容器技术系列

Docker 技术入门与实战

第 2 版

杨保华 戴王剑 曹亚仑 编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

本书从 Docker 基本原理开始，深入浅出地讲解 Docker 的构建与操作，内容系统全面，可帮助开发人员、运维人员快速部署 Docker 应用。本书分为四大部分：基础入门、实战案例、进阶技能、开源项目，第一部分（第 1～8 章）介绍 Docker 与虚拟化技术的基本概念，包括安装、镜像、容器、仓库、数据卷，端口映射等；第二部分（第 9～16 章）通过案例介绍 Docker 的应用方法，包括与各种操作系统平台、SSH 服务的镜像、Web 服务器与应用、数据库的应用、各类编程语言的接口、容器云等，还介绍了作者在容器实战中的思考与经验总结；第三部分（第 17～21 章）是一些进阶技能，如 Docker 核心技术实现原理、安全、高级网络配置、libernetwork 插件化网络功能等；第四部分（第 22～28 章）介绍与容器开发相关的开源项目，包括 Etcd、Docker Machine、Docker Compose、Docker Swarm、Mesos、Kubernetes 等。

第 2 版参照 Docker 技术的最新进展对全书内容进行了修订，并增加了第四部分专门介绍与容器相关的知名开源项目，利用好这些优秀的开源平台，可以更好地在生产实践中受益。

Docker 技术入门与实战（第 2 版）

出版发行：机械工业出版社（北京市西城区百万庄大街22号　邮政编码：100037）

责任编辑：吴怡　　责任校对：

印　　刷：　　版　　次：2017年3月第1版第1次印刷

开　　本：186mm×240mm　1/16　　印　　张：

书　　号：ISBN 978-7-111-　　定　　价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

第 2 版前言

自云计算步入市场算起，新一代计算技术刚好走过了第一个十年。

过去十年里，围绕计算、存储、网络三大基础服务，围绕敏捷服务和规模处理两大核心诉求，新的概念、模式和工具争相涌现。这些创新的开源技术成果，提高了整个信息产业的生产效率，降低了应用信息技术的门槛，让“互联网+”成为可能。

如果说软件定义网络（SDN）和网络功能虚拟化（NFV）让互联网络的虚拟化踏上了崭新的阶段，那么容器技术的出现，毫无疑问称得上计算虚拟化技术的又一大创新。从 Linux Container 到 Docker，看似是计算技术发展的一小步，却是极为重要的历史性突破。容器带来的不仅仅是技术体验上的改进，更多的是新的开发模式、新的应用场景、新的业务可能……

容器技术自身在快速演进的同时，笔者也很欣喜地看到，围绕着容器的开源生态系统越发繁盛。Docker 三剑客 Machine、Compose、Swarm 相辅相成，集团作战；搜索巨人则推出 Kubernetes，领航新一代容器化应用集群平台；更不要说 Mesos、CoreOS，以及其它众多的开源工具。这些工具的出现，弥补了现有容器技术栈的不足，极大丰富了容器技术的应用场景，增强了容器技术在更多领域的竞争力。

在这一版中，笔者参照最新进展对全书内容进行了修订完善，并专门增加了第四部分来介绍跟容器相关的知名开源项目。利用好这些优秀的开源平台，可以更好地在生产实践中受益。

成书之际，Docker 发布了 1.13 版本，带来了更稳定的性能和更多有趣的特性。

再次感谢容器技术，感谢开源文化，希望开源技术能得到更多的支持和贡献！

最后，IBM 中国研究院的刘天成、李玉博等帮忙审阅了部分内容，在此表达最深厚的感谢！

笔者

2016 年 11 月于北京

目录

第一部分 基础入门

本部分共有 8 章内容，笔者将介绍 Docker 和容器的相关基础知识。

第 1 章将介绍容器和 Docker 的来源以及它与现有的虚拟化技术，特别是 Linux 容器技术的关系。

第 2 章将介绍 Docker 的三大核心概念，以及如何在常见的操作系统环境中安装 Docker。

第 3 章到第 5 章通过具体的示例操作，讲解使用 Docker 的常见操作，包括镜像、容器和仓库。

第 6 章将剖析如何在 Docker 中使用数据卷来保存持久化数据。

第 7 章将介绍如何使用端口映射和容器互联来方便外部对容器服务的访问。

第 8 章将介绍如何编写 Dockerfile 配置文件，以及使用 Dockerfile 来创建镜像的具体方法和注意事项。

第1章 初识容器与 Docker

如果说主机时代大家比拼的是单个服务器物理性能（如 CPU 主频和内存）的强弱，那么在云时代，最为看重的则是凭借虚拟化技术所构建的集群处理能力。

伴随着信息技术的飞速发展，虚拟化技术早已经广泛应用到各种关键场景中。从上世纪 60 年代 IBM 推出的大型主机虚拟化，到后来 Xen、KVM 为代表的虚拟机虚拟化，再到现在以 Docker 为代表的容器技术，虚拟化技术自身也在不断进行创新和突破。

传统来看，虚拟化既可以通过硬件模拟来实现，也可以通过操作系统软件来实现。而容器技术则更为优雅，它充分利用了操作系统本身已有的机制和特性，可以实现远超传统虚拟机的轻量级虚拟化。因此，有人甚至把它称为“新一代的虚拟化”技术，并将基于容器打造的云平台亲切称为“容器云”。

Docker 毫无疑问正是众多容器技术中的佼佼者，是容器技术发展过程中耀眼的一抹亮色。

那么，什么是 Docker？它会带来哪些好处？它跟现有虚拟化技术又有何关系？

本章首先会介绍 Docker 项目的起源和发展过程，之后会为大家剖析 Docker 和相关容器技术，以及它在 DevOps 等场景带来的巨大便利。最后，还将阐述 Docker 在整个虚拟化领域中的技术定位。

1.1 什么是 Docker

1. Docker 开源项目背景

Docker 是基于 Go 语言实现的开源容器项目，诞生于 2013 年初，最初发起者是 dotCloud 公司。Docker 自开源后受到广泛的关注和讨论，目前已有多个相关项目（包括 Docker 三件套、Kubernetes 等），逐渐形成了围绕 Docker 容器的生态体系。

由于 Docker 在业界造成的影响力实在太大，dotCloud 公司后来也直接改名为 Docker Inc，并专注于 Docker 相关技术和产品的开发。

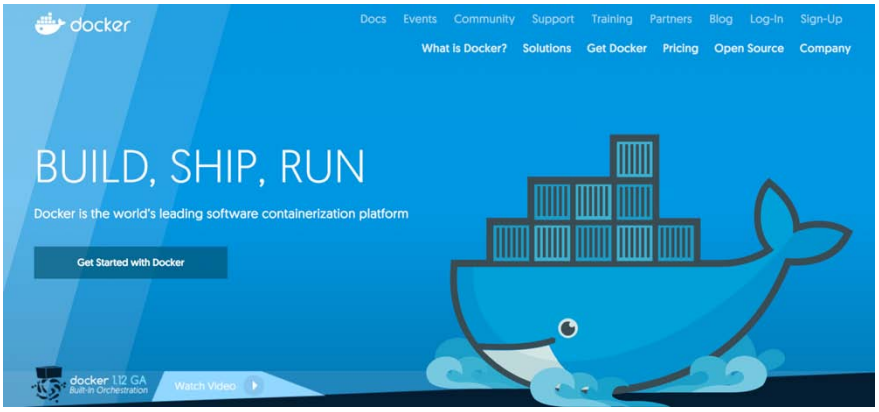


图 1-1 Docker 官方网站

Docker 项目已加入了 Linux 基金会，并遵循 Apache，2.0 协议，全部开源代码均在 https://github.com/docker/docker 上进行维护。在 Linux 基金会最近一次关于“最受欢迎的云计算开源项目”的调查中，Docker 仅次于 2010 年发起的 OpenStack 项目，并仍处于上升趋势。

现在主流的 Linux 操作系统都已经支持 Docker。例如，红帽公司的 RHEL 6.5/CentOS 6.5 往上的操作系统、Ubuntu 14.04 往上的操作系统，都已经在软件源中默认带有 Docker 软件

包。Google 公司宣称在其 PaaS（Platform as a Service）平台及服务产品中广泛应用了 Docker 容器。IBM 公司跟 Docker 公司达成了战略合作伙伴关系。微软公司在其云平台 Azure 上加强了对 Docker 的支持。公有云提供商亚马逊也推出了 AWS EC2 Container 服务，提供对 Docker 和容器业务的支持。

Docker 的构想是要实现“Build, Ship and Run Any App, Anywhere”，即通过对应用的封装（Packaging）、分发（Distribution）、部署（Deployment）、运行（Runtime）生命周期进行管理，达到应用组件级别的“一次封装，到处运行”。这里的应用组件，既可以是一个 Web 应用，一个编译环境，也可以是一套数据库平台服务，甚至是一个操作系统或集群。

基于 Linux 平台上的多项开源技术，Docker 提供了高效、敏捷和轻量级的容器方案，并支持部署到本地环境和多种主流云平台。可以说 Docker 首次为应用的开发、运行和部署提供了“一站式”的实用解决方案。

2. Linux 容器技术——巨人的肩膀

跟大部分新兴技术的诞生一样，Docker 也并非“从石头缝里蹦出来的”，而是站在前人的肩膀上。其中最重要的就是 Linux 容器（Linux Containers，LXC）技术。

IBM DeveloperWorks 网站关于容器技术的描述十分准确。

容器有效地将由单个操作系统管理的资源划分到孤立的组中，以更好地在孤立的组之间平衡有冲突的资源使用需求。与虚拟化相比，这样既不需要指令级模拟，也不需要即时编译。容器可以在核心 CPU 本地运行指令，而不需要任何专门的解释机制。此外，也避免了准虚拟化（para-virtualization）和系统调用替换中的复杂性。

当然，LXC 也经历了长期的演化。最早的容器技术可以追溯到 1982 年 Unix 系列操作系统上的 chroot 工具（直到今天，主流的 Unix、Linux 操作系统仍然支持和带有该工具）。早期的容器实现技术包括 Sun Solaris 操作系统上的 Solaris Containers（2004 年发布），FreeBSD 操作系统上的 FreeBSD jail（2000 年左右出现），以及 GNU/Linux 上的 Linux-VServer 和 OpenVZ。

在 LXC 之前，这些相关技术经过多年的演化已经十分成熟和稳定，但是由于种种原因，它们并没有被很好地集成到主流的 Linux 内核中，用户使用起来并不方便。例如，如果用户要使用 OpenVZ 技术，需要先手动给操作系统打上特定的内核补丁方可使用，而且不同版本并不一致。类似的困难，造成在很长一段时间内，这些优秀的技术只流传于技术人员的小圈子中。

后来 LXC 项目借鉴了前人成熟的容器设计理念，并基于一系列新引入的内核特性，实现了更具扩展性的虚拟化容器方案。更加关键地是，LXC 终于被集成得到了主流 Linux 内核中，进而成为了 Linux 系统轻量级容器技术的事实标准。

从技术层面来看，LXC 已经趟过了绝大部分的“坑”，完成了容器技术实用化的大半历程。

3. 从 Linux 容器到 Docker

在 LXC 的基础上，Docker 进一步优化了容器的使用体验，让它进入寻常百姓家。

首先，Docker 提供了各种容器管理工具（如分发、版本、移植等）让用户无需关注底层的操作，可以更简单明了地管理和使用容器；其次，Docker 通过引入分层文件系统构建和高效的镜像机制，降低了迁移难度，极大提升了用户体验。用户操作 Docker 容器就像操作应用自身一样简单。

实现上，早期的 Docker 代码实现是直接基于 LXC 的。自 0.9 版本开始，Docker 开发了 libcontainer 项目，作为更广泛的容器驱动实现，从而替换掉了 LXC 的实现。目前，Docker 还积极推动成立了 runC 标准项目，试图让容器的支持不再局限于 Linux 操作系统，而是更

安全、更具扩展性。

简单地讲，读者可以将 Docker 容器理解为一种轻量级的沙盒（Sandbox）。每个容器内运行着一个应用，不同的容器相互隔离，容器之间也可以通过网络互相通信。容器的创建和停止都十分快速，几乎跟创建和终止原生应用一致；另外，容器自身对系统资源的额外需求也十分有限，远远低于传统虚拟机。很多时候，甚至直接把容器当作应用本身也没有任何问题。

笔者相信，随着 Docker 技术的进一步成熟，它将成为更受欢迎的容器虚拟化技术实现，并在云计算和 Devops 等领域得到更广泛的应用。

1.2 为什么要使用 Docker

1. Docker 容器虚拟化的好处

Docker 项目的发起人，同时也是 Docker 公司 CTO Solomon Hykes 曾认为，Docker 在正确的地点、正确的时间顺应了正确的趋势——如何正确地构建应用。

在云时代，开发者创建的应用必须要能很方便地在网络上传播，也就是说应用必须脱离底层物理硬件的限制；同时必须是“任何时间任何地点”可获取的。因此，开发者们需要一种新型的创建分布式应用程序的方式，快速分发和部署，这正是 Docker 所能够提供的最大优势。

举个简单的例子，假设用户试图基于最常见的 LAMP（Linux+Apache+MySQL+PHP）组合来构建一个网站。按照传统的做法，首先，需要安装 Apache、MySQL 和 PHP 以及它们各自运行所依赖的环境；之后分别对它们进行配置（包括创建合适的用户、配置参数等）；经过大量的操作后，还需要进行功能测试，看是否工作正常；如果不正常，则进行调试追踪，意味着更多的时间代价和不可控的风险。可以想象，如果应用数目变多，事情会变得更加难以处理。

更为可怕的是，一旦需要服务器迁移（例如从亚马逊云迁移到其他云），往往需要对每个应用都进行重新部署和调试。这些琐碎而无趣的“体力活”，极大的降低了工作效率。究其根源，是这些应用直接运行在底层操作系统上，无法保证同一份应用在不同的环境中行为一致。

而 Docker 提供了一种更为聪明的方式，通过容器来打包应用，解耦应用和运行平台。意味着迁移的时候，只需要在新的服务器上启动需要的容器就可以了，无论新旧服务器是否是同一类型的平台。这无疑将节约大量的宝贵时间，并降低部署过程出现问题的风险。

2. Docker 在开发和运维中的优势

对开发和运维（DevOps）人员来说，可能最梦寐以求的效果就是一次创建或配置，之后可以在任意地方、任意时间让应用正常运行。而 Docker 恰恰是可以实现这一终极目标的“瑞士军刀”。

具体说来，Docker 在开发和运维过程中，具有如下几个方面的优势。

- 更快速的交付和部署。使用 Docker，开发人员可以使用镜像来快速构建一套标准的开发环境；开发完成之后，测试和运维人员可以直接使用完全相同环境来部署代码。只要开发测试过的代码，就可以确保在生产环境无缝运行。Docker 可以快速创建和删除容器，实现快速迭代，大量节约开发、测试、部署的时间。并且，整个过程全程可见，使团队更容易理解应用的创建和工作过程。

- 更高效的资源利用。Docker 容器的运行不需要额外的虚拟化管理程序（Virtual Machine Manager，VMM，以及 Hypervisor）支持，它是内核级的虚拟化，可以实

现更高的性能，同时对资源的额外需求很低。跟传统虚拟机方式相比，要提高一到两个数量级。

- 更轻松的迁移和扩展。Docker 容器几乎可以在任意的平台上运行，包括物理机、虚拟机、公有云、私有云、个人电脑、服务器等，同时支持主流的操作系统发行版本。这种兼容性让用户可以在不同平台之间轻松地迁移应用。

- 更简单的更新管理。使用 Dockerfile，只需要小小的配置修改，就可以替代以往大量的更新工作。并且所有修改都以增量的方式被分发和更新，从而实现自动化并且高效的容器管理。

3.Docker 与虚拟机比较

作为一种轻量级的虚拟化方式，Docker 在运行应用上跟传统的虚拟机方式相比具有显著优势：

- Docker 容器很快，启动和停止可以在秒级实现，这相比传统的虚拟机方式（数分钟）要快得多。

- Docker 容器对系统资源需求很少，一台主机上可以同时运行数千个 Docker 容器(在 IBM Power 服务器上已经实现了同时运行 10KB 量级的容器实例）。

- Docker 通过类似 Git 设计理念的操作来方便用户获取、分发和更新应用镜像，存储复用，增量更新。

- Docker 通过 Dockerfile 支持灵活的自动化创建和部署机制，提高工作效率，使流程标准化。

Docker 容器除了运行其中应用外，基本不消耗额外的系统资源，保证应用性能的同时，尽量减小系统开销。传统虚拟机方式运行 N 个不同的应用就要起 N 个虚拟机（每个虚拟机需要单独分配独占的内存、磁盘等资源），而 Docker 只需要启动 N 个隔离的“很薄的”容器，并将应用放进容器内即可。应用获得的是接近原生的运行性能。

当然，在隔离性方面，传统的虚拟机方式提供的是相对封闭的隔离。但这并不意味着 Docker 就不安全。Docker 利用 Linux 系统上的多种防护技术实现了严格的隔离可靠性，并且可以整合众多安全工具。从 1.3.0 版本开始，Docker 重点改善了容器的安全控制和镜像的安全机制，极大提高了使用 Docker 的安全性。在已知的大规模应用中，目前尚未出现值得担忧的安全隐患。

表 1-1 总结了使用 Docker 容器技术与传统虚拟机技术的特性比较，可见容器技术在很多应用场景下都具有巨大的优势。

表 1-1 Docker 容器技术与传统虚拟机技术的特性比较

特性	容器	虚拟机
启动速度	秒级	分钟级
性能	接近原生	较弱
内存代价	很小	较多
硬盘使用	一般为 MB	一般为 GB
运行密度	单机支持上千个容器	一般几十个
隔离性	安全隔离	完全隔离
迁移性	优秀	一般

1.3 Docker 与虚拟化

虚拟化（Virtualization）技术是一个通用的概念，在不同领域有不同的理解。在计算领域，一般指的是计算虚拟化（Computing Virtualization），或通常说的服务器虚拟化。按照维基百科上的定义：“虚拟化是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来，打破实体结构间的不可切割的障碍，使用户可以比原本的组态更好的方式来应用这些资源。”

可见，虚拟化的核心是对资源的抽象，目标往往是为了在同一个主机上同时运行多个系统或应用，从而提高系统资源的利用率，并且带来降低成本、方便管理和容错容灾等好处。

从大类上分，虚拟化技术可分为基于硬件的虚拟化和基于软件的虚拟化。其中，真正意义上的基于硬件的虚拟化技术不多见，少数如网卡中的单根多 IO 虚拟化（Single Root I/O Virtualization and Sharing Specification，SR-IOV）等技术，也超出了本书的讨论范畴。

基于软件的虚拟化从对象所在的层次，又可以分为应用虚拟化和平台虚拟化（通常说的虚拟机技术即属于这个范畴）。其中，前者一般指的是一些模拟设备或诸如 Wine 这样的软件。后者又可以细分为如下几个子类：

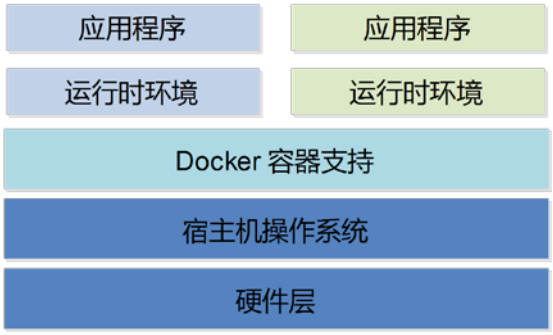
- 完全虚拟化。虚拟机模拟完整的底层硬件环境和特权指令的执行过程，客户操作系统无需进行修改。例如 IBM p 和 z 系列的虚拟化、VMware Workstation、VirtualBox、QEMU 等。
- 硬件辅助虚拟化。利用硬件（主要是 CPU）辅助支持（目前 x86 体系结构上可用的硬件辅助虚拟化技术包括 Intel-VT 和 AMD-V）处理敏感指令来实现完全虚拟化的功能，客户操作系统无需修改，例如 VMwareWorkstation、Xen、KVM。
- 部分虚拟化。只针对部分硬件资源进行虚拟化，客户操作系统需要进行修改。现在有些虚拟化技术的早期版本仅支持部分虚拟化。
- 超虚拟化（Paravirtualization）。部分硬件接口以软件的形式提供给客户机操作系统，客户操作系统需要进行修改，例如早期的 Xen。
- 操作系统级虚拟化。内核通过创建多个虚拟的操作系统实例（内核和库）来隔离不同的进程。容器相关技术即在这个范畴。

可见，Docker 以及其他容器技术，都属于操作系统虚拟化这个范畴，操作系统虚拟化最大的特点就是不需要额外的 supervisor 支持。

Docker 虚拟化方式之所有有众多优势，这跟操作系统虚拟化技术自身的设计和实现是分不开的。

图 1-2 比较了 Docker 和常见的虚拟机方式的不同之处。

a)传统的虚拟化



b)Docker 虚拟化

图 1-2 Docker 和传统的虚拟机方式的不同之处

传统方式是在硬件层面实现虚拟化，需要有额外的虚拟机管理应用和虚拟机操作系统层。

Docker 容器是在操作系统层面上实现虚拟化，直接复用本地主机的操作系统，因此更加轻量级。

1.4 本章小结

本章介绍了容器虚拟化的基本概念、Docker 的诞生历史，以及容器在云时代应用分发场景下的巨大优势。

相比对传统的虚拟机方式，容器虚拟化方式在很多场景下都存在极为明显的优势。无论是系统管理员、应用开发人员、测试人员以及运维管理人员，都应该尽快掌握 Docker，尽早享受其带来的巨大便利。

后续章节，笔者将结合实践案例具体介绍 Docker 的安装、使用，让我们一起开启精彩的 Docker 之旅。

第2章 核心概念与安装配置

本章首先介绍 Docker 的三大核心概念：

- 镜像（Image）
- 容器（Container）
- 仓库（Repository）

读者理解了这三个核心概念，才能顺利地理解 Docker 容器的整个生命周期。

随后，笔者将介绍如何在常见的操作系统平台上安装 Docker，包括 Ubuntu、CentOS、MacOS 和 Windows 等主流操作系统。

2.1 核心概念

Docker 大部分的操作都围绕着它的三大核心概念：镜像、容器和仓库。因此，准确把握这三大核心概念对于掌握 Docker 技术尤为重要。

1. Docker 镜像

Docker 镜像类似于虚拟机镜像，可以将它理解为一个只读的模板。例如，一个镜像可以包含一个基本的操作系统环境，里面仅安装了 Apache 应用程序（或用户需要的其他软件）。可以把它称为一个 Apache 镜像。

镜像是创建 Docker 容器的基础。通过版本管理和增量的文件系统，Docker 提供了一套十分简单的机制来创建和更新现有的镜像，用户甚至可以从网上下载一个已经做好的应用镜像，并直接使用。

2. Docker 容器

Docker 容器类似于一个轻量级的沙箱，Docker 利用容器来运行和隔离应用。容器是从镜像创建的应用运行实例。可以将其启动、开始、停止、删除，而这些容器都是彼此相互隔离的、互不可见的。

可以把容器看做是一个简易版的 Linux 系统环境（这包括 root 用户权限、进程空间、用户空间和网络空间等）以及运行在其中的应用程序打包而成的盒子。

> 注意：镜像自身是只读的。容器从镜像启动的时候，会在镜像的最上层创建一个可写层。

3. Docker 仓库

Docker 仓库，类似于代码仓库，它是 Docker 集中存放镜像文件的场所。

有时候会看到有资料将 Docker 仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器是存放仓库的地方，其上往往存放着多个仓库。每个仓库集中存放某一类镜像，往往包括多个镜像文件，通过不同的标签（tag）来进行区分。例如存放 Ubuntu 操作系统镜像的仓库，被称为 Ubuntu 仓库，其中可能包括 14.04、12.04 等不同版本的镜像。仓库注册服务器的示例如图 2-1 所示。

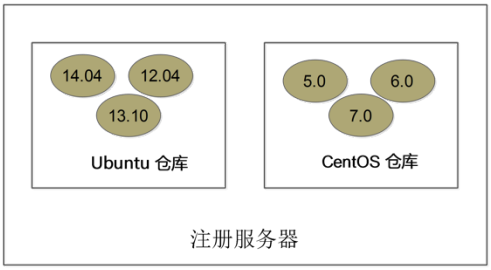


图 2-1 注册服务器与仓库

根据所存储的镜像公开分享与否，Docker 仓库可以分为公开仓库（Public）和私有仓库（Private）两种形式。目前，最大的公开仓库是官方提供的 Docker Hub，存放了数量庞大的镜像供用户下载。国内不少云服务提供商（如时速云、阿里云等）也提供了仓库的本地源等，可以提供稳定的国内访问。

当然，用户如果不希望公开分享自己的镜像文件，Docker 也支持用户在本地网络内创建一个只能自己访问的私有仓库。当用户创建了自己的镜像之后就可以使用 `push` 命令将它上传到指定的公有或者私有仓库。这样用户下次在另外一台机器上使用该镜像时，只需要将其从仓库上 `pull` 下来就可以了。

提示：可以看出，Docker 利用仓库管理镜像的设计理念与 Git 非常相似，实际上在理念设计上借鉴了 Git 的很多优秀思想。

2.2 安装 Docker

Docker 在主流的操作系统和云平台上都可以使用，包括 Linux 操作系统（如 Ubuntu、Debian、CentOS、Redhat 等 ，MacOS 和 Windows 操作系统，以及 AWS 等云平台。

如图 2-2 所示，用户可以访问 Docker 官网的 Get Docker（https://www.docker.com/products/overview）页面，查看获取 Docker 的方式，以及 Docker 支持的平台类型。

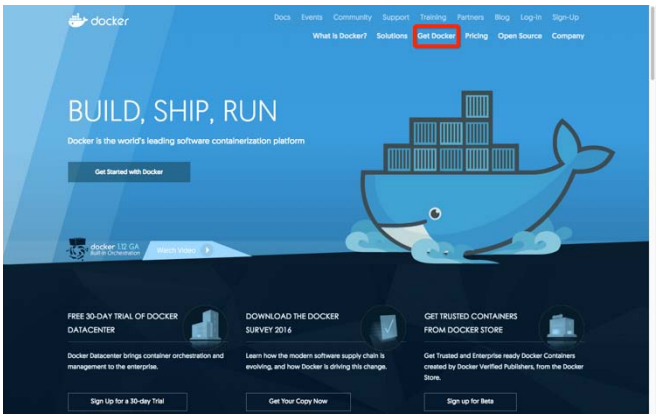


图 2-2 Docker 支持多种平台

在 Get Docker 页面，我们可以看到目前 Docker 支持 Docker Platform、Docker Hub、Docker Cloud、Docker DataCenter。

- Docker Platform：支持在桌面系统或云平台安装 Docker；
- DockerHub：官方提供的云托管服务，可以提供公有或私有的镜像仓库；

- DockerCloud：官方提供的容器云服务，可以完成容器的部署与管理，可以完整地支持容器化项目，还有 CI、CD 功能；
- Docker DataCenter：提供企业级的简单安全弹性的容器集群编排和管理。

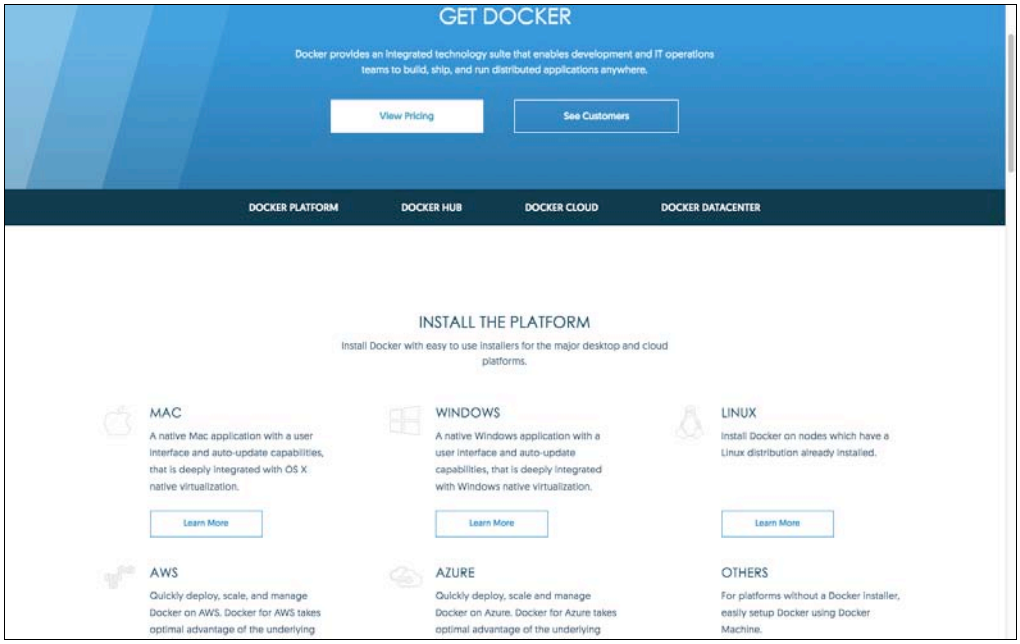


图 2-3 获取 Docker

笔者推荐尽量使用 Linux 操作系统来运行 Docker。这是因为目前 Linux 操作系统对 Docker 的支持是原生的，使用体验也最好。

2.2.1 Ubuntu 环境下安装 Docker

1. 系统要求

Docker 目前只能运行在 64 位平台上，并且要求内核版本不低于 3.10，实际上内核越新越好，过低的内核版本容易造成功能的不稳定。

用户可以通过如下命令检查自己的内核版本详细信：

```
$ uname -a
Linux Host 3.16.0-43-generic #58~14.04.1-Ubuntu SMP Mon Jun 22 10:21:20 UTC 2015 x86_64 x86_64
x86_64 GNU/Linux
```

或者：

```
$ cat /proc/version
Linux version 3.16.0-43-generic (buildd@brownie) (gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1) )
#58~14.04.1-Ubuntu SMP Mon Jun 22 10:21:20 UTC 2015
```

Docker 目前支持的最低 Ubuntu 版本为 12.04 LTS，但实际上从稳定性上考虑，推荐至少使用 14.04 LTS 版本。

如果使用 12.04 LTS 版本，首先要更新系统内核和安装可能需要的软件包，包括：

- linux-image-generic-lts-trusty（必备）
- linux-headers-generic-lts-trusty（必备）

- xserver-xorg-lts-trusty（带图形界面时必备）
- libgl1-mesa-glx-lts-trusty（带图形界面时必备）

另外，为了让 Docker 使用 aufs 存储，推荐安装 `linux-image-extra` 软件包。

```
$ sudo apt-get install -y linux-image-extra-$(uname -r)
```

注意：Ubuntu 发行版中，LTS（Long-Term-Support）意味着更稳定的功能和更长期（目前为 5 年）的升级支持，生产环境中尽量使用 LTS 版本。

2. 添加镜像源

首先需要安装 apt-transport-https 包支持 https 协议的源：

```
$ sudo apt-get install -y apt-transport-https
```

添加源的 gpg 密钥：

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
58118E89F3A912897C070ADBF76221572C52609D
```

获取当前操作系统的代号：

```
$ lsb_release -c
Codename:       trusty
```

一般情况下，12.04(LTS)代号为 precise，14.04(LTS)代号为 trusty，15.04 代号为 vivid，15.10 代号为 wily。这里获取到代号为 trusty。

接下来就可以添加 Docker 的官方 apt 软件源了。通过下面命令创建 `/etc/apt/sources.list.d/docker.list` 文件，并写入源的地址内容。非 trusty 版本的系统注意修改为自己对应的代号：

```
$ sudo cat <<EOF > /etc/apt/sources.list.d/docker.list
deb https://apt.dockerproject.org/repo ubuntu-trusty main
EOF
```

添加成功后，更新 apt 软件包缓存：

```
$ sudo apt-get update
```

3. 开始安装 Docker

在成功添加源之后，就可以安装最新版本的 Docker 了，软件包名称为 docker-engine。

```
$ sudo apt-get install -y docker-engine
```

如果系统中存在较旧版本的 Docker（lxc-docker），会提示是否先删除，选择是即可。

除了基于手动添加软件源的方式，另外，也可以使用官方提供的脚本来自动化安装 Docker：

```
$ sudo curl -sSL https://get.docker.com/ | sh
```

安装成功后，启动 docker 服务：

```
$ sudo service docker start
```

2.2.2 CentOS 环境下安装 Docker

系统的要求跟 Ubuntu 情况类似：64 位操作系统，内核版本至少为 3.10。

Docker 目前支持 CentOS6.5 及以后的版本，推荐使用 CentOS 7 系统。

首先，也是要添加 yum 软件源：

```
$ sudo tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

之后更新 yum 软件源缓存，并安装 docker-engine：

```
$ sudo yum update
$ sudo yum install -y docker-engine
```

对于 CentOS 7 系统，`CentOS-Extras` 源中已内置 Docker，如果已经配置了 `CentOS-Extras` 源，可以直接通过上面的 yum 命令进行安装。

2.2.3 通过脚本安装

用户还可以使用官方提供的 shell 脚本来在 Linux 系统（目前支持 Ubuntu、Debian、Oracleserver、Fedora、Centos、OpenSuse、Gentoo 等常见发行版）上安装 Docker 的最新正式版本，该脚本会自动检测系统信息并进行相应配置：

```
$ curl -fsSL https://get.docker.com/ | sh
```

或者：

```
$ wget -qO- https://get.docker.com/ | sh
```

如果想尝鲜，使用最新功能，可以使用下面的脚本来安装预发布版本。但要注意，预发布版本往往意味着功能还不够稳定，不要在生产环境中使用：

```
$ curl -fsSL https://test.docker.com/ | sh
```

另外，也可以从 github.com/docker/docker/releases 找到所有的发行版本信息和二进制包，自行下载使用。

2.2.4 Mac OS 环境下安装 Docker

Docker 官方非常重视其在 Mac 环境下的易用性。目前 Docker 支持原生 Mac 客户端，内置图形界面，支持自动升级。此客户端与 Mac OS X 的原生虚拟化深度结合，摒弃了之前安装 VirtualBox（即 Docker Toolbox）的简单粗暴的做法。我们先从官方默认的 Docker for Mac 开始。

1. Docker for Mac

第一步，下载安装包。访问 https://docs.docker.com/docker-for-mac/下载页面。目前 Docker for Mac 分为稳定版和 Beta 版两种更新通道，我们可以按需选择。下载完成后，双击安装包:

图 2-4 下载后打开安装包

第二部，开始安装。将 Docker.app 拖拽至 Applications 文件夹，即可完成安装：

图 2-5 安装 Docker 到 Applications 文件夹

第三步：运行 Docker for Mac。在欢迎窗口点击 Next：

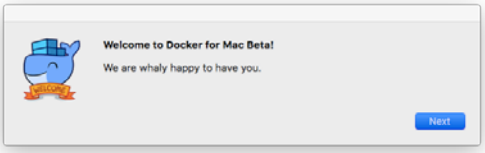


图 2-6 欢迎窗口

允许 Docker 获得系统权限，它需要将 Mac 网卡链接至 Docker.app。点击 OK 后输入系统管理员密码。

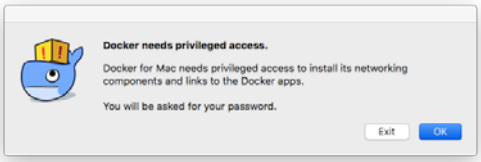


图 2-7 运行 Docker for Mac

此时系统状态栏会出现 Docker 的 Icon 图标，点击后如果出现“Docker is running!”，则说明安装成功：

第四步：验证 Docker 安装。打开终端控制器或其他系统命令行，执行 `docker version` 命令。

```
$ docker version
Client:
 Version:      1.12.0
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   8eab29e
 Built:        Thu Jul 28 21:15:28 2016
 OS/Arch:      darwin/amd64

Server:
 Version:      1.12.0
 API version:  1.24
 Go version:   go1.6.3
```

```
Git commit:   8eab29e
Built:        Thu Jul 28 21:15:28 2016
OS/Arch:      linux/amd64
```

如果我们看到 Client 和 Server 均有输出，则说明 Docker for Mac 已经正常启动。如果我们看到报错：Cannot connect to the Docker daemon. Is the docker daemon running on this host?，则说明 Docker for Mac 没有启动或启动失败。

下面启动一个 Ninx 容器，检查能正确获取镜像并运行：

```
$ docker run -d -p 80:80 --name webserver nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
51f5c6a04d83: Pull complete
a3ed95caeb02: Pull complete
51d229e136d0: Pull complete
bcd41daec8cc: Pull complete
Digest:
sha256:0fe6413f3e30fcc5920bc8fa769280975b10b1c26721de956e1428b9e2f29d04
Status: Downloaded newer image for nginx:latest
34bcd01998a76f67b1b9e6abe5b7db5e685af325d6fafb1acd0ce84e81e71e5d
```

然后使用 `docker ps` 指令查看当前运行的 `docker ps` 指令查看当前运行的容器：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS                           NAMES
34bcd01998a7        nginx               "nginx -g 'daemon off"   2 minutes ago       Up 2 minutes
0.0.0.0:80->80/tcp, 443/tcp   webserver
```

可见 `Nginx` 容器已经在 `0.0.0.0:80` 启动，并映射了 80 端口，下面我们打开浏览器访问此地址：

图 2-8 允许访问系统权限

第五步：常用配置。首先，点击系统状态栏的 Docker 图标，会出现操作菜单：

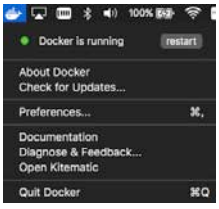


图 2-9 访问 Nginx 服务映射

然后，点击 Preferences，进入标准配置页面，我们可以设置是否自动启动与更新，设置备份工具 Time Machine 是否备份 VM，还可以配置 Docker 使用的 CPU 数、内存容量：

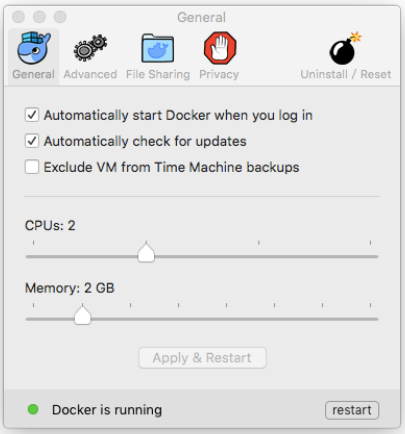


图 2-10 状态栏 Docker 图标

点击进入 Advanced 进阶配置。鉴于国内的网络现状，为了更好地使用 Docker Hub，我们可以使用 Registry 镜像站点进行加速。点击+后，加入镜像站点配置。这里还可以配置 HTTP 代理服务器：

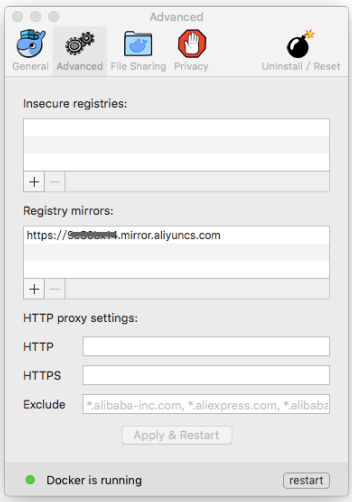


图 2-11 标准配置页面

点击进入 File Sharing 标签页，此处可以配置挂载至容器中的本地目录。点击+后可以继续添加本地目录：

图 2-12 高级配置页面

点击进入 Privacy 标签页，此处可以配置隐私选项，如是否发送使用信息，以及是否发送程序崩溃报告：

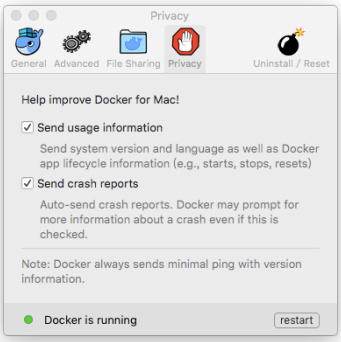


图 2-13 文件分享配置页面

2. Docker Toolbox

在 Mac OS X 操作系统上安装 Docker，除了 Docker for Mac 的原生方式之外，还可以使用官方提供的 Docker ToolBox 工具。

首先前往 https://www.docker.com/products/docker-toolbox 下载对应版本的 ToolBox。目前 Docker 支持的 Mac OS X 版本为 10.6+。

图 2-14 隐私配置页面

双击运行安装包。这个过程将安装一个 VirtualBox 虚拟机，内置了 Docker Engine、Compose、Machine、Kitematic 等管理工具，如图 2-15 所示。

图 2-15 Toolbox 安装页面

安装成功后，找到 Boot2Docker 并运行它。现在进行 Boot2Docker 的初始化：

```
$ boot2docker init

$ boot2docker start

$ $(boot2docker shellinit)
```

将看到虚拟机在命令行窗口中启动运行。当虚拟机初始化完毕后，可以使用 `boot2docker stop` 和 `boot2docker start` 来控制它。

注意，如果在命令行中看到如下提示信息：

```
To connect the Docker client to the Docker daemon, please set: export

DOCKER_HOST=tcp://192.168.59.103:2375
```

可以执行提示信息中的语句：`export DOCKER_HOST=tcp://192.168.59.103:2375`。此语句的作用是在系统环境变量中设置 Docker 的主机地址。

2.2.5 Windows 环境下安装 Docker

目前 Docker 可以通过虚拟机方式来支持 Windows 7.1 和 8，只要平台 CPU 支持硬件虚拟化特性即可。读者如果无法确定自己计算机的 CPU 是否支持该特性也无需担心，实际上，目前市面上主流的 CPU 都早已支持了硬件虚拟化特性。

对于Windows10 用户，Docker官方提供了原生虚拟化应用Docker for Windows。详情见：https://docs.docker.com/windows/step_one/。目前国内Windows 7 还是主导地位的版本，所以下面主要讲解如何在Windows 7 环境下安装Docker环境。

由于 Docker 引擎使用了 Linux 内核特性，所以要在 Windows 10 之外的 Windows 上运行的话，需要额外使用一个虚拟机来提供 Linux 支持。这里推荐使用 Boot2Docker 工具,它会首先安装一个经过加工与配置的轻量级虚拟机，然后在其中运行 Docker。主要步骤如下：

首先，从https://docs.docker.com/installation/windows/下载最新官方Docker for Windows Installer。

双击打开 Installer。这个过程将安装 VirtualBox、MSYS-git、boot2docker Linux ISO 镜像，以及 Boot2Docker 管理工具。如下图所示。

图 2-16 Toolbox 安装器

最后, 打开桌面的 Boot2Docker Start 程序, 或者 Program Files\ Boot2Docker for Windows。此初始化脚本在第一次运行时需要输入一个 SSH Key Passphrase(用于 SSH 密钥生成的口令)。读者可以自行设定，也可以直接按回车键，跳过此设定。如下图所示。

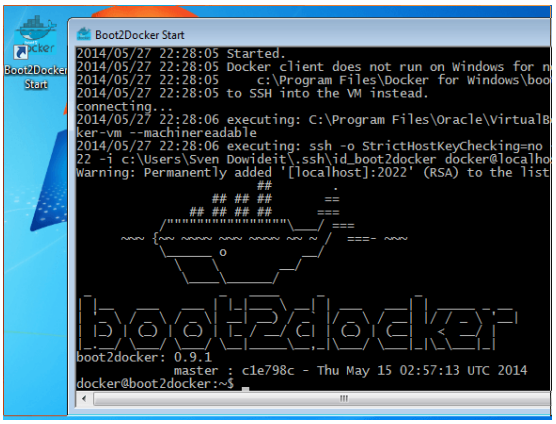


图 2-17 Boot2Docker 安装器

此时 Boot2Docker Start 程序将连接至虚拟机中的 Shell 会话，Docker 已经运行起来了！

2.3 配置 Docker 服务

为了避免每次使用 docker 命令要用特权身份，可以将当前用户加入到安装中自动创建的 `docker` 用户组：

```
$ sudo usermod -aG docker USER_NAME
```

用户更新组信息后，退出并重新登录后即可生效。

另外，Docker 服务支持多种启动参数。以 Ubuntu 系统为例，Docker 服务的默认配置文件为`/etc/default/docker`，可以通过修改其中的 `DOCKER_OPTS` 来修改服务启动的参数，例如如下一行让 Docker 服务可以通过本地 2375 端口接收到来自外部的请求：

```
DOCKER_OPTS="$DOCKER_OPTS -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock"
```

修改之后，通过 service 命令来重启 Docker 服务。

```
$ sudo service docker restart
```

一般情况下，Docker 服务的管理脚本为`/etc/init.d/docker`，通过查看其中内容，发现主要是将 Docker 进程的 id 写入`/var/run/docker.pid` 文件，以及通过 ulimit 调整系统的资源限制。

如果是通过较新的 upstart 工具来管理服务，则管理服务配置文件在

`/etc/init/docker.conf`。

另外，对于 CentOS、Redhat 等系统，服务可能是通过 systemd 来管理，略有不同，可以查阅 systemd 相关手册。

例如，需要通过 systemctl 命令来管理 Docker 服务：

```
$ sudo systemctl start docker.service
```

此外，如果服务工作不正常，可以通过查看 Docker 服务的日志信息来确定问题，例如在 Ubuntu 系统上日志文件可能为`/var/log/upstart/docker.log`：

```
$ sudo tail /var/log/upstart/docker.log
```

每次重启 Docker 服务后，可以通过查看 Docker 版本信息，确保服务已经正常运行：

```
$ docker version
Client:
 Version:      1.12.0
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   8eab29e
 Built:        Thu Jul 28 21:15:28 2016
 OS/Arch:      darwin/amd64

Server:
 Version:      1.12.0
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   8eab29e
 Built:        Thu Jul 28 21:15:28 2016
 OS/Arch:      linux/amd64
```

2.4 推荐实践环境

从稳定性上考虑，本书推荐的实践环境的操作系统是 Ubuntu 14.04.3 LTS 系统，带有 linux-image-3.16.0-71-generic 内核。Docker 版本为最新的 1.12 稳定版本。不同版本的 API 会略有差异，推荐根据需求选择较新的稳定版本。另外，如无特殊说明，默认数据网段地址范围为 `10.0.0.0/24`，管理网段地址范围为 `192.168.0.0/24`。

执行命令代码中以`$`开头的，表明为普通用户；以`#`开头的，表明为特权用户（root）。如果用户已经添加到了 `docker` 用户组（参考上一小节），大部分时候都无需管理员权限，否则需要在命令前使用 `sudo` 来临时提升权限。

部分命令执行结果输出内容较长的，只给出了关键部分输出。读者可根据自己的实际情况，搭建类似的环境。

2.5 本章小结

本章介绍了 Docker 的三大核心概念：镜像、容器和仓库。

在后面的实践中，读者会感受到，基于三大核心概念所构建的高效工作流程，正是 Docker 从众多容器虚拟化方案中脱颖而出的重要原因。实际上，Docker 的工作流也并非凭空创造的，很大程度上参考了 Git 和 Github 的设计理念，从而为应用分发和团队合作都带

来了众多优势。

在后续章节，笔者将具体讲解围绕这三大核心概念的 Docker 操作命令。

第3章 使用 Docker 镜像

镜像（image）是 Docker 三大核心概念中最为重要的，自 Docker 诞生之日起 镜像 就是相关社区最为热门的关键词。

Docker 运行容器前需要本地存在对应的镜像，如果镜像不存在本地，Docker 会尝试先从默认镜像仓库下载（默认使用 Docker Hub 公共注册服务器中的仓库），用户也可以通过配置，使用自定义的镜像仓库。

本章将介绍围绕镜像这一核心概念的具体操作，包括如何使用 pull 命令从 Docker Hub 仓库中下载镜像到本地；如何查看本地已有的镜像信息和管理镜像标签；如何在远端仓库使用 search 命令进行搜索和过滤；如何删除镜像标签和镜像文件；如何创建用户定制的镜像并且保存为外部文件。最后，还将介绍往 Docker Hub 仓库中推送自己的镜像。

> 提示：一份非官方研究报告表明：image 一直是 Docker 官方社区（2014～2016）和 StackOverFlow Docker 板块（2013～2016）的年度热词。

3.1 获取镜像

镜像是运行容器的前提，官方的 Docker Hub 网站已经提供了数十万个镜像供大家开放下载。

可以使用 `docker pull` 命令直接从 Docker Hub 镜像源来下载镜像。该命令的格式为 docker pull NAME[:TAG]。其中，`NAME` 是镜像仓库名称（用来区分镜像），`TAG` 是镜像的标签（往往用来表示版本信息）。通常情况下，描述一个镜像需要包括“名称+标签”信息。

例如，获取一个 Ubuntu 14.04 系统的基础镜像可以使用如下的命令：

```
$ docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu
6c953ac5d795: Pull complete
3eed5ff20a90: Pull complete
f8419ea7c1b5: Pull complete
51900bc9e720: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:97421885f3da3b23f52eeddcaa9f8f91172a8ac3cd5d3cd40b51c7aad09f66cc
Status: Downloaded newer image for ubuntu:14.04
```

对于 Docker 镜像来说，如果不显式指定 TAG，则默认会选择 `latest` 标签，这会下载仓库中最新版本的镜像。

下面的例子将从 Docker Hub 的 Ubuntu 仓库下载一个最新的 Ubuntu 操作系统的镜像。

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
ac6ad7efd0f9: Pull complete
e7491a747824: Pull complete
```

```
a3ed95caeb02: Pull complete
Digest: sha256:46fb5d001b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
```

该命令实际上下载的就是 `ubuntu:latest` 镜像。

注意，一般来说，镜像的 `latest` 标签意味着该镜像的内容会跟踪最新的非稳定版本的发布，内容是不稳定的。当前 Ubuntu 最新的发行版本为 `16.04`，`latest` 镜像实际上就是 `16.04` 镜像，用户可以下载 `ubuntu:16.04` 镜像并查看两者的数字摘要值是一致的。从稳定性上考虑，不要在生产环境中忽略镜像的标签信息或使用默认的 `latest` 标记的镜像。

下载过程中可以看出，镜像文件一般由若干层（layer）组成，`6c953ac5d795` 这样的串是层的唯一 id（实际上完整的 id 包括 256 比特，64 个十六进制字符组成）。使用 `docker pull` 命令下载中会获取并输出镜像的各层信息。当不同的镜像包括相同的层时，本地仅存储了层的一份内容，减小了存储空间。

读者可能会想到，在不同的镜像仓库服务器的情况下，可能会出现镜像重名的情况。

严格地讲，镜像的仓库名称中还应该添加仓库地址（即 registry，注册服务器）作为前缀，只是默认使用的是 Docker Hub 服务，该前缀可以忽略。

例如，`docker pull ubuntu:14.04` 命令相当于 `docker pull registry.hub.docker.com/ubuntu:14.04` 命令，即从默认的注册服务器 Docker Hub Registry 中的 ubuntu 仓库来下载标记为 14.04 的镜像。

如果从非官方的仓库下载，则需要在仓库名称前指定完整的仓库地址。例如从网易蜂巢的镜像源来下载 ubuntu:14.04 镜像，可以使用如下命令，此时下载的镜像名称为 `hub.c.163.com/public/ubuntu:14.04`。

```
$ docker pull hub.c.163.com/public/ubuntu:14.04
```

`pull` 子命令支持的选项主要包括：

-a, --all-tags=true|false：是否获取仓库中的所有镜像，默认为否。

下载镜像到本地后，即可随时使用该镜像了，例如利用该镜像创建一个容器，在其中运行 bash 应用，执行 ping localhost 命令。

```
$ docker run -it ubuntu:14.04 bash
root@9c74026df12a:/# ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.058 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.023 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.018 ms
^C
--- localhost ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.018/0.033/0.058/0.017 ms
root@9c74026df12a:/# exit
exit
```

3.2 查看镜像信息

1.使用 images 命令列出镜像

使用`docker images`命令可以列出本地主机上已有镜像的基本信息。

例如，下面的命令列出了上一小节中下载的镜像信息。

```
$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
ubuntu                        16.04               2fa927b5cdd3        2 weeks ago         122
MB
ubuntu                        latest              2fa927b5cdd3        2 weeks ago         122
MB
ubuntu                        14.04               8f1bd21bd25c        2 weeks ago         188
MB
```

在列出信息中，可以看到几个字段信息：

- 来自于哪个仓库，比如 ubuntu 仓库用来保存 ubuntu 系列的基础镜像；
- 镜像的标签信息，比如 14.04、latest 用来标注不同的版本信息。标签只是标记，并不能标识镜像内容；
- 镜像的 `ID`（唯一标识镜像），如 ubuntu:latest 和 ubuntu:16.04 镜像的都为 `2fa927b5cdd3`，说明它们目前实际上指向了同一个镜像；
- 创建时间，说明镜像最后的更新时间；
- 镜像大小，优秀的镜像往往体积都较小。

其中镜像的 ID 信息十分重要，它唯一标识了镜像。在使用镜像 ID 的时候，一般可以使用该 ID 的前若干个字符组成的可区分串来替代完整的 ID。

TAG 信息用来标记来自同一个仓库的不同镜像。例如 ubuntu 仓库中有多个镜像，通过 TAG 信息来区分发行版本，包括 10.04、12.04、12.10、13.04、14.04、16.04 等标签。

镜像大小信息只是表示了该镜像的逻辑体积大小，实际上由于相同的镜像层本地只会存储一份，物理上占用的存储空间会小于各镜像逻辑体积之和。

`images`子命令主要支持如下选项，用户可以自行进行尝试：

- `-a, --all=true|false`：列出所有（包括临时文件）镜像文件，默认为否；
- `--digests=true|false`：列出镜像的数字摘要值，默认为否；
- `-f, --filter=[]`：过滤列出的镜像，如 `dangling=true` 只显示没有被使用的镜像；也可指定带有特定标注的镜像等；
- `--format="TEMPLATE"`:控制输出格式，如 `.ID` 代表 ID 信息，`.Repository` 代表仓库信息等；
- `--no-trunc=true|false`：对输出结果中太长的部分是否进行截断，如镜像的 ID 信息，默认为是；

- `-q, --quiet=true|false`：仅输出 ID 信息，默认为否。

其中，对输出结果进行控制的选项如 `-f, --filter=[]`、`--no-trunc=true|false`、`-q, --quiet=true|false` 等大部分子命令都支持。

更多子命令选项还可以通过 `man docker-images` 来查看。

2. 使用 tag 命令添加镜像标签

为了方便在后续工作中使用特定镜像，还可以使用 `docker tag` 命令来为本地镜像任意添加新的标签。例如添加一个新的 `myubuntu:latest` 镜像标签：

```
$ docker tag ubuntu:latest myubuntu:latest
```

再次使用 `docker images` 列出本地主机上镜像信息，可以看到多了一个 `myubuntu:latest` 标签的镜像。

```
$ docker images
REPOSITORY       TAG       IMAGE ID          CREATED         SIZE
ubuntu           16.04     2fa927b5cdd3      2 weeks ago     122 MB
ubuntu           latest    2fa927b5cdd3      2 weeks ago     122 MB
myubuntu         latest    2fa927b5cdd3      2 weeks ago     122 MB
ubuntu           14.04     8f1bd21bd25c      2 weeks ago     188 MB
```

之后，用户就可以直接使用 `myubuntu:latest` 来表示这个镜像了。

细心的读者可能注意到，这些 `myubuntu:latest` 镜像的 ID 跟 `ubuntu:latest` 是完全一致的。它们实际上指向了同一个镜像文件，只是别名不同而已。`docker tag` 命令添加的标签实际上起到了类似链接的作用。

3. 使用 inspect 命令查看详细信息

使用 `docker inspect` 命令可以获取该镜像的详细信息，包括制作者、适应架构、各层的数字摘要等。

```
$ docker inspect ubuntu:14.04
[
    {
        "Id": "sha256:8f1bd21bd25c3fb1d4b00b7936a73a0664f932e11406c48a0ef19d82fd0b7342",
        "RepoTags": [
            "ubuntu:14.04"
        ],
        "RepoDigests": [],
        "Parent": "",
        "Comment": "",
        "Created": "2016-05-27T14:13:04.103044105Z",
        "Container": "eb8c67a3bff6e93658d18ac14b3a2134488c140a1ae1205c0cfdfd49f087113f",
        "ContainerConfig": {
            "Hostname": "fff5562e8198",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
```

```
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [],
    "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) CMD [\"/bin/bash\"]"
    ],
    "Image": "f9cdf71c33f14c7af4b75b651624e9ac69711630e21ceb289f69e0300e90c57d",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": {}
},
"DockerVersion": "1.9.1",
"Author": "",
"Config": {
    "Hostname": "fff5562e8198",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
    "Env": [],
    "Cmd": [
        "/bin/bash"
    ],
    "Image": "f9cdf71c33f14c7af4b75b651624e9ac69711630e21ceb289f69e0300e90c57d",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": null,
    "OnBuild": null,
    "Labels": {}
},
"Architecture": "amd64",
"Os": "linux",
"Size": 187957543,
"VirtualSize": 187957543,
```

```
        "GraphDriver": {
            "Name": "aufs",
            "Data": null
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:a7e1c363defb1f80633f3688e945754fc4c8f1543f07114befb5e0175d569f4c",
                "sha256:dc109d4b4ccf69361d29292fb15e52707507b520aba8cd43a564182f26725d74",
                "sha256:9f7ab087e6e6574225f863b6013579a76bd0c80c92fefe7aea92c4207b6486cb",
                "sha256:6f8be37bd578bbabe570b0181602971b0ea3509b79a1a3dd5528a4e3fc33dd6f",
                "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef"
            ]
        }
    }
]
```

返回的是一个 JSON 格式的消息，如果我们只要其中一项内容时，可以使用 `-f` 来指定，例如，获取镜像的 `Architecture`：

```
$ docker inspect -f {{".Architecture"}}
amd64
```

4. 使用 history 命令查看镜像历史

既然镜像文件由多个层组成，那么怎么知道各个层的内容具体是什么呢？这时候可以使用 `history` 子命令，该命令将列出各层的创建信息。

例如，查看 ubuntu:14.04 镜像的创建过程，可以使用如下命令。

```
$ docker history ubuntu:14.04
IMAGE       CREATED       CREATED BY                          SIZE COMMENT
8f1bd21bd25c  2 weeks ago   /bin/sh -c #(nop) CMD ["/bin/bash"]    0 B
<missing>   2 weeks ago   /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/  1.895 kB
<missing>   2 weeks ago   /bin/sh -c rm -rf /var/lib/apt/lists/*      0 B
<missing>   2 weeks ago   /bin/sh -c set -xe   && echo '#!/bin/sh' > /u  194.5 kB
<missing>   2 weeks ago   /bin/sh -c #(nop) ADD file:aca501360d0937bc49  187.8 MB
```

注意过长的命令被自动截断了，可以使用前面提到的`--no-trunc` 选项来输出完整命令。

3.3 搜寻镜像

使用 `docker search` 命令可以搜索远端仓库中共享的镜像，默认搜索官方仓库中的镜像。用法为 `docker search TERM`，支持的参数主要包括：

- `--automated=true|false`：仅显示自动创建的镜像，默认为否；
- `--no-trunc=true|false`：输出信息不截断显示，默认为否；
- `-s, --stars=X`：指定仅显示评价为指定星级以上的镜像，默认为 0，即输出所有镜像。

例如，搜索所有自动创建的评价为 1+ 的带 nginx 关键字的镜像，如下所示。

```
$ docker search --automated -s 3 nginx
NAME                      DESCRIPTION           STARS     OFFICIAL   AUTOMATED
jwilder/nginx-proxy       Automated Nginx reverse proxy for docker c...   670  [OK]
richarvey/nginx-php-fpm   Container running Nginx + PHP-FPM capable ...  206  [OK]
million12/nginx-php       Nginx + PHP-FPM 5.5, 5.6, 7.0 (NG), CentOS...   67  [OK]
maxexcloo/nginx-php       Docker framework container with Nginx and ...   57  [OK]
webdevops/php-nginx       Nginx with PHP-FPM                              38  [OK]
h3nrik/nginx-ldap         NGINX web server with LDAP/AD, SSL and pro...   27  [OK]
bitnami/nginx             Bitnami nginx Docker Image                      18  [OK]
maxexcloo/nginx           Docker framework container with Nginx inst...   7   [OK]
million12/nginx           Nginx: extensible, nicely tuned for better...   4   [OK]
webdevops/nginx           Nginx container                                 3   [OK]
ixbox/nginx               Nginx on Alpine Linux.                          3   [OK]
evild/alpine-nginx        Minimalistic Docker image with Nginx            3   [OK]
```

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、星级（表示该镜像的受欢迎程度）、是否官方创建、是否自动创建等。

默认的输出结果将按照星级评价进行排序。

3.4 删除镜像

1. 使用标签删除镜像

使用 `docker rmi` 命令可以删除镜像，命令格式为 `docker rmi IMAGE [IMAGE...]`，其中 IMAGE 可以为标签或 ID。

例如，要删除掉 myubuntu:latest 镜像，可以使用如下命令：

```
$ docker rmi myubuntu:latest
Untagged: myubuntu:latest
```

读者可能会担心，本地的 ubuntu:latest 镜像是否会受到此命令的影响。无需担心，当同一个镜像拥有多个标签的时候，`docker rmi` 命令只是删除了该镜像多个标签中的指定标签而已，并不影响镜像文件。因此上述操作相当于只是删除了镜像 `2fa927b5cdd3` 的一个标签而已。

保险起见，再次查看本地的镜像，发现 ubuntu:latest 镜像（准确地说是 2fa927b5cdd3 镜像）仍然存在：

```
$ docker images
REPOSITORY       TAG         IMAGE ID            CREATED             SIZE
ubuntu           16.04       2fa927b5cdd3        2 weeks ago         122 MB
ubuntu           latest      2fa927b5cdd3        2 weeks ago         122 MB
ubuntu           14.04       8f1bd21bd25c        2 weeks ago         188 MB
```

但当镜像只剩下一个标签的时候就要小心了，此时再使用 `docker rmi` 命令会彻底删除镜像。

例如删除标签为 `ubuntu:14.04` 的镜像，由于该镜像没有额外的标签指向它，执行 `docker rmi` 命令，可以看出它会删除这个镜像文件的所有层。

```
$ docker rmi ubuntu:14.04
```

```
Untagged: ubuntu:14.04
Deleted: sha256:8f1bd21bd25c3fb1d4b00b7936a73a0664f932e11406c48a0ef19d82fd0b7342
Deleted: sha256:8ea3b9ba4dd9d448d1ca3ca7afa8989d033532c11050f5e129d267be8de9c1b4
Deleted: sha256:7db5fb90eb6ffb6b5418f76dde5f685601fad200a8f4698432ebf8ba80757576
Deleted: sha256:19a7e879151723856fb640449481c65c55fc9e186405dd74ae6919f88eccce75
Deleted: sha256:c357a3f74f16f61c2cc78dbb0ae1ff8c8f4fa79be9388db38a87c7d8010b2fe4
Deleted: sha256:a7e1c363defb1f80633f3688e945754fc4c8f1543f07114befb5e0175d569f4c
```

2. 使用镜像 ID 删除镜像

当使用 `docker rmi` 命令，并且后面跟上镜像的 ID （也可以是能进行区分的部分 ID 串前缀）时，会先尝试删除所有指向该镜像的标签，然后删除该镜像文件本身。

注意，当有该镜像创建的容器存在时，镜像文件默认是无法被删除的，例如：

先利用 `ubuntu:14.04` 镜像创建一个简单的容器来输出一段话。

```
$ docker run ubuntu:14.04 echo 'hello! I am here!'
hello! I am here!
```

使用 docker ps -a 命令可以看到本机上存在的所有容器。

```
$ docker ps -a
CONTAINER ID   IMAGE   COMMAND   CREATED    STATUS  PORTS   NAMES
a21c0840213e        ubuntu:14.04        "echo 'hello! I am he"   About a minute ago   Exited (0)
About a minute ago     romantic_euler
```

可以看到，后台存在一个退出状态的容器，是刚基于 ubuntu:14.04 镜像创建的。

试图删除该镜像，Docker 会提示有容器正在运行，无法删除。

```
$ docker rmi ubuntu:14.04
Error response from daemon: conflict: unable to remove repository reference "ubuntu:14.04" (must
force) - container a21c0840213e is using its referenced image 8f1bd21bd25c
```

如果要想强行删除镜像，可以使用 `-f` 参数。

```
$ docker rmi -f ubuntu:14.04
Untagged: ubuntu:14.04
Deleted: sha256:8f1bd21bd25c3fb1d4b00b7936a73a0664f932e11406c48a0ef19d82fd0b7342
```

注意，通常并不推荐使用 `-f` 参数来强制删除一个存在容器依赖的镜像。

正确的做法是，先删除依赖该镜像的所有容器，再来删除镜像。

首先删除容器 a21c0840213e。

```
$ docker rm a21c0840213e
```

再来使用 ID 来删除镜像，此时会正常打印出删除的各层信息。

```
$ docker rmi 8f1bd21bd25c
Untagged: ubuntu:14.04
Deleted: sha256:8f1bd21bd25c3fb1d4b00b7936a73a0664f932e11406c48a0ef19d82fd0b7342
Deleted: sha256:8ea3b9ba4dd9d448d1ca3ca7afa8989d033532c11050f5e129d267be8de9c1b4
Deleted: sha256:7db5fb90eb6ffb6b5418f76dde5f685601fad200a8f4698432ebf8ba80757576
Deleted: sha256:19a7e879151723856fb640449481c65c55fc9e186405dd74ae6919f88eccce75
Deleted: sha256:c357a3f74f16f61c2cc78dbb0ae1ff8c8f4fa79be9388db38a87c7d8010b2fe4
```

```
Deleted: sha256:a7e1c363defb1f80633f3688e945754fc4c8f1543f07114befb5e0175d569f4c
```

3.5 创建镜像

创建镜像的方法主要有三种：基于已有镜像的容器创建、基于本地模板导入、基于 Dockerfile 创建。

本节将重点介绍前两种方法。最后一种基于 Dockerfile 创建的方法将在后续章节专门予以详细介绍。

1. 基于已有镜像的容器创建

该方法主要是使用 `docker commit` 命令。

命令格式为 docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]，主要选项包括。

- `-a, --author=""`：作者信息；
- `-c, --change=[]`：提交的时候执行 Dockerfile 指令，包括 `CMD|ENTRYPOINT|ENV|EXPOSE|LABEL|ONBUILD|USER|VOLUME|WORKDIR` 等；
- `-m, --message=""`：提交消息；
- `-p, --pause=true`：提交时暂停容器运行。

下面将演示如何使用该命令创建一个新镜像。

首先，启动一个镜像，并在其中进行修改操作，例如创建一个 test 文件，之后退出。

```
$ docker run -it ubuntu:14.04 /bin/bash
root@a925cb40b3f0:/# touch test
root@a925cb40b3f0:/# exit
```

记住容器的 ID 为 `a925cb40b3f0`。

此时该容器跟原 `ubuntu:14.04` 镜像相比，已经发生了改变，可以使用 `docker commit` 命令来提交为一个新的镜像。提交时可以使用 ID 或名称来指定容器。

```
$ docker commit -m "Added a new file" -a "Docker Newbee" a925cb40b3f0 test:0.1
9e9c814023bcffc3e67e892a235afe61b02f66a947d2747f724bd317dda02f27
```

顺利的话，会返回新创建的镜像的 ID 信息，例如

`9e9c814023bcffc3e67e892a235afe61b02f66a947d2747f724bd317dda02f27`。

此时查看本地镜像列表，会发现新创建的镜像已经存在了。

```
$ docker images
REPOSITORY        TAG     IMAGE ID    CREATED    VIRTUAL SIZE
test              0.1     9e9c814023bc        4 seconds ago       188 MB
```

2. 基于本地模板导入

用户也可以直接从一个操作系统模板文件导入一个镜像，主要使用 `docker import` 命令。

主要命令格式为 docker import [OPTIONS] file|URL|-[REPOSITORY[:TAG]]

要直接导入一个镜像，可以使用 OpenVZ 提供的模板来创建，或者用其他已导出的镜像模板来创建。

OPENVZ 模板的下载地址为http://openvz.org/Download/templates/precreated。

例如，下载了 ubuntu-14.04 的模板压缩包，之后使用以下命令导入：

```
$ cat ubuntu-14.04-x86_64-minimal.tar.gz | docker import - ubuntu:14.04
```

然后查看新导入的镜像，已经在本地存在了。

```
$ docker images
REPOSITORY          TAG         IMAGE ID         CREATED             VIRTUAL SIZE
ubuntu              14.04       05ac7c0b9383     17 seconds ago      215.5 MB
```

3.6 存出和载入镜像

用户可以使用 `docker save` 和 `docker load` 命令来存出和载入镜像。

1. 存出镜像

如果要导出镜像到本地文件，可以使用 `docker save` 命令。

例如，导出本地的 ubuntu:14.04 镜像为文件 ubuntu_14.04.tar，如下所示：

```
$ docker images
REPOSITORY          TAG         IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu              14.04       8f1bd21bd25c    2 weeks ago     188 MB
...
$ docker save -o ubuntu_14.04.tar ubuntu:14.04
```

之后，用户就可以通过复制 ubuntu_14.04.tar 文件将该镜像分享给他人。

2. 载入镜像

可以使用 `docker load` 将导出的 tar 文件再导入到本地镜像库，例如从文件 ubuntu_14.04.tar 导入镜像到本地镜像列表，如下所示：

```
$ docker load --input ubuntu_14.04.tar
```

或：

```
$ docker load < ubuntu_14.04.tar
```

这将导入镜像及其相关的元数据信息（包括标签等）。导入成功后，可以使用 `docker images` 命令进行查看。

3.7 上传镜像

可以使用 `docker push` 命令上传镜像到仓库，默认上传到 Docker Hub 官方仓库（需要登录）。命令格式为：

```
docker push NAME[:TAG] | [REGISTRY_HOST[:REGISTRY_PORT]/]NAME[:TAG]
```

用户在 Docker Hub 网站注册后可以上传自制的镜像。例如用户 `user` 上传本地的 test:latest 镜像，可以先添加新的标签 `user/test:latest`，然后用 `docker push` 命令上传镜像：

```
$ docker tag test:latest user/test:latest
$ docker push user/test:latest
The push refers to a repository [docker.io/user/test]
Sending image list
```

```
Please login prior to push:
Username:
Password:
Email:
```

第一次上传时，会提示输入登录信息或进行注册。

3.8 本章小结

本章具体介绍了围绕 Docker 镜像的一系列重要命令操作，包括获取、查看、搜索、删除、创建、存出和载入、上传等。

镜像是使用 Docker 的前提，也是最基本的资源。所以，在平时的 Docker 使用中，要注意积累自己定制的镜像文件，并将自己创建的高质量镜像，分享到社区中。

在后续章节，笔者将会介绍更多对镜像进行操作的场景。

第4章 操作 Docker 容器

容器是 Docker 的另一个核心概念。

简单来说，容器是镜像的一个运行实例。所不同的是，镜像是静态的只读文件，而容器带有运行时需要的可写文件层。

如果认为虚拟机是模拟运行的一整套操作系统（包括内核，应用运行态环境和其他系统环境）和跑在上面的应用。那么 Docker 容器就是独立运行的一个（或一组）应用，以及它们必需的运行环境。

本章将具体介绍围绕容器的重要操作，包括创建一个容器、启动容器、终止一个容器、进入容器内执行操作、删除容器和通过导入导出容器来实现容器迁移等。

4.1 创建容器

从现在开始，忘掉“臃肿”的虚拟机吧。对容器的操作就跟直接操作应用一样简单和快速。

Docker 容器实在太轻量级了，用户可以随时创建或删除容器。

1. 新建容器

可以使用 `docker create` 命令新建一个容器，例如

```
$ docker create -it ubuntu:latest
af8f4f922dafee22c8fe6cd2ae11d16e25087d61f1b1fa55b36e94db7ef45178
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
af8f4f922daf ubuntu:latest "/bin/bash" 17 seconds ago Created silly_euler
```

使用 `docker create` 命令新建的容器处于停止状态，可以使用 `docker start` 命令来启动它。

`create` 和 后续的 `run` 命令支持的命令选项都十分复杂，主要包括如下几大类：与容器运行模式相关、与容器和环境配置相关、与容器资源限制和安全保护相关等。参见表 4-1、表 4-2、表 4-3。

表 4-1 `create` 命令与容器运行模式相关的选项

选项	说明
-a, --attach=[]	是否绑定到标准输入、输出和错误
-d, --detach=true|false	是否在后台运行容器，默认为否
--detach-keys=""	从 attach 模式退出的快捷键
--entrypoint=""	镜像存在入口命令时，覆盖为新的命令
--expose=[]	指定容器会暴露出来的端口或端口范围
--group-add=[]	运行容器的用户组
-i, --interactive=true|false	保持标准输入打开，默认为 false
--ipc=""	容器 IPC 命名空间，可以为其他容器或主机
--isolation="default"	容器使用的隔离机制
--log-driver="json-file"	指定容器的日志驱动类型，可以为 json-file、syslog、journald、gelf、fluentd、awslogs、splunk、etwlogs、gcplogs 或 none
--log-opt=[]	传递给日志驱动的选项
--net="bridge"	指定容器网络模式，包括 bridge、none、其他容器内网络、host 的网络或某个现有网络等
--net-alias=[]	容器在网络中的别名

-P, --publish-all=true|false	通过 NAT 机制将容器标记暴露的端口自动映射到本地主机的临时端口
-p, --publish=[]	指定如何映射到本地主机端口，例如 -p 11234-12234:1234-2234
--pid=host	容器的 PID 命名空间
--userns=""	启用 userns-remap 时配置用户命名空间的模式
--uts=host	容器的 UTS 命名空间
--restart="no"	容器的重启策略,包括 no、on-failure[:max-retry]、always、unless-stopped 等
--rm=true|false	容器退出后是否自动删除，不能跟-d 同时使用
-t, --tty=true|false	是否分配一个伪终端，默认为 false
--tmpfs=[]	挂载临时文件系统到容器
-v|--volume[=[[HOST-DIR:]CONTAINER-DIR[:OPTIONS]]]	挂载主机上的文件卷到容器内
--volume-driver=""	挂载文件卷的驱动类型
--volumes-from=[]	从其他容器挂载卷
-w, --workdir=""	容器内的默认工作目录

表 4-2 create 命令与容器环境和配置相关的选项

选项	说明
--add-host=[]	在容器内添加一个主机名到 IP 地址的映射关系（通过 /etc/hosts 文件）
--device=[]	映射物理机上的设备到容器内
--dns-search=[]	DNS 搜索域
--dns-opt=[]	自定义的 DNS 选项
--dns=[]	自定义的 DNS 服务器
-e, --env=[]	指定容器内环境变量
--env-file=[]	从文件中读取环境变量到容器内
-h, --hostname=""	指定容器内的主机名
--ip=""	指定容器的 IPv4 地址
--ip6=""	指定容器的 IPv6 地址
--link=[<name or id>:alias]	链接到其他容器
--mac-address=""	指定容器的 Mac 地址
--name=""	指定容器的别名

表 4-3 create 命令与容器资源限制和安全保护相关的选项

选项	说明
--blkio-weight=10~1000	容器读写块设备的 IO 性能权重，默认为 0
--blkio-weight-device=[DEVICE_NAME:WEIGHT]	指定各个块设备的 IO 性能权重
--cpu-shares=0	允许容器使用 CPU 资源的相对权重,默认一个容器能用满一个核的 CPU
--cap-add=[]	增加容器的 Linux 指定安全能力
--cap-drop=[]	移除容器的 Linux 指定安全能力
--cgroup-parent=""	容器 cgroups 限制的创建路径
--cidfile=""	指定容器的进程 ID 号写到文件
--cpu-period=0	限制容器在 CFS 调度器下的 CPU 占用时间片
--cpuset-cpus=""	限制容器能使用哪些 CPU 核心
--cpuset-mems=""	NUMA 架构下使用哪些核心的内存
--cpu-quota=0	限制容器在 CFS 调度器下的 CPU 配额
--device-read-bps=[]	挂载设备的读吞吐率（以 bps 为单位）限制
--device-write-bps=[]	挂载设备的写吞吐率（以 bps 为单位）限制
--device-read-iops=[]	挂载设备的读速率（以每秒 io 次数为单位）限制

--device-write-iops=[]	挂载设备的写速率（以每秒 io 次数为单位）限制
--kernel-memory=""	限制容器使用内核内存大小，单位可以是 b、k、m 或 g
-m, --memory=""	限制容器内应用使用的内存，单位可以是 b、k、m 或 g
--memory-reservation=""	当系统中内存过低时，容器会被强制限制内存到给定值，默认情况下等于内存限制值
--memory-swap="LIMIT"	限制容器使用内存和交换区的总大小
--oom-kill-disable=true|false	内存耗尽（Out-Of-Memory）时候是否杀死容器
--oom-score-adj=""	调整容器的内存耗尽参数
--pids-limit=""	限制容器的 pid 个数
--privileged=true|false	是否给以容器以高权限，这意味着容器内应用将不受到权限下限制，一般不推荐
--read-only=true|false	是否让容器内文件系统只读
--security-opt=[]	指定一些安全参数，包括权限、安全能力、apparmor 等
--stop-signal=SIGTERM	指定停止容器的系统信号
--shm-size=""	/dev/shm 的大小
--sig-proxy=true|false	是否代理收到的信号给应用，默认为 true，不能代理 SIGCHLD、SIGSTOP 和 SIGKILL 信号
--memory-swappiness="0~100"	调整容器的内存交换区参数
-u, --user=""	指定在容器内执行命令的用户信息
--ulimit=[]	通过 ulimit 来限制最大文件数、最大进程数等

其他比较重要的选项还包括：

- -l, --label=[]：以键值对方式指定容器的标签信息；
- --label-file=[]：从文件中读取标签信息。

2. 启动容器

使用 docker start 命令来启动一个已经创建的容器。例如启动刚创建的 ubuntu 容器。

```
$ docker start af
af
```

此时，通过 docker ps 命令，可以查看到一个运行中的容器。

```
$ docker ps
CONTAINER ID    IMAGE COMMAND        CREATED      STATUS           PORTS NAMES
af8f4f922daf  ubuntu:latest "/bin/bash"   2 minutes ago  Up 7 seconds silly_euler
```

3. 新建并启动容器

除了创建容器后通过 start 命令来启动，也可以直接新建并启动容器。

所需要的命令主要为 docker run，等价于先执行 docker create 命令，再执行 docker start 命令。

例如，下面的命令输出一个 “Hello World”，之后容器自动终止。

```
$ docker run ubuntu  /bin/echo 'Hello world'
Hello world
```

这跟在本地直接执行 /bin/echo 'hello world' 几乎感觉不出任何区别。

当利用 `docker run` 来创建并启动容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载；
- 利用镜像创建一个容器，并启动该容器；
- 分配一个文件系统给容器，并在只读的镜像层外面挂载一层可读写层；
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去；
- 从网桥的地址池配置一个 IP 地址给容器；
- 执行用户指定的应用程序；
- 执行完毕后容器被自动终止。

下面的命令则启动一个 bash 终端，允许用户进行交互。

```
$ docker run -it ubuntu:14.04 /bin/bash
root@af8bae53bdd3:/#
```

其中，`-t` 选项让 Docker 分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上，`-i` 则让容器的标准输入保持打开。更多的命令选项可以通过 `man docker-run` 命令来查看。

在交互模式下，用户可以通过所创建的终端来输入命令，例如

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@af8bae53bdd3:/# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   11 ?        00:00:00 ps
```

在容器内用 ps 命令查看进程，可以看到，只运行了 bash 应用，并没有运行其他无关的进程。

用户可以按 Ctrl+d 或输入 `exit` 命令来退出容器。

```
root@af8bae53bdd3:/# exit
exit
```

对于所创建的 bash 容器，当使用 `exit` 命令退出之后，容器就自动处于退出（Exited）状态了。这是因为对于 Docker 容器来说，当运行的应用退出后，容器也就没有继续运行的必要了。

某些时候，执行 `docker run` 时候因为命令无法正常执行容器会出错直接退出，此时可以查看退出的错误代码。

默认情况下，常见错误代码包括：

- 125：Docker daemon 执行出错，例如指定了不支持的 Docker 命令参数；
- 126：所指定命令无法执行，例如权限出错；
- 127：容器内命令无法找到。

命令执行后出错，会默认返回命令的退出错误码。

4. 守护态运行

更多的时候，需要让 Docker 容器在后台以守护态（Daemonized）形式运行。此时，可以通过添加 `-d` 参数来实现。

例如下面的命令会在后台运行容器。

```
$ docker run -d ubuntu  /bin/sh -c "while true; do echo hello world; sleep 1; done"
ce554267d7a4c34eefc92c5517051dc37b918b588736d0823e4c846596b04d83
```

容器启动后会返回一个唯一的 id，也可以通过 `docker ps` 命令来查看容器信息。

```
$ docker ps
CONTAINER ID   IMAGE           COMMAND    CREATED   STATUS PORTS   NAMES
ce554267d7a4  ubuntu:latest  "/bin/sh -c 'while t  About a minute ago  Up About a minute
determined_pik
```

此时，要获取容器的输出信息，可以通过 `docker logs` 命令。

```
$ docker logs ce5
hello world
hello world
hello world
. . .
```

4.2 终止容器

可以使用 `docker stop` 来终止一个运行中的容器。

该命令的格式为 docker stop [-t|--time[=10]] [CONTAINER...]。

首先向容器发送 SIGTERM 信号，等待一段超时时间后（默认为 10 秒），再发送 SIGKILL 信号来终止容器。

```
$ docker stop ce5
ce5
```

> 注意：`docker kill` 命令会直接发送 SIGKILL 信号来强行终止容器。

此外，当 Docker 容器中指定的应用终结时，容器也会自动终止。例如对于上一章节中只启动了一个终端的容器，用户通过 `exit` 命令或 `Ctrl+d` 来退出终端时，所创建的容器立刻终止，处于 stopped 状态。

可以用 `docker ps -qa` 命令看到所有容器的 ID。例如

```
$ docker ps -qa
ce554267d7a4
d58050081fe3
e812617b41f6
```

处于终止状态的容器，可以通过 `docker start` 命令来重新启动。

```
$ docker start ce5
ce5
$ docker ps
```

```
CONTAINER ID   IMAGE     COMMAND       CREATED     STATUS        PORTS               NAMES
ce554267d7a4   ubuntu:latest   "/bin/sh -c 'while t   4 minutes ago   Up 5 seconds
determined_pike
```

此外，`docker restart` 命令会将一个运行态的容器先终止，然后再重新启动它。

```
$ docker restart ce5
ce5
$ docker ps
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
ce554267d7a4   ubuntu:latest   "/bin/sh -c 'while t   5 minutes ago   Up 14 seconds
determined_pike
```

4.3 进入容器

在使用 `-d` 参数时，容器启动后会进入后台，用户无法看到容器中的信息，也无法进行操作。

这个时候如果需要进入容器进行操作，有多种方法，包括使用官方的 `attach` 或 `exec` 命令，以及第三方的 `nsenter` 工具等。

1. attach 命令

`attach` 是 Docker 自带的命令，命令格式为

docker attach [--detach-keys[=[]]] [--no-stdin] [--sig-proxy[=true]] CONTAINER

支持三个主要选项：

- `--detach-keys[=[]]`：指定退出 attach 模式的快捷键序列，默认是 `CTRL-p CTRL-q`；
- `--no-stdin=true|false`：是否关闭标准输入，默认是保持打开；
- `--sig-proxy=true|false`：是否代理收到的系统信号给应用进程，默认为 true。

下面示例如何使用该命令。

```
$ docker run -itd ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ docker ps
CONTAINER ID   IMAGE    COMMAND   CREATED   STATUS    PORTS   NAMES
243c32535da7   ubuntu:latest  "/bin/bash"    18 seconds ago   Up 17 seconds   nostalgic_hypatia
$ docker attach nostalgic_hypatia
root@243c32535da7:/#
```

但是使用 `attach` 命令有时候并不方便。当多个窗口同时 attach 到同一个容器的时候，所有窗口都会同步显示。当某个窗口因命令阻塞时,其他窗口也无法执行操作了。

2. exec 命令

Docker 从 1.3.0 版本起，提供了一个更加方便的工具 `exec` 命令，可以直接在容器内直接执行任意命令。

该命令的基本格式为

```
docker exec [-d|--detach] [--detach-keys[=[]]] [-i|--interactive] [--privileged] [-t|--tty]
[-u|--user[=USER]] CONTAINER COMMAND [ARG...]。
```

比较重要的参数有：

- -i, --interactive=true|false :打开 标准输入接受用户输入命令, 默认为 false ;
- --privileged=true|false：是否给执行命令以高权限, 默认为 false ;
- -t, --tty=true|false：分配伪终端, 默认为 false ;
- -u, --user=""：执行命令的用户名或 ID。

例如进入到刚创建的容器中，并启动一个 bash。

```
$ docker exec -it 243c32535da7  /bin/bash
root@243c32535da7:/#
```

可以看到，一个 bash 终端打开了，在不影响容器内其他应用的前提下，用户可以很容易与容器进行交互。

注意通过指定 -it 参数来保持标准输入打开，并且分配一个伪终端。通过 exec 命令对容器执行操作是最为推荐的方式。

3.nsenter 工具

nsenter 工具在 util-linux 软件包版本 2.23+ 中包含。如果系统中 util-linux 包没有该命令，可以按照下面的方法从源码安装。

```
$ cd /tmp; curl https://www.kernel.org/pub/linux/utils/util-linux/v2.24/util-linux-2.24.tar.gz |
tar -zxf-; cd util-linux-2.24;
$ ./configure --without-ncurses
$ make nsenter && cp nsenter /usr/local/bin
```

为了使用 nsenter 连接到容器，还需要找到容器的进程的 PID，可以通过下面的命令获取。

```
PID=$(docker inspect --format "{{ .State.Pid }}" <container>)
```

通过这个 PID，就可以连接到这个容器：

```
$ nsenter --target $PID --mount --uts --ipc --net --pid
```

下面给出一个完整的例子，通过 nsenter 命令进入容器。

```
$ docker run -idt ubuntu
243c32535da7d142fb0e6df616a3c3ada0b8ab417937c853a9e1c251f499f550
$ docker ps
CONTAINER ID   IMAGE   COMMAND   CREATED  STATUS   PORTS  NAMES
243c32535da7  ubuntu:latest   "/bin/bash"  18 seconds ago  Up 17 seconds nostalgic_hypatia
$ PID=$(docker-pid 243c32535da7)
10981
$ nsenter --target 10981 --mount --uts --ipc --net --pid
root@243c32535da7:/#
```

进一步可在容器中查看容器中的用户和进程信息。

```
root@ce554267d7a4:/# w
 11:07:36 up  3:14,  0 users,  load average: 0.00, 0.02, 0.05
```

```
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root@ce554267d7a4:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root       1     0  0 10:56 ?   00:00:00 /bin/sh -c while true; do echo hello world; sleep 1; done
root       699     0  0 11:07 ?        00:00:00 /bin/bash
root       716     1  0 11:07 ?        00:00:00 sleep 1
root       717   699  0 11:07 ?        00:00:00 ps -ef
```

4.4 删除容器

可以使用 docker rm 命令来删除处于终止或退出状态的容器，命令格式为 docker rm [-f|--force] [-l|--link] [-v|--volumes] CONTAINER [CONTAINER...]。

主要支持的选项包括：

- -f, --force=false：是否强行终止并删除一个运行中的容器；
- -l, --link=false：删除容器的连接，但保留容器；
- -v, --volumes=false：删除容器挂载的的数据卷。

例如，查看处于终止状态的容器，并删除。

```
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND     CREATED    STATUS      PORTS               NAMES
ce554267d7a4    ubuntu:latest   "/bin/sh -c 'while t   3 minutes ago  Exited (-1) 13 seconds ago
determined_pike
d58050081fe3    ubuntu:latest   "/bin/bash"   About an hour ago   Exited (0) About an hour ago
berserk_brattain
e812617b41f6        ubuntu:latest       "echo 'hello! I am h   2 hours ago         Exited (0) 3
minutes ago


$ docker rm ce554267d7a4
ce554267d7a4
```

默认情况下，docker rm 命令只能删除已经处于终止或退出状态的容器，并不能删除还处于运行状态的容器。

如果要直接删除一个运行中的容器，可以添加 -f 参数。Docker 会先发送 SIGKILL 信号给容器，终止其中的应用，之后强行删除。

```
$ docker run -d ubuntu:14.04  /bin/sh -c "while true; do echo hello world; sleep 1; done"
2aed76caf8292c7da6d24c3c7f3a81a135af942ed1707a79f85955217d4dd594
$ docker rm 2ae
Error response from daemon: You cannot remove a running container. Stop the container before
attempting removal or use -f
2016/07/03 09:02:24 Error: failed to remove one or more containers
$ docker rm -f 2ae
2ae
```

4.5 导入和导出容器

某些时候，需要将容器从一个系统迁移到另外一个系统，此时可以使用 Docker 的导入和导出功能。这也是 Docker 自身提供的一个重要特性。

1. 导出容器

导出容器，是指导出一个已经创建的容器到一个文件，不管此时这个容器是否处于运行状态，可以使用 `docker export` 命令，该命令格式为 `docker export [-o|--output[=""]] CONTAINER`。

其中，可以通过 `-o` 选项来指定导出的 tar 文件名，也可以直接通过重定向来实现。

首先，查看所有的容器，如下所示。

```
$ docker ps -a
CONTAINER ID   IMAGE  COMMAND    CREATED    STATUS     PORTS    NAMES
ce554267d7a4   ubuntu:latest     "/bin/sh -c 'while t"   3 minutes ago       Exited (-1) 13
seconds ago                          determined_pike
d58050081fe3        ubuntu:latest       "/bin/bash"            About an hour ago   Exited (0)
About an hour ago                        berserk_brattain
e812617b41f6        ubuntu:latest       "echo 'hello! I am h"   2 hours ago         Exited (0) 3
minutes ago                             silly_leakey
```

分别导出 `ce554267d7a4` 容器和 `e812617b41f6` 容器到文件 test_for_run.tar 文件和 test_for_stop.tar 文件。

```
$ docker export -o test_for_run.tar ce5
$ ls
test_for_run.tar
$ docker export e81 >test_for_stop.tar
$ ls
test_for_run.tar test_for_stop.tar
```

之后，可将导出的 tar 文件传输到其他机器上，然后再通过导入命令导入到系统中，实现容器的迁移。

2. 导入容器

导出的文件又可以使用 `docker import` 命令导入变成镜像，该命令格式为

```
docker import [-c|--change[=[]]] [-m|--message[=MESSAGE]] file|URL|-[REPOSITORY[:TAG]]
```

用户可以通过`-c, --change=[]`选项在导入的同时执行对容器进行修改的 Dockerfile 指令（可参考后续相关章节）。

下面将导出的 `test_for_run.tar` 文件导入到系统中：

```
$ docker import test_for_run.tar - test/ubuntu:v1.0
$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED        VIRTUAL SIZE
test/ubuntu         v1.0       9d37a6082e97   About a minute ago   171.3 MB
```

之前镜像章节中笔者曾介绍过使用 `docker load` 命令来导入一个镜像文件，与 `docker import` 命令十分类似。

实际上，既可以使用 `docker load` 命令来导入镜像存储文件到本地镜像库，也可以使

用`docker import`命令来导入一个容器快照到本地镜像库。

这两者的区别在于容器快照文件将丢弃所有的历史记录和元数据信息(即仅保存容器当时的快照状态)，而镜像存储文件将保存完整记录，体积也要大。此外，从容器快照文件导入时可以重新指定标签等元数据信息。

4.6 本章小结

容器是直接提供应用服务的组件，也是 Docker 实现快速的启停和高效服务性能的基础。

通过本章内容的介绍和示例，相信读者很快能掌握对容器整个生命周期进行管理的各项操作命令。

在生产环境中，因为容器自身的轻量级特性，笔者推荐使用容器时在一组容器前引入 HA（High Availability，高可靠性）机制。例如使用 HAProxy 工具来代理容器访问，这样在容器出现故障时候，可以快速切换到功能正常的容器。此外，建议通过指定合适的容器重启策略，来自动重启退出的容器。

第5章 访问 Docker 仓库

仓库（Repository）是集中存放镜像的地方，分公共仓库和私有仓库。

一个容易与之混淆的概念是注册服务器（Registry）。实际上注册服务器是存放仓库的具体服务器，一个注册服务器上可以有多个仓库，而每个仓库下面可以有多个镜像。从这方面来说，可将仓库看做一个具体的项目或目录。例如对于仓库地址 `private-docker.com/ubuntu` 来说，`private-docker.com` 是注册服务器地址，`ubuntu` 是仓库名。

本章将介绍如何使用 Docker Hub 官方仓库进行登录、下载等基本操作，以及使用国内社区提供的仓库下载镜像介绍；最后还将介绍创建和使用私有仓库的基本操作。

5.1 Docker Hub 公共镜像市场

目前 Docker 官方维护了一个公共镜像仓库 https://hub.docker.com，其中已经包括超过 15,000 的镜像。大部分镜像需求，都可以通过在 Docker Hub 中直接下载镜像来实现。

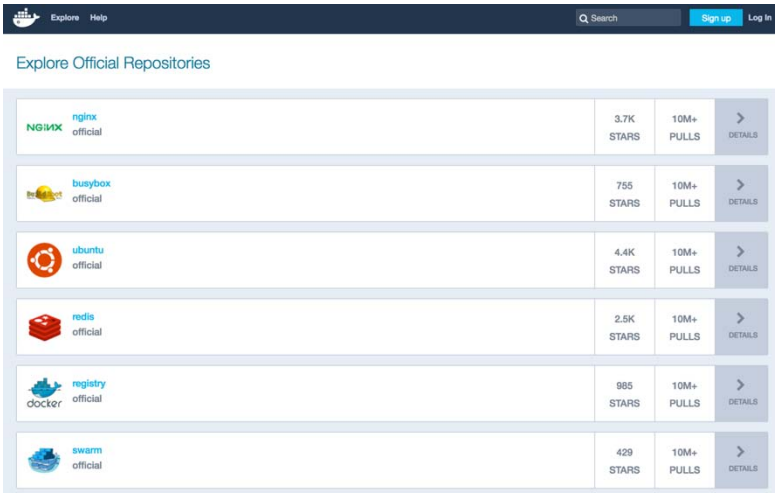


图 5-1 DockerHub - 最大的公共镜像仓库

1. 登录

可以通过命令行执行 `docker login` 命令来输入用户名、密码和邮箱来完成注册和登录。 注册成功后，本地用户目录的 `.dockercfg` 中将保存用户的认证信息。

登录成功的用户可以上传个人制造的镜像。

2. 基本操作

用户无需登录即可通过 `docker search` 命令来查找官方仓库中的镜像，并利用 `docker pull` 命令来将它下载到本地。

在搜寻镜像的章节，已经具体介绍了如何使用 `docker pull` 命令。 例如以 centos 为关键词进行搜索：

```
$ docker search centos
NAME                     DESCRIPTION          STARS     OFFICIAL   AUTOMATED
centos                   The official build of CentOS.          2507       [OK]
ansible/centos7-ansible  Ansible on Centos7                       82       [OK]
```

```
jdeathe/centos-ssh    CentOS-6 6.8 x86_64 / CentOS-7 7.2.1511 x8...   27   [OK]
nimmis/java-centos    This is docker images of CentOS 7 with dif...   13   [OK]
million12/centos-supervisor  Base CentOS-7 with supervisord launcher, h...12  [OK]
...
```

根据是否为官方提供，可将这些镜像资源分为两类。 一种是类似 centos 这样的基础镜像，称为基础或根镜像。这些镜像是由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。

还有一种类型，比如 `ansible/centos7-ansible` 镜像，它是由 Docker 用户 ansible 创建并维护的，带有用户名称为前缀，表明是某用户下的某仓库。可以通过用户名称前缀 `user_name/`镜像名 来指定使用某个用户提供的镜像。

另外，在查找的时候通过 `-s N` 参数可以指定仅显示评价为 `N` 星以上的镜像。

下载官方 centos 镜像到本地，如下所示。

```
$ docker pull centos
Pulling repository centos
0b443ba03958: Download complete
539c0211cd76: Download complete
511136ea3c5a: Download complete
7064731afe90: Download complete
```

用户也可以在登录后通过 `docker push` 命令来将本地镜像推送到 Docker Hub。

提示：Ansible 是知名自动化部署配置管理工具。

3. 自动创建

自动创建（Automated Builds）功能对于需要经常升级镜像内程序来说，十分方便。有时候，用户创建了镜像，安装了某个软件，如果软件发布新版本则需要手动更新镜像。

而自动创建允许用户通过 Docker Hub 指定跟踪一个目标网站（目前支持 GitHub 或 BitBucket）上的项目，一旦项目发生新的提交，则自动执行创建。

要配置自动创建，包括如下的步骤：

1） 创建并登陆 Docker Hub，以及目标网站；* 在目标网站中连接帐户到 Docker Hub；

2） 在 Docker Hub 中配置一个 自动创建；

3） 选取一个目标网站中的项目（需要含 Dockerfile）和分支；

4） 指定 Dockerfile 的位置，并提交创建。

之后，可以在 Docker Hub 的“自动创建”页面中跟踪每次创建的状态。

5.2 时速云镜像市场

国内不少云服务商都提供了 Docker 镜像市场，下面以时速云为例，介绍如何使用这些市场。

图 5-2 时速云镜像市场

1. 查看镜像

访问 https://hub.tenxcloud.com，即可看到已存在的仓库和存储的镜像，包括 Ubuntu、Java、Mongo、MySQL、Nginx 等热门仓库和镜像。时速云官方仓库中的镜像会保持跟 DockerHub 中官方镜像的同步。

以 MongoDB 仓库为例，其中包括了 2.6、3.0 和 3.2 等镜像。

2. 下载镜像

下载镜像也是使用 docker pull 命令，但是要在镜像名称前添加注册服务器的具体地址。格式为 index.tenxcloud.com/<namespace>/<repository>:<tag>。

例如，要下载 Docker 官方仓库中的 node:latest 镜像，可以使用如下命令：

```
$ docker pull index.tenxcloud.com/docker_library/node:latest
```

正常情况下，镜像下载会比直接从 DockerHub 下载快得多。

通过 `docker images` 命令来查看下载到本地的镜像。

```
$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZEindex.tenxcloud.com/docker_library/node latest e79fe5711c94
4 weeks ago 660.7 MB
```

下载后，可以更新镜像的标签，与官方标签保持一致，方便使用。

```
$ docker tag index.tenxcloud.com/docker_library/node:latest node:latest
```

另外，阿里云等服务商也已经提供了 Docker 镜像的下载服务，用户可以根据服务质量自行选择。

除了使用这些公共镜像服务外，还可以搭建本地的私有仓库服务器，将在下一节介绍。

5.3 搭建本地私有仓库

1. 使用 registry 镜像创建私有仓库

安装 Docker 后，可以通过官方提供的 registry 镜像来简单搭建一套本地私有仓库环境。

```
$ docker run -d -p 5000:5000 registry
```

这将自动下载并启动一个 registry 容器，创建本地的私有仓库服务。

默认情况下，会将仓库创建在容器的 `/tmp/registry` 目录下。可以通过 `-v` 参数来将镜像文件存放在本地的指定路径。

例如下面的例子将上传的镜像放到 `/opt/data/registry` 目录。

```
$ docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/registry registry
```

此时，在本地将启动一个私有仓库服务，监听端口为 5000。

2. 管理私有仓库

首先在本书环境的笔记本上（Linux Mint）搭建私有仓库，查看其地址为 `10.0.2.2:5000`。 然后在虚拟机系统（Ubuntu 14.04）里测试上传和下载镜像。

在 Ubuntu 14.04 系统查看已有的镜像。

```
$ docker images
REPOSITORY              TAG          IMAGE ID            CREATED     VIRTUAL SIZE
ubuntu                  14.04        ba5877dc9bec        6 days ago   199.3 MB
```

使用 docker tag 命令将这个镜像标记为 10.0.2.2:5000/test（格式为 docker tag IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]）。

```
$ docker tag ubuntu:14.04 10.0.2.2:5000/test
$ docker images
REPOSITORY           TAG       IMAGE ID            CREATED      VIRTUAL SIZE
ubuntu               14.04     ba5877dc9bec        6 days ago   199.3 MB
10.0.2.2:5000/test   latest    ba5877dc9bec        6 days ago   199.3 MB
```

使用 `docker push` 上传标记的镜像。

```
$ docker push 10.0.2.2:5000/test
The push refers to a repository [10.0.2.2:5000/test] (len: 1)
Sending image list
Pushing repository 10.0.2.2:5000/test (1 tags)
Image 511136ea3c5a already pushed, skipping
Image 9bad880da3d2 already pushed, skipping
Image 25f11f5fb0cb already pushed, skipping
Image ebc34468f71d already pushed, skipping
Image 2318d26665ef already pushed, skipping
Image ba5877dc9bec already pushed, skipping
Pushing tag for rev [ba5877dc9bec] on {http://10.0.2.2:5000/v1/repositories/test/tags/latest}
```

用 curl 查看仓库 `10.0.2.2:5000` 中的镜像。

```
$ curl http://10.0.2.2:5000/v1/search
{"num_results": 1, "query": "", "results": [{"description": "", "name": "library/test"}]}
```

在结果中可以看到{"description": "", "name":"library/test"}，表明镜像已经被成功上传了。

现在可以到任意一台能访问到 `10.0.2.2` 地址的机器去下载这个镜像了。

比较新的 Docker 版本对安全性要求较高，会要求仓库支持 SSL/TLS 证书。对于内部使用的私有仓库，可以自行配置证书或关闭对仓库的安全性检查。

首先，修改 Docker daemon 的启动参数，添加如下参数，表示信任这个私有仓库，不

进行安全证书检查。

```
DOCKER_OPTS="--insecure-registry 10.0.2.2:5000"
```

之后重启 Docker 服务，并从私有仓库中下载镜像到本地。

```
$ sudo service docker restart
$ docker pull 10.0.2.2:5000/test
Pulling repository 10.0.2.2:5000/test
ba5877dc9bec: Download complete
511136ea3c5a: Download complete
9bad880da3d2: Download complete
25f11f5fb0cb: Download complete
ebc34468f71d: Download complete
2318d26665ef: Download complete
$ docker images
REPOSITORY              TAG         IMAGE ID       CREATED      VIRTUAL SIZE
10.0.2.2:5000/test      latest      ba5877dc9bec   6 days ago   199.3 MB
```

下载后，还可以添加一个更通用的标签 `ubuntu:14.04`。

```
$ docker tag 10.0.2.2:5000/test ubuntu:14.04
```

注意：如果要使用安全证书，用户也可以从较知名的 CA 服务商（如 verisign）申请公开的 SSL/TLS 证书，或者使用 openssl 等软件来自行生成。

5.4 本章小结

仓库概念的引入，为 Docker 镜像文件的分发和管理提供了便捷的途径。

本章介绍的 Docker Hub 和时速云镜像市场两个公共仓库服务，可以方便个人用户进行镜像的下载和使用等操作。

在企业的生产环境中，则往往需要使用私有仓库来维护内部镜像。在后续部分中，将介绍私有仓库的更多配置选项。