。

相信没有人会希望我们的世界变成索马里！

即使是数字货币的死忠也不会完全信任他们自己的系统。93% 的比特币是由受监管的团体挖出来的，但这些团体并没有使用智能合约来管理支出款项。

“丝绸之路（Silk Road）”是以数字货币为驱动的在线毒品交易市场。这个网站的关键之处并非比特币（用于规避政府的监管），而是一种“声誉值”，声誉值越高表示毒贩越值得信任。但这个声誉值并不是通过可以防篡改的区块链系统来跟踪的，而是由一个中间人来操控！

如果 Ripple、丝绸之路、Slush Pool 和 DAO 都依赖“传统”系统来建立信任，那说明我们的世界根本不存在真正的去信任系统！

是时候抛弃区块链了

使用去中心化的防篡改仓库来跟踪芒果的产地、新鲜度、是否用过农药，看起来像那么回事。但实际上，食品安全法、非盈利机构或政府的检验员、独立的新闻媒体、获得一定授权的举报者、有信誉的食品商店、本地农场市场在这方面做得更好。真正关心食品安全的人不会使用区块链，因为他们认为信任强过无信任。区块链技术已经暴露出它的弊端，将数据保存成一长串小文件与果农是否如实报告芒果是否使用了农药之间并没有必然的联系。类似的，点对点的交互缺乏监管、规范约束、中间人或信任实体，是一种非常糟糕的授权方式。

去信任的项目之所以会失败，是因为它们无法给用户带来真正的好处。信任真的太重要了！在一个无法律、无信任的世界里，自利是仅剩的原则，安全只存在于妄想之中，这样的世界绝对不是天堂，而是藏污纳垢之地。

一个可供中小团队参考的微服务架构技术栈

作者 杨波



前言

近年，Spring Cloud 俨然已经成为微服务开发的主流技术栈，在国内开发者社区非常火爆。

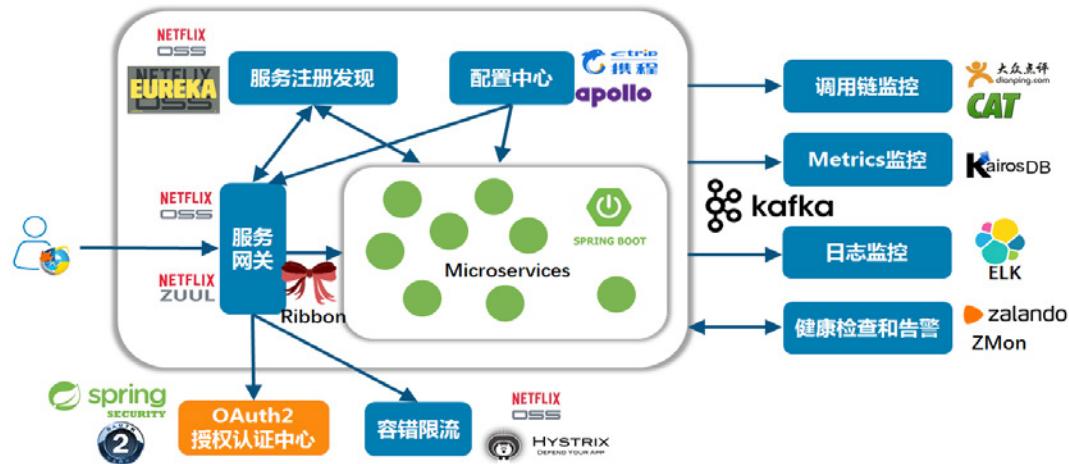
我近年一直在一线互联网公司（携程，拍拍贷等）开展微服务架构实践，根据我个人的一线实践经验和我平时对 Spring Cloud 的调研，我认为 Spring Cloud 技术栈中的有些组件离生产级开发尚有一定距离。

比方说 Spring Cloud Config 和 Spring Cloud Sleuth 都是 Pivotal 自研产品，尚未得到大规模企业级生产应用，很多企业级特性缺失（具体见我后文描述）。另外 Spring Cloud 体系还缺失一些关键的微服务基础组件，比

如 Metrics 监控，健康检查和告警等。

所以我在参考 Spring Cloud 微服务技术栈的基础上，结合自身的实战落地经验，也结合国内外一线互联网公司（例如 Netflix，点评，携程，Zalando 等）的开源实践，综合提出更贴近国内技术文化特色的轻量级的微服务参考技术栈。希望这个参考技术栈对一线的架构师（或者是初创公司）有一个好的指导，能够少走弯路，快速落地微服务架构。

这个参考技术栈和总体架构如下图所示：



主要包含 11 大核心组件，分别是：

核心支撑组件

1. 服务网关 Zuul
2. 服务注册发现 Eureka+Ribbon
3. 服务配置中心 Apollo
4. 认证授权中心 Spring Security OAuth2
5. 服务框架 Spring MVC/Boot

监控反馈组件

1. 数据总线 Kafka
2. 日志监控 ELK
3. 调用链监控 CAT
4. Metrics 监控 KairosDB

5. 健康检查和告警 ZMon
6. 限流熔断和流聚合 Hystrix/Turbine

核心支撑组件

服务网关Zuul

2013 年左右，infoq 曾经对前 Netflix 架构总监 Adrian Cockcroft 有过一次专访 [附录 1]，其中有问 Adrian：“Netflix 开源这么多项目，你认为哪一个是最不可或缺的 (MOST Indispensable)”，Adrian 回答说：“在 NetflixOSS 开源项目中，有一个容易被忽略，但是 Netflix 最强大的基础服务之一，它就是 Zuul 网关服务。Zuul 网关主要用于智能路由，同时也支持认证，区域和内容感知路由，将多个底层服务聚合成统一的对外 API。Zuul 网关的一大亮点是动态可编程，配置可以秒级生效”。从 Adrian 的回答中，我们可以感受到 Zuul 网关对微服务基础架构的重要性。



Zuul 在英文中是一种怪兽，星际争霸中虫族里头也有 Zuul，Netflix 为网关起名 Zuul，寓意看门神兽

Zuul 网关在 Netflix 经过生产级验证，在纳入 Spring Cloud 体系之后，在社区中也有众多成功的应用。Zuul 网关在携程（日流量超 50 亿），拍拍贷等公司也有成功的落地实践，是微服务基础架构中网关一块的首选。

其它开源产品像 Kong 或者 Nginx 等也可以改造支持网关功能，但是较复杂门槛高一点。

Zuul 网关虽然不完全支持异步，但是同步模型反而使它简单轻量，易于编程和扩展，当然同步模型需要做好限流熔断（和限流熔断组件 Hystrix 配合），否则可能造成资源耗尽甚至雪崩效应（cascading failure）。

服务注册发现Eureka + Ribbon

针对微服务注册发现场景，社区里头的开源产品当中，经过生产级大流量验证的，目前只有 Netflix Eureka 一个，它也已经纳入 Spring Cloud 体系，在社区中有众多成功应用，例如携程 Apollo 配置中心也是使用 Eureka 做软负载。其它产品如 Zookeeper/Etcd/Consul 等，都是比较通用的产品，还需要进一步封装定制才可生产级使用。Eureka 支持跨数据中心高可用，但它是 AP 最终一致系统，不是强一致性系统。



Eureka 是阿基米德洗澡时发现浮力原理时发出的惊叹声，在微服务中寓意发现

Ribbon 是可以和 Eureka 配套对接的客户端软负载库，在 Eureka 的配合下能够支持多种灵活的动态路由和负载均衡策略。内部微服务直连可以直接走 Ribbon 客户端软负载，网关上也可以部署 Ribbon，这时网关相当于一个具有路由和软负载能力的超级客户端。



Ribbon 是蝴蝶结的意思

服务配置中心Apollo

Spring Cloud 体系里头有个 Spring Cloud Config 产品，但是功能远远达不到生产级，只能小规模场景下用，中大规模企业级场景不建议采用。携程框架研发部开源的 Apollo 是一款在携程和其它众多互联网公司生产落地下来的产品，开源两年多，目前在 github 上有超过 4k 星，非常成功，文档齐全也是它的一大亮点，推荐作为企业级的配置中心产品。Apollo 支持完善的管理界面，支持多环境，配置变更实时生效，权限和配置审计等多种生产级功能。Apollo 既可以用于连接字符串等常规配置场景，也可用于发布开关（Feature Flag）和业务配置等高级场景。在《微服务架构实战 160 讲》课程中，第二个模块就配置中心相关主题，会深度剖析携程 Apollo 的架构和实践，预计 6 月份推出，欢迎大家关注学习。



阿波罗是希腊神话中太阳神的意思

认证授权中心Spring Security OAuth2

目前开源社区还没有特别成熟的微服务安全认证中心产品，之前我工作过的一些中大型互联网公司，比如携程，唯品会等，在这一块基本都是

定制自研的，但是对一般企业来说，定制自研还是有门槛的。OAuth2 是一种基于令牌 Token 的授权框架，已经得到众多大厂（Google, Facebook, Twitter, Microsoft 等）的支持，可以认为是事实上的微服务安全协议标准，适用于开放平台联合登录，现代微服务安全（包括单页浏览器 App/ 无线原生 App/ 服务器端 WebApp 接入微服务，以及微服务之间调用等场景），和企业内部应用认证授权 (IAM/SSO) 等多种场景。

Spring Security OAuth2 是 Spring Security 基础上的一个扩展，支持四种主要的 OAuth2 Flows，基本可以作为微服务认证授权中心的推荐产品。但是 Spring Security OAuth2 还只是一个框架，不是一个端到端的开箱即用的产品，企业级应用仍需在其上进行定制，例如提供 Web 端管理界面，对接企业内部的用户认证登录系统，使用 Cache 缓存令牌，和微服务网关对接等，才能作为生产级使用。在《2018 波波的微服务基础架构和实践》课程中，第一个模块就是微服务安全架构和实践相关主题，会深度剖析 OAuth2 原理和 Spring Security OAuth2 实践，欢迎大家关注学习。



Spring Security OAuth2 是 Spring Security 框架的一个扩展
服务框架Spring/Boot

Spring 可以说是史上最成功的 Web App/API 开发框架之一，它融入了 Java 社区中多年来沉淀下来的最佳实践，虽然有将近 15 年历史，但目前的社区活跃度仍呈上升趋势。Spring Boot 在 Spring 的基础上进一步打包封装，提供更贴心的 Starter 工程，自启动能力，自动依赖管理，基于代码的配置等特性进一步降低接入门槛。另外 Spring Boot 也提供 actuator 这样的生产级监控特性，支持 DevOps 研发模式，它是微服务开发框架的推荐首选。

REST 契约规范 Swagger 和 Spring 有比较好的集成，使得 Spring 也支持契约驱动开发 (Contract Driven Development) 模型。对于一些中大规模的企业，如果业务复杂团队较多，考虑到互操作性和集成成本，建议采用契约驱动开发模型，也就是开发时先定义 Swagger 契约，然后再通过契约生成服务端接口和客户端，再实现服务端业务逻辑，这种开发模型能够标准化接口，降低系统间集成成本，对于多团队协同并行开发非常重要。



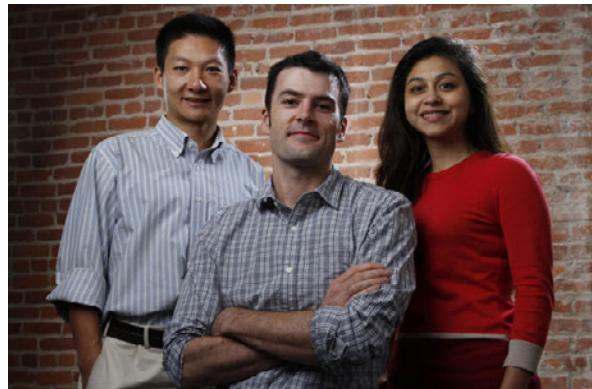
SPRING BOOT

监控反馈组件

数据总线 Kafka

最初由 Linkedin 研发并在其内部大规模成功应用，然后在 Apache 上开源的 Kafka，是业内数据总线 (Databus) 一块的标配，几乎每一家互联网公司都可以看到 Kafka 的身影。Kafka 堪称开源项目的一个经典成功案例，其创始人团队从 Linkedin 离职后还专门成立了一家叫 confluent 的企业软件服务公司，围绕 Kafka 周边提供配套和增值服务。在监控一块，日志和 Metrics 等数据可以通过 Kafka 做收集、存储和转发，相当于中间增加了一个大容量缓冲，能够应对海量日志数据的场景。除了日志监控数据收集，Kafka 在业务大数据分析，IoT 等场景都有广泛应用。如果对 Kafka 进行适当定制增强，还可以用于传统消息中间件场景。

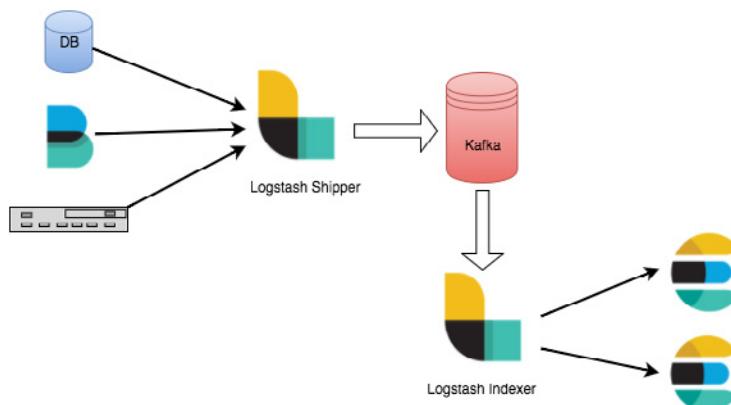
Kafka 的特性是大容量，高吞吐，高可用，数据可重复消费，可水平扩展，支持消费者组等。Kafka 尤其适用于不严格要求实时和不丢数据的大数据日志场景。



Kafka创始人三人组，离开Linkedin后，创立了基于Kafka的创业公司Confluent

日志监控ELK

ELK (ElasticSearch/Logstash/Kibana) 是日志监控一块的标配技术栈，几乎每一家互联网公司都可以看到 ELK 的身影，据称携程是国内 ELK 的最大用户，每日增量日志数据量达到 80~90TB。ELK 已经非常成熟，基本上是开箱即用，后续主要的工作在运维、治理和调优。ELK 一般和 Kafka 配套使用，因为日志分词操作还是比较耗时的，Kafka 主要作为前置缓冲，起到流量消峰作用，抵消日志流量高峰和消费（分词建索引）的不匹配问题。一旦反向索引建立，日志检索是非常快的，所以日志检索快和灵活是 ElasticSearch 的最大亮点。另外 ELK 还有大容量，高吞吐，高可用，可水平扩容等企业级特性。



ELK + Kafka 参考部署架构

创业公司起步期，考虑到资源时间限制，调用链监控和 Metrics 监控

可以不是第一优先级，但是 ELK 是必须搭一套的，应用日志数据一定要收集并建立索引，基本能够覆盖大部分 Trouble Shooting 场景（业务，性能，程序 bug 等）。另外用好 ELK 的关键是治理，需要制定一些规则（比如只收集 Warn 级别以上日志），对应用的日志数据量做好监控，否则开发人员会滥用，什么垃圾数据都往 ELK 里头丢，造成大量空间被浪费，严重的还可能造成性能可用性问题。

调用链监控CAT

Spring Cloud 支持基于 Zipkin 的调用链监控，我个人基于实践经验认为 Zipkin 还不能算一款企业级调用链监控产品，充其量只能算是一个半成品，很多重要的企业级特性缺失。Zipkin 最早是由 Twitter 在消化 Google Dapper 论文的基础上研发，在 Twitter 内部有较成功应用，但是在开源出来的时候把不少重要的统计报表功能给阉割了（因为依赖于一些比较重的大数据分析平台），只是开源了一个半成品，能简单查询和呈现可视化调用链，但是细粒度的调用性能数据报表没有开源。

Google 大致在 2007 年左右开始研发称为 Dapper 的调用链监控系统，但在远远早于这个时间（大致在 2002 左右），eBay 就已经有了自己的调用链监控系统 CAL (Centralized Application Logging)，Google 和 eBay 的设计思路大致相同，但是也有一些差别。CAL 在 eBay 有大规模成功应用，被称为是 eBay 的四大神器之一（另外三个是 DAL，Messaging 和 SOA）。开源调用链监控系统 CAT 的作者吴其敏（我曾经和他同事，习惯叫他老吴），曾经在 eBay 工作近十年，期间深入消化吸收了 CAL 的设计。2011 年后老吴离开 eBay 去了点评，用三年时间在点评再造了一款调用链监控产品 CAT (Centralized Application Tracking)，CAT 具有 CAL 的基因和影子，同时也融入了老吴在点评的探索实践和创新。

CAT 是一款更完整的企业级调用链监控产品，甚至已经接近一个 APM (Application Performance Management) 产品的范畴，它不仅支持调用链的查询和可视化，还支持细粒度的调用性能数据统计报表，这块是

CAT 和市面上其它开源调用链监控产品最本质的差异点，实际上开发人员大部分时间用 CAT 是看性能统计报表（主要是 CAT 的 Transaction 和 Problem 报表），这些报表相当于给了开发人员一把尺子，可以自助测量并持续改进应用性能。另外 CAT 还支持应用报错大盘，自助告警等功能，也是企业级监控非常实用的功能。

CAT 在点评，携程，陆金所，拍拍贷等公司有成功落地案例，因为是国产调用链监控产品，界面展示和功能等更契合国内文化，更易于在国内公司落地。个人推荐 CAT 作为微服务调用链监控的首选。至于社区里头有人提到 CAT 的侵入性问题，我觉得是要一分为二看，有利有弊，有耦合性但是性能更好，一般企业中基础架构团队会使用 CAT 统一为基础组件埋点，开发人员一般不用自己埋点；另外企业用了一款调用链监控产品以后，一般是不会换的，开发人员用习惯就好了，侵入不是大问题。

项目: Cat [切换] [常用]		【报表时间】From 2013-09-16 12:00:00 to 2013-09-16 12:59:59									
Machines:	[All]	101.84	[102]	108	[126]	3.128	6.145	6.37	6.64		
Type	Total Count	Failure Count	Failure%	Sample Link	Min(ms)	Max(ms)	Avg(ms)	95Line(ms)	99.9Line(ms)	Std(ms)	QPS
[:: show ::] Checkpoint	113	0	0.0000%	Log View	0	88489.3	4021.8	35592.1	35592.1	11732.9	0.0
[:: show ::] System	4,018	0	0.0000%	Log View	7.1	469587.7	1416.9	2458.0	304513.4	15837.9	1.3
[:: show ::] Dependency	636	0	0.0000%	Log View	3.2	5430	975.3	2112.0	5430.0	767.0	0.2
[:: show ::] Task	775	0	0.0000%	Log View	12.2	27520.4	173.4	373.0	27520.0	1281.6	0.2
[:: show ::] BucketService	62	0	0.0000%	Log View	0.1	1079.5	167.1	580.0	1079.0	221.8	0.0
[:: show ::] MetricAlert	53	0	0.0000%	Log View	28.1	130.9	49.1	115.0	130.0	25.0	0.0
[:: show ::] SQL	21,855	0	0.0000%	Log View	0	3268.7	8.4	24.1	929.3	55.5	6.9
[:: show ::] ModelService	581,564	142	0.0244%	Log View	0	1424	6.0	9.9	664.8	39.5	183.1
[:: show ::] URL	530,411	0	0.0000%	Log View	0.1	3844.4	4.4	6.5	540.7	40.7	167.0
[:: show ::] ABTest	424	0	0.0000%	Log View	1.5	22	4.0	6.1	11.5	1.9	0.1
[:: show ::] MVC	1,591,233	0	0.0000%	Log View	0	3844.1	1.4	1.0	433.7	23.6	500.9
[:: show ::] URL_Forward	530,411	0	0.0000%	Log View	0	3283.8	0.2	0.0	23.5	5.4	167.0
[:: show ::] Gzip	22,404	0	0.0000%	Log View	0	85.8	0.1	0.0	2.7	0.9	7.1
[:: show ::] Decode	22,403	0	0.0000%	Log View	0	155.6	0.1	0.0	1.9	1.2	7.1

CAT 的 Transaction 报表

Metrics 监控 KariosDB

除了日志和调用链，Metrics 也是应用监控的重要关注点。互联网应用提倡度量驱动开发（Metrics Driven Development），也就是说开发人员不仅要关注功能实现，做好单元测试（TDD），还要做好业务层（例如注册，登录和下单数等）和应用层（例如调用数，调用延迟等）的监控埋点，这个也是 DevOps（开发即运维）理念的体现，DevOps 要求开发人员必须

关注运维需求，监控埋点是一种生产级运维需求。

Metrics 监控产品底层依赖于时间序列数据库（TSDB），最近比较热的开源产品有 Prometheus 和 InfluxDB，社区用户数量和反馈都不错，可以采纳。但是这些产品分布式能力比较弱，定制扩展门槛比较高，一般建议刚起步量不大的公司采用。如果企业业务和团队规模发展到一定阶段，建议考虑支持分布式能力的时间序列监控产品，例如 KairosDB 或者 OpenTSDB，我本人对这两款产品都有一些实践经验，KairosDB 基于 Cassandra，相对更轻量一点，建议中大规模公司采用，如果你们公司已经采用 Hadoop/HBase，则 OpenTSDB 也是不错选择。

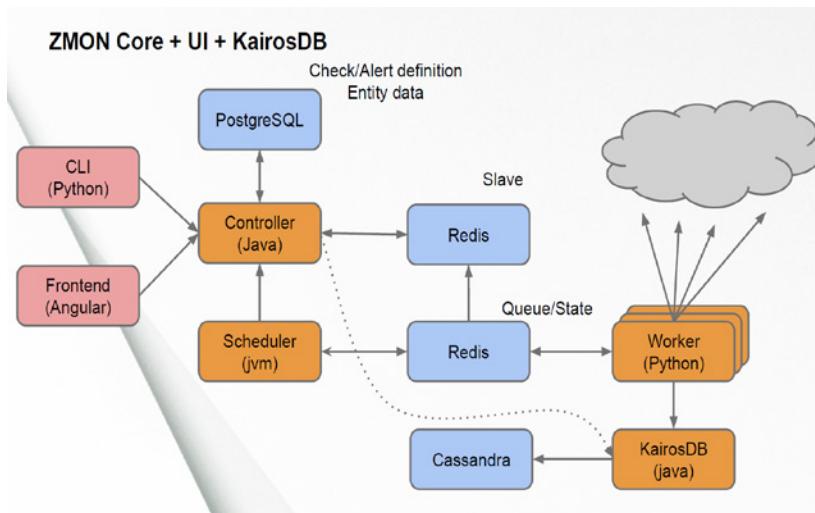
KairosDB 一般也和 Kafka 配套使用，Kafka 作为前置缓冲。另外注意使用 KairosDB 打点的话 tag 的值不能太离散，否则会有查询性能问题，这个和 KairosDB 底层存储结构有关系。Grafana 是 Metrics 展示标配，可以和 KairosDB 无缝集成。



Grafana 是 Metrics 展示标配，和主流时间序列数据库都可以集成
健康检查和告警ZMon

除了上述监控手段，我们仍需要健康检查和告警系统作为配套的监控手段。ZMon 是德国电商公司 Zalando 开源的一款健康检查和告警平台，具备强大灵活的监控告警能力。ZMon 本质上可以认为是一套分布式监控

任务调度平台，它提供众多的 Check 脚本（也可以自己再定制扩展），能够对各种硬件资源或者目标服务（例如 HTTP 端口，Spring 的 Actuator 端点，KairosDB 中的 Metrics，ELK 中的错误日志等等）进行定期的健康检查和告警，它的告警逻辑和策略采用 Python 脚本实现，开发人员可以实现自助式告警。ZMon 同时适用于系统，应用，业务，甚至端用户体验层的监控和告警。



ZMon 分布式监控告警系统架构，底层基于 KairosDB 时间序列数据库

限流熔断和流聚合Hystrix+Turbine

2010 年左右，Netflix 也饱受分布式微服务系统中雪崩效应（Cascading Failure）的困扰，于是专门启动了一个叫做弹性工程的项目来解决这个问题，Hystrix 就是弹性工程最终落地下来的一个产品。Hystrix 在 Netflix 微服务系统中大规模推广应用后，雪崩效应问题基本得到解决，整个系统更具弹性。之后 Netflix 把 Hystrix 开源贡献给了社区，短期获得社区的大量正面反馈，目前 Hystrix 在 github 上有超过 1.3 万颗星，据说支持奥巴马总统选举的系统也曾使用 Hystrix 进行限流熔断保护 [参考附录 2]，可见限流熔断是分布式系统稳定性的强需求，Netflix 很好的抓住了这个需求并给出了经过生产级验证的解决方案。Hystrix 已经被纳入 Spring Cloud 体系，

它是Java社区中限流熔断组件的首选（目前还看不到第二个更好的产品）。

Turbine是和Hystrix配套的一个流聚合服务，能够对Hystrix监控数据流进行聚合，聚合以后可以在Hystrix Dashboard上看到集群的流量和性能情况。



Hystrix在英文中是豪猪兽的意思，豪猪兽通过身上的刺保护自己，Netflix为限流熔断组件起名Hystrix，寓意Hystrix能够保护微服务调用。

结论

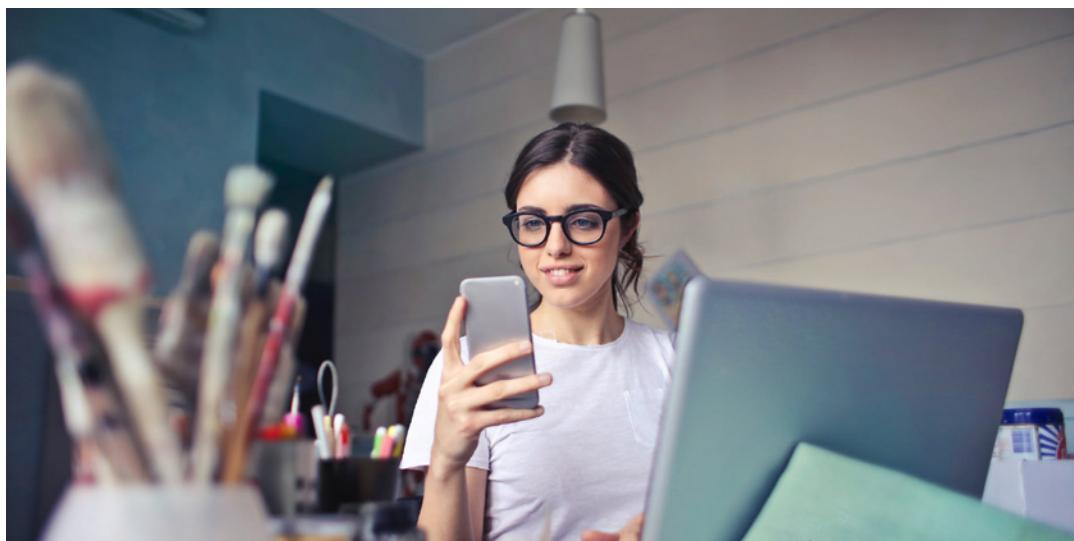
1. 技术栈没有好坏之分，只有适合一说。本文推荐的技术栈主要基于我个人的实践和总结，但是未必适合所有场景，毕竟每个企业的上下文各不相同。作为架构师你可以参考我推荐的技术栈，但不可拘泥照搬，你必须在深入理解分布系统原理的基础上，再结合企业实际场景灵活应用。
2. 本文推荐的技术栈主要面向微服务基础架构，也是《微服务架构实战160讲》课程要深度剖析的主题。在整个互联网基础技术平台体系中，还有消息，任务，数据访问层，发布系统，容器云平台，分布式事务，分布式一致性，测试，CI/CD等其它重要主题，这些是波波在2019第三季和2020第四季要陆续推出的内容，请大家持续关注。

附录

1. Interview: Adrian Cockcroft on High Availability, Best Practices, and Lessons Learned in the Cloud <https://www.infoq.com/articles/cockcroft-high-availability>
2. Netflix云端开源工具 [Hystrix](#): 曾助奥巴马竞选。

实现生产级的 Migrate 操作

作者 风爻，编辑 吕昭波



前言

在涉及到关系型数据库开发时、我们每次在线上操作数据库都需要找一个叫 DBA 的角色，时光荏苒，就这样过了一年又一年。一群不堪压迫，集 颜值、智商、动手能力于一身的程序员们发现，明明我们手里有操纵 DB 的能力，为什么要把命运交给别人主宰呢？

这群程序员做了一个伟大的决定！

让数据库的数据结构也能够通过版本追踪，这种操作，就被称为“Database Migrate”。本文将以 Flask-Migrate 作为基础蓝本，讲解如何实现生产级质量的 Migrate。



其实 Migrate 实现并不复杂，只要我们：

- 支持追踪每一次的数据库结构变更 【构建commit】
- 新开发者参与到项目中可以自动的应用变更到最新版本 【生效 commit】
- 回滚更新 【revert】

一个基础完备且健壮的 Migrate 就算实现了。常见的 Migrate 实现都是直接使用当前数据库来进行构建的。Flask-Migrate 也不例外。

在 Flask-Migrate 中，构建 commit 是通过文件的方式生成的。以一个简单的项目为例：

```
location:ekumen $ ll migrations/versions/ | grep -v __pycache__
total 16t table_name as 'Table List' from information_schema.tables t wh
lrwxr-xr-x  5 wangwenpei  staff   160  4 28 10:23 .
lrwxr-xr-x  10 wangwenpei  staff   320  4  8 14:04 ..
-rw-r--r--  1 wangwenpei  staff   904  4 17 18:29 02cb0d950c5d_.py
-rw-r--r--  1 wangwenpei  staff  1593  4 28 10:23 3fe5e6030507_.py
location:ekumen $
```

而生效的 commit，存储在 DB 的 alembic_version 中：

The screenshot shows two terminal windows for MySQL. The left window runs a 'select * from information_schema.tables' query, displaying a table list with columns like 'Table', 'Type', and 'Comment'. The right window runs a 'select version' query on the 'alembic_version' table, showing a single row with the value '02cb0d950c5d'.

```
[MariaDB [ekumen]> select * from information_schema.tables where table_name like 'alembic%';
+-----+-----+-----+
| Table | Type | Comment |
+-----+-----+-----+
| alembic_version | view | 生活小常识 |
| alembic_version | table | 我的牛刊 |
+-----+-----+-----+
4 rows in set (0.01 sec)

[MariaDB [ekumen]> select version
+-----+
| alembic_version.version_num |
+-----+
| 02cb0d950c5d |
+-----+
1 row in set (0.00 sec)

MariaDB [ekumen]>
```

在本例中，当前的 migrate 有两个，但是只生效了一个。当我们再次执行 Migrate 操作时，系统会对比当前目录和数据库中的记录，根据依赖关系顺序生效还未生效的 commit。

近十年来，随着 MVC、敏捷开发、DevOps 等等设计、开发模式从出现、探讨到成熟。Migrate 已经是一种不争的维护关系型数据库表结构的手段之一。

然而很不幸的是，虽然 Flask 有比较稳定的 flask-migrate 来支持这一功能。但是官方文档给出的做法确差强人意。

Using Flask-Script

Flask-Migrate also supports the Flask-Script command-line interface. This is an example application that exposes all the database migration commands through Flask-Script:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)
migrate = Migrate(app, db)

manager = Manager(app)
manager.add_command('db', MigrateCommand)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128))

if __name__ == '__main__':
    manager.run()
```

Assming the above script is stored in a file named `manage.py`, all the database migration commands can be accessed by running the script:

```
$ python manage.py db init
$ python manage.py db migrate
$ python manage.py db upgrade
$ python manage.py db --help
```

我们认为 Flask 生态最大的优势是极其简单、学习迅速，但是用好 Flask，做到生产级可用，真的不简单。Flask 不像 Django 用约束换取自由，Flask 本身可以认为是完全自由的。

不禁想起了社区的金句：Pirates use Flask, the navy uses Django

但是因为 Flask 自身的 Micro 风格，对等的带来一个巨大的优势，可以将 Flask 代码作为一个项目的内嵌项目来实现，特别是对于一些老旧项目的重构改造等，Flask 有着难以比拟的整合性。从 Migrate 角度讲，Flask-Migrate 拥有更广泛的使用场景。

生产级的 Migrate

我们认为一个生产级的 Migrate 的指标有：

基础指标

- 正确认别表结构变动并支持Migrate
- 支持回滚操作（可以是有条件限制的）
- 支持多个模块的Migrate

进阶指标

- 自动触发Migrate操作
- 新项目可以只复用表结构而无需复用Migrations

很不幸的是，flask-migrate 除了基础指标的 1 和 2 之外，均不能达成……所以，是时候从入门到放弃了？

当然不！要告诉你的是：这世上不存在开源且完全符合自身需求的程序。如果有，在编程这条路上，你离被淘汰不远了。

解决方案

Step1. 实现跨模块

分析

- Flask 框架风格追求 Micro。所以每个 App 风格都是尽可能少的引

用，于是每个App中的Model只会包含数据库中的一部分而不是全部。

- 其实和Migrate的思路产生了冲突。因为Migrate的目标是掌控整个数据库。如果我们有办法让Migrate识别出所有的Model，那问题不是就解决了？

```
78
71     for m in modules:
72         importlib.import_module(m + '.models')
73         pass
74
75     migrate_exec(migrations_root)
76     mig.init_app(app, app.db)
77     pass
78
```

核心代码

完整版代码传送门：<https://github.com/wangwenpei/fantasy/blob/master/fantasy/cli.py#L71>

Step2. 自动触发Migrate

分析

- 如果你对Migrate的概念比较熟悉。你就会意识到，Migrate操作始终是在当前版本代码载入生效之前执行的。
- 常见的思路是，我们通常会把这种操作构建到持续部署（continuous deployment）系统中。而基于Flask自由的灵魂，我们还可以构造出更加灵活简单的方式，只要我们保证在代码生效前执行就可以了。

核心代码

```

62     def smart_migrate(app, migrations_root):
63         """如果存在migration且指定为primary_node则执行migrate操作"""
64
65
66         db = app.db
67         if os.path.exists(migrations_root) and \
68             os.environ['FANTASY_PRIMARY_NODE'] != 'no':
69             from flask_migrate import (Migrate,
70                                         upgrade as migrate_upgrade)
71
72             migrate = Migrate(app, db, directory=migrations_root)
73             migrate.init_app(app, db)
74             migrate_upgrade(migrations_root)
75             pass
76         pass
77

```

完整版代码传送门: <https://github.com/wangwenpei/fantasy/blob/master/fantasy/init.py#L63>

Step3. 多项目无冲突复用表结构

使用场景

- 随着公司和项目的成长，举例：人事的变动，已有的项目A负责人开始接手新项目B。老项目A和项目B之间是完全独立的。而有一部分数据库表结构，希望只共用结构，而不共用数据。比如：消息发送记录表（table1）。
- 一个粗暴的办法，我们当然可以直接施展复制粘贴大法，从此分道扬镳。在风爻看来，这种策略实在低级，牺牲了太多特性。

好的策略

我们认为一个好的策略是这样的：

- 一方负责维护，其他方仅引用使用

这样一旦有商议通过的更新，所有人都会同步更新，项目组之间无需独立维护。

- 新项目只依赖需要的数据结构，创建全新的migrations文件
项目 A 的依赖关系没有任何变动，项目 B 的依赖关系为全新，互不影响。

完整版 Demo 传送门: <https://github.com/wangwenpei/shining-flask/tree/master/aivptr>

Migrate 风格 DB 设计原则

以下为风爻建议的一般通用性原则。

- 保持简单

你应该保持当前 DB 表结构的简单，不要做过度的设计，尽量不要预留字段。

特别是不太确定是否要添加的字段。

- 保持小巧

在生成 Migrate 时候，应该尽可能的发挥 DB 的原子操作性。

不要多个表的 Migrate 混在一起操作。

在一些极端的情况下，执行 Migrate 时 DB 连接会出现中断，

而利用好 DB 原子特性的 Migrate，可以将风险和损失降到最低。

- 保持单一的核心

关系型数据库作为一种经典数据库，作为核心数据库，数据存储应该单一化，突出关系特性。

做关系型数据库擅长的事情，比如订单、会员资料等是适用于关系型数据库的。

一个简单（粗略但快速的）的判断标准是，你使用到的数据是否需要事务特性。

- 拥抱多元的扩展

现代数据库相比 10 年前已经出现了极大的演进，仅开源 DB 据统计就有 100 种之多。

随着需求的不断更新，关系型数据库虽然大部分特定需求都可以满足，但相比专用数据库实现复杂，性能和效率也差强人意。

对于一些专用数据，风爻建议使用外围 DB 更合适，如 MongoDB，LevelDB，Cassandra 等。

随着云计算的成熟，引入多种 DB 成本是极低的。

作者简介



王文沛（风爻）开源爱好者。专注于Python社区，曾为数个明星社区贡献过代码，如Mongoengine、Django Plugins级项目等。对Javascript也有少量涉猎，曾为Pug（原名Jade）社区等贡献过代码。也为国内一些云厂商SDK捐助过代码或撰写过第三方独立扩展。对于开源生态：主张融合与协作，尽量避免重复造轮子。目前就职于UCloud，担任资深SRE专家职务。

QCon

全球软件开发大会2018

[上海站]

2018.10.18–20



7折 预售中, 现在报名
立减 2040元

团购享受更多优惠, 截至2018年7月1日



微服务架构实战

—— 8大核心模块精讲
打通架构师进阶之路 ——

160讲



杨波

拍拍贷研发总监、资深架构师
微服务技术专家

「 ￥299 / 160讲
扫码免费试看 → 」





架构师 月刊 2018年5月

本期主要内容：Stream：我们为何要从 Python 转到 Go 语言？Jeff Dean 在 SystemML 会议上的论文解读：学习索引结构的一些案例；谷歌开源针对 iOS 的可访问性测试框架；Node.js 10 带着 npm 6 来了！



深度学习器： TensorFlow程序设计

本书详细介绍了 TensorFlow 程序设计中的几个关键技术。



AI前线特刊： AI领域2017进展总结

AI 前线在 2018 年之初为各位读者奉上这样一本迷你书，涵盖了来自全球 AI 和大数据领域技术专家的年终总结与趋势解读，同时还有世界知名技术大厂的年终技术总结与趋势预测。



架构师特刊 范式大学

构建商业 AI 能力的五大要素；判别 AI 改造企业的 70 个指标；用最小成本 验证 AI 可行性；企业技术人员如何向人工智能靠拢？人工智能的下一个技术风口与商业风口。