漫谈中小企业研发技术栈

--- 中小企业技术领导者的自我修养

顺哥 著

这远远不是终点，而不过漫漫旅途中的一个小驿站。驻足于过去和未来交织的路口，我记录下那些曾经的思虑，也有幸能和你们分享，然后收拾行囊，继续奔向更遥远的未来！

数字签名者：SHUN
日期：2019.01.31
16:52:13 +08'00'

2019-01-08 雨夜

目录

1. 前言

1.1 缘起

2015 年的秋天，我通过猎头收到了两份回南宁工作的邀请，一份是去一个已小有规模的公司当技术总监，另一份是去一家智慧农业公司当高级软件工程师。第一份工作的内容跟云计算相关，而之前的三年我也是一直从事云计算基础平台方面的工作，因此看起来和个人经历、能力比较匹配，另一份工作邀请的公司目前还没有软件人员，但是有扩大队伍的需求，主要做大型农场的智能灌溉系统。

在从深圳离职之前，我的目标是再用几年，使自己成为云计算领域名副其实的专家，之前的三年，我在云计算领域付出了非常多的时间和精力，通读了云计算方面的各种图书资料，搭建过各种虚拟化云平台。计算、存储、网络是云计算的三个主要构成部分，每一部分我都曾经有较深入研究，延续这样的技术路线似乎也是合理的。但是聘请我当技术总监的这家公司，他们的研发团队在北京，南宁团队主要还是负责维护方面的工作，和自己的想法有点差距。而又转念一想，云计算主要还是巨头们玩的东西，大部分公司只需要了解怎么用好

云计算资源就好了，而进入第二家公司，我有机会从无到有去构建一个软件部门的研发体系，从无到有去设计一个包含前后端的应用系统架构，而我已有的云平台方面的技术经验，则会让自己知道怎么更好地去使用云平台的资源。因此，我选择了第二份工作，实现了从系统平台架构到应用架构的转型，就像三年前，我从深耕了五年的 Windows 客户端开发转向 Linux 服务端开发运维一样，也算是一个较新的开始。

然而由于人手不够，加上团队成本等方面的控制和高管对软件开发进度的要求，一些基础的环境都还没

搭建起来，就开始进行编码了。而一般我们说留到后面再弄的东西，往往都是再也不会弄了。这样做从短期看起来，似乎研发进度有保障，但是从长远来看，效率上会差挺多。没有持续集成、持续部署环境，只是写了个简单的 Hook 脚本，当开发人员上传网站代码到 SVN 时，这些代码会被自动同步到本地网站的根目录。对于编译型的项目，因为只有一两个项目，也是每次修改完代码，手动去执行 make 命令进行编译。

两年后，因为公司搬迁离家太远，我离开了这家智慧农业公司，去做了一阵智慧停车方面的项目。在做

停车项目期间，接触到一些不大不小的公司，了解到他们都还在用一些老旧的系统和架构，在软件架构设计上非常落后，高层领导虽想找一些高手来重构或者重新设计软件系统，但是苦于高手难找而且往往太贵。这让我看到了一些机会，我想如果将自己在平台和应用方面的架构经验沉淀下来，成为一套套方案提供给这些中小型企业，这样既能帮助他们解决技术上的问题，又能实现自己的技术理想，将自己所知转化为产品，同时获取报酬，岂不美哉。在这种想法的驱动下，我和几个志同道合的朋友共同创建了优笛瑞博，这个名字来自

“UTLab”的音译，全称为“Unite Talent Laboratory”。

随着这本小书的成书，优笛瑞博的研发体系也在逐步构建并不断完善。本书不会教你使用命令行一步一步搭建某个系统，不会带你进入技术细节，而是通过阐述我个人的所思所想，给你一些方向上的指引和资料推荐，因此，它更适合那些具有一定开发经验、有志在技术架构方向有更进一步发展并成为中小企业技术负责人的程序员、架构师等人士阅读。而对我自己而言，也是给自己多年来汗水付出的一个交代。好了，地图就在你面前，路需要你自己来走完，请开

启你的愉快旅程吧！

2. 研发体系

2.1 研发体系定义

我所理解的研发体系包括研发规范、标准、流程，规范即研发过程中需要遵守的规矩和准则，比如项目构建方式、文件组织方式、文档撰写方式，标准即做到怎样的程度能满足一般性需求，流程即需要以怎样的步骤去进行研发才能顺利完成项目。将规范、标准、流程糅合在一起进行工程化，再将工程化后的流程进行自动化。自动化并不包括在研发体系的概

念里面，不过自动化水平某种程度上体现了一个公司的生产力水平，不容忽视。

2.2 生产闭环系统

你可能听说过闭环系统，也可能并不清楚其实际含义，不知道怎么个闭环法，只貌似觉得这种说法很高大上。下面以工厂的流水线为例来阐述我们对闭环系统的理解。

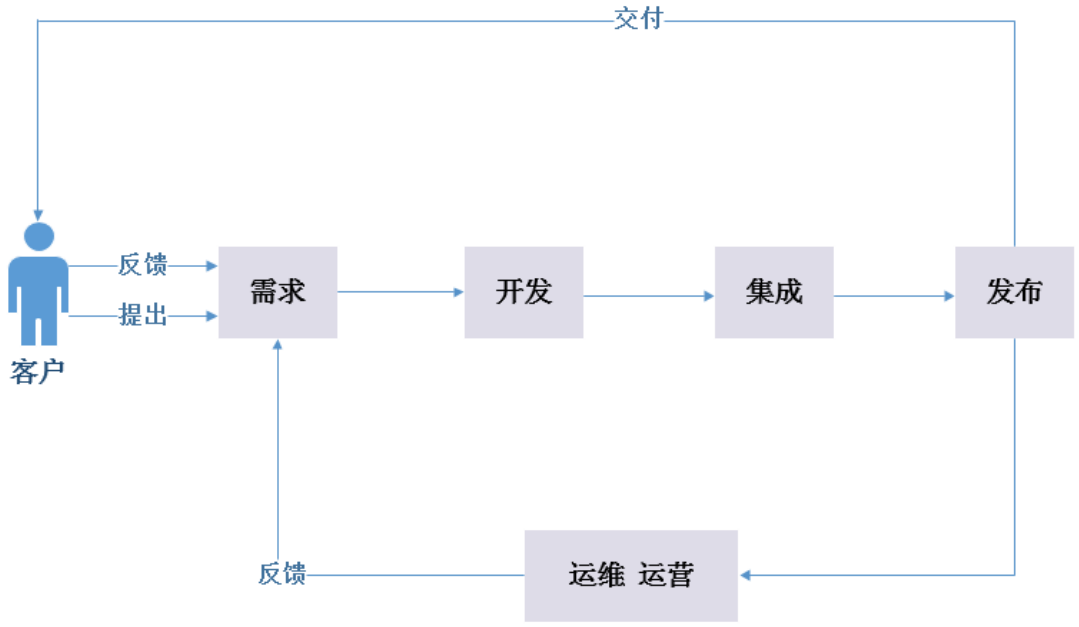
当我们构建了一个自动化的工程环境之后，这个环境就像工厂里的流水线，从获取原材料到各个环节的加工步骤都串联在了一起，生产者按

照这样的流水线各司其职，到达最后一个环节就会生产出完整的产品来交付给用户。而用户的反馈数据或运维系统的反馈可以说是一种新的需求，这些需求就类似于新的原材料又进入到生产线的原材料获取环节，最后又生产出了新的或者改进后的产品，形成生产线的一个闭环系统。当然在这个过程中，这个闭环系统本身也在不断的改进以适应新产品或新方案的进入。

为什么要闭环系统呢，你可能已经看出来了，其实是为了效率和反馈，这两样东西是促使企业进步的重要因素。没有效率意味很容易被淘

汰，没有反馈意味着难以修正自身存在的问题，因为根本不知道问题在哪里。

因此，我们的目标是要将研发体系打造成一条类似工厂流水线的生产线闭环系统，在软件研发中，这个闭环系统大致如下图所示：



2.3 研发体系成效

研发体系的具体成效也就是研发目标，我们认为目标包括研发效率、研发过程的可管理性和风险可控性，但是不包括你那一个亿的小目标。

研发效率也能体现一个企业的生产力水平，而研发过程如果可管理性不高，其实很难有很高的研发效率，因此，可管理性是高效率的必要条件。风险可控性是研发管理中需要特别注意的，因为风险不可控的话，一旦出现风险企业可能就失去造血或换血能力了。

3. 研发目标和资源

3.1 业务目标

你可能有遇到过这种情况，就是久不久会被 Boss 问一句：“我们技术团队人员需要调整吗？需要加人吗？”。对一个技术领导者来说，这时候一般都会先问问 Boss 在业务上面的想法，比如是否要开拓新业务，是否想把现在的业务规模扩大之类的，是否要缩短目前项目完成时间等等。公司的业务目标，也间接是研发团队的研发目标，研发目标不能脱离业务目标来设定，要不然就是耍流氓

了。比如你想让技术可以支撑怎样规模的业务系统，那技术团队的研发目标就相应要达到一个怎样的要求才能满足，比如性能、可用性等，这些就成为了技术团队研发目标的一部分。

3.2 研发资源

有了研发目标，接着就要评估完成这些目标需要怎样的技术实力，简单说就是需要大概多少个研发人员，每个人大概是怎样的技术背景，这些人如何进行协作才可以完成业务功能。这些研发人员的薪资大概是多

少，研发过程中需要怎样的设备支持，这些资源的总和就是公司需要准备的财力支持。作为技术负责人，如果你的老板不懂技术，那你更需要清楚的向他汇报这些情况，以便他做出下一步的决策。

现实中有很多不懂技术的老板经常会说，你们给我弄几个界面，每个界面也不用太复杂，实现功能就可以了，然而他却说不出具体的需求来，要实现哪些具体功能研发人员心理也没谱。这个时候你可能需要跟老板解释什么叫需求文档、需求评审之类的，技术团队经常不担心技术实现的难度，最心悸的往往是老板的一句

话需求，能坑死人啊。

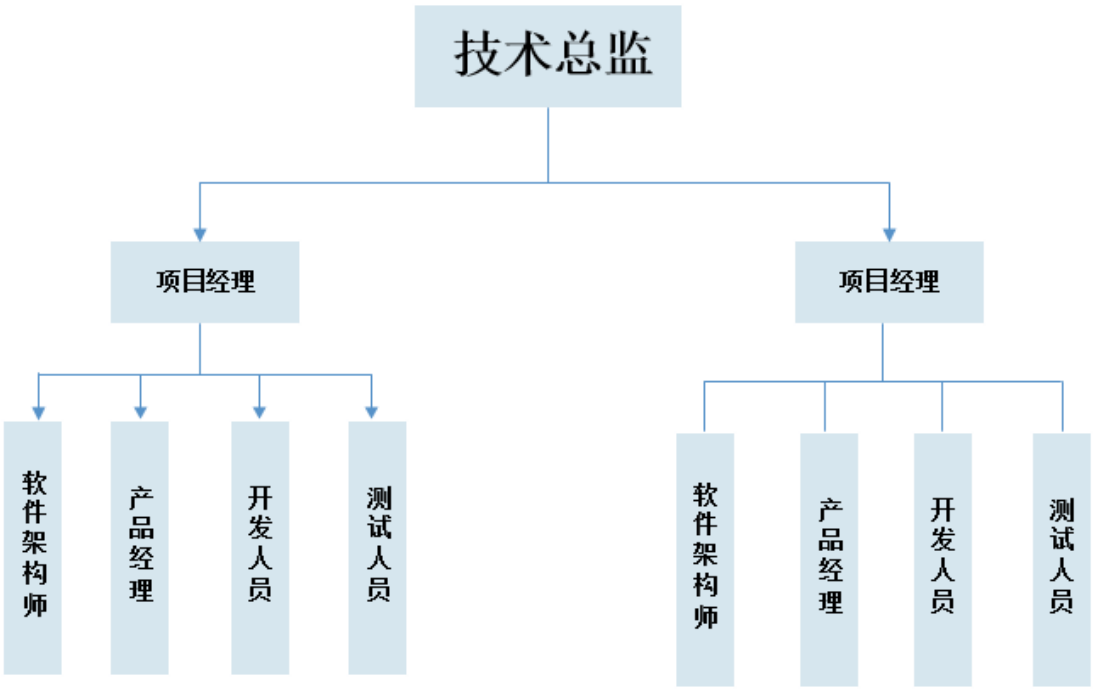
3.3 组织架构

一个团队要能高效率地完成任务，需要一个高效的组织架构。架构其实在我们的生活中无处不在，比如一个公司的老板，他本质上就是一个架构师，只不过不是我们常说的软件架构师，但他需要去做公司的组织架构，才能让员工各司其职，完成整个公司的目标。

研发团队中一般有一个技术负责人，若干项目经理，若干产品经理，一个或多个架构师，若干开发人员，

测试人员，这些人员之间是向下的一个树形结构。如果没有这种组织架构，那会乱成一团，要是技术负责人需要直接去跟进每个开发人员、测试人员的进度的话，在人员比较多的情况下是无法很好地做到的。高层管理人员只管中层管理人员，中层管理人员管理比如组长之类的，组长再跟进具体的基层员工的工作。说了这么多，感觉有点废话了，哪个公司现在不这么做呢，哈哈。

举个例子吧，下面的图就表示了一个简单的组织架构，切记，组织架构不是越复杂越好，而是越高效越好。



4. 技术方向选择

虽然这一章叫“技术方向选择”，但我在这里也是没有办法给各位某种具体的技术选择的，因为具体到每个公司每个项目，情况千差万别。因此，这一章我将企业研发中可能遇到

的相关技术方向先做一个罗列，然后阐述我个人在遇到这些问题时的思考方式，供您参考。

4.1 编程语言

在初学编程的那些年，本人也乐于关注网上的编程语言之争。很多争论刚开始可能还是摆事实讲道理，渐渐的就偏离了讨论的原本目的，进行骂娘式的口水战了。其实在顺哥看来，大部分人并不具备对编程语言品头论足的能力，当然你可以说自己就是喜欢某种语法、写法，这个没人可以否定你，谁晓得您是否属于骨骼清

奇、脑回路特殊的那个物种呢？另一部分人呢，假装看破红尘似地淡淡说道：编程语言不就是一种工具罢了。这么说确实也有道理，但顺哥认为编程语言对一个人的思维方式还是有影响的，因为使用了某种语言，可能导致代码中使用了某种编程范式，调用了某个充满淫技的类库，这些都会对你的思维有不一样的影响，只是我们可能没有意识到罢了。

在团队选择开发语言时，顺哥建议各位从多方面综合考虑，比如团队成员的技术背景、产品性能的需求、开发工具链的完整性、社区活跃状况、技术未来发展趋势等等。考虑这

么多因素的目的是，它们会对我们产品开发的成本和效果会有影响，比如选择 A 技术这个团队 2 个月能完成该项目，选择 B 技术同样的团队能在 1 个月完成，而且产品性能、维护性等方面差别不大，那么你应该选择 B 技术，加快产品上线速度进而降低开发成本。

再具体一点吧，如果你的任务是搞个官网，你熟悉 PHP，那你用 PHP 就好了，没听过这是世界上最好的编程语言吗呢，别听人家说用 Java 才高大上，你就用 Java，结果多花了 2 周时间还搞不好。听某某说现在流行搞前后端分离架构，你搞个简单的官

网，也非要前后端分离，这不是吃饱了撑着没事干吗，哈哈，跑题了，就此打住。

4.2 系统平台

系统平台严格地说一般包括软硬件环境，对于一个公司来说，系统平台一般指的是公司产品依赖的平台或者是应用服务在上面运行的平台。比如做嵌入式系统产品的公司，ARM + Win CE 可能就是他们使用的系统平台，而他们的云端应用可能运行在 CentOS 或 Windows Server 上，这些个操作系统和他们所购买的虚拟

机配置就统称为他们后端应用所采用的系统平台。

上一节中编程语言的选择，有可能也会影响到系统平台的选择，比如你选择了 C# .Net 作为开发语言，那你可能会选择 Windows Server 2008 作为服务器，而如果你使用 MySQL 和 PHP，那你可能走的 LAMP 路径，会选择 CentOS 系统。当你觉得无法选择使用哪种系统平台更好时，不妨从采购成本、相关技术人员招聘难易程度、建设周期、后续维护难易等角度去综合考虑。系统平台的选择也没有一套固定的方法可循，还是得具体情况具体分析，根据实际情况去做各

方面的评估后才能做出恰当的选择。

4.3 是否云化

百度CEO李彦宏曾经在互联网峰会上说云计算是新瓶装旧酒，后来估计他发现不是这么回事，才不得不赶紧在百度云的研发上加大投入以追赶其他巨头。而马云认为云计算的发展状况与阿里集团存亡相关，必须要做好，因此阿里云早早发力，目前两家云厂商的市场份额也相差甚远。如果说十年前很多人还在质疑云计算的话，今天几乎可以说大家已经全面拥抱云计算了。就本质而言，云计算

提供了一种新型的计算资源交付的模式。过去我们在物理机上安装虚拟机来模拟多台电脑，云计算也类似这么做的，只是它在一个大的集群里面去创建数以万计的虚拟机或者容器，然后通过网络交付给用户，用户只要在有网络的地方可以随时随地使用计算资源。按需使用、自助使用、无处不在的网络接入、弹性计算，这些都是云计算的显著特征。云计算典型的分层是 IAAS，PAAS 和 SAAS 三层，这些相关概念的解释网上汗牛充栋，而且云计算目前已经发展了十来个年头，顺哥觉得自己没有必要在这里对基础概念再做解释，想深入了解的

同学自行觅粮吧。

那么在公司自己搭建服务器和使用云平台产品的选择时，我们一般认为中小型企业基本不需要考虑自建的问题，因为自建成本往往会更高、周期更长，成本主要包括服务器成本、搭建时间、运维成本、安全质量保证等等，这些都不适合人、财、物都不充裕的中小企业去折腾，一般的中小企业应该专注于业务创新和应用产品的研发，把计算、存储、网络、安全等底层基础设施交给专业的云厂商更合适，当然，如果你的目标是建设一个提供基础设施的云平台，那另当别论了。

4.4 微服务化

微服务架构是近几年最为流行的一种架构模式，据说是从 SOA 衍生而来的，很多大牛级的人甚至调侃说自己区分不了这两种架构模式。有人开玩笑说，现在出去面试时，说不懂微服务架构，你自己都有点不好意思。IT 界大名鼎鼎的 Martin Fowler 和 James Lewis 大叔在个人博客上对微服务做了如下这么一个定义：

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and

communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

-- James Lewis and Martin Fowler

理解他们老人家的话之后，我们总结一下，微服务架构大致有以下几个特点：

1) 小，每个服务都很小；

2) 独，每个服务运行于独立的进程中，独立部署；

3) 轻，服务间通讯一般使用轻量级协议，如 HTTP；

4) 集，服务集中管理；

5) 多，服务可用多种语言编写；

那企业为什么采用微服务架构呢？这要从微服务的优势说起。微服务因为以上几个特征，使得这种架构具备很好的可扩展性，并且每个服务符合高内聚低耦合的设计原则，服务间通过轻量级协议如 HTTP 进行交互，使得这些服务可以使用不同的编程语言编写，只要接口遵循 HTTP 规范就好了。当某个服务出现问题时，只需要修改该服务的代码然后重新部

署，其他服务不用重新部署，避免了单体服务牵一发动全身的缺点。

虽然如此，企业一上来就采用微服务，可能并不是最合适的，如果业务规模和团队规模都比较小，采用微服务架构并不见得有什么好处，而且一般每个微服务部署在独立的 Docker 容器，这样就会使容器数量大大增加，每个容器的业务逻辑又比较小，白白给部署增添负担并耗费更多的内存和 CPU 资源。因此建议在业务量还不大，比如只需要一两台服务器承载业务时，可以先不考虑将服务分得太细，但是在设计时可以先考虑日后业务量上去时如何进行微服务划分

以及采用怎样的部署方案。

说到微服务，就要提一下 Spring Cloud。Spring Cloud 是基于 Spring Boot 的微服务开发框架，目前已经成为微服务事实上的标准，相对于国内出自阿里的服务治理框架 Dubbo，Spring Cloud 集成了更多组件，提供从服务治理到服务运维、监控等多个方面的解决方案。如果使用 Dubbo，可能需要自己去集成第三方组件，而使用 Spring Cloud，这些组件社区已经帮我们集成好了，知道怎么去使用就可以。近两年的 Service Mesh 大有取代 Spring Cloud 之意，但是目前成熟度还有待提高，比较有名的如 Istio，下一

节中我们会讲到。如果项目采用 Java，那么微服务架构建议选择 Spring Cloud 或 Dubbo，其他语言也有一些开源的微服务框架，比如 Python 方面有 Nameko，本人曾经对这个框架的源码进行过剖析放到技术博客上，Go 社区有 Go-micro 等等不胜枚举，有兴趣了解和学习更多的童鞋，我建议你关注 GitHub 上的项目，在这里：awesome-microservices，里面列出了基于各种编程语言的在微服务架构方面的一些基础组件。

4.5 网格服务

网格服务也叫 Service Mesh，是近两年兴起的一种架构模式，大有取代现有的微服务架构之势。而实际上在我们看来，网格服务是对现有微服务架构的补充和加强，将微服务系统中一些基础又普遍性的功能特性标准化，然后集成到微服务系统的基础设施中去，变成微服务架构基础设施的一部分。比如 Kubernetes，可以看作是微服务应用架构的一种基础设施，网格服务将一些功能标准化后，可以随着 Kubernetes 一起部署，也就是只要部署了 Kubernetes 环境，这些

标准化组件就已经在运行了，用户编写的微服务应用相应地会去掉这些已经标准化在 Kubernetes 中的功能，减轻微服务应用开发人员的工作量。这就好比 TCP/IP 标准确立前，应用开发人员如果进行网络编程，那么需要根据 TCP/IP 协议去处理数据，而当这些协议标准化后，对该协议的实现也成为操作系统或者基础库的一部分后，编写网络应用时你只需要调用这个库的接口去获取数据，而不需要自己去解析协议本身，解放了程序员并大大提高生产率了有木有。

在网格服务方面，Istio 名声比较大并且目前发展比较好，其背后有

IBM 和 Google 等这样的大厂背书也是原因之一。顺哥在这里只是列出这些概念，更多内容大家还是得去阅读官方文档，也可以去看 QCon 大会上关于这方面的讲座，PPT 都可以免费下载。国内在 Service Mesh 方面有尝试的有华为、新浪微博、网易等等，并且这些公司的相关技术人员都在各种大会上进行了分享，作为准公司技术负责人的你，需要具备自己去检索资料的能力，难道不是吗？我仿佛看到你在点头了。

4.6 云原生应用

近几年云原生这个概念也炒得比较火热，云原生应用的意思就是这些应用天生就是为云环境准备的，这些应用利用了云计算平台的某些特征，充分发挥云平台的技术优势，成为云计算平台的原住民一般，少了水土不服的潜在隐患。我们为这些云原生应用采用的架构模式就叫云原生应用架构，这种架构也是基于云平台的，是因为云的出现而产生或者为了适应云基础设施而从其它架构模式演化过来的。

云原生应用架构的特性，包括比

如微服务、十二因素应用程序、自助基础设施、基于 API 的协作、抗压性。关于云原生应用架构的更多内容，大家可以去看顺哥给的这些链接，都已经写得很好了，因此没有必要在这里再长篇累牍浪费彼此的时间。关于云原生架构，Pivotal 的技术产品经理 Matt Stine 曾经写过一本书，英文名全称是：《Migrating to Cloud-Native Application Architectures》，中文翻译版可以看这里：迁移到云原生应用架构。

在是否云化那一节中，我就说过目前云化的趋势不可逆反，因此建议大家在进行应用架构设计和开发时，

多从云原生应用架构需要具备哪些特性的角度去考虑问题。

4.7 大数据技术

说到大数据，就不能不提 Hadoop，Hadoop 是雅虎发起的开源项目，已经成为离线大数据分析的重要项目，它的灵感源自于 Google 三驾马车之二的 GFS 和 MapReduce 论文。Hadoop 是一套用于在由通用硬件构建的大型集群上运行应用程序的框架，它实现了 Map/Reduce 编程范式，计算任务会被分割成小块运行在不同的节点上，它还提供了一款分布式文件系

统（HDFS），类似于 Google 的 GFS。由于 Hadoop 处理的数据是存放在基于磁盘的文件系统上，使得它用来分析数据时效率不是那么的高，于是后来又出现了基于内存的 Spark，以及流式计算 Storm，因为基于内存，这些大数据开源框架能够更高效实时的分析处理数据。

对于一个中小企业来说，有的研发团队只有几号人，老板让你搞时髦的大数据，这时候你该怎么办呢。去研究使用 Hadoop 还是 Spark 还是 Storm，然后跟老板汇报说要买多少台服务器来搭建大数据系统，需要多少人进行系统运维等等吗？这样的话，

估计老板听了你的预算，还是会愁眉不展，没想到搞大数据那么费钱啊。是的，顺哥认为一般的中小企业根本没有能力组建大数据团队，除非是那种技术驱动型的公司，或者创业方向就是大数据产品的，普通的中小企业尤其是传统的企业想进行互联网+转型的，一般不具备自建大数据团队的技术和资金实力。

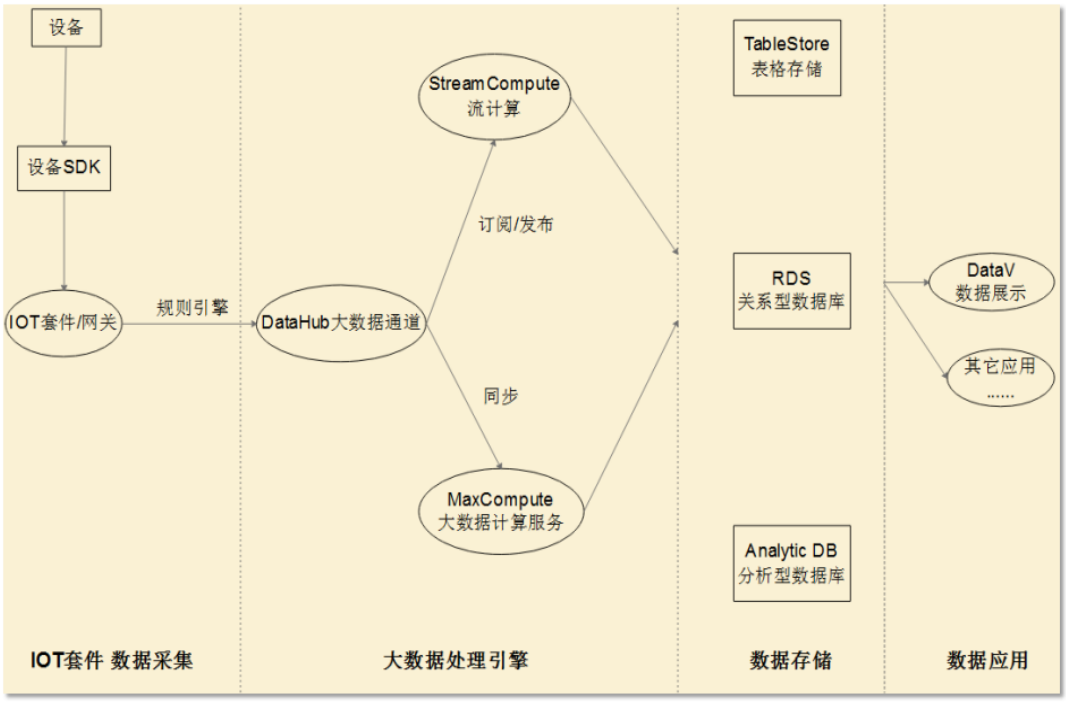
关于大数据，有一个广为流传的幽默段子是这么说的：Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.

【大数据就像青少年性爱：每个人都在谈论它，但没人知道怎么做，每个人都以为别人正在做，因此所有人都声称自己也在做】

实际情况也差不多如此，因此，顺哥建议，作为中小企业的技术负责人，你这时候应该将目光转向大的云计算平台，这些云平台一般都有大数据组件和解决方案，只需要半个小时，你就能购买到一个可用于大数据分析的计算和存储集群，而这些成本我相信不会比你自建的成本高。因此我们个人认为，一般的中小企业，能充分利用云平台提供的各种产品组件构建自己的研发环境，是非常关键

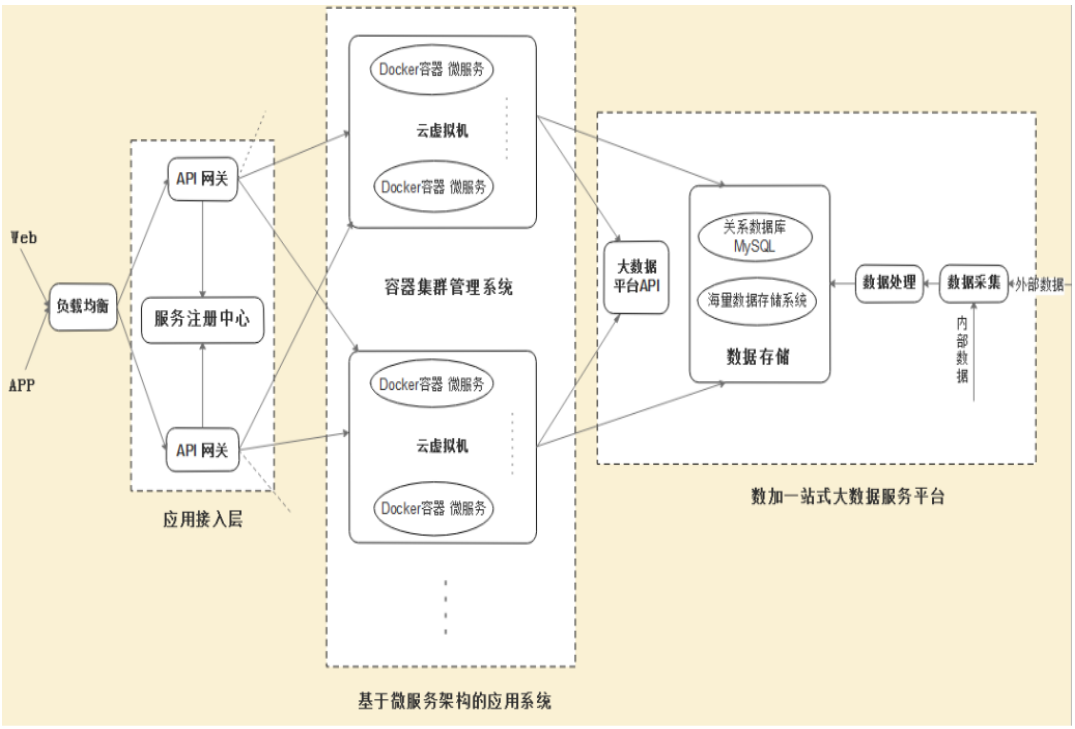
的。

顺哥之前所就职的公司基本都是使用阿里云的产品，阿里云的产品线丰富是一个原因，还有就是大平台的品牌可信度一般会比较高，因此阿里云也成为很多公司采购云产品时的首选。下面是本人几年前在使用阿里云产品进行物联网项目的架构设计时画的一张图，仅供参考。图中的各种组件都是直接从阿里云购买，只需要你懂得如何去搭配使用，而要懂得如何搭配使用，进一步需要你先对每个组件的功能和使用场景有一个大概的了解。



从上面这张图看，如果我们搭建的是一个物联网平台，那么可以利用阿里云的这些组件快速搭建数据平台，包括数据采集、分析、存储和展示，当然，更进一步的分析可能需要开发人员去编写应用服务来进行，也就是对应到上图中右边的“其它应

用”所表示的内容。那么如果把这部分展开，我们还可以得到下面这样一张图，是基于微服务架构的一个应用系统，这个应用系统将对数据进行处理，然后通过大数据 API 或者直接写入等方式将数据分析结果存储到大数据存储系统，这个存储系统就是阿里云一站式大数据处理平台数加，之前也叫作 ODPS(Open Data Processing Service)，即开放数据处理服务。



这张架构图仅供你参考，先不做过多解释。

4.8 人工智能

人工智能(AI, Artificial Intelligence)在未来将成为生产力，彻底地改变这

个世界，虽然这项技术几十年前就有人研究，但随着这些年深度学习领域的迅猛发展，AI 已经走进了普通人的生活。各种人脸识别、语音识别设备也层出不穷，在这里不得不提的就是 Google 研发的 AlphaGo、微软小冰、百度大脑、阿里云 ET 等等，是这个领域非常有代表性的产品。

目前来看，AI 已经不是什么炒作了，而是实实在在地改变一些东西，所以我认为，无论是现在还是在遥远的未来(唱起来呀)，AI 会深刻地改变这个世界是毫无疑问的。有人担心人工智能会取代人类，会威胁人类，而在顺哥看来也许是杞人忧天吧，人类

有必要研发一个东西来打败自己吗？当然，极端的情况下，就是掌握这个领域精尖技术的一部分人是坏人，他们不计后果地使用这些技术，那有可能会给人来带来灾难，但人类可以未雨绸缪，建立一个安全稳健的社会架构，防止这种情况的发生，把它扼死在襁褓中，这也是必须的。你想想，AI 的目的是什么呢？应该是为人类服务的吧，而不是取代人类。

本人在人工智能方面没有研究，但也零星学过一些，从初学者的角度给想入门的同学推荐一些学习资料吧。首先是本人博客的文章里边的一些链接，这些文章相对通俗易懂，在

这里：机器学习资料，还有就是大名鼎鼎的吴恩达的课程，虽然本人没有看过，但如果你去网上找资料的话，相信很多人都会给你推荐他的课程，关于他的经历网上也有很多，爱八卦的请自行前往搜索。GitHub 上也有相当多的学习资料，随便搜一下就能找到，搜索资料是程序员的必备技能，善于搜索资料使你能够更快地成长。

很多同学看到哪里火热就想往那个方向涌，因此会问到，既然人工智能这么有前途，我是不是应该赶紧去学习呢？在顺哥看来，人工智能的学习需要比较好的数学基础，人工智能是一个专业性非常强的领域，一般

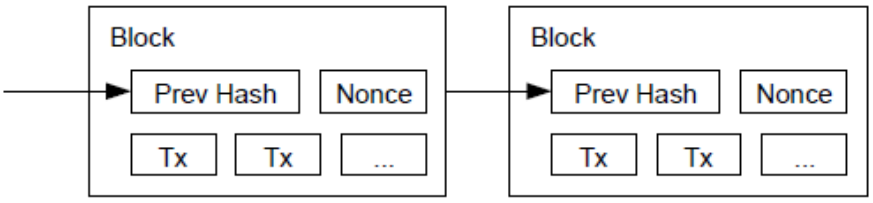
来说，这方面的研究留给那些专注于这方面的公司，比如商汤科技、旷视科技等等，大部分的从业者，只需要了解这些技术发展到大致哪个阶段，然后会将这些技术集成运用到自己公司的产品里面就够了。当然，如果你意志坚定，对人工智能技术又特别感兴趣，那你去研究吧，或者加入这方面的独角兽公司都可以，像你这种骨骼清奇的人士，做什么别人拦都拦不住的，当然你也是对的。如果只是听说这方面火，能赚钱就跟风去学习，你还是先歇歇，然后该干嘛干嘛去吧。

顺便推荐一下李开复老师的书，

叫《人工智能》，而且开复老师还出了一系列视频，叫《想象视频》，本人都没来得及看。

4.9 区块链

区块链的数据以区块的形式存在而得名，有人调侃说它是史上最慢的分布式数据库，但区块链技术在这两年发展得如火如荼，著名的以太坊就是区块链应用的开发平台，根据 CSDN 的调查，基于以太坊开发的人员数量在迅速增加。大家熟悉的比特币就是区块链技术的一种应用，比特币区块的存储形式如下图：



上面图中，每个 Block 就是一个区块，存放的是交易数据相关信息比如时间戳，并且有个指针指向了前一个区块，其实这个指针存放的是前一个区块的哈希值，通过加密算法来生成。如果有人修改了某个区块的值，那么其哈希值也会被改变，指向它的区块就要更新哈希值，这样会引起连锁反应，使得后面所有的区块信息都需要修改。又因为有基于工作量证明的共识机制，超过 50%的算力都同意

修改一个区块，这个区块的修改才能成功，因此区块链的不可篡改性就是因为篡改成本极高而得名，因为你要掌握 50%以上的算力，在这种分布式网络上掌握 50%的算力是极为困难的。

综上，区块链技术的特点是去中心化、不可篡改，分布式保证了去中心化，数字签名保证不能伪造交易数据，工作量证明保证区块的信息不被篡改，最终保证区块链每一条交易数据都是可信的。需要更深入了解的，建议去读一下比特币的设计白皮书。

八卦时间到了。据说著名的天使投资人徐小平曾在微信群要求他所

投资的公司全面拥抱区块链，要毫不犹豫地。在他看来区块链是一种技术革命，会彻底改变这个世界的生产关系。后来都怎么样了，我们不得而知。不过如果你现在看到有人要发币了，建议你提高警惕。技术是中立的，区块链是无辜的，只是很多人利用这种技术和噱头来圈钱，等玩不下去了，留给用户只剩一个公告：再见，我们跑路了！

区块链技术目前还没有到需要选择哪种的时候，不过基于以太坊开发的人数应该是最多的，可以了解一下以太坊的开发生态。对于真正想学习区块链技术的同学，顺哥在这里又

不厌其烦的推荐，推荐你看看 GitHub 这个系列：《awesome-blockchain-cn》。还有陈皓在极客时间专栏里面关于区块链的几篇文章，他发表了自己对区块链技术及其相关行业的诸多个人看法，看完你可能也会觉得他的思想还是相当深邃的。

5. 技术架构设计

设计一个中大型后台系统的技术架构时，这个技术架构需要具备怎样的一些特质，可能会用到哪些云产品或组件，这一章我将根据自己的经验和所思所想，逐步给你大部分的答

案，不能保证面面俱到，错漏之处也在所难免，仅供参考，还望海涵。

5.1 可扩展架构

可扩展性对现今的软件系统来说，怎么强调都不过分，因为如果系统不能扩展，那意味着业务量上来时可能随时面临重新整一套系统的风险，而当你弄出来一套新系统时，原有的客户早已经离你而去了。从广义上来说，可扩展包括两个维度，分别为纵向扩展和横向扩展，但我们一般强调更多的往往是横向扩展。纵向扩展很简单，比如一台服务器内存不够

用了，我们再买几块内存条插上去，使单个服务器能承受更多的业务负载。横向扩展是原来的服务器不需要添加硬件部件比如内存条，而是添加一台服务器，由一台服务器变成两台，再有两台变成三台等等。

那为什么更应该强调横向扩展呢？聪明的你估计已经想到了，因为纵向扩展很快就遇到瓶颈了，你不可能一直给旧机器添加内存条吧，况且服务器的内存槽也有限制，而添加新的服务器，只要购买新服务器并且分布式集群软件能够支持这么多台服务器就行。目前我们说的貌似都是硬件方面的横向扩展，而软件方面的横

向扩展是跟系统架构和应用架构有关。也就是你添加新服务器了，分布式集群软件能够将这些服务器联合起来提供物理资源，公司开发人员编写的业务应用也能充分利用集群的资源为客户提供业务服务。这么说来，硬件、分布式集群软件、应用服务这三个因素都具有可横向扩展性，整个系统的可扩展性才有可能被称之为高可扩展性。

问题又来了，这三个因素又是如何才能扩展呢？硬件就不用说了吧，一个字：买，但是硬件的增加又需要分布式集群软件的支持，集群软件呢，需要公司自己研发或者鼓捣开源

的东西，使得集群软件能够支持硬件设备的不断增加，比如阿里云的飞天集群，很早以前据说是突破 5000 台服务器，因此集群软件也不是可以支撑硬件的无限扩展的。最后到应用服务，因为底层环境已经可以扩展了，应用运行的模式必须也能随着集群资源的增多而横向扩展，比如一个应用原来启动一个容器来运行，现在可以启动多个容器来运行多个副本，然后还要保证各个端口使用没有冲突，整个系统还是一个完整的可对外正常提供服务的系统。对于非云计算厂商的中小企业来说，一般主要考虑应用层的扩展问题，而前面两个维度的

扩展一般云厂商都已经解决了，只是需要我们去了解如何使用底层资源以及使用过程中有哪些限制。

对可扩展应用架构来说，还有一个非常重要的思想，就是尽量将所有应用设计成无状态的，有状态的内容都存储到数据库或者其它第三方存储空间，这样在迁移应用时才不会引发状态不一致的问题。典型的例子就是用户 session 问题，如果 session 保存在内存中，用户第一次登录时被被负载均衡器转发请求到某台服务器，session 就被保存在这台服务器的内存中，后面访问时可能就被转发到其它服务器上，这时候从内存就获取不

到用户 session 了，就会出现将用户重定向到登录界面的诡异现象。在这里，session 就是状态数据，应该存储到数据库或者其它共享存储空间中。当然，我们也可以设置用户亲和性，比如 IP Hash 方式，将同一个用户的请求始终转发到同一台服务器，这种方法属于负载均衡器的一项功能，这里就不展开讨论了。

5.2 高可用架构

高可用架构要求服务不中断或者中断频率和时长都在一个很小的范围内，因此就涉及到可用性的度

量。比如我们说一个星期 7 天，总时间按秒算的话是 24*7*60*60=604800 秒，在这么一个时间段内，如果服务只发生一次中断，并且中断时间为 10 秒以内，那么我们可能会把这个架构称为高可用架构。当然这里只是举个例子，业界更常用的衡量方法是 SLA(Service Level Agreement)，即服务等级协议，一般以百分数表示，按全年来算。很多服务商都声称自己的服务 SLA 为多少个 9，比如五个 9，那么可用性为 99.999%，这样全年时间乘以可用性百分比，即可得到该厂商承诺的一年内服务可持续提供的时间长度。我们可以算一下在五个 9 情况

下，一年时间内服务中断时间为24*365*60*(1-99.999%)=5.256分钟，如果一年内服务中断时间到了6分钟以上，我们就可以说改服务厂商违反了SLA协议。

那么如何进行高可用性架构设计呢？我们应该马上想到避免单点故障(Single Point Of Failure)，具备高可用性的系统中不允许单点存在，每个组件都应该有冗余，因此一般一个组件会被部署到多个节点上，节点之间通过心跳(HeartBeat)之类的方法来感知对方，整个系统是一个分布式集群系统。冗余具体来说比如物理服务器有冗余、磁盘存储有冗余、应用服

务有冗余，这样在某个组件崩溃或者失效时，另外的服务可以继续提供服务，当然如果冗余部分也崩溃时，那服务就无法提供的了，比如机房的大片断电。我们说的冗余其实也是相对的，绝对的冗余不存在，世界这么大什么情况都有可能发生对不对，这也是我们上面提到的 SLA 存在的部分原因吧，因此高可用性架构是在系统中消除了单点，每个组件都有冗余的情况下的尽力而为的系统和应用架构。

还有就是系统容量的设计，其实也就是系统的可扩展性，因此这么来看的话，高可用性的系统一般来说也是可扩展的，当请求激增达到容量上

限或接近某个值时，可以自动进行扩展，这样才能保证系统的高可用性。

这里还要介绍一下分布式领域的 CAP 理论，CAP 即 Consistency(一致性)、Availability(可用性)、Partition tolerance(分区可容忍性)。CAP 理论认为，在分布式系统中，因为网络故障不可避免，系统要么选择可用性，要么选择一致性。如果你选择了一致性，因为网络故障已经分区了，那么系统不可用，如果你选择可用性，那么也因为网络故障导致分区已经发生，继续使用系统的话会导致数据无法同步而出现不一致的状况。CAP 理论经常被人误解为要么选择 C 要么选

择 A，实际上这是在网络故障出现的时候，如果网络正常，CA 可以同时得到满足。在实际中我们应该根据业务需求来选择系统是偏向 C 还是偏向 A，然后在出现相应的状况时根据我们的选择给用户返回合理的结果。

5.3 高并发架构

高并发架构在顺哥看来跟高性能架构差不太多，只是强调了并发数，即这种架构需要在请求数量非常高的情况下依然坚挺。那么在考虑高并发架构时，同时要考虑可扩展性，因为高并发系统往往面临着随时扩

展的要求。

为了应对高并发请求，在架构时应该思考每个请求在整个系统中的调用链路，尽量将请求拦截在上游，将数据存放在离用户最近的地方(比如浏览器缓存)。比如我们常见的秒杀系统是典型的高并发架构，那么在一件商品库存量只有几百的情况下，将几万个请求下放到后台数据库是没有必要而且是对计算资源的极大浪费。这时候我们可以把前端的几百个请求放到队列(消息队列是系统解耦的利器)里面，然后另外一个消息处理系统对这些请求进行后续处理(也就是异步处理)，其它多余的几万个请求

因为已经没有商品库存了，没有必要放进队列，直接返回秒杀失败。

在进行了以上处理以后，如果系统还是无法处理这么多请求，那么可能是数据库的读写瓶颈，可以进行数据库设计上的优化，可以加入缓存系统，如 memcached 或者 redis，或者确实到了需要添加服务器资源的时候了，也就是可扩展架构需要做的事情，因此，高并发架构往往首先是一个可扩展架构。当然，实际情况可能不是这么简单，这里只举个例子，各位同学在遇到问题时要根据具体情况去分析瓶颈所在，寻找恰当的应对策略。

5.4 高性能架构

在聊高性能架构之前，首先我们得明白什么是高性能。在我们看来，高性能主要体现在这两个方面：

1) 吞吐量大，单位时间(一般是每秒)内可以处理的请求数；

2) 系统延迟低，一个请求从发起请求到获取回复数据的时间足够短；

为了更形象一点，我们就拿 API 接口为例吧，比如一个远程接口 getUserName，满足同时十万个调用

请求，并且全部在 0.5 秒内完成数据返回，我们就说这个接口的性能达到了一定的水准，具体是否高性能要看各个公司对自己业务的要求可能有不同的看法。

上面的吞吐量和系统延迟，在服务器资源一定的情况下会相互制约，因此在我们看来，高性能架构必须在某个维度上达到一定高度时，另一个维度同时保持在较高的水准。简单来说吧，如果吞吐量每秒一百万，系统延迟达到一分钟，那么客户端在请求一分钟之后才有数据返回，一般来说这种系统不会有正常人使用，这样看来这个高吞吐量并没有什么卵用。同

样的道理，如果系统延迟在 0.1 秒，但吞吐量不过百，也许能满足特定场景需求，比如请求数很小的情况，但我们一般不会说这是一个高性能架构的系统，因为一旦请求量上来，系统不是挂掉就是延迟增加了。因此，需要以上两个因素同时达到一定高度才能称之为高性能。

前面说的其实只是服务器端接口的高性能，作为一个系统来说，我们还要包括前端的内容，因此高性能架构需要从前端到后端进行整体设计，在后端接口能快速响应前端请求的基础上，还要进行更精细的数据传输设计，比如返回怎样的数据信息在

满足前端数据需求的同时可以减少流量消耗，怎样的数据结构有利于前端快速解析，因为这些都会影响到前端的用户体验。前端是用户直接触及的部分，也就是给到用户的最终产品，因此良好的用户体验非常有必要，而后端对普通用户来说，是无法直接感知的。

5.5 数据库架构

很难想象现今哪个系统没有用到数据库，因此数据库的重要性无需多言。在数据库架构设计过程中，一般从以下几个维度去考虑。

1) 可用性，也就是高可用的问题，需要冗余机制；

2) 可扩展性，数据量暴增时，如何扩展数据库以保证满足需求；

3) 一致性，当使用分布式数据库时，主从数据库的同步方式是怎样的；

4) 性能，在前面几个维度都考虑好的情况下，数据库的读写性能如何，比如可能需要读写分离等方案来满足性能需求；

5) 灾备方案，如何进行数据备份和恢复，极其重要，数据可能是一个公司的命根子，是生产资料；

架构做好了，就需要设计数据结构，也就是数据库表及其之间的关联。可以说一个系统的数据结构反应了这个系统的本质属性，因此设计一个能真实反应系统中各种实体及其之间关系的数据结构是应用系统健壮的关键因素之一。业务应用都是基于这些数据结构进行各种操作，因此需要数据结构的设计者对业务流程有非常清晰的理解，然后进行抽象，将这些实体映射到系统的数据结构中去。

数据结构设计好了，还要考虑索引的问题，索引的用处不需我多说

了，具体怎么使用索引，网上也有不少资料，这里本人推荐极客时间上丁奇老师的 MySQL 专栏，还有数据库大牛何登成的技术博客上相关文章。

5.6 安全架构

安全架构是每个企业都应该非常重视却往往被忽略的一种架构，这也许是源于很大一部分人不见棺材不掉泪的本性，又或者是不想花费时间和金钱去构筑安全堡垒等等原因。

那企业又怎么建设安全架构呢，这真的比较难，首先企业中需要有安全意识比较强烈的人，而且他在企业

中要有一定的话语权，或者他常常能说服 Boss 去开展一些事情，这样的人才能推动企业的安全架构建设，毕竟做什么事情都是需要血本啊。从技术上，还需要一些对安全技术有一定了解的人员，对基本的攻防姿势有一定程度的熟悉，这样才能从攻击者的角度去思考，有助于构建真正安全的系统。建议安全相关人员阅读吴翰清(江湖人称道哥)的书《白帽子讲 Web 安全》，关于道哥的一些传奇经历网上可以轻易搜到。

在这里，本人墙裂建议每一位对安全有兴趣的同学去读一读 Google 的安全架构白皮书，链接我已经很体

贴的为你找好了，在这里《Google 基础设施安全架构设计白皮书》，你会看到 Google 是从物理层的硬件设备到固件 firmware、到操作系统、到应用服务通信的层层安全防护。当然，顺哥认为大部分企业不太可能做到 Google 那样的安全防护，尤其是中小企业，但这篇文章依然可以让我们极大地开阔视野，让我们没法参与其中的人也能了解人家是怎么做的，也许能让我们在做一些其它领域产品的设计时获得灵感。总之，视野决定格局，自己看着办咯。

5.7 监控架构

对一个系统尤其是大型分布式系统来说，监控是必需品，也是系统可运维的基础。前面我们曾提到过，一个系统不仅仅只有应用层，还有中间件层、基础设施层，分别对应云计算的 SAAS、PAAS、IAAS 这三层。我们需要知道每个层次的服务处于怎样的状态，以便对系统的故障发生有所准备，甚至在系统达到某个设定状态时自动采取某种防护措施。

而要得到每个层次的状态，其实就是需要进行全栈监控，监控内容诸如物理层的 CPU、内存、磁盘状态，

中间件 MySQL、消息队列的性能、容量，应用服务健康状况、接口调用延迟时间等等。不过对于使用平台的用户来说，一般只需要考虑 SAAS 或者应用层的监控，而 PAAS、IAAS 层的运维监控一般由云平台厂商来完成。作为一个中小企业的技术负责人来说，对于哪些内容需要平台方去做，哪些内容需要企业自己来做，甚至哪些内容需要客户来做，都要有一个清晰的认识，否则就可能属于传说中的 PPT 架构师或技术负责人了。

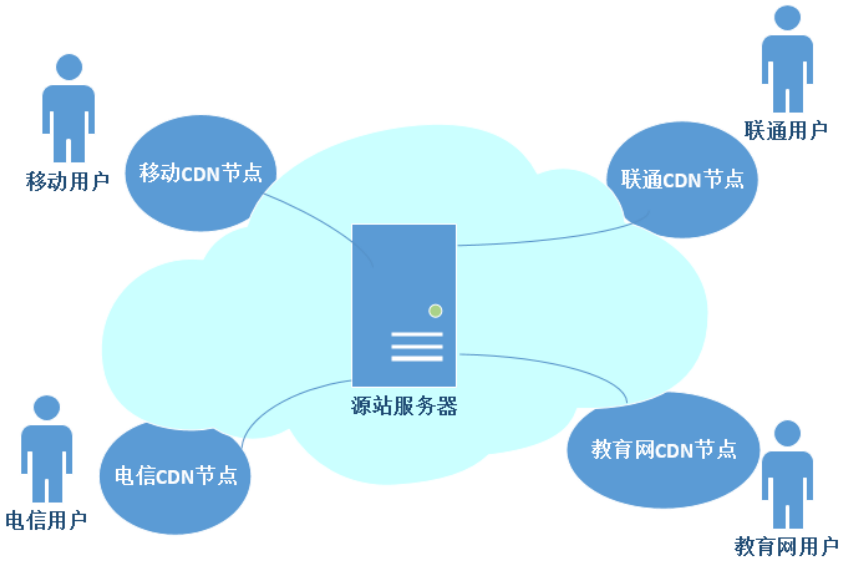
顺哥我还有一个观点，就是系统监控架构最好是尽可能没有侵入性，也就是运维路径和业务路径是分开

的。为了形象一点，我们打个比方：业务路径就像高速公路的行车道，供车辆高速行驶，运维路径就像紧挨着的应急车道包括服务区，运维人员就像高速公路上的监控人员。如果高速公路不出什么事故，一般来说没有人会干预车辆的行驶，也就是在业务系统健壮的情况下，监控系统只是在监视，不对业务系统进行任何干预，但当业务系统出现问题时，运维系统将采取相应措施，就像高速路出现事故时，高速监控人员会做出某种反应一样。

5.8 CDN 架构

CDN(Content Delivery Network)，内容分发网络，根据就近的原则，将服务资源分散到网络上的各个边缘服务器，加快用户访问速度。一般来说，CDN 就用来加速的是静态(据说还可以放代码，需要跟服务商沟通)内容，比如网站的 CSS、HTML、音视频等，举例来说，如果将视频文件放到本人所在的大南宁某个机房的服务器上，南宁的用户访问该视频时直接从南宁机房的服务器上获取，大大提高了视频访问速度。如果你还不太明白，顺哥这里画了一张稍微粗略的

图，这张图非常直观的展示了 CDN 服务器在物理上的部署情况，很容易理解。



CDN 的核心技术是智能 DNS(Domain Name Service)，即智能域名服务，本质上可以说是域名劫持，只是这种劫持是 DNS 服务主动配合 CDN 服务厂商的。简单地说，DNS 就是一种将域

名如(www.nndev.cn)转换为 IP 地址的技术，那么在 CDN 中，智能 DNS 又是什么原理呢？几年前我自己在学习的过程中，因为没有实际去使用过，偶尔也会有原理上理解不清晰导致困惑很久的地方，这里通过举个简单的例子从使用者的角度来让各位同学看到它的真实技术原理。

以顺哥的技术博客为例，假设 www.nndev.cn 域名对应的 IP 地址为：120.20.30.40，那么在不使用 CDN 的时候，我直接在阿里云的域名解析中添加一条类型为 A 的解析记录，将 www.nndev.cn 解析为 120.20.30.40，这个 IP 对应的是我在阿里云购买的

ECS服务器的公网IP。如果使用了CDN呢，我是否需要将域名解析到南宁的机房服务器对应的 IP 呢？要是这么做的话，用户需要访问那些放在我自己购买的ECS服务器上的那部分内容就访问不了，因为动态内容并不会放到南宁机房的服务器上。所以，这里不能直接将A解析指向CDN服务商在南宁服务器的 IP 地址，况且 CDN 服务商不止在南宁有机房，IP地址有多个还可能会变化，就算只有一个IP，那他修改 IP 时我是不是得自己去修改解析域名解析了呢。实际上，CDN使用了 DNS 的 cname 记录，即将一个域名 A 解析到另一个域名 B，然后根

据 B 解析到实际的 IP 地址，本人刚开始学习了解 CDN 的时候，困惑就在于不知道 cname 记录的作用，一直搞不明白整个流程的原理。

这里顺哥以使用阿里云 CDN 服务为例，首先通过 CDN 控制台添加博客域名 www.nndev.cn 为加速域名，阿里云会给该加速域名分配一个 cname 值比如：abcxfa.nndev.cn，然后我先把 www.nndev.cn 的 A 记录删除，给源站域名(www.nndev.cn 的上一级域名)nndev.cn 添加一条 cname 记录，主机记录为加速域名：www.nndev.cn ，记录值为：abcxfa.nndev.cn，这个 cname 记录值

将指向 CDN 服务厂商的 DNS 智能服务器，由 CDN 厂商的 DNS 服务器对域名进行进一步解析，将离用户最近的 CDN 服务器的 IP 地址返回给用户。那么用户在访问顺哥的站点时是使用加速域名(网上说使用源站域名，曾让顺哥陷入误区)www.nndev.cn 的，先将域名解析得到 cname 值：abcxfa.nndev.cn，这个值通过解析得到的是 CDN 服务厂商的 DNS 服务器，这时候 CDN 厂商接管了 DNS 服务，CDN 厂商的智能 DNS 服务根据用户 IP 地址去获取到离用户最近的 CDN 服务器的 IP 地址，并且它知道我们请求的 www.nndev.cn 对应的源站点域名

是 nndev.cn 或者源站点确切的 IP 地址，这时候如果网站内容不存在于 CDN 服务器上，CDN 服务器将请求发回到源站域名 nndev.cn 获取数据再返回给用户。这些流程原理，阿里的帮助文档也没有解释得很清楚，在这里要推荐一篇文章供大家参考，文章的说法正好印证了顺哥对整个流程的猜测理解，《CDN 设置原理,源站 IP 和源站域名的区别,回源 host 的作用》。

各位同学是不是有点晕菜呢？好吧，哥再总结一下配置流程：

1) 在域名解析服务商那里删掉 www.nndev.cn 的 A 记录；

2) 在阿里云 CDN 控制台添加 www.nndev.cn 为加速域名，并将源 IP 指向 1)中 A 记录所执向的 IP；

3) 从 CDN 控制台中获取加速域名 www.nndev.cn 分配到的 cname 值如：abcxfa.nndev.cn；

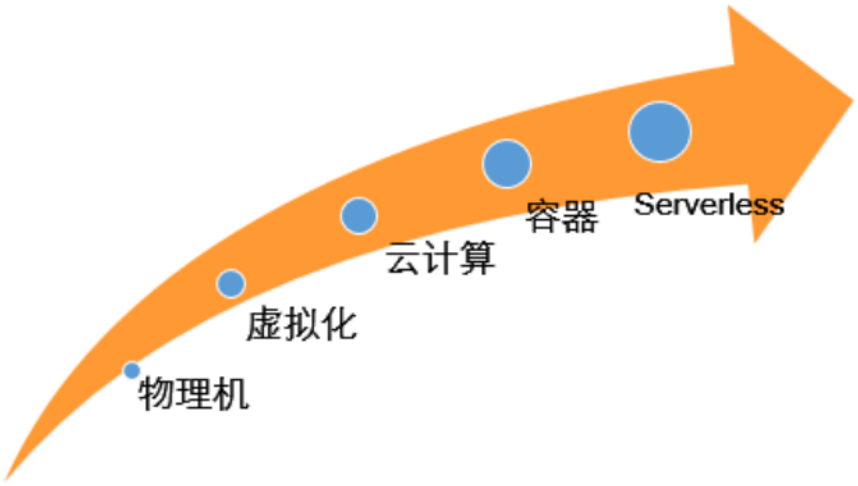
4) 到域名解析服务商控制台，添加 cname 记录，主机记录为 www.nndev.cn，值为 abcxfa.nndev.cn；

通过以上 4 个步骤后，顺利的情况下 CDN 加速就生效了，用户访问的时候，还是访问加速域名 www.nndev.cn，因

此，部署 CDN 前后用户是无感知的。

5.9 无服务器架构

无服务器架构是一种新兴的技术架构，也叫 Serverless 架构。无服务器的意思并不是后端真的没有服务器，而是从基于云平台进行应用开发的开发者的角度来看，不再需要像以往那样去购买、配置、运维服务器。IT 基础设施技术的发展历程，下面这张图可以大致概括，通过这张图，你就能更清晰理解 Serverless 架构是和其它哪些概念有相关性和处于怎样的位置。



我们还是根据这张图对 IT 基础设施发展历程进行一个简单说明吧：

1) 直接使用单个物理机或物理机集群；

2) 通过虚拟化将物理机虚拟成多个 VM 资源；

3) 将虚拟化技术进行云化，打造云平台；

4) 将物理机或者虚拟机切分成更轻

量级的 Docker 容器；

5) Serverless 架构，开发者无需考虑 VM、Dockers 等部署问题；

到这里我们已经明确了一个观点，就是无服务器架构并非不需要服务器，而是后端开发人员不需要去管理服务器，服务器配置和应用运行环境的设置维护都是由云平台厂商代劳了。在这里，我们引入另外两个概念，即 BAAS(Backend as a Service)和 FAAS(Function as a Service)。它们都是实现无服务器架构的一种技术形式，为简单起见，下面我们只谈论 FAAS。

亚马逊的无服务器架构提供的

服务叫 AWS Lambda，它就是一种 FAAS 的实现。FAAS 如何使用呢，我们从如何使用 AWS Lambda 就可以明白。AWS Lambda 可以做下列几个事情：

1) 可以运行几乎所有应用类型的服务代码；

2) 零管理，只需要上传代码，其它事情由 Lambda 代劳；

3) 几乎无限的扩展性；

4) 高可用性，代码可以由其它 AWS 服务触发或直接被调用；

总体来说，FAAS 可以让你在不关

注服务器系统的情况下运行服务器端代码，而且你的服务不是一直在待命着的 daemon 进程，一般是由某种事件触发才被运行起来，比如你往 S3 存储添加一个新文件的时候被触发运行，这个跟之前所有的架构模式都不太相同。

从 FAAS 提供的功能和使用方式来看，我们可以看出来，使用无服务器架构时，我们的代码运行完就结束了，好像没有来过这个世界一样。同一段代码两次运行之间不能共享状态，状态数据应该被保存到数据库或者其它第三方存储系统。因此，我们编写代码时应该注意避免存在跨函

数调用的状态共享，这也是函数式编程的思想，输入数据后运行得到结果输出，中间没有任何状态存储。在可扩展性那一节里，我们也提到过，无状态的应用是系统可扩展性的关键因素之一，无独有偶，无服务器架构的可扩展性也对我们编写的应用代码提出了这一要求。

无服务器架构的事件触发、不需要后台服务进程的方式确实是比较先进的玩法，也是将云平台的按需使用特性发挥到了新的高度，不过目前来说，这种架构在各个云平台的成熟度还不是那么高，是否采用这种架构目前还值得商榷，但这种技术本身值

得我们持续关注。

5.10 负载均衡

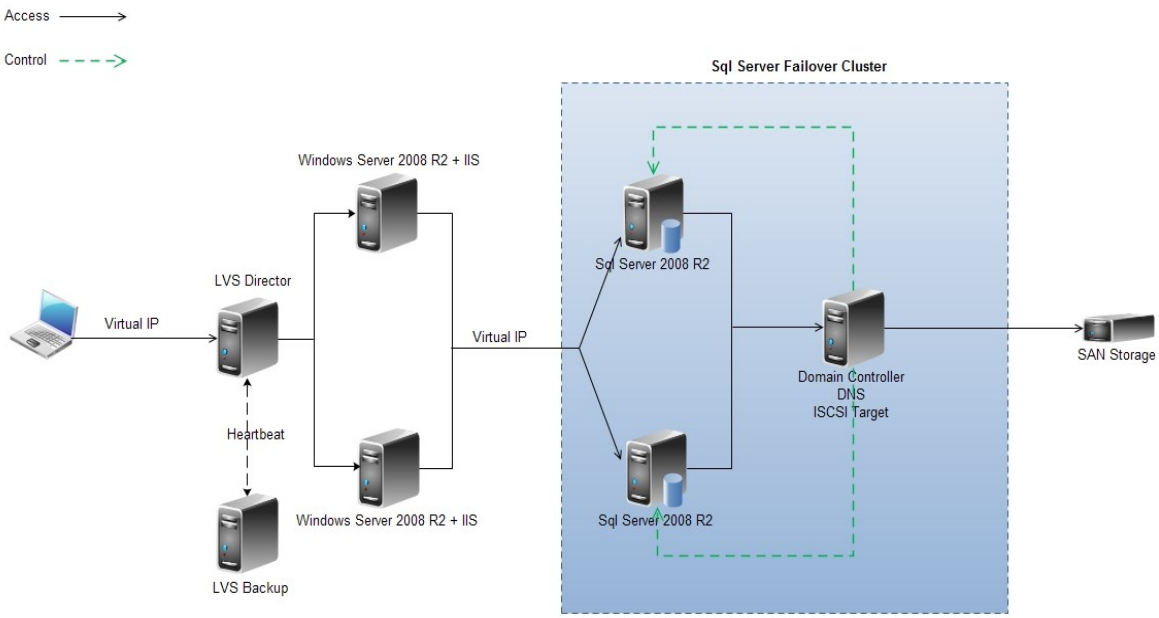
业务量上去之后，我们需要增加服务器，但是对于通过 API 请求获取后端数据的客户端程序来说，后端服务器的增加和部署变更是透明的。那么这样就需要有个统一的后端入口，也就是唯一的 IP 和端口。要做到这些，一般要用到负载均衡服务。负载均衡服务的原理就是将接收到的请求根据某种算法(比如轮询)转发到后端真实的服务器上，后端服务器之所以被称为真实服务器，是因为实际的

业务代码是运行在后端服务器上的。

负载均衡又分硬件负载均衡和软件负载均衡，硬件负载均衡据说一般使用 F5，本人没有用过，软件负载均衡就是通过软件来实现的，常见的有 LVS、HAProxy、Nginx 等。本人最早接触的是 LVS，LVS 作为一个开源项目已经被集成进了 Linux 内核，它的作者是前阿里云 CTO 章文嵩博士，膜拜啊。当时可以说本人对 LVS 是有过一定深度的研究，配置过好几套系统，对 LVS 的比如三种路由方式等内容也是如数家珍，不过现在渐渐地忘掉了细节的东西。HAProxy 没有使用过，Nginx 的话用来做反向代理，多

个域名的时候可以根据域名转到到后端不同的站点。Nginx 同样具有负载均衡的功能，但本人没有使用过这个功能。

为了直观一点，现将本人几年前搭建的一个使用了 LVS 负载均衡服务的系统架构图展示下，如图：



其中 LVS Director 和 LVS Backup 是负载均衡服务所使用的两台服务器，正常情况下 LVS Backup 服务器作为备份，当 LVS Director 服务器出现故障时，LVS Backup 会顶替继续提供转发服务。右边两台安装 Windows Server 2008 R2 的服务器是真实服务器，里边运行实际的业务代码，比如跑 PHP 代码，再右边是存储相关的，跟负载均衡服务关系不大，这里不做过多解释。

为什么这些年来本人没有再去进行负载均衡服务的配置呢？仅仅是因为懒吗？不对，负载均衡的配置需要有两台服务器，一台当作 Master

点，另一台作为 Backup 节点，如果在阿里云你就得为此买两台 ECS 实例，费用也不少，自己搭建后还需要去维护。所以一般情况下，我们直接购买阿里云的负载均衡服务 SLB，一分钟就搞定了，这个服务一般是按量付费。为什么本人会选择购买云厂商的服务呢，主要是觉得，云厂商的服务一般都是有大量的用户使用，这些服务可能都受过各种严苛情况下的考验，出问题的几率会比我们自己搭建的小，而且直接购买服务效率高，成本还可能更低。不要为了技术而技术，技术是为了解决公司的业务问题，脱离业务去玩技术炫技无异于耍

流氓。

5.11 消息队列

如果你学过数据结构基础课程，那么你就会知道，队列的特点就是先进先出，简单来说，就类似你去食堂排队吃饭，先到的人排在前边，拿到饭菜后就先从队列里出去，在软件系统中，这样的好处是保证消息处理时序上的一致性，也就是先来的消息先得到处理。

消息队列是系统解耦的利器，其生产者和消费者并不需要同时操作队列。一般生产者只负责将消息放进

队列，至于消费者什么时候从队列里取出消息进行处理，消息生产者是不关心的，消息的处理是异步的方式。消息队列适用于系统间、进程间、线程间的通信。

但是如果没有统一的标准协议，各个公司实现的消息队列中间件就会五花八门，会大大增加使用者的使用难度，因此，在开源社区出现了以下三种消息协议的标准：

1) AMQP(Advanced Message Queuing Protocol)，高级消息队列协议；
2) STOMP，全称为“Streaming Text Oriented Messaging Protocol”，面向

文本流的消息队列协议；

3) MQTT (formerly MQ Telemetry Transport)，面向嵌入式设备的轻量级消息传输；

ActiveMQ 和 RabbitMQ 是比较有名的消息队列中间件，实现了包括以上列出的主流的消息协议。各个云平台一般也有自己的消息队列产品，有的是在这些开源的中间件上做一些改进，有的是完全重新发明轮子，根据消息协议内容进行自主研发。那我们在各种不同消息中间件中进行选择时，一般是从成本、可扩展性、可维护性、可靠性这四个方面综合考

虑。

5.12 边缘计算

边缘计算也叫 Edge Computing，是一种分布式计算模式，将计算任务分配到手持设备或者边缘服务器上，而任务的分发和管理则集中在云计算平台。通过云服务统一编排管理任务，然后将任务下发到各个边缘节点(或设备)，任务在各个边缘节点执行，于是得以充分发挥散步在世界各个角落的数以亿万计设备的计算力。

边缘计算看起来和物联网类似，再想想貌似和 CDN 都有点类似，哈

哈。只不过物联网强调的是万物互联，而边缘节点强调的是使用边缘节点的计算力，这些边缘节点也许就是通过物联网连接起来的各种设备。在顺哥看来，物联网和边缘计算融合，是一个非常有前景的技术方向。

从 Wikipedia 上，我们可以看到边缘计算有以下有点：

1) 因为数据就近处理了，减少数据传输；

2) 去中心计算化，更容易避免单点问题；

3) 复用已有公有云和私有云的

架构体系，降低建设成本；

而边缘计算的挑战性在于：

1) 边缘设备的可用性，可连接性；
2) 边缘计算要求应用可横向扩展，通常遵守 12 要素规范；
3) 边缘计算需要将任务分到到很多个设备，跨设备的状态协调和存储是一个问题；

尽管边缘计算有这些挑战性存在，顺哥依然认为，随着物联网的发展普及，边缘计算不可避免会蓬勃发展，各位同学有机会的话赶紧占坑。

5.13 权限系统示例

我们一般使用 RBAC(Role-Based Access Control)实现权限控制系统。基于 RBAC 的权限控制系统里面，一个用户可以有多个角色，一个角色有多个权限。

在这里我们举个例子吧，基于 RBAC 实现一个结合了 Spring Security + JWT + Oauth2 的权限控制系统。如果你不熟悉 Spring Security 或者属于那种不太懂代码又想当技术负责人的类型，自行跳过这一节吧，免得被绊倒姿势就就没那么优雅了。网上一

些基于 Spring Security 实现权限控制的例子，一般都是在方法上加声明：

```
@PreAuthorize("hasRole('ADMIN')")
```

或者

```
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
```

这样如果当前用户具有 ADMIN 角色，则可以访问该接口。这种情况下一般涉及到两个类 Role 和 User，代码大致如下：

```
@Entity
public class User implements UserDetails, Serializable {
@Id
  @GeneratedValue(strategy                                    =
```

```
GenerationType.IDENTITY)
private Long id;

@Column(nullable = false, unique = true)
private String username;

@Column
private String password;

@ManyToMany(cascade = CascadeType.ALL, fetch =
FetchType.EAGER)@JoinTable(name="user_role",joinCol
umns=@JoinColumn(name="user_id",referencedColum
nName="id"),inverseJoinColumns=@JoinColumn(name=
"role_id",referencedColumnName = "id"))
private List<Role> authorities;

  public User() { }

  @Override
    Public Collection<? extends GrantedAuthority>
getAuthorities() {
                return authorities;
```

```
        }

        public void setAuthorities(List<Role> authorities) {
            this.authorities = authorities;
        }
        ......省略各种 get,set 接口
    }

@Entity
public class Role implements GrantedAuthority {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
        private Integer id;
        @Column(nullable = false)
        private String name;
        @Override
        public String getAuthority() {
          return name;
        }
        ......省略各种 get,set 接口
    }
```

当我们在接口上添加了如下标注：@PreAuthorize("hasRole('ADMIN')")后，流程大概是这样：

先调用 User 类的 getAuthorities 接口，取出 authorities，其类型为 List<Role> ，然后调用每个 Role 实例的 getAuthority 接口，该接口返回 Role 名称，比如“ADMIN”，只要其中某个 Role 返回了“ADMIN”，即可停止遍历，表示当前用户具备了访问该接口的权限，放行。

这样在数据库里面就会有 user、role、user_role 表，分表存储用户信息、角色信息以及用户和角色关联(多对多)信息。如果每个接口对应一个

role，那实际上作为角色的 role 在我们看来跟以往的权限(permission)对应了，即实际上是 user 和 permission 的关系，只是叫做 role 罢了。具有 ADMIN 角色的用户才能访问的接口添加标注：@PreAuthorize("hasRole('ADMIN')")。

这样的话，我们在编写接口代码的时候，就要把这个标注写上去，让具备 ADMIN 角色的用户可以访问之，那如果某天我不想让 ADMIN 用户访问这个接口呢，我该怎么办？要么我需要回收该用户的 ADMIN 角色，要么我得去修改接口标注。如果该角色确实只对应一个接口的权限，那回收倒是没有问题，但 Boss 要你实现一个角

色拥有多个权限，实际上会在多个接口上做了同样的标注，回收角色后你会发现其他一些本来该用户可以访问的接口现在访问不了了，头大吧，想来想去你只能去改代码了，去修改某个特定接口的标注。

还记得基于 RBAC 的权限控制系统里面，一个用户可以有多个角色，一个角色有多个权限，因此我们最好是按照 RBAC 的方式来做，这样的话，一个接口就对应一个权限，一个角色具有 N 个权限，一个用户可以有 M 个角色。这个也好理解，不过怎么实现修改用户角色、权限的时候不需要修改接口标注呢？一般来说，这种修改

都是管理员通过操作后台管理界面来实现的。

我们的方法是添加 permission 表和 role_permission 表，即权限表、角色权限关系表。权限类的代码很简单，跟数据库字段对应，不过你得注意下，这个类跟 Role 类一样，都要实现了 GrantedAuthority 接口：

```
@Entity
Public class Permission implements GrantedAuthority {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;
@Column(nullable = false)
private String name;

    @Override
```

```
    public String getAuthority() {
            return name;
    }
    ......省略各种 get,set 接口
}
```

之前提到过，是否具有访问接口权限是由 Role 类的 getAuthority 的返回值来判断的，这些 Role 实例又是从 User 对象的 getAuthorities 接口获取的，那么我们就想到，我们通过修改 User 类的 getAuthorities 接口，让它直接返回 List <Permission>而不是 List <Role>，getAuthorities 接口返回类型是 Collection<?extends GrantedAuthority>，因为 Permission 和 Role 类都实现了 GrantedAuthority 的接口，因此返回

Permission 和 Role 列表都能满足要求。于是 Role 类改成这样：

```
@Entity
public class Role implements GrantedAuthority {

@ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
	@JoinTable(name="role_permission",joinColumns=@JoinColumn(name="role_id",referencedColumnName="id"),inverseJoinColumns=@JoinColumn(name="permission_id", referencedColumnName = "id"))
	private List<Permission> authorities;
	    public List<Permission> getAuthorities() {
	    return authorities;
	   }
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
	 private Integer id;

	 @Column(nullable = false)
```

```
    private String name;

    @Override
    public String getAuthority() {
        return name;
    }

      ......省略各种 get,set 接口

  }
```

User 类改成：

```
@Entity
public class User implements UserDetails, Serializable {
    ......
@Override
Public Collection<? extends GrantedAuthority>
getAuthorities()
 {
    List<Permission>  permission_authorities;
```

```
    permission_authorities = new    ArrayList<>();
    for (Role role: authorities ) {
        for(Permission permission: role.getAuthorities())
        {
            permission_authorities.add(permission);
        }
    }
    return    permission_authorities;
     }
        .....省略各种 get,set 接口
}
```

这样最终会调用到的是Permission类的getAuthority接口而不是 Role 类的接口。这样一来，我们就可以规定，每个接口的权限名称、对应的数据库里面的权限名称、接口函数名称这三者相同，只要遵守这个约

定去写接口标注和接口名称，从界面添加接口对应的权限时也使用同样的名称，就可以达到无论如何修改用户和角色、权限关系时，都不需要修改代码标注，而且实现了 RBAC 方式的权限控制系统。示例代码如下：

```
import ...

@RestController
@RequestMapping("/test")
public class TestController {

    /* @PreAuthorize里面判断的权限名称、接口名称、数据库里的权限
      名称三者相同，然后通过界面给角色配置权限，给用户配置角色，
      就能达到可动态改变一个用户是否具有权限访问某个接口的效果
     *******/
    @PreAuthorize("hasAuthority('hasPrivilege')")
    @GetMapping("/permission")
    public String hasPrivilege() { return  "Test Privilege"; }
}
```

hasPrivilege

5.14 架构案例分享

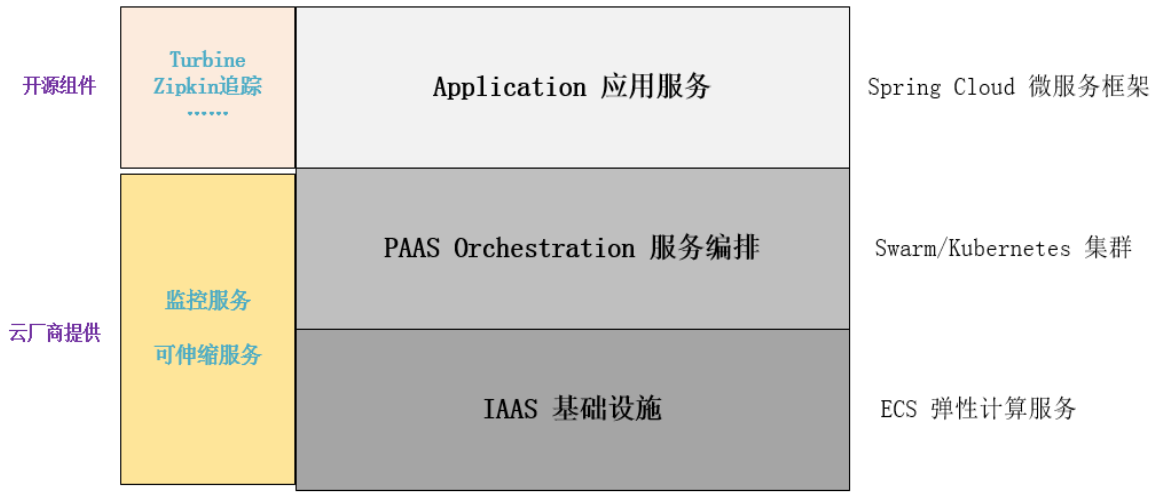
好多同学以为顺哥满篇都是方向性的指引而已，妹有，妹有呢，下面我将指引大伙设计一个具体的系统，要求使用微服务应用架构并使用 Docker 部署。

很显然，我们知道要架构一个应用系统，还需要底层的基础设施等。架构师杨波曾经说，架构师需要具备的思维包括：抽象思维、分层思维、分治思维、演化思维，我深以为然。然而我们不可能一上来就具备这些思维，只有经过不断踩坑不断总结的人才能悟出这些东西，那作为初学

者，我们按着分层思维来看看吧。

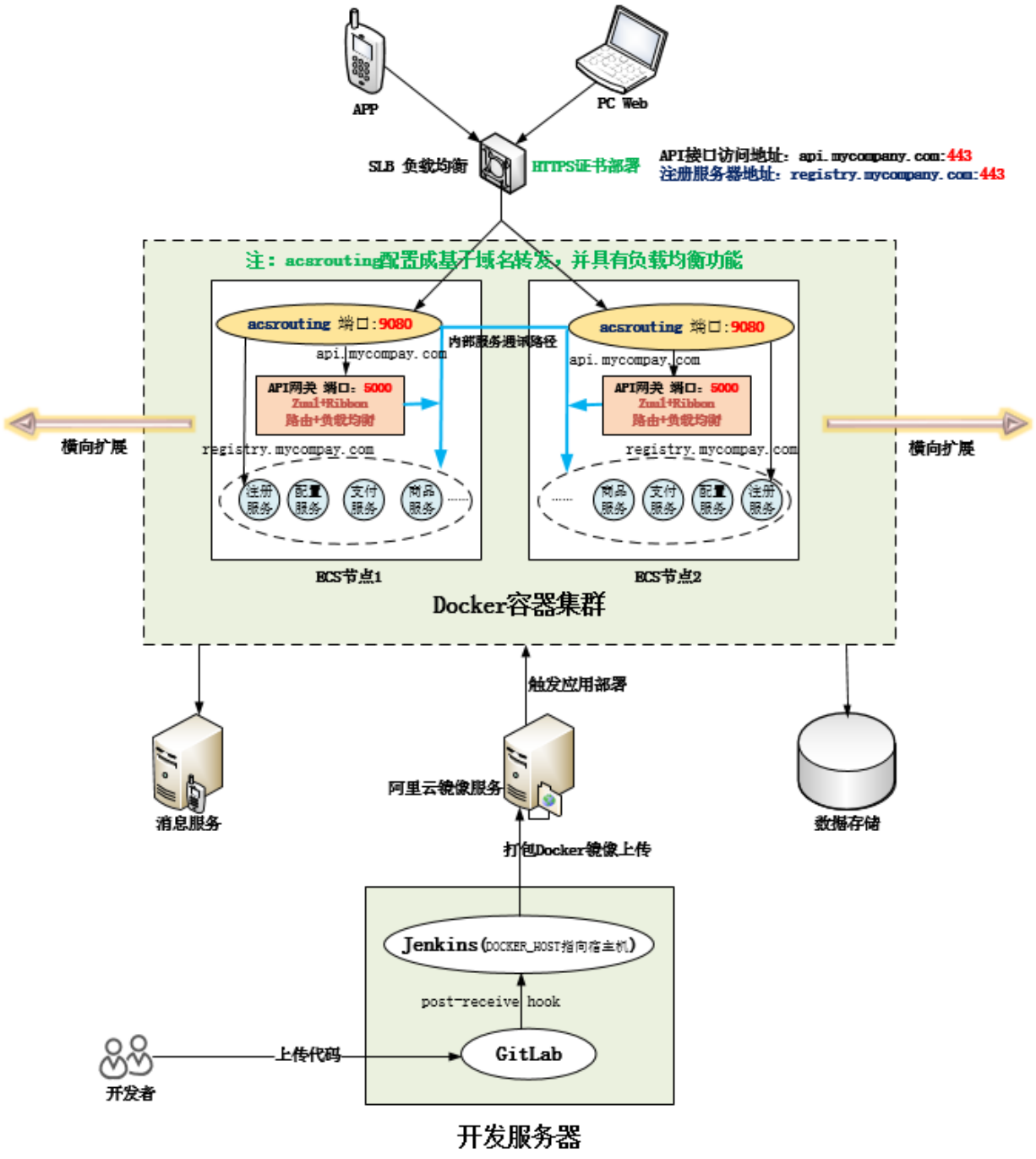
你的应用需要跑起来，是不是需要服务器？是。为了让各种服务能够被编排调度，是不是需要一些编排系统，如果需要多个服务器，是不是需要分布式集群软件？是。各种服务间通信，是不是需要某种协议，服务的管理等是不是需要一些框架来帮助完成？是。上面所说是从下往上的视角，一般的应用开发人员会先想到最上面那一层，也就是应用层，其实也无所谓。对于应用层，因为要求是微服务，那么我们需要采用一个微服务应用框架，这里我们选择 Spring Cloud。如何部署这些 Spring Cloud 服

务呢？因为要求采用 Docker 容器，那么常见的容器集群技术方案为 Swarm 和 Kubernetes，后者对服务器的资源貌似要求更高，比如阿里云不让你创建只有一个节点的 Kubernetes 集群，而 Swarm 集群可以，那我们先选择 Swarm 集群吧。Swarm 集群又需要机器来承载吧，因为我们是在使用云平台，最容易的办法就是购买虚拟机了，在阿里云就是购买 ECS 实例。这样一来，我们整个架构的层次从大的方面来看就是下面这张图：



这张图的下面两部分对应的组件都是直接从阿里云购买，当然你会说我怎么知道这些啊，你是对的，因为你没有对云平台有一个比较好的理解，不知道云平台能给我们提供怎样的服务，所以，你可能需要补一些云计算方面的知识，了解了解前面我提到过的 IAAS、PAAS、SAAS 这些概念以及具体的一些云产品。

了解了上面那些概念以后，对于你来说，也许完成这个系统架构设计还是有困难，好吧，我直接端出我之前设计的架构图，供你参考，其中还包括了代码上传自动构建 Docker 镜像的流程。



这张图上，采用的阿里云产品包括负载均衡、容器服务、容器镜像服务、

ECS 弹性云计算服务。这张图是对 API 服务的部署，你可能猜出来我们也是搞前后端分离的架构了。本来想在这里对这个架构进行一番详细的解释，但你要了解顺哥向来是那种不屑于向别人解释自己言行的人，剽悍的人生不需要解释，嘿嘿。因此，顺哥在这里偷懒了一回，如果各位同学看着就能理解了，那不错，不过要是没理解，可以当作家庭作业一样来分析，同时也能加深理解。如果还有其它疑问，欢迎通过微信或者“技术人成长”社群向本人提问，只要时间允许，我会耐心给各位解答。

6. 软件设计原则

软件设计原则可以说是无数前辈在踩过无数坑之后总结出来的提醒后人遵循的一些基本思想、规范、模式。遵循这些原则，有利于我们做出良好的设计，比如达到高内聚低耦合、模块划分清晰、源码可读性可维护性良好的效果。下面将对最广为人知的八大原则进行说明，有些原则还会给出代码示例。

6.1 KISS 原则

KISS 是“keep it simple, stupid”的简称，这种观点认为，一个简单的

系统往往比复杂的系统运转得更好，因此，在进行系统设计时应尽量保持简单，避免不必要的复杂性。

这里强调的是不必要的复杂性，有时候一些复杂性不可避免，但我们无论在产品设计还是架构设计方面，在保持功能完整性的同时，应该尽量做减法，坚守 KISS 原则。

6.2 DRY 原则

DRY 是“Don't repeat yourself ”的简称，这个原则的目标在于通过抽象的手段来减少重复的模式，避免冗余，包括设计、代码等方面的冗余，

强调系统中的每个实体元素都是单一的、无歧义的。DRY 原则运用好的话，系统中任何元素的修改都不会导致逻辑上跟它无关的元素的修改，而逻辑上相关的元素即使被修改，也是以统一的可预测的方式被修改。

怎么做到遵守 DRY 原则呢？本人认为要依靠抽象思维，我们在设计系统和数据结构时，尽量理解透彻系统中所存在的实体及其关系，将共性的内容抽出来成为一个抽象层，具体的实体再从抽象层派生，这里其实就对应了面向对象里面继承的概念，继承使得代码可以被复用。

6.3 开闭原则

开闭原则也叫 Open-Close 原则，大致思想是：软件实体(包括类、模块、函数等)应该对扩展开放，而对修改关闭。以类为例来说明，就是你可以从一个基类派生新的类作为子类，子类可以扩展基类的功能，但不能修改基类的代码。开闭原则也是最基本的原则，这一章中所有其它原则本身都遵守了开闭原则。

现实中，很多软件中的插件，就是开闭原则思想的具体实践。通常允许你通过添加插件的方式来扩展软件的行为，而不允许修改软件的一些

基础功能。如果允许你修改软件的基本功能，那么有一天，这个软件具备的功能和软件原来声称的功能大相径庭，想想都觉得是件诡异的事情。

6.4 里氏替换原则

里氏替换原则(Liskov substitution)可以表述为：所有出现基类的地方都可以由子类来代替，并且替换后不需要修改其它代码系统还可以继续正常工作。这样看来，里氏替换原则也符合开闭原则，不修改基类，而是将子类替换基类，就能够达到扩展系统功能的目的。

怎么运用这个原则呢？个人认为在设计系统时，要关注系统实体间的关系(比如是否是继承关系)和扩展性，这两个特性结合起来的地方可能就是使用了里氏替换原则。

6.5 依赖倒置原则

依赖倒置原则是软件模块解耦的一种方式，其核心思想是：高层模块不应该依赖于底层模块，它们都应该依赖于抽象，也就是抽象不能依赖于细节，细节应该依赖抽象。

前后端分离架构就是一个例子，前端只管使用 ajax 请求服务端的 API，

不在乎服务器端 API 服务是如何实现和部署的，只需要 API 接口遵守协议(比如 HTTP)就好了。

6.6 单一职责原则

顾名思义，单一职责原则的核心思想是：一个类，只做一件事情，只有一个理由引发它的改变。这也和 Unix 的设计哲学之一“Do One Thing and Do It Well”有异曲同工之妙，即只做一件事，并把它做好。

这个原则说起来最简单，而实际上执行起来不容易，稍微不小心就容易违背它。只做一件事，对应到软件

设计里边，就是一个模块(类、函数)尽量少做事情，但要做好，这样也符合我们常说的高内聚低耦合的设计要求，所以在单一原则里边，我们需要注意的是模块(类、函数)的边界和职责划分，思考怎样划分才能符合该原则，这里就不举例子了。

6.7 接口隔离原则

接口隔离原则是面向对象设计的一个原则，倡导将接口分离，即用户不需要实现他使用不到的接口。比如一个接口定义有两个方法：

```
interface ShapeInterface {
```

```
    public function area();
    public function volume();
}
```

而有时候实现类只使用到其中一个方法，但编程语言规范又迫使派生类去实现基类的两个接口。如果遵循接口隔离原则，这时候应该将接口分解为两个接口，然后实现类根据需要可以只实现其中一个接口，如果需要两个接口方法都实现，就去实现两个接口好了。代码如下：

```
interface ShapeInterface {
        public function area();
}
interface SolidShapeInterface {
```

```
        public function volume();
}
                class Cuboid implements
ShapeInterface,     SolidShapeInterface
{
        public function area() { }
        public function volume() {}
}
```

原则是一种思想，并不局限于某种编程语言，上面以 Java 代码为例来说明，其他语言中可以根据各个语言的不同规范采用不同的方式来是接口符合这个原则。

6.8 最少知识原则

最少知识原则(Least Knowledge Principle)也叫迪米特法则(Law of Demeter)，其来源于 1987 年荷兰大学的一个叫做 Demeter 的项目。Craig Larman 把 Law of Demeter 又称作“不要和陌生人说话”，其核心思想是提倡一个模块只知道自己模块内部的东西，对其它模块知之甚少，与其它模块的交互都是通过接口来实现，这个原则可以引导我们完成低耦合的系统设计。

上面说到的是模块，那其实我们可以小到一个类，一个函数，都可以

遵循最少知识原则。比如我们尽量将类的成员变量声明为私有的，只能通过类的公共方法来访问该变量，外部不能直接使用该变量。在函数里面，我们尽量使用局部变量，少用全局变量，因为局部变量只在这个函数内有效，全局变量可以共用，使得函数的耦合性变高了。因此，提倡使用局部变量，也是遵循了最少知识原则。

6.9 好莱坞明星原则

好莱坞明星原则叫 IOC(Inversion of control)原则或者控制反转原则，英文里比较形象的说法就是“Don’t call

me, I will call you back”，源于这么一个现象：好莱坞的经纪人们一般不希望你去联系他们，而是他们会在需要的时候来联系你，因为主动权确实是掌握在他们手上，耍大牌不用商量。

传统的编程方式是，用户代码直接调用底层库函数，而控制反转原则是底层框架反过来调用用户编写的代码，类似基于事件的编程。当事件触发时，底层框架代码被调用，然后再反过来调用用户自定义的事件处理程序。这样来看的话，就是框架提供了程序的引擎和接口定义，用户代码负责接口实现，应用程序的开发模

式就掌控在框架开发者的手中。引用台湾著名架构师高焕堂的话，就是好莱坞明星原则保证了强盛，保证了底层框架的龙头地位。

这里我们举个简单的例子吧，比如我们在进行 GUI 编程时，通常要为控件编写一个 OnCreate 函数，但是这个函数我们从没有主动去调用它，而是框架在适当的时机来调用，这就是将控制权交给了底层框架，完成了控制反转。

6.10 面向接口原则

面向对象设计模式中有一个模式叫桥接模式(Bridge pattern)，提倡基于接口编程，英文里边的说法是：Program to an interface, not an implementation。因为同一个接口可以衍生出各种不同的实现，接口和实现分离使得用户程序可以根据不同的情况选择不同的实现，实现的修改独立于接口的调用者，这样当需要修改实现代码时，接口调用者是无感知的，这样也就降低了软件的耦合度。

下面我们以 Java 代码为例来看一下如何面向接口编程，首先，我们有一个显示接口，定义如下：

```
interface displayModule {
   public void display();
}
```

显示器类，实现 displayModule 接口

```
                public class Monitor implements
                   displayModule{
    public void display() {
          System.out.println(“Display through
Monitor”);
    }
}
```

投影仪类，实现 displayModule 接口

```
public class Projector implements displayModule
{
    public void display(){
        System.out.println(“Display through projector”);
    }
}
```

主机类，聚合了显示接口

```
public class Computer
{
    // 面向接口编程
    displayModule dm;
    Public void setDisplayModule
    (displayModule dm)
    {
        this.dm=dm;
    }
    public void display()
    {
        dm.display();
```

```
    }
 }
```

程序主函数，类似于一个装配器

```
public static void main(String args[])
{
    Computer cm = new Computer();
    displayModule dm = new Monitor();
    displayModule dm1=new Projector();
    cm.setDisplayModule(dm);
    cm.display();
    cm.setDisplayModule(dm1);
    cm.display();
}
```

上面的例子中，Computer 类聚合了 displayModule 接口，然后通过一个方法 setDisplayModule 来设置具体的接口实现对象，比如显示器或者投影仪。主程序就类似于装配器，程序

作者就像装配工人，根据需要装配不同的插件(显示器或者投影仪对象)。如果 Computer 类聚合的不是接口而是某个实现类，也就是直接聚合某个插件，这样就不需要方法 setDisplayModule 了，但是失去了灵活性，装配工人也就无法根据需要装配不同的插件了。

7. 持续集成部署

持续集成(CI, Continuous Integration)是指经常将多个代码分支合并进主干分支的行为，并且合并后的主干代码能够通过自动化测试。持

续部署(CD, Continuous Deployment)是指通过自动化流程以较快的频率部署新功能的行为，持续部署后新功能就已经交付用户使用。一个系统中不仅仅有业务功能代码需要持续集成，还有配置文件、数据库等非功能代码性质的内容需要持续更新，这些都需要依靠持续集成环境来完成，并且往往是以自动化的形式。

7.1 必要性

前面提到过，自动化水平某种程度上可以反映一个企业的生产力水平，那么软件研发要达到几乎全流程

的自动化，持续集成和部署是不可缺少的环节。企业的发展会带来业务的增长和业务的多样化，这些变化需要企业去开发更多软件产品或者升级现有的软件产品来应对，这样就涉及到集成更多功能到现有产品、新增流水线等等问题。自动化程度较高的持续集成和部署环境，是企业能够快速提供新功能和新产品的标志之一，也是企业生产力水平的标志之一，因此其必要性不言而喻。

7.2 源码管理

源码管理系统目前大多使用 SVN

和 Git，其它更古老的源码管理系统越来越少有人使用了。一提到 SVN 和 Git，一般就会说 SVN 是集中式管理，而 Git 是分布式的，因此 Git 支持离线工作，SVN 必须联网才能工作。其实这后半句经常也带来一些误解，让人以为不联网就不能继续写代码似的，因此，要把话讲清楚了，也就是在不联网的情况下无法执行 SVN 命令罢了，而 Git 在本地就是一个完整的仓库，不联网也可以执行几乎所有命令，比如 add、commit 命令，如果需要，在联网的时候 Push 到源头再进行 Merge。

两者的优劣，仁者见仁智者见

智，顺哥不想在这里评判了，免得引发口水战，总之，适合自己的那款就是最好的。

在实践中，我们多次使用过 GitLab 搭建私有源码库，并结合 Jenkins 进行自动化编译构建。在 GitLab 项目上设置钩子，当开发者上传代码时，触发钩子，其实也就是一段代码的执行，这样就会触发 Jenkins 上某个 Job 的执行，在 Job 执行时又运行我们的自动化构建脚本，对源代码进行编译、打包镜像等可自定义的操作。

7.3 配置管理

刚开始我们在 Jenkins 的构建步骤中写了一些 Shell 脚本，接着发现脚本功能越来越多，这个时候要去修改脚本就直接打开 Jenkins UI 界面去修改。后来想想，这些脚本也是一种代码，何不像源代码一样放到 GitLab 上进行管理呢。如果你用 Laravel，每个项目都有.env 文件，这些文件作为配置文件，但实际上我们可以把它当作源代码看待，即配置文件源码化。

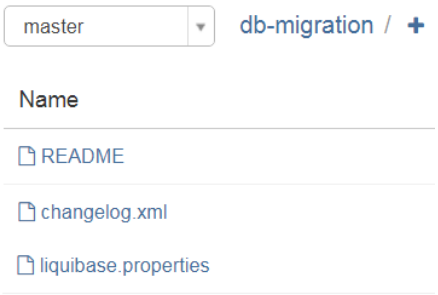
7.4 数据库管理

我们一般使用 Git 进行源码版本管

理，但是数据库同样需要管理，在一个多人的团队里面，经常是某个人改了数据库比如增加字段，这个字段没有默认值的话，导致其他使用该数据库的程序崩溃。或者是修改了数据库却没有做相应的记录，过了一段时间后，生产环境的数据库也需要做相应的修改，但是却已经记不清修改了哪个表哪些字段。直接粗暴的用一些工具来比较两个数据库结构虽然可行，但是感觉很繁琐，并且在修改的比较多了以后，要让两边的数据库结构变得一样还是有点麻烦。Laravel 框架也有 Database Migration 的概念，但我们的项目中并不只有 PHP，还用到其他

各种编程语言和框架，因此，使用 Liquibase 进行数据库版本管理、持续集成比较符合我们的要求。

我们在 GitLab 上新建一个叫 db-migration 的项目，这个项目下面有如下文件：



changelog.xml 文件原始内容如下：

```
<databaseChangeLog
        xmlns=”http://www.liquibase.org/xml/ns/dbchangelog”
        xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
        xmlns:ext=”http://www.liquibase.org/x
```

```
        ml/ns/dbchangelog-ext”
        xsi:schemaLocation=”http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org  /xml/ns/dbchangelog/dbchangelog-3.1.xsd
        http://www.liquibase.org/xml/ns/dbchangelog-ext
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd”>
</databaseChangeLog>
```

比如我现在想添加一张表，名为“db_migration_test_table”，其中有两个字段，分别为 id 和 name，类型分别为 int 和 varchar(50)，其中 id 为主键，则可以通过将 changelog.xml 文件改为如下内容：

```
<databaseChangeLog
        xmlns=”http://www.liquibase.org/xml/ns
        /dbchangelog”
        xmlns:xsi=”http://www.w3.org/2001/XM
        LSchema-instance”
        xmlns:ext=”http://www.liquibase.org/xm
        l/ns/dbchangelog-ext”
        xsi:schemaLocation=”http://www.liquiba
        se.org/xml/ns/dbchangelog
        http://www.liquibase.org
        /xml/ns/dbchangelog/dbchangelog-3.1.x
        sd
        http://www.liquibase.org/xml/ns/dbchan
        gelog-ext
        http://www.liquibase.org/xml/ns/dbchan
        gelog/dbchangelog-ext.xsd”>

  <changeSet id=”1”   author=”shun”>

      <createTable
tableName=”db_migration_test_table”>
        <column name=”id” type=”int”>
          <constraints primaryKey=”true”
```

```
nullable=”false”/>
        </column>
        <column name=”name”
type=”varchar(50)”>
          <constraints nullable=”false”/>
        </column>
    </createTable>

  </changeSet>

</databaseChangeLog>
```

然后执行命令行：

```
java -jar liquibase.jar –
defaultsFile=liquibase.properties –
changeLogFile=changelog.xml update
```

成功执行后数据库将进行相应的更新。

liquibase.properties 文件包含了建立数据库连接需要的信息，内容示例如下：

driver: com.mysql.jdbc.Driver

classpath: /liquibase/lib/mysql-connector-java-5.1.30.jar

url: jdbc:mysql://localhost/mytestdb

username: root

password: my_db_password

我们在 Jenkins 上建立一个自动部署项目，当 db-migration 项目有 push event 也就是 changelog.xml 或者 liquibase.properties 有更新时，自动触发 Jenkins 项目执行，Jenkins 项目先下载 db-migration 项目 master 分支，然后到相应目录执行“java -jar liquibase.jar –defaultsFile=liquibase.properties –

changeLogFile=changelog.xml update”来达到自动更新数据库的效果，之后所有的开发人员对数据库的更改只需要编辑 db-migration 项目的 changelog.xml 文件，而不需要直接修改数据库，这样数据库的更改就完全记录在 changelog.xml 文件中了，一目了然。不过这种方法也是不太好推广，开发人员不习惯通过这种方式去变动数据，都习惯于使用 Navicat 这种 GUI 程序去完成。

当然，我在这里只是展示了怎么使用 Liquibase 去进行数据库版本管理，还有一些其他工具你可以去尝

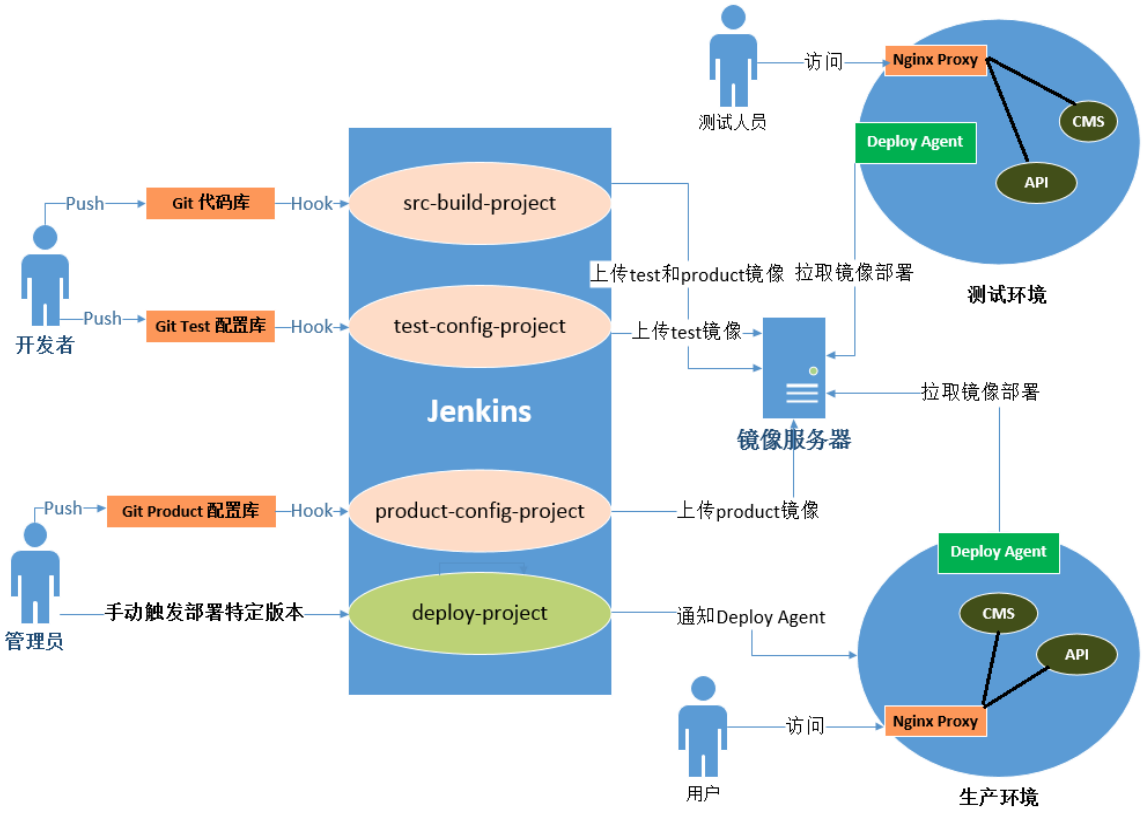
试，但大体思路应该是一样的，具体实践中可以根据需要选择其它工具。

7.5 构建部署流程

构建、部署流程就是从开发者上传代码到完成编译、打包、部署的整个自动化流程，下面我们直接以一个例子来进行阐述。

在这个示例中，源码管理系统采用 GitLab，构建工具采用 Jenkins，部署环境是 Docker。这三者的结合方式是这样的：开发人员上传代码到 GitLab，GitLab 调用回调钩子，于是

Jenkins 对应的 Job 开始构建，生成一个个 Docker 镜像。所谓的回调钩子，其实就是在 GitLab 项目里设置的一个 Jenkins Job 地址。Job 是 Jenkins 的一个项目或者叫任务，当 GitLab 调用钩子时，这个 Job 就运行起来，可以执行一系列的脚本来完成项目的构建，比如自动从 GitLab 下载源码，然后执行编译命令等，整个部署架构图大致如下：



基于 Jenkins 的自动化部署环境

上面的架构图中，每个项目在 GitLab 上分别对应 3 个 Project，一个放置代码，一个放置测试环境的配置文件，另外一个放置生产环境的配置文件。而在 Jenkins 中，需要建立 4 个 Project，其中 3 个 Project 和 GitLab 上的 Project 一一对应，当 GitLab 上

的项目有更新时，对应的 Jenkins 项目会被调用。Jenkins 的第 4 个 Project 在 GitLab 没有对应的 Project，也就没有跟任何 Web Hook 关联，需要运维管理员手动触发构建。

在我们的例子中，测试环境和生产环境目前分别对应阿里云的一台虚拟服务器，从 Docker 的角度看，这两台虚拟机就是 Docker 容器的宿主机。再回到架构图，两个环境都有 Deploy Agent，也就是一个轻量级的部署服务，运行在宿主机中。CMS，API 等应用都分别运行在自己的 Docker 容器中，每个容器都有 Nginx 和 PHP

环境，这样容器里面 Nginx 的配置就很简单了。每个环境都有一个 Nginx Proxy 容器，这个容器起到反向代理的作用，所有对 CMS 等应用的请求都先经过 Nginx Proxy 容器。新增一个后端容器应用时，只需要在 Nginx Proxy 的配置文件中添加一条代理配置即可。

部署流程详解

部署流程主要涉及到两个角色，即开发者和运维管理员，触发构建或部署分为 4 个场景，分别如下：

1)开发者将代码合并到 master 分支：

Git 的 Web Hook 调用 Jenkins 中 src-build-project 项目，Jenkins 项目将自动从 GitLab 下载对应 master 分支的代码，并从 Test 配置库和 Product 配置库下载最新配置文件，执行构建脚本，分别生成 Test 环境和 Product 环境的项目镜像并上传到 Docker 私有镜像服务器。然后 Jenkins 通知 Test 环境的 Deploy Agent 有哪些更新，Deploy Agent 根据通知内容将刚刚更新的镜像从镜像服务器拉到 Test 环境并进行部署(停止旧的容器，使用新镜像启动新容器)。

2)开发者更新置项目文件：

Git 的 Web Hook 调用 Jenkins 中 test-config-project 项目，Jenkins 自动从 GitLab 下载对应项目 master 分支的代码和 Test 配置库中的配置文件，执行预先写好的编译构建脚本，生成 Test 环境镜像并上传到 Docker 私有镜像服务器。然后 Jenkins 通知 Test 环境的 Deploy Agent 有哪些更新，Deploy Agent 根据通知内容将刚刚更新的镜像从镜像服务器拉到 Test 环境并进行部署。

3)运维管理员将配置文件上传或更新到某个配置项目：

Git 的 Web Hook 调用 Jenkins 中的 product-config-project 项目，Jenkins 自动从 GitLab 下载对应项目 master 分支的代码和 Product 配置库中的配置文件，执行构建脚本生成 Product 环境镜像并上传到 Docker 私有镜像服务器。

4)运维管理员手动触发 Jenkins 进行生产环境部署：

运维管理员登录到 Jenkins 的 Web 界面，选择 deploy-project，选择带参数的构建方式，这样就可以把需要部署的版本号等信息以参数的形式传递给 deploy-project，接着这些信

息被传递到生产环境的 Deploy Agent，根据这些信息 Deploy Agent 就会从镜像服务器拉取相应的镜像进行部署。

刚开始的时候，我们的项目不多，那么只要求这个逻辑跑得通就好了，也就是自动构建能完成就好了。对于每一个 GitLab 上的项目，我们在 Jenkins 创建对应的一个 Job，然后给 GitLab 的项目设置钩子，每次 Job 运行起来时执行脚本构建 Docker 镜像。但是我们自动构建不仅仅是生产环境，开发环境、测试环境也要自动构建，这样一来，一个 GitLab 项目就要

对应 3 个 Jenkins Job。后来，我们把项目的配置和源码分开，生产环境的配置文件包含密钥等信息，只有运维人员或者公司具有更高权限的人员才能访问。对于 GitLab 上的每个项目，我们建了一个新的项目，里面只包含配置文件，这个项目同样设置钩子，当配置文件有修改时，同样触发 Jenkins 的自动构建。再后来，我们的 V1 版本开发告一段落了，开发人员进入 V2 版本的开发，比如原来项目的名称叫 API-v1，现在又多了一个项目 API-v2，V2 可能采用的技术跟 V1 完全不一样，构建的方式也不一样，因此针对 API-v2 和其配置项目，我们又

得在 Jenkins 上创建新的 Job 来完成自动构建。

我们发现这样下去的话，GitLab 上的 Project 和 Jenkins 上的 Job 越来越多，越来越难维护。比如一个 API 项目，在 GitLab 上有两个源码 Project 了，还有两个存放配置文件的 Project，以后到了 V3，又得增加 2 个 Project。Jenkins 上的 Job 也是一样增加，Jenkins 我们是基于 Pipeline 构建的 Job，每个 Job 的构建逻辑直接通过 UI 界面编辑，比如这样：

这样每次我们要修改构建方式时，直接在这个 UI 界面修改，也就是直接在 UI 界面写代码，只是代码保存在 Jenkins 里面。我们又发现，现在看起来是两套源码管理系统了，一个是 GitLab，还有一个是 Jenkins，因为它也保存了构建代码，只是少量而已。

针对上述种种情况，我们的优化思路是，尽量减少 GitLab 上的 Project 数量和 Jenkins 上的 Job 数量，同时不直接在 Jenkins 的 UI 界面上编写构建

脚本，而是把构建脚本写进 Jenkinsfile 中，Jenkinsfile 同源码一样托管在 GitLab 上，修改它就像修改项目功能源码一样，这样代码相关的内容都在 GitLab 上统一管理，Jenkins 构建时会从 GitLab 上获取 Jenkinsfile 然后执行构建。然后对应比如 API 项目，不同版本不再分 Project，而是用 branch，不过目前 GitLab 没看到可以基于 branch 设置钩子，但是 Jenkins 有插件可以获取到更新的是哪个 branch 的代码，这样在 Jenkins 上针对 API 的 Job 也可以不同源码版本使用同一个，只是需要在 Jenkinsfile 里面区分是哪个 branch 提交了代码，执行相应

的构建逻辑即可，这样大大减少了 Jenkins 上的 Job 数量和 GitLab 上的 Project 数量，检索和维护起来更加方便。

7.6 发布方式

7.6.1 蓝绿发布

这种发布方式要求生产环境有两个，并且几乎完全相同。任何时候，蓝版处于激活状态，也就是对外提供服务，当需要发布新版本时，先在绿版环境进行最终测试验收，等到绿版完全验证没问题了，将客户请求路由

改到绿版环境，然后蓝版和绿版颜色对换。

这种部署方式的特点为：

1) 优点：部署过程中无需停机，服务一直处于可用状态；

2) 缺点：冗余的软硬件环境，当服务规模增大时，这种冗余过于耗费资源；

关于蓝绿发布，推荐 Martin 大叔的博客文章：BlueGreenDeployment

7.6.2 灰度发布

灰度发布是一种平滑过渡的发

布方式，介于黑白之间，比如 A/B Testing 是一种灰度发布方式，这种方式就是让一部分用户继续使用 A 环境，另一部分用户迁移到 B 环境，等到 B 环境的用户使用都没什么问题了，再将更多 A 环境的用户迁移到 B 环境，直到所有用户迁移到 B 环境为止。金丝雀发布也是一种灰度发布方式，它将一个应用的新版本放到生产环境去测试，一旦发现有问题及时撤回来，免得集成更多功能到生产环境后才发现更严重的问题。

关于灰度发布，推荐下面这篇文章，国内这方面的一些文章也是从这里翻译过来的:

Blue-green Deployments, A/B Testing, and Canary Releases

8. 运维和运营系统

运维和运营系统也是开发出来的，只不过它们的作用不像用户直接接触到的软件产品那样，能让用户有直观的感受。相对来说，运维系统是从更技术的角度来说，即如何维持系统的持续可用，而运营则偏向于从业务的角度来说，即如何进行业务运转。

8.1 运维系统

8.1.1 传统集群运维方式

在折腾云平台的那些年，本人使用过各种各样的工具软件，这些工具当年很火，目前也有很多人在使用，但有些渐渐地要退出历史舞台，毕竟技术的发展一日千里，我们身在其中的人有时候都觉疲惫，有一种已被潮流远远抛在后面的悲壮感。

本人当年鼓捣过的开源项目有OpenStack、VMWare、CloudStack、Ovirt、XenServer 等虚拟化相关技术，并且写过一些源码剖析文章放到博客上。因为是玩云计算，需要对计算、

存储、网络都有一定程度的了解，可以说那些年读遍了云计算相关图书和资料，较大提升了自己的技术实力和视野。

上面提到这些虚拟化平台的搭建中，我们使用到了 Puppet、SaltStack 等工具，这些工具通过定义主机或应用的状态，然后工具保证主机或应用能处于这种给定的状态，这样就达到自动化配置和部署的目的。

如果使用 Puppet，你应该看看官方文档，另外顺哥推荐一篇文章，比较通俗的：Puppet 使用方法总结。

SaltStack 当时有接触但是没有深度使用，貌似功能比 Puppet 更加强

大，需要使用的请阅读官方文档，一般来说，没有比官方文档解释更到位的地方，很多文章也是摘抄官方文档或者翻译过来的。

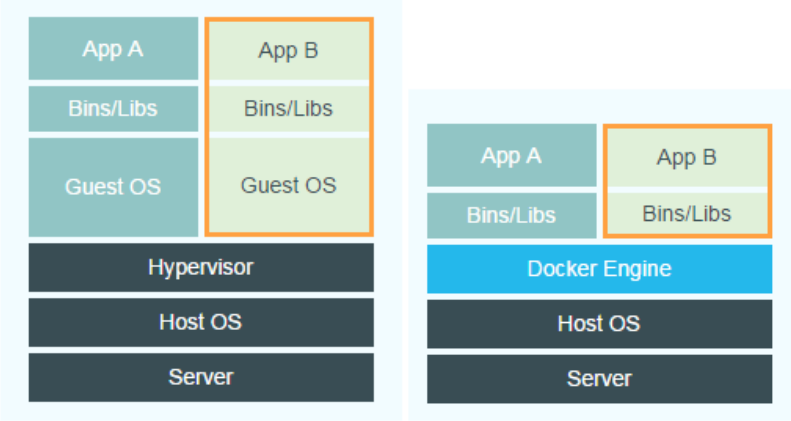
8.1.2 Docker 运维方式

当年在折腾云平台的时候，我就感觉到，云计算是非常有前景的，但是呢，前景不在我们这些非顶尖团队手中，而是属于 BAT 或者业界大牛离职创业的公司。虽然一度想深耕云计算领域成为专家，但后来因公司业务发展不好等各方面原因，我就回到了老家广西南宁，做起了应用开发架构

和管理方面的工作。

经过这几年的发展，云计算可以说普及程度很高了，那么上一节中所说的那种物理集群的搭建运维方式只是少数人接触到的东西，更多的开发人员每天都是在撸业务代码，而一些应用架构师，也不太关注底层的技术，或者必要性没有那么大，面对业务的压力只想着如何去架构好应用系统。这种情况下，应用架构师关注的就应该是云平台中的 PAAS 层，知道云计算厂商能为我们提供怎么样的服务，这些服务如何为我们的应用系统服务，如何搭配起来构建一个稳健的系统。

Docker是操作系统级别的虚拟化技术，也叫容器技术，跟传统的虚拟化技术直接虚拟一个主机不一样，这种虚拟化类似于 Linux 中的 chroot 命令，并没有为容器虚拟出一个硬件环境来然后在虚拟出来的硬件环境上安装操作系统。Docker 技术使用了 Linux 的 Cgroup 和 Namespace 技术，通过对资源使用的限制和命名空间的隔离来完成虚拟化，并且它不应该像使用虚拟机那样被使用，这方面接下来我们会细谈。那么 Docker 虚拟化和传统虚拟机技术的区别具体是怎样的呢，还是看看我从网上弄下来的这张图吧：



左边是传统虚拟机的层次结构，右边是 Docker 的层次结构，对比知道，传统虚拟机一般需要一个 Hypervisor，也就是虚拟化层，这一层虚拟出硬件环境，然后在上面装上虚拟机操作系统，也就是图中的 Guest OS，而右边的 Docker 没有 Hypervisor 层，只是有一个 Docker Engine，实际上也是一个后台进程，再上面是

Bins/Libs，所以在 Docker 容器看来，它有独立的系统库，貌似是处在独立的操作系统中，实际上所有内核调用都进入到宿主机的内核中，而左边的虚拟机则有自己的操作系统内核。从这个角度看的话，我们可以认为传统虚拟化技术是比较重的，而 Docker 是轻量级的虚拟化技术，在性能和资源使用上 Docker 有一些优势，一台物理机上往往跑几十个 Docker 容器，但同样的物理机跑几台虚拟机都费劲。当然，从隔离性上看，虚拟机隔离性更强，一台虚拟机系统挂了不会影响其它虚拟机，但 Docker 容器如果影响到内核并且是系统挂了，那就是宿主

机也挂了。从这几年来看，Docker 的发展势头更为强盛。

Kubernetes 是基于 Go 语言的容器编排系统，已得到 Docker 官方的认可，成为 Docker 编排系统事实上的标准，这个项目是由 Google 技术团队维护的，吸取了 Google 大量使用容器的经验。在使用 Docker 和 Kubernetes 时，很多人因为对概念理解的不准确，导致没有遵守最佳实践方式去使用，在这里顺哥把自己博客上的文章使用 Docker+Kubernetes 的正确姿势摘抄下来，全文如下：

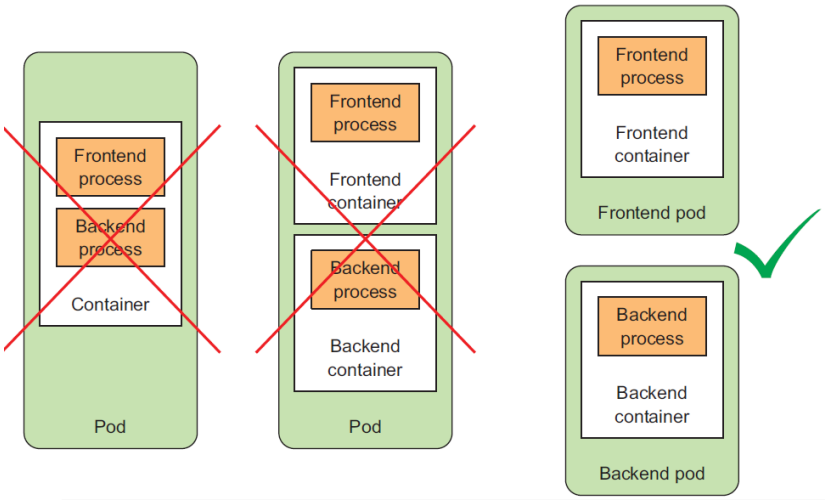
Docker 技术源于虚拟化，因此一般人都会把 Docker 容器和虚拟机相

提并论，认为容器只不过是在性能损耗上有优势。对于虚拟机，我们一般就把它当作物理机来使用，而对于容器，也可以这么做，但却不是个好的方式，较好的方式是把容器看作一个进程，每个容器运行一个应用进程。我们在宿主机上使用 Docker 提供的各种命令，可以查看到 Docker 容器的诸多状态，就像在 Linux 系统上使用 ps 命令看进程信息类似，而对于虚拟机，从宿主机上使用命令很难做到这一点。从这个角度来看，也可以得出应该每个容器运行一个进程，而不应该把容器当作虚拟机来看待的结论。最近在看《Kubernetes in action》，里

面也提到了一个容器运行多个应用进程带来的一些问题，比如管理容器里面那些进程的任务就交给了我们自己，并且容器输出到 stdout 的 log 也会混乱，难以区分哪部分 log 是哪个进程输出的。所以一个容器应该只运行一个应用进程，这个进程可能会生成其他子进程，但主进程只有这一个，这样对这个进程的生命周期管理就变成了对 Docker 容器生命周期的管理。

之前学习 Kubernetes 的时候，也理解不透 Pod 这个概念的真正目的，只知道一个 Pod 可以包含多个 Container，Pod 不能跨 Node 节点运

行。《Kubernetes in action》这本书给了一个清晰的解释，Kubernetes 里边 Pod 是作为一个最小可扩展单元的，当你把 2 个 Container 放进同一个 Pod 里面时，这两个 Container 要不同时扩展，要不都不扩展，而且这 2 个 Container 没办法运行在不同的机器上，因为 Pod 不能跨机器运行。因此一般来说，一个 Pod 只包含一个 Container，除非有些紧密耦合的 Container，才会考虑放到同一个 Pod 里，这些 Container 在扩展和收缩上都是一致的。比如对于前端服务和后台数据库服务，应该如何部署，书上给了这么一张图：



这样一来，Frontend pod 和 Backend pod 可以运行在不同的机器上，可以各自扩展而不相互影响。

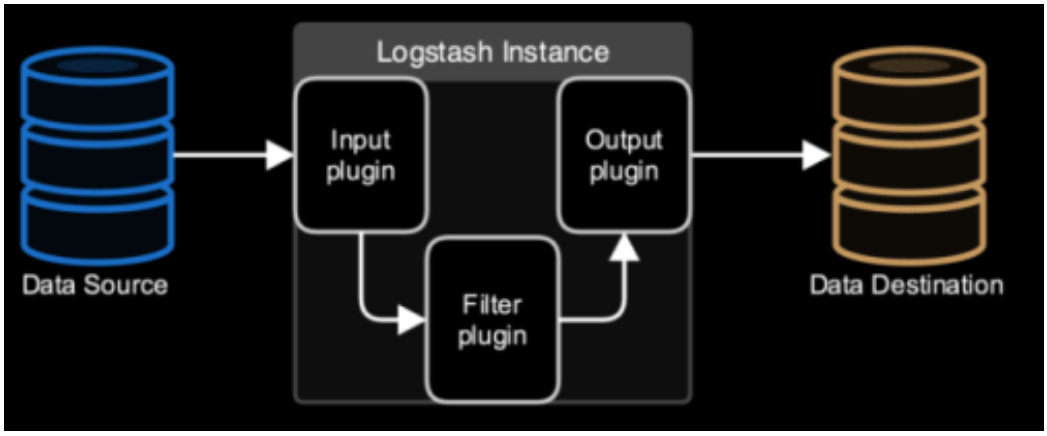
Service 在 Kubernetes 中可以看作一个包含 Pod 的集群，一个 Service 可以包含多个 Pod，对 Service 的请求可以负载均衡到其承载的某个 Pod 中去。Service 和 Pod 更详细的关系描述可参看官方文档，这里有篇文章也解释的挺不错：[Kubernetes 中的 PodIP、ClusterIP 和外部 IP]

八卦时间到了。我在写这一章的过程中，看到有人在知识星球的小密圈里问 Kubernetes 技术是否有前途。我认为 Kubernetes 出自 Google 容器团队，又已经被 Docker 官方接受，当然是非常有前途的。但是，但是你首先要对自己进行定位，如果你不是搭建平台的话，只需要大概了解这种技术，并了解在云平台购买 Kubernetes 集群后如何部署应用就够了，而且一般云平台都有使用这类产品的最佳实践文档供你参考。

8.1.3 ELK 基本原理

ELK 是 ElasticSearch、Logstash、Kibana 的简称，这三种工具往往要一起使用来构建运维系统。ElasticSearch 是一个搜索分析引擎，Logstash 是服务器端数据处理管道，可以同时接收多个数据源的数据，进行转化后发送到 ElasticSearch。Kibana 是一款可视化工具，它将 ElasticSearch 的数据以图表的形式呈现给用户。

ELK 基本的逻辑如下，这是网上的图，拿来主义就不重新画了。



上面的逻辑图中，Data Source(数据源)为各种日志数据或业务数据，Data Destination(数据目的地)为Elasticsearch，Kibana 图中没有画，Kibana 从 ElasticSearch 获取数据再显示到 UI 界面。

8.1.4 故障记录、复盘

在系统运行过程中，不可避免地

会出现各种各样的故障，我们在系统设计时就应该想到大部分的故障场景，并根据这些场景制定尽可能合理的故障恢复计划。Google 的设计理念里面就包含了设计时为故障做好准备(Design for failure)的说法。

那么现实中，我们不可能预想到所有的故障场景，并且不可能完全做到自动化故障恢复。但是在每次故障发生时，我们可以做详细的记录，分析故障产生的本质原因。然后每隔一段时间对这些故障问题进行复盘，甚至根据已发生的故障预测到未来可能产生的故障，提前做好故障修复准备。我们认为现实中没有哪个大平台

不是这样走过来的，大平台并不是从一开始就开发出来的，而是不断修复、迭代、扩展、演化出来的。所以我们一般也认为，很多大牛都是随着平台的成长而成长起来的，比如 BAT 早期的关键技术人员，经历过几百并发到亿级并发，技术功底会非常深厚，而等这些公司壮大以后才加入的普通技术人员，可能只了解到大系统里面的某一部分。当然，这也不能一概而论，但大体上会是这样。

8.2 运营系统

在运营方面，目前大家谈得比较

多的概念是数据化运营和精细化运营。数据化运营提出以数据驱动运营的理念，精细化运营强调的是对数据的处理和新需求的挖掘。虽然运营系统的研发需要研发部门去进行，但因为本书面向的不是运营人员，我也不想过多去谈自己不熟悉的领域，因此在这里我们只对运营系统做粗略地介绍。

8.2.1 业务管理

运营系统是根据业务需求建立的，不同的业务类型对应的运营系统也不太一样，但从本质上来看的话，

很多东西又是相同的，比如建立数据化运营系统的一些基本步骤。下面我们来看看如何建立一个数据化的运营系统。

首先是运营需求，需求来自运营部门或者运营人员，他们是运营系统的客户，只不过一般是内部客户。因此，运营系统的功能取决于运营部门的功能需求，运营部门人员必须梳理好运营逻辑，然后研发部门只是将这些逻辑数字化，数字化后的运营系统就像一个业务管理系统，帮助企业完成基本的业务运作。

8.2.2 运营维度

有了业务管理系统之后，要真正实现数据驱动和精细化运营，一般还需要从流量、产品、用户、内容这四个方面进行分析，这也是 GrowingIO 商务分析师范芊芸在公众号文章中提到的四个运营维度，在此我推荐这篇文章给各位：精细化数据运营：流量、用户、产品、内容四个层面。

要从上面四个维度实现数据驱动运营，首先我们需要获取各个维度需要的数据，这些数据都来自我们的业务管理系统，只不过我们需要将这些数据进行分类，然后运营恰当的算

法来处理数据，再将处理结果进行展示或者通过其它方式反馈给运营部门，运营部门根据这些反馈进行运营策略的调整。如果运营部门对各种反馈结果有了预判并且提前制定应对此策略，那么这些内容也可以自动化到运营系统中，这样的运营系统就更少需要人员干预，变得更加智能。

8.3 运维&运营式研发

目前接触到的开发人员甚至是职位比较高的技术管理者中，很少有具备运维、运营式研发思维的。开发人员普遍眼里只看到业务代码，认为

完成功能就万事大吉了，技术管理者则有一部分不是技术出身或者技术功底不怎么样却靠其他手段稀里糊涂上位的，往往会认为研发是研发，运维是运维，运营是运营。而我们说的运维、运营式研发则是要求在进行研发时，多从可运维、易运营的角度去考虑问题。如果等到系统研发出来了，才考虑运维、运营问题，则需要花更多的时间和精力投入，并且往往不能做得很好。近几年大家都说DevOps，DevOps 倡导软件开发人员和运维人员更加紧密协作来缩短软件开发周期、提高软件质量，但它除了强调工具和流程外，更强调从人员

协作的角度去解决问题，其实在我看来，软件开发人员和运营人员也需要这样的密切合作。

运维、运营式研发需要从架构上去着手，那么在做系统架构时，就需要思考运维架构，思考系统上线后如何运维，遇到系统故障时如何诊断定位问题。运营方面，则需要根据系统的业务目标，思考怎样的数据、怎样的展示方式可以让运营人员更容易看明白各项数据的含义以便对研发部门或相关人员进行正确的信息反馈。也就是说，研发部门在研发业务系统的同时，需要多思考业务系统如何帮助到 8.2.2 中运营系统的建设。

9. 项目管理

我们在项目管理中使用过禅道、Tower 和 Teambition 等软件，每个公司可以根据自身的情况或喜好选择管理工具，选择适合自己的就是最好的。

9.1 需求沟通

沟通包括技术人员之前的沟通以及技术人员与非技术人员之间的沟通，也包括对外部比如客户的沟通和对内部其它业务部门的沟通，这种

沟通一般源于业务需要。

我们在进行需求沟通的时候，应该多注意换位思考。技术人员在跟非技术人员沟通时，尽量不要满口专业术语，可以将自己所理解的内容进行适当转换或者打比方，比如可以将“给 MySQL 数据库加一行数据”转换成“添加一个数据项”，让非技术人员听起来也是通俗易懂。

有时候需求方并不清楚知道自己想要什么，也没有任何需求文档，只是模糊地说想怎么怎么做事情，这时候技术人员应尽量站在对方的角度来思考问题，向对方询问我们对他真正的需求的揣测是否正确，也就是

从对方嘴里挖出他们真正的需求，而不是双方互相抱怨，都觉得对方傻 X 不可理喻的样子。需求沟通的重要性我想我们不用怀疑，正确的需求沟通方式可以帮助我们挖掘到真正需求，否则，拿到错误的需求可能会给双方带来损失。

9.2 功能排期

对一个项目，我们一般先进行任务划分，每个任务对应若干功能，每个功能分配给相应人员去完成，最终达到责任明晰化，时间明细化，检查节点化。要达到这样的效果，需要对

各个功能进行时间明细化，也就是每个功能什么时候能够完成。同时，功能的排期还要考虑到人员之间的协作问题导致的功能优先级问题，优先级定好了，大家都按照这个顺序去完成各自的任务。

9.3 进度管理

进度的管理需要检查节点化，而且在任务进行过程中，能够通过管理系统看到各项任务目前的大致状态，比如完成 80%等等这些信息。当然进度数值是各个功能负责人自己估计出来的，不一定要求有多准确，但能

传达一个大概的信息，以便其他相互协作的人员做好相应的准备。

作为领导者，当下属人员没有按要求更新进度时，可以适当提醒，但不要逼得太急。本人早些年就碰到过那种急急火火的领导，隔三差五就来打断你，生怕你做不好事情的样子，一般来说，急急火火的人都是缺少智慧的人。平等对待下属，不要居高临下，你们只是分工不同而已，不要把下属当奴隶，要让他们知道，进度管理对整个团队来说是必要的，而不只是领导者你需要的，这样他们才没有总是被逼问的感觉，才会配合你做事。

9.4 质量管理

软件质量保证是一项工程，我们通过工程化来管理软件的质量，才能保证产出符合用户预期的产品。软件的质量属性有很多，一般来说，主要的质量属性包括以下几个：

1) 正确性，软件是否正确实现了功能需求；

2) 可用性，软件出现故障的几率是否足够小；

3) 可靠性，在正确性的基础上，异常处理情况；

4) 可维护性，出现故障时是否能够及时解决；

5) 可扩展性，是否可进行功能扩展；

6) 高效性，是否能够帮助用户高效率完成任务；

那么到底如何来保证软件的这些质量属性满足要求呢？我们认为是通过软件测试、开发迭代、回归测试来达成，因此，软件的测试需要从上面所列的六个方面去进行。开发人员和测试人员之间的配合大致如下：

1) 开发团队发布新功能，测试团队

进行测试，将问题记录到 Bug 管理系统中;

2) 开发人员得到通知后，对 Bug 进行修改并标记成已处理状态;

3) 测试人员重新测试该功能，根据实际测试结果更新 Bug 状态，如果验证未通过，回到步骤 1)，否则关闭 Bug;

当然，现实中不应该这么死板，要灵活多变，提高合作效率。比如测试人员发现前端某个按钮对齐方式出现了问题，而开发人员就坐在旁边，那么可以直接提醒开发人员修改就好了。如果还是那么死板，要测试

人员进行截图、起 Bug 标题、添加 Bug 描述、重现步骤等等然后记录到 Bug 管理系统，等开发人员得到通知了再修改这么个不痛不痒的问题，效率反而变低了，测试人员也不能追求 Bug 数量，而应该注重质量。

在项目各个阶段，可以根据 Bug 数量及其严重程度来评估以上六个质量属性的分值，这些分值在软件开发过程中可能会存在起伏的状况。比如引入一项新功能后，如果影响到了已有的部分功能，那么如果新功能质量比较差，可能使得已有功能的质量属性分值下降，需要经过开发人员不断修正，使得在交付用户使用之前，

所有质量属性的分值达到预期值即可。

9.5 文档撰写

一提到写文档，很多开发人员就开始挠头了，说不会写。我就纳了闷了，上学时候写作文都懂得瞎编，项目中按实际工作内容来写文档却不会写了。比如软件需求规格说明书、系统设计说明书、详细设计说明书等，这些文档的大概格式网上都可以搜索到，下载个模板来改改总会的吧。

写设计文档的目的是主要是为

了信息的准确传达和备份。我们有时候会是开了，但是嘴巴上说的东西往往大家都貌似不太认真听，导致在会后真正实施一些东西时又偏离了方向。如果写个文档，让大家都参与审核修改，达成一致后再按文档去实施，就不会让大家都可能有做出来的东西跟自己想象的不一样的感觉。一个团队里面思想统一的重要性，可见一斑。文档的备份作用就是，有时候我们自己去做一件事情，做过了就忘了，过一段时间已经记不得当初是为什么这么做的，文档就成为我们拿来温习的资料，就像代码注释一样。而且如果责任人离职或者去做其他项

目了，文档能让接手的人快速去了解将要接手的任务，快速完成工作交接，最终也是为了提高工作的效率。

还有一种文档的目的是作为规范，比如开发文档，大家可能知道《阿里巴巴 Java 开发手册》吧，这些文档一般是公司内部实践总结出来的，每个开发人员都遵守这些规范，这样不管是谁写的代码，别的同事在阅读时也更能容易理解，减少了大量的内部沟通成本。

八卦时间到了。虽然不少人对软件外包公司有些偏见，但就本人接触过的一些软件外包公司，他们的文档就做得不错，比如他们的开发手册，

会包含搭建一个新项目的详细步骤，这样在他们接到新项目时，很快就把项目的框架和基础代码整好了，然后就着手开发业务功能。当有新人入职时，通过阅读这个文档，马上就能去搭建并负责一个项目的研发。

9.6 人员备份

在允许的情况下，一般我们希望每个任务除了责任人了解细节外，起码还有一个人能了解大概的实现方式或技术细节，这样在责任人突然离职或者休假时，另外的人能够顶替出马。以前在我们的项目中，经常是一

个功能模块只有一个人负责，并且没有任何的文档，一旦负责人有事请假期间系统出现故障，大家都瞎了。

还记得我们前面提到的高可用架构设计吗？哈哈，拿到这里来，也就是技术人员的高可用，最好不允许单点存在，只要有单点，风险就增加。再扯一句，公司的组织架构也应该没有单点最好，否则重要的人员离职了，公司将无法正常运转，只能等到有人填补这个空缺才能回到正常轨道上来，这种情况是相当危险的。

10. 破除谜团

这个章节的内容主要缘于我看到的很多人对一些概念的误解或认识上的偏见，因此也想一吐为快，同时给也有此类误解的同学破除下谜团。可能还有很多例子，我现在已经记不清楚了，就想到什么写什么吧，技术的非技术的统统一锅煮，愿你阅读愉快！

10.1 重定向就好了吗？

曾经听到有人说，你们的接口是 HTTPS 的吧，但我们调用的时候写的是 HTTP，你们在后端实现 HTTP 到

HTTPS 重定向就好了，我们调用接口的时候还可以直接写 HTTP。

这样子真的好吗？这种方式的流程是这样的：当你从前端发一个 HTTP 请求过来后，后端发现不是 HTTPS，根据后端配置将你请求重定向到 HTTPS，这回就满足了，数据得以返回。这个过程中，第一次前端 HTTP 请求的数据已经发送过来了，只是后端限制导致重定向到 HTTPS，然后同样的数据再走一遍 HTTPS。我们使用 HTTPS 的原本目的是为了安全，给数据加密，可是之前的 HTTP 请求数据没有加密，已经暴露了，再来个 HTTPS 有什么用呢。这样不仅不安全，

第一次 HTTP 请求还白白浪费流量了有木有？所以，前端老老实实写 HTTPS 吧，不要偷懒，加个 S 有那么难吗？

那么这种重定向什么时候有用呢，本人认为在访问比如门户网站时有用。比如有人访问顺哥博客，他输入了：http://www.nndev.cn，那么我会将他重定向到这个网址：https://www.nndev.cn，之后他浏览网站内容的时候，也都是 HTTPS 协议。如果我不做重定向，那么用户输入刚才的网址时提示无法访问，用户还以为网站挂了或者不存在呢，我们总不能要求每个用户都懂得把网址改为

HTTPS 吧。

因此，我们在使用一项技术时，应该去了解这种技术的初衷是解决什么问题，然后还了解是如何解决的，如果还能知道这样解决问题后会引入哪些新的问题以便及时去规避，那就更 Perfect 一些了。

10.2 跨域是哪端的问题？

有些写 JavaScript 的同学，看到调试控制台提示跨域问题，就瞎估摸着是前端或者是后端的问题，提的问题也是让人无法回答，这往往源于对基本概念的不完全理解造成的。

首先我们要明白，同源策略是浏览器的限制，跨域请求直观举例来说就是浏览器正在运行的 JavaScript 脚本来自 A 网站，但这段 JavaScript 代码在请求后续数据时，却去请求 B 站点或其它非 A 站点的数据，其实这时候请求已经发出去了，只是返回时收到浏览器同源策略的阻止。跨域请求违反了浏览器的同源策略，但是浏览器在某种情况下却又允许跨域请求，这种情况就是服务器端在请求返回头中明确设置了允许哪些源站点可以访问自己的数据。

所以当出现跨域请求错误时，一般来说是后端没有正确设置请求的

响应头信息。如果后端接口是自己的团队或同一个公司的程序团队开发的，前端可以直接跟后端开发人员协商解决，让后端开发人员给前端请求设置正确的返回值，如果是第三方接口，或者是那些前端开发人员无法沟通协调的，那前端只能通过设置前端代理去实现跨域访问，设置代理之后，在浏览器看来，没有了跨域请求，也就不受浏览器同源策略的限制了。

更多内容，含泪推荐顺哥技术博客的文章，亲身经历，被坑老惨了：被 cors-post-json 坑到哭的经历。

10.3 搞底层的才厉害吗？

可以说在职业生涯的前 8 年，本人还一直迷恋底层技术，觉得搞底层的才更厉害，才能看透了系统的本质，就好像看懂了每个汽车零部件怎么对接后怎么运转一样。直至今天，本人还是认为这些底层技术有助于你在学习新东西时更快速地去理解消化，上层的一些设计思想有时候也是从底层的设计进行参考借鉴。

所谓底层技术，就是从硬件到操作系统内核和系统应用层面的技术，比如顺哥曾经花时间去阅读 Intel 的 3 卷本，学习段页式内存管理，学习

Windows 破解技术和内核驱动开发，折腾 IDA Pro 等反编译工具，并阅读 Linux 早期版本的内核源码，阅读 Linux 内核技术相关书籍，阅读虚拟化底层源码等等。当时本人翻译的 Windows 和 Linux 驱动架构比较的文章在这里，各位可以看看，如果想看更多相关书籍，可以加本人微信，以后会给你推荐更多。

Linux 和 Windows 设备驱动架构比较

研究那么多底层技术后就，来到上层比如应用层开发时，感觉从学习的性价比来看，似乎没有一直专注于

应用层的同学来得高。搞应用层的同学，也并不是一点都不懂底层的知识，相反他们可能不是因为底层而底层，是在需要深入的时候才去钻研相关的底层知识，这样学以致用的效果可能还更好。而如果只研究底层，没有结合应用层的一些具体问题，可能并不明白底层为何如此设计，知其然不知其所以然，随着时间流逝已经记不得学过什么了。因此，顺哥在这里给各位同学一个建议，就是要注意：学以致用、学以致用、学以致用，重要的内容说三遍，这是多么痛的领悟。希望你能付诸实践，而不是看完这本小书，还是“听过无数的道理，

依然过不好这一生”的感觉。

八卦时间到了。我一个前同事，不搞什么底层或者架构，只专注撸业务代码，照样每个月 6-8 万的工资，相信他应用层开发的效率一定很高。所以我们不能说搞底层技术的才厉害，厉害不厉害要看人，而不是看他从事的具体工作内容，哪个行业或方向没有高人和菜鸟呢。

10.4 KPI 能救你们吗？

KPI 是 Key Performance Indicator 的缩写，即关键绩效指标考核法。绩效管理可分为两大类，一类是激励型

绩效管理，另一类是管控型绩效管理。

有人以为 KPI 是万能，但 KPI 也被部分人调侃为 Kill People Idea，然而不管怎样，KPI 是无辜的，只是有人把它整歪了，很多老板对员工都是管控型心态，于是出现了比如单纯为了 KPI 而 KPI 的现象，老板您不是重视 KPI 吗，那么我们就想想办法对付您老呗，我们尽量让 KPI 考核显得漂亮些，但却不知不觉忘掉了原本的目的是什么，这是忘了初心啊各位，要不得。

尤其是在 IT 行业，有些东西很难用 KPI 来衡量，有些东西本身就很难

量化，这时候非要量化，可能会激起员工的逆反心理，进而影响团队的士气，团队成员都觉得定 KPI 的上级人员脑抽筋，一旦出现这种情况，一般都不会有什么好的结果。但是反过来，没有 KPI 的话，怎么对员工进行绩效考评呢？难道放羊式管理就合理？

在顺哥看来，KPI 是一种辅助管理的手段，而不能指望单纯靠着 KPI 就能管理好团队。管理需要懂得常识和人性，缺乏常识和人性的管理，再怎么强调 KPI 都是无济于事的。那些硬性指标、必须完成的，可以通过 KPI 进行考核，然后进行激励。另外一些

东西，则需要其他的管理方式。面对着高傲的猿(媛)们，你想以一个死的指标来衡量他们的业绩，他们会有 N 种方法对付你，而如果你懂的关心他(她)们、体谅员工的辛苦，那员工往往会有更好的表现，最终能更好地完成公司的业务目标。

11. 技术精进

交流分享促使进步，顺哥对此坚信不疑。每个人的思想都有局限性，而通过和别人分享，不仅仅是教别人东西，通过思维的碰撞后，自己在分享的同时往往也会理解得更深刻，这

就是所谓的分享即学习。既然这种做法利人利己，那何乐而不为呢。当然，有些人觉得自己太牛，分享给别人可能会让自己失去市场优势，于是藏着掖着那点本事不让人知道，对此本人也没什么可说的，能说什么呢？

11.1 技术分享

近些年来，业界各种技术会议明显增多，很多公司的技术人员都把自己公司的部分技术在会议上进行分享，有的干货还真不少。因为供职于公司时间等方面不允许的缘故，本人也是没法去参加很多高大上的会议，

在深圳的时候倒是去过几次 3w 咖啡参加一些免费的技术交流，见过青云、七牛等公司的一些技术大咖，逐渐消除神秘感，哈哈。

如果我们水平还达不到一定高度，没法作为分享者的角色参与交流，那你可以作为听众多吸收一些营养呗，当不了听众，你还可以免费下载大会 PPT 来学习，然后平时多在公司内部进行分享，等到你足够牛了，你作为某某技术大会的讲师也是有可能的。

11.2 踩坑日记

本人曾经想搞过一款 APP，专门就来记录程序猿(媛)们在开发过程中踩过的坑。我们学计算机，就是在学习它的历史，你所踩到的坑，基本上都是前人已经踩过并且早就跳出来了的，只是没有人告诉你那里有个坑让你路过的时候绕着点，或者干脆帮你把坑填好。

有一种说法是，你所经历的痛苦，是人类历史上多少人都经历过的，你的痛苦并不比他们深刻，所以不要侥情。放到这里，就是你所踩到的坑，都是别人甚至是你自己都踩过

的，然而让你痛不欲生的是，你第二次路过的时候，还是掉进坑里了，或者因为没有人告诉你那里有坑，你多花了几天的时间才跳出那个坑。因此顺哥觉得，如果程序员们都把踩坑经历记录到某个地方，等新人再掉到这个坑里时，他从前人的踩坑记录中知道前人是怎么出坑的，也许就能很快出坑了，不用扑腾那么久，壮士扑腾久了，那是对人类劳动力和时间的极大浪费。

当然，这样也不能保证我们程序员开发时就顺顺利利了，还有很多坑，别人说了可能你也不放心上，需要亲自去踩才能悟出道理，那个另当

别论。到现在，这个 APP 顺哥还没有做，各位同学，你们的踩坑记录先记录到“技术人成长”社群里面吧，哈哈。

11.3 知识付费

近两年知识付费流行起来了，罗振宇搞的得到 APP，据说估值早已经上百亿。在顺哥看来，知识付费领域目前做得还不错，知识分享者大多也是真的拿出诚意来奉献精品。其实知识付费不是新鲜事，我们从上学开始就在为知识付费了，只是形式不同而已。我有一个大胆的预想，就是随着

人们版权保护、尊重原创的增强，以后越来越难以免费拿到真正有价值的高质量内容，这些内容可能需要付费才能获取。

以前我们要学习某方面的知识，有些网上搜索到的也是零散的资料、论文等，难以系统，而市面上又缺少这方面的书籍，这方面的专家要么在身兼要职，要么没兴趣写书，要知道写书需要厚积薄发，而且收益和付出往往不成正比。那在知识付费时代，只要我们愿意花点钱，可能就能学到你所要学的知识，为知识的获取提供了很多便利。我们也可以说知识本来就在那里，只是以前没有人为你梳理

并讲解，现在有了这样的人，你需要给他们支付服务费。

极客时间号称技术人的得到APP，顺哥在上面也购买了好些个专栏，而且参与微博转发时，还中了两次奖，免费拿到两个专栏的阅读码，老开心了。觉得自己需要某方面进行提高的同学，建议可以适当付费学习一些专栏，真不是帮他们打广告，其实还是那句话：You are your greatest investment，最好的投资就是投资你自己。

话说回来，这本精致小书也是一种知识付费形式的商品，不过顺哥掏心掏肺，把自己十余年的经验心得浓

缩后和盘托出，却只需要你一中杯星巴克咖啡的花费就可以品味，所谓的最佳自我投资，恐怕也就如此了吧(有点膨胀，砖头拿小块点的哦)。

12. 后记

12.1 缘续

凡事有始有终会更妥当些，既然有前言，何不来个后记。正如前言里说的，这本小书不会带你进入技术细节，而是从较为宏观的技术角度和方向阐述个人的理解和观点。从目标读者的定位来看，我想有志于成为中小企业技术领导者的你，此刻也不需要

我来做技术细节上的分享和指导，对吧？当然，我今后的技术文章中，不排除有对某些开源项目进行源码剖析的，要看我有没有动力去写了(你的鼓励，我的动力)！

这本小书其实比较适合在架构方面有些经验的技术人员，对于行业新人，你可能看完了还是觉得云里雾里，但你不必焦虑，这是正常现象，很多事情需要你自己去经历才能体会深刻。李嘉诚曾经说过这么一句话：“世界上最浪费时间的事就是给年轻人讲经验，讲一万句不如你自己摔一跤，眼泪教你做人，后悔帮你成长，疼痛才是最好的老师。人生该走

的弯路，其实一米都少不了”。那么，对于新人来说，本书对你的意义又是什么呢？我认为它能让你提前知道了在到达你目标的路上，大致都会有哪些坑等着你，让你有目的地踩，踩得坦然，踩得有效率。

如果你看完了，觉得值，我很欣慰，同时欢迎你推荐给身边需要的朋友；如果觉得不值呢，你就算是我的赞助商，慷慨赞助我一中杯星巴克咖啡吧，也好让我今后漫漫长夜中码字时每每想到你的赞举，心底里多涌起一丝温情，谢谢。无论如何，欢迎扫描下面的二维码加我微信或者关注公众号并加入社群，你可以私信或者

在“技术人成长”社群里给我提些问题或建议，我争取拿出时间给你解惑，同时帮助我自己进步，后会有期，朋友们！

顺哥 微信号

技术人成长 公众号