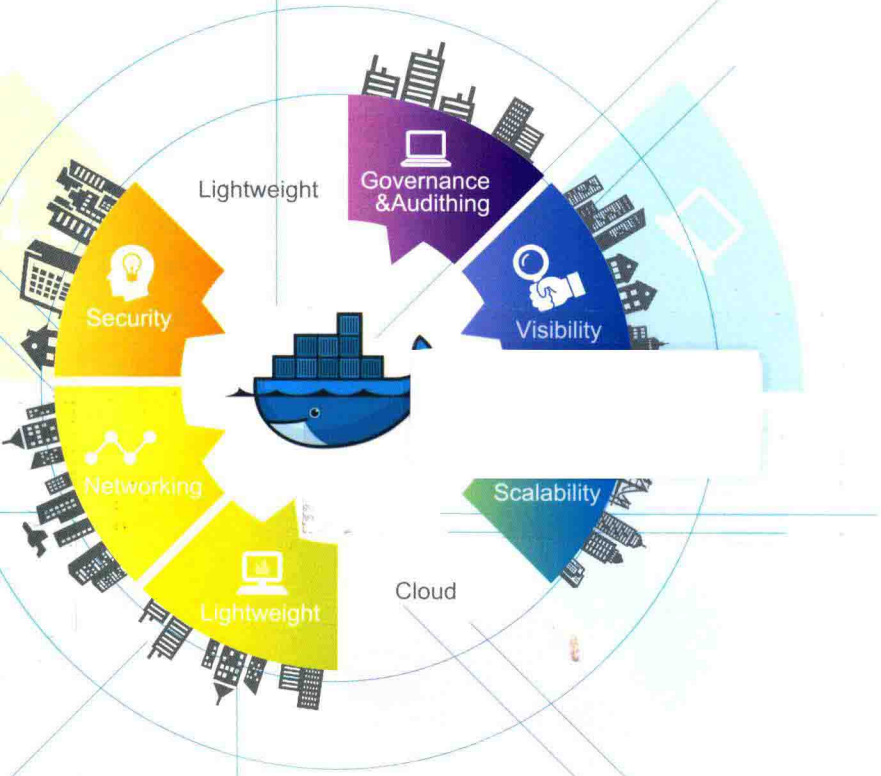
Broadview®
www.broadview.com.cn

Docker全攻略

集作者多年Docker实战经验，内容具有较高的实战价值和深度

张涛 编著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Docker全攻略

张涛 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Docker 是一个充满挑战性和趣味性的开源项目，它彻底释放了 Linux 虚拟化的威力，极大地缓解了云计算资源供应紧张的局面。与此同时，Docker 也成倍地降低了云计算供应成本，让应用的部署、测试和开发变成了一件轻松、高效和有意思的事情。

本书由浅入深，从基本原理入手，系统讲解了 Docker 的原理、构建与操作。同时讲解了 Docker 在实际生产环境中的使用，最后还探讨了 Docker 的底层实现技术和基于 Docker 的相关开源技术。前 4 章为基础内容，供用户理解 Docker 和配置 Docker 运行环境。第 5 章到第 9 章为 Docker 基本操作，主要讲解了 Docker 命令操作实例和 Docker 命令实现原理。第 10 到第 12 章为 Docker 高级操作，介绍了 Docker 内核相关知识点，适合高级用户参考其内核运行机制。第 13 章到第 15 章给出了 Docker 典型应用场景和实践案例。

本书既适用于具备 Linux 基础知识的 Docker 初学者，也适用于具有开发功底想深入研究 Docker 内核的高级用户。书中所提供的实践案例，可供在实际生产环境部署时借鉴。

图书在版编目（CIP）数据

Docker 全攻略 / 张涛编著. —北京：电子工业出版社，2016.4
ISBN 978-7-121-28238-6

Ⅰ. ①D… Ⅱ. ①张… Ⅲ. ①Linux 操作系统—程序设计 Ⅳ. ①TP316.89

中国版本图书馆 CIP 数据核字（2016）第 042573 号

责任编辑：安　娜
印　　刷：北京京科印刷有限公司
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：33.75　　字数：952 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
印　　数：3000 册　　定价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

自从 Docker 横空出世以来，有关 Docker 的讨论就一直非常热烈，并涌现出了一批优秀的文档。但在 Docker 开发方面，却大多限于 Docker 如何使用，更倾向于运维层面。因此使很多人误认为 Docker 就是虚拟化容器，最多再有点资源限制操作。但这却是 Docker 众多功能中的冰山一角，并非全部。

本书的写作目的不仅是在技术层面深入分析 Docker 背后的技术原理和设计思想，更想结合笔者所在团队的工作经验，理清 Docker 的技术脉络和内核原理，同时附加 Docker 生态圈的实际案例，以期对开发运维人员、容器云服务提供商以及 Docker 技术爱好者在技术选型、技术路线规划上有所帮助。

笔者所在团队从 2014 年开始关注 Docker，并且开始深入研究 Docker。当时 Docker 还是一个基于 Local 模式的虚拟化工具，并没有当前丰富的生态圈技术。我们团队基于 Docker 打造了一款企业级的私有云平台，是国内最先使用 Docker 的一批人。除了感受到 Docker 在效率方面所带来的革命性提高外，还不得不忍受 Docker 与企业级虚拟化工具之间的差距。

但随着 Docker 的不断发展和完善，我们真真切切地感受到了 Docker 是如何从一个鲜有耳闻的名词变成了虚拟化首选工具。基于 Docker 的云平台解决方案如雨后春笋般涌现，基于 Docker 的中国本地化解决方案也开始逐步出现，各类国内镜像加速器和仓库也开始层出不穷。

当前，中国互联网已进入了“互联网+”时代，云平台即将进入爆发式发展的时代。在“互联网+”这样一个以软件连接人与人、连接人与企业、连接企业与企业的时代，Docker 自然是技术选型的不二之选。因此在可预见的未来，中国 IT 界自当需要大量的 Docker 开发人员，而本书的目的就是希望可以为有志做 Docker 开发的工程师提供一些帮助。

本书分为三部分，以 Docker1.7 源码为基础。由浅入深，从介绍 Docker 的使用入手，逐步演进到 Docker 的功能实现，最后是 Docker 的内核机制。

第一部分包括第 1 章至第 5 章。第 1 章至第 3 章介绍了 Docker 技术背景，使读者能够初步了解 Docker 的来龙去脉和 Docker 的未来发展方向。第 4 章介绍了在不同平台中如何安装 Docker，而第 5 章是第一部分的重点章节，介绍了 Docker 各种命令的使用方式。

第二部分包括第 6 章至第 9 章。第二部分深入讲解了 Docker 各功能实现方法以及相关技术。第 6 章介绍了 Docker 所有功能的实现方式，例如 Docker 网络命令、Docker 安全命令、Docker 资源命令等。Dockerfile 作为构建镜像的唯一方式，其用法在第 7 章中有详细介绍，而第 8 章则介绍了 Dockerfile 的最佳实践。第 9 章通过实际的 Nginx 部署案例来介绍 Docker 的使用。

第三部分包括第 10 章至第第 15 章。第三部分深入讲解了 Docker 内核机制。第 10 章，第 11 章和第 12 章分别从 Docker 生命周期、namespace 和资源管理等方面入手，完完整整地讲解了 Docker

的内核运行方式。第 13 章、第 14 章和第 15 章则通过基于 Amazon 和 Coreos 的实际部署案例介绍了 Docker 生态圈。

对于能够编写这么一本用来介绍 Docker 内核的书，笔者感到非常荣幸。笔者在此向所有在编写本书期间提供帮助的朋友们表示最诚挚的谢意。没有朋友们的帮助，我一人是无法顺利完成这项工作的。

感谢与我一直奋战在云平台开发一线的各位兄弟，是你们为我提供了众多写作灵感，并且提供了大量的 Docker 案例。

感谢 Jack Gao（高建国）、Tracy LI（李星旺）、Echo Guo（郭静田）在我写作本书期间提供的无私帮助，正是由于你们的帮助，才能支持我一直写完此书。

在这里需要特别感谢我的妻子彭欢。她毫无怨言的忍受了我写书时无法陪伴她的日日夜夜，同时也感谢她无私照顾我的日子，最后感谢上天将她送给了我。

最后感谢所有人的努力，才得以让此书顺利面世。本人希望此书可以为 Docker 技术布道之路贡献一份微薄的力量。

张涛

2016 年 2 月 21 日

北京

目　录

第一篇　Docker 简介

第二篇　Docker 基础

第三篇 Docker 进阶

第四篇 Docker 生态圈

第五篇 Docker 案例

第一篇
Docker 简介

本篇包括

- 第 1 章　Docker 的前世今生
- 第 2 章　Docker 现状
- 第 3 章　Docker 将如何改变未来

第 1 章　Docker 的前世今生

1.1　什么是 LXC

在引入 Docker 之前，或许有必要先聊聊 LXC。在 Linux 使用过程中，大家很少会接触到 LXC，因为 LXC 对于大多数人来说仍然是一个比较陌生的词汇。那为什么我们要在开篇之时，先聊这个陌生的概念呢？这是因为 LXC 是整个 Docker 运行的基础。

众所周知，CPU、内存、I/O、网络等都称之为系统资源，而 Linux 内核有一套机制来管理其所拥有的这些资源，这套机制的核心被称之为 CGroups 和 Namespaces。

CGroups 可以限制、记录、调整进程组所使用的物理资源。比如说：使用 CGroups 可以给某项进程组多分配一些 CPU 使用周期。同样也可以通过 CGroups 限制某项进程组可使用的内存上限，一旦达到上限，内核就会发出 Out Of Memory 错误。同时 CGroups 也具有记录物理资源使用情况的功能，比如 CGroups 调用 cpuacct 子系统就可以记录每个进程所使用的内存数量、CPU 时间等数据。正因为 Linux 有了 CGroups 资源管理机制，内核虚拟化才变成了可能。

Namespaces 则是另外一个重要的资源隔离机制。Namespaces 将进程、进程组、IPC、网络、内存等资源都变得不再是全局性资源，而是将这些资源从内核层面属于某个特定的 Namespace。在不同的 Namespace 之间，这些资源是相互透明、不可见的。比如说，A 用户登录系统后，可以查看到 B 用户的进程 PID。虽说 A 用户不能杀死 B 用户的进程，但 A 和 B 却能相互感知。但假如 A 用户在 Namespace-A 中，B 用户在 Namespace-B 中，虽然 A 和 B 仍然共存于同一个 Linux 操作系统当中，但 A 却无法感知到 B。在这种情况下，Linux 内核不但将 Namespace 相互隔离，而且将所分配的资源牢牢固定在各自空间之中。

而 LXC 就是基于 Linux 内核通过调用 CGroups 和 Namespaces，来实现容器轻量级虚拟化的一项技术，与此同时，LXC 也是一组面向 Linux 内核容器的用户态 API 接口。用户通过 LXC 提供的资源限制和隔离功能，可以创建一套完整并且相互隔离的虚拟应用运行环境。

本书后续章节提到的 Docker，就是采用 LXC 技术来创建容器的工具，因此才说：LXC 是 Docker 运行的基础，而 Docker 则是 LXC 的杀手级应用。

1.2　Docker 为什么选择了 AUFS

Docker 为什么选择了 AUFS？回答这个问题，需要从 AUFS 的起源谈起。AUFS 原名为 Another UnionFS，从名称可以看出，AUFS 是对 UnionFS 的补充。UnionFS 是一个堆栈式的联合文件系统，打包在 Linux 发行版中。但 UnionFS 很久不进行更新，同时也存在一些不稳定的问题，因此在 UnionFS 的基础之上进行功能完善，推出了一个新版本，名为 AUFS。

当 AUFS 发布之后，最新版的 UnionFS 又吸收了 AUFS 的很多功能，并随之发布在最新的 UnionFS 版本之中。 AUFS 也同步更名为 Advanced Multi Layered Unification Filesystem。

虽然名称发生了变更，但 AUFS 本质上仍是堆栈式的联合文件系统。AUFS 的功能简单说就是，可以将分布在不同地方的目录挂载到同一个虚拟文件系统当中。

这句话不长，但理解起来颇需一些脑力。没关系，我们慢慢来分析这句话。

首先我们将思路切换到 Linux 启动阶段。典型的 Linux 启动时，首先加载 bootfs（Boot File System）目录。这个目录里面包括 Bootloader 和 kerenl。Bootloader 用来加载启动 kerenl。当 kerenl 成功加载到内存中后，bootfs 就会释放掉，kerenl 随之开始加载 rootfs。

rootfs（Root File System）包含的是 Linux 系统中标准的/dev、/proc、/bin、/etc 等文件。因为 rootfs 是后续 kerenl 启动的基础，对于 kerenl 来说异常重要，因此此时 kerenl 将 Rootfs 加锁——设为 readonly。 在只读权限下，kerenl 进行一系列的检查操作。当 kerenl 确认 rootfs 包含的文件正确无误后，将 readonly 改为 readwrite（可读可写），以后用户就可以按照正确的权限对这些目录进行操作了。

说到这里，就轮到到 AUFS 登场了。当 Docker 利用 LXC 虚拟化出来一个容器之后，就相当于购买了一台裸机，有内存、CPU、硬盘，但没有操作系统。Docker 参考 Linux 的启动过程，将一个 readonly 权限的 bootfs 挂载到容器文件系统中，然后通过 AUFS，再将 readonly 权限的 rootfs 添加到 bootfs 之上，当 rootfs 检查完毕之后，再将用户所要使用的文件内容挂载到 rootfs 之上，同样是 readonly 权限。每次挂载一个 FS 文件层，并且每层之间只会挂载增量（在这里大家可以借助于 SVN 进行理解，相当每个 FS 层都是 SVN 提交上去的数据增量）。

这些文件层就是堆栈式文件系统中所保存的数据。将不同的文件层挂载到同一个文件系统中的文件系统，就是联合文件系统；而 AUFS 就是用来管理、使用这些文件层的文件系统，因此也称之为高级多层次统一文件系统（Advanced Multi Layered Unification Filesystem）。

但是每个 FS 层都是 readonly 权限，那么容器内部如何向这些文件写入数据呢？其实当 Docker 利用 AUFS 加载完最高一层之后，会在最上面再添加一个 FS 层，而这个层是 readwrite 权限。容器内部的应用，对当前文件系统所有的写操作（包括删除）都会保存在这个 FS 层当中，而当容器向 Docker 发出 commit 命令后，Docker 会将这个 FS 层中的数据作为单独一个文件层保存到 AUFS 之中。

而一个镜像（image）就可以理解为：特定 FS 层的集合。所以可以看出镜像的层次关系，处于下层的 image 是上层 image 的父类，而没有父类 image 的就是 baseimage。因此需要从 image 启动 container 时，Docker 会依次加载 baseimage 和父类 image，而用户所有的操作就都保存在最高层的 readwrite 的 layer 中。

通过将镜像“分隔”为 AUFS 的文件层，使得所有容器都可以共享文件层，且不会发生写冲突。但在 Docker 中，所有的镜像都是只读的，所有的镜像也都不保存用户信息，只会用于新建和复制。而对于容器而言，其所看到的所有文件都是可读写的，只不过所有的写操作都被保存在最上层的文件层当中。

Docker 正是通过 AUFS 的这些特性，解决了容器初始化和写时复制问题，所以 Docker 选择 AUFS 作为其第二个核心组件。

1.3　Docker 是如何产生的

在前两节中，我们介绍了 LXC 和 AUFS 两项技术，这两项技术是 Docker 运行的基础。本节我

们就开始介绍 Docker。

Docker 最初诞生于 dotCloud 公司，这是一家于 2010 年成立的，专注于 PAAS（Platform as a Service）平台的创业型公司。PAAS 概念可以说是 SAAS 概念的升级版，而且直接面向广大程序员，旨在减少软件开发周期中最烦琐、最耗时的环境准备环节。所以这个概念一经推出，就立刻得到了程序员的热捧。

但在 2010 年的时候，市面上已经有一些科技巨头进入了 PAAS 这个领域，比如 IBM、Amazon、Google、VMWare 和微软。这些巨头纷纷推出了自己的产品，比如 Amazon 的 EC2、google 的 GAE、VMWare 的 Cloud Foundry、IBM 的 Blue Mix 以及微软的 Azure。所以在 2010 年的时候，dotCloud 的日子并不是很好过，它虽然拿到了 1000 万美元的风投，但仍然举步维艰。在经过深思熟虑之后，dotCloud 创始人 Solomon Hykes 提议，将他们的核心虚拟化产品 Docker 开源。

山穷水尽疑无路，柳暗花明又一村。

Docker 一经开源，马上得到业界程序员的热烈吹捧。这个基于 Linux Container 技术的虚拟化产品大大降低了容器技术的使用门槛，程序员所希望的免费、轻量级、可移植、虚拟化和与语言无关、封装后的镜像可以随处部署和迁移等各种苛刻的要求，在 Docker 上面都一一得到了满足。

哇！开发界，测试界和生产环境三界统一了！

Docker 得到了极大的关注度，大有席卷市场的意思。这时，各大科技巨头也马上改变策略，先后宣布将在各自的云平台中支持 Docker，就连微软都宣布一定会在 Windows 环境中原生支持 Docker。

至此，Docker 在市场上站稳了脚跟。而 dotCloud 公司也顺势在 2014 年 8 月卖给了德国的 cloudControl 公司，Solomon Hykes 等人也全职开始维护 Docker 开源社区，为广大使用 Docker 的公司提供技术支持。

Docker 经过多个版本的迭代发展，到目前为止，已经成为云计算领域最受欢迎的开源项目。理论上说，只要有 Linux 的地方，就能运行 Docker。

Docker 在其自身的发展过程中，不断地进行完善，目前稳定的版本是 Docker 1.7。本书后续内容也将以 Docker 1.7 为基础展开陈述。

第2章 Docker现状

2.1 Docker应用范围

在简单介绍了Docker的前世今生之后，那么Docker最佳的应用范围在哪里呢？将Docker放到什么地方，可以最大程度的发挥Docker的威力呢？

云平台，当仁不让的是Docker最佳的使用场景。在Docker推出之前，导致云平台发展缓慢的一个原因就是：云平台之间标准规范不统一。各个平台之间无法做到相互兼容，相互对接。

各家企业的产品在制定云产品开发计划之时，总是要考虑如何兼容不同的云平台。但每个云平台都有各自独立的资源管理策略，网络映射策略和内部依赖关系。所以一款产品从一个公有云平台“迁移”到另外一个公有云平台几乎是不可能的。

但Docker的出现，打破了这种局面。Docker弥合了各个云平台之间的差异，同当年的Java一样，Docker屏蔽硬件层的差异，提供了统一的用户应用层。通过Docker，企业用户所提供的产品可以自由地在“混合云”之间移动，而这种迁移所付出的成本却是极低的。

无论是封闭的私有云环境，还是开发的公有云环境。只要满足Docker的运行条件，企业所发布的产品就可以对外提供服务，同时不会因为产品运行时的环境差异而导致功能上有所差异。

所以云平台是Docker最佳使用范围之一。除了云平台，Docker还在潜移默化地改变着另外一个领域——Devops。

Docker的出现，令开发人员和运维人员都眼前一亮，同时也间接地推动了Devops的推广。在没有Docker之前，企业内部为了提高产品开发效率，降低各部门之间的沟通成本，制定了很多的流程和规范，但结果往往都不尽如人意。究其原因就在于无论流程和规范多么严谨，总归是需要有人来执行，但牵涉到的部门和人员越多，出错的概率就越高，因此带来的沟通成本就无法降低。

Docker能作为一个封闭的运行环境在各部门之间流转，这无形当中就降低了各部门之间的沟通成本。只要各部门使用相同的数据镜像，就不会出现环境差异，同样也就不会出现代码运行差异。这使得在企业内部，产品开发团队可以将精力最大化地集中于产品本身，而不是流程。

但因Docker而受益的不止是Devops，产品管理同样也因为Docker的出现而悄然发生了变化。

企业应用级别的产品通常有严格的管理程序，日志和监控是必不可少的两个功能。在Docker出现之前，各种产品的运行机制都各不相同，需要针对每款产品定制开发监控功能。但Docker的出现，改善了这种状况。因为Docker拥有统一的数据规范和接口规范，所以只需针对Docker开发一套管理和监控功能就可以。同时得益于Docker的虚拟化、资源隔离、数据统一的特性，企业用户可以更容易、更高效地进行产品服务编排。

综上所述，结合目前Docker在实际环境之中的运用，如果一个企业需要将产品发布在云平台之上，同时也想提高开发效率，最好还能方便管理产品，那么使用Docker绝对是一个好的选择。

2.2 Docker 的优缺点

既然 Docker 可以给产品开发带来这么多的益处，那么 Docker 是否就是完美无瑕的呢？本节我们将讨论 Docker 的优缺点，从正反两面来衡量使用 Docker 是否是一个好的选择。

虽然我们还没有从 Docker 原理的角度来分析 Docker，但目前，我们也可以得出下面的结论。

1. Docker 资源利用率比传统虚拟机要高

传统的虚拟机从硬件层面就已经发生了隔离，虚拟机之间是互不可见的，资源也独立不共享。但 Docker 当中所有的容器共享同一个系统内核，共享所有的 CPU 和内存（这里的共享是指在默认条件下，在后面的章节中，我们会讲到如何为容器指配 CPU 节点和内存），所以在 Docker 当中，几乎不存在资源浪费，利用率比传统虚拟机要高很多。

2. Docker 支持跨节点部署

Docker 之所以受到 PaaS 厂商的追捧，其中一个原因就是标准的数据模型。通过 Docker 定义的标准镜像数据格式，使得所有的镜像都可以自由迁移，而且不需要关心 Docker 所处的实际操作系统和实际物理硬件环境。如同 Java 一样，Docker 宣传“一次构建，自由分发”。这种特性不但给目前的云平台架构体系带来了一些新思路，也实实在在地影响了云计算的发展方向。

3. 版本可控，组件可复用

Docker 通过 AUFS 文件系统的特性，使得镜像之间不再是相互隔离的。镜像与镜像之间也可以产生“若即若离”的松耦合关系。镜像也不再是“铁板一块”，而是变成了多层文件的联合体。这些文件层作为数据元素，通过不同的“组合”就可以产生不同的镜像。Docker 还借助 AUFS 文件系统，为每个镜像制作了标签，每个标签就代表了唯一的镜像。通过这些唯一的标签，Docker 为每个镜像提供了历史回溯，可以随时加载特定标签的镜像。

4. 共享镜像

作为一款优秀的开源软件，Docker 秉承了开源软件的理念，通过 Docker 构建出的镜像可以自由进行分发。这就意味着全世界所有的用户都可以通过 Docker hub 下载使用从操作系统到 Web 容器各种各样的镜像。而所有的用户也可以自由构建镜像，并且上传到 Docker hub 供其他用户使用。

5. 轻量，易维护

因为 Docker 是基于 Linux 内核进行虚拟化操作，并且所有的容器都是共享内核资源的，所以从内核角度来看，Docker 可以认为是 Linux 当中一个普通的进程。因此 Docker 就可以做到非常轻量级，启停几乎都是在一瞬间完成。同时受益于 Golang 语言的协程，Docker 可以轻松应对并发处理请求。所以无论是维护还是监控，Docker 都非常容易。

以上是目前业界针对 Docker 所总结出的优点，但金无足赤，Docker 难道只有优势，没有缺点吗？下面就来看看 Docker 的劣势。

1. 宿主资源没有完成做到隔离

Docker 所有容器都是共享宿主机资源的，但每个容器所使用的 CPU、内存、文件系统、进程、网络等都是相互隔离开的。

容器之间的隔离度看似很高，但其实还有一些内核资源未被隔离开来。例如/proc、/sys 等这些目录还在共享使用，SELinux、syslog 这些内核功能也未被隔离，/dev 里面的设备同样也未被隔离。而这些未隔离的资源导致用户对 Docker 的稳定性提出了疑问，担心 Docker 同 Node.js 一样快而不

稳，无法应对复杂场景。

虽然在实际环境当中，这些问题还没有爆发出来，但这的确是 Docker 实际存在的一个缺点。

2. Golang 语言尚未成熟

Golang 语言相对于 C、C++、Java 而言，是一门较新的操作系统语言。Golang 的语法在 2014 年才基本稳定下来，而 Docker 全部都是采用 Golang 来编写的，用户会不由地担心 Docker 会不会有先天性的不足。用户的这种担心并不是多余的，开发语言尚不稳定，难保 Golang 以后不会发生大的处理机制变更。到那时，Docker 又该如何应对？

3. Docker 虽已开源，但事实上被一家公司所控制

Docker 技术目前被 dotCloud 公司所推广。世界上所有的公司都是以盈利为目标的，现在 Docker 是开源的，但用户担心 Docker 会不会一直免费下去。比如到某一日，Docker 开源版本不再持续更新了，而只能使用黑匣子编译出的二进制发行版，那时又该怎么办？

目前 dotCloud 公司已经推出了面向企业用户的咨询、支持和服务这些企业收费服务，所以这种担心看来也不是杞人忧天。

综上所述，使用 Docker 有利有弊。但作为现阶段的 Docker 来说，提高 Docker 稳定性和大力推广 Docker 技术是 dotCloud 的当务之急。只要 Golang 不拖后腿，Docker 应该会继续推出更多功能。而收费应该还是很久远的事情。即便以后收费了，如果使用 Docker 可以带来更大的经济价值，付费购买商业服务也是可以考虑的事情。

第3章 Docker将如何改变未来

Docker将“云”由虚幻变得有些现实

在Docker还未发布之前，几乎所有的云平台都是采用虚拟机的架构来部署应用。传统虚拟机产品，例如VMwar、VirtualBox和KVM等包装的是一个完整的“机器”，有独立的内核调度，有独立的资源管理。但Docker却没有走这样的老路子，Docker包装的是一个应用。

通过Docker提供的标准规范，用户更容易通过自动化工具完成打包和部署，所以相对于VM方案而言，Docker更适合搭建一个开放的PaaS平台。

而这也是Docker风靡世界、炙手可热的一个原因。

从Docker于2013年正式开源，到目前的2015年开始大规模推广，世界上许多的公司都开始分享它们与Docker之间相互集成的故事。虽然每家公司都有各自的集成方案，但都承认Docker改变了它们原有的业务体系。

在VM刚刚推广时，各大厂商只是将VM作为虚拟化工作站来使用，很少人会意识到若干年之后，VM会作为云平台的基础解决方案而异军突起。同样Docker也在走VM曾经走过的路子，目前各大互联网基础提供商还在将Docker作为云计算平台的计算单元使用，但随着Google已经开始将Docker用作其Paas平台的基础构成单元，Docker已经开始悄然改变云平台的基础架构。

得益于Docker所提供的更细粒度的资源管理，公有云厂商，例如AWS、Azure这些厂商可以对其提供的IaaS服务做更细粒度的管理，以进一步提高硬件使用率。而随着硬件使用率的提高，又反过来可以降低公有云的使用成本，从而为广大的创业公司降低了跨入云计算平台的门槛。使得云平台不再是高高在上，可望而不可即的概念，而变为实实在在可以自由使用的基础平台。

相比较而言IaaS平台侧重于提供硬件，PaaS平台更侧重于提供应用。而事实上，应用千奇百怪，无法统一，这就造成了PaaS厂商更期待一个统一、规范的标准部署环境。Docker的标准化规范，弥合了开发、业务和运维三方的需求差异，可以无缝地将最终产品在开发环境、测试环境和生产环境之间自由移动。这使得PaaS以更简洁的方式发布应用变成了可能。通过Docker，开发人员不再需要为处理各种开发、测试、生产环境的差异而花费大量精力，他们可以将一个干净的开发环境直接迁移到生产环境，而不必担心各种依赖和配置问题。因此开发者越来越多地考虑以Micro Service（微服务）的方式来实现他们的应用。

对于创业公司来说，Docker的出现降低了配置依赖和环境依赖，使得产品开发所涉及的各个环节都可以将精力集中于产品本身，而不是各种依赖。同时，将Docker作为持续集成和持续交付的核心，也极大地提高了产品开发效率，降低了产品迭代间隔期。无论是对于创业公司，还是对于

最终客户而言，都是利大于弊。

Docker 虽然前景广阔，但除了 Google、AWS 等有雄厚的技术实力可以保障大规模 Docker 集群运行之外，市面上还缺乏大规模运行和维护 Docker 集群的经验。与此同时，也缺乏跨平台管理 Docker 的工具。同时 Docker 自身也处于快速生长期，无论是在功能性还是在稳定性上面，离大家的期望值都还有一些距离。目前的 Docker 正如幼年的 Java 一样，虽然年轻，但却已经得到了大家的一致认可，只是缺乏历练和经验。