图书在版编目（CIP）数据

深入剖析Kubernetes / 张磊著. -- 北京 : 人民邮电出版社, 2021.3
（图灵原创）
ISBN 978-7-115-56001-8

Ⅰ. ①深… Ⅱ. ①张… Ⅲ. ①Linux操作系统－程序设计 Ⅳ. ①TP316.85

中国版本图书馆CIP数据核字(2021)第029515号

内 容 提 要

本书基于 Kubernetes v1.18，深入剖析 Kubernetes 的本质、核心原理和设计思想。本书从开发者和使用者的真实逻辑出发，逐层剖析 Kubernetes 项目的核心特性，全面涵盖集群搭建、容器编排、网络、资源管理等核心内容，以生动有趣的语言揭示了 Kubernetes 的设计原则和容器编排理念，是一本全面且深入的 Kubernetes 技术指南。

本书适合软件开发人员、架构师、运维工程师以及具备一定服务器端基础知识且对容器感兴趣的互联网从业者阅读。

◆ 著　　张　磊
责任编辑　张　霞
责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　　印刷
◆ 开本：800×1000　1/16
印张：24.5
字数：579千字　　2021年 3 月第 1 版
印数：1－4 000册　　2021年 3 月北京第 1 次印刷

定价：99.00元

读者服务热线：(010)84084456　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

前　言

很多人在学习 Kubernetes，但也有很多人在抱怨 Kubernetes“太复杂了”。

这里的根本问题在于，Kubernetes 项目的定位是“平台的平台”（The Platform for Platform），所以其核心功能、原语服务的对象是基础平台工程师，而非业务研发人员与运维人员；它的声明式 API 设计、CRD Operator 体系，也是为了方便基础平台工程师接入和构建新基础设施能力而设计的。这就导致作为这些能力的最终使用者——业务人员，实际上跟 Kubernetes 核心定位之间存在明显的错位；而且现有的运维体系和系统，跟 Kubernetes 体系之间也存在巨大的鸿沟。

所以首先需要说明的是，本书面向的最主要受众是广大基础平台工程师。

实际上，与传统中间件从业务研发视角出发不同，云原生基础设施的革命是自底向上的。它始于谷歌 Borg/Omega 这样比“云计算”还要底层的容器基础设施构建理念，然后逐层向上对底层的计算、存储、网络进行了统一的抽象，这些抽象就是今天我们所熟知的 Pod、NetworkPolicy、Volume 等概念。出于基础设施与生俱来的高门槛和声明式应用管理理论被接纳的速度，直到 2019 年，社区对 Kubernetes 体系的认识才刚刚从“类 IaaS 基础设施”“资源管理与调度”，上升到“运维”这个维度。

所以，Kubernetes 的“复杂”是与生俱来的，这是一个专注于对底层基础设施能力进行统一抽象的“能力接入层”的价值所在。而作为基础平台工程师，你应该接受这种“复杂度”，并利用好这种“复杂度”背后各种精妙的设计，构建出真正面向用户的上层系统来服务自己的用户。

这也是为何本书会反复强调 Kubernetes 作为“标准化基础设施能力接入层”这个定位和理念，带着这样的核心思想去审视和研究 Kubernetes 中的各种功能，去讨论它的基础模型与核心设计。我们希望通过不断强调，能够让读者在这个复杂而庞大的项目中抓到主线，真正起到“授之以渔”的效果。

最后，希望你在学习完本书之后，能够理解所谓“声明式 API 和控制器模式”的本质是将底层基础设施能力和运维能力接入 Kubernetes 的一种手段。而这个手段达成的最终效果就是如今 Kubernetes 生态中数以千计的插件化能力，让你能够基于 Kubernetes 轻松构建出各种各样、面向用户的上层平台。

我们更加希望的是，你能够将本书内容“学以致用”，使用 Kubernetes 打造出下一代“以应用为中心”、高可扩展的云原生平台系统。我们希望这些平台的使用者真正能够以用户视角来描述与部署应用，而不是“强迫”自己成为“Kubernetes 专家”。

这时候，作为基础平台工程师，你可能就会更加理解“声明式 API”的真谛：把简单留给用户，把复杂留给自己。

致谢

本书的成书离不开国内云原生开源技术社区的大力支持，尤其是董殿宇、黄福临、欧阳、潘冬子四位优秀的社区志愿者，对本书的审稿、术语翻译、实践环节 Kubernetes 版本的验证等做出了非常巨大的贡献，在此表示最真挚的谢意！

目　　录

第一部分　Kubernetes 基础

第 1 章　背景回顾：云原生大事记 ……2

1.1　初出茅庐 ……2

1.2　崭露头角 ……6

1.3　群雄并起 ……8

1.4　尘埃落定 ……11

第 2 章　容器技术基础 ……17

2.1　从进程开始说起 ……17

2.2　隔离与限制 ……21

2.3　深入理解容器镜像 ……27

2.4　重新认识 Linux 容器 ……36

第 3 章　Kubernetes 设计与架构 ……47

3.1　Kubernetes 核心设计与架构 ……47

3.2　Kubernetes 核心能力与项目定位 ……51

第 4 章　Kubernetes 集群搭建与配置 ……56

4.1　Kubernetes 部署利器：kubeadm ……56

4.2　从 0 到 1：搭建一个完整的 Kubernetes 集群 ……63

4.3　第一个 Kubernetes 应用 ……73

第二部分 Kubernetes 核心原理

第 5 章 Kubernetes 编排原理……82

5.1 为什么我们需要 Pod ……82
5.2 深入解析 Pod 对象 ……91
5.3 Pod 对象使用进阶 ……96
5.4 编排确实很简单：谈谈“控制器”思想 ……109
5.5 经典 PaaS 的记忆：作业副本与水平扩展 ……112
5.6 深入理解 StatefulSet（一）：拓扑状态 ……121
5.7 深入理解 StatefulSet（二）：存储状态 ……127
5.8 深入理解 StatefulSet（三）：有状态应用实践……133
5.9 容器化守护进程：DaemonSet ……147
5.10 撬动离线业务：Job 与 CronJob ……156
5.11 声明式 API 与 Kubernetes 编程范式 ……166
5.12 声明式 API 的工作原理……175
5.13 API 编程范式的具体原理……185
5.14 基于角色的权限控制：RBAC ……196
5.15 聪明的微创新：Operator 工作原理解读……204

第 6 章 Kubernetes 存储原理……217

6.1 持久化存储：PV 和 PVC 的设计与实现原理……217
6.2 深入理解本地持久化数据卷 ……226
6.3 开发自己的存储插件：FlexVolume 与 CSI ……233
6.4 容器存储实践：CSI 插件编写指南……242

第 7 章 Kubernetes 网络原理……253

7.1 单机容器网络的实现原理 ……253
7.2 深入解析容器跨主机网络 ……260
7.3 Kubernetes 网络模型与 CNI 网络插件……268
7.4 解读 Kubernetes 三层网络方案……276
7.5 Kubernetes 中的网络隔离：`NetworkPolicy` ……285
7.6 找到容器不容易：Service、DNS 与服务发现 ……293

7.7　从外界连通 Service 与 Service 调试“三板斧”……299
7.8　Kubernetes 中的 Ingress 对象……304

第 8 章　Kubernetes 调度与资源管理……312

8.1　Kubernetes 的资源模型与资源管理……312
8.2　Kubernetes 的默认调度器……317
8.3　Kubernetes 默认调度器调度策略解析……321
8.4　Kubernetes 默认调度器的优先级和抢占机制……326
8.5　Kubernetes GPU 管理与 Device Plugin 机制……330

第 9 章　容器运行时……335

9.1　幕后英雄：SIG-Node 与 CRI……335
9.2　解读 CRI 与容器运行时……339
9.3　绝不仅仅是安全：Kata Containers 与 gVisor……343

第 10 章　Kubernetes 监控与日志……349

10.1　Prometheus、Metrics Server 与 Kubernetes 监控体系……349
10.2　Custom Metrics：让 Auto Scaling 不再“食之无味”……353
10.3　容器日志收集与管理：让日志无处可逃……358

第三部分　Kubernetes 实践进阶

第 11 章　Kubernetes 应用管理进阶……366

11.1　再谈 Kubernetes 的本质与云原生……366
11.2　声明式应用管理简介……368
11.3　声明式应用管理进阶……370
11.4　打造以应用为中心的 Kubernetes……374

第 12 章　Kubernetes 开源社区……378

结语　Kubernetes：赢开发者赢天下……382

第一部分

Kubernetes 基础

- 第 1 章　背景回顾：云原生大事记
- 第 2 章　容器技术基础
- 第 3 章　Kubernetes 设计与架构
- 第 4 章　Kubernetes 集群搭建与配置

第 1 章

背景回顾：云原生大事记

1.1 初出茅庐

如果我问你，现今最热门的服务器端技术是什么？想必你不假思索就能回答上来：当然是容器！可是，如果现在不是 2021 年而是 2013 年，你的回答还能这么斩钉截铁吗？

现在就让我们把时间拨回到 8 年前去看看吧。

2013 年的后端技术领域，已经太久没有出现过令人兴奋的东西了。曾经被人们寄予厚望的云计算技术，已经从当初虚无缥缈的概念蜕变成了实实在在的虚拟机和账单。而相比于如日中天的 AWS 和盛极一时的 OpenStack，以 Cloud Foundry 为代表的开源 PaaS 项目，却成了当时云计算技术中的一股清流。

当时，Cloud Foundry 项目已经基本度过了最艰难的概念普及和用户教育阶段，吸引了百度、京东、华为、IBM 等一大批国内外技术厂商，开启了以开源 PaaS 为核心构建平台层服务能力的变革。如果你有机会问问当时的云计算从业者，他们十有八九会告诉你：PaaS 的时代就要来了！

这种说法其实一点儿也没错，如果不是后来一个叫 Docker 的开源项目突然冒出来的话。

事实上，当时还名叫 dotCloud 的 Docker 公司，也是这波 PaaS 热潮中的一份子。只不过相比于 Heroku、Pivotal、Red Hat 等 PaaS 弄潮儿，dotCloud 公司实在是太微不足道了，而它的主打产品由于跟主流的 Cloud Foundry 社区脱节，长期以来无人问津。眼看就要被如火如荼的 PaaS 风潮抛弃，这时 dotCloud 公司却做出了这样一个决定：将自己的容器项目 Docker 开源。

显然，这个决定在当时根本没人在乎。

“容器”这个概念从来就不新鲜，也不是 Docker 公司发明的。即使在当时最热门的 PaaS 项目 Cloud Foundry 中，容器也只是其最底层、最没人关注的那一部分。说到这里，我就以当时的事实标准 Cloud Foundry 为例来解说 PaaS 技术。

PaaS 项目被大家接纳的一个主要原因，就在于它提供了一种名为“应用托管”的能力。当

时，虚拟机和云计算已经是比较普遍的技术和服务了，主流用户的普遍用法就是租一批 AWS 或者 OpenStack 的虚拟机，然后像以前管理物理服务器那样，用脚本或者手动的方式在这些机器上部署应用。

当然，在部署过程中难免会遇到云端虚拟机和本地环境不一致的问题，所以当时的云计算服务比的就是谁能更好地模拟本地服务器环境，提供更好的“上云”体验。而 PaaS 开源项目的出现就是当时这个问题的最佳解决方案。

举个例子，虚拟机创建好之后，运维人员只需在这些机器上部署一个 Cloud Foundry 项目，然后开发者只要执行一条命令就能把本地应用部署到云上，这条命令就是：

```
$ cf push _我的应用_
```

是不是很神奇？

事实上，像 Cloud Foundry 这样的 PaaS 项目，最核心的组件就是一套应用的打包和分发机制。Cloud Foundry 为每种主流编程语言都定义了一种打包格式，而 `cf push` 的作用，基本上等同于用户把应用的可执行文件和启动脚本打进一个压缩包内，上传到云上 Cloud Foundry 的存储中。接着，Cloud Foundry 会通过调度器选择一个可以运行这个应用的虚拟机，然后通知这个机器上的 Agent 下载应用压缩包并启动。

这时关键点来了，由于需要在一个虚拟机上启动多个来自不同用户的应用，Cloud Foundry 会调用操作系统的 Cgroups 和 Namespace 机制为每一个应用单独创建一个称为“沙盒”的隔离环境，然后在“沙盒”中启动这些应用进程。这样就实现了把多个用户的应用互不干涉地在虚拟机里批量地、自动地运行起来的目的。

这正是 PaaS 项目最核心的能力。这些 Cloud Foundry 用来运行应用的隔离环境，或者说“沙盒”，就是所谓的“容器”。

Docker 项目实际上跟 Cloud Foundry 的容器并没有太大不同，所以在它发布后不久，Cloud Foundry 的首席产品经理 James Bayer 就在社区里做了一次详细对比，告诉用户 Docker 实际上只是一个同样使用 Cgroups 和 Namespace 实现的“沙盒”而已，没有什么特别的“黑科技”，也不需要特别关注。

然而短短几个月，Docker 项目就迅速崛起了。它的崛起速度如此之快，以至于 Cloud Foundry 以及所有的 PaaS 社区还没来得及成为它的竞争对手，就直接被宣告出局。当时一位多年的 PaaS 从业者如此感慨道：这简直就是一场“降维打击”啊。

难道这一次，连闯荡多年的“老江湖”James Bayer 也看走眼了吗？

并没有。事实上，Docker 项目确实与 Cloud Foundry 的容器在大部分功能和实现原理上一样，可偏偏就是这剩下的一小部分不同的功能，成了 Docker 项目接下来“呼风唤雨”的不二法宝。

这个功能就是 **Docker 镜像**。

恐怕连 Docker 项目的作者 Solomon Hykes 自己当时都没料到，这个小小的创新，在短短几年内就迅速改变了整个云计算领域的发展历程。

如前所述，PaaS 之所以能够帮助用户大规模地部署应用到集群里，是因为它提供了一套应用打包的功能。可偏偏就是这个打包功能，成了 PaaS 日后不断被用户诟病的一个“软肋”。

出现这个问题的根本原因是，一旦用上 PaaS，用户就必须为每种语言、每种框架，甚至每个版本的应用维护一个打好的包。这个打包过程没有任何章法可循，更麻烦的是，明明在本地运行得好好的应用，却需要做很多修改和配置工作才能在 PaaS 里运行起来。而这些修改和配置并没有什么经验可以借鉴，基本上得靠不断试错，直到摸清了本地应用和远端 PaaS 匹配的“脾气”才能搞定。

结局就是，`cf push` 确实能一键部署了，但是为了实现这个一键部署，用户为每个应用打包的工作可谓一波三折，费尽心机。

而 Docker 镜像解决的恰恰就是打包这个根本性的问题。所谓 Docker 镜像，其实就是一个压缩包。但是这个压缩包里的内容，比 PaaS 的应用可执行文件+启停脚本的组合要丰富多了。实际上，大多数 Docker 镜像是直接由一个完整操作系统的所有文件和目录构成的，所以这个压缩包里的内容跟你本地开发和测试环境用的操作系统完全一样。

这就有意思了：假设你的应用在本地运行时，能看见的环境是 CentOS 7.2 操作系统的所有文件和目录，那么只要用 CentOS 7.2 的 ISO 做一个压缩包，再把你的应用可执行文件也压缩进去，那么无论在哪里解压这个压缩包，都可以得到与你本地测试时一样的环境。当然，你的应用也在里面！

这就是 Docker 镜像最厉害的地方：只要有这个压缩包在手，你就可以使用某种技术创建一个“沙盒”，在“沙盒”中解压这个压缩包，然后就可以运行你的程序了。

更重要的是，这个压缩包包含了完整的操作系统文件和目录，也就是包含了这个应用运行所需要的所有依赖，所以你可以先用这个压缩包在本地进行开发和测试，完成之后再上传到云端运行。

在此过程中，你完全不需要进行任何配置或者修改，因为这个压缩包赋予了你一种极其宝贵的能力：本地环境和云端环境高度一致！

这正是 Docker 镜像的精髓。

那么，有了 Docker 镜像这个利器，PaaS 里最核心的打包系统顿时就没了用武之地，最让用户头疼的打包过程中的麻烦也随之消失了。相比之下，在当今的互联网世界，Docker 镜像需要的操作系统文件和目录可谓唾手可得。

所以，你只需要提供下载好的操作系统文件与目录，然后使用它制作一个压缩包即可，这个命令就是：

```
$ docker build _我的镜像_
```

一旦镜像制作完成，用户就可以让 Docker 创建一个“沙盒”来解压这个镜像，然后在“沙盒”中运行自己的应用，这个命令就是：

```
$ docker run _我的镜像_
```

当然，`docker run` 创建的“沙盒”也是使用 Cgroups 和 Namespace 机制创建出来的隔离环境。后文会详细介绍该机制的实现原理。

所以，Docker 项目给 PaaS 世界带来的“降维打击”，其实是它提供了一种非常便利的打包机制。这种机制直接打包了应用程序运行所需要的整个操作系统，从而保证了本地环境和云端环境的高度一致，避免了用户通过“试错”来匹配不同运行环境之间差异的痛苦过程。

对于开发者来说，在终于体验到了生产力解放所带来的痛快之后，他们自然选择了用“脚”投票，直接宣告了 PaaS 时代的结束。

不过，虽然 Docker 项目解决了应用打包的难题，但如前所述，它并不能代替 PaaS 完成大规模部署应用的职责。

遗憾的是，考虑到 Docker 公司是一个与自己有潜在竞争关系的商业实体，再加上对 Docker 项目普及程度的误判，Cloud Foundry 并没有第一时间使用 Docker 作为核心依赖，去替换那套饱受诟病的打包流程。

反倒是一些机敏的创业公司纷纷在第一时间推出了 Docker 容器集群管理的开源项目（比如 Deis 和 Flynn），它们一般称自己为 CaaS（Container-as-a-Service），用来跟“过时”的 PaaS 划清界限。

在 2014 年底的 DockerCon 上，Docker 公司雄心勃勃地对外发布了自家研发的“Docker 原生”容器集群管理项目 Swarm，不仅将这波“CaaS”热推向了一个前所未有的高潮，更是寄托了整个 Docker 公司重新定义 PaaS 的宏伟愿望。

在 2014 年的这段巅峰岁月里，Docker 公司离自己的理想真的只有一步之遥。

小结

2013~2014 年，以 Cloud Foundry 为代表的 PaaS 项目逐渐完成了教育用户和开拓市场的艰巨任务，也正是在这个将概念逐渐落地的过程中，应用“打包”困难这个问题成了整个后端技术圈子的一个心病。

Docker 项目的出现则为这个根本性的问题提供了一个近乎完美的解决方案。这正是 Docker 项目刚刚开源不久就能够带领一家原本默默无闻的 PaaS 创业公司脱颖而出并迅速占领所有云计算领域头条的技术原因。

在成为了基础设施领域近 10 年难得一见的技术明星之后，dotCloud 公司在 2013 年底大胆改

名为 Docker 公司。不过，这个在当时就颇具争议的改名举动成为了日后容器技术圈风云变幻的一个关键伏笔。

1.2　崭露头角

上一节讲到，伴随着 PaaS 概念的逐渐普及，以 Cloud Foundry 为代表的经典 PaaS 项目开始进入基础设施领域的视野，平台化和 PaaS 化成了这个生态中最重要的进化趋势。

就在对开源 PaaS 项目落地的不断尝试中，这个领域的从业者发现了 PaaS 中最为棘手也最亟待解决的一个问题：究竟如何打包应用？

遗憾的是，无论是 Cloud Foundry、OpenShift，还是 Clodify，面对这个问题都没能给出一个完美的答案，反而在竞争中走向了碎片化的歧途。

就在这时，一个并不引人瞩目的 PaaS 创业公司 dotCloud 选择了将自家的容器项目 Docker 开源。更出人意料的是，就是这样一个普通到不能再普通的技术，却开启了一个名为“Docker”的全新时代。

你可能会有疑问，Docker 项目的崛起是不是偶然呢？

事实上，这个以“鲸”为注册商标的技术创业公司，最重要的战略之一就是：坚持把开发者群体放在至高无上的位置。

相比于其他正在企业级市场里厮杀得头破血流的经典 PaaS 项目，Docker 项目的推广策略从一开始就呈现出一副“憨态可掬”的亲人姿态，把每一位后端技术人员（而不是他们的老板）作为主要的传播对象。

简洁的 UI，有趣的 demo，“1 分钟部署一个 WordPress 网站”“3 分钟部署一个 Nginx 集群”，这种同开发者之间与生俱来的亲近关系，使 Docker 项目迅速成为了全世界 Meetup 上最受欢迎的一颗新星。

在过去的很长一段时间里，相较于前端和互联网技术社区，服务器端技术社区一直是一个相对沉闷而小众的圈子。在这里，从事 Linux 内核开发的极客自带“不合群”的“光环”，后端开发者“啃”着多年不变的 TCP/IP 发着牢骚，运维人员更是天生注定的幕后英雄。

而 Docker 项目给后端开发者提供了走向聚光灯下的机会。就比如 Cgroups 和 Namespace 这种已经存在多年却很少被人们关心的特性，在 2014 年和 2015 年竟然频繁入选各大技术会议的分享议题，就因为听众想知道 Docker 这个东西到底是怎么一回事儿。

Docker 项目之所以能获得如此高的关注度，一方面是因为它解决了应用打包和发布这一困扰运维人员多年的技术难题；另一方面是因为它第一次把一个纯后端的技术概念，通过非常友好的设计和封装，交到了最广大的开发者群体手里。

在这种独特的氛围烘托下，你不需要精通 TCP/IP，也无须深谙 Linux 内核原理，哪怕你只是前端或者网站的 PHP 工程师，都会对如何把代码打包成一个随处可以运行的 Docker 镜像充满好奇和兴趣。

这种受众群体的变革，正是 Docker 这样一个后端开源项目取得巨大成功的关键。这也是经典 PaaS 项目想做却没有做好的一件事情：PaaS 的最终用户和受益者，一定是为这个 PaaS 编写应用的开发者；而在 Docker 项目开源之前，PaaS 与开发者之间的关系从未如此紧密过。

Docker 解决了应用打包这个根本性的问题，同开发者有着与生俱来的亲密关系，再加上 PaaS 概念已经深入人心的完美契机，都是让 Docker 这个技术上看似平淡无奇的项目一举走红的重要原因。

一时之间，“容器化”取代“PaaS 化”成为了基础设施领域最炙手可热的关键词，一个以“容器”为中心的全新的云计算市场正呼之欲出。而作为这个生态的一手缔造者，此时的 dotCloud 公司突然宣布将公司改名为“Docker”。

这个举动在当时颇受质疑。在大家的印象中，Docker 只是一个开源项目的名字。可是现在，这个单词却成了 Docker 公司的注册商标，任何人在商业活动中使用这个单词以及鲸的 logo，都会立刻受到法律警告。

对“Docker 公司这个举动到底葫芦里卖的什么药”这个问题，我们不妨后面再做解读，因为相较于这件“小事儿”，Docker 公司在 2014 年发布 Swarm 项目才是真正的“大事儿”。

那么，Docker 公司为什么一定要发布 Swarm 项目呢?

通过我对 Docker 项目崛起背后原因的分析，你应该能发现这样一个有意思的事实：虽然通过“容器”这个概念完成了对经典 PaaS 项目的“降维打击”，但是 Docker 项目和 Docker 公司兜兜转转了一年多，还是回到了 PaaS 项目原本深耕了多年的那个战场：如何让开发者把应用部署在我的项目上。

没错，Docker 项目从发布之初就全面发力，从技术、社区、商业、市场全方位争取到的开发者群体，实际上是为此后将整个生态吸引到自家 PaaS 上的一个铺垫。只不过那时，PaaS 的定义已经全然不是 Cloud Foundry 描述的那样，而是变成了一套以 Docker 容器为技术核心、以 Docker 镜像为打包标准的全新的“容器化”思路。

这正是 Docker 项目从一开始悉心运作“容器化”理念和经营整个 Docker 生态的主要目的。

而 Swarm 项目正是接下来承接 Docker 公司所有这些努力的关键所在。

小结

本节着重介绍了 Docker 项目在短时间内迅速崛起的 3 个重要原因：

- Docker 镜像通过技术手段解决了 PaaS 的根本性问题；

- Docker 容器同开发者之间有着与生俱来的密切关系；
- PaaS 概念已经深入人心的完美契机。

1.3　群雄并起

上一节解读了 Docker 项目迅速走红的技术原因与非技术原因，也介绍了 Docker 公司开启平台化战略的野心。可是，Docker 公司为什么在 Docker 项目已经取得巨大成功之后，却执意要走回那条已经让无数先驱沉沙折戟的 PaaS 之路呢？

实际上，Docker 项目一日千里的发展势头一直伴随着公司管理层和股东们的阵阵担忧。他们心里明白，虽然 Docker 项目备受追捧，但用户最终要部署的还是他们的网站、服务、数据库，甚至是云计算业务。

这就意味着，只有那些能够为用户提供平台层能力的工具才会真正成为开发者关心和愿意付费的产品。而 Docker 项目这样一个只能用来创建和启停容器的小工具，最终只能充当这些平台项目的“幕后英雄”。

谈到 Docker 项目的定位问题，就不得不说说 Docker 公司的老朋友和老对手 CoreOS 了。

CoreOS 是一个基础设施领域的创业公司。它的核心产品是一个定制化的操作系统，用户可以按照分布式集群的方式管理所有安装了这个操作系统的节点。如此一来，用户在集群里部署和管理应用就像使用单机一样方便了。

Docker 项目发布后，CoreOS 公司很快就认识到可以把“容器”的概念无缝集成到自己的这套方案中，从而为用户提供更高层次的 PaaS 能力。所以，CoreOS 很早就成了 Docker 项目的贡献者，并在短时间内成为了 Docker 项目中第二重要的力量。

然而，这段短暂的“蜜月期”到 2014 年底就草草结束了。CoreOS 公司以强烈的措辞宣布与 Docker 公司停止合作，并直接推出了自己研制的 Rocket（后来更名为 rkt）容器。

这次决裂的根本原因正是源于 Docker 公司对 Docker 项目定位的不满足。而 Docker 公司解决这种不满足的方法，则是让 Docker 项目提供更多的平台层能力，即向 PaaS 项目进化。这显然与 CoreOS 公司的核心产品和战略发生了严重冲突。

也就是说，Docker 公司在 2014 年就已经定好了平台化的发展方向，并且绝对不会跟 CoreOS 在平台层面开展任何合作。这样看来，Docker 公司在 2014 年 12 月的 DockerCon 上发布 Swarm 的举动，也就一点儿都不突然了。

相较而言，CoreOS 是依托于一系列开源项目（比如 Container Linux 操作系统、Fleet 作业调度工具、systemd 进程管理和 rkt 容器）一层层搭建起来的平台产品，Swarm 项目则是以一个整体对外提供集群管理功能。Swarm 的最大亮点是它完全使用 Docker 项目原本的容器管理 API 来完成集群管理，比如：

单机 Docker 项目：

```
$ docker run "我的容器"
```

多机 Docker 项目：

```
$ docker run -H "我的 Swarm 集群 API 地址" "我的容器"
```

所以，在部署了 Swarm 的多机环境中，用户只需要使用原先的 Docker 指令创建一个容器，Swarm 就会拦截这个请求并处理，然后通过具体的调度算法找到一个合适的 Docker Daemon 运行起来。

这种操作方式简洁明了，了解过 Docker 命令行的开发者也很容易掌握。所以，这样一个“原生”的 Docker 容器集群管理项目一经发布，就受到了 Docker 用户群的热捧。相比之下，CoreOS 的解决方案就显得非常另类，更不用说用户还要去接受完全让人摸不着头脑、新造的容器项目 rkt 了。

当然，Swarm 项目只是 Docker 公司重新定义 PaaS 的关键一环而已。在 2014 年到 2015 年这段时期，Docker 项目的迅速走红催生了一个非常繁荣的 Docker 生态。在这个生态里，围绕 Docker 在各个层次进行集成和创新的项目层出不穷。

此时已经大红大紫到“不差钱”的 Docker 公司，开始及时借助这波浪潮通过并购来完善自己的平台层能力。其中最成功的案例莫过于收购 Fig 项目。

要知道，Fig 项目基本上是只靠两个人全职开发和维护的，可它当时在 GitHub 上是热度堪比 Docker 项目的明星。

Fig 项目之所以广受欢迎，是因为它首次提出了“容器编排”（container orchestration）的概念。其实，“编排”在云计算行业不算是新词，它主要是指用户通过某些工具或者配置来完成一组虚拟机以及关联资源的定义、配置、创建、删除等工作，然后由云计算平台按照指定逻辑来完成的过程。

在容器时代，“编排”显然就是对 Docker 容器的一系列定义、配置和创建动作的管理。而 Fig 的工作实际上非常简单：假如现在用户需要部署的是应用容器 A、数据库容器 B、负载均衡容器 C，那么 Fig 就允许用户把 A、B、C 这 3 个容器定义在一个配置文件中，并且可以指定它们之间的关联关系，比如容器 A 需要访问数据库容器 B。

接下来，你只需要执行一条非常简单的指令：

```
$ fig up
```

Fig 就会把这些容器的定义和配置交由 Docker API 按照访问逻辑依次创建，你的一系列容器就都启动了；而容器 A 与容器 B 之间的关联关系，也会通过 Docker 的 Link 功能写入 hosts 文件的方式进行配置。更重要的是，你还可以在 Fig 的配置文件里定义各种容器的副本个数等编排参数，再加上 Swarm 的集群管理能力，一个活脱脱的 PaaS 就呼之欲出了。

Fig 项目被收购后改名为 Compose，它成了 Docker 公司到目前为止第二大受欢迎的项目，直到今日依然被很多人使用。

在当时的容器生态里，还有很多令人眼前一亮的开源项目或公司。比如，专门负责处理容器网络的 SocketPlane 项目（后来被 Docker 公司收购）、专门负责处理容器存储的 Flocker 项目（后来被 EMC 公司收购）、专门给 Docker 集群做图形化管理界面和对外提供云服务的 Tutum 项目（后来被 Docker 公司收购），等等。

一时之间，整个后端和云计算领域的聪明才俊都汇集在了这头“鲸”的周围，为 Docker 生态的蓬勃发展献出了自己的智慧。

除了这个异常繁荣、围绕 Docker 项目和公司的生态，还有一股势力在当时可谓风头正劲，这就是老牌集群管理项目 Mesos 和它背后的创业公司 Mesosphere。

Mesos 作为 Berkeley 主导的大数据套件之一，是大数据火热时最受欢迎的资源管理项目，也是跟 Yarn 项目厮杀得难解难分的实力派选手。

不过，大数据所关注的计算密集型离线业务，其实并不像常规的 Web 服务那样适合用容器进行托管和扩容，对应用打包也没有强烈需求，所以 Hadoop、Spark 等项目到现在也没在容器技术上投下更大的赌注；但是对于 Mesos 来说，天生的两层调度机制让它非常容易从大数据领域抽身，转而支持受众更广的 PaaS 业务。

在这种思路的指导下，Mesosphere 公司发布了一个名为 Marathon 的项目，而这个项目很快就成为了 Docker Swarm 的有力竞争对手。

虽然不能提供像 Swarm 那样的原生 Docker API，但 Mesos 社区拥有一项独特的竞争力：超大规模集群的管理经验。

早在几年前，Mesos 就已经通过了万台节点的验证，2014 年之后又在 eBay 等大型互联网公司的生产环境中被广泛使用。而这次通过 Marathon 实现了诸如应用托管和负载均衡的 PaaS 功能之后，Mesos+Marathon 的组合实际上进化成了一个高度成熟的 PaaS 项目，同时还能很好地支持大数据业务。

所以，在这波容器化浪潮中，Mesosphere 公司不失时机地提出了一个名为“DC/OS”（数据中心操作系统）的口号和产品，旨在让用户能够像管理一台机器那样管理一个万级别的物理机集群，并且使用 Docker 容器在这个集群里自由地部署应用。而这对很多大型企业来说具有非同寻常的吸引力。

此时再审视当时的容器技术生态，就不难发现 CoreOS 公司竟然显得有些尴尬了。它的 rkt 容器完全打不开局面，Fleet 集群管理项目更是少有人问津，CoreOS 完全被 Docker 公司压制了。

处境同样不容乐观的似乎还有 Red Hat，作为 Docker 项目早期的重要贡献者，Red Hat 也是因为对 Docker 公司的平台化战略不满而愤然退出。但此时，它只剩下 OpenShift 这个跟 Cloud

Foundry 同时代的经典 PaaS 一张牌可以打，跟 Docker Swarm 和转型后的 Mesos 完全不在一条赛道上。

那么，事实果真如此吗？

2014 年注定是一个神奇的年份。就在这一年的 6 月，基础设施领域的翘楚谷歌公司突然发力，正式宣告了 Kubernetes 项目的诞生。这个项目不仅挽救了当时的 CoreOS 和 Red Hat，还如同当年 Docker 项目的横空出世一样，再一次改变了整个容器市场的格局。

小结

本节介绍了 Docker 公司平台化战略的来龙去脉，阐述了 Docker Swarm 项目发布的意义和它背后的设计思想，介绍了 Fig（后来的 Compose）项目如何成为了继 Docker 之后最受瞩目的新星。

同时回顾了 2014 年至 2015 年如火如荼的容器化浪潮里群雄并起的繁荣姿态。在这次生态大爆发中，Docker 公司和 Mesosphere 公司依托自身优势率先占据了有利位置。

但是，更强大的挑战者即将在不久后纷纷登场。

1.4 尘埃落定

上一节介绍了随着 Docker 公司一手打造出来的容器技术生态在云计算市场中站稳脚跟，围绕 Docker 项目进行的各个层次的集成与创新产品也如雨后春笋般出现在这个新兴市场中。而 Docker 公司不失时机地发布了 Docker Compose、Swarm 和 Machine“三件套”，在重新定义 PaaS 的方向上迈出了最关键的一步。

这段时间也正是 Docker 生态创业公司的春天，大量围绕 Docker 项目的网络、存储、监控、CI/CD，甚至 UI 项目纷纷出台，也涌现出了很多像 Rancher、Tutum 这样在开源与商业上均取得了巨大成功的创业公司。

2014 年至 2015 年，整个容器社区可谓热闹非凡。

但在这令人兴奋的繁荣背后浮现出了更多的担忧。其中最主要的负面情绪是对 Docker 公司商业化战略的种种顾虑。

事实上，很多从业者也看得明白，Docker 项目此时已经成为 Docker 公司一个商业产品。而开源只是 Docker 公司吸引开发者群体的一个重要手段。不过这么多年来，开源社区的商业化其实都是类似的思路，无非是高不高调、心不心急的问题罢了。

而真正令大多数人不满意的是，Docker 公司在 Docker 开源项目的发展上始终保持着绝对的权威和发言权，并在多个场合用实际行动挑战了其他玩家（比如 CoreOS、Red Hat，甚至谷歌和

微软）的切身利益。

那么，此时大家的不满也就不再是在 GitHub 上发发牢骚这么简单了。

相信容器领域的很多老玩家听说过，Docker 项目刚刚兴起时，谷歌也开源了一个在内部使用了多年、经过生产环境验证的 Linux 容器：lmctfy（Let Me Container That For You）。

然而，面对 Docker 项目的强势崛起，这个对用户没那么友好的谷歌容器项目几乎毫无招架之力。所以，知难而退的谷歌公司向 Docker 公司表示了合作的愿望：关停这个项目，和 Docker 公司共同推进一个中立的**容器运行时**（container runtime）库作为 Docker 项目的核心依赖。

不过，Docker 公司并没有认同谷歌这个明显会削弱自己地位的提议，还在不久后独自发布了一个容器运行时库 Libcontainer。这次匆忙的、由一家主导并带有战略性考量的重构，成了 Libcontainer 被社区长期诟病代码可读性差、可维护性不强的一个重要原因。

至此，Docker 公司在容器运行时层面上的强硬态度，以及 Docker 项目在高速迭代中表现出来的不稳定和频繁变更的问题，开始让社区叫苦不迭。

这种情绪在 2015 年达到了一个小高潮，容器领域的其他几位玩家开始商议“切割”Docker 项目的话语权。而“切割”的手段也非常经典，那就是成立一个中立的基金会。

于是，2015 年 6 月 22 日，由 Docker 公司牵头，CoreOS、谷歌、Red Hat 等公司共同宣布，Docker 公司将 Libcontainer 捐出，并改名为 RunC 项目，交由一个完全中立的基金会管理，然后以 RunC 为依据，大家共同制定一套容器和镜像的标准和规范。

这套标准和规范就是 OCI（Open Container Initiative）。OCI 的提出意在将容器运行时和镜像的实现从 Docker 项目中完全剥离。这样做一方面可以改善 Docker 公司在容器技术上一家独大的现状，另一方面也为其他玩家不依赖 Docker 项目构建各自的平台层能力提供了可能。

不过，不难看出，OCI 的成立更多的是这些容器玩家出于自身利益进行干涉的一个妥协结果。所以，尽管 Docker 是 OCI 的发起者和创始成员，它却很少在 OCI 的技术推进和标准制定等事务上扮演关键角色，也没有动力去积极地推进这些所谓的标准。这也是迄今为止 OCI 组织效率持续低下的根本原因。

眼看着 OCI 并没能改变 Docker 公司在容器领域一家独大的现状，于是谷歌和 Red Hat 等公司把第二件武器摆上了台面。

Docker 之所以不担心 OCI 的威胁，原因就在于它的 Docker 项目是容器生态的事实标准，而它所维护的 Docker 社区也足够庞大。可是，一旦这场斗争被转移到容器之上的平台层，或者说 PaaS 层，Docker 公司的竞争优势便立刻捉襟见肘了。

在这个领域里，像谷歌和 Red Hat 这样的成熟公司，都拥有深厚的技术积累；而像 CoreOS 这样的创业公司，也拥有像 etcd 这样被广泛使用的开源基础设施项目。可是 Docker 公司呢？它只有一个 Swarm。

所以这次，谷歌、Red Hat 等开源基础设施领域玩家共同牵头成立了一个名为 CNCF（Cloud Native Computing Foundation）的基金会。这个基金会的目的其实很容易理解：以 Kubernetes 项目为基础，建立一个由开源基础设施领域厂商主导的、按照独立基金会方式运营的平台级社区，来对抗以 Docker 公司为核心的容器商业生态。

为了打造这样一条围绕 Kubernetes 项目的“护城河”，CNCF 社区需要至少确保两件事情：

- Kubernetes 项目必须能够在容器编排领域取得足够大的竞争优势；
- CNCF 社区必须以 Kubernetes 项目为核心，覆盖足够多的场景。

我们先来看看 CNCF 社区是如何解决 Kubernetes 项目在编排领域的竞争力的问题的。

在容器编排领域，Kubernetes 项目需要面对来自 Docker 公司和 Mesos 社区两个方向的压力。不难看出，Swarm 和 Mesos 实际上分别从不同的方向讲出了自己最擅长的故事：Swarm 擅长跟 Docker 生态的无缝集成，而 Mesos 擅长大规模集群的调度与管理。

这两个方向也是大多数人做容器集群管理项目时最容易想到的两个出发点。也正因为如此，Kubernetes 项目如果继续在这两个方向上做文章恐怕就不太明智了。所以这一次，Kubernetes 选择的应对方式是：Borg。

如果你看过 Kubernetes 项目早期的 GitHub Issue 和 Feature，就会发现它们大多来自 Borg 和 Omega 系统的内部特性，这些特性落到 Kubernetes 项目上，就是 Pod、sidecar 等功能和设计模式。

这就解释了为什么 Kubernetes 发布后，很多人“抱怨”其设计思想过于“超前”：Kubernetes 项目的基础特性并不是几个工程师突然“拍脑袋”想出来的，而是谷歌公司在容器化基础设施领域多年来实践经验的沉淀与升华。这正是 Kubernetes 项目能够从一开始就避免同 Swarm 和 Mesos 社区同质化的重要手段。

于是，CNCF 接下来的任务就是如何把这些先进的思想通过技术手段在开源社区落地，并培育出一个认同这些理念的生态。这时，Red Hat 发挥了重要作用。

当时，Kubernetes 团队规模很小，能够投入的工程能力也十分紧张，而这恰恰是 Red Hat 的长处。更难得的是，Red Hat 是世界上为数不多的、能真正理解开源社区运作和项目研发真谛的合作伙伴。

所以，Red Hat 与谷歌联盟的建立，不仅保证了 Red Hat 在 Kubernetes 项目上的影响力，也正式开启了容器编排领域“三国鼎立”的局面。

这时再重新审视容器生态的格局，就不难发现 Kubernetes 项目、Docker 公司和 Mesos 社区这三大玩家的关系已经发生了微妙的变化。

其中，Mesos 社区与容器技术的关系更像是“借势”，而不是该领域真正的参与者和领导者。而 Mesos 所属的 Apache 基金会一直以来运营方式都相对封闭，很少跟基金会之外的世界进行过多的交互和流动。“借势”的关系，加上封闭的社区形态，最终导致了 Mesos 社区虽然技术最为

成熟，却在容器编排领域鲜有创新。

这也是为何谷歌公司很快就把注意力转向了动作更加激进的 Docker 公司。

有意思的是，Docker 公司对 Mesos 社区的看法，与本章前面的分析也是类似的。所以从一开始，Docker 公司就把应对 Kubernetes 项目的竞争摆在了首要位置：一方面，不断强调“Docker Native”的重要性；另一方面，与 Kubernetes 项目在多个场合进行了直接的碰撞。

不过，这次竞争的发展态势很快就超出了 Docker 公司的预期。

Kubernetes 项目并没有跟 Swarm 项目展开同质化的竞争，所以“Docker Native”的说辞并没有太大的杀伤力。相反，Kubernetes 项目让人耳目一新的设计理念和号召力，很快就构建出了一个与众不同的容器编排与管理的生态。

就这样，Kubernetes 项目在 GitHub 上的各项指标开始一骑绝尘，将 Swarm 项目远远地甩在了身后。

有了这个基础，CNCF 社区就可以放心地解决第二个问题了。

在已经囊括了容器监控事实标准的 Prometheus 项目之后，CNCF 社区迅速在成员项目中添加了 Fluentd、OpenTracing、CNI 等一系列容器生态的知名工具和项目。

在看到了 CNCF 社区对用户表现出来的巨大吸引力之后，大量的公司和创业团队也开始专门针对 CNCF 社区而非 Docker 公司制定推广策略。

面对这样的竞争态势，Docker 公司决定更进一步。在 2016 年，Docker 公司宣布了一个令所有人震惊的计划：放弃现有的 Swarm 项目，将容器编排和集群管理功能全部内置到 Docker 项目当中。

显然，Docker 公司意识到了 Swarm 项目目前唯一的竞争优势就是跟 Docker 项目的无缝集成。那么，如何让这种优势最大化呢？那就是把 Swarm 内置到 Docker 项目当中。

实际上，从工程角度来看，这种做法的风险很大。内置容器编排、集群管理和负载均衡能力，固然可以让 Docker 项目的边界直接扩大到一个完整的 PaaS 项目的范畴，但这种变更带来的技术复杂度和维护难度，从长远来看对 Docker 项目是不利的。

不过，在当时的大环境下，Docker 公司的选择恐怕也带有一丝孤注一掷的意味。

Kubernetes 的应对策略则是反其道而行之，开始在整个社区推行“民主化”架构，即从 API 到容器运行时的每一层，Kubernetes 项目都为开发者暴露了可以扩展的插件机制，鼓励用户通过代码的方式介入 Kubernetes 项目的每一个阶段。

Kubernetes 项目这个变革的效果立竿见影，很快在整个容器社区中催生出了大量基于 Kubernetes API 和扩展接口的二次创新工作，比如：

- 目前热度极高的微服务治理项目 Istio；

- 被广泛采用的有状态应用部署框架 Operator；
- 还有像 Rook 这样的开源创业项目，它通过 Kubernetes 的可扩展接口，把 Ceph 这样的重量级产品封装成了简单易用的容器存储插件。

就这样，在这种鼓励二次创新的整体氛围当中，Kubernetes 社区在 2016 年之后得到了空前的发展。更重要的是，不同于之前局限于“打包、发布”这样的 PaaS 化路线，容器社区的这一次繁荣是一次完全以 Kubernetes 项目为核心的“百花齐放”。

面对 Kubernetes 社区的崛起和壮大，Docker 公司也不得不面对自己豪赌失败的现实。但早前拒绝了微软的天价收购，Docker 公司实际上已经没有什么回旋的余地了，只能选择逐步放弃开源社区而专注于自己的商业化转型。

所以，从 2017 年开始，Docker 公司先是将 Docker 项目的容器运行时部分 Containerd 捐赠给 CNCF 社区，标志着 Docker 项目已经全面升级为一个 PaaS 平台；紧接着，Docker 公司宣布将 Docker 项目改名为 Moby，然后交给社区自行维护，而 Docker 公司的商业产品将占有 Docker 这个注册商标。

Docker 公司这些举措背后的含义非常明确：它将全面放弃在开源社区同 Kubernetes 生态的竞争，转而专注于自己的商业业务，并且通过将 Docker 项目改名为 Moby 的举动，将原本属于 Docker 社区的用户转化成了自己的客户。

2017 年 10 月，Docker 公司出人意料地宣布，将在自己的主打产品 Docker 企业版中内置 Kubernetes 项目，这标志着持续了近两年之久的“编排之争”至此落下帷幕。

2018 年 1 月 30 日，Red Hat 宣布斥资 2.5 亿美元收购 CoreOS。

2018 年 3 月 28 日，这一切纷争的“始作俑者”——Docker 公司的 CTO Solomon Hykes 宣布辞职。曾经纷纷扰扰的容器技术圈子，至此尘埃落定。

小结

容器技术圈子在短短几年里出现了很多变数，但很多事情其实也在情理之中。就像 Docker 这样一家创业公司，在通过开源社区的运作取得了巨大的成功之后，不得不面对来自整个云计算产业的竞争和围剿。而这个产业的垄断特性，对于 Docker 这样的技术型创业公司其实天生就不友好。

在这种局势下，接受微软的天价收购，在大多数人看来是一个非常明智和实际的选择。可是 Solomon Hykes 却多少有一些理想主义，既然不甘于“寄人篱下”，那他就必须带领 Docker 公司去对抗来自整个云计算产业的压力。

只不过，Docker 公司最后选择的对抗方式是将开源项目与商业产品紧密绑定，打造了一个极端封闭的技术生态。而这其实违背了 Docker 项目与开发者保持亲密关系的初衷。相比之下，

Kubernetes 社区正是以一种更加温和的方式承接了 Docker 项目的未竟事业，即以开发者为核心，构建一个相对“民主”和开放的容器生态。

这也是为何说 Kubernetes 项目的成功其实是必然的。

现在，我们很难想象如果 Docker 公司最初选择了跟 Kubernetes 社区合作，如今的容器生态又将会是怎样的一番景象。不过可以肯定的是，Docker 公司在过去几年里的风云变幻，以及 Solomon Hykes 的传奇经历，都已经在云计算的历史中留下了浓墨重彩的一笔。